공학박사학위논문

# 효율적인 텐서 분해 방법을 통한 실세계 텐서 마이닝

## Mining Real World Tensors via Efficient Tensor Decomposition Methods

2023년 2월

서울대학교 대학원

컴퓨터공학부

장준기

# 효율적인 텐서 분해 방법을 통한 실세계 텐서 마이닝

## Mining Real World Tensors via Efficient Tensor Decomposition Methods

지도교수 강 유

이 논문을 공학박사 학위논문으로 제출함

2023년 1월


서울대학교 대학원

컴퓨터공학부

장준기


장준기의 박사 학위논문을 인준함

2022년 12월

| | | |
|---|---|---|
| 위 원 장 | 문 봉 기 | (인) |
| 부 위 원 장 | 강 유 | (인) |
| 위 원 | 박 근 수 | (인) |
| 위 원 | 황 승 원 | (인) |
| 위 원 | 신 기 정 | (인) |

# Mining Real World Tensors via Efficient Tensor Decomposition Methods

Jun-Gi Jang
Department of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

## Abstract

Many real-world data can be represented as tensors including vectors (1-order tensor), matrices (2-order tensor), and higher-order tensors. For example, there are stock data, healthcare data, video data, sensor data, and movie rating data represented as tensors. Tensor decomposition has been widely used in applications to analyze real-world tensors. Since knowledge is inherent in real-world tensors, it is crucial to devise effective Tensor decomposition methods. However, existing Tensor decomposition-based methods require high computational costs and space costs. Therefore, it is very challenging to discover hidden information in large-scale tensors without efficient tensor decomposition methods.

In this thesis, I overcome the limitations of previous tensor analysis methods based on tensor decomposition. Since existing tensor decomposition methods require heavy computations involved with large-scale input tensors, it is crucial to avoid

the computations to achieve high efficiency. My proposed methods achieve high efficiency by approximating an input tensor and exploiting the approximation result. I devise highly efficient methods for regular and irregular tensors by exploiting the characteristics of real-world tensors and carefully determining the order of computations. Furthermore, I develop a fast and memory-efficient tensor decomposition-based method that analyzes diverse time ranges.

Extensive experiments show that the proposed methods achieve higher efficiency than existing methods while having comparable errors. The proposed methods decompose regular and irregular tensors up to 38.4× and 6× faster than existing methods, respectively. In addition, the proposed method analyzes various time range queries up to 171.9× faster than existing methods. Consequently, the proposed methods allow us to explore meaningful knowledge from various real-world tensors efficiently.

**Keywords :** Tensor Mining, Efficiency, Tensor Decomposition, Tucker Decomposition, PARAFAC2 Decomposition, Real-world Regular Tensors, Real-world Irregular Tensors

**Student Number :** 2017-23528

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Tensors are natural representations of many real-world data such as sensor data, stock data, video data, and electronic health data. Vectors and matrices are first-order tensors and second-order tensors, respectively, and higher-order tensors denote third or higher tensors. For example, we construct stock data as a third-order tensor of the following form: (time, feature, stock). The tensor can be viewed as the collection of stock matrices whose rows and columns correspond to time and features (e.g., the opening price, the closing price, the trade volume, etc.), respectively. In addition to stock data, we represent sensor data as a third-order tensor of (time, location, sensor) form.

Tensor mining has attracted much attention from various research and industrial fields since it allows us to find key information that provides deeper insights into the complex phenomena inherent in real-world tensors, and support making effective decisions in the age of information overload. To achieve it, many researchers have been exploiting tensor decomposition for various applications such as missing value prediction, anomaly detection, recommendation, and so on. Tensor decomposition decomposes an input tensor into latent factor matrices which contain information hidden in the tensor. Table 1.1 summarizes the details of real-world applications with tensor decomposition.

Although the importance of tensor mining has been emerging, it is challenging since the size of real-world tensors explosively increases, and tensor analysis requires

Table 1.1: **Tensor decomposition-based real-world applications**

| Application & Its References | Brief Description |
|---|---|
| Feature analysis [1, 2, 3, 4, 5] | Discover knowledge from latent features obtained by tensor decomposition |
| Missing value prediction [6, 7, 8, 9] | Predict missing values in a tensor |
| Recommendation [10, 11, 12, 13, 14] | Recommend new items that users would prefer |
| Anomaly detection [15, 16, 17, 18] | Detect anomalous patterns in a tensor |
| Model compression [19, 20, 21, 22, 23, 24] | Compress deep learning models |

expensive tensor operations. In addition, it is necessary to analyze various forms of data in various settings as environments where data is generated become more complex. However, previous tensor decomposition-based methods fail to provide high efficiency, and thus they are very limited to be used in real-world settings. Therefore, it is crucial to devise efficient tensor decomposition methods to extract knowledge from real-world tensors of various forms in real-world settings.

In this thesis, I concentrated on developing highly efficient tensor decomposition methods for large real-world tensors in real-world settings. Two main topics to be addressed are to 1) devise efficient tensor decomposition methods for regular and irregular tensors in real-world settings and 2) analyze a temporal tensor for diverse time ranges. We started with improving the performance of tensor decomposition in terms of time and space costs. Given a large tensor in real-world settings, we aim at the following research questions:

- **[Chapter 3]** how can we efficiently discover hidden concepts and patterns of large tensors in static and online streaming settings?
- **[Chapter 4]** how can we efficiently analyze large irregular tensors in a static

Table 1.2: **An overview of works studied in this thesis.**

| Tensor Form \ Setting | Static Setting | Online Setting |
|---|---|---|
| **Regular Tensor** | D-Tucker [Chapter 3] | D-TuckerO [Chapter 3]<br>Zoom-Tucker [Chapter 5] |
| **Irregular Tensor** | DPar2 [Chapter 4] | Online irregular tensor decomposition (Future work [Chapter 6]) |

setting?

Note that an irregular tensor is a collection of matrices where the number of columns is the same and the number of rows is different from each other. In an online streaming setting, the size of the time dimension of a tensor grows over time.

The second topic is to analyze diverse time ranges when a large tensor with the time dimension is given. Assume that users are interested in exploring knowledge from various time ranges. For example, given a temporal tensor including matrices collected from Jan. 1, 2008, to May 6, 2020, a user can be interested in hidden information inherent in the range (from Jan. 1, 2020, to April 30, 2020). This thesis answers the following question to provide an opportunity to efficiently explore knowledge from various perspectives:

- **[Chapter 5]** Given a temporal tensor and a time range, how can we efficiently analyze the tensor in the given time range?

The main challenge to be addressed is to improve the efficiency of tensor decomposition for real-world tensors. ALS (Alternating Least Square) has been widely used for obtaining factor matrices of tensor decomposition. Until convergence, it iteratively updates a target factor matrix while fixing all the other factor matrices. However, computations with an input tensor at each iteration require high costs due to the large size of the tensor. In addition, performing tensor decomposition several

times is also burdensome. Therefore, avoiding repeated computations involved with a given tensor is required to achieve the goal.

To address the challenge, I approximate an input tensor before iterations, and then obtain factor matrices by exploiting the approximated results. The methods to be developed generate approximated results which are obtained with high efficiency and are much smaller than the input tensor. Then, they obtain factor matrices by exploiting the approximated results in iterative computations. Note that the input tensor is not used in the iterations. Although they do not generate the same results as the methods that use the input tensor and sacrifice a little accuracy, I empirically show that our proposed methods have better trade-offs between efficiency and accuracy than existing tensor decomposition-based methods. The accuracy loss is not significant compared to the efficiency improvement.

## 1.1 Contributions

Table 1.2 describes an overview of works studied in this thesis, and I summarize our contributions as follows:

- **Efficient Tucker Decomposition in Large-scale Regular Tensors (Chapter 3).** I propose D-Tucker and D-TuckerO, fast and memory-efficient Tucker decomposition methods for regular tensors in static and online streaming settings, respectively. Both methods achieve high efficiency by approximating a given tensor, and then efficiently computing Tucker decomposition only using the approximated results. D-Tucker achieves up to 38.4× faster and requires up to 17.2× less space than existing Tucker decomposition methods while having comparable accuracy. In addition, D-TuckerO successfully works in the online streaming setting by efficiently dealing with new incoming tensors, and out-

performs its competitors by up to 6.1× faster than them.

- **Efficient PARAFAC2 Decomposition in Large-scale Irregular Tensors (Chapter 4).** I devise DPar2, a fast and scalable PARAFAC2 decomposition method for irregular tensors. DPar2 achieves high efficiency by reducing numerical computations and intermediate data, and maximizing multi-core parallelism. DPar2 is the fastest PARAFAC2 decomposition method by giving up to 6.0× faster than existing PARAFAC2 decomposition methods. It is also scalable with respect to input and output sizes.

- **Efficient Tucker Decomposition for Diverse Time Range Queries (Chapter 5).** I propose Zoom-Tucker to analyze a temporal tensor for diverse time ranges. Zoom-Tucker effectively approximates a given tensor before time range queries are given, and answers diverse time range queries quickly and memory-efficiently by exploiting the approximated results and fruitful mathematical techniques. Note that Zoom-Tucker works in an online setting since the preprocessing phase is extensible for new incoming tensors by performing Tucker decomposition of them. Given a time range query, Zoom-Tucker is up to 171.9× times faster and requires up to 230× less space than previous methods. Zoom-Tucker provides an opportunity to explore diverse time ranges in large-scale temporal dense tensors.

## 1.2 Overall Impact

My research outcomes leave a lasting impact in academic and industrial fields as the followings:

- **Efficiency Improvement.** My proposed methods significantly improve the efficiency of tensor decomposition methods for regular and irregular tensors in terms of speed, space, and scalability.
- **Effective Analysis.** My proposed methods allow researchers to effectively analyze tensors in real-world applications such as anomaly detection and feature analysis.

Most of the algorithms introduced in this thesis are open to the public for reproducibility and my research achieved the following results:

- Our work [25] was selected as the best research paper in KDD 2021 and awarded the Qualcomm Innovation Fellowship Korea.
- Our work [1] was selected as the best paper (honorable mention) in ICDE 2022.
- Our research works [1, 25, 26, 27] included in this thesis were supported by the Yulchon AI Star Award, Naver Ph.D. Fellowship, and Future Gauss Lecture Program.

## 1.3 Thesis Organization

The rest of this thesis proposal is organized as follows. The background on tensor notations, tensor decomposition, and related works are provided in Chapter 2. In Chapter 3, we propose the fast and memory-efficient tucker decomposition methods, D-Tucker and D-TuckerO, for large real-world tensors in static and streaming settings. In Chapter 4, we propose DPar2, a fast and scalable PARAFAC2 decomposition method for real-world irregular tensors. In Chapter 5, we propose a novel method Zoom-Tucker that efficiently analyzes various time ranges using Tucker decomposition. Finally, I present future works in Chapter 6, and conclude in Chapter 7.

# Chapter 2

# Background

This section describes notations, definitions, and related works for tensors and tensor decomposition.

## 2.1 Tensor

### 2.1.1 Tensor Notation

Each 'dimension' of a tensor (i.e., a multi-dimensional array) is denoted by *mode*. 'dimensionality' of a mode denotes the length of it. An $N$-order tensor is represented as a boldface Euler script capital (e.g. $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$) letter, and matrices are denoted by boldface capitals (e.g. $\mathbf{A}$). A mode-$n$ fiber is a vector having fixed indices except for the $n$-th index in a tensor. A sliced matrix is a matrix having fixed all indices except for two indices in a tensor. An irregular tensor is a 3-order tensor $\boldsymbol{\mathcal{X}}$ whose $k$-frontal slice $\boldsymbol{\mathcal{X}}(:,:,k)$ is $\mathbf{X}_k \in \mathbb{R}^{I_k \times J}$. We denote irregular tensors by $\{\mathbf{X}_k\}_{k=1}^{K}$ instead of $\boldsymbol{\mathcal{X}}$ where $K$ is the number of $k$-frontal slices of the tensor.

### 2.1.2 Tensor Operation

We use the following tensor operations in this thesis: Frobenius norm, matricization, $n$-mode product, Kronecker product, and slicing.

**Frobenius Norm.** The Frobenius norm of $\boldsymbol{\mathcal{X}}$ $(\in \mathbb{R}^{I_1 \times \cdots \times I_N})$ is denoted by $\|\boldsymbol{\mathcal{X}}\|_F$

and defined as follows:

$$\|\mathcal{X}\|_F = \sqrt{\sum_{\forall(i_1,\ldots,i_N)\in\mathcal{X}} \mathcal{X}^2_{(i_1,\ldots,i_N)}}.$$

**Matricization.** Mode-$n$ matricization converts a given tensor into a matrix form along $n$-th mode. We denote the mode-$n$ matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ as $\mathbf{X}_{(n)}$. Each element $(i_1, \ldots, i_N)$ of $\mathcal{X}$ is mapped to an element $(i_n, j)$ of $\mathbf{X}_{(n)}$ such that

$$j = 1 + \sum_{\substack{k=1 \\ k\neq n}}^{N} \left( (i_k - 1) \prod_{\substack{m=1 \\ m\neq n}}^{k-1} I_m \right),$$

where all indices start from 1.

***n*-mode product.** The $n$-mode product $\mathcal{X} \times_n \mathbf{A}$ of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a matrix $\mathbf{A} \in \mathbb{R}^{J_n \times I_n}$ has the size of $I_1 \times \cdots I_{n-1} \times J_n \times I_{n+1} \cdots \times I_N$, and defined by

$$(\mathcal{X} \times_n \mathbf{A})_{i_1 \ldots i_{n-1} j_n i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} a_{j_n i_n}$$

where $a_{j_n i_n}$ is the $(j_n, i_n)$-th entry of $\mathbf{A}$. The result of $n$-mode product of a tensor $\mathcal{X}$ with a matrix $\mathbf{A}$ is identical to that of the following three operations: 1) performing mode-$n$ matricization $\mathbf{X}_{(n)}$, 2) computing $\mathbf{Y}_{(n)} = \mathbf{A}\mathbf{X}_{(n)}$, and 3) reshaping the result $\mathbf{Y}_{(n)}$ as a tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots I_{n-1} \times J_n \times I_{n+1} \cdots \times I_N}$.

**Kronecker product.** Kronecker product of a matrix $\mathbf{A} \in \mathbb{R}^{p \times q}$ with a matrix $\mathbf{B} \in \mathbb{R}^{r \times s}$ produces the output $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ of the size $pr \times qs$. Each element of the output is defined as follows:

$$\mathbf{C}_{r(t-1)+u, s(v-1)+w} = a_{t,v} \times b_{u,w} \tag{2.1}$$

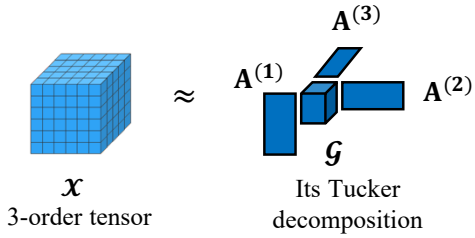



Figure 2.1: Example of Tucker decomposition. Given a tensor $\mathcal{X}$, Tucker decomposition decomposes it into the factor matrices $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, $\mathbf{A}^{(3)}$, and core tensor $\mathcal{G}$. Note that $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ are column orthogonal matrices.

where $a_{t,v}$ is $(t,v)$-th element of the matrix $\mathbf{A}$ and $b_{u,w}$ is $(u,w)$-th element of the matrix $\mathbf{B}$.

**Khatri-Rao product.** The Khatri-Rao product between two matrices $\mathbf{X} \in \mathbb{R}^{p \times q}$ and $\mathbf{Y} \in \mathbb{R}^{r \times q}$ is denoted by $(\mathbf{X} \odot \mathbf{Y}) \in \mathbb{R}^{pr \times q}$. The Khatri-Rao product performs the Kronecker product column by column: $(\mathbf{X} \odot \mathbf{Y}) = [\mathbf{X}(:,1); \cdots ; \mathbf{X}(:,q)] \odot [\mathbf{Y}(:,1); \cdots ; \mathbf{Y}(:,q)] = [\mathbf{X}(:,1) \otimes \mathbf{Y}(:,1); \cdots ; \mathbf{X}(:,q) \otimes \mathbf{Y}(:,q)]$, where ; denotes the horizontal concatenation.

**Slicing a tensor.** Slicing an $N$-order tensor $\mathcal{X}$ $(\in \mathbb{R}^{I_1 \times \ldots \times I_N})$ along modes not in $\{m,n\}$ decomposes $\mathcal{X}$ into $L$ sliced matrices of size $I_m \times I_n$, where $L = I_1 \times \ldots \times I_{m-1} \times I_{m+1} \times \ldots \times I_{n-1} \times I_{n+1} \times \ldots \times I_N$. For example, consider a 3-order tensor $\mathcal{X}$ $(\in \mathbb{R}^{I_1 \times I_2 \times K_3})$ in Figure 3.1. Slicing $\mathcal{X}$ along mode 3 leads to $K_3$ sliced matrices of size $I_1 \times I_2$.

## 2.2 Tensor Decomposition

This section describes two representative tensor decomposition methods used in this thesis: Tucker decomposition and PARAFAC2 decomposition.

### 2.2.1 Tucker Decomposition

**Algorithm 1:** Tucker-ALS (HOOI) [28]

**Input:** tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ and core tensor dimensionality $J_1, \ldots, J_N$
**Output:** core tensor $\mathcal{G} \in \mathbb{R}^{J_1, \ldots, J_N}$ and factor matrices $\mathbf{A}^{(i)}$ $(i = 1, \ldots, N)$

```
initialize: factor matrices A^(i) (i = 1,...,N)
repeat
    for i = 1,...,N do
        Y ← X ×_1 A^(1)T ··· ×_{i-1} A^(i-1)T ×_{i+1} A^(i+1)T ··· ×_N A^(N)T
        A^(i) ← J_i leading left singular vectors of Y_(i)
    end for
until the maximum iteration is reached, or the error ceases to decrease;
G ← X ×_1 A^(1)T ×_2 A^(2)T ··· ×_N A^(N)T
```

**Definition 2.1** (Tucker Decomposition). *Given an N-order tensor* $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$*, Tucker decomposition decomposes* $\mathcal{X}$ *into the core tensor* $\mathcal{G} \in \mathbb{R}^{J_1 \times \ldots \times J_N}$ *and factor matrices* $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ *for* $n = 1 \ldots N$. □

Note that $\mathbf{A}^{(n)}$ is a column orthogonal matrix, i.e. $\mathbf{A}^{(n)T}\ \mathbf{A}^{(n)} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix, and core tensor $\mathcal{G}$ is small and dense. Figure 2.1 shows an example of Tucker decomposition. The objective function of Tucker decomposition is given as follows.

$$\min_{\mathcal{G}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}} ||\mathcal{X} - \mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)}|| \tag{2.2}$$

where we represent the given tensor $\mathcal{X}$ using the core tensor $\mathcal{G}$ and factor matrices $\mathbf{A}^{(n)}$:

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)} \tag{2.3}$$

In addition, we re-express Equation (2.3) with matricization and Kronecker product as follows:

$$\mathbf{X}_{(n)} \approx \mathbf{A}^{(n)} \mathbf{G}_{(n)} (\otimes_{k \neq n}^{N} \mathbf{A}^{(k)T}) \tag{2.4}$$

where $(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)T})$ indicates Kronecker product of $\mathbf{A}^{(k)T}$ for $k = N, N-1, ..., n+1, n-1, ..., 2, 1$.

**Computing the Tucker decomposition.** A common approach to minimize Equation (2.2) is ALS (Alternating Least Square). ALS approach iteratively updates the factor matrix of a mode while fixing all factor matrices of other modes. Algorithm 1 describes Tucker decomposition based on ALS approach, which is called higher-order orthogonal iteration (HOOI). A bottleneck of ALS approach for a dense tensor is to compute Equation (2.5) (line 4 in Algorithm 1) which requires $O(\prod_{m=1}^{N} I_m)$ space and $O(J_1 \times \prod_{m=1}^{N} I_m)$ computational time even to compute the first $n$-mode product between an input tensor $\mathcal{X}$ and the factor matrix $\mathbf{A}^{(1)}$.

$$\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{i-1} \mathbf{A}^{(i-1)T} \times_{i+1} \mathbf{A}^{(i+1)T} \cdots \times_N \mathbf{A}^{(N)T} \Leftrightarrow \mathbf{Y}_{(i)} \leftarrow \mathbf{X}_{(i)} \left(\otimes_{k\neq i}^{N}\mathbf{A}^{(k)}\right) \tag{2.5}$$

Note that Equation (2.5) re-expresses line 4 of Algorithm 1 with mode-$i$ matricization and Kronecker product (see details in [29]). Moreover, the computational time grows as the number of iterations increases.

### 2.2.2 PARAFAC2 decomposition

PARAFAC2 decomposition proposed by Harshman [30] successfully deals with irregular tensors. The definition of PARAFAC2 decomposition is as follows:

**Definition 2.2** (PARAFAC2 Decomposition)**.** *Given a target rank $R$ and a 3-order tensor $\{\mathbf{X}_k\}_{k=1}^{K}$ whose $k$-frontal slice is $\mathbf{X}_k \in \mathbb{R}^{I_k \times J}$ for $k = 1, ..., K$, PARAFAC2 decomposition approximates each $k$-th frontal slice $\mathbf{X}_k$ by $\mathbf{U}_k\mathbf{S}_k\mathbf{V}^T$. $\mathbf{U}_k$ is a matrix of the size $I_k \times R$, $\mathbf{S}_k$ is a diagonal matrix of the size $R \times R$, and $\mathbf{V}$ is a matrix of the size $J \times R$*

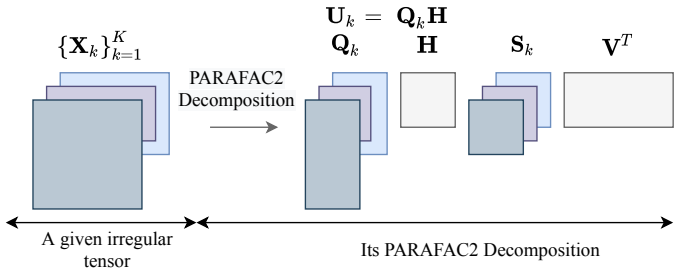



Figure 2.2: Example of PARAFAC2 decomposition. Given an irregular tensor $\{\mathbf{X}_k\}_{k=1}^{K}$, PARAFAC2 decomposition decomposes it into the factor matrices $\mathbf{H}$, $\mathbf{V}$, $\mathbf{Q}_k$, and $\mathbf{S}_k$ for $k = 1, \ldots, K$. Note that $\mathbf{Q}_k\mathbf{H}$ is equal to $\mathbf{U}_k$.

*which are common for all the slices.* □

The objective function of PARAFAC2 decomposition [30] is given as follows.

$$\min_{\{\mathbf{U}_k\},\{\mathbf{S}_k\},\mathbf{V}} \sum_{k=1}^{K} ||\mathbf{X}_k - \mathbf{U}_k\mathbf{S}_k\mathbf{V}^T||_F^2 \tag{2.6}$$

For uniqueness, Harshman [30] imposed the constraint (i.e., $\mathbf{U}_k^T\mathbf{U} = \Phi$ for all $k$), and replace $\mathbf{U}_k^T$ with $\mathbf{Q}_k\mathbf{H}$ where $\mathbf{Q}_k$ is a column orthogonal matrix and $\mathbf{H}$ is a common matrix for all the slices. Then, Equation (2.6) is reformulated with $\mathbf{Q}_k\mathbf{H}$:

$$\min_{\{\mathbf{Q}_k\},\{\mathbf{S}_k\},\mathbf{H},\mathbf{V}} \sum_{k=1}^{K} ||\mathbf{X}_k - \mathbf{Q}_k\mathbf{H}\mathbf{S}_k\mathbf{V}^T||_F^2 \tag{2.7}$$

Figure 2.2 shows an example of PARAFAC2 decomposition for a given irregular tensor. A common approach to solve the above problem is ALS (Alternating Least Square) which iteratively updates a target factor matrix while fixing all factor matrices except for the target. Algorithm 2 describes PARAFAC2-ALS. First, we update each $\mathbf{Q}_k$ while fixing $\mathbf{H}$, $\mathbf{V}$, $\mathbf{S}_k$ for $k = 1, \ldots, K$ (lines 4 and 5). By computing SVD of $\mathbf{X}_k\mathbf{V}\mathbf{S}_k\mathbf{H}^T$ as $\mathbf{Z}_k'\mathbf{\Sigma}_k'\mathbf{P}_k'^T$, we update $\mathbf{Q}_k$ as $\mathbf{Z}_k'\mathbf{P}_k'^T$, which minimizes Equation (2.8) over $\mathbf{Q}_k$ [2, 31, 32]. After updating $\mathbf{Q}_k$, the remaining factor matrices $\mathbf{H}$, $\mathbf{V}$, $\mathbf{S}_k$ is updated by minimizing

**Algorithm 2:** PARAFAC2-ALS [31]

**Input:** $\mathbf{X}_k \in \mathbb{R}^{I_k \times J}$ for $k = 1, ..., K$
**Output:** $\mathbf{U}_k \in \mathbb{R}^{I_k \times R}$, $\mathbf{S}_k \in \mathbb{R}^{R \times R}$ for $k = 1, ..., K$, and $\mathbf{V} \in \mathbb{R}^{J \times R}$.
**Parameters:** target rank $R$

1: initialize matrices $\mathbf{H} \in \mathbb{R}^{R \times R}$, $\mathbf{V}$, and $\mathbf{S}_k$ for $k = 1, ..., K$
2: **repeat**
3:  **for** $k = 1, ..., K$ **do**
4:  compute $\mathbf{Z}'_k \mathbf{\Sigma}'_k \mathbf{P}'^T_k \leftarrow \mathbf{X}_k \mathbf{V} \mathbf{S}_k \mathbf{H}^T$ by performing truncated SVD at rank $R$
5:  $\mathbf{Q}_k \leftarrow \mathbf{Z}'_k \mathbf{P}'^T_k$
6:  **end for**
7:  **for** $k = 1, ..., K$ **do**
8:  $\mathbf{Y}_k \leftarrow \mathbf{Q}_k^T \mathbf{X}_k$
9:  **end for**
10:  construct a tensor $\mathcal{Y} \in \mathbb{R}^{R \times J \times K}$ from slices $\mathbf{Y}_k \in \mathbb{R}^{R \times J}$ for $k = 1, ..., K$
/* running a single iteration of CP-ALS on $\mathcal{Y}$ */
11:  $\mathbf{H} \leftarrow \mathbf{Y}_{(1)} (\mathbf{W} \odot \mathbf{V}) (\mathbf{W}^T \mathbf{W} * \mathbf{V}^T \mathbf{V})^\dagger$
12:  $\mathbf{V} \leftarrow \mathbf{Y}_{(2)} (\mathbf{W} \odot \mathbf{H}) (\mathbf{W}^T \mathbf{W} * \mathbf{H}^T \mathbf{H})^\dagger$
13:  $\mathbf{W} \leftarrow \mathbf{Y}_{(3)} (\mathbf{V} \odot \mathbf{H}) (\mathbf{V}^T \mathbf{V} * \mathbf{H}^T \mathbf{H})^\dagger$
14:  **for** $k = 1, ..., K$ **do**
15:  $\mathbf{S}_k \leftarrow diag(\mathbf{W}(k, :))$
16:  **end for**
17: **until** the maximum iteration is reached, or the error ceases to decrease;
18: **for** $k = 1, ..., K$ **do**
19:  $\mathbf{U}_k \leftarrow \mathbf{Q}_k \mathbf{H}$
20: **end for**

the following objective function:

$$\min_{\{\mathbf{S}_k\}, \mathbf{H}, \mathbf{V}} \sum_{k=1}^{K} ||\mathbf{Q}_k^T \mathbf{X}_k - \mathbf{H} \mathbf{S}_k \mathbf{V}^T||_F^2 \tag{2.8}$$

Minimizing this function is to update $\mathbf{H}$, $\mathbf{V}$, $\mathbf{S}_k$ using CP decomposition of a tensor $\mathcal{Y} \in \mathbb{R}^{R \times J \times K}$ whose $k$-th frontal slice is $\mathbf{Q}_k^T \mathbf{X}_k$ (lines 8 and 10). We run a single iteration of CP decomposition for updating them [31] (lines 11 to 16). $\mathbf{Q}_k$, $\mathbf{H}$, $\mathbf{S}_k$, and $\mathbf{V}$ are alternatively updated until convergence.

Iterative computations with an irregular dense tensor require high computational costs and large intermediate data. RD-ALS [33] reduces the costs by prepro-

Table 2.1: **An overview of related works corresponding to our proposed works in this thesis.**

| | **Regular Tensor** & **Static Setting** | **Irregular Tensor** & **Static Setting** | **Regular Tensor** & **Online Setting** |
|---|---|---|---|
| **Related Works** | Chapter 2.3.1 | Chapter 2.3.2 | Chapter 2.3.3<br>Chapter 2.3.4 |
| **Our Works** | Chapter 3 (D-Tucker) | Chapter 4 (DPar2) | Chapter 3 (D-TuckerO)<br>Chapter 5 (Zoom-Tucker) |

cessing a given tensor and performing PARAFAC2 decomposition using the preprocessed result, but the improvement of RD-ALS is limited. Also, recent works successfully have dealt with sparse irregular tensors by exploiting sparsity. However, the efficiency of their models depends on the *sparsity* patterns of a given irregular tensor, and thus there is little improvement on irregular *dense* tensors. Specifically, computations with large dense slices $\mathbf{X}_k$ for each iteration are burdensome as the number of iterations increases. We focus on improving the efficiency and scalability in irregular dense tensors.

## 2.3 Related Works

I describe related works for tensor decomposition methods working in real-world settings. I deal with two tensor forms (i.e., regular tensors and irregular tensors) and two real-world settings (i.e., static and online settings). Table 2.1 shows an overview of related works corresponding to our proposed works studied in this thesis. In Chapter 2.3.1, I present static tensor decomposition methods for regular tensors, which is related to D-Tucker (Chapter 3). In Chapter 2.3.2, I present static tensor decomposition methods for irregular tensors, which is related to DPar2 (Chapter 4). In Chapter 2.3.3, I present online tensor decomposition methods for regular tensors, which is related to D-TuckerO (Chapter 3). In Chapter 2.3.4, I describe methods for answering time

range queries on regular tensors, which is related to Zoom-Tucker (Chapter 5).

### 2.3.1 Tensor Decomposition on Regular Tensors

De Lathauwer et al. [28] proposed Tucker-ALS (Algorithm 1) which alternately updates factor matrices and obtains core tensor. A few Tucker decomposition methods slightly reduce the computational time using efficient matrix operations [34, 35]. Che et al. [36] applied randomized algorithms for Tucker decomposition. The main challenges of Tucker decomposition are heavy computational time and large memory requirements due to large-scale dense tensors. To overcome the challenges, MACH [37] is designed to reduce the computational time and the memory requirement by sampling input tensors. Also, Malik et al. [38] used a sketch of input tensors to overcome the challenges. However, there is still plenty of room for improvement in terms of efficiency. Several works [5, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57] optimize tensor decomposition in parallel systems and distributed systems.

Tucker decomposition has been widely used for several applications including dimensionality reduction [19, 58], recommendation [7, 59, 11], clustering [60, 61], image tag refinement [62, 63], phenotype discovery [2, 3, 4], and many others [64, 65, 1]. Oh et al. [7] analyzed movie rating data and discovered relations between movie and time attributes by considering only observable entries. Kim et al. [19] used Tucker decomposition for compressing a deep convolutional neural network. Jang et al. [25] proposed a Tucker decomposition-based method to efficiently analyze a given time range.

### 2.3.2 PARAFAC2 Decomposition on Irregular Tensors

Cheng and Haardt [33] proposed RD-ALS which preprocesses a given tensor and performs PARAFAC2 decomposition using the preprocessed result. However, RD-ALS requires high computational costs to preprocess a given tensor. Also, RD-ALS is less efficient in updating factor matrices since it computes reconstruction errors for the convergence criterion at each iteration. Recent works [2, 3, 66] attempted to analyze irregular sparse tensors. SPARTan [2] is a scalable PARAFAC2-ALS method for large electronic health records (EHR) data. COPA [3] improves the performance of PARAFAC2 decomposition by applying various constraints (e.g., smoothness). REPAIR [66] strengthens the robustness of PARAFAC2 decomposition by applying low-rank regularization. We do not compare DPar2 with COPA and REPAIR since they concentrate on imposing practical constraints to handle irregular sparse tensors, especially EHR data. However, we do compare DPar2 with SPARTan which the efficiency of COPA and REPAIR is based on. TASTE [67] is a joint PARAFAC2 decomposition method for large temporal and static tensors. Although the above methods are efficient in PARAFAC2 decomposition for irregular tensors, they concentrate only on irregular sparse tensors, especially EHR data. LogPar [68], a logistic PARAFAC2 decomposition method, analyzes temporal binary data represented as an irregular binary tensor. SPADE [69] efficiently deals with irregular tensors in a streaming setting. TedPar [4] improves the performance of PARAFAC2 decomposition by explicitly modeling the temporal dependency. Although the above methods effectively deal with irregular sparse tensors, especially EHR data, none of them focus on devising an efficient PARAFAC2 decomposition method on irregular dense tensors.

### 2.3.3 Online Streaming Tensor Decomposition

Many works [70, 71, 72, 73, 74, 75, 76] have developed CP decomposition methods in an online streaming setting. RLST (Recursive Least Squares Tracking) and SDT (Simultaneous Diagonalization Tracking) [70] are adaptive PARAFAC decomposition methods of a third-order tensor in an online streaming setting. Zhou et al. [71] developed onlineCP, a streaming CP decomposition method, while Zhou et al. [75] extend onlineCP for sparse tensors. Gujral et al. [74] and Smith et al. [73] proposed streaming CP decomposition methods in parallel systems. Lee et al. [77] proposed a robust tensor factorization that leverages two temporal characteristics: graduality and seasonality. Ahn et al. [76, 78] proposed tensor factorization methods by capturing temporal locality patterns. Son et al. [79] proposed a n online tensor factorization method by capturing sudden change in data. The main difference between the above methods and our proposed method is that they focus on developing online versions of CP decomposition while D-TuckerO is based on Tucker decomposition.

Sun et al. [17] incrementally analyzed temporal tensors over time: they proposed two algorithms, DTA (dynamic tensor analysis) and STA (streaming tensor analysis). However, the above methods update factor matrices and core tensor by naively using a new incoming tensor without compression, thereby efficiency improvement is limited when a new incoming tensor is sufficiently large. In addition, tucker-ts and tucker-ttmts [38] can be applied to online streaming settings. However, they fail to avoid increasing the running time over time. Sun et al. [80] proposed a streaming Tucker decomposition method with a sketching technique in distributed systems, assuming that time slices are stored in several machines. MAST [72] deals with the scenario in which a given tensor grows in multiple modes while D-TuckerO runs on the setting where only one mode increases.

### 2.3.4 Answering Time Range Queries on Regular Tensors

Zoom-SVD [81] deals with the time range query problem, but it is suitable only for multiple time series data represented as a matrix. Although there is no existing method that precisely addresses the time range query problem for tensors, there are several methods [26, 37, 38] that can be adapted to solve the problem. They perform a preprocessing phase by exploiting a sampling technique [37] or randomized SVD [26] before the query phase, and then obtain Tucker results using the preprocessed results in the query phase. However, they do not satisfy the desired properties for the solution: fast running time, low space cost, and accuracy.

# Chapter 3

# Efficient Static and Streaming Tensor Decomposition in Regular Tensors

## 3.1 Motivation

How can we efficiently discover hidden concepts and patterns of large dense tensors? Many real-world data including video, music, and air quality, can be represented as dense tensors. Tucker decomposition is a fundamental tool for factorizing a given tensor into factor matrices and a core tensor to find hidden concepts and latent patterns. Tucker decomposition has spurred much interest with various applications including dimensionality reduction [19, 58], recommendation [7, 59], and clustering [60, 61].

Alternating Least Square (ALS) is the most widely used method for Tucker decomposition. Existing ALS-based methods, however, fail to satisfy all the desired properties for dense tensor decompositions: fast running time, low memory requirement, and high accuracy. Tucker-ALS which updates factor matrices iteratively is slow when the number of iterations is large. Moreover, Tucker-ALS has a memory problem to obtain final factor matrices and a core tensor since it directly handles large dense tensors in order to update the factor matrices and the core tensor at each iteration. A few static Tucker decomposition methods reduce the computational cost using efficient matrix operations [34, 35] or applying randomized algorithms [36, 82]. In addition, other Tucker decomposition methods [37, 38] reduce the computational time and the memory requirement by approximating large dense tensors. However,

none of them provide both fast running time and accuracy. The major challenges to deal with large dense tensors are 1) how to efficiently approximate a large dense tensor with low error, and 2) how to update factor matrices by using approximated results.

In this work, we propose D-Tucker and D-TuckerO, efficient Tucker decomposition methods on large dense tensors. D-Tucker and D-TuckerO run in static and online streaming settings, respectively. The main ideas of D-Tucker are as follows: 1) slice an input tensor into matrices and compress each matrix by exploiting randomized singular value decomposition (SVD), 2) initialize and update factor matrices and a core tensor using the SVD results, and 3) carefully determine the ordering of computations for efficiency. Similar to D-Tucker, D-TuckerO tackles Tucker decomposition for an online streaming setting with the following ideas: 1) avoid direct computations related to previous time steps, 2) approximate each new incoming tensor, and then 3) carefully update factor matrices by determining the ordering of computations.

D-Tucker has three main phases: approximation, initialization, and iteration (see Figure 3.1). The approximation phase of D-Tucker slices an input tensor into matrices, and then performs randomized SVD [83] of each sliced matrix. It allows us to reduce the size of the input tensor for updating the factor matrices and the core tensor. The initialization phase of D-Tucker initializes factor matrices by computing orthogonal factor matrices using the SVD results of sliced matrices. The iteration phase of D-Tucker updates the factor matrices and the core tensor by carefully exploiting the SVD results. D-Tucker achieves better time and space efficiency by carefully dealing with SVD results. Experimental results show that D-Tucker is faster and more memory-efficient than existing methods.

In an online streaming setting, D-TuckerO efficiently deals with each new in-

coming tensor by updating the temporal factor matrix, and then updating factor matrices of non-temporal modes. To update the temporal factor matrix, we leverage only the new incoming tensor and factor matrices of non-temporal modes obtained at the previous time step. For factor matrices of non-temporal modes, we avoid direct computations related to the entire tensor and the temporal factor matrix obtained at previous time steps. It enables that computational cost and memory requirements are proportional to the size of a new incoming tensor, not the entire tensor. In addition, at each time step, we approximate a new incoming tensor using the approximation phase of D-Tucker, and then update the factor matrices by carefully using the approximation results. Exploiting the approximation phase gives D-TuckerO the same benefit as D-Tucker: it allows us to use a smaller size of the approximated results than that of a new incoming tensor in updating the factor matrices and the core tensor, to achieve better time and space efficiency. Through comprehensive experiments, we show that D-TuckerO is more efficient than existing streaming methods, and the running time of D-TuckerO is proportional to the size of a newly arrived tensor, not the accumulated tensor.

The contributions of the paper are as follows.

- **Algorithm.** We propose D-Tucker and D-TuckerO, efficient methods for decomposing dense tensors in static and online streaming settings.
- **Analysis.** We provide analysis for the time and space complexities of our proposed methods D-Tucker and D-TuckerO.
- **Experiment.** We experimentally show that D-Tucker 1) is up to 38.4× faster and requires up to 17.2× less space than competitors (see Figure 3.3), and 2) provides good starting points to minimize the running time. Moreover, D-Tucker is scalable in handling dense tensors in terms of dimensionality, rank,

Table 3.1: Symbol description.

| Symbol | Description |
|---|---|
| $\mathcal{X}_r$ | Reordered tensor ($\in I_1 \times I_2 \times K_3 \times \ldots \times K_N$) |
| $\mathcal{G}$ | Core tensor ($\in J_1 \times J_2 \times \ldots \times J_N$) |
| $\mathbf{A}^{(n)}$ | Factor matrix of the $n$-th mode |
| $I_n$ | Dimensionality of the $n$-th mode of $\mathcal{X}_r$ for modes $n = 1$ and 2 |
| $K_n$ | Dimensionality of the $n$-th mode of $\mathcal{X}_r$ for mode $n = 3, 4, \ldots, N$ |
| $J_n$ | Dimensionality of the $n$-th mode of core tensor |
| $\left[\mathbf{X}\ ;\ \mathbf{Y}\right]$ | Horizontal concatenation of two matrices $\mathbf{X}$ and $\mathbf{Y}$ |
| $\mathbf{X}_{::k_3 \ldots k_N}$ | $(k_3, \ldots, k_N)$-th sliced matrix of size $I_1 \times I_2$ |
| $\mathbf{U}_{::k_3 \ldots k_N}$ | Left singular vector matrix of $\mathbf{X}_{::k_3 \ldots k_N}$ |
| $\mathbf{\Sigma}_{::k_3 \ldots k_N}$ | Singular value matrix of $\mathbf{X}_{::k_3 \ldots k_N}$ |
| $\mathbf{V}_{::k_3 \ldots k_N}$ | Right singular vector matrix of $\mathbf{X}_{::k_3 \ldots k_N}$ |
| $L$ | Number of sliced matrices ($= K_3 \times \cdots K_N$) |
| $r$ | Number of singular values for SVD |
| $N$ | Order of the given tensor |
| $\varepsilon$ | Error tolerance in the iteration phase |
| $t_{new}$ | New time-step in an online streaming setting |
| $\mathcal{X}_{old}$ | Accumulated tensor |
| $\mathcal{X}_{new}$ | New time slice at a time step $t_{new}$ |
| $T_{old}$ | Dimensionality of the temporal mode of an accumulated tensor ($\in I_1 \times I_2 \times K_3 \times \ldots \times T_{old}$) |
| $T_{new}$ | Dimensionality of the temporal mode of a new time slice ($\in I_1 \times I_2 \times K_3 \times \ldots \times T_{new}$) |
| $blkdiag(\{\mathbf{A}_l\}_{l=1}^{L})$ | Block diagonal matrix consisting of $\mathbf{A}_l$ for $l = 1, \ldots L$ (see Equation (3.5)) |
| $\mathbf{A}_{old}^{(n)}$ | Pre-existing factor matrix of the $n$-th mode in an online streaming setting |
| $\otimes$ | Kronecker product |
| $\dagger$ | Pseudoinverse |

order, and the number of iterations. D-TuckerO is up to $6.1\times$ faster than competitors in an online streaming setting (see Figure 3.7).

In the rest of the chapter, we describe the preliminaries in Section 3.2, propose our methods D-Tucker and D-TuckerO in Sections 3.3 and 3.4, respectively, present experimental results in Section 3.5, and conclude in Section 3.6. The code and datasets are available at https://datalab.snu.ac.kr/dtucker.

## 3.2 Preliminaries

Table 3.1 shows the symbols used in this chapter.

**Algorithm 3:** Randomized SVD [84]

**Input:** matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank $k$, and sampling parameters $p$ and $l$
**Output:** SVD results $\mathbf{U} \in \mathbb{R}^{m \times k}$, $\mathbf{\Sigma} \in \mathbb{R}^{k \times k}$, $\mathbf{V} \in \mathbb{R}^{n \times k}$

1: draw random matrices $\mathbf{\Omega} \in \mathbb{R}^{p \times m}$ and $\mathbf{\Psi} \in \mathbb{R}^{l \times n}$
2: form matrices $\mathbf{Y} = \mathbf{\Omega A}$ and $\mathbf{Z} = \mathbf{\Psi A}^T$
3: obtain column orthogonal matrices $\mathbf{Q}$ and $\mathbf{P}$ by QR factorization of $\mathbf{Y}^T$ and $\mathbf{Z}^T$.
4: form matrices $\mathbf{W} = \mathbf{\Omega P}$ and $\mathbf{B} = \mathbf{YQ}$.
5: obtain a matrix $\mathbf{X}$ which minimizes $\|\mathbf{WX} - \mathbf{B}\|$
6: compute SVD of $\mathbf{X} = \tilde{\mathbf{U}}\mathbf{\Sigma}\tilde{\mathbf{V}}^T$
7: $\mathbf{U} \leftarrow \mathbf{P}\tilde{\mathbf{U}}_k$, $\mathbf{\Sigma} \leftarrow \tilde{\mathbf{\Sigma}}_k$, $\mathbf{V} \leftarrow \mathbf{Q}\tilde{\mathbf{V}}_k$

### 3.2.1 Singular Value Decomposition (SVD)

Given a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, Singular Value Decomposition (SVD) decomposes it into the three matrices $\mathbf{U} \in \mathbb{R}^{m \times r}$, $\mathbf{\Sigma} \in \mathbb{R}^{r \times r}$, and $\mathbf{V} \in \mathbb{R}^{n \times r}$ where $\mathbf{X}$ is equal to $\mathbf{U\Sigma V}^{\mathrm{T}}$. $\mathbf{U}$ is a column orthogonal matrix (i.e., $\mathbf{U}^T\mathbf{U} = \mathbf{I}$) consisting of left singular vectors of $\mathbf{X}$; $\mathbf{\Sigma}$ is an $r \times r$ diagonal matrix consisting of singular values $\sigma_r$ where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$. $\mathbf{V} \in \mathbb{R}^{n \times r}$ is a column orthogonal matrix (i.e., $\mathbf{V}^T\mathbf{V} = \mathbf{I}$) consisting of right singular vectors of $\mathbf{X}$.

**SVD with randomized algorithm**. Randomized SVD efficiently approximates a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with a low rank using randomization techniques (See Algorithm 3). The main idea of randomized SVD is 1) to generate random matrices $\mathbf{\Omega} \in \mathbb{R}^{p \times m}$ and $\mathbf{\Psi} \in \mathbb{R}^{l \times n}$ where $p$ and $l$ are sampling parameters, and find column orthogonal matrices $\mathbf{Q} \in \mathbb{R}^{n \times p}$ and $\mathbf{P} \in \mathbb{R}^{m \times l}$ of sketches $\mathbf{Y}^T = (\mathbf{\Omega}\boldsymbol{A})^T \in \mathbb{R}^{n \times p}$ and $\mathbf{Z}^T = (\mathbf{\Psi A}^T)^T \in \mathbb{R}^{m \times l}$, respectively, 2) to construct a smaller matrices $\mathbf{W} = \mathbf{\Omega P} \in \mathbb{R}^{p \times l}$ and $\mathbf{B} = \mathbf{YQ} \in \mathbb{R}^{p \times p}$, and find $\mathbf{X} \in \mathbb{R}^{l \times p}$ that minimizes $\|\mathbf{WX} - \mathbf{B}\|$, 3) to compute $\mathbf{X} \approx \tilde{\mathbf{U}}_k\mathbf{\Sigma}_k\tilde{\mathbf{V}}_k^T$ by truncated SVD at target rank $k$, and 4) to compute $\mathbf{U} = \mathbf{P}\tilde{\mathbf{U}}_k \in \mathbb{R}^{m \times k}$ and $\mathbf{V} = \mathbf{Q}\tilde{\mathbf{V}}_k \in \mathbb{R}^{n \times k}$. The dominant terms to compute randomized SVD are to form sketches $\mathbf{Y}$ and $\mathbf{Z}$. Recent works [38, 85] require $O(mn)$ time to construct random matrix and form matrix $\mathbf{Y}$ using sparse embedding matrix $\mathbf{\Omega} \in \mathbb{R}^{p \times m} = \mathbf{\Phi}\boldsymbol{D}$.

- $h$: $[m] \to [p]$ is a random map so that $h(m') = p'$ for $p' \in [p]$ with probability $1/p$ for each $m' \in [m]$, where $[m] = \{1, 2, ..., m\}$ and $[p] = \{1, 2, ..., p\}$.
- $\mathbf{\Phi} \in \{0, 1\}^{p \times m}$: for each $m'$-th column of $\mathbf{\Phi}$, all the entries are 0 except that $h(m')$-th entry is 1; each column vector is a one-hot encoding vector whose only one entry is 1 and remaining entries are 0.
- Diagonal matrix $\mathbf{D} \in \mathbb{R}^{m \times m}$: diagonal entries are randomly chosen to be 1 or $-1$ with equal probability.

Due to the special form of $\mathbf{\Phi}$ and $\mathbf{D}$, the complexity of multiplying $\mathbf{\Omega}$ to $\mathbf{A}$ is $O(mn)$ (see [85] for details). $\mathbf{Z}$ is also constructed like $\mathbf{Y}$ using sparse embedding matrix. Therefore, the time complexity of randomized SVD is $O(mn)$ when we use sparse embedding matrices. In the paper, we use randomized SVD to efficiently deal with large dense matrices in the approximation phase. We use standard SVD [86] with time complexity $O(mnk)$ to stably deal with relatively small matrices in the initialization and iteration phases.

### 3.2.2 Streaming Tucker Decomposition

We formally define the problem of Tucker decomposition in an online streaming setting as follows:

**Definition 3.1.** (Tucker decomposition in a streaming fashion)
***Given:*** *a time slice* $\mathcal{X}_{new} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{new}}$ *at a time-step* $t_{new}$*, a pre-existing set of factor matrix* $\mathbf{A}^{(n)}_{old}$ *for* $n = 1, 2, ..., N$*, and a pre-existing core tensor* $\mathcal{G}_{old}$ *where* $\mathbf{A}^{(n)}_{old}$ *and* $\mathcal{G}_{old}$ *approximate* $\mathcal{X}_{old} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{old}}$*,*
***Update:*** *the factor matrix* $\mathbf{A}^{(n)}_{new}$ *for* $n = 1, 2, ..., N$ *and the core tensor* $\mathcal{G}_{new}$ *to approximate the accumulated tensor* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{total}}$ *where* $T_{total} = T_{old} + T_{new}$*.*

$\mathbf{A}^{(n)}_{new} \in \mathbb{R}^{I_n \times J_n}$ (or $\mathbb{R}^{K_n \times J_n}$) for $n = 1, 2, ..., N-1$ is a factor matrix updated at $t_{new}$, $\mathbf{A}^{(N)}_{inc} \in \mathbb{R}^{T_{new} \times J_N}$ is the temporal factor matrix corresponding to $t_{new}$, and $\mathbf{A}^{(N)}_{new} = \begin{bmatrix} \mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc} \end{bmatrix} \in \mathbb{R}^{T_{total} \times J_N}$ is the temporal factor matrix corresponding to $t_{total} = t_{old} + t_{new}$ where $\mathbf{A}^{(N)}_{old} \in \mathbb{R}^{T_{old} \times J_N}$ is the pre-existing temporal factor matrix.

**Computing the Tucker decomposition in an online streaming setting.** We can deal with a newly arrived tensor using a static version of Tucker decomposition. However, it is inevitable that running times and memory requirements increase over time. Recent works have tried to update factor matrices and a core tensor without the growth of the costs. DTA [17] updates factor matrices and core tensor by efficiently updating covariance matrices $\mathbf{X}^T_{(n)}\mathbf{X}_{(n)}$. STA [17] is an approximate version of DTA by exploiting SPIRIT [87] which efficiently deals with newly arrived vectors. Tucker-ts and Tucker-ttmts can be adapted to an online streaming setting: 1) approximating each newly arrived tensor using a sketching technique, and 2) updating factor matrices and core tensor using the approximated results of the whole tensor. Although they avoid increasing running time and memory requirements over time, there remains a need for accelerating the update process since computations involved with a large dense incoming tensor are still time-consuming. To efficiently update factor matrices and a core tensor in an online streaming setting, we need to 1) prevent the increase of cost over time, 2) reduce the cost of approximating a newly arrived tensor, and 3) update them using the approximated results of the newly arrived tensor.

## 3.3 Proposed Method for Static Tensors: D-Tucker

We propose D-Tucker, a fast and memory-efficient Tucker decomposition method for large-scale dense tensors. We first give an overview of D-Tucker in Section 3.3.1. We

describe details of D-Tucker in Sections 3.3.2 to 3.3.4. Finally, we analyze D-Tucker's complexities in Sections 3.3.5 and 3.3.6.

### 3.3.1 Overview

D-Tucker efficiently computes Tucker decomposition of large dense tensors. The main challenges are as follows:

1. **Exploiting the characteristics of real-world tensors.** Many real-world tensors are dense, provoking time and space problems. Furthermore, many real-world tensors are skewed (i.e., one of the dimensionality is much smaller than the others) and have low dimensional structures. How can we exploit such characteristics of real-world tensors to compress a dense input tensor with low computational cost and error?

2. **Minimizing intermediate data.** Existing methods require heavy computations and large space while updating factor matrices and a core tensor in the iteration phase. How can we minimize the size of intermediate data when updating the factor matrices and the core tensor?

3. **Reducing numerical computation.** Tucker decomposition deals with a large number of tensor computations. How can we reduce the computational time of Tucker decomposition?

We address the above challenges with the following ideas:

1. **Slicing an input tensor into matrices and computing randomized SVD of sliced matrices** minimize the computational cost and error, by utilizing the low dimensional structure of sliced matrices (Section 3.3.2).

**Algorithm 4:** D-Tucker

**Input:** tensor $\mathcal{X}$
**Output:** factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$, and core tensor $\mathcal{G}$
**Parameters:** rank $J_i$ $(i = 1, 2, ..., N)$, and error tolerance $\varepsilon$

1: approximate slices of $\mathcal{X}$ by Algorithm 5
2: initialize factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$ by Algorithm 6
3: **repeat**
4: update factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$ and core tensor $\mathcal{G}$ by Algorithm 7
5: **until** the maximum iteration is reached, or the error difference is smaller than the error tolerance $\varepsilon$

2. **Avoiding the reconstruction from SVD results** reduces the computational time as well as memory usage. By replacing a dense input tensor with SVD results of sliced matrices, we overcome a bottleneck of Tucker decomposition, $n$-mode product with a dense input tensor (Sections 3.3.3 and 3.3.4).

3. **Careful ordering for matrix operations** reduces the memory usage and minimizes the computations. (Sections 3.3.3 and 3.3.4)

As shown in Figure 3.1 and Algorithm 4, D-Tucker comprises three phases: approximation (Algorithm 5), initialization (Algorithm 6), and iteration (Algorithm 7). In the approximation phase, D-Tucker reorders modes of the input tensor in descending order for efficiency, extracts matrices of size $I_1 \times I_2$ by slicing the reordered tensor where $I_1$ and $I_2$ are the two largest dimensionalities, and performs randomized SVD of sliced matrices in order to support fast and memory-efficient Tucker decomposition (line 1 in Algorithm 4). In the initialization phase, we obtain initial factor matrices using the SVD results of sliced matrices (line 2 in Algorithm 4). This phase provides a good starting point for the iteration phase, reducing the number of iterations. In the iteration phase, we obtain the factor matrices and the core tensor using the initial factor matrices and the SVD results of sliced matrices (line 4 in Algorithm 4).

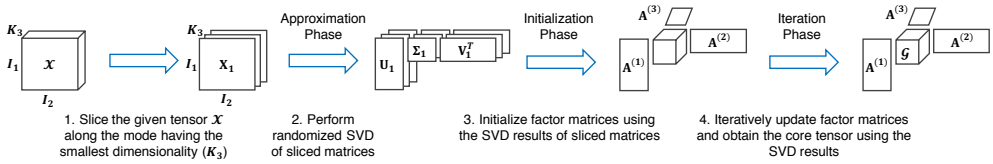



Figure 3.1: Overview of D-Tucker. We first slice the given 3-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3}$ along the mode having the smallest dimensionality ($K_3$), and approximate sliced matrices using singular value decomposition (SVD). Then, we compute factor matrices using the SVD results of sliced matrices in the initialization step. We iteratively update factor matrices using SVD results of sliced matrices. After that, we obtain the core tensor using the updated factor matrices and SVD results of sliced matrices.

### 3.3.2 Approximation Phase

The main goal of the approximation phase is to compress the input tensor with low error; it enables the iteration phase to reduce the memory requirements and the number of flops. Given a large-scale dense tensor, previous works based on ALS approach require heavy computations and memory usage in updating a factor matrix at each iteration step since they directly process the given tensor. Although a few methods tried to solve the above problem by approximating the input tensor, they give high errors, or require heavy computations. The approximation phase of D-Tucker enables efficiently updating the factor matrices and the core tensor in the iteration phase based on two characteristics of real-world tensors: 1) skewed shape, and 2) low dimensional structure in sliced matrices. We reorder modes of a given tensor based on the first characteristic, and compress the sliced matrices of the reordered tensor using a fast dimensionality reduction technique, randomized SVD.

**Skewed shape of real-world tensors.** A skewed shape, where there are gaps between the dimensionalities of modes, exists in many real-world tensors. For example, a 3-order Air Quality tensor (see Table 3.3) of size $(30648, 376, 6)$ in the form of (timestamp in second, location, atmospheric pollutants; measurement) has a skewed shape where the dimensionality of the last mode is much smaller than those of oth-

**Algorithm 5:** Approximation phase of D-Tucker

**Input:** tensor $\mathcal{X}$
**Output:** sets of SVD result $\mathbf{U}_{::k_3,\ldots,k_N}\mathbf{\Sigma}_{::k_3,\ldots,k_N}\mathbf{V}^{\mathrm{T}}_{::k_3,\ldots,k_N}$ of sliced matrix $\mathbf{X}_{::k_3,\ldots,k_N}$
**Parameters:** rank $r$

1: reorder modes of the input tensor by dimensionality in descending order
2: extract the matrix $\mathbf{X}_{::k_3,\ldots,k_N} \in \mathbb{R}^{I_1 \times I_2}$ by slicing the reordered tensor where $I_1$ and $I_2$ are the two largest dimensionalities
3: **for each** $(k_3, \ldots, k_N)$ **do**
4: perform randomized SVD of $\mathbf{X}_{::k_3,\ldots,k_N} \simeq \mathbf{U}_{::k_3,\ldots,k_N}\mathbf{\Sigma}_{::k_3,\ldots,k_N}\mathbf{V}^{\mathrm{T}}_{::k_3,\ldots,k_N}$
5: **end for**

ers. We reorder modes in descending order of dimensionality (line 1 in Algorithm 5). Reordered tensor is defined as follows:

**Definition 3.2** (Reordered tensor $\mathcal{X}_r$)**.** *Given an $N$-mode input tensor $\mathcal{X}$, we reorder the input tensor by dimensionality in descending order. We represent the reordered tensor as $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$ where $I_1$ and $I_2$ are the two largest dimensionalities, $K_n$ for $n = 3, 4, \ldots, N$ are the remaining dimensionalities, and $I_1 \geq I_2 \geq K_3 \geq \cdots \geq K_N$.* □

This reordering helps minimize the output size of the approximation phase, which is described in the analysis of space complexity in Section 3.3.5.

**Low dimensional structure in sliced matrices.** Many real-world data represented as a matrix have a low dimensional structure since they have redundant and correlated components. Similarly, sliced matrices of a given real-world tensor for any two modes often have a low dimensional structure. For example, consider the 3-order Air Quality tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3}$ of size $(30648, 376, 6)$ in Table 3.3 containing (timestamp in second, location, atmospheric pollutants; measurement), sliced along modes 3. Out of the 6 sliced matrices, the $i$th sliced matrix $\mathbf{X}_{::i} \in \mathbb{R}^{I_1 \times I_2}$ indicates the matrix containing (timestamp in second, location; measurement) for the $i$th atmospheric pollutant. We observe that the number $r$ of singular values to keep 90% energy of each sliced matrix is $(28, 8, 6, 7, 6, 18)$, which is much smaller than

$I_1 = 30648$ and $I_2 = 360$. Note that the energy of a matrix $\mathbf{X}_{::i} \in \mathbb{R}^{I_1 \times I_2}$ is defined as $\sum_{r=1}^{min(I_1,I_2)} \sigma_r^2$ where $\sigma_r$ is the $r$th singular value of $\mathbf{X}_{::i}$. This result indicates that the sliced matrices have low dimensional structures. D-Tucker compresses the given tensor by exploiting the low dimensional structure, achieving low errors. Moreover, this structure provides the following computational benefit: the approximation phase of D-Tucker yields faster performance by leveraging the randomized SVD [84] of sliced matrices. It enables us to avoid performing tensor decomposition methods for sub-tensors, which makes D-Tucker efficient since the tensor-based methods iteratively perform expensive operations such as $n$-mode product. Therefore, D-Tucker achieves high efficiency and low errors even on single-core systems.

We express a reordered tensor $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$ as a collection of sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$. We formally define the sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$ in Definition 3.3.

**Definition 3.3** (Sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$)**.** *Given a reordered tensor $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$, each sliced matrix of size $I_1 \times I_2$ is extracted by slicing the reordered tensor $\mathcal{X}_r$. The size of a sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$ is $I_1 \times I_2$ where $I_1$ is the number of rows and $I_2$ is the number of columns of the sliced matrix.* □

After slicing the tensor $\mathcal{X}_r$ into the matrices $\mathbf{X}_{::k_3 \ldots k_N}$, we decompose the sliced matrix using randomized SVD [84] with sparse embedding matrix [38, 85] (line 4 in Algorithm 5).

$$\mathbf{X}_{::k_3 \ldots k_N} \simeq \mathbf{U}_{::k_3 \ldots k_N} \mathbf{\Sigma}_{::k_3 \ldots k_N} \mathbf{V}^{\mathrm{T}}_{::k_3 \ldots k_N} \tag{3.1}$$

where $\mathbf{U}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{I_1 \times r})$ is a left singular vector matrix, $\mathbf{\Sigma}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{r \times r})$ is a singular value matrix, and $\mathbf{V}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{I_2 \times r})$ is a right singular vector matrix. Note that the number $r$ of singular values is much smaller than the dimensionalities $I_1$ and $I_2$. By

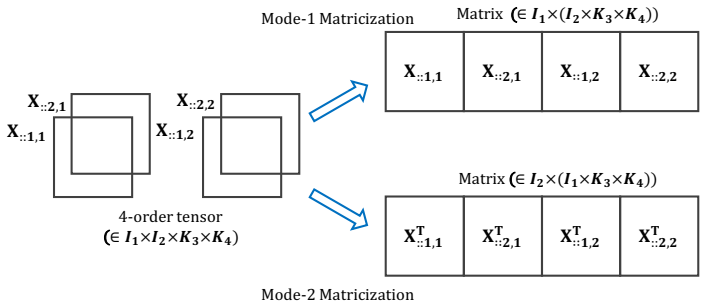



Figure 3.2: Example of matricizing a 4-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times K_4}$ using sliced matrices for the first and the second mode when $K_3 = 2$ and $K_4 = 2$.

computing Equation (3.1) for all sliced matrices, we achieve high efficiency in terms of time and space, to obtain factor matrices and a core tensor. In the following initialization and iteration phases, we describe how to perform Tucker decomposition efficiently with the SVD results of sliced matrices rather than the raw input tensor.

### 3.3.3 Initialization Phase

The initialization phase, which initializes factor matrices of a given tensor $\mathcal{X}$, enables the iteration phase to reduce the number of iterations by providing a good starting point of the ALS algorithm. The main challenge is how to handle the SVD results for efficient initialization of the factor matrices. Truncated HOSVD has provided good initial factor matrices to compute Tucker decomposition based on ALS approach [28]. However, truncated HOSVD has limitations in efficiently initializing factor matrices using the SVD results because it cannot avoid reconstructing the reordered tensor for the mode $i = 3, 4, \ldots, N$ using the SVD results. To avoid reconstructing the reordered tensor using the SVD results, we apply Sequentially Truncated Higher-Order SVD (ST-HOSVD) [88], which is a variant of HOSVD. Note that ST-HOSVD obtains factor matrix $\mathbf{A}^{(i)}$ which contains left singular vectors of mode-$i$ matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathbf{T}}$. Applying ST-SHOVD allows us

to efficiently initialize factor matrices using the results of the approximation phase, in contrast to HOSVD. The detail is described in initializing factor matrices for the remaining modes.

D-Tucker initializes factor matrices by obtaining the left singular vectors efficiently using the SVD results. For the first mode, we efficiently obtain the factor matrix by reusing the SVD results from the approximation phase. For the second mode, we efficiently compute mode-1 product between the first factor matrix and the SVD results by carefully ordering matrix multiplications, and then obtain the initial factor matrix. For the remaining modes, we process a small tensor computed by $n$-mode products between the SVD results and the factor matrices of the first and the second modes. These enable D-Tucker to achieve high efficiency in terms of time and space. Note that we use standard SVD [86] in the initialization phase since the randomized SVD can decrease the effectiveness of the initialization. We describe the initialization of the first two modes corresponding to the dimensionalities $I_1$ and $I_2$, and then describe those of remaining modes [82].

**First mode.** Our goal is to initialize the factor matrix of the first mode as left singular vectors of mode-1 matricization of $\mathcal{X}$. A naive approach would compute SVD of mode-1 matricization of $\mathcal{X}$. However, this approach requires heavy computation and high memory usage since it directly deals with a large-scale dense tensor. Our idea is to avoid reconstructing $\mathcal{X}$ from the SVD results of sliced matrices, initializing the factor matrix of the first mode. Without the reconstruction, we reduce the computational cost and memory usage.

As shown in Figure 3.2, we represent mode-1 matricized matrix $\mathbf{X}_{(1)}$ of the re-

**Algorithm 6:** Initialization phase of D-Tucker

**Input:** SVD results $\mathbf{U}_l$, $\mathbf{\Sigma}_l$, and $\mathbf{V}_l$ for $l = 1, 2, ..., L$
where $L$ is the number of sliced matrices
**Output:** initialized factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$
**Parameters:** rank $J_i$ $(i = 1, 2, ..., N)$

1: perform SVD of $\left[\mathbf{U}_1\mathbf{\Sigma}_1; \cdots; \mathbf{U}_L\mathbf{\Sigma}_L\right] \simeq \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$
2: $\mathbf{A}^{(1)} \leftarrow \mathbf{U}$
3: compute $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$
4: $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{(2),inter}$
5: $\mathbf{Y}_{(2),inter} \leftarrow \mathbf{Y}_{(2),inter} blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$
6: $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(2),inter}, [J_1, I_2, K_3, \cdots, K_N])$
7: $\mathbf{A}^{(2)} \leftarrow J_2$ leading singular vectors of $\mathbf{Y}_{(2)}$
8: **for** $i \leftarrow 3$ to $N$ **do**
9:  **if** $i = 3$ **then**
10:  $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{reuse} blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\mathbf{A}^{(2)}\}_{l=1}^{L})$
11:  $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{reuse}, [J_1, J_2, K_3, \cdots, K_N])$
12:  **else**
13:  $\mathcal{Y} \leftarrow \mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)\mathbf{T}}$
14:  **end if**
15:  $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
16:  $\mathcal{Y}_{reuse} \leftarrow \mathcal{Y}$
17: **end for**

ordered tensor $\mathcal{X}_r$ as follows:

$$\mathbf{X}_{(\mathbf{1})} = \left[\mathbf{X}_{::1,\ldots,1}; \cdots; \mathbf{X}_{::K_3,\ldots,K_N}\right] = \left[\mathbf{X}_1; \cdots; \mathbf{X}_l; \cdots; \mathbf{X}_L\right]$$

where $L$ is equal to $K_3 \times \cdots \times K_N$, and the index $l$ is defined as in Equation (3.2).

$$l = 1 + \sum_{i=3}^{N}\left((k_i - 1)\prod_{m=3}^{i-1} K_m\right) \tag{3.2}$$

where $K_m$ is the dimensionality of mode-$m$, $N$ is the order of the input tensor, and $\prod_{m=3}^{i-1} K_m$ is equal to 1 if $i - 1 < m$. Note that we represent a sliced matrix as $\mathbf{X}_l$ with the index $l$ instead of $\mathbf{X}_{::k_3\ldots k_N}$ for brevity. Using the SVD of sliced matrices, the mode-

1 matricized matrix $\mathbf{X}_{(1)}$ is expressed as follows:

$$\mathbf{X}_{(\mathbf{1})} = \left[\mathbf{X}_1; \cdots; \mathbf{X}_l; \cdots; \mathbf{X}_L\right] \simeq \left[\tilde{\mathbf{X}}_1; \cdots; \tilde{\mathbf{X}}_l; \cdots; \tilde{\mathbf{X}}_L\right] \tag{3.3}$$

where $\tilde{\mathbf{X}}_l$ is a representation of $\mathbf{U}_l\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}$. The computational cost to explicitly reconstruct the matrices $\tilde{\mathbf{X}}_l$ for $l = 1..L$ from SVD results and to obtain left singular vectors of $\mathbf{X}_{(1)}$ is expensive in terms of time and space. D-Tucker obtains left singular vectors of the first mode without the reconstruction of $\tilde{\mathbf{X}}_l$. The main idea is to carefully decouple $\mathbf{U}_l\mathbf{\Sigma}_l$ and $\mathbf{V}_l^{\mathrm{T}}$, and perform SVD of a concatenated matrix consisting of $\mathbf{U}_l\mathbf{\Sigma}_l$ for $l = 1..L$. The above idea allows us to efficiently obtain left singular vectors of the concatenated matrix based on block structure [89, 81]. Performing SVD of the concatenated matrix ($\in \mathbb{R}^{I_1 \times (r \times K_3 \times \cdots \times K_N)}$) consisting of $\mathbf{U}_l\mathbf{\Sigma}_l$ for $l = 1..L$ is more efficient than SVD of the mode-1 matricized matrix $\mathbf{X}_{(1)}$ ($\in \mathbb{R}^{I_1 \times (I_2 \times K_3 \times \cdots \times K_N)}$) (line 1 in Algorithm 6).

$$\mathbf{X}_{(\mathbf{1})} \simeq \left[\mathbf{U}_1\mathbf{\Sigma}_1; \cdots; \mathbf{U}_L\mathbf{\Sigma}_L\right] \times (blkdiag(\{\mathbf{V_l}\}_{l=1}^{L}))^{\mathbf{T}} \simeq \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}(blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}))^{\mathbf{T}} \tag{3.4}$$

where $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$ is the SVD result of the concatenated matrix $\left[\mathbf{U}_1\mathbf{\Sigma}_1; \cdots; \mathbf{U}_L\mathbf{\Sigma}_L\right]$, and the number $L$ of sliced matrices is equal to $K_3 \times \cdots \times K_N$. $blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}) \in \mathbb{R}^{I_2L \times rL}$ is a block diagonal matrix consisting of $\mathbf{V}_l \in \mathbb{R}^{I_2 \times r}$ for $l = 1, ..., L$:

$$blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}) = \begin{bmatrix} \mathbf{V}_1 & \mathbf{O} & \cdots & \mathbf{O} \\ \mathbf{O} & \mathbf{V}_2 & \cdots & \mathbf{O} \\ \vdots & \mathbf{O} & \ddots & \vdots \\ \mathbf{O} & \mathbf{O} & \cdots & \mathbf{V}_L \end{bmatrix} \tag{3.5}$$

$\mathbf{U}$ and $\mathbf{V}^{\mathrm{T}}(blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}))^{\mathbf{T}}$ are column orthogonal and $\Sigma$ has the property of singular value matrix, and thus the last term of Equation (3.4) has the SVD form. Therefore we obtain the initial factor matrix $\mathbf{A}^{(1)} = \mathbf{U}$ (line 2 in Algorithm 6).

**Second mode.** Our goal is to initialize the factor matrix of the second mode as left singular vectors of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}}$ like ST-HOSVD. As in the first mode, a naive approach would compute SVD of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}}$, but it has the same problems of heavy computational cost and high memory requirement. Our idea is to compute $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}}$ without reconstructing $\mathcal{X}$ from a set of SVD results of sliced matrices, and then compute SVD of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}}$. By avoiding the reconstruction, we reduce the computational cost and memory usage to compute $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}}$.

To compute $n$-mode product for mode-1, we exploit SVD results computed from the approximation phase, instead of the given tensor, and then obtain left singular vectors for the second mode. In detail, we perform matrix multiplication between $\mathbf{A}^{(1)\mathbf{T}}$ and mode-1 matricized matrix described in Equation (3.3) as follows:

$$\begin{aligned} \mathbf{A}^{(1)\mathbf{T}}\mathbf{X}_{(\mathbf{1})} &\simeq \mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1\mathbf{\Sigma}_1\mathbf{V}_1^{\mathrm{T}};\cdots;\mathbf{U}_L\mathbf{\Sigma}_L\mathbf{V}_L^{\mathrm{T}}\right] \\ &= \left(\mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1;\cdots;\mathbf{U}_L\right]\right) blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L}) = \mathbf{Y}_{(2),inter} blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L}) \end{aligned} \tag{3.6}$$

where $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1;\cdots;\mathbf{U}_L\right]$, $L$ is equal to $K_3 \times \cdots \times K_N$, and $blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ is a block diagonal matrix consisting of $\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}$. In Equation (3.6), $\mathbf{Y}_{(2),inter}$ is computed, and then multiplied with the block diagonal matrix. After reshaping the result of $\mathbf{Y}_{(2),inter} blkdiag(\{\mathbf{\Sigma}_l\ \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ as a tensor $\mathcal{Y}$ of the size $J_1 \times I_2 \times K_3 \times \cdots \times K_N$, we compute left singular vectors of mode-2 matricized matrix $\mathbf{Y}_{(2)}$ to initialize $\mathbf{A}^{(2)}$ (lines 3 to 7 in Algorithm 6).

**Remaining modes.** For mode $i = 3, ..., N$, our goal is to initialize $\mathbf{A}^{(i)}$ as left singular vectors of mode-$i$ matricization $\mathbf{Y}_{(i)}$ of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathbf{T}}$. For a mode-$i$, explicitly computing $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}} \cdots \times_{i-2} \mathbf{A}^{(i-2)\mathbf{T}}$ is inefficient since it is computed for the previous mode-$(i-1)$. Our idea is to reuse the result of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}} \cdots \times_{i-2} \mathbf{A}^{(i-2)\mathbf{T}}$ to initialize the factor matrix of the mode-$i$.

Now, we describe how to obtain the factor matrix of the third mode, and then the factor matrix of the modes $i = 4, 5, ..., N$. For mode-3, the goal is to obtain $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}}$, and perform SVD. The following equation re-expresses the mode-1 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}}$.

$$\begin{aligned}
&\mathbf{A}^{(1)\mathbf{T}}\mathbf{X}_{(\mathbf{1})} blkdiag(\{\mathbf{A}^{(2)}\}_{l=1}^{L}) \simeq \left(\mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1; \cdots ; \mathbf{U}_L\right]\right) blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L}) \\
&= \mathbf{Y}_{(2),inter} blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L})
\end{aligned} \tag{3.7}$$

Note that we save $\mathbf{Y}_{reuse} = \left(\mathbf{A}^{(1)\mathbf{T}}\left[\mathbf{U}_1; \cdots ; \mathbf{U}_L\right]\right)$ to reuse when computing Equation (3.7) (line 4 in Algorithm 6). After computing Equation (3.7), we 1) reshape the result as a tensor $\mathcal{Y}$ of size $J_1 \times J_2 \times K_3 \times \cdots \times K_N$, 2) perform SVD of $\mathbf{Y}_{(3)}$, and 3) store $\mathcal{Y}$ as $\mathcal{Y}_{reuse}$ for remaining modes (lines 10, 11, 15, and 16 in Algorithm 6).

Next, factor matrices for mode $i = 4, 5, ..., N$ are initialized by using the result of the previous mode. For mode $i$, we compute $\mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)T}$, and then perform SVD of mode-$i$ matricization of $\mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)T}$. Since $\mathbf{Y}_{(i)}$ is much smaller than an input tensor $\mathcal{X}$, we efficiently initialize factor matrices $\mathbf{A}^{(i)}$ for $i = 3, ..., N$. This is the reason why we apply ST-HOSVD, not HOSVD which requires high computational costs to compute left singular vectors for the remaining modes $i = 3, ..., N$. HOSVD needs to perform SVD of mode-$i$ matricization of $\mathcal{X}$. Then, we initialize the factor

**Algorithm 7:** Iteration phase of D-Tucker

**Input:** SVD results $\mathbf{U}_l$, $\mathbf{\Sigma}_l$, and $\mathbf{V}_l$ ( $l = 1, 2, ..., L$),
factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$, and core tensor $\mathcal{G}$
**Output:** updated factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$, and core tensor $\mathcal{G}$
**Parameters:** Rank $J_i$ $(i = 1, ..., N)$

1: **for** $i \leftarrow 1$ to 2 **do**
2:  **if** $i = 1$ **then**
3:  $\mathbf{Y}_{(1),inter} \leftarrow \mathbf{A}^{(2)\mathbf{T}} \left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]$
4:  $\mathbf{Y}_{(1),inter} \leftarrow \mathbf{Y}_{(1),inter} blkdiag(\{\mathbf{\Sigma}_l \mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$
5:  $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(1),inter}, [I_1, J_2, K_3, \cdots, K_N])$
6:  **else**
7:  $\mathbf{Y}_{(2),inter} \leftarrow \mathbf{A}^{(1)\mathbf{T}} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$
8:  $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{(2),inter}$
9:  $\mathbf{Y}_{(2),inter} \leftarrow \mathbf{Y}_{(2),inter} blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$
10:  $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(2),inter}, [J_1, I_2, K_3, \cdots, K_N])$
11:  **end if**
12:  $\mathcal{Y} \leftarrow \mathcal{Y} \times_3 \mathbf{A}^{(3)\mathbf{T}} \cdots \times_N \mathbf{A}^{(N)\mathbf{T}}$
13:  $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
14: **end for**
15: $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{reuse} blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L})$
16: $\mathcal{Y}_{reuse} \leftarrow reshape(\mathbf{Y}_{reuse}, [J_1, J_2, K_3, \cdots, K_N])$
17: **for** $i \leftarrow 3$ to $N$ **do**
18:  $\mathcal{Y} \leftarrow \mathcal{Y}_{reuse} \times_3 \mathbf{A}^{(3)\mathbf{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathbf{T}} \times_{i+1} \mathbf{A}^{(i+1)\mathbf{T}} \cdots \times_N \mathbf{A}^{(N)\mathbf{T}}$
19:  $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
20: **end for**
21: $\mathcal{G} \leftarrow \mathcal{Y}_{reuse} \times_3 \mathbf{A}^{(3)\mathbf{T}} \cdots \times_N \mathbf{A}^{(N)\mathbf{T}}$

matrix $\mathbf{A}^{(i)}$ as the left singular vectors of the SVD result (line 15 in Algorithm 6).

### 3.3.4 Iteration Phase

The goal of the iteration phase is to alternately update factor matrices and compute core tensor by efficiently computing $n$-mode products in lines 4 and 8 of Algorithm 1. As described in Section 2.2.1, a naive ALS approach is much inefficient in terms of time and space due to a large intermediate tensor including the input tensor. Furthermore, increasing the number of iterations affects the overall running time. Therefore, the main challenge of the iteration phase is how to reduce the number of flops by minimizing the intermediate data. Our ideas to tackle the challenge are to 1) exploit

the special structure of SVD results, 2) carefully determine the ordering of matrix multiplications, and 3) avoid redundant computations for the first and second modes.

Our ideas allow D-Tucker to be less affected by the number of iterations, and to avoid rapid growth of computational time as the number of iterations increases, due to the small amount of computations. We describe how to update 1) the factor matrices of the first two modes corresponding to the dimensionalities $I_1$ and $I_2$, and 2) those of other modes and the core tensor. Note that we use standard SVD [86] for stable convergence in the iteration phase.

**First mode.** Consider updating the first factor matrix $\mathbf{A}^{(1)}$. We use the initialized factor matrices and SVD results of the sliced matrices for $\mathbf{A}^{(1)}$. Following line 4 of Algorithm 1, we efficiently compute $n$-mode product for mode-2 using SVD results obtained in the approximation phase instead of the given tensor, and then perform products for remaining modes $3, 4, ..., N$. We matricize the tensor along mode-2 with the sliced matrices as follows:

$$\mathbf{X}_{(\mathbf{2})} = \left[\mathbf{X}_1^{\mathrm{T}}; \cdots; \mathbf{X}_l^{\mathrm{T}}; \cdots; \mathbf{X}_L^{\mathrm{T}}\right] \simeq \left[\tilde{\mathbf{X}}_1^{\mathbf{T}}; \cdots; \tilde{\mathbf{X}}_l^{\mathbf{T}}; \cdots; \tilde{\mathbf{X}}_L^{\mathbf{T}}\right]$$

After that, we perform matrix multiplication between $\mathbf{A}^{(2)\mathbf{T}}$ and the mode-2 matricized matrix as follows:

$$\begin{aligned} &\mathbf{A}^{(2)\mathbf{T}}\mathbf{X}_{(\mathbf{2})} \simeq \mathbf{A}^{(2)\mathbf{T}}\left[\mathbf{V}_1\mathbf{\Sigma}_1\mathbf{U}_1^{\mathrm{T}}; \cdots; \mathbf{V}_L\mathbf{\Sigma}_L\mathbf{U}_L^{\mathrm{T}}\right] \\ &= \left(\mathbf{A}^{(2)\mathbf{T}}\left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]\right) blkdiag(\{\mathbf{\Sigma}_l\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L}) = \mathbf{Y}_{(1),inter} blkdiag(\{\mathbf{\Sigma}_l\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L}) \end{aligned} \tag{3.8}$$

where $\mathbf{Y}_{(1),inter} = \mathbf{A}^{(2)\mathbf{T}}\left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]$, $L$ is equal to $K_3 \times \cdots \times K_N$, and $blkdiag(\{\mathbf{\Sigma}_l\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ is a block diagonal matrix consisting of $\mathbf{\Sigma}_l\mathbf{U}_l^{\mathrm{T}}$. In Equation (3.8), we compute $\mathbf{Y}_{(1),inter}$, and multiply it with the block diagonal matrix. Then, we reshape the result of $\mathbf{Y}_{(1),inter}$

$blkdiag(\{\mathbf{\Sigma}_l\ \mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ as $\mathcal{Y}$ of size $I_1 \times J_2 \times K_3 \times \cdots \times K_N$ (lines 3 to 5 in Algorithm 7). After that, we perform the remaining $n$-mode products with $\mathcal{Y}$ for $n = 3, 4, \ldots, N$, and then update the factor matrix $\mathbf{A}^{(1)}$ by computing SVD of mode-1 matricized matrix $\mathbf{Y}_{(1)}$ (lines 12 and 13 in Algorithm 7).

**Second mode.** Next, to update $\mathbf{A}^{(2)}$, we compute $n$-mode product for mode-1 using SVD results obtained in the approximation phase instead of the given tensor. Then, we perform $n$-mode products for remaining modes $3, 4, \ldots, N$. As in Equation (3.6), we perform matrix multiplication between $\mathbf{A}^{(1)\mathbf{T}}$ and the mode-1 matricized matrix which is the matricization of the tensor along mode-1 with the sliced matrices in Equation (3.3). For efficiency, we compute Equation (3.6) with the following order: 1) $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathbf{T}} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$, 2) multiply it with the block diagonal matrix $blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$, and 3) reshape the result of $\mathbf{Y}_{(2),inter} blkdiag\ (\{\mathbf{\Sigma}_l\ \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ as $\mathcal{Y}$ $(\in \mathbb{R}^{J_1 \times I_2 \times K_3 \times \cdots \times K_N})$ (lines 7, 9, and 10 in Algorithm 7). Note that $\mathbf{Y}_{(2),inter}$ is reused when computing $\mathbf{A}^{(i)}$ for $i = 3, 4, \ldots, N$ and the core tensor (line 8 in Algorithm 7). We update $\mathbf{A}^{(2)}$ by performing the remaining $n$-mode products with $\mathcal{Y}$ for $n = 3, 4, \ldots, N$, and computing SVD of mode-2 matricized matrix $\mathbf{Y}_{(2)}$ (lines 12 and 13 in Algorithm 7).

**Remaining modes and core tensor.** Consider updating factor matrices $\mathbf{A}^{(i)}$ for all $i = 3, 4, \ldots, N$, and the core tensor $\mathcal{G}$. The mode-1 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \times_2 \mathbf{A}^{(2)\mathbf{T}}$ is given by Equation (3.7). In computing Equation (3.7), reusing the saved $\mathbf{Y}_{(2),inter}$ at line 8 of Algorithm 7 allows us to avoid redundant computation, sufficiently reducing computational costs; the reason is that $\mathbf{Y}_{(2),inter}$ is much smaller than the input tensor $\mathcal{X}$ and the SVD results of sliced matrices. We compute $\mathbf{Y}_{(2),inter}\ blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}}\ \mathbf{A}^{(2)}\}_{l=1}^{L})$ and reshape the result $\mathcal{Y}_{reuse}$ of size $J_1 \times J_2 \times K_3 \cdots \times K_N$ once, which is reused to compute factor matrices $\mathbf{A}^{(i)}$ for $i = 3, 4, \ldots, N$ and the core tensor $\mathcal{G}$ (lines 15

and 16 in Algorithm 7). For $i = 3, \ldots, N$, we update $\mathbf{A}^{(i)}$ by performing the remaining $n$-mode products, and SVD of $\mathbf{Y}_{(i)}$ (lines 17 to 20 in Algorithm 7). In addition, we update the core tensor by performing $n$-mode products between the reshaped tensor $\mathcal{Y}_{reuse}$ $(\in \mathbb{R}^{J_1 \times J_2 \times K_3 \cdots \times K_N})$ and $\mathbf{A}^{(n)\mathbf{T}}$ for all $n = 3, 4, \ldots, N$ (line 21 in Algorithm 7).

### 3.3.5 Lemmas and Theorems

We theoretically analyze the time complexity, the space complexity, and the error of D-Tucker, as summarized in Table 3.2. For brevity, we assume $I_1 = I_2 = I$, $K_1 = K_2 = \ldots = K_N = K$, $r = J_1 = J_2 = \ldots = J_N = J$. For readability, we provide proofs of Lemmas and Theorems in Section 3.3.6.

**Time complexity.** We analyze the time complexities of D-Tucker in Theorem 3.1.

**Lemma 3.1.** *The approximation phase of D-Tucker takes $O(I^2K^{N-2})$ where $I$ is the largest dimensionality, and $K$ is the remaining dimensionality.* □

*Proof.* See the proof in Section 3.3.6.1. □

**Lemma 3.2.** *The initialization phase of D-Tucker takes $O\left(IK^{N-2}J^2\right)$ where $I$ is the largest dimensionality, $K$ is the remaining dimensionality, and $J$ is the rank.* □

*Proof.* See the proof in Section 3.3.6.2. □

**Lemma 3.3.** *The time complexity of an iteration at the iteration phase is $O(NIK^{N-2}J^2)$ where $N$ is the order of a given tensor, $I$ is the largest dimensionality, $K$ is the remaining dimensionality, and $J$ is the rank.* □

*Proof.* See the proof in Section 3.3.6.3. □

Table 3.2: Time and space costs of D-Tucker and competitors. Space cost indicates the requirement for updating factor matrices and the core tensor. Boldface denotes the optimal complexities. $I$ denotes the two largest dimensionalities, $K$ is the remaining dimensionalities, $M$ is the number of iterations, $J$ is the dimensionality of the core tensor, and $N$ is the order of the given tensor.

| Algorithm | Time | Space |
|---|---|---|
| D-Tucker | $\mathbf{O(I^2K^{N-2} + MNIK^{N-2}J^2)}$ | $O(IK^{N-2}J)$ |
| Tucker-ALS [90] | $O(MNI^2K^{N-2}J)$ | $O(I^2K^{N-2})$ |
| MACH [37] | $O(MNI^2K^{N-2}J)$ | $O(I^2K^{N-2})$ |
| RTD [36] | $O(MNI^2K^{N-2})$ | $O(I^2K^{N-2})$ |
| Tucker-ts [38] | $O(NI^2K^{N-2} + MN(IJ^N + J^{2N}))$ | $O(NIJ^N + J^{2N})$ |
| Tucker-ttmts [38] | $O(NI^2K^{N-2} + MN(IJ^{2N-2} + J^{2N-2}))$ | $O(NIJ^N + J^{2N-1})$ |

**Theorem 3.1.** *The total time complexity of D-Tucker is $O(I^2K^{N-2}J + MNIK^{N-2}J^2)$ where $M$ is the number of iterations, $N$ is the order of a given tensor, $I$ is the largest dimensionality, $K$ is the remaining dimensionality, and $J$ is the rank.* □

*Proof.* See the proof in Section 3.3.6.4. □

Note that the time complexity of the approximation phase of D-Tucker is proportional only to the size $I^2K^{N-2}$ of the input tensor without any parameters such as rank $J$ and order $N$. Also, D-Tucker is less affected by the number of iterations because the time complexity $O(NIK^{N-2}J^2)$ per iteration of the iteration phase is much smaller than the time complexity $O(I^2K^{N-2})$ of the approximation phase: $I$ is much larger than $NJ^2$ since $I \gg J$ and $I \gg N$. Thus, D-Tucker avoids rapid growth of computational time as the number of iterations increases.

**Space complexity.** We analyze space requirements of D-Tucker for initializing and updating factor matrices.

**Theorem 3.2.** *D-Tucker requires $O(IK^{N-2}J)$ space for initializing and updating factor matrices.* □

*Proof.* See the proof in Section 3.3.6.5. □

Note that the original input tensor requires $O(I^2K^{N-2})$ space. Thanks to the reordering in the approximation phase, the space complexity of D-Tucker is $I/J$ times smaller than directly using the raw input tensor. Without the reordering, the compression rate would worsen; e.g., if we have decomposed sliced matrices of size $I \times K_n$, the compression rate would have decreased to $K_n/J$, which is worse than $I/J$ since $I > K_n > J$.

### 3.3.6 Proofs of Lemmas and Theorems

We provide proofs for lemmas and theorems described in Section 3.3.5.

#### 3.3.6.1 Proof of Lemma 3.1

*Proof.* Performing randomized SVD of each sliced matrix takes $O(I^2)$ (Algorithm 3). Since the number of sliced matrices is $K^{N-2}$, the time complexity of the approximation phase is $O(I^2K^{N-2})$. □

#### 3.3.6.2 Proof of Lemma 3.2

*Proof.* For the first mode, size of $\left[\mathbf{U}_1\mathbf{\Sigma}_1;\cdots;\mathbf{U}_l\mathbf{\Sigma}_l\right]$ is $I \times K^{N-2}J$. Then, performing SVD [86] takes $O(IK^{N-2}J^2)$ for the first mode. For the second mode, it takes $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(2),inter}$ (line 3 of Algorithm 6), $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(2),inter}blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$, and $O(IK^{N-2}J^2)$ to perform SVD of $\mathbf{Y}_{(2)}$. Then, it takes $O(IK^{N-2}J^2)$ to initialize the factor matrix of the second mode. For the remaining modes, it takes $O(IK^{N-2}J^2 + \sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ to compute the remaining $n$-mode products for all $n = 2,3,...,N$, and $O(J\sum_{k=0}^{N-3} K^{N-2-k}J^{2+k})$ to compute SVD for all $n = 3,4,...,N$. For all modes, the dominant term is $O(IK^{N-2}J^2)$ since $I > K > J$, thus we simplify the time complexity of the initialization phase as $O(IK^{N-2}J^2)$. □

#### 3.3.6.3 Proof of Lemma 3.3

*Proof.* For the first mode, the time complexity is $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(1),inter}$ and $\mathbf{Y}_{(1),inter}\ blkdiag(\{\mathbf{\Sigma}_l\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ in Equation (3.8), and $O(I\sum_{k=0}^{N-3}K^{N-2-k}J^{2+k})$ for computing the remaining $n$-mode products for all $n = 3,4,...,N$. We simplify $O(I\sum_{k=0}^{N-3}K^{N-2-k}J^{2+k})$ as $O(NIK^{N-2}J^2)$ since $J < K$. The computational time of the second mode is the same as that of the first mode. Before computing for remaining modes, it takes $O(IK^{N-2}J^2 + K^{N-2}J^3)$ to compute $\mathbf{Y}_{reuse}$ in line 15 of Algorithm 7. For mode-$i$, it takes $O(-K^{N-(i-1)}J^i + \sum_{k=0}^{N-3}K^{N-2-k}J^{3+k})$ to perform $n$-mode products for all $n = 3,4,i-1,i+1,...,N$. For core tensor, it takes $O(\sum_{k=0}^{N-3}K^{N-2-k}J^{3+k})$ to perform $n$-mode products for all $n = 3,4,...,N$. We simplify the complexity $O(-K^{N-(i-1)}J^i + \sum_{k=0}^{N-3}K^{N-2-k}J^{3+k})$ and $O(\sum_{k=0}^{N-3}K^{N-2-k}\ J^{3+k})$ to $O(NK^{N-2}J^3)$ since $K > J$. Therefore, the time complexity of one iteration in the iteration phase is $O(IK^{N-2}J^2 + NIK^{N-2}J^2 + IK^{N-2}J^2 + K^{N-2}J^3 + NK^{N-2}J^3)$. Without loss of generality, we express the time complexity of the iteration phase as $O(NIK^{N-2}J^2)$ since $I > K > J$. □

#### 3.3.6.4 Proof of Theorem 3.1

*Proof.* The total time complexity of D-Tucker is the summation of time complexities for the three phases: approximation, initialization, and iteration. By Lemmas 3.1 to 3.3, the time complexity is $O(I^2K^{N-2}J + MNIK^{N-2}J^2)$, which is simplified from $O(I^2K^{N-2}J + IK^{N-2}J^2 + MNIK^{N-2}J^2)$ without loss of generality. □

#### 3.3.6.5 Proof of Theorem 3.2

*Proof.* In the initialization phase, initializing for the first two modes requires $O(IK^{N-2}J)$ space to deal with $\left[\mathbf{U}_1\mathbf{\Sigma}_1;\cdots;\mathbf{U}_l\mathbf{\Sigma}_l\right]$, mode-2 matricization matrix $\mathbf{Y}_{(2)}$, and related tensors in lines 1 to 7 of Algorithm 6. Initializing for the remaining modes requires

$O(K^{N-2}J^2)$ to store $\mathbf{Y}_{(\mathbf{i})}$, $\mathbf{Y}_{reuse}$, $\mathcal{Y}_{reuse}$, and related tensors in lines 8 to 17 of Algorithm 6.

The iteration phase requires $O(IK^{N-2}J)$ space for matrices $\mathbf{Y}_{(2),inter}blkdiag(\{\mathbf{\Sigma}_i\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ and $\mathbf{Y}_{(1),inter}blkdiag(\{\mathbf{\Sigma}_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ in lines 4 and 9 of Algorithm 7. The dominant term for the remaining modes is $O(K^{N-2}J^2)$ to store $\mathcal{Y}_{reuse}$ in line 16 of Algorithm 7.

Considering $I > K > J$, the total space complexity is $O(IK^{N-2}J)$. □

## 3.4 Proposed Method for Online Tensors: D-TuckerO

### 3.4.1 Overview

We propose D-TuckerO, an efficient Tucker decomposition method in an online streaming setting. Our goal is to design D-TuckerO to efficiently update factor matrices and core tensor for a new incoming tensor slice. The main challenges that need to be tackled for an efficient Tucker decomposition method in an online streaming setting are as follows:

1. **Preventing the increase of costs over time.** How can we prevent increasing the computational cost and space cost as tensors continuously arrive over time?

2. **Accelerating updates.** How can we accelerate the update process for each incoming time slice?

We address the challenges with the following main ideas:

1. **Avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}_{old}^{(N)}$** enables D-TuckerO to update factor matrices and core tensor without increasing the costs where $\mathcal{X}_{old}$

and $\mathbf{A}^{(N)}_{old}$ are a pre-existing tensor and a pre-existing temporal factor matrix, respectively.

2. **Applying the approximation phase for an incoming time slice** accelerates the update procedure for factor matrices and core tensor.

As shown in Algorithm 8, D-TuckerO efficiently updates factor matrices when a new incoming tensor is given. We present an efficient update procedure for each new incoming tensor in Section 3.4.2, and then describe how to apply the approximation phase to the update procedure in Section 3.4.3. Lastly, we analyze the time and space complexities of D-TuckerO. For brevity, we set the last mode $N$ as the temporal mode when an $N$-order tensor repeatedly comes.

### 3.4.2 Efficient Update for Time Slice

Our goal is to update factor matrices and the core tensor for a new incoming tensor slice $\mathcal{X}_{new}$. D-TuckerO alternately updates factor matrices, and core tensor as in ALS algorithm; D-TuckerO updates the $n$-th factor matrix while fixing the other factor matrices and core tensor. We present how to update the temporal factor matrix $\mathbf{A}^{(N)}$ and then factor matrices of non-temporal modes.

**Temporal Mode.** Consider updating the temporal factor matrix $\mathbf{A}^{(N)}$. A naive approach is to update it by computing lines 4 and 5 in Tucker-ALS. However, dealing with an accumulated tensor $\mathcal{X}$ is impractical since the size of the tensor $\mathcal{X}$ increases over time. To efficiently update the factor without dealing with the accumulated tensor, we only update a part of the temporal factor matrix, i.e., $\mathbf{A}^{(N)}_{inc}$, corresponding to $t_{new}$. Lemma 3.4 describe an update rule for $\mathbf{A}^{(N)}_{inc}$.

**Lemma 3.4** (Update rule for temporal mode)**.** *When fixing all non-temporal factor*

**Algorithm 8:** Update phase of D-TuckerO

**Input:** a time slice $\mathcal{X}_{new} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times t_{new}}$, a pre-existing set $\mathcal{A}$ of factor matrix $\mathbf{A}_{old}^{(n)}$ $(n = 1, \ldots, N)$, and core tensor $\mathcal{G}_{old}$, a set of $\mathbf{P}_{old}^{(n)}$, $\mathbf{Q}_{old}^{(n)}$ for $n = 1, \ldots, N-1$, $\mathbf{P}_{old}^{(N+1)}$, and $\mathbf{Q}_{old}^{(N+1)}$

**Output:** updated factor matrices $\mathbf{A}_{new}^{(n)}$ $(n = 1, \ldots, N)$ and core tensor $\mathcal{G}_{new}$

**Parameters:** Rank $J_i$ $(i = 1, \ldots, N)$

1: obtain SVD results $\mathbf{U}_l$, $\mathbf{\Sigma}_l$, and $\mathbf{V}_l$ ( $l = 1, 2, \ldots, L$) of $\mathcal{X}_{new}$ using the approximation phase.
2: obtain $\mathbf{A}_{new}^{(N)}$ by computing Equation (3.9) with the SVD results of $\mathcal{X}_{new}$
3: **for** $n \leftarrow 1$ to $N-1$ **do**
4: obtain $\mathbf{A}_{new}^{(n)}$ by computing Equation (3.10) with the SVD results of $\mathcal{X}_{new}$
5: update $\mathbf{P}_{old}^{(n)} \leftarrow \mathbf{P}_{old}^{(n)} + \mathbf{P}_{new}^{(n)}$ by Equation (3.14)
6: update $\mathbf{Q}_{old}^{(n)} \leftarrow \mathbf{Q}_{old}^{(n)} + \mathbf{Q}_{new}^{(n)}$ by Equation (3.18)
7: **end for**
8: obtain $\mathcal{G}_{new}$ by computing Equation (3.19) with the SVD results of $\mathcal{X}_{new}$
9: update $\mathbf{P}_{old}^{(N+1)} \leftarrow \mathbf{P}_{old}^{(N+1)} + \mathbf{P}_{new}^{(N+1)}$ by Equation (3.20)
10: update $\mathbf{Q}_{old}^{(N+1)} \leftarrow \mathbf{Q}_{old}^{(N+1)} + \mathbf{Q}_{new}^{(N+1)}$ by Equation (3.21)

*matrices,* $\mathbf{A}_{inc}^{(N)}$ *is updated as follows:*

$$\mathbf{A}_{inc}^{(N)} \leftarrow \mathbf{X}_{(N),new} \left( \otimes_{k=1}^{N-1} \left( \mathbf{A}^{(k)T} \right)^{\dagger} \right) \mathbf{G}_{(N)}^{\dagger} \tag{3.9}$$

*where* $\dagger$ *indicates a pseudo-inverse of a matrix, and* $(\otimes_{k=1}^{N-1} \left( \mathbf{A}^{(k)T} \right)^{\dagger})$ *indicates the entire Kronecker product of* $\left( \mathbf{A}^{(k)T} \right)^{\dagger}$ *for* $k = N-1, N-2, \ldots, 2, 1$. □

*Proof.* See the proof in Section 3.4.5.1. □

Since $\mathbf{A}_{old}^{(N)}$ is already computed at the previous step, we only compute $\mathbf{A}_{inc}^{(N)}$ using $\mathcal{X}_{new}$, $\mathbf{G}_{(N)}$, and $\mathbf{A}^{(n)}$ for $n = 1, 2, \ldots, N-1$. In updating the temporal factor matrix $\mathbf{A}^{(N)}$, we exploit $\mathbf{G}_{(N),old}$ and $\mathbf{A}_{old}^{(n)}$ for $n = 1, 2, \ldots, N-1$ to compute $\mathbf{G}_{(N)}$ and $(\otimes_{k=1}^{N-1} \left( \mathbf{A}^{(k)T} \right)^{\dagger})$, respectively. In Equation (3.9), we compute 1) $\left( \mathbf{A}^{(k)T} \right)^{\dagger}$ for $k = 1, \ldots, N-1$, 2) the Kronecker product, and 3) matrix multiplication between $\mathbf{X}_{(N),new}$, the result of the Kronecker product, and $\mathbf{G}_{(N),old}^{\dagger}$ in Equation (3.9).

**Non-temporal Modes.** Our goal is to update $\mathbf{A}^{(n)}$ when a new incoming tensor $\mathcal{X}_{new}$ is given. By avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}_{old}^{(N)}$ whose size increases over time, we efficiently update $\mathbf{A}^{(n)}$. We first introduce an update rule for $\mathbf{A}^{(n)}$, and then provide details on efficiently computing $\mathbf{A}^{(n)}$ based on the rule.

**Lemma 3.5** (Update rule for non-temporal mode)**.** *When fixing $\mathbf{A}^{(k)}$ for $k = 1, \ldots, n-1, n+1, \ldots, N$, $\mathbf{A}_{new}^{(n)}$ is updated as follows:*

$$\mathbf{A}_{new}^{(n)} \leftarrow \mathbf{P}^{(n)} \left(\mathbf{Q}^{(n)}\right)^{-1} \tag{3.10}$$

*where $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ are equal to $\mathbf{X}_{(n)}(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})\mathbf{G}_{(n)}^{T}$ and $\left(\mathbf{G}_{(n)}(\otimes_{k\neq n}^{N}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}_{(n)}^{T}\right)$, respectively.* □

*Proof.* See the proof in Section 3.4.5.2. □

To update $\mathbf{A}^{(n)}$, we compute $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ in Equation (3.10). However, a naive computation for Equation (3.10) is impractical since the size of $\mathbf{X}_{(n)}$ and $\mathbf{A}^{(N)}$ increases over time. To achieve the efficiency, we avoid explicit computations with $\mathbf{X}_{(n),old}$ and $\mathbf{A}_{old}^{(N)}$ decoupled from $\mathbf{X}_{(n)}$ and $\mathbf{A}^{(N)}$, respectively, in $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$.

We now describe details on efficient computations of $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$. Given $\mathbf{P}^{(n)}$, we divide it into $\mathbf{P}_{old}^{(n)}$ and $\mathbf{P}_{new}^{(n)}$ where $\mathbf{P}_{old}^{(n)}$ and $\mathbf{P}_{new}^{(n)}$ are equal to Equations (3.12)

and (3.13), respectively.

$$\mathbf{P}^{(n)} = \begin{bmatrix}\mathbf{X}_{(n),old} & \mathbf{X}_{(n),new}\end{bmatrix} \times \left( \begin{bmatrix}\mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc}\end{bmatrix} \otimes (\otimes_{k\neq n}^{N-1} \mathbf{A}^{(k)}) \right) \mathbf{G}^T_{(n)} \tag{3.11}$$

$$= \left( \mathbf{X}_{(n),old} (\mathbf{A}^{(N)}_{old} \otimes (\otimes_{k\neq n}^{N-1} \mathbf{A}^{(k)})) \mathbf{G}^T_{(n)} \right) \tag{3.12}$$

$$+ \left( \mathbf{X}_{(n),new} (\mathbf{A}^{(N)}_{inc} \otimes (\otimes_{k\neq n}^{N-1} \mathbf{A}^{(k)})) \mathbf{G}^T_{(n)} \right) \tag{3.13}$$

$$= \mathbf{P}^{(n)}_{old} + \mathbf{P}^{(n)}_{new} \tag{3.14}$$

We only compute $\mathbf{P}^{(n)}_{new}$ for Equation (3.14) as $\mathbf{P}^{(n)}_{old}$ is computed and stored at the previous step. $\mathbf{P}^{(n)}$ is used as $\mathbf{P}^{(n)}_{old}$ at the next step.

Next, we efficiently compute $\mathbf{Q}^{(n)}$; we divide $\mathbf{Q}^{(n)}$ into $\mathbf{Q}^{(n)}_{old}$ and $\mathbf{Q}^{(n)}_{new}$ which are equal to Equations (3.16) and (3.17), respectively.

$$\mathbf{Q}^{(n)} = \mathbf{G}_{(n)} \left( \begin{bmatrix}\mathbf{A}^{(N)T}_{old} & \mathbf{A}^{(N)T}_{inc}\end{bmatrix} \begin{bmatrix}\mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc}\end{bmatrix} \right) \otimes \left( \otimes_{k\neq n}^{N-1} (\mathbf{A}^{(k)T} \mathbf{A}^{(k)}) \right) \mathbf{G}^T_{(n)} \tag{3.15}$$

$$= \mathbf{G}_{(n)} \left( (\mathbf{A}^{(N)T}_{old} \mathbf{A}^{(N)}_{old}) \otimes (\otimes_{k\neq n}^{N-1} (\mathbf{A}^{(k)T} \mathbf{A}^{(k)})) \right) \mathbf{G}^T_{(n)} \tag{3.16}$$

$$+ \mathbf{G}_{(n)} \left( (\mathbf{A}^{(N)T}_{inc} \mathbf{A}^{(N)}_{inc}) (\otimes_{k\neq n}^{N-1} (\mathbf{A}^{(k)T} \mathbf{A}^{(k)})) \right) \mathbf{G}^T_{(n)} \tag{3.17}$$

$$= \mathbf{Q}^{(n)}_{old} + \mathbf{Q}^{(n)}_{new} \tag{3.18}$$

Similar to $\mathbf{P}^{(n)}$, $\mathbf{Q}^{(n)}_{old}$ is computed and stored at the previous step. We only compute $\mathbf{Q}^{(n)}_{new}$ for Equation (3.18). $\mathbf{Q}^{(n)}$ is also used as $\mathbf{Q}^{(n)}_{old}$ at the next step.

**Core tensor.** After updating factor matrices, we update the core tensor with Lemma 3.6. By avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}^{(N)}_{old}$, we efficiently update $\mathcal{G}$. We first derive an equation for updating the core tensor, and then describe how to efficiently update it.

**Lemma 3.6** (Update rule for core tensor)**.** *When fixing all factor matrices, we update the core tensor with the following equation:*

$$\mathbf{G}_{(N)} = \left(\mathbf{Q}^{(N+1)}\right)^{-1} \mathbf{P}^{(N+1)} \tag{3.19}$$

*where* $\mathbf{P}^{(N+1)}$ *and* $\mathbf{Q}^{(N+1)}$ *are equal to* $\mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1})$ *and* $\left(\mathbf{A}^{(N)T}\mathbf{A}^{(N)}\right)$*, respectively. Note that* $(N+1)$ *in* $\mathbf{P}^{(N+1)}$ *and* $\mathbf{Q}^{(N+1)}$ *corresponds to the core tensor.* □

*Proof.* See the proof in Section 3.4.5.3. □

A naive computation for Equation (3.19) is expensive due to $\mathcal{X}$ and $\mathbf{A}^{(N)}$ corresponding to $t_{total}$. Therefore, we precisely divide $\mathbf{P}^{(N+1)}$ and $\mathbf{Q}^{(N+1)}$ to avoid computing the terms related to $\mathcal{X}_{old}$ and $\mathbf{A}_{old}^{(N)}$. $\mathbf{P}^{(N+1)}$ is divided as follows:

$$\begin{aligned}
&\mathbf{P}^{(N+1)} \\
&= \mathbf{A}^{(N)T}\mathbf{X}_{(N)}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) = \left(\begin{bmatrix}\mathbf{A}_{old}^{(N)T} & \mathbf{A}_{inc}^{(N)T}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{(N),old} \\ \mathbf{X}_{(N),new}\end{bmatrix}\right)\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) \\
&= \mathbf{A}_{old}^{(N)T}\mathbf{X}_{(N),old}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) + \mathbf{A}_{inc}^{(N)T}\mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) \\
&= \mathbf{P}_{old}^{(N+1)} + \mathbf{P}_{new}^{(N+1)}
\end{aligned} \tag{3.20}$$

Similar to updating the factor matrices of the non-temporal modes, we only compute $\mathbf{P}_{new}^{(N+1)}$ for updating the core tensor since $\mathbf{P}_{old}^{(N+1)}$ is already computed at the previous step.

Next, we divide $\mathbf{Q}^{(N+1)}$ into $\mathbf{Q}_{old}^{(N+1)}$ and $\mathbf{Q}_{new}^{(N+1)}$.

$$\mathbf{Q}^{(N+1)} = \mathbf{A}^{(N)T}\mathbf{A}^{(N)} = \begin{bmatrix} \mathbf{A}_{old}^{(N)T} & \mathbf{A}_{inc}^{(N)T} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{old}^{(N)} \\ \mathbf{A}_{inc}^{(N)} \end{bmatrix} = \left(\mathbf{A}_{old}^{(N)T}\mathbf{A}_{old}^{(N)}\right) + \left(\mathbf{A}_{inc}^{(N)T}\mathbf{A}_{inc}^{(N)}\right)$$
$$= \mathbf{Q}_{old}^{(N+1)} + \mathbf{Q}_{new}^{(N+1)} \tag{3.21}$$

Then, we compute $\mathbf{Q}_{new}^{(N+1)}$, and obtain $\mathbf{Q}^{(N+1)}$; note that $\mathbf{Q}_{old}^{(N+1)}$ is already computed at the previous step.

### 3.4.3 Applying Approximation Phase

The objective of applying the approximation phase is to accelerate the update process for each incoming time slice. The main ideas are to 1) approximate a time slice by performing randomized SVD of each sliced matrix of a time slice and 2) update factor matrices and a core tensor with the SVD results of the time slice instead of the time slice $\mathcal{X}_{new}$. We accelerate computing Equation (3.9) for the temporal mode, the computation of $\mathbf{P}^{(n)}$ for non-temporal modes, and $\mathbf{P}^{(N+1)}$ for the core tensor.

**Temporal Mode.** To obtain the factor matrix $\mathbf{A}_{inc}^{(N)}$ of the temporal mode, we first apply the approximation phase for a new incoming tensor $\mathcal{X}_{new}$ and then efficiently compute Equation (3.9). With the temporal mode fixed to the last mode, we assume that dimensionalities of an incoming tensor $\mathcal{X}_{new} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times T}$ are sorted in descending order.

To apply the approximation phase to Equation (3.9), we start from re-expressing the term $\mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right)$ in tensor form. Referring to Equation (2.5), we can

rewrite the term as follows:

$$\mathcal{X}_{new} \times_1 \left(\mathbf{A}^{(1)}\right)^{\dagger} \cdots \times_{N-1} \left(\mathbf{A}^{(N-1)}\right)^{\dagger}$$

Since the above equation has the same form as line 4 of Algorithm 1 for the $N$-th mode, computing it is the same as computing the $N$-th factor matrix in the iteration phase of D-Tucker. D-TuckerO first performs randomized SVD of each sliced matrix of a time slice $\mathcal{X}_{new}$ where the size of a sliced matrix is $I_1 \times I_2$. Then, we compute the term $\mathcal{X}_{new} \times_1 \left(\mathbf{A}^{(1)}\right)^{\dagger} \times_2 \left(\mathbf{A}^{(2)}\right)^{\dagger}$. The mode-1 matricization of the term is given by the following equation.

$$\mathbf{Y}_{inter} = \left(\left(\mathbf{A}^{(1)}\right)^{\dagger} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]\right) \times blkdiag\left(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}} \left(\mathbf{A}^{(2)T}\right)^{\dagger}\}_{l=1}^{L}\right) \tag{3.22}$$

We compute $\left(\mathbf{A}^{(1)}\right)^{\dagger} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$ and $blkdiag(\{\mathbf{\Sigma}_l \mathbf{V}_l^{\mathrm{T}} \left(\mathbf{A}^{(2)T}\right)^{\dagger}\}_{l=1}^{L})$ , respectively; then, $\mathbf{Y}_{inter}$ is obtained by multiplying the two results. After that, we perform $n$-mode products between $\mathcal{Y}_{inter}$ and $\left(\mathbf{A}^{(n)}\right)^{\dagger}$ for $n = 3, 4, ..., N-1$. Lastly, $\mathbf{A}_{inc}^{(N)}$ is updated by multiplying the mode-$N$ matricization of the result of the $n$-mode products and $\mathbf{G}_{(N)}^{\dagger}$.

**Non-temporal Modes.** For a mode $n$ except for $N$, the goal is to efficiently compute $\mathbf{P}_{new}^{(n)} = (\mathbf{X}_{(n),new} \left(\mathbf{A}_{inc}^{(N)} \otimes (\otimes_{k \neq n}^{N-1} \mathbf{A}^{(k)})\right) \mathbf{G}_{(n)}^{T})$ for updating the $n$-th factor matrix. Due to the expensive computations with $\mathcal{X}_{new}$ as in Tucker-ALS, we apply the approximation phase to reduce the computational cost of computing $\mathbf{P}_{new}^{(n)}$. We perform randomized SVD of each sliced matrix of a new incoming time slice $\mathcal{X}_{new}$ where the size of a sliced matrix is $I_1 \times I_2$, and then compute $\mathbf{P}_{new}^{(n)}$ using the SVD results.

To obtain $\mathbf{P}_{new}^{(n)}$, we compute $(\mathbf{X}_{(n),new} \left(\mathbf{A}_{inc}^{(N)} \otimes (\otimes_{k \neq n}^{N-1} \mathbf{A}^{(k)})\right)$, and then multiply it with $\mathbf{G}_{(n)}^{T}$. Before applying the approximation phase, we re-express $(\mathbf{X}_{(n),new} \left(\mathbf{A}_{inc}^{(N)} \otimes\right.$

$(\otimes_{k\neq n}^{N-1}\mathbf{A}^{(k)}))$ in tensor form as follows:

$$\mathcal{X}_{new} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}_{inc}^{(N)T} \tag{3.23}$$

Since the above equation has the same form as line 4 of Algorithm 1 for the $n$-th mode, computing it is the same as computing the $n$-th factor matrix in the iteration phase of D-Tucker. By using the SVD results of each sliced matrix of the time slice $\mathcal{X}_{new}$, we compute Equation (3.23) in the same way as computing an $n$-th factor matrix in the iteration phase of D-Tucker. Then, we obtain $\mathbf{P}_{new}^{(n)}$ by performing matrix multiplication between the mode-$n$ matricized version of the result of Equation (3.23) and $\mathbf{G}_{(n)}^T$. After that, we update the $n$-th factor matrix using $\mathbf{P}_{new}^{(n)}$.

**Core tensor.** To efficiently update the core tensor $\mathcal{G}$, we focus on accelerating the computation for the matrix $\mathbf{P}_{new}^{(N+1)}$ since the matrices $\mathbf{P}_{old}^{(N+1)}$ and $\mathbf{Q}_{old}^{(N+1)}$ are already computed and the computational cost of $\mathbf{Q}_{new}^{(N+1)}$ is relatively low. For $\mathbf{P}_{new}^{(N+1)} = \mathbf{A}_{inc}^{(N)T}\mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right)$, directly using $\mathbf{X}_{(N),new}$ is inefficient, so we apply the approximation phase. We first re-express $\mathbf{P}_{new}^{(N+1)}$ in tensor form:

$$\mathcal{X}_{new} \times_1 \mathbf{A}^{(1)\dagger} \cdots \times_{N-1} \mathbf{A}^{(N-1)\dagger} \times_N \mathbf{A}_{inc}^{(N)T} \tag{3.24}$$

$\mathbf{P}_{new}^{(N+1)}$ is obtained by computing the above equation in the following order: computing 1) Equation (3.22) for $\mathcal{X}_{new} \times_1 \left(\mathbf{A}^{(1)}\right)^{\dagger} \times_2 \left(\mathbf{A}^{(2)}\right)^{\dagger}$, 2) $n$-mode products with $\mathbf{A}^{(n)\dagger}$ for $n = 3, \ldots, N-1$, and 3) $n$-mode product with $\mathbf{A}_{inc}^{(N)T}$. Note that we use the SVD results of $\mathcal{X}_{new}$ in Equation (3.22), thereby we reduce the computational cost to update the core tensor compared to using $\mathcal{X}_{new}$. After that, we compute Equation (3.19) using $\mathbf{P}_{new}^{(N+1)}$.

### 3.4.4 Theoretical Analysis

**Theorem 3.3.** *Given a time slice $\mathcal{X}_{new}$ of size $I^2 \times K^{N-3} \times T_{new}$, the total time complexity of D-TuckerO to update factor matrices and core tensor is $O(I^2K^{N-3}T_{new} + NIK^{N-3}T_{new}J^2)$ where $N$ is the order of a given tensor, $I$ is the largest dimensionality, $K$ is the remaining dimensionality, and $J$ is the rank.*

*Proof.* See the proof in Section 3.4.5.4. □

**Theorem 3.4.** *D-TuckerO requires $O(IK^{N-3}T_{new}J)$ space for updating factor matrices when a new incoming tensor $\mathcal{X}$ of the size $I_1 \times I_2 \times K^{N-3} \times T_{new}$ is given.*

*Proof.* See the proof in Section 3.4.5.5. □

### 3.4.5 Proofs of Lemmas and Theorems

We provide proofs for Lemmas and Theorems described in Section 3.4.

#### 3.4.5.1 Proof of Lemma 3.4

*Proof.* The following equation represents the mode-$N$ matricized version of Equation (2.4) by replacing $\mathcal{X}$ with $\mathcal{X}_{old}$ and $\mathcal{X}_{new}$:

$$\mathbf{X}_{(N)} = \begin{bmatrix} \mathbf{X}_{(N),old} \\ \mathbf{X}_{(N),new} \end{bmatrix} \approx \begin{bmatrix} \mathbf{A}^{(N)}_{old}\mathbf{G}_{(N)}(\otimes^{N-1}\mathbf{A}^{(n)T}) \\ \mathbf{A}^{(N)}_{inc}\mathbf{G}_{(N)}(\otimes^{N-1}\mathbf{A}^{(n)T}) \end{bmatrix}$$

where $\mathbf{X}_{(N)}$ is the mode-$N$ matricized matrix of an accumulated tensor $\mathcal{X}$, and $\mathbf{X}_{(N),old}$ and $\mathbf{X}_{(N),new}$ are the mode-$N$ matricization of a pre-existing tensor $\mathcal{X}_{old}$ and a new incoming tensor slice $\mathcal{X}_{new}$, respectively. By fixing the factor matrix $\mathbf{A}^{(n)}$ for $n =$

$1, 2, ..., N-1$, we update the factor matrix $\mathbf{A}^{(N)}$ of the temporal mode as follows:

$$\begin{bmatrix} \mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_{(N),old} \left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} \\ \mathbf{X}_{(N),new} \left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} \end{bmatrix}$$

By adapting the properties, $(\mathbf{AB})^{\dagger} = \mathbf{B}^{\dagger}\mathbf{A}^{\dagger}$ and $(\mathbf{C} \otimes \mathbf{D})^{\dagger} = \mathbf{C}^{\dagger} \otimes \mathbf{D}^{\dagger}$ to the above equation, we obtain the following equation:

$$\mathbf{A}^{(N)}_{inc} \leftarrow \mathbf{X}_{(N),new} \left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} = \mathbf{X}_{(N),new} \left(\otimes_{k=1}^{N-1} \left(\mathbf{A}^{(k)} \left(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right)^{-1}\right)\right) \mathbf{G}^{\dagger}_{(N)}$$

□

#### 3.4.5.2 Proof of Lemma 3.5

*Proof.* For mode $n$, we formulate the loss function $L_{(n)}$ as follows:

$$L_{(n)} = \frac{1}{2}\|\mathbf{X}_{(n)} - \mathbf{A}^{(n)}\mathbf{G}_{(n)}(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})^{T}\|^2 \tag{3.25}$$

where $(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})$ indicates Kronecker products of $\mathbf{A}^{(k)}$ for $k = N, N-1, ..., n+1, n-1, ..., 1$. When fixing $\mathbf{A}^{(k)}$ for $k = 1, ..., n-1, n+1, ..., N$, the partial derivative of the function $L_{(n)}$ with respect to $\mathbf{A}^{(n)}$ is as follows:

$$\frac{\partial L_{(n)}}{\partial \mathbf{A}^{(n)}} = -\mathbf{X}_{(n)}(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)} + \mathbf{A}^{(n)}\mathbf{G}_{(n)}(\otimes_{k\neq n}^{N}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}$$

To minimize $\frac{\partial L_{(n)}}{\partial \mathbf{A}^{(n)}}$, we set it to zero and compute $\mathbf{A}^{(n)}$ as follows:

$$\begin{aligned} \mathbf{A}^{(n)} &= \mathbf{X}_{(n)}(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)} \times \left(\mathbf{G}_{(n)}(\otimes_{k\neq n}^{N}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right)^{-1} \\ &= \mathbf{P}^{(n)}\left(\mathbf{Q}^{(n)}\right)^{-1} \end{aligned}$$

where $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ are equal to $\mathbf{X}_{(n)}(\otimes_{k\neq n}^{N}\mathbf{A}^{(k)})\mathbf{G}_{(n)}^{T}$ and $\left(\mathbf{G}_{(n)}(\otimes_{k\neq n}^{N}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}_{(n)}^{T}\right)$, respectively. □

#### 3.4.5.3 Proof of Lemma 3.6

*Proof.* To update core tensor, we start from the following equation:

$$\mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)} = \mathcal{X}$$

For each mode $n$, we multiply $\mathbf{A}^{(n)\dagger} = (\mathbf{A}^{(n)T}\mathbf{A}^{(n)})^{-1}\mathbf{A}^{(1)T}$ on both left and right terms. Then, we obtain the core tensor by computing the following equation:

$$\mathcal{G} = \mathcal{X} \times_1 \mathbf{A}^{(1)\dagger} \cdots \times_N \mathbf{A}^{(N)\dagger}$$

For brevity, we compute the core tensor with mode-$N$ matricization. We carefully decouple the computations for $\mathbf{A}_{old}^{(N)}$ and $\mathbf{A}_{new}^{(N)}$. It leads to avoiding explicit computations related to $\mathbf{A}_{old}^{(N)}$ and $\mathbf{X}_{(N),new}$.

$$\begin{aligned}\mathbf{G}_{(N)} &= (\mathbf{A}^{(N)T}\mathbf{A}^{(N)})^{-1} \times \mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1}) \\ &= \left(\mathbf{Q}^{(N+1)}\right)^{-1}\mathbf{P}^{(N+1)}\end{aligned} \tag{3.26}$$

where $\mathbf{P}^{(N+1)}$ and $\mathbf{Q}^{(N+1)}$ are equal to $\mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1})$ and $\left(\mathbf{A}^{(N)T}\mathbf{A}^{(N)}\right)$, respectively. □

#### 3.4.5.4 Proof of Theorem 3.3

*Proof.* There are two dominant terms in the time complexity of D-TuckerO: 1) the approximation of a new time slice $\mathcal{X}_{new}$, and 2) $n$-mode products between the approx-

imation result and factor matrices $\mathbf{A}^{(k)}$ (or $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$). Approximating a new time slice $\mathcal{X}_{new}$ require $O(I^2K^{N-3}T_{new})$ by Lemma 3.1. In addition, the time complexity of updating all factor matrices is $O(NIK^{N-3}\ T_{new}J^2)$ since updating them includes $n$-mode products between the approximation of $\mathcal{X}_{new}$ and $\mathbf{A}^{(k)}$ (or $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$) whose complexity is analyzed in Lemma 3.3. Therefore, the total time complexity of D-TuckerO for each time slice is $O(I^2K^{N-3}T_{new} + NIK^{N-3}T_{new}J^2)$. □

#### 3.4.5.5 Proof of Theorem 3.4

*Proof.* The space of of D-TuckerO is determined by storing $\mathbf{P}_{(n),old}$ and $\mathbf{Q}_{(n),old}$, and computing $\mathbf{P}_{(n),new}$ and $\mathbf{Q}_{(n),new}$. Space costs of $\mathbf{P}_{(n),old}$ and $\mathbf{Q}_{(n),old}$ are $O((I_1 + I_2 + (N-3)K)J)$ and $O((N-1)J^2)$ for all $n = 1, ..., N-1$, respectively. We perform $n$-mode product between $\mathcal{G}$ of the size $J^N$ and $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for $\mathbf{Q}_{(n),new}$ of the size $J \times J$. Since the intermediate data are always smaller than $\mathcal{G}$, the space cost of $\mathbf{Q}_{(n),new}$ is $O(J^N)$ which is the size of $\mathcal{G}$. Additionally, the space cost of $\mathbf{P}_{(n),new}$ is $O(IK^{N-3}T_{new}J)$ since the size of the SVD results of $\mathcal{X}_{new}$ is $O(IK^{N-3}T_{new}J)$, and the size of intermediate data of $\mathbf{P}_{(n),new}$ is always smaller than $O(IK^{N-3}T_{new}J)$. The total space cost to update factor matrices and core tensor for $\mathcal{X}_{new}$ is $O((I_1 + I_2 + (N-3)K)J + (N-1)J^2 + J^N + IK^{N-3}T_{new}J)$. We simplify the space cost as $O(IK^{N-3}T_{new}J)$ since the dominant term is to compute $\mathbf{P}_{(n),new}$. □

## 3.5 Experiment

In this section, we experimentally evaluate the performance of D-Tucker and D-TuckerO. We answer the following questions:

- **Q1. Time cost and reconstruction error (Section 3.5.2).** How quickly does D-Tucker obtain factor matrices and core tensor compared to other competi-

tors, while having low reconstruction error?

- **Q2. Effectiveness of the initialization phase (Section 3.5.3).** How much does the initialization phase reduce the number of iterations in D-Tucker?
- **Q3. Efficiency of the iteration phase (Section 3.5.4).** How efficient is the iteration phase of D-Tucker compared to other methods?
- **Q4. Space cost (Section 3.5.5).** How much space does D-Tucker require to obtain factor matrices and core tensor compared to other competitors?
- **Q5. Scalability (Section 3.5.6).** How well does D-Tucker scale up with regard to dimensionality, rank, order, and a number of iterations?
- **Q6. Running time and error in online streaming setting (Section 3.5.7).** For each new incoming tensor, how efficiently does D-TuckerO update factor matrices and core tensor?
- **Q7. Size of a time slice in an online streaming setting (Section 3.5.8).** How efficiently does D-TuckerO handle an incoming tensor slice of various sizes?

### 3.5.1 Experimental Settings

We describe experimental settings for the datasets, competitors, and environments.

**Machine.** We use a workstation with a single CPU (Intel Xeon E5-2630 v4 @ 2.2GHz), and 512GB memory.

**Dataset.** For static Tucker decomposition, we use four real-world tensors in Table 3.3 for evaluating the performance. Brainq dataset[1] [91] contains fMRI information consisting of (word, voxel, person; measurement). Boats dataset[2] [92] contains

[1] http://www.cs.cmu.edu/afs/cs/project/theo-73/www/science2008/data.html
[2] http://changedetection.net/

grayscale videos in the form of (height, width, frame; value). Air quality dataset[3] contains air pollutant information in Korea, in the form of (timestamp in second, location, atmospheric pollutants; measurement). HSI dataset[4] [93] contains hyperspectral images of natural scenes in the form of (spatial dimension (x), spatial dimension (y), spectral dimension, scene index; value).

For online streaming decomposition, we use four real-world tensors described in Table 3.4. Stock dataset contains features of stocks over 200 days in South Korea. The features consist of (adjusted opening price / previous day's adjusted closing price), (adjusted highest price / previous day's adjusted closing price), (adjusted lowest price / previous day's adjusted closing price), and (adjusted closing price / previous day's adjusted closing price). FMA dataset[5] [94, 95] is a song dataset whose form is (song, frequency, time; value). Each song is represented as an image of a log-power spectrogram. Traffic dataset[6] [96] contains traffic volume measurements from $1,084$ sensors over 200 days, and each sensor yields 96 observations per day. Absorb dataset[7] is a 4-order tensor containing aerosol absorption; the form is (longitudes, latitudes, altitude, time; measurement). Note that the original values in this data are so small that we use a tensor multiplied by 10.

**Competitors.** We compare D-Tucker with static Tucker decomposition methods based on ALS approach. All the methods including D-Tucker are implemented in MATLAB (R2019b).

- **D-Tucker** [26]: we use randomized SVD [85] in the approximation phase using the implementation of Malik and Becker [38], standard SVD (*svds()* function in

[3]https://www.airkorea.or.kr
[4]https://personalpages.manchester.ac.uk/staff/d.h.foster/Hyperspectral_images_of_natural_scenes_04.html
[5]https://github.com/mdeff/fma
[6]https://github.com/florinsch/BigTrafficData
[7]https://www.earthsystemgrid.org/

Table 3.3: Description of real-world tensor datasets for evaluating the performance of static Tucker decomposition methods.

| Dataset | Order | Dimensionality | Rank |
|---|---|---|---|
| Brainq[1] [91] | 3 | $(360, 21764, 9)$ | $(10, 10, 5)$ |
| Boats[2] [92] | 3 | $(320, 240, 7000)$ | $(10, 10, 10)$ |
| Air Quality[3] | 3 | $(30648, 376, 6)$ | $(10, 10, 5)$ |
| HSI[4] [93] | 4 | $(1021, 1340, 33, 8)$ | $(10, 10, 10, 5)$ |

Table 3.4: Description of real-world tensor datasets for evaluating the performance of streaming Tucker decomposition methods.

| Dataset | Order | Dimensionality | Rank |
|---|---|---|---|
| Stock | 3 | $(3028, 4, 200)$ | $(10, 4, 10)$ |
| FMA[5] [94, 95] | 3 | $(7994, 1025, 200)$ | $(10, 10, 10)$ |
| Traffic[6] [96] | 3 | $(1084, 96, 200)$ | $(10, 10, 10)$ |
| Absorb[7] | 4 | $(192, 288, 30, 200)$ | $(10, 10, 10, 10)$ |

MATLAB) in the initialization and iteration phases, and Tensor Toolbox [97] for tensor operations such as $n$-mode product and matricization.

- **Tucker-ALS**: Tucker decomposition method based on ALS. We use the implementation in Tensor Toolbox [97].
- **MACH** [37]: Tucker decomposition method which samples entries of an input tensor and runs Tucker-ALS for the sampled tensor. We run Tucker-ALS in Tensor Toolbox [97] after sampling elements of a tensor.
- **Randomized Tucker Decomposition (RTD)** [36]: Tucker decomposition using a randomized algorithm. We use the Matlab code provided by authors.
- **Tucker-ts**, **Tucker-ttmts** [38]: Tucker-ts is a Tucker decomposition method using tensor sketch designed to approximate the solution of a large least-squares problem. Tucker-ttmts is a variant of Tucker-ts for better efficiency. We use the Matlab code[8] provided by authors.

[8] https://github.com/OsmanMalik/tucker-tensorsketch

We also compare D-TuckerO with the following streaming Tucker decomposition methods in an online streaming setting:

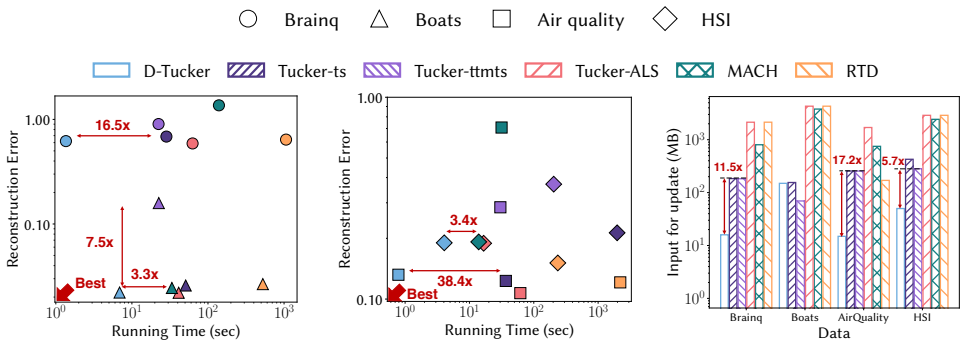
- **D-TuckerO**: We leverage Tensor Toolbox [97] for tensor operations such as $n$-mode product and matricization.
- **Tucker-ALS**: Tucker decomposition method based on ALS. We use the implementation in Tensor Toolbox [97].
- **Tucker-ts**, **Tucker-ttmts** [38]: Tucker-ts and Tucker-ttmts are easily adapted to online streaming settings.
- **DTA** (Dynamic Tensor Analysis): DTA finds factor matrices and core tensor to fit newly arrived tensors. We use the Matlab code[9] provided by the authors.
- **STA** (Streaming Tensor Analysis): STA is an approximation version of DTA that finds factor matrices and core tensor to fit newly arrived tensors. We use the Matlab code[9] provided by authors.

**Parameters.** We use the following parameters.

- **Number of threads:** we use a single thread.
- **Max number of iterations:** we set the maximum number of iterations to 50.
- **Rank:** the dimensionality $J_n$ of the $n$th mode of a core tensor is set to 10. We set it to 4 and 5, respectively, when the dimensionality is smaller than 5 and 10, respectively. We also set the rank $J$ of randomized SVD to 10 which is the same as the dimensionality $J_n$ of core tensor.
- **Tolerance:** the iteration stops when the variation of the error $\frac{\sqrt{\|\mathcal{X}\|_{\mathbf{F}}^2 - \|\mathcal{G}\|_{\mathbf{F}}^2}}{\|\mathcal{X}\|_{\mathbf{F}}}$ [29] is less than $\varepsilon = 10^{-4}$ except in Section 3.5.3 where we vary it.

We set other parameters of competitors based on their original papers. To compare

[9] http://www.cs.cmu.edu/~jimeng/code/tensorCode.zip




(a) Running time vs. error for Brainq and Boats datasets

(b) Running time vs. error for Air quality and HSI datasets

(c) Memory Usage

Figure 3.3: D-Tucker achieves the best performance in terms of error, running time, and memory usage. (a) (b) Comparison for the tradeoff between running time and error; D-Tucker is up to 38.4× faster than the second-fastest competitor while having a similar error. (c) Space cost of D-Tucker. D-Tucker initializes and updates factor matrices and core tensor by using up to 17.2× smaller space than competitors except for Boats dataset. Note that, for Boats dataset, D-Tucker requires 2× higher space than Tucker-ttmts which has 7.5× higher error than our method.

running time, we run each method 10 times for D-Tucker and D-TuckerO, and report the average.

**Reconstruction error.** In a static setting, we evaluate the accuracy in terms of reconstruction error defined as $\frac{\|\mathcal{X}-\hat{\mathcal{X}}\|_F^2}{\|\mathcal{X}\|_F^2}$ where $\mathcal{X}$ is an input tensor and $\hat{\mathcal{X}}$ is the reconstruction of the output of Tucker decomposition.

In an online streaming setting, we measure two kinds of errors, global and local reconstruction errors. The global reconstruction error is defined as $\sqrt{\frac{\sum_{i=1}^{T}\|\mathcal{X}_i-\hat{\mathcal{X}}_i\|_F^2}{\sum_{i=1}^{T}\|\mathcal{X}_i\|_F^2}}$ where $\mathcal{X}_i$ is a tensor obtained at time $i$ and $\hat{\mathcal{X}}_i$ is a reconstructed tensor from factor matrices and core tensor of D-TuckerO. The global error indicates how well the results of a Tucker decomposition method represent an accumulated tensor over time. The local reconstruction error is defined as $\sqrt{\frac{\|\mathcal{X}_{new}-\hat{\mathcal{X}}_{new}\|_F}{\|\mathcal{X}_{new}\|_F}}$. In contrast to the global error, the local error indicates how well the results of a Tucker decomposition method represent a new incoming tensor.

### 3.5.2 Time Cost and Reconstruction Error

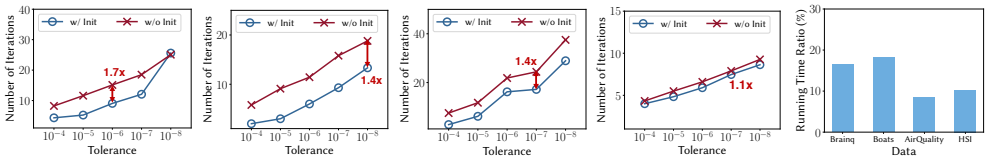
We measure the running time and the reconstruction error of D-Tucker and competitors. As shown in Figures 3.3(a) and 3.3(b), D-Tucker achieves the best trade-offs between the time and error, achieving up to 38.4× faster running time than Tucker-ts, Tucker-ttmts, and MACH with smaller or similar reconstruction errors. Tucker-ALS and RTD have smaller reconstruction errors for Air quality and HSI datasets, but they are at least 3.4× and 42× slower than D-Tucker, respectively.

### 3.5.3 Effectiveness of the Initialization Phase

We show that the initialization phase of D-Tucker provides a good starting point for the iteration step, by measuring the number of iterations in the iteration phase. We vary the error tolerance $\varepsilon$ in the iteration phase from $10^{-4}$ to $10^{-8}$. As shown in Figure 3.4, the number of iterations with the initialization phase is up to 1.7× smaller than that without the initialization phase. The initialization phase allows D-Tucker to reduce the total running time since the running time of the initialization phase is less than the reduction time of the iteration phase. Moreover, the average ratio of the initialization phase's running time to the total running time in D-Tucker does not exceed 20%. This indicates that the initialization phase of D-Tucker reduces the number of iterations significantly with little additional overhead on the total running time.

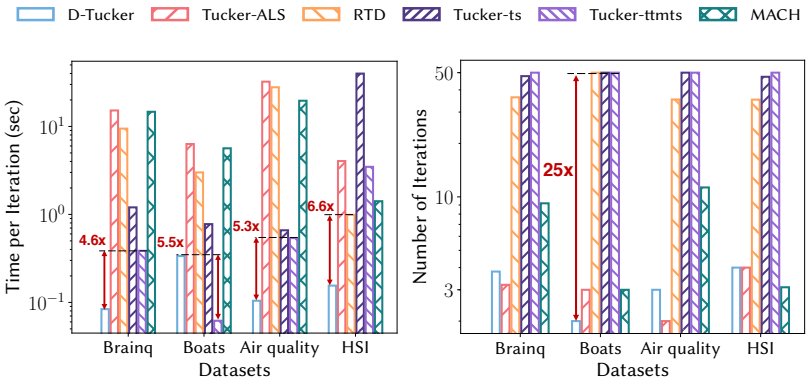
### 3.5.4 Efficiency of the Iteration Phase

We investigate the number of iterations and the running time per iterations. In Figure 3.5, For each iteration, D-Tucker is at least 4.6× faster than competitors on all datasets except for Boat dataset, and consumes a smaller number of iterations than




(a) Brainq (b) Boats (c) Air quality (d) HSI (e) Running time ratio of the initialization phase

Figure 3.4: The initialization phase of D-Tucker helps reduce the number of iterations and thus the total running time. (a-d) The number of iterations with the initialization phase is up to 1.7×, 1.4×, 1.4×, and 1.1× smaller than those without the initialization phase for Brainq, Boats, Air quality, and HSI datasets, respectively. (e) The average ratio of the running time in the initialization phase compared to the total running time does not exceed 20% for all the datasets.




(a) Time per iteration (b) Number of iterations

Figure 3.5: In the iteration phase, D-Tucker is the most efficient compared to competitors. (a) The running time of each iteration of D-Tucker is up to 6.6× faster than those of competitors except for the Boats dataset. For the Boats dataset, Tucker-ttmts achieves the fastest running time per iteration, but requires a much larger number of iterations, and has a much higher error than D-Tucker. (b) The number of iterations of D-Tucker is in general smaller than others; while there are cases D-Tucker requires more number of iterations, the difference is negligible considering the running time per iteration.

the competitors. Although Tucker-ttmts is faster than D-Tucker at each iteration, it requires a larger number of iterations than D-Tucker; hence, the total running time of D-Tucker is 4.5× longer than that of Tucker-ttmts at the iteration phase. For the number of iterations, Figure 3.5(b) shows that D-Tucker requires a smaller number of iterations than all the competitors except for Tucker-ALS on 3-order datasets; how-

ever, the difference is quite small considering the running time per iteration.

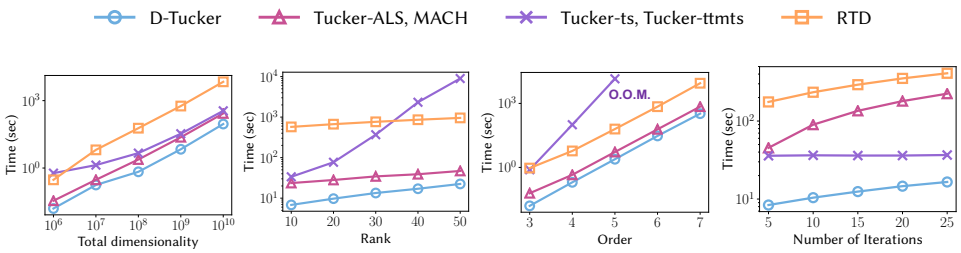
### 3.5.5 Space Cost

We investigate the memory requirements of D-Tucker and competitors for initializing and updating factor matrices and a core tensor. Figure 3.3(c) shows that D-Tucker requires up to $17.2\times$ smaller space than the second best methods Tucker-ts and Tucker-ttmts in terms of memory usage. For Boats dataset, Tucker-ts, and Tucker-ttmts require small space since this dataset has the following setting where the two methods operate well: 1) order $N$ and rank $J$ are very small, and 2) dimensionalities $I$ and $K$ are very large. Note that D-Tucker has $7.5\times$ less error than Tucker-ttmts while requiring $2.1\times$ more space than Tucker-ttmts.

### 3.5.6 Scalability

We investigate the scalability of D-Tucker and competitors with regard to dimensionality, target rank, order, and number of iterations in Figure 3.6. In sum, D-Tucker is the most scalable with the smallest running time. Since the time complexities of Tucker-ts and Tucker-ttmts are proportional to $J^N$, they are not scalable for the target rank, and order of an input tensor. RTD operates for all the given experimental settings, but RTD is much slower than D-Tucker. MACH and Tucker-ALS also operate for all the given experimental settings, but they are at least $2\times$ slower than D-Tucker. Furthermore, they become much slower than D-Tucker as the number of iterations increases (e.g., when setting smaller tolerance $\varepsilon$ or when converging slowly in real-world datasets). The details of scalability experiments are as follows.

**Dimensionality.** For investigating the scalability related to dimensionality, we generate synthetic 3-order tensors of true rank $J_{true} = 10$, while increasing the total




(a) Total dimensionality (b) Rank (c) Order (d) Number of Iterations

Figure 3.6: Scalability of D-Tucker compared to other Tucker decomposition methods. O.O.M.: out of memory. For clarity, we show 4 groups of methods having similar tendencies. Note that D-Tucker is the most scalable with the smallest running time: for all settings, D-Tucker is at least 2.1× faster than competitors. Tucker-ts and Tucker-ttmts have limited scalability with respect to the target rank and the order. RTD has good scalability for all aspects, but it is up to 76× slower than D-Tucker. MACH and Tucker-ALS are also scalable for all aspects, but they are at least 2× slower than D-Tucker. Furthermore, their performance gaps compared to D-Tucker become even worse when the number of iterations increases.

dimensionality $I_1 I_2 K_3$ from $10^6$ to $10^{10}$ (dimensionality list: $\{(10^2, 10^2, 10^2), (10^3, 10^2, 10^2), (10^3, 10^3, 10^2), (10^3, 10^3, 10^3), (10^4, 10^3, 10^3)\}$). As shown in Figure 3.6(a), D-Tucker is the fastest for various dimensionalities, and runs at least 2.7× faster than all competitors.

**Target rank.** For investigating the scalability related to target rank, we generate synthetic 3-order tensors of size $I_1 = I_2 = K_3 = 10^3$ and true rank $J_{true} = 10$, while varying the target rank from 10 to 50. As shown in Figure 3.6(b), D-Tucker is the fastest for various target ranks. Tucker-ts and Tucker-ttmts provide the worst scalabilities since their time complexities are proportional to $J^N$. The running times of all competitors except Tucker-ts and Tucker-ttmts scale with regard to target ranks, but they are at least 2.1× slower than D-Tucker.

**Order.** For investigating the scalability related to order $N$, we generate synthetic $N$-order tensors of true rank $J_{true} = 10$, while varying the order from 3 to 7. We set dimensionalities of synthetic tensors to $I_1 = 10^3$, $I_2 = 10^2$, and $K_i = 10$ for $i = 3, 4, \ldots, 7$.

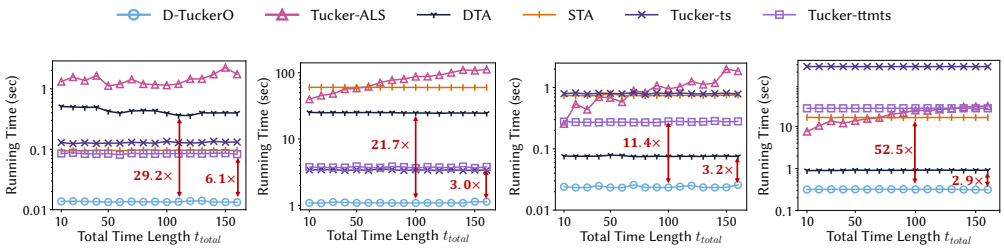
In Figure 3.6(c), D-Tucker is the fastest for various orders of input tensors. Since the time and memory complexities of Tucker-ts and Tucker-ttmts are proportional to $J^{2N}$, they are 5883× slower than D-Tucker, and cannot deal with 6 and 7-order tensors. Although all competitors except Tucker-ts and Tucker-ttmts can process higher order tensors, they are at least 2.1× slower than D-Tucker.

**Number of iterations.** We generate synthetic 3-order tensors of size $I_1 = I_2 = K_3 = 10^3$ with true rank $J_{true} = 10$. Then we evaluate the running time varying the number of iterations from 5 to 25. As shown in Figure 3.6(d), D-Tucker is the fastest for varying numbers of iterations. In addition, the running time of D-Tucker is not affected much by the number of iterations while those of all competitors except Tucker-ts and Tucker-ttmts are affected much by the number of iterations. Note that the running time of Tucker-ts and Tucker-ttmts are 3.6× slower than that of D-Tucker although those are less affected by the number of iterations than D-Tucker.

### 3.5.7 Streaming Setting

We compare D-TuckerO with streaming Tucker decomposition methods. We initially construct factor matrices and a core tensor using the first 20% of a whole tensor, and then measure the running time of updating a new incoming tensor at each time point. In addition, we set $t_{new}$ of each time slice to 10.

**Running Time.** As shown in Figure 3.7, we compare the running time of D-TuckerO with those of competitors. For the 3-order datasets, D-TuckerO is up to 6.1× faster than the second-fastest competitor Tucker-ttmts as shown in Figures 3.7(a) to 3.7(c). Also, D-TuckerO is at least 2.9× faster than the competitors for Absorb dataset which is a 4-order tensor. In addition, the running time of D-TuckerO does not increase over time since it is proportional to the size of a new incoming tensor,

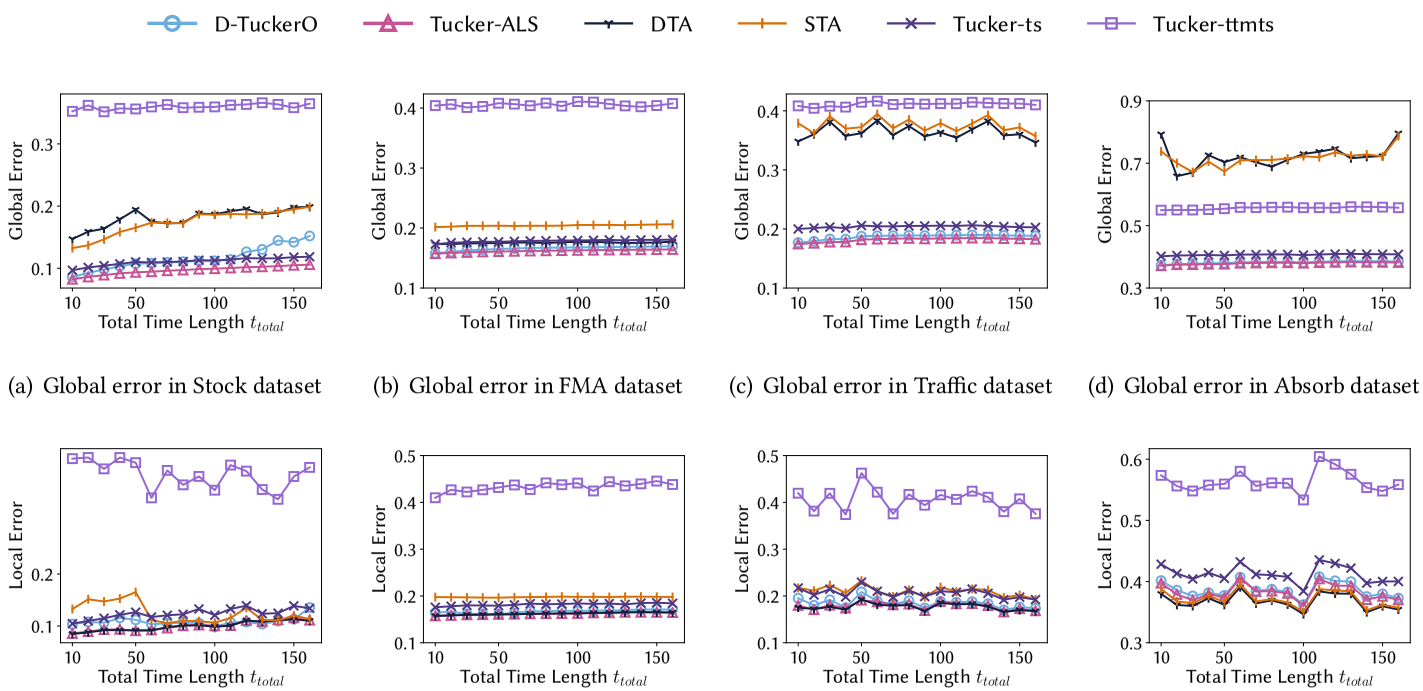



(a) Update time in Stock dataset (b) Update time in FMA dataset (c) Update time in Traffic dataset (d) Update time in Absorb dataset

Figure 3.7: Running time of D-TuckerO and competitors over time. D-TuckerO outperforms competitors when we compare the running time of updating factor matrices and core tensor for each new incoming tensor. D-TuckerO is up to $6.1\times$ faster than the second fastest method, and the running time does not increase over time.

not the accumulated tensor.

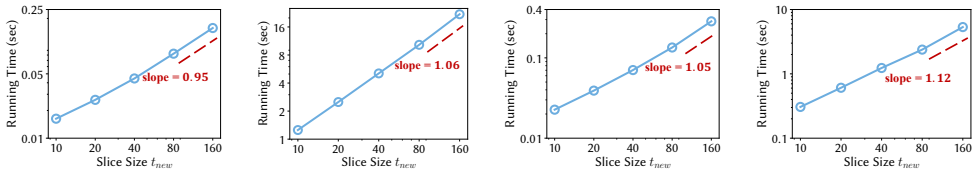
**Error.** We measure global and local reconstruction errors of D-TuckerO and competitors. Figures 3.8(a) to 3.8(d) show the results for global reconstruction errors, and Figures 3.8(e) to 3.8(h) show the results for local reconstruction errors. As shown in Figures 3.8(a) to 3.8(d), D-TuckerO has comparable global errors with Tucker-ALS which performs Tucker decomposition for accumulated tensors, while DTA and STA have higher global errors than D-TuckerO. These results indicate that updated results of D-TuckerO sufficiently contain global patterns of an accumulated tensor. As shown in Figures 3.8(e) to 3.8(h), the local errors of D-TuckerO are close to those of Tucker-ALS which is a static version of Tucker decomposition since updated results of D-TuckerO sufficiently contains information of a new incoming tensor. In addition, the approximation phase of D-TuckerO does not hurt accuracy much since a time slice of real-world datasets has a low-rank structure.




(a) Global error in Stock dataset (b) Global error in FMA dataset (c) Global error in Traffic dataset (d) Global error in Absorb dataset

(e) Local error in Stock dataset (f) Local error in FMA dataset (g) Local error in Traffic dataset (h) Local error in Absorb dataset

Figure 3.8: Global and local errors in an online streaming setting. D-TuckerO achieves comparable global and local errors with Tucker-ALS which is a static version of Tucker decomposition.




(a) Stock dataset (b) FMA dataset (c) Traffic dataset (d) Absorb dataset

Figure 3.9: We measure the running time of D-TuckerO, varying the size of a time slice. The running time of D-TuckerO increases near-linearly as the size of a time slice increases. Note that a slope equal to 1 indicates linear scalability.

### 3.5.8 Size of Time Slice

We evaluate the performance of D-TuckerO, varying the size $t_{new}$ of a time slice: 10, 20, 40, 80, and 160. Figure 3.9 shows that there are near-linear relationships between $t_{new}$ and the running time of D-TuckerO in an online streaming setting; for all the four datasets, the slopes are close to 1. This is because the running time of D-TuckerO is proportional to the size of a new incoming tensor.

## 3.6 Summary

We propose D-Tucker and D-TuckerO, efficient Tucker decomposition methods for large-scale dense tensors in static and online streaming settings. D-Tucker and D-TuckerO accelerate computing Tucker decomposition by approximating a given dense tensor, and carefully computing Tucker results from the approximated tensor. We show D-Tucker provides the fastest running time and the smallest memory usage. Furthermore, D-TuckerO is also the fastest method to update factor matrices and a core tensor for new incoming tensors. We also provide theoretical analysis for the time and space complexities of D-Tucker and D-TuckerO. Extensive experiments show that D-Tucker is up to 38.4× faster, and requires up to 17.2× less space than existing methods with little sacrifice in accuracy. D-Tucker is also scalable with re-

gard to dimensionality, rank, order, and the number of iterations. D-TuckerO is up to $6.1\times$ faster than existing methods running in an online streaming setting, while not increasing the running time over time.

# Chapter 4

# Efficient Tensor Decomposition in Irregular Tensors

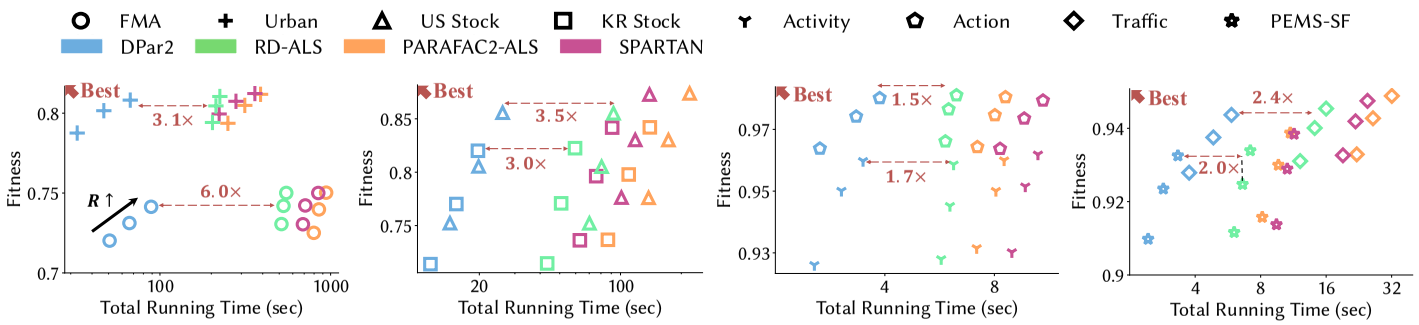
## 4.1 Motivation

*How can we efficiently analyze an irregular dense tensor?* Many real-world multi-dimensional arrays are represented as irregular dense tensors; an irregular tensor is a collection of matrices with different row lengths. For example, stock data can be represented as an irregular dense tensor; the listing period is different for each stock **(irregularity)**, and almost all of the entries of the tensor are observable during the listing period **(high density)**. The irregular tensor of stock data is the collection of the stock matrices whose row and column dimension corresponds to time and features (e.g., the opening price, the closing price, the trade volume, etc.), respectively. In addition to stock data, many real-world data including music song data and sound data are also represented as irregular dense tensors. Each song can be represented as a slice matrix (e.g., time-by-frequency matrix) whose rows correspond to the time dimension. Then, the collection of songs is represented as an irregular tensor consisting of slice matrices of songs each of whose time length is different. Sound data are represented similarly.

Tensor decomposition has attracted much attention from the data mining community to analyze tensors [98, 99, 25, 41, 16, 76, 78, 100, 11, 56]. Specifically, PARAFAC2 decomposition has been widely used for modeling irregular tensors in various appli-

cations including phenotype discovery [2, 3], trend analysis [101], and fault detection [102]. However, existing PARAFAC2 decomposition methods are not fast and scalable enough for irregular dense tensors. Perros et al. [2] improve the efficiency of handling irregular sparse tensors, by exploiting the sparsity patterns of a given irregular tensor. Many recent works [3, 66, 67, 68] adopt their idea to handle irregular sparse tensors. However, they are not applicable to irregular *dense* tensors that have no sparsity pattern. Although Cheng and Haardt [33] improve the efficiency of PARAFAC2 decomposition by preprocessing a given tensor, there is plenty of room for improvement in terms of computational costs. Moreover, there remains a need for fully employing multicore parallelism. The main challenge to successfully design a fast and scalable PARAFAC2 decomposition method is how to minimize the computational costs involved with an irregular dense tensor and the intermediate data generated in updating factor matrices.

In this work, we propose DPar2 (Dense PARAFAC2 decomposition), a fast and scalable PARAFAC2 decomposition method for irregular dense tensors. Based on the characteristics of real-world data, DPar2 compresses each slice matrix of a given irregular tensor using randomized Singular Value Decomposition (SVD). The small compressed results and our careful ordering of computations considerably reduce the computational costs and the intermediate data. In addition, DPar2 maximizes multi-core parallelism by considering the difference in sizes between slices. With these ideas, DPar2 achieves higher efficiency and scalability than existing PARAFAC2 decomposition methods on irregular dense tensors. Extensive experiments show that DPar2 outperforms the existing methods in terms of speed, space, and scalability while achieving a comparable fitness, where the fitness indicates how a method approximates a given data well (see Section 4.4.1).




(a) Trade-off (b) Trade-off (c) Trade-off (d) Trade-off

Figure 4.1: **[Best viewed in color]** Measurement of the running time and fitness on real-world datasets for three target ranks $R$: 10, 15, and 20. DPar2 provides the best trade-off between speed and fitness. DPar2 is up to 6.0× faster than the competitors while having comparable fitness.

The contributions of this work are as follows.

- **Algorithm.** We propose DPar2, a fast and scalable PARAFAC2 decomposition method for decomposing irregular dense tensors.
- **Analysis.** We provide analysis for the time and space complexities of our proposed method DPar2.
- **Experiment.** DPar2 achieves up to 6.0× faster running time than previous PARAFAC2 decomposition methods based on ALS while achieving a similar fitness (see Figure 4.1).
- **Discovery.** With DPar2, we find that the Korean stock market and the US stock market have different correlations (see Figure 4.11) between features (e.g., prices and technical indicators). We also find similar stocks (see Table 4.3) on the US stock market during a specific event (e.g., COVID-19).

In the rest of the chapter, we propose our method DPar2 in Section 4.3, present experimental results in Section 4.4, and conclude in Section 4.5. The code and datasets are available at https://datalab.snu.ac.kr/dpar2.

## 4.2 Preliminaries

We use the symbols listed in Table 4.1.

### 4.2.1 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) decomposes $\mathbf{A} \in \mathbb{R}^{I \times J}$ to $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$. $\mathbf{U} \in \mathbb{R}^{I \times R}$ is the left singular vector matrix of $\mathbf{A}$; $\mathbf{U} = \begin{bmatrix} \mathbf{u}_1 \cdots \mathbf{u}_r \end{bmatrix}$ is a column orthogonal matrix where $R$ is the rank of $\mathbf{A}$ and $\mathbf{u}_1, \cdots, \mathbf{u}_R$ are the eigenvectors of $\mathbf{A}\mathbf{A}^{\mathrm{T}}$. $\mathbf{\Sigma}$ is an $R \times R$ diagonal matrix whose diagonal entries are singular values. The $i$-th singular value

Table 4.1: Symbol description.

| Symbol | Description |
|---|---|
| $\{\mathbf{X}_k\}_{k=1}^{K}$ | irregular tensor of slices $\mathbf{X}_k$ for $k = 1, ..., K$ |
| $\mathbf{X}_k$ | slice matrix ($\in I_k \times J$) |
| $\mathbf{X}(i,:)$ | $i$-th row of a matrix $\mathbf{X}$ |
| $\mathbf{X}(:,j)$ | $j$-th column of a matrix $\mathbf{X}$ |
| $\mathbf{X}(i,j)$ | $(i,j)$-th element of a matrix $\mathbf{X}$ |
| $\mathbf{X}_{(n)}$ | mode-$n$ matricization of a tensor $\mathcal{X}$ |
| $\mathbf{Q}_k$, $\mathbf{S}_k$ | factor matrices of the $k$th slice |
| $\mathbf{H}$, $\mathbf{V}$ | factor matrices of an irregular tensor |
| $\mathbf{A}_k$, $\mathbf{B}_k$, $\mathbf{C}_k$ | SVD results of the $k$th slice |
| $\mathbf{D}$, $\mathbf{E}$, $\mathbf{F}$ | SVD results of the second stage |
| $\mathbf{F}^{(k)}$ | $k$th vertical block matrix ($\in \mathbb{R}^{R\times R}$) of $\mathbf{F}(\in \mathbb{R}^{KR\times R})$ |
| $\mathbf{Z}_k$, $\mathbf{\Sigma}_k$, $\mathbf{P}_k$ | SVD results of $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$ |
| $R$ | target rank |
| $\otimes$ | Kronecker product |
| $\odot$ | Khatri-Rao product |
| $*$ | element-wise product |
| $\|\|$ | horizontal concatenation |
| $vec(\cdot)$ | vectorization of a matrix |

$\sigma_i$ is in $\mathbf{\Sigma}_{i,i}$ where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_R \geq 0$. $\mathbf{V} \in \mathbb{R}^{J\times R}$ is the right singular vector matrix of $\mathbf{A}$; $\mathbf{V} = \begin{bmatrix}\mathbf{v}_1 \cdots \mathbf{v}_R\end{bmatrix}$ is a column orthogonal matrix where $\mathbf{v}_1, \cdots, \mathbf{v}_R$ are the eigenvectors of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$.

**Randomized SVD**. Many works [84, 83, 85] have introduced efficient SVD methods to decompose a matrix $\mathbf{A} \in \mathbb{R}^{I\times J}$ by applying randomized algorithms. We introduce a popular randomized SVD in Algorithm 9. Randomized SVD finds a column orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{I\times(R+s)}$ of $(\mathbf{A}\mathbf{A}^T)^q\mathbf{A}\mathbf{\Omega}$ using random matrix $\mathbf{\Omega}$, constructs a smaller matrix $\mathbf{B} = \mathbf{Q}^T\mathbf{A}$ ($\in \mathbb{R}^{(R+s)\times J}$), and finally obtains the SVD result $\mathbf{U}$ ($= \mathbf{Q}\tilde{\mathbf{U}}$), $\mathbf{\Sigma}$, $\mathbf{V}$ of $\mathbf{A}$ by computing SVD for $\mathbf{B}$, i.e., $\mathbf{B} \approx \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^T$. Given a matrix $\mathbf{A}$, the time complexity of randomized SVD is $\mathcal{O}(IJR)$ where $R$ is the target rank.

**Algorithm 9:** Randomized SVD [83]

**Input:** $\mathbf{A} \in \mathbb{R}^{I \times J}$
**Output:** $\mathbf{U} \in \mathbb{R}^{I \times R}$, $\mathbf{S} \in \mathbb{R}^{R \times R}$, and $\mathbf{V} \in \mathbb{R}^{J \times R}$.
**Parameters:** target rank $R$, and an exponent $q$

1: generate a Gaussian test matrix $\mathbf{\Omega} \in \mathbb{R}^{J \times (R+s)}$
2: construct $\mathbf{Y} \leftarrow (\mathbf{A}\mathbf{A}^T)^q \mathbf{A}\mathbf{\Omega}$
3: $\mathbf{QR} \leftarrow \mathbf{Y}$ using QR factorization
4: construct $\mathbf{B} \leftarrow \mathbf{Q}^T\mathbf{A}$
5: $\tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^T \leftarrow \mathbf{B}$ using truncated SVD at rank $R$
6: **return** $\mathbf{U} = \mathbf{Q}\tilde{\mathbf{U}}$, $\mathbf{\Sigma}$, and $\mathbf{V}$

## 4.3 Proposed Method

In this section, we propose DPar2, a fast and scalable PARAFAC2 decomposition method for irregular dense tensors.

### 4.3.1 Overview

Before describing main ideas of our method, we present main challenges that need to be tackled.

C1. **Dealing with large irregular tensors.** PARAFAC2 decomposition (Algorithm 2) iteratively updates factor matrices (i.e., $\mathbf{U}_k$, $\mathbf{S}_k$, and $\mathbf{V}$) using an input tensor. Dealing with a large input tensor is burdensome to update the factor matrices as the number of iterations increases.

C2. **Minimizing numerical computations and intermediate data.** How can we minimize the intermediate data and overall computations?

C3. **Maximizing multi-core parallelism.** How can we parallelize the computations for PARAFAC2 decomposition?

The main ideas that address the challenges mentioned above are as follows:

I1. **Compressing an input tensor using randomized SVD** considerably re-

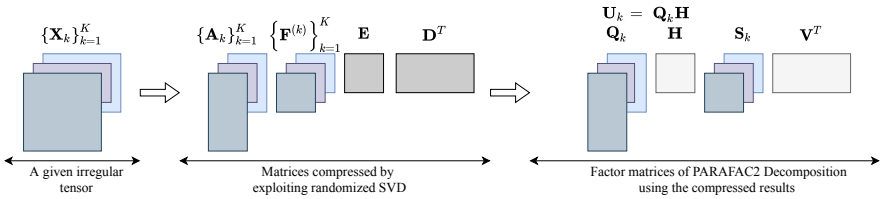



Figure 4.2: Overview of DPar2. Given an irregular tensor $\{\mathbf{X}_k\}_{k=1}^{K}$, DPar2 first compresses the given irregular tensor by exploiting randomized SVD. Then, DPar2 iteratively and efficiently updates the factor matrices, $\mathbf{Q}_k$, $\mathbf{H}$, $\mathbf{S}_k$, and $\mathbf{V}$, using only the compressed matrices, to get the result of PARAFAC2 decomposition.

duces the computational costs to update factor matrices (Section 4.3.2).

I2. **Careful reordering of computations with the compression results** minimizes the intermediate data and the number of operations (Sections 4.3.3 to 4.3.5).

I3. **Careful distribution of work between threads** enables DPar2 to achieve high efficiency by considering various lengths $I_k$ for $k = 1, \ldots, K$ (Section 4.3.6).

As shown in Figure 4.2, DPar2 first compresses each slice of an irregular tensor using randomized SVD (Section 4.3.2). The compression is performed once before iterations, and only the compression results are used at iterations. It significantly reduces the time and space costs in updating factor matrices. After compression, DPar2 updates factor matrices at each iteration, by exploiting the compression results (Sections 4.3.3 to 4.3.5). Careful reordering of computations is required to achieve high efficiency. Also, by carefully allocating input slices to threads, DPar2 accelerates the overall process (Section 4.3.6).

### 4.3.2 Compressing an irregular input tensor

DPar2 (see Algorithm 10) is a fast and scalable PARAFAC2 decomposition method based on ALS described in Algorithm 2. The main challenge that needs to be tackled is to minimize the number of heavy computations involved with a given irregular tensor $\{\mathbf{X}_k\}_{k=1}^{K}$ consisting of slices $\mathbf{X}_k$ for $k = 1, \ldots, K$ (in lines 4 and 8 of Algorithm 2). As the

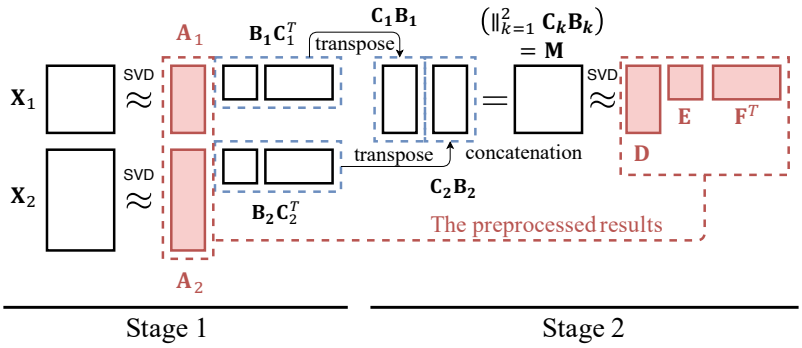



Figure 4.3: Two-stage SVD for a given irregular tensor. In the first stage, DPar2 performs randomized SVD of $\mathbf{X}_k$ for all $k$. In the second stage, DPar2 performs randomized SVD of $\mathbf{M} \in \mathbb{R}^{J \times KR}$ which is the horizontal concatenation of $\mathbf{C}_k\mathbf{B}_k$.

number of iterations increases (lines 2 to 17 in Algorithm 2), the heavy computations make PARAFAC2-ALS slow. For efficiency, we preprocess a given irregular tensor into small matrices, and then update factor matrices by carefully using the small ones.

Our approach to address the above challenges is to compress a given irregular tensor $\{\mathbf{X}_k\}_{k=1}^{K}$ before starting iterations. As shown in Figure 4.3, our main idea is two-stage lossy compression with randomized SVD for the given tensor: 1) DPar2 performs randomized SVD for each slice $\mathbf{X}_k$ for $k = 1, \ldots, K$ at target rank $R$, and 2) DPar2 performs randomized SVD for a matrix, the horizontal concatenation of singular value matrices and right singular vector matrices of slices $\mathbf{X}_k$. Randomized SVD allows us to compress slice matrices with low computational costs and low errors.

**First Stage.** In the first stage, DPar2 compresses a given irregular tensor by performing randomized SVD for each slice $\mathbf{X}_k$ at target rank $R$ (line 3 in Algorithm 10).

$$\mathbf{X}_k \approx \mathbf{A}_k\mathbf{B}_k\mathbf{C}_k^T \tag{4.1}$$

where $\mathbf{A}_k \in \mathbb{R}^{I_k \times R}$ is a matrix consisting of left singular vectors, $\mathbf{B}_k \in \mathbb{R}^{R \times R}$ is a diagonal matrix whose elements are singular values, and $\mathbf{C}_k \in \mathbb{R}^{J \times R}$ is a matrix consisting

**Algorithm 10:** DPAR2

**Input:** $\mathbf{X}_k \in \mathbb{R}^{I_k \times J}$ for $k = 1, ..., K$
**Output:** $\mathbf{U}_k \in \mathbb{R}^{I_k \times R}$, $\mathbf{S}_k \in \mathbb{R}^{R \times R}$ for $k = 1, ..., K$, and $\mathbf{V} \in \mathbb{R}^{J \times R}$.
**Parameters:** target rank $R$

1: initialize matrices $\mathbf{H} \in \mathbb{R}^{R \times R}$, $\mathbf{V}$, and $\mathbf{S}_k$ for $k = 1, ..., K$
/* Compressing slices in parallel */
2: **for** $k = 1, ..., K$ **do**
3: compute $\mathbf{A}_k \mathbf{B}_k \mathbf{C}_k^T \leftarrow \text{SVD}(\mathbf{X}_k)$ by performing randomized SVD at rank $R$
4: **end for**
5: $\mathbf{M} \leftarrow \|_{k=1}^{K} (\mathbf{C}_k \mathbf{B}_k)$
6: compute $\mathbf{DEF}^T \leftarrow \text{SVD}(\mathbf{M})$ by performing randomized SVD at rank $R$
/* Iteratively updating factor matrices */
7: **repeat**
8: **for** $k = 1, ..., K$ **do**
9: compute $\mathbf{Z}_k \mathbf{\Sigma}_k \mathbf{P}_k^T \leftarrow \text{SVD}(\mathbf{F}^{(k)} \mathbf{E} \mathbf{D}^T \mathbf{V} \mathbf{S}_k \mathbf{H}^T)$ by performing SVD at rank $R$
10: **end for**
/* no explicit computation of $\mathbf{Y}_k$ */
11: **for** $k = 1, ..., K$ **do**
12: $\mathbf{Y}_k \leftarrow \mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)} \mathbf{E} \mathbf{D}^T$
13: **end for**
/* running a single iteration of CP-ALS on $\mathcal{Y}$ */
14: compute $\mathbf{G}^{(1)} \leftarrow \mathbf{Y}_{(1)} (\mathbf{W} \odot \mathbf{V})$ based on Lemma 4.1
15: $\mathbf{H} \leftarrow \mathbf{G}^{(1)} (\mathbf{W}^T \mathbf{W} * \mathbf{V}^T \mathbf{V})^\dagger$ ▷ Normalize $\mathbf{H}$
16: compute $\mathbf{G}^{(2)} \leftarrow \mathbf{Y}_{(2)} (\mathbf{W} \odot \mathbf{H})$ based on Lemma 4.2
17: $\mathbf{V} \leftarrow \mathbf{G}^{(2)} (\mathbf{W}^T \mathbf{W} * \mathbf{H}^T \mathbf{H})^\dagger$ ▷ Normalize $\mathbf{V}$
18: compute $\mathbf{G}^{(3)} \leftarrow \mathbf{Y}_{(3)} (\mathbf{V} \odot \mathbf{H})$ based on Lemma 4.3
19: $\mathbf{W} \leftarrow \mathbf{G}^{(3)} (\mathbf{V}^T \mathbf{V} * \mathbf{H}^T \mathbf{H})^\dagger$
20: **for** $k = 1, ..., K$ **do**
21: $\mathbf{S}_k \leftarrow diag(\mathbf{W}(k, :))$
22: **end for**
23: **until** the maximum iteration is reached, or the error ceases to decrease;
24: **for** $k = 1, ..., K$ **do**
25: $\mathbf{U}_k \leftarrow \mathbf{A}_k \mathbf{Z}_k \mathbf{P}_k^T \mathbf{H}$
26: **end for**

of right singular vectors.

**Second Stage.** Although small compressed data are generated in the first step, there is room to further compress the intermediate data from the first stage. In the second stage, we compress a matrix $\mathbf{M} = \|_{k=1}^{K} (\mathbf{C}_k \mathbf{B}_k)$ which is the horizontal concatenation of $\mathbf{C}_k \mathbf{B}_k$ for $k = 1, ..., K$. Compressing the matrix $\mathbf{M}$ maximizes the efficiency

of updating factor matrices $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ (see Equation (2.8)) at later iterations. We construct a matrix $\mathbf{M} \in \mathbb{R}^{J \times KR}$ by horizontally concatenating $\mathbf{C}_k\mathbf{B}_k$ for $k = 1, \ldots, K$ (line 5 in Algorithm 10). Then, DPar2 performs randomized SVD for $\mathbf{M}$ (line 6 in Algorithm 10):

$$\mathbf{M} = [\mathbf{C}_1\mathbf{B}_1; \cdots; \mathbf{C}_K\mathbf{B}_K] = \|_{k=1}^{K}(\mathbf{C}_k\mathbf{B}_k) \approx \mathbf{DEF}^T \tag{4.2}$$

where $\mathbf{D} \in \mathbb{R}^{J \times R}$ is a matrix consisting of left singular vectors, $\mathbf{E} \in \mathbb{R}^{R \times R}$ is a diagonal matrix whose elements are singular values, and $\mathbf{F} \in \mathbb{R}^{KR \times R}$ is a matrix consisting of right singular vectors.

With the two stages, we obtain the compressed results $\mathbf{D}$, $\mathbf{E}$, $\mathbf{F}$, and $\mathbf{A}_k$ for $k = 1, \ldots, K$. Before describing how to update factor matrices, we re-express the $k$-th slice $\mathbf{X}_k$ by using the compressed results:

$$\mathbf{X}_k \approx \mathbf{A}_k\mathbf{F}^{(k)}\mathbf{ED}^T \tag{4.3}$$

where $\mathbf{F}^{(k)} \in \mathbb{R}^{R \times R}$ is the $k$th vertical block matrix of $\mathbf{F}$:

$$\mathbf{F} = \begin{bmatrix} \mathbf{F}^{(1)} \\ \vdots \\ \mathbf{F}^{(K)} \end{bmatrix} \tag{4.4}$$

Since $\mathbf{C}_k\mathbf{B}_k$ is the $k$th horizontal block of $\mathbf{M}$ and $\mathbf{DEF}^{(k)T}$ is the $k$th horizontal block of $\mathbf{DEF}^T$, $\mathbf{B}_k\mathbf{C}_k^T$ corresponds to $\mathbf{F}^{(k)}\mathbf{ED}^T$. Therefore, we obtain Equation (4.3) by replacing $\mathbf{B}_k\mathbf{C}_k^T$ with $\mathbf{F}^{(k)}\mathbf{ED}^T$ from Equation (4.1).

In updating factor matrices, we use $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{ED}^T$ instead of $\mathbf{X}_k$. The two-stage compression lays the groundwork for efficient updates.

### 4.3.3 Overview of update rule

Our goal is to efficiently update factor matrices, $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{S}_k$ and $\mathbf{Q}_k$ for $k = 1, ..., K$, using the compressed results $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$. The main challenge of updating factor matrices is to minimize numerical computations and intermediate data by exploiting the compressed results obtained in Section 4.3.2. A naive approach would reconstruct $\tilde{\mathbf{X}}_k = \mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ from the compressed results, and then update the factor matrices. However, this approach fails to improve the efficiency of updating factor matrices. We propose an efficient update rule using the compressed results to 1) find $\mathbf{Q}_k$ and $\mathbf{Y}_k$ (lines 5 and 8 in Algorithm 2), and 2) compute a single iteration of CP-ALS (lines 11 to 13 in Algorithm 2).

There are two differences between our update rule and PARAFAC2-ALS (Algorithm 2). First, we avoid explicit computations of $\mathbf{Q}_k$ and $\mathbf{Y}_k$. Instead, we find small factorized matrices of $\mathbf{Q}_k$ and $\mathbf{Y}_k$, respectively, and then exploit the small ones to update $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$. The small matrices are computed efficiently by exploiting the compressed results $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ instead of $\mathbf{X}_k$. The second difference is that DPar2 obtains $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ using the small factorized matrices of $\mathbf{Y}_k$. Careful ordering of computations with them considerably reduces time and space costs at each iteration. We describe how to find the factorized matrices of $\mathbf{Q}_k$ and $\mathbf{Y}_k$ in Section 4.3.4, and how to update factor matrices in Section 4.3.5.

### 4.3.4 Finding the factorized matrices of $\mathbf{Q}_k$ and $\mathbf{Y}_k$

The first goal of updating factor matrices is to find the factorized matrices of $\mathbf{Q}_k$ and $\mathbf{Y}_k$ for $k = 1, ..., K$, respectively. In Algorithm 2, finding $\mathbf{Q}_k$ and $\mathbf{Y}_k$ is expensive due to the computations involved with $\mathbf{X}_k$ (lines 4 and 8 in Algorithm 2). To reduce the costs for $\mathbf{Q}_k$ and $\mathbf{Y}_k$, our main idea is to exploit the compressed results $\mathbf{A}_k$, $\mathbf{D}$, $\mathbf{E}$, and

$\mathbf{F}^{(k)}$, instead of $\mathbf{X}_k$. Additionally, we exploit the column orthogonal property of $\mathbf{A}_k$, i.e., $\mathbf{A}_k^T\mathbf{A}_k = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

We first re-express $\mathbf{Q}_k$ using the compressed results obtained in Section 4.3.2. DPAR2 reduces the time and space costs for $\mathbf{Q}_k$ by exploiting the column orthogonal property of $\mathbf{A}_k$. First, we express $\mathbf{X}_k\mathbf{V}\mathbf{S}_k\mathbf{H}^T$ as $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$ by replacing $\mathbf{X}_k$ with $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$. Next, we need to obtain left and right singular vectors of $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$. A naive approach is to compute SVD of $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$, but there is a more efficient way than this approach. Thanks to the column orthogonal property of $\mathbf{A}_k$, DPAR2 performs SVD of $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\,\mathbf{H}^T \in \mathbb{R}^{R\times R}$, not $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T \in \mathbb{R}^{I_k\times R}$, at target rank $R$ (line 9 in Algorithm 10):

$$\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T \overset{\text{SVD}}{=} \mathbf{Z}_k\mathbf{\Sigma}_k\mathbf{P}_k^T \tag{4.5}$$

where $\mathbf{\Sigma}_k$ is a diagonal matrix whose entries are the singular values of $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$, the column vectors of $\mathbf{Z}_k$ and $\mathbf{P}_k$ are the left singular vectors and the right singular vectors of $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$, respectively. Then, we obtain the factorized matrices of $\mathbf{Q}_k$ as follows:

$$\mathbf{Q}_k = \mathbf{A}_k\mathbf{Z}_k\mathbf{P}_k^T \tag{4.6}$$

where $\mathbf{A}_k\mathbf{Z}_k$ and $\mathbf{P}_k$ are the left and the right singular vectors of $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$, respectively. We avoid the explicit construction of $\mathbf{Q}_k$, and use $\mathbf{A}_k\mathbf{Z}_k\mathbf{P}_k^T$ instead of $\mathbf{Q}_k$. Since $\mathbf{A}_k$ is already column-orthogonal, we avoid performing SVD of $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$, which are much larger than $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$.

Next, we find the factorized matrices of $\mathbf{Y}_k$. DPAR2 re-expresses $\mathbf{Q}_k^T\mathbf{X}_k$ (line 8 in Algorithm 2) as $\mathbf{Q}_k^T\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ using Equation (4.3). Instead of directly computing

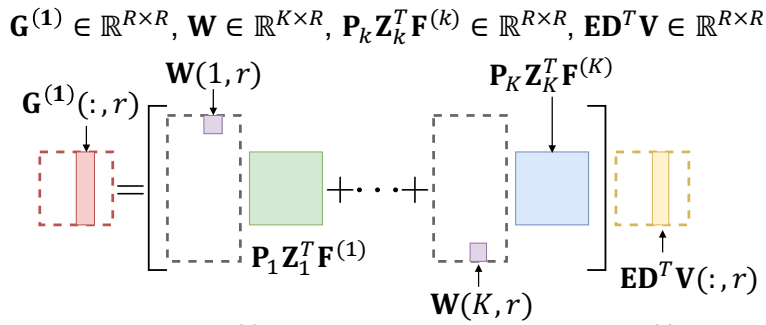



Figure 4.4: Computation for $\mathbf{G}^{(1)} = \mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$. The $r$th column $\mathbf{G}^{(1)}(:,r)$ of $\mathbf{G}^{(1)}$ is computed by $\left(\sum_{k=1}^{K} \mathbf{W}(k,r)\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)\mathbf{E}\mathbf{D}^T\mathbf{V}(:,r)$.

$\mathbf{Q}_k^T\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$, we replace $\mathbf{Q}_k^T$ with $\mathbf{P}_k\mathbf{Z}_k^T\mathbf{A}_k^T$. Then, we represent $\mathbf{Y}_k$ as the following expression (line 12 in Algorithm 10):

$$\mathbf{Y}_k \leftarrow \mathbf{Q}_k^T\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T = \mathbf{P}_k\mathbf{Z}_k^T\mathbf{A}_k^T\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T = \mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$$

Note that we use the property $\mathbf{A}_k^T\mathbf{A}_k = \mathbf{I}_{R\times R}$, where $\mathbf{I}_{R\times R}$ is the identity matrix of size $R \times R$, for the last equality. By exploiting the factorized matrices of $\mathbf{Q}_k$, we compute $\mathbf{Y}_k$ without involving $\mathbf{A}_k$ in the process.

### 4.3.5 Updating H, V, and W

The next goal is to efficiently update the matrices $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ using the small factorized matrices of $\mathbf{Y}_k$. Naively, we would compute $\mathcal{Y}$ and run a single iteration of CP-ALS with $\mathcal{Y}$ to update $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ (lines 11 to 13 in Algorithm 2). However, multiplying a matricized tensor and a Khatri-Rao product (e.g., $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$) is burdensome, and thus we exploit the structure of the decomposed results $\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ of $\mathbf{Y}_k$ to reduce memory requirements and computational costs. In other word, we do not compute $\mathbf{Y}_k$, and use only $\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ in updating $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$. Note that the $k$-th frontal slice of $\mathcal{Y}$, $\mathcal{Y}(:,:,k)$, is $\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$.

**Updating H.** In $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})(\mathbf{W}^T\mathbf{W} * \mathbf{V}^T\mathbf{V})^{\dagger}$, we focus on efficiently computing $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ based on Lemma 4.1. A naive computation for $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ requires a high computational cost $\mathcal{O}(JKR^2)$ due to the explicit reconstruction of $\mathbf{Y}_{(1)}$. Therefore, we compute that term without the reconstruction by carefully determining the order of computations and exploiting the factorized matrices of $\mathbf{Y}_{(1)}$, $\mathbf{D}$, $\mathbf{E}$, $\mathbf{P}_k$, $\mathbf{Z}_k$, and $\mathbf{F}^{(k)}$ for $k = 1, ..., K$. With Lemma 4.1, we reduce the computational cost of $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ to $\mathcal{O}(JR^2 + KR^3)$.

**Lemma 4.1.** *Let us denote* $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ *with* $\mathbf{G}^{(1)} \in \mathbb{R}^{R \times R}$. $\mathbf{G}^{(1)}(:, r)$ *is equal to* $\left(\left(\sum_{k=1}^{K} \mathbf{W}(k, r) \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \mathbf{E}\mathbf{D}^T \mathbf{V}(:, r)\right)$. □

*Proof.* $\mathbf{Y}_{(1)}$ is represented as follows:

$$\begin{aligned}
\mathbf{Y}_{(1)} &= \left[\mathbf{P}_1 \mathbf{Z}_1^T \mathbf{F}^{(1)} \mathbf{E}\mathbf{D}^T \quad ; \cdots ; \quad \mathbf{P}_K \mathbf{Z}_K^T \mathbf{F}^{(K)} \mathbf{E}\mathbf{D}^T\right] \\
&= \left(\|_{k=1}^{K} \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \begin{bmatrix} \mathbf{E}\mathbf{D}^T & \cdots & \mathbf{O} \\ \vdots & \ddots & \vdots \\ \mathbf{O} & \cdots & \mathbf{E}\mathbf{D}^T \end{bmatrix} = \left(\|_{k=1}^{K} \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \left(\mathbf{I}_{K \times K} \otimes \mathbf{E}\mathbf{D}^T\right)
\end{aligned}$$

where $\mathbf{I}_{K \times K}$ is the identity matrix of size $K \times K$. Then, $\mathbf{G}^{(1)} = \mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ is expressed as follows:

$$\begin{aligned}
\mathbf{G}^{(1)} &= \left(\|_{k=1}^{K} \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \left(\mathbf{I}_{K \times K} \otimes \mathbf{E}\mathbf{D}^T\right) \left(\|_{r=1}^{R} (\mathbf{W}(:, r) \otimes \mathbf{V}(:, r))\right) \\
&= \left(\|_{k=1}^{K} \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \left(\|_{r=1}^{R} \left(\mathbf{W}(:, r) \otimes \mathbf{E}\mathbf{D}^T \mathbf{V}(:, r)\right)\right)
\end{aligned}$$

The mixed-product property (i.e., $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$)) is used in the above equation. Therefore, $\mathbf{G}^{(1)}(:, r)$ is equal to $\left(\|_{k=1}^{K} \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right)\right) \left(\mathbf{W}(:, r) \otimes \mathbf{E}\mathbf{D}^T \mathbf{V}(:, r)\right)$. We represent it as $\sum_{k=1}^{K} \mathbf{W}(k, r) \left(\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)}\right) \mathbf{E}\mathbf{D}^T \mathbf{V}(:, r)$ using block matrix multipli-

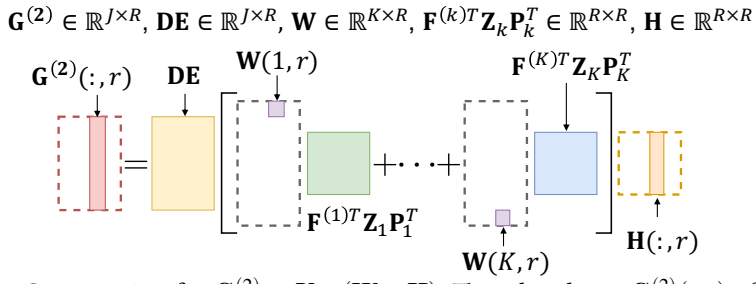



Figure 4.5: Computation for $\mathbf{G}^{(2)} = \mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{H})$. The $r$th column $\mathbf{G}^{(2)}(:,r)$ of $\mathbf{G}^{(2)}$ is computed by $\mathbf{DE}\sum_{k=1}^{K}\left(\mathbf{W}(k,r)\mathbf{F}^{(k)T}\mathbf{Z}_k\mathbf{P}_k^T\mathbf{H}(:,r)\right)$.

cation since the $k$-th vertical block vector of $\left(\mathbf{W}(:,r) \otimes \mathbf{ED}^T\mathbf{V}(:,r)\right) \in \mathbb{R}^{KR}$ is $\mathbf{W}(k,r)$ $\mathbf{ED}^T\mathbf{V}(:,r) \in \mathbb{R}^R$. □

As shown in Figure 4.4, we compute $\mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$ column by column. In computing $\mathbf{G}^{(1)}(:,r)$, we compute $\mathbf{ED}^T\mathbf{V}(:,r)$, sum up $\mathbf{W}(k,r)\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)$ for all $k$, and then perform matrix multiplication between the two preceding results (line 14 in Algorithm 10). After computing $\mathbf{G}^{(1)} \leftarrow \mathbf{Y}_{(1)}(\mathbf{W} \odot \mathbf{V})$, we update $\mathbf{H}$ by computing $\mathbf{G}^{(1)}(\mathbf{W}^T\mathbf{W} * \mathbf{V}^T\mathbf{V})^\dagger$ where $\dagger$ denotes the Moore-Penrose pseudoinverse (line 15 in Algorithm 10). Note that the pseudoinverse operation requires a lower computational cost compared to computing $\mathbf{G}^{(1)}$ since the size of $(\mathbf{W}^T\mathbf{W} * \mathbf{V}^T\mathbf{V}) \in \mathbb{R}^{R\times R}$ is small.

**Updating V.** In computing $\mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{U})(\mathbf{W}^T\mathbf{W} * \mathbf{U}^T\mathbf{U})^\dagger$, we need to efficiently compute $\mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{U})$ based on Lemma 4.2. As in updating $\mathbf{H}$, a naive computation for $\mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{U})$ requires a high computational cost $\mathcal{O}(JKR^2)$. We efficiently compute $\mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{U})$ with the cost $\mathcal{O}(JR^2 + KR^3)$, by carefully determining the order of computations and exploiting the factorized matrices of $\mathbf{Y}_{(2)}$.

**Lemma 4.2.** *Let us denote* $\mathbf{Y}_{(2)}(\mathbf{W} \odot \mathbf{H})$ *with* $\mathbf{G}^{(2)} \in \mathbb{R}^{J\times R}$. $\mathbf{G}^{(2)}(:,r)$ *is equal to* $\mathbf{DE}$ $\left(\sum_{k=1}^{K}\left(\mathbf{W}(k,r)\mathbf{F}^{(k)T}\mathbf{Z}_k\mathbf{P}_k^T\mathbf{H}(:,r)\right)\right)$. □

*Proof.* $\mathbf{Y}_{(2)}$ is represented as follows:

$$\mathbf{Y}_{(2)} = \left[\mathbf{DEF}^{(1)T}\mathbf{Z}_1\mathbf{P}_1^T \quad ;\cdots; \quad \mathbf{DEF}^{(K)T}\mathbf{Z}_K\mathbf{P}_K^T\right] = \mathbf{DE}\left(\|_{k=1}^{K}\mathbf{F}^{(k)T}\mathbf{Z}_k\mathbf{P}_k^T\right)$$

Then, $\mathbf{G}^{(2)} = \mathbf{Y}_{(2)}(\mathbf{W}\odot\mathbf{H})$ is expressed as follows:

$$\mathbf{G}^{(2)} = \mathbf{DE}\left(\|_{k=1}^{K}\mathbf{F}^{(k)T}\mathbf{Z}_k\mathbf{P}_k^T\right)\begin{bmatrix}\mathbf{W}(1,1)\mathbf{H}(:,1); & \cdots & ;\mathbf{W}(1,R)\mathbf{H}(:,R)\\ \vdots & \vdots & \vdots\\ \mathbf{W}(K,1)\mathbf{H}(:,1); & \cdots & ;\mathbf{W}(K,R)\mathbf{H}(:,R)\end{bmatrix}$$

$\mathbf{G}^{(2)}(:,r)$ is equal to $\mathbf{DE}\sum_{k=1}^{K}\left(\mathbf{W}(k,r)\mathbf{F}^{(k)T}\mathbf{Z}_k\mathbf{P}_k^T\mathbf{H}(:,r)\right)$ according to the above equation. □

As shown in Figure 4.5, we compute $\mathbf{G}^{(2)} \leftarrow \mathbf{Y}_{(2)}(\mathbf{W}\odot\mathbf{H})$ column by column. After computing $\mathbf{G}^{(2)}$, we update $\mathbf{V}$ by computing $\mathbf{G}^{(2)}(\mathbf{W}^T\mathbf{W}*\mathbf{H}^T\mathbf{H})^{\dagger}$ (lines 16 and 17 in Algorithm 10).

**Updating W.** In computing $\mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})(\mathbf{V}^T\mathbf{V}*\mathbf{H}^T\mathbf{H})^{\dagger}$, we efficiently compute $\mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ based on Lemma 4.3. As in updating $\mathbf{H}$ and $\mathbf{V}$, a naive computation for $\mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ requires a high computational cost $\mathcal{O}(JKR^2)$. We compute $\mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ with the cost $\mathcal{O}(JR^2+KR^3)$ based on Lemma 4.3. Exploiting the factorized matrices of $\mathbf{Y}_{(3)}$ and carefully determining the order of computations improves the efficiency.

**Lemma 4.3.** *Let us denote* $\mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ *with* $\mathbf{G}^{(3)}\in\mathbb{R}^{K\times R}$. $\mathbf{G}^{(3)}(k,r)$ *is equal to* $vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)^T\left(\mathbf{ED}^T\mathbf{V}(:,r)\otimes\mathbf{H}(:,r)\right)$ *where* $vec(\cdot)$ *denotes the vectorization of a matrix.* □

$$\mathbf{G}^{(3)} \in \mathbb{R}^{K\times R},\ \mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)} \in \mathbb{R}^{R\times R}, \mathbf{E}\mathbf{D}^T\mathbf{V} \in \mathbb{R}^{R\times R},\ \mathbf{H} \in \mathbb{R}^{R\times R}$$

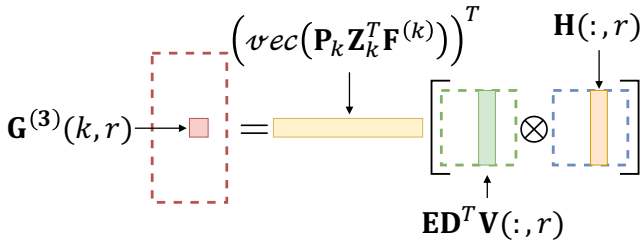



Figure 4.6: Computation for $\mathbf{G}^{(3)} = \mathbf{Y}_{(3)}(\mathbf{V} \odot \mathbf{H})$. $\mathbf{G}^{(3)}(k,r)$ is computed by $\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)^T \left(\mathbf{E}\mathbf{D}^T\mathbf{V}(:,r) \otimes \mathbf{H}(:,r)\right)$.

*Proof.* $\mathbf{Y}_{(3)}$ is represented as follows:

$$\mathbf{Y}_{(3)} = \begin{bmatrix} \left(vec\left(\mathbf{P}_1\mathbf{Z}_1^T\mathbf{F}^{(1)}\mathbf{E}\mathbf{D}^T\right)\right)^T \\ \vdots \\ \left(vec\left(\mathbf{P}_K\mathbf{Z}_K^T\mathbf{F}^{(K)}\mathbf{E}\mathbf{D}^T\right)\right)^T \end{bmatrix} = \left(\|_{k=1}^K\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\right)\right)\right)^T$$

$$= \left(\|_{k=1}^K(\mathbf{D}\mathbf{E}\otimes\mathbf{I})\, vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)^T = \left(\|_{k=1}^K\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)\right)^T \left(\mathbf{E}\mathbf{D}^T \otimes \mathbf{I}_{R\times R}\right)$$

where $\mathbf{I}_{R\times R}$ is the identity matrix of size $R \times R$. The property of the vectorization (i.e., $vec(\mathbf{A}\mathbf{B}) = (\mathbf{B}^T \otimes \mathbf{I})vec(\mathbf{A})$) is used. Then, $\mathbf{G}^{(3)} = \mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ is expressed as follows:

$$\mathbf{G}^{(3)} = \left(\|_{k=1}^K\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)\right)^T \left(\|_{r=1}^R\left(\mathbf{E}\mathbf{D}^T\mathbf{V}(:,r)\otimes\mathbf{H}(:,r)\right)\right)$$

$\mathbf{G}^{(3)}(k,r)$ is $\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)^T \left(\mathbf{E}\mathbf{D}^T\mathbf{V}(:,r)\otimes\mathbf{H}(:,r)\right)$ according to the above equation. □

We compute $\mathbf{G}^{(3)} = \mathbf{Y}_{(3)}(\mathbf{V}\odot\mathbf{H})$ row by row. Figure 4.6 shows how we compute $\mathbf{G}^{(3)}(k,r)$. In computing $\mathbf{G}^{(3)}$, we first compute $\mathbf{E}\mathbf{D}^T\mathbf{V}$, and then obtain $\mathbf{G}^{(3)}(k,:)$ for all $k$ (line 18 in Algorithm 10). After computing $\mathbf{G}^{(3)}$, we update $\mathbf{W}$ by computing $\mathbf{G}^{(3)}(\mathbf{V}^T\mathbf{V} * \mathbf{H}^T\mathbf{H})^\dagger$ where $\dagger$ denotes the Moore-Penrose pseudoinverse (line 19 in

Algorithm 10). We obtain $\mathbf{S}_k$ whose diagonal elements correspond to the $k$th row vector of $\mathbf{W}$ (line 21 in Algorithm 10).

After convergence, we obtain the factor matrices, ($\mathbf{U}_k \leftarrow \mathbf{A}_k\mathbf{Z}_k\mathbf{P}_k^T\mathbf{H} = \mathbf{Q}_k\mathbf{H}$), $\mathbf{S}_k$, and $\mathbf{V}$ (line 25 in Algorithm 10).

**Convergence Criterion.** At the end of each iteration, we determine whether to stop or not (line 23 in Algorithm 10) based on the variation of $e = \left(\sum_{k=1}^{K} \|\mathbf{X}_k - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2\right)$ where $\hat{\mathbf{X}}_k = \mathbf{Q}_k\mathbf{H}\mathbf{S}_k\mathbf{V}^T$ is the $k$th reconstructed slice. However, measuring reconstruction errors $\sum_{k=1}^{K} \|\mathbf{X}_k - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2$ is inefficient since it requires high time and space costs proportional to input slices $\mathbf{X}_k$. To efficiently verify the convergence, our idea is to exploit $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ instead of $\mathbf{X}_k$, since the objective of our update process is to minimize the difference between $\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ and $\hat{\mathbf{X}}_k = \mathbf{Q}_k\mathbf{H}\mathbf{S}_k\mathbf{V}^T$. With this idea, we improve the efficiency by computing $\sum_{k=1}^{K} \|\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \mathbf{H}\mathbf{S}_k\mathbf{V}^T\|_{\mathbf{F}}^2$, not the reconstruction errors. Our computation requires the time $\mathcal{O}(JKR^2)$ and space costs $\mathcal{O}(JKR)$ which are much lower than the costs $\mathcal{O}(\sum_{k=1}^{K} I_kJR)$ and $\mathcal{O}(\sum_{k=1}^{K} I_kJ)$ of naively computing $\sum_{k=1}^{K} \|\mathbf{X}_k - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2$, respectively. From $\|\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \mathbf{H}\mathbf{S}_k\mathbf{V}^T\|_{\mathbf{F}}^2$, we derive $\|\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2$. Since the Frobenius norm is unitarily invariant, we modify the computation as follows:

$$\begin{aligned}
&\|\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \mathbf{H}\mathbf{S}_k\mathbf{V}^T\|_{\mathbf{F}}^2 = \|\mathbf{Q}_k\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \mathbf{Q}_k\mathbf{H}\mathbf{S}_k\mathbf{V}^T\|_{\mathbf{F}}^2 \\
&= \|\mathbf{A}_k\mathbf{Z}_k\mathbf{P}_k^T\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \mathbf{Q}_k\mathbf{H}\mathbf{S}_k\mathbf{V}^T\|_{\mathbf{F}}^2 = \|\mathbf{A}_k\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2
\end{aligned}$$

where $\mathbf{P}_k^T\mathbf{P}_k$ and $\mathbf{Z}_k\mathbf{Z}_k^T$ are equal to $\mathbf{I} \in \mathbb{R}^{R\times R}$ since $\mathbf{P}_k$ and $\mathbf{Z}_k$ are orthonormal matrices. Note that the size of $\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T$ and $\mathbf{H}\mathbf{S}_k\mathbf{V}^T$ is $R \times J$ which is much smaller than the size $I_k \times J$ of input slices $\mathbf{X}_k$. This modification completes the efficiency of our update rule.

**Algorithm 11:** Careful distribution of work in DPar2

**Input:** the number $T$ of threads, $\mathbf{X}_k \in \mathbb{R}^{I_k \times J}$ for $k = 1, ..., K$
**Output:** sets $\mathcal{T}_i$ for $i = 1, ..., T$.

```
initialize T_i ← ∅ for i = 1, ..., T.
construct a list S of size T whose elements are zero
construct a list L_init containing the number of rows of X_k for k = 1, ..., K
sort L_init in descending order, and obtain lists L_val and L_ind that contain sorted values
    and those corresponding indices
for k = 1, ..., K do
    t_min ← argmin S
    l ← L_ind[k]
    T_t_min ← T_t_min ∪ {X_l}
    S[t_min] ← S[t_min] + L_val[k]
end for
```

### 4.3.6 Careful distribution of work

The last challenge for an efficient and scalable PARAFAC2 decomposition method is how to parallelize the computations described in Sections 4.3.2 to 4.3.5. Although a previous work [2] introduces the parallelization with respect to the $K$ slices, there is still room for maximizing parallelism. Our main idea is to carefully allocate input slices $\mathbf{X}_k$ to threads by considering the irregularity of a given tensor.

The most expensive operation is to compute randomized SVD of input slices $\mathbf{X}_k$ for all $k$; thus we first focus on how well we parallelize this computation (i.e., lines 2 to 4 in Algorithm 10). A naive approach is to randomly allocate input slices to threads, and let each thread compute randomized SVD of the allocated slices. However, the completion time of each thread can vary since the computational cost of computing randomized SVD is proportional to the number of rows of slices; the number of rows of input slices is different from each other as shown in Figure 4.7. Therefore, we need to distribute $\mathbf{X}_k$ fairly across each thread considering their numbers of rows.

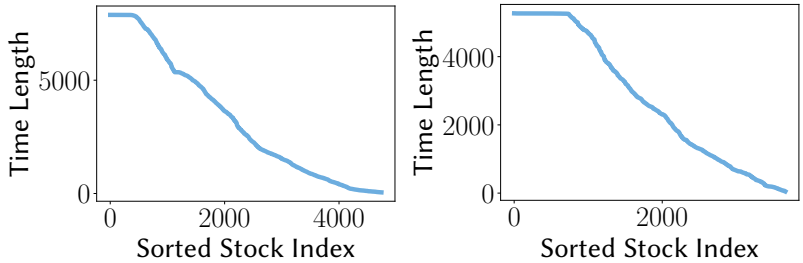
For $i = 1, .., T$, consider that an $i$th thread performs randomized SVD for slices in a set $\mathcal{T}_i$ where $T$ is the number of threads. To reduce the completion time, the sums of

rows of slices in the sets should be nearly equal to each other. To achieve it, we exploit a greedy number partitioning technique that repeatedly adds a slice into a set with the smallest sum of rows. Algorithm 11 describes how to construct the sets $\mathcal{T}_i$ for compressing input slices in parallel. Let $L_{init}$ be a list containing the number of rows of $\mathbf{X}_k$ for $k = 1, \ldots, K$ (line 3 in Algorithm 11). We first obtain lists $L_{val}$ and $L_{ind}$, sorted values and those corresponding indices, by sorting $L_{init}$ in descending order (line 4 in Algorithm 11). We repeatedly add a slice $\mathbf{X}_k$ to a set $\mathcal{T}_i$ that has the smallest sum. For each $k$, we find the index $t_{min}$ of the minimum in $S$ whose $i$th element corresponds to the sum of row sizes of slices in the $i$th set $\mathcal{T}_i$ (line 6 in Algorithm 11). Then, we add a slice $\mathbf{X}_l$ to the set $\mathcal{T}_{t_{min}}$ where $l$ is equal to $L_{ind}[k]$, and update the list $S$ by $S[t_{min}] \leftarrow S[t_{min}] + L_{val}[k]$ (lines 7 to 9 in Algorithm 11). Note that $S[k]$, $L_{ind}[k]$, and $L_{val}[k]$ denote the $k$th element of $S$, $L_{ind}$, and $L_{val}$, respectively. After obtaining the sets $\mathcal{T}_i$ for $i = 1, .., T$, $i$th thread performs randomized SVD for slices in the set $\mathcal{T}_i$.

After decomposing $\mathbf{X}_k$ for all $k$, we do not need to consider the irregularity for parallelism since there is no computation with $\mathbf{A}_k$ which involves the irregularity. Therefore, we uniformly allocate computations across threads for all $k$ slices. In each iteration (lines 8 to 22 in Algorithm 10), we easily parallelize computations. First, we parallelize the iteration (lines 8 to 10) for all $k$ slices. To update $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$, we need to compute $\mathbf{G}^{(1)}$, $\mathbf{G}^{(2)}$, and $\mathbf{G}^{(3)}$ in parallel. In Lemmas 4.1 and 4.2, DPar2 parallelly computes $\mathbf{W}(k,r)\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)$ and $\mathbf{W}(k,r)\mathbf{F}^{(k)}\mathbf{Z}_k\mathbf{P}_k^T\mathbf{H}(:,r)$ for $k$, respectively. In Lemma 4.3, DPar2 parallelly computes $\left(vec\left(\mathbf{P}_k\mathbf{Z}_k^T\mathbf{F}^{(k)}\right)\right)^T\left(\mathbf{E}\mathbf{D}^T\mathbf{V}(:,r)\otimes\mathbf{H}(:,r)\right)$ for $k$.

### 4.3.7 Complexities

We analyze the time complexity of DPar2.




(a) US stock data (b) KR stock data

Figure 4.7: The length of the temporal dimension of input slices $\mathbf{X}_k$ on US Stock and Korea Stock data. We sort the lengths in descending order.

**Lemma 4.4.** *Compressing input slices takes* $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k JR\right) + JKR^2\right)$ *time.*

*Proof.* The SVD in the first stage takes $\mathcal{O}\left(\sum_{k=1}^{K} I_k JR\right)$ times since computing randomized SVD of $\mathbf{X}_k$ takes $\mathcal{O}(I_k JR)$ time. Then, the SVD in the second stage takes $\mathcal{O}\left(JKR^2\right)$ due to randomized SVD of $\mathbf{M}_{(2)} \in \mathbb{R}^{J \times KR}$. Therefore, the time complexity of the SVD in the two stages is $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k JR\right) + JKR^2\right)$. □

**Lemma 4.5.** *At each iteration, computing* $\mathbf{Y}_k$ *and updating* $\mathbf{H}$, $\mathbf{V}$, *and* $\mathbf{W}$ *takes* $\mathcal{O}(JR^2 + KR^3)$ *time.*

*Proof.* For $\mathbf{Y}_k$, computing $\mathbf{F}^{(k)}\mathbf{E}\mathbf{D}^T\mathbf{V}\mathbf{S}_k\mathbf{H}^T$ and performing SVD of it for all $k$ take $\mathcal{O}(JR^2 + KR^3)$. Updating each of $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ takes $\mathcal{O}(JR^2 + KR^3 + R^3)$ time. Therefore, the complexity for $\mathbf{Y}_k$, $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{W}$ is $\mathcal{O}\left(JR^2 + KR^3\right)$. □

**Theorem 4.1.** *The time complexity of* DPAR2 *is* $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k JR\right) + JKR^2 + MKR^3\right)$ *where* $M$ *is the number of iterations.*

*Proof.* The overall time complexity of DPAR2 is the summation of the compression cost (see Lemma 4.4) and the iteration cost (see Lemma 4.5): $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k JR\right) + JKR^2 + M(JR^2 + KR^3)\right)$. Note that $MJR^2$ term is omitted since it is much smaller than $\left(\sum_{k=1}^{K} I_k JR\right)$ and $JKR^2$. □

Table 4.2: Description of real-world tensor datasets.

| Dataset | Max Dim. $I_k$ | Dim. $J$ | Dim. $K$ | Summary |
|---|---|---|---|---|
| FMA[1] [94] | 704 | 2,049 | 7,997 | music |
| Urban[2] [103] | 174 | 2,049 | 8,455 | urban sound |
| US Stock[3] | 7,883 | 88 | 4,742 | stock |
| Korea Stock[4] [25] | 5,270 | 88 | 3,664 | stock |
| Activity[5] [104, 105] | 553 | 570 | 320 | video feature |
| Action[5] [104, 105] | 936 | 570 | 567 | video feature |
| Traffic[6] [96] | 2,033 | 96 | 1,084 | traffic |
| PEMS-SF[7] | 963 | 144 | 440 | traffic |

**Theorem 4.2.** *The size of preprocessed data of DPAR2 is* $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k R\right) + KR^2 + JR\right)$.

*Proof.* The size of preprocessed data of DPAR2 is proportional to the size of $\mathbf{E}$, $\mathbf{D}$, $\mathbf{A}_k$, and $\mathbf{F}^{(k)}$ for $k = 1, \ldots, K$. The size of $\mathbf{E}$ and $\mathbf{D}$ is $R$ and $J \times R$, respectively. For each $k$, the size of $\mathbf{A}$ and $\mathbf{F}$ is $I_k \times R$ and $R \times R$, respectively. Therefore, the size of preprocessed data of DPAR2 is $\mathcal{O}\left(\left(\sum_{k=1}^{K} I_k R\right) + KR^2 + JR\right)$. □

## 4.4 Experiments

In this section, we experimentally evaluate the performance of DPAR2. We answer the following questions:

Q1 **Performance (Section 4.4.2).** How quickly and accurately does DPAR2 perform PARAFAC2 decomposition compared to other methods?

Q2 **Data Scalability (Section 4.4.3).** How well does DPAR2 scale up with respect to tensor size and target rank?

Q3 **Multi-core Scalability (Section 4.4.4).** How much does the number of threads affect the running time of DPAR2?

Q4 **Discovery (Section 4.4.5).** What can we discover from real-world tensors using DPAR2?

### 4.4.1 Experimental Settings

We describe experimental settings for the datasets, competitors, parameters, and environments.

**Machine.** We use a workstation with 2 CPUs (Intel Xeon E5-2630 v4 @ 2.2GHz), each of which has 10 cores, and 512GB memory for the experiments.

**Real-world Data.** We evaluate the performance of DPar2 and competitors on real-world datasets summarized in Table 4.2. FMA dataset[1] [94] is the collection of songs. Urban Sound dataset[2] [103] is the collection of urban sounds such as drilling, siren, and street music. For the two datasets, we convert each time series into an image of a log-power spectrogram so that their forms are (time, frequency, song; value) and (time, frequency, sound; value), respectively. US Stock dataset[3] is the collection of stocks on the US stock market. Korea Stock dataset[4] [25] is the collection of stocks on the South Korea stock market. Each stock is represented as a matrix of (date, feature) where the feature dimension includes 5 basic features and 83 technical indicators. The basic features collected daily are the opening, the closing, the highest, and the lowest prices and trading volume, and technical indicators are calculated based on the basic features. The two stock datasets have the form of (time, feature, stock; value). Activity data[5] and Action data[5] are the collection of features for motion videos. The two datasets have the form of (frame, feature, video; value). We refer the reader to [104] for their feature extraction. Traffic data[6] is the collection of traffic volume around Melbourne, and its form is (sensor, frequency, time; measurement). PEMS-

[1] https://github.com/mdeff/fma
[2] https://urbansounddataset.weebly.com/urbansound8k.html
[3] https://datalab.snu.ac.kr/dpar2
[4] https://github.com/jungijang/KoreaStockData
[5] https://github.com/titu1994/MLSTM-FCN
[6] https://github.com/florinsch/BigTrafficData

SF data[7] contain the occupancy rate of different car lanes of San Francisco bay area freeways: (station, timestamp, day; measurement). Traffic data and PEMS-SF data are 3-order regular tensors, but we can analyze them using PARAFAC2 decomposition approaches.

**Synthetic Data.** We evaluate the scalability of DPar2 and competitors on synthetic tensors. Given the number $K$ of slices, and the slice sizes $I$ and $J$, we generate a synthetic tensor using *tenrand(I, J, K)* function in Tensor Toolbox [97], which randomly generates a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. We construct a tensor $\{\mathbf{X}_k\}_{k=1}^{K}$ where $\mathbf{X}_k$ is equal to $\mathcal{X}(:,:,k)$ for $k = 1, ...K$.

**Competitors.** We compare DPar2 with PARAFAC2 decomposition methods based on ALS. All the methods including DPar2 are implemented in MATLAB (R2020b).

- **DPar2**: the proposed PARAFAC2 decomposition model which preprocesses a given irregular dense tensor and updates factor matrices using the preprocessing result.
- **RD-ALS [33]**: PARAFAC2 decomposition which preprocesses a given irregular tensor. Since there is no public code, we implement it using Tensor Toolbox [97] based on its paper [33].
- **PARAFAC2-ALS**: PARAFAC2 decomposition based on ALS approach. It is implemented based on Algorithm 2 using Tensor Toolbox [97].
- **SPARTan [2]**: fast and scalable PARAFAC2 decomposition for irregular sparse tensors. Although it targets on sparse irregular tensors, it can be adapted to irregular dense tensors. We use the code implemented by authors[8].

**Parameters.** We use the following parameters.

- **Number of threads:** we use 6 threads except in Section 4.4.4.

[7] http://www.timeseriesclassification.com/
[8] https://github.com/kperros/SPARTan

- **Max number of iterations:** the maximum number of iterations is set to 32.
- **Rank:** we set the target rank $R$ to 10 except in the trade-off experiments of Section 4.4.2 and Section 4.4.4. We also set the rank of randomized SVD to 10 which is the same as the target rank $R$ of PARAFAC2 decomposition.

To compare running time, we run each method 5 times, and report the average.

**Fitness.** We evaluate the fitness defined as follows:

$$1 - \left( \frac{\sum_{k=1}^{K} \|\mathbf{X}_k - \hat{\mathbf{X}}_k\|_{\mathbf{F}}^2}{\sum_{k=1}^{K} \|\mathbf{X}_k\|_{\mathbf{F}}^2} \right)$$

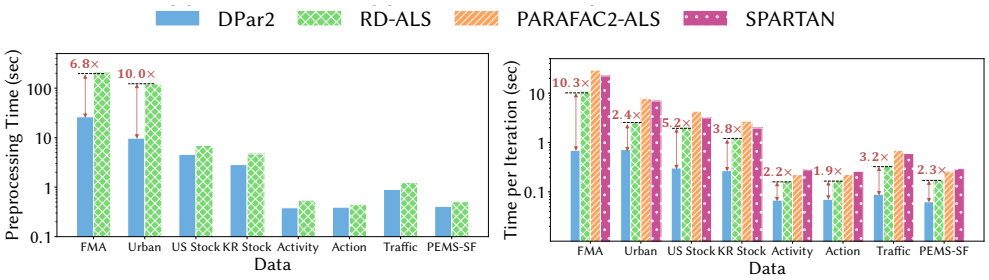
where $\mathbf{X}_k$ is the $k$-th input slice and $\hat{\mathbf{X}}_k$ is the $k$-th reconstructed slice of PARAFAC2 decomposition. Fitness close to 1 indicates that a model approximates a given input tensor well.

### 4.4.2 Performance

We evaluate the fitness and the running time of DPar2, RD-ALS, SPARTan, and PARAFAC2-ALS.

**Trade-off.** Figure 4.1 shows that DPar2 provides the best trade-off of running time and fitness on real-world irregular tensors for the three target ranks: 10, 15, and 20. DPar2 achieves 6.0× faster running time than the competitors for FMA dataset while having a comparable fitness. In addition, DPar2 provides at least 1.5× faster running times than the competitors for the other datasets. The performance gap is large for FMA and Urban datasets whose sizes are larger than those of the other datasets. It implies that DPar2 is more scalable than the competitors in terms of tensor sizes.

**Preprocessing time.** We compare DPar2 with RD-ALS and exclude SPARTan




(a) Preprocessing time

(b) Iteration time

Figure 4.8: **[Best viewed in color]** (a) DPar2 efficiently preprocesses a given irregular dense tensor, which is up to 10× faster compared to RD-ALS. (b) At each iteration, DPar2 runs by up to 10.3× faster than the second-best method.

and PARAFAC2-ALS since only RD-ALS has a preprocessing step. As shown in Figure 4.8(a), DPar2 is up to 10× faster than RD-ALS. There is a large performance gap on FMA and Urban datasets since RD-ALS cannot avoid the overheads for the large tensors. RD-ALS performs SVD of the concatenated slice matrices $\|_{k=1}^{K} \mathbf{X}_k^T$, which leads to its slow preprocessing time.

**Iteration time.** Figure 4.8(b) shows that DPar2 outperforms competitors for running time at each iteration. Compared to SPARTan and PARAFAC2-ALS, DPar2 significantly reduces the running time per iteration due to the small size of the preprocessed results. Although RD-ALS reduces the computational cost at each iteration by preprocessing a given tensor, DPar2 is up to 10.3× faster than RD-ALS. Compared to RD-ALS that computes the variation of $\left(\sum_{k=1}^{K} \|\mathbf{X}_k - \mathbf{Q}_k \mathbf{H} \mathbf{S}_k \mathbf{V}^T\|_{\mathbf{F}}^2\right)$ for the convergence criterion, DPar2 efficiently verifies the convergence by computing the variation of $\sum_{k=1}^{K} \|\mathbf{P}_k \mathbf{Z}_k^T \mathbf{F}^{(k)} \mathbf{E} \mathbf{D}^T - \mathbf{H} \mathbf{S}_k \mathbf{V}^T\|_{\mathbf{F}}^2$, which affects the running time at each iteration. In summary, DPar2 obtains $\mathbf{U}_k$, $\mathbf{S}_k$, and $\mathbf{V}$ in a reasonable running time even if the number of iterations increases.

**Size of preprocessed data.** We measure the size of preprocessed data on real-world datasets. For PARAFAC2-ALS and SPARTan, we report the size of an input

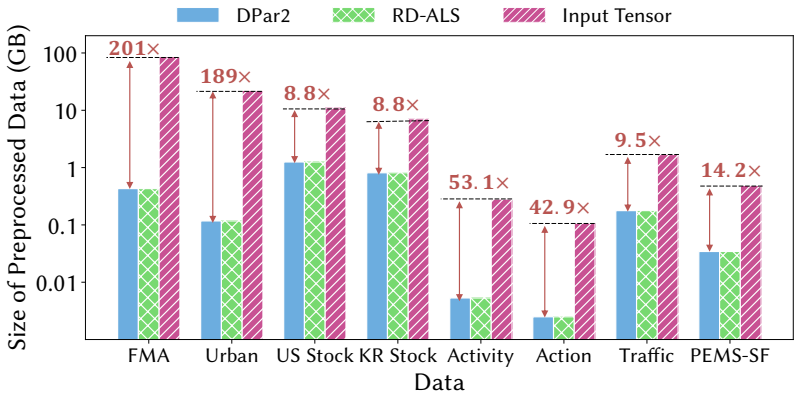


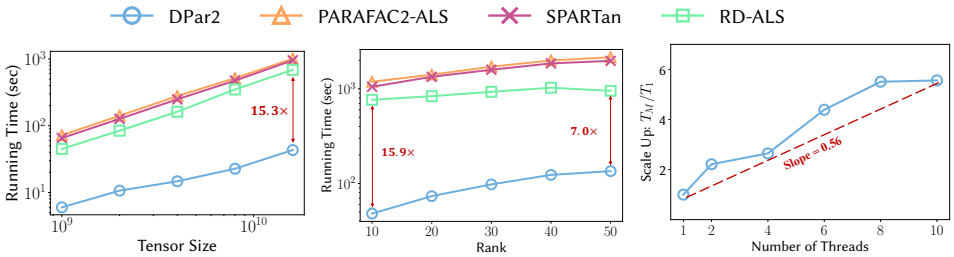

Figure 4.9: The size of preprocessed data. DPar2 generates up to 201× smaller preprocessed data than input tensors used for SPARTan and PARAFAC2-ALS.

irregular tensor since they have no preprocessing step. Compared to an input irregular tensor, DPar2 generates much smaller preprocessed data by up to 201 times as shown in Figure 4.9. Given input slices $\mathbf{X}_k$ of size $I_k \times J$, the compression ratio increases as the number $J$ of columns increases; the compression ratio is larger on FMA, Urban, Activity, and Action datasets than on US Stock, KR Stock, Traffic, and PEMS-SF. This is because the compression ratio is proportional to $\frac{\text{Size of an irregular tensor}}{\text{Size of the preprocessed results}} \approx \frac{IJK}{IKR+KR^2+JR} = \frac{1}{R/J+R^2/IJ+R/IK}$ assuming $I_1 = \ldots = I_K = I$; $R/J$ is the dominant term which is much larger than $R^2/IJ$ and $R/IK$.

### 4.4.3 Data Scalability

We evaluate the data scalability of DPar2 by measuring the running time on several synthetic datasets. We first compare the performance of DPar2 and the competitors by increasing the size of an irregular tensor. Then, we measure the running time by changing a target rank.

**Tensor Size.** To evaluate the scalability with respect to the tensor size, we generate 5 synthetic tensors of the following sizes $I \times J \times K$: $\{1000 \times 1000 \times 1000, 1000 \times 1000 \times 2000, 2000 \times 1000 \times 2000, 2000 \times 2000 \times 2000, 2000 \times 2000 \times 4000\}$. For sim-




(a) Scalability for tensor size (b) Scalability for rank (c) Machine Scalability

Figure 4.10: Data scalability. DPar2 is more scalable than other PARAFAC2 decomposition methods in terms of both tensor size and rank. (a) DPar2 is 15.3× faster than the second-fastest method on the irregular dense tensor of the total size $1.6 \times 10^{10}$. (b) DPar2 is 7.0× faster than the second-fastest method even when a high target rank is given. (c) Multi-core scalability with respect to the number of threads. $T_M$ indicates the running time of DPar2 on the number $M$ of threads. DPar2 gives near-linear scalability, and accelerates 5.5× when the number of threads increases from 1 to 10.

plicity, we set $I_1 = \cdots = I_K = I$. Figure 4.10(a) shows that DPar2 is up to 15.3× faster than competitors on all synthetic tensors; in addition, the slope of DPar2 is lower than that of competitors. We also note that only DPar2 obtains factor matrices of PARAFAC2 decomposition within a minute for all the datasets.

**Rank.** To evaluate the scalability with respect to rank, we generate the following synthetic data: $I_1 = \cdots = I_K = 2,000$, $J = 2,000$, and $K = 4,000$. Given the synthetic tensors, we measure the running time for 5 target ranks: 10, 20, 30, 40, and 50. DPar2 is up to 15.9× faster than the second-fastest method with respect to rank in Figure 4.10(b). For higher ranks, the performance gap slightly decreases since DPar2 depends on the performance of randomized SVD which is designed for a low target rank. Still, DPar2 is up to 7.0× faster than competitors with respect to the highest rank used in our experiment.

### 4.4.4 Multi-core Scalability

We generate the following synthetic data: $I_1 = \cdots = I_K = 2,000$, $J = 2,000$, and $K = 4,000$, and evaluate the multi-core scalability of DPar2 with respect to the number of threads: $1, 2, 4, 6, 8$, and 10. $T_M$ indicates the running time when using the number $M$ of threads. As shown in Figure 4.10(c), DPar2 gives near-linear scalability, and accelerates $5.5\times$ when the number of threads increases from 1 to 10.

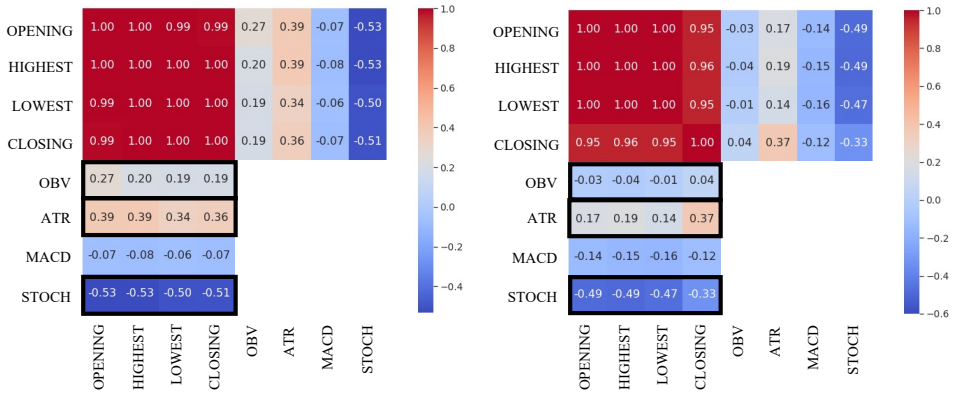
### 4.4.5 Discoveries

We discover various patterns using DPar2 on real-world datasets.

#### 4.4.5.1 Feature Similarity on Stock Dataset

We measure the similarities between features on US Stock and Korea Stock datasets, and compare the results. We compute Pearson Correlation Coefficient (PCC) between $\mathbf{V}(i,:)$, which represents a latent vector of the $i$th feature. For effective visualization, we select 4 price features (the opening, the closing, the highest, and the lowest prices), and 4 representative technical indicators described as follows:

- **OBV (On Balance Volume)**: a technical indicator for cumulative trading volume. If today's closing price is higher than yesterday's price, OBV increases by the amount of today's volume. If not, OBV decreases by the amount of today's volume.
- **ATR (Average True Range)**: a technical indicator for volatility developed by J. Welles Wilder, Jr. It increases in high volatility while decreasing in low volatility.
- **MACD (Moving Average Convergence and Divergence)**: a technical indicator for trend developed by Gerald Appel. It indicates the difference between




(a) US stock data (b) Korea stock data

Figure 4.11: The similarity patterns of features are different on the two stock markets. (a) For US Stock data, ATR and OBV have a positive correlation with the price features. (b) For Korea Stock data, they are uncorrelated with the price features in general.

long-term and short-term exponential moving averages (EMA).

- **STOCH (Stochastic Oscillator)**: a technical indicator for momentum developed by George Lane. It indicates the position of the current closing price compared to the highest and the lowest prices in a duration.

Figures 4.11(a) and 4.11(b) show correlation heatmaps for US Stock data and Korea Stock data, respectively. We analyze correlation patterns between price features and technical indicators. On both datasets, STOCH has a negative correlation and MACD has a weak correlation with the price features. On the other hand, OBV and ATR indicators have different patterns on the two datasets. On the US stock dataset, ATR and OBV have a positive correlation with the price features. On the Korea stock dataset, OBV has little correlation with the price features. Also, ATR has little correlation with the price features except for the closing price. These different patterns are due to the difference of the two markets in terms of market size, market stability, tax, investment behavior, etc.

#### 4.4.5.2 Finding Similar Stocks

*On US Stock dataset, which stock is similar to a target stock $s_T$ in a time range that a user is curious about?* In this section, we provide analysis by setting the target stock $s_T$ to *Microsoft* (Ticker: MSFT), and the range a duration when COVID-19 was very active (Jan. 2, 2020 - Apr. 15, 2021). We efficiently answer the question by 1) constructing the tensor included in the range, 2) obtaining factor matrices with DPAR2, and 3) post-processing the factor matrices of DPAR2. Since $\mathbf{U}_k$ represents temporal latent vectors of the $k$th stock, the similarity $sim(s_i, s_j)$ between stocks $s_i$ and $s_j$ is computed as follows:

$$sim(s_i, s_j) = \exp\left(-\gamma \|\mathbf{U}_{s_i} - \mathbf{U}_{s_j}\|_F^2\right) \tag{4.7}$$

where exp is the exponential function. We set $\gamma$ to 0.01 in this section. Note that we use only the stocks that have the same target range since $\mathbf{U}_{s_i} - \mathbf{U}_{s_j}$ is defined only when the two matrices are of the same size.

Based on $sim(s_i, s_j)$, we find similar stocks to $s_T$ using two different techniques: 1) $k$-nearest neighbors, and 2) Random Walks with Restarts (RWR). The first approach simply finds stocks similar to the target stock, while the second one finds similar stocks by considering the multi-faceted relationship between stocks.

**$k$-nearest neighbors.** We compute $sim(s_T, s_j)$ for $j = 1, \ldots, K$ where $K$ is the number of stocks to be compared, and find top-10 similar stocks to $s_T$, *Microsoft* (Ticker: MSFT). In Table 4.3(a), the *Microsoft* stock is similar to stocks of the Technology sector or with a large capitalization (e.g., Amazon.com, Apple, and Alphabet) during the COVID-19. Moody's is also similar to the target stock.

**Random Walks with Restarts (RWR).** We find similar stocks using another

Table 4.3: Based on the results of DPar2, we find similar stocks to Microsoft (MSFT) during COVID-19. (a) Top-10 stocks from $k$-nearest neighbors. (b) Top-10 stocks from RWR. The blue color refers to the stocks that appear only in one of the two approaches among the top-10 stocks.

(a) Similarity based Result

| Rank | Stock Name | Sector |
|---|---|---|
| 1 | Adobe | Technology |
| 2 | Amazon.com | Consumer Cyclical |
| 3 | Apple | Technology |
| 4 | Moody's | Financial Services |
| 5 | **Intuit** | Technology |
| 6 | ANSYS | Technology |
| 7 | Synopsys | Technology |
| 8 | **Alphabet** | Communication Services |
| 9 | **ServiceNow** | Technology |
| 10 | **EPAM Systems** | Technology |

(b) RWR Result

| Rank | Stock Name | Sector |
|---|---|---|
| 1 | Synopsys | Technology |
| 2 | ANSYS | Technology |
| 3 | Adobe | Technology |
| 4 | Amazon.com | Consumer Cyclical |
| 5 | **Netflix** | Communication Services |
| 6 | **Autodesk** | Technology |
| 7 | Apple | Technology |
| 8 | Moody's | Financial Services |
| 9 | **NVIDIA** | Technology |
| 10 | **S&P Global** | Financial Services |

approach, Random Walks with Restarts (RWR) [106, 107, 108, 109, 110, 111]. To exploit RWR, we first a similarity graph based on the similarities between stocks. The elements of the adjacency matrix $\mathbf{A}$ of the graph are defined as follows:

$$\mathbf{A}(i,j) = \begin{cases} sim(s_i, s_j) & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases} \tag{4.8}$$

We ignore self-loops by setting $\mathbf{A}(i,i)$ to 0 for $i = 1, \ldots, K$.

After constructing the graph, we find similar stocks using RWR. The scores $\mathbf{r}$ is computed by using the power iteration [112] as described in [111]:

$$\mathbf{r}^{(i)} \leftarrow (1-c)\tilde{\mathbf{A}}^T \mathbf{r}^{(i-1)} + c\mathbf{q} \tag{4.9}$$

where $\tilde{\mathbf{A}}$ is the row-normalized adjacency matrix, $\mathbf{r}^{(i)}$ is the score vector at the $i$th iteration, $c$ is a restart probability, and $\mathbf{q}$ is a query vector. We set $c$ to 0.15, the maximum iteration to 100, and $\mathbf{q}$ to the one-hot vector where the element corresponding to Microsoft is 1, and the others are 0.

As shown in Table 4.3, the common pattern of the two approaches is that many stocks among the top-10 belong to the technology sector. There is also a difference. In Table 4.3, the blue color indicates the stocks that appear only in one of the two approaches among the top-10. In Table 4.3(a), the $k$-nearest neighbor approach simply finds the top-10 stocks which are closest to *Microsoft* based on distances. On the other hand, the RWR approach finds the top-10 stocks by considering more complicated relationships. There are 4 stocks appearing only in Table 4.3(b). S&P Global is included since it is very close to Moody's which is ranked 4th in Table 4.3(a). Netflix, Autodesk, and NVIDIA are relatively far from the target stock compared to stocks such as Intuit and Alphabet, but they are included in the top-10 since they are very close to Amazon.com, Adobe, ANSYS, and Synopsys. This difference comes from the fact that the $k$-nearest neighbors approach considers only distances from the target stock while the RWR approach considers distances between other stocks in addition to the target stock.

DPar2 allows us to efficiently obtain factor matrices, and find interesting patterns in data.

## 4.5 Summary

We propose DPar2, a fast and scalable PARAFAC2 decomposition method for irregular dense tensors. By compressing an irregular input tensor, careful reordering of the operations with the compressed results in each iteration, and careful partitioning of input slices, DPar2 successfully achieves high efficiency to perform PARAFAC2 decomposition for irregular dense tensors. Experimental results show that DPar2 is up to 6.0× faster than existing PARAFAC2 decomposition methods while achieving comparable accuracy, and it is scalable with respect to the tensor size and target rank.

With DPar2, we discover interesting patterns in real-world irregular tensors. Future work includes devising an efficient PARAFAC2 decomposition method in a streaming setting.

# Chapter 5

# Efficient Tensor Decomposition for Diverse Time Ranges in Regular Tensors

## 5.1 Motivation

Given a temporal dense tensor and a time range (e.g., January - March 2019), how can we efficiently analyze the tensor in the given time range? Many real-world data including stock data, video data, and traffic volume data are represented as temporal dense tensors. Tensor decomposition has played an important role in various applications including data clustering [60, 61], concept discovery [44, 100, 113], dimensionality reduction [19, 114], anomaly detection [16], and link prediction [115, 116]. Tucker decomposition, one of the tensor decomposition methods, has been recognized as a crucial tool for discovering latent factors and detecting relations between them.

In practice, we analyze a given temporal tensor from various perspectives. Assume a user is interested in investigating patterns of various time ranges using Tucker decomposition. Given a temporal tensor and a user-provided time range (start time and end time) query, our goal is to find the patterns of the temporal tensor at the range using Tucker decomposition. For example, given a temporal tensor including matrices collected between Jan. 1, 2008 to May 6, 2020, a user may be interested in Tucker decomposition of a subrange between Jan. 1, 2020 to April 30, 2020 (see Figure 5.1). Since Tucker decomposition generates factor matrices and a core tensor to accurately approximate an input tensor, answering time range queries, (i.e., performing

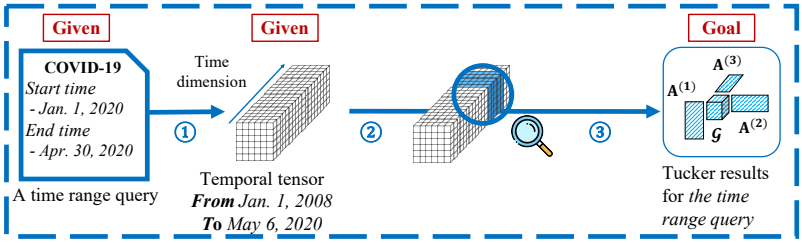
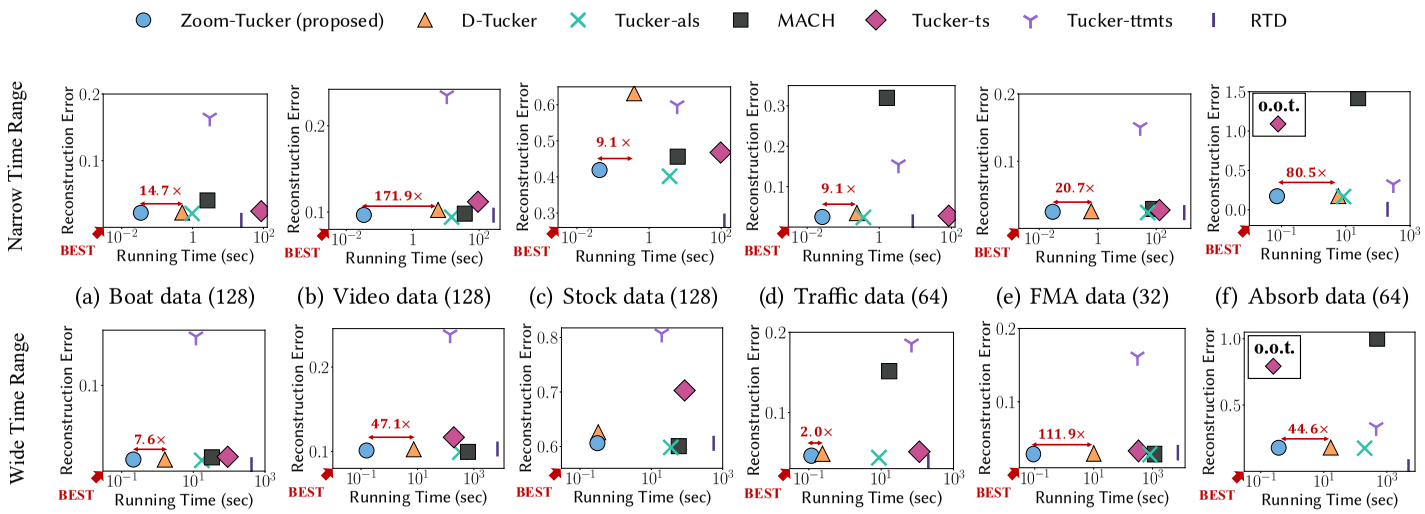



Figure 5.1: Given a temporal tensor and a user-provided time range (start time and end time) query, the goal of the time-ranged Tucker decomposition is to find the patterns of the temporal tensor at the range using Tucker decomposition.

Tucker decomposition of different sub-tensors) yields different Tucker results. However, conventional Tucker decomposition methods [36, 38, 28] based on Alternating Least Square (ALS) is not appropriate for answering diverse time range queries since they target performing Tucker decomposition once for a given tensor; the methods require a high computational cost and large storage space since they need to perform Tucker decomposition of the sub-tensor included in a time range query from scratch, every time the query is given. Due to this limitation, the existing methods are not efficient in exploring diverse time ranges for a given temporal tensor.

A few methods [37, 26] with a preprocessing phase can be adapted to the time range query problem; before the query phase, they preprocess a given tensor, and perform Tucker decomposition with the preprocessed tensor for each time range query. However, they suffer from an accuracy issue for narrow time ranges since preprocessed results are tailored for performing Tucker decomposition of the whole given temporal tensor. The results fail to capture local patterns that appear only in a specific range.




(a) Boat data (128) (b) Video data (128) (c) Stock data (128) (d) Traffic data (64) (e) FMA data (32) (f) Absorb data (64)

(g) Boat data (2048) (h) Video data (2048) (i) Stock data (2048) (j) Traffic data (1024) (k) FMA data (512) (l) Absorb data (1024)

Figure 5.2: Trade-off between query time and reconstruction error of Zoom-Tucker and competitors, for narrow (a-f) and wide (g-l) time range queries. o.o.t.: out of time (takes more than 20,000 seconds). Numbers after the data name represent the length of time ranges; e.g., (128) means the length of a time range is 128 timesteps. Zoom-Tucker is closest to the best point with the fastest query speed and the lowest reconstruction error.

Table 5.1: Symbol description.

| Symbol | Description |
|---|---|
| $\mathcal{X}$ | temporal tensor ($\in I_1 \times ... \times I_N$) |
| $I_n$ & $J_n$ | dimensionality of the $n$-th mode of $\mathcal{X}$ and $\mathcal{G}$ |
| $b$ | block size |
| $t_s$ & $t_e$ | starting and ending points of time range query |
| $[t_s, t_e]$ | time range of a query |
| $\mathcal{X}^{<i>}$ | $i$-th temporal block tensor ($\in I_1 \times ...I_{N-1} \times b$) |
| $(\mathbf{A}^{<i>})^{(k)}$ | $k$-th factor matrix of $i$-th temporal block tensor |
| $\mathcal{G}^{<i>}$ | core tensor of $i$-th temporal block tensor |
| $\tilde{\mathcal{X}}$ | temporal tensor obtained in the time range $[t_s, t_e]$ |
| $\tilde{\mathbf{A}}^{(k)}$ | $k$-th factor matrix of time range query $[t_s, t_e]$ |
| $\tilde{\mathcal{G}}$ | core tensor of time range query $[t_s, t_e]$ |
| $S$ | index of temporal block tensor corresponding to $t_s$ |
| $E$ | index of temporal block tensor corresponding to $t_e$ |
| $\otimes$ | Kronecker product |
| $\times_n$ | $n$-mode product |

In this paper, we propose Zoom-Tucker (Zoomable Tucker decomposition), a fast and memory-efficient Tucker decomposition method to analyze a temporal tensor for diverse time ranges. Zoom-Tucker enables us to discover local patterns in a narrow time range (zoom-in), or global patterns in a wider time range (zoom-out). Zoom-Tucker consists of two phases: the preprocessing phase and the query phase. The preprocessing phase of Zoom-Tucker exploits block structure to lay the groundwork in achieving an efficient query phase and capturing local information. In the query phase, Zoom-Tucker addresses the high computational cost and space cost by elaborately decoupling block results and carefully determining the order of computation. Thanks to these ideas, Zoom-Tucker answers an arbitrary time range query with higher efficiency than existing methods. Through extensive experiments, we demonstrate the effectiveness and efficiency of our method compared to other methods. The main contributions of this paper are as follows:

- **Algorithm.** We propose Zoom-Tucker, a fast and memory-efficient Tucker decomposition method for answering diverse time range queries.

- **Analysis.** We provide both time and space complexities for the preprocessing and query phases of Zoom-Tucker.
- **Experiment.** Experimental results show that Zoom-Tucker answers time range queries up to 171.9× faster and requires up to 230× less space than other methods while providing comparable accuracy, as shown in Figures 5.2 and 5.6.
- **Discovery.** Thanks to Zoom-Tucker, we discover anomalous ranges and trend changes in Stock dataset (Figures 5.9 and 5.10).

In the rest of this chapter, we describe the preliminaries, and formally define the problem in Section 5.2. We then propose our method Zoom-Tucker in Section 5.3, and present experimental results in Section 5.4. Then, we conclude in Section 5.5. The code of our method and datasets are available at https://datalab.snu.ac.kr/zoomtucker.

## 5.2 Problem Definition

We define the problem addressed in this work and the symbols are described in Table 5.1. We describe the formal definition of the time range query problem as follows:

**Problem 1** (Time Range Query on Temporal Tensor)**.**
***Given:*** *a temporal dense tensor* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ *and a time range* $[t_s, t_e]$ *where* $I_N$ *is the length of the time dimension, and* $I_n$ *is the dimensionality of mode-n for* $n = 1, \ldots, N-1$*,*
***Find:*** *the Tucker results of the sub-tensor* $\tilde{\mathcal{X}}$ *of* $\mathcal{X}$ *in the time range* $[t_s, t_e]$ *efficiently. The Tucker result includes factor matrices* $\tilde{\mathbf{A}}^{(1)}$*, ...,* $\tilde{\mathbf{A}}^{(N)}$*, and core tensor* $\tilde{\mathcal{G}}$*.*

To address the time range query problem, a method should efficiently handle various time range queries. Given an arbitrary time range query, existing methods [36, 38, 28] performing Tucker decomposition from scratch requires a high computational cost and large space cost. Compared to the aforementioned methods, Tucker decom-

position methods [26, 37] with a preprocessing phase save time and space costs in that they allow us to compress a whole tensor before a query phase, and then perform Tucker decomposition of a sub-tensor corresponding to a given time range query by exploiting the compressed tensor instead of the input tensor. However, they are still unsatisfactory in terms of time, space, and accuracy for the time range query problem since they are tailored for performing Tucker decomposition of only the whole tensor once.

## 5.3 Proposed Method

In this section, we propose Zoom-Tucker, a novel method for extracting key patterns of a temporal tensor in an arbitrary time range. The following challenges need to be tackled:

C1 **Dealing with various time range queries.** Each user deals with different time ranges or a user analyzes patterns for various time ranges. How can we preprocess a temporal tensor to deal with various time ranges?

C2 **Minimizing computational cost.** Tucker decomposition requires a high computational cost. How can we quickly perform Tucker decomposition for a given time range query?

C3 **Minimizing intermediate data.** Imprudent computation for Tucker decomposition provokes huge intermediate data. How can we avoid generating huge intermediate data?

We address the challenges with the following main ideas:

I1 **Exploiting block structure** enables a query phase to decrease the number of operations and memory requirements while capturing local information.

I2 **Elaborately decoupling block results** decreases the computational cost of

Tucker decomposition for a tensor obtained in a given time range.

I3 **Carefully determining the order of computation** minimizes intermediate data generation while avoiding redundant computation.

Zoom-Tucker efficiently computes Tucker decomposition for various time range queries. Zoom-Tucker consists of two phases: the preprocessing phase and the query phase. The preprocessing phase is computed once for a given temporal tensor while the query phase is computed using the results of the preprocessing phase for each time range query. Zoom-Tucker compresses a given tensor block by block along the time dimension in the preprocessing phase. Zoom-Tucker performs Tucker decomposition for each block. In the query phase, Zoom-Tucker performs Tucker decomposition for each time range query by 1) adjusting the first and the last blocks included in the time range to fit the range and 2) carefully stitching the block results in the time range.

### 5.3.1 Preprocessing Phase

The objective of the preprocessing phase is to manipulate a given temporal tensor for an efficient query phase. In the query phase, performing Tucker decomposition from scratch requires high computational cost and large space cost as the number of queries increases. To avoid it, compressing a given tensor is inevitable to provide fast processing in the query phase. Additionally, we consider that compressed results need to contain local patterns that appear only in specific ranges. The preprocessing phase of existing Tucker decomposition methods [37, 38, 26] fails to support high efficiency of the query phase while maintaining local patterns. Then, how can we compress a given tensor to deal with various time range queries? Our main idea is to exploit a block structure: 1) carefully designating the form of a block, and 2) selecting a

**Algorithm 12:** Preprocessing phase of ZOOM-TUCKER

**Input:** temporal tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N-1} \times I_N}$
**Output:** result sets $\mathcal{C}_n$ for $n = 1, ..., N+1$
**Parameters:** block size $b$

1: compute the number $B = \lceil \frac{I_N}{b} \rceil$ of blocks
2: split $\mathcal{X}$ into block tensors $\mathcal{X}^{<i>} \in \mathbb{R}^{I_1 \times \ldots \times b}$ for $i = 1, ..., B$
3: **for** $i \leftarrow 1$ to $B$ **do**
4: perform Tucker decomposition of $\mathcal{X}_i \approx \mathcal{G}^{<i>} \times_1 (\mathbf{A}^{<i>})^{(1)} \cdots \times_N (\mathbf{A}^{<i>})^{(N)}$
5: store each factor matrices $(\mathbf{A}^{<i>})^{(n)}$ in the results set $\mathcal{C}_n$, for $n = 1, ..., N$
6: store core tensor $\mathcal{G}^{<i>}$ in the result set $\mathcal{C}_{N+1}$
7: **end for**

compression approach for each block. In this paper, we 1) split a given temporal tensor into sub-tensors along the time dimension, and 2) leverage Tucker decomposition for each sub-tensor. The idea allows ZOOM-TUCKER to support an efficient query phase and capture local patterns. Additionally, the preprocessing phase is extensible for new incoming tensors by performing Tucker decomposition of them.

To capture local information, we split a given tensor along the time dimension. Let the reconstruction error at each timestep $t$ be measured by performing Tucker decomposition. The reconstruction error is defined as $\frac{\|\mathcal{X}(t) - \hat{\mathcal{X}}(t)\|_{\mathbf{F}}^2}{\|\mathcal{X}(t)\|_{\mathbf{F}}^2}$ where $\mathcal{X}(t)$ is an input sub-tensor obtained at each timestep $t$ and $\hat{\mathcal{X}}(t)$ is the sub-tensor at timestep $t$ reconstructed from Tucker results. Figure 5.3 shows the reconstruction errors of Stock dataset at each time point. Given a sub-tensor in a range that has relatively high errors, performing Tucker decomposition of the sub-tensor (orange line in Figure 5.3) provides lower errors than the preceding result computed from a whole temporal tensor (blue line in Figure 5.3). This observation implies that decomposing a sub-tensor allows us to capture local information, leading to low errors. Based on the observation, we construct sub-tensors by splitting a temporal tensor along the time dimension and perform Tucker decomposition of each sub-tensor. It provides lower error than performing Tucker decomposition of a whole tensor on all the timesteps,

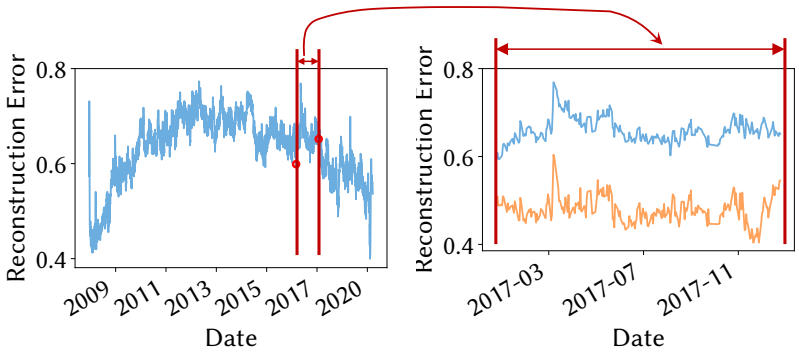


Figure 5.3: Reconstruction errors at each time point on Stock dataset. The blue line presents reconstruction errors computed from a whole temporal tensor, while the orange line describes reconstruction errors computed from a sub-tensor in a range. Performing Tucker decomposition from a sub-tensor provides relatively low reconstruction errors.

by capturing local information.

To support an efficient query phase, we store the Tucker decomposition results of sub-tensors. There are two benefits to leveraging Tucker decomposition in the preprocessing phase: 1) saving the space cost due to the small preprocessed results compared to the given tensor, and 2) enabling the query phase to exploit *the mixed-product property* applicable to mixing matrix multiplication and Kronecker product, i.e., $(\mathbf{A}^T \otimes \mathbf{B}^T)(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A}^T\mathbf{C} \otimes \mathbf{B}^T\mathbf{D})$. Computing $(\mathbf{A}^T\mathbf{C} \otimes \mathbf{B}^T\mathbf{D})$ requires less costs than computing $(\mathbf{A}^T \otimes \mathbf{B}^T)(\mathbf{C} \otimes \mathbf{D})$ when the size of the four matrices is $I \times J$ and $I >> J$. The reason is that the size of $\mathbf{A}^T\mathbf{C}$ and $\mathbf{B}^T\mathbf{D}$ is only $J \times J$ while the size of $(\mathbf{A}^T \otimes \mathbf{B}^T)$ and $(\mathbf{C} \otimes \mathbf{D})$ is $J^2 \times I^2$ and $I^2 \times J^2$, respectively. We further present the exploitation of this property to achieve high efficiency of the query phase in Sections 5.3.2.3 and 5.3.2.4.

Figure 5.4 presents an overview of the preprocessing phase. Without loss of generality, we assume that the temporal mode is the last mode ($N$th mode). We express a given tensor $\mathcal{X}$ as temporal block tensors $\mathcal{X}^{<i>} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{N-1} \times b}$ for $i = 1, \ldots, \lceil \frac{I_N}{b} \rceil$ (line 2 in Algorithm 12) where $b$ is a block size and $I_N$ is the dimensionality of the time dimension. Then, we perform Tucker decomposition for each temporal block

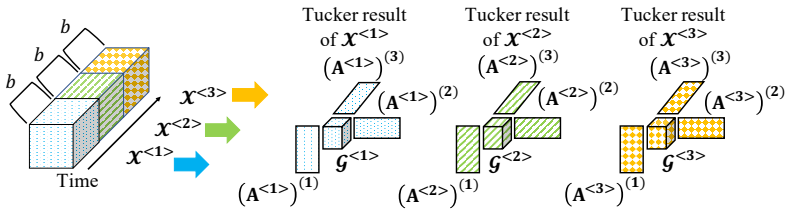



Figure 5.4: Preprocessing phase of Zoom-Tucker. Zoom-Tucker splits a temporal tensor into temporal block tensors along the time dimension. Then, Zoom-Tucker performs Tucker decomposition for each temporal block tensor.

tensor $\mathcal{X}^{<i>}$ (line 4 in Algorithm 12), and store each factor matrix $(\mathbf{A}^{<i>})^{(n)}$ in a set $\mathcal{C}_n$ and the core tensor $\mathcal{G}^{<i>}$ in a set $\mathcal{C}_{N+1}$ (lines 5 and 6 in Algorithm 12). Since the preprocessing phase is computed once and affects errors of the query phase, this phase prefers an accurate but slow Tucker decomposition method rather than a fast but approximate Tucker decomposition one. Specifically, we use Tucker-ALS, which is stable and accurate, in this phase.

### 5.3.2 Query Phase

The objective of the query phase is to efficiently compute Tucker decomposition for a given time range $[t_s, t_e]$. The query phase of Zoom-Tucker operates as follows:

S1. Given a time range $[t_s, t_e]$, we load Tucker results (i.e., $\mathcal{G}^{<i>}$, $(\mathbf{A}^{<i>})^{(n)}$) of temporal block tensors $\mathcal{X}^{<i>}$ for $i = S, \ldots, E$ where $S = \lceil \frac{t_s}{b} \rceil$ and $E = \lceil \frac{t_e}{b} \rceil$ are the indices of the first and the last temporal block tensors including $t_s$ and $t_e$, respectively.

S2. We adjust the Tucker results of $\mathcal{X}^{<S>}$ and $\mathcal{X}^{<E>}$ to fit the range since a part of them may not be within the given range.

S3. Given the Tucker results of $\mathcal{X}^{<i>}$ for $i = S, .., E$ included in the range, Zoom-Tucker updates factor matrices by efficiently stitching the Tucker results.

S4. After that, Zoom-Tucker updates the core tensor using factor matrices updated

at Step S3 and the Tucker results.

S5. Zoom-Tucker repeatedly performs Steps S3 and S4 until convergence.

The most important challenge of the efficient query phase is how to minimize the computational cost for updating factor matrices (Step S3) and the core tensor (Step S4) of the time range while minimizing the intermediate data. To tackle the challenge, our main ideas are to 1) elaborately decouple $\tilde{\mathbf{X}}_{(n)}\left(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)T}\right)$ based on preprocessed results, and 2) carefully determine the order of computation. We first give an objective function and an update rule for the query phase (Section 5.3.2.1). Then, we describe how to achieve high efficiency of Zoom-Tucker in detail (Sections 5.3.2.2 to 5.3.2.4).

#### 5.3.2.1 Objective function and update rule

In the query phase, our goal is to obtain factor matrices $\tilde{\mathbf{A}}^{(1)}$, ..., $\tilde{\mathbf{A}}^{(N)}$, and core tensor $\tilde{\mathcal{G}}$ for a given time range query $[t_s, t_e]$. The query phase of Zoom-Tucker alternately updates factor matrices, and core tensor as in ALS. We minimize the following objective function as mode-$n$ matricized form for a time range $[t_s, t_e]$:

$$L_{(n)} = \|\tilde{\mathbf{X}}_{(n)} - \tilde{\mathbf{A}}^{(n)}\tilde{\mathbf{G}}_{(n)}(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)T})\|_F^2 \tag{5.1}$$

where $\tilde{\mathbf{X}}_{(n)}$ is the mode-$n$ matricized version of a tensor obtained in the time range $[t_s, t_e]$, and $\tilde{\mathbf{G}}_{(n)}$ is the mode-$n$ matricized version of $\tilde{\mathcal{G}}$. From the objective function (5.1), we derive the following update rule for $n$-th factor matrix (see the proof in Section 5.3.4.1):

**Lemma 5.1** (Update rule)**.** *When fixing all but the n-th factor matrix, the following*

**Algorithm 13:** Query phase of Zoom-Tucker

**Input:** a time range $[t_s, t_e]$, and Tucker result sets $\mathcal{C}_n$ for n = $1, ..., N+1$
**Output:** factor matrices $\tilde{\mathbf{A}}^{(n)}$ for $n = 1, .., N$, and core tensor $\tilde{\mathcal{G}}$
**Parameters:** tolerance $\varepsilon$, and block size $b$

1: $S \leftarrow \lceil \frac{t_s}{b} \rceil$ and $E \leftarrow \lceil \frac{t_e}{b} \rceil$
2: load $(\mathbf{A}^{<i>})^{(k)}$ and $\mathcal{G}^{<i>}$ for $i = S, ..., E$ from $\mathcal{C}_k$ for $k = 1, ..., N+1$
3: obtain $(\bar{\mathbf{A}}^{<S>})^{(N)}$ and $(\bar{\mathbf{A}}^{<E>})^{(N)}$ by eliminating the rows of $(\mathbf{A}^{<S>})^{(N)}$ and $(\mathbf{A}^{<E>})^{(N)}$ excluded in the range
4: $(\bar{\mathbf{A}}^{<S>})^{(N)} \rightarrow \mathbf{Q}^{<S>}\mathbf{R}^{<S>}$, $(\bar{\mathbf{A}}^{<E>})^{(N)} \rightarrow \mathbf{Q}^{<E>}\mathbf{R}^{<E>}$
5: $(\mathbf{A}^{<S>})^{(N)} \leftarrow \mathbf{Q}^{<S>}$, $\mathcal{G}^{<S>} \leftarrow \mathcal{G}^{<S>} \times_N \mathbf{R}^{<S>}$, $(\mathbf{A}^{<E>})^{(N)} \leftarrow \mathbf{Q}^{<E>}$, and $\mathcal{G}^{<E>} \leftarrow \mathcal{G}^{<E>} \times_N \mathbf{R}^{<E>}$
6: **repeat**
7: **for** $k = 1 ... N-1$ **do**
8: update $\tilde{\mathbf{A}}^{(k)}$ by computing Equation (5.4) and orthogonalizing it with QR decomposition
9: **end for**
10: update $\tilde{\mathbf{A}}^{(N)}$ by computing Equation (5.6) and orthogonalizing it with QR decomposition
11: update core tensor $\tilde{\mathcal{G}}$ by computing Equation (5.7)
12: **until** the variation of an error is less than $\varepsilon$ or the number of iterations is larger than the maximum number of iterations
13: **return** $\tilde{\mathbf{A}}^{(k)}$ for $k = 1, ..., N$ and $\tilde{\mathcal{G}}$

*update rule for the n-th factor matrix minimizes the objective function* (5.1).

$$\tilde{\mathbf{A}}^{(n)} \leftarrow \tilde{\mathbf{X}}_{(n)} \left( \otimes_{k \neq n}^{N} \tilde{\mathbf{A}}^{(k)} \right) \tilde{\mathbf{G}}_{(n)}^{T} \left( \mathbf{C}^{(n)} \right)^{-1} \tag{5.2}$$

*where* $\mathbf{C}^{(n)} \in \mathbb{R}^{J_n \times J_n}$ *of the n-th mode is given by*

$$\mathbf{C}^{(n)} = \tilde{\mathbf{G}}_{(n)} \left( \otimes_{k \neq n}^{N} \tilde{\mathbf{A}}^{(k)T} \tilde{\mathbf{A}}^{(k)} \right) \tilde{\mathbf{G}}_{(n)}^{T} \qquad \square$$

In contrast to naively computing Equation (5.2) with $\tilde{\mathbf{X}}_{(n)}$, Zoom-Tucker efficiently computes Equation (5.2) by exploiting preprocessed results obtained in the preprocessing phase.

Before describing an efficient update procedure, we introduce a useful lemma (see the proof in Section 5.3.4.2).

**Lemma 5.2.** *Let $\mathcal{S} \in \mathbb{R}^{J\times\ldots\times J}$ and $\mathcal{S}' \in \mathbb{R}^{J\times\ldots\times J}$ be $N$-order tensors, and $\mathbf{U}^{(n)}$ and $\mathbf{V}^{(n)}$ for $n = 1,\ldots,n-1,n+1,\ldots,N$ be matrices of size $I \times J$. Assume our goal is to compute the following equation:*

$$\mathbf{S}_{(n)} \left( \otimes_{k\neq n}^{N} \mathbf{U}^{(k)T} \mathbf{V}^{(k)} \right) \mathbf{S'}_{(n)}^{T} \tag{5.3}$$

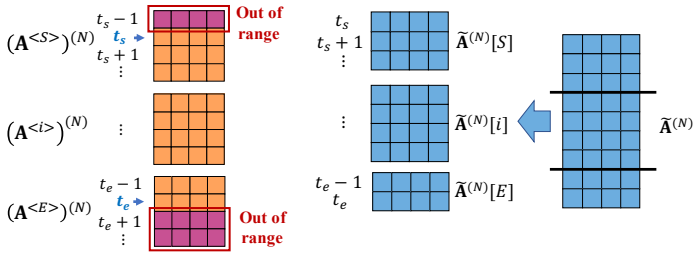
*where $\mathbf{S}_{(n)}$ and $\mathbf{S'}_{(n)}$ are the mode-n matricized version of $\mathcal{S}$ and $\mathcal{S}'$, respectively. Naively computing Equation* (5.3) *by first computing $\otimes_{k\neq n}^{N} \mathbf{U}^{(k)T} \mathbf{V}^{(k)}$ and multiply with the remaining matrices requires $\mathcal{O}(NIJ^2 + J^{2N} + J^{N+1})$ time and $\mathcal{O}(J^{2N} + NIJ)$ space. Instead, exploiting Equation* (2.4) *enables to compute Equation* (5.3) *efficiently: $\mathcal{O}(NIJ^2 + NJ^{N+1})$ time and $\mathcal{O}(J^N + NIJ)$ space.* □

For all $n = 1,\ldots,N$, $\mathbf{C}^{(n)}$ is computed based on Lemma 5.2, by replacing $\mathbf{S}_{(n)}$, $\mathbf{U}^{(k)}$, $\mathbf{V}^{(k)}$, and $\mathbf{S'}_{(n)}$ with $\tilde{\mathbf{G}}_{(n)}$, $\tilde{\mathbf{A}}^{(k)}$, $\tilde{\mathbf{A}}^{(k)}$, and $\tilde{\mathbf{G}}_{(n)}$, respectively.

#### 5.3.2.2 Adjusting edge blocks of time range query (Step S2)

Before updates, we adjust the Tucker results of $\mathcal{X}^{<S>}$ and $\mathcal{X}^{<E>}$, the temporal block tensors corresponding to $t_s$ and $t_e$ of the given time range $[t_s, t_e]$, respectively. The temporal factor matrices $(\mathbf{A}^{<S>})^{(N)}$ of $\mathcal{X}^{<S>}$ and $(\mathbf{A}^{<E>})^{(N)}$ of $\mathcal{X}^{<E>}$ may contain the rows that are not included in the range (see Figure 5.5(a)). To fit to the given time range, we need to remove the non-included rows of $(\mathbf{A}^{<S>})^{(N)}$ and $(\mathbf{A}^{<E>})^{(N)}$, and adjust the Tucker results of $\mathcal{X}^{<S>}$ and $\mathcal{X}^{<E>}$.

Let $p$ be $S$ or $E$. For the temporal factor matrix $(\mathbf{A}^{<p>})^{(N)}$ of $\mathcal{X}^{<p>}$ in the range, Zoom-Tucker obtains the manipulated temporal factor matrix $(\bar{\mathbf{A}}^{<p>})^{(N)}$ by removing the rows of $(\mathbf{A}^{<p>})^{(N)}$ that are not included in the time range (line 3 in Algorithm 13). Next, we perform QR decomposition to make $(\bar{\mathbf{A}}^{<p>})^{(N)}$ maintain column-




(a) Example of adjustment (b) Example of division

Figure 5.5: Examples of adjustment (Section 5.3.2.2) and division (Section 5.3.2.3).

orthogonality (line 4 in Algorithm 13); we use $(\mathbf{Q}^{<p>})^{(N)}$ as the temporal factor matrix of $\mathcal{X}^{<p>}$ and update the core tensor $\mathcal{G}^{<p>} \leftarrow \mathcal{G}^{<p>} \times_N (\mathbf{R}^{<p>})^{(N)}$ where $(\mathbf{Q}^{<p>})^{(N)}$ and $(\mathbf{R}^{<p>})^{(N)}$ are the results of QR decomposition (line 5 in Algorithm 13).

#### 5.3.2.3 Efficient update of factor matrices (Step S3)

We present how to efficiently update the factor matrix of the non-temporal modes and the temporal mode.

**Updating factor matrix of non-temporal modes.** Consider updating the $n$-th factor matrix, which corresponds to a non-temporal mode. A naive approach is to reconstruct $\tilde{\mathbf{X}}_{(n)}$ from the Tucker results of the preprocessing phase and compute Equation (5.2). However, it requires large time and space costs since the reconstructed tensor is much larger than the preprocessed results. Our main ideas are to 1) elaborately decouple $\tilde{\mathbf{X}}_{(n)} \left(\otimes_{k \neq n}^{N} \tilde{\mathbf{A}}^{(k)}\right)$ block by block using the preprocessed results, and 2) carefully determine the order of computations, which significantly reduces time and space costs compared to the naive approach. We derive Equation (5.4) in Lemma 5.3 to update $\tilde{\mathbf{A}}^{(n)}$ (see the proof in Section 5.3.4.3).

**Lemma 5.3** (Updating factor matrix of a non-temporal mode)**.** *Assume that* $\tilde{\mathbf{X}}_{(n)}$ *is replaced with the preprocessed results (i.e.,* $(\mathbf{A}^{<i>})^{(n)}$ *and* $\mathcal{G}^{<i>}$*). Then, the following*

*equation is equal to Equation* (5.2) *in Lemma 5.1 for n-th mode:*

$$\tilde{\mathbf{A}}^{(n)} \leftarrow \sum_{i=S}^{E} (\mathbf{A}^{<i>})^{(n)} (\mathbf{B}^{<i>})^{(n)} \left(\mathbf{C}^{(n)}\right)^{-1} \tag{5.4}$$

*where the i-th block matrix* $(\mathbf{B}^{<i>})^{(n)}$ *of the n-th mode is*

$$(\mathbf{B}^{<i>})^{(n)} = \mathbf{G}_{(n)}^{<i>} \left( (\mathbf{A}^{<i>})^{(N)T} \tilde{\mathbf{A}}^{(N)}[i] \otimes \left( \otimes_{k \neq n}^{N-1} (\mathbf{A}^{<i>})^{(k)T} \tilde{\mathbf{A}}^{(k)} \right) \right) \tilde{\mathbf{G}}_{(n)}^{T}, \tag{5.5}$$

*and* $\mathbf{C}^{(\mathbf{n})}$ *is defined in Lemma 5.1.* $(\mathbf{A}^{<i>})^{(k)}$ *is the k-th factor matrix of the temporal block tensor* $\mathcal{X}^{<i>}$*, and* $\mathbf{G}_{(n)}^{<i>}$ *is the mode-n matricized version of the core tensor of* $\mathcal{X}^{<i>}$*.* $\tilde{\mathbf{A}}^{(N)}[i]$ *is a sub-matrix of the temporal factor matrix* $\tilde{\mathbf{A}}^{(N)}$ *such that,*

$$\begin{bmatrix} \tilde{\mathbf{A}}^{(N)}[S] \\ \vdots \\ \tilde{\mathbf{A}}^{(N)}[E] \end{bmatrix} = \tilde{\mathbf{A}}^{(N)}$$

*To compute* $(\mathbf{A}^{<i>})^{(N)T} \tilde{\mathbf{A}}^{(N)}[i]$*, we split* $\tilde{\mathbf{A}}^{(N)}$ *into sub-matrices* $\tilde{\mathbf{A}}^{(N)}[i]$ $(i = S, \ldots, E)$ *along the time dimension (see Figure 5.5(b)); the size of* $\tilde{\mathbf{A}}^{(N)}[i]$ *for* $i = S+1, \ldots, E-1$ *is* $b \times J_N$*, and that of* $\tilde{\mathbf{A}}^{(N)}[S]$ *and* $\tilde{\mathbf{A}}^{(N)}[E]$ *is* $(b - r_S) \times J_N$ *and* $(b - r_E) \times J_N$*, respectively, where* $r_S$ *and* $r_E$ *are the number of the rows removed with respect to* $t_s$ *and* $t_e$*, respectively.* □

Zoom-Tucker efficiently updates $\tilde{\mathbf{A}}^{(n)}$ with Equation (5.4). Zoom-Tucker minimizes the intermediate data and reduces the high computational cost by independently computing $\mathbf{C}^{(n)}$ and $(\mathbf{B}^{<i>})^{(n)}$ for $i = S, \ldots, E$. Note that $(\mathbf{B}^{<i>})^{(n)}$ for $i = S, \ldots, E$ is computed based on Lemma 5.2, by replacing $\mathbf{S}_{(n)}$, $\mathbf{U}^{(k)}$, $\mathbf{V}^{(k)}$, and $\mathbf{S'}_{(n)}$ with $\mathbf{G}_{(n)}^{<i>}$, $(\mathbf{A}^{<i>})^{(k)}$, $\tilde{\mathbf{A}}^{(k)}$ (or $\tilde{\mathbf{A}}^{(N)}[i]$), and $\tilde{\mathbf{G}}_{(n)}$, respectively. Next, we obtain $\tilde{\mathbf{A}}^{(n)}$ by summing up the results of $(\mathbf{A}^{<i>})^{(n)} (\mathbf{B}^{<i>})^{(n)} \left(\mathbf{C}^{(n)}\right)^{-1}$ for $i = S, \ldots, E$. For orthogonalization, we then update $\tilde{\mathbf{A}}^{(n)} \leftarrow \tilde{\mathbf{Q}}^{(n)}$ after QR decomposition $\tilde{\mathbf{A}}^{(n)} \rightarrow \tilde{\mathbf{Q}}^{(n)} \tilde{\mathbf{R}}^{(n)}$ (line 8 in Algorithm 13).

**Updating factor matrix of temporal mode.** The goal is to update the factor matrix $\tilde{\mathbf{A}}^{(N)}$ of the temporal mode by using the preprocessed results instead of $\tilde{\mathbf{X}}_{(N)}$. Reconstructing $\tilde{\mathbf{X}}_{(N)}$ requires high space and time costs in Equation (5.2). Based on our ideas used for the non-temporal modes, we efficiently update $\tilde{\mathbf{A}}^{(N)}$ by computing Equation (5.6) in Lemma 5.4 (see the proof in Section 5.3.4.4).

**Lemma 5.4** (Updating factor matrix of temporal mode)**.** *Assume that $\tilde{\mathbf{X}}_{(N)}$ is replaced with the preprocessed results (i.e., $(\mathbf{A}^{<i>})^{(n)}$ and $\mathcal{G}^{<i>}$). Then, the following equation is equal to Equation* (5.2) *in Lemma 5.1 for the temporal mode:*

$$\tilde{\mathbf{A}}^{(N)} \leftarrow \begin{bmatrix} (\mathbf{A}^{<S>})^{(N)}(\mathbf{B}^{<S>})^{(N)} \\ \vdots \\ (\mathbf{A}^{<E>})^{(N)}(\mathbf{B}^{<E>})^{(N)} \end{bmatrix} \left(\mathbf{C}^{(N)}\right)^{-1} \tag{5.6}$$

*where the i-th matrix $(\mathbf{B}^{<i>})^{(N)} \in \mathbb{R}^{J_N \times J_N}$ for $i = S, \ldots, E$ is*

$$(\mathbf{B}^{<i>})^{(N)} = \mathbf{G}^{<i>}_{(N)} \left(\otimes_{k=1}^{N-1} (\mathbf{A}^{<i>})^{(k)T} \tilde{\mathbf{A}}^{(k)}\right) \tilde{\mathbf{G}}^{T}_{(N)}$$

*$(\mathbf{A}^{<i>})^{(k)}$ is the k-th factor matrix of $\mathcal{X}^{<i>}$, $\mathbf{G}^{<i>}_{(N)}$ is the mode-N matricized version of the core tensor of $\mathcal{X}^{<i>}$, and $\mathbf{C}^{(N)}$ is equal to $\tilde{\mathbf{G}}_{(N)} \left(\otimes_{k=1}^{N-1} \tilde{\mathbf{A}}^{(k)T} \tilde{\mathbf{A}}^{(k)}\right) \tilde{\mathbf{G}}^{T}_{(N)}$.* □

We obtain $\tilde{\mathbf{A}}^{(N)}$ by using $(\mathbf{C}^{(N)})^{-1}$, $(\mathbf{A}^{<i>})^{(N)}$, and $(\mathbf{B}^{<i>})^{(N)}$ for $i = S, \ldots, E$. Zoom-Tucker efficiently updates $\tilde{\mathbf{A}}^{(N)}$ by independently computing $\mathbf{C}^{(N)}$ and $(\mathbf{B}^{<i>})^{(N)}$ for $i = S, \ldots, E$. $(\mathbf{B}^{<i>})^{(N)}$ is efficiently computed based on Lemma 5.2, by replacing $\mathbf{S}_{(n)}$, $\mathbf{U}^{(k)}$, $\mathbf{V}^{(k)}$, and $\mathbf{S}'_{(n)}$ with $\mathbf{G}^{<i>}_{(N)}$, $(\mathbf{A}^{<i>})^{(k)}$, $\tilde{\mathbf{A}}^{(k)}$, and $\tilde{\mathbf{G}}_{(N)}$, respectively. For orthogonalization, we update $\tilde{\mathbf{A}}^{(N)} \leftarrow \tilde{\mathbf{Q}}^{(N)}$ after QR decomposition $\tilde{\mathbf{A}}^{(N)} \rightarrow \tilde{\mathbf{Q}}^{(N)}\tilde{\mathbf{R}}^{(N)}$ (line 10 in Algorithm 13).

#### 5.3.2.4 Efficient update of core tensor (Step S4).

At the end of each iteration, Zoom-Tucker updates the core tensor using the factor matrices: $\tilde{\mathbf{G}}_{(N)} \leftarrow \tilde{\mathbf{A}}^{(N)T}\tilde{\mathbf{X}}_{(N)}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right)$ (mode-$N$ matricization of line 8 in Algorithm 1). We efficiently compute the core tensor by avoiding reconstruction of $\tilde{\mathbf{X}}_{(N)}$ and carefully determining the order of computation. We replace $\tilde{\mathbf{X}}_{(N)}$ with the preprocessed results and refine the equation with block decoupling and the *mixed-product property* (see Equation (5.9) in Section 5.3.4.4).

$$\tilde{\mathbf{G}}_{(N)} \leftarrow \left(\sum_{i=S}^{E}(\tilde{\mathbf{A}}^{(N)T}[i])(\mathbf{A}^{<i>})^{(N)}\mathbf{G}_{(N)}^{<i>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<i>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\right) \tag{5.7}$$

With Equation (5.7), Zoom-Tucker efficiently updates $\tilde{\mathcal{G}}$, reducing the intermediate data and the computational cost. For each $i$, Zoom-Tucker computes $(\tilde{\mathbf{A}}^{(N)T}[i])(\mathbf{A}^{<i>})^{(N)}$ $\mathbf{G}_{(N)}^{<i>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<i>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)$ after transforming it into $n$-mode products as in Equation (2.4). After that, Zoom-Tucker obtains $\tilde{\mathbf{G}}_{(N)}$ by summing up the results and reshape it to the core tensor $\tilde{\mathcal{G}}$ (line 11 in Algorithm 13).

### 5.3.3 Analysis

We analyze the time and space complexities of Zoom-Tucker in the preprocessing phase and the query phase. We assume that $I = I_1 = \ldots = I_{N-1}$, and $J = J_1 = \ldots = J_N$. $M$ is the number of iterations, $l_{[t_s,t_e]} = t_e - t_s + 1$ is the length of a time range query, $N$ is the order of a given tensor, $I$ is the dimensionality, $b$ is the block size, $B$ is the number of blocks, and $J$ is the rank. All proofs are summarized in Sections 5.3.4.5 to 5.3.4.8.

**Time complexity.** We analyze the computational cost of Zoom-Tucker in the preprocessing phase and the query phase.

**Theorem 5.1.** *The preprocessing phase takes* $\mathcal{O}(MNI^{N-1}JbB)$ *time.*

Table 5.2: Time and space complexities of Zoom-Tucker and other methods for a time range $[t_s, t_e]$. The optimal complexities are in bold. $I$, $J$, $M$, $N$, and $l_{[t_s,t_e]}$ are described in Section 5.3.3. $S$ is a sampling rate for MACH.

| Algorithm | Time | Space |
|---|---|---|
| Zoom-Tucker | $\mathbf{\mathcal{O}(l_{[t_s,t_e]}IMN^2J^2/b)}$ | $\mathbf{\mathcal{O}(l_{[t_s,t_e]}NIJ/b)}$ |
| D-Tucker [26] | $\mathcal{O}(l_{[t_s,t_e]}I^{N-2}MNJ^2)$ | $\mathcal{O}(l_{[t_s,t_e]}I^{N-2}J)$ |
| Tucker-ALS | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}MNJ)$ | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1})$ |
| MACH [37] | $\mathcal{O}(Sl_{[t_s,t_e]}I^{N-1}MNJ)$ | $\mathcal{O}(Sl_{[t_s,t_e]}I^{N-1})$ |
| RTD [36] | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}MN)$ | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1})$ |
| Tucker-ts [38] | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}N+MNIJ^N)$ | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}+NIJ^N)$ |
| Tucker-ttmts [38] | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}N+MNIJ^{2N-2})$ | $\mathcal{O}(l_{[t_s,t_e]}I^{N-1}+NIJ^N)$ |

**Theorem 5.2.** *Given a time range query $[t_s, t_e]$, the query phase of Zoom-Tucker takes $\mathcal{O}\left(MNJ^2 l_{[t_s,t_e]}\left(1+\frac{NI}{b}+\frac{NJ^{N-1}}{b}\right)\right)$ time.* □

**Space complexity.** We provide analysis for the space cost of Zoom-Tucker in the preprocessing phase and the query phase.

**Theorem 5.3.** *Zoom-Tucker requires $\mathcal{O}\left(NIJ(\lceil\frac{I_N}{b}\rceil)+I_NJ\right)$ space to store the Tucker results in the preprocessing phase.* □

**Theorem 5.4.** *Given a time range query $[t_s, t_e]$, Zoom-Tucker requires $\mathcal{O}\Big(NIJ(\lceil\frac{l_{[t_s,t_e]}}{b}\rceil)$ $+Jl_{[t_s,t_e]}\Big)$ space in the query phase.* □

Table 5.2 shows the time and space complexities of Zoom-Tucker and competitors for a given time range query $[t_s, t_e]$. The time and space complexities of Zoom-Tucker mainly depend on $I$ and $l_{[t_s,t_e]}$. We also note that the block size $b$ reduces the complexities of Zoom-Tucker. We compare the time and space complexities of Zoom-Tucker with those of the second-best method, D-Tucker. For both time and space complexities, the result of dividing the complexity of Zoom-Tucker by that of D-Tucker is $\frac{N}{I^{N-3}b}$. Zoom-Tucker has better time and space complexities than D-Tucker since $I^{N-3}b$ is larger than $N$ in real-world datasets; for example, in the experiments, we use 50 as the default block size $b$ while the order of the real-world datasets

is 3 or 4. As $b$ increases, the space complexity of the preprocessing and the query phases, and the time complexity of the query phase decrease; however, a large block size $b$ can provoke a high reconstruction error for a narrow time range query since the preprocessing phase with the large $b$ cannot capture local information. In Section 5.4.5, we experimentally find a block size that enables the preprocessing phase to capture local information with low reconstruction errors for narrow time range queries.

### 5.3.4 Proofs of Lemmas and Theorems

#### 5.3.4.1 Proof of Lemma 5.1

*Proof.* After fixing all factor matrices except for the $n$-th factor matrix, the partial derivative of the Equation (5.1) with respect to the factor matrix $\tilde{\mathbf{A}}^{(n)}$ is as follows:

$$\frac{\partial L_{(n)}}{\partial \tilde{\mathbf{A}}^{(n)}} = -2\tilde{\mathbf{X}}_{(n)}(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)})\tilde{\mathbf{G}}_{(n)}^{T} + 2\tilde{\mathbf{A}}^{(n)}\tilde{\mathbf{G}}_{(n)}\left(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(n)}^{T}$$

We set $\frac{\partial L_{(n)}}{\partial \tilde{\mathbf{A}}^{(n)}}$ to zero, and solve the equation with respect to the factor matrix $\tilde{\mathbf{A}}^{(n)}$:

$$\begin{aligned}
&\tilde{\mathbf{A}}^{(n)}\left(\tilde{\mathbf{G}}_{(n)}\left(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(n)}^{T}\right) = \tilde{\mathbf{X}}_{(n)}\left(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(n)}^{T} \\
&\Leftrightarrow \tilde{\mathbf{A}}^{(n)} = \tilde{\mathbf{X}}_{(n)}\left(\otimes_{k\neq n}^{N}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(n)}^{T}\left(\mathbf{C}^{(n)}\right)^{-1}
\end{aligned}$$

□

#### 5.3.4.2 Proof of Lemma 5.2

*Proof.* A naive approach computing Equation (5.3) is to explicitly compute the entire Kronecker product $\left(\otimes_{k\neq n}^{N}\mathbf{U}^{(k)T}\mathbf{V}^{(k)}\right)$ of the size $J^{N-1} \times J^{N-1}$. We compute matrix

multiplication between the preceding result $\mathbf{S}_{(n)}$ and $\mathbf{S}'_{(n)}$. Therefore, the time and space complexities are $\mathcal{O}(NIJ^2 + J^{2N} + J^{N+1})$ and $\mathcal{O}(J^{2N} + NIJ)$, respectively.

We compute Equation (5.3) using $n$-mode product instead of Kronecker product. Let $\mathbf{Z}_{(n)} = \mathbf{S}_{(n)} \left(\otimes_{k\neq n}^{N} \mathbf{U}^{(k)T}\mathbf{V}^{(k)}\right)$ be equal to $\mathbf{I}^{(n)}\mathbf{S}_{(n)} \left(\otimes_{k\neq n}^{N} \mathbf{U}^{(k)T}\mathbf{V}^{(k)}\right)$ where $\mathbf{I}^{(n)} \in \mathbb{R}^{J\times J}$ is an identity matrix. Then, we transform $\mathbf{Z}$ into Equation (5.8) using Equation (2.4).

$$\begin{aligned} \mathcal{Z} = \mathcal{S} \times_1 (\mathbf{U}^{(1)T}\mathbf{V}^{(1)})^T \cdots \times_{n-1} (\mathbf{U}^{(n-1)T}\mathbf{V}^{(n-1)})^T \\ \times_n \mathbf{I}^{(n)} \times_{n+1} (\mathbf{U}^{(n+1)T}\mathbf{V}^{(n+1)})^T \cdots \times_N (\mathbf{U}^{(N)T}\mathbf{V}^{(N)})^T \end{aligned} \tag{5.8}$$

Based on Equation (5.8), we compute Equation (5.3) in the following order: 1) $\mathbf{U}^{(k)T}\mathbf{V}^{(k)}$ for $k = 1, ..., n-1, n+1, ..., N$, 2) $\mathbf{Z}_{(n)}$, and 3) $\mathbf{Z}_{(n)}\mathbf{S}'^{T}_{(n)}$. Therefore, the computational cost is $\mathcal{O}(NIJ^2 + NJ^{N+1})$. In addition, the size of intermediate data is always no larger than $J^N$ so that the space complexity is $\mathcal{O}(J^N + NIJ)$. □

#### 5.3.4.3 Proof of Lemma 5.3

*Proof.* From Equation (5.2), we carefully decouple $\tilde{\mathbf{X}}_{(n)} \left(\otimes_{k\neq n}^{N} \tilde{\mathbf{A}}^{(k)}\right)$ block by block so that we represent the term as a summation of block matrices:

$$\begin{aligned} \tilde{\mathbf{A}}^{(n)} &= \left[\mathbf{X}_{(n)}^{<S>} \cdots \mathbf{X}_{(n)}^{<E>}\right] \left( \begin{bmatrix} \tilde{\mathbf{A}}^{(N)}[S] \\ \vdots \\ \tilde{\mathbf{A}}^{(N)}[E] \end{bmatrix} \otimes \left(\otimes_{k\neq n}^{N-1} \tilde{\mathbf{A}}^{(k)}\right) \right) \tilde{\mathbf{G}}_{(n)}^{T} \left(\mathbf{C}^{(n)}\right)^{-1} \\ &= \left( \sum_{i=S}^{E} \mathbf{X}_{(n)}^{<i>} \left(\tilde{\mathbf{A}}^{(N)}[i] \otimes \left(\otimes_{k\neq n}^{N-1} \tilde{\mathbf{A}}^{(k)}\right)\right) \right) \tilde{\mathbf{G}}_{(n)}^{T} \left(\mathbf{C}^{(n)}\right)^{-1} \end{aligned}$$

Next, we express $i$-th block matrix $\mathbf{X}_{(n)}^{<i>}$ as the result $(\mathbf{A}^{<i>})^{(n)}\mathbf{G}_{(n)}^{<i>}\left((\mathbf{A}^{<i>})^{(N)T} \otimes \left(\otimes_{k\neq n}^{N-1}(\mathbf{A}^{<i>})^{(k)T}\right)\right)$ obtained in the preprocessing step.

$$\tilde{\mathbf{A}}^{(n)} = \sum_{i=S}^{E}(\mathbf{A}^{<i>})^{(n)}\mathbf{G}_{(n)}^{<i>}\left((\mathbf{A}^{<i>})^{(N)T}\tilde{\mathbf{A}}^{(N)}[i] \otimes \left(\otimes_{k\neq n}^{N-1}(\mathbf{A}^{<i>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\right)$$
$$\times \tilde{\mathbf{G}}_{(n)}^{T}\left(\mathbf{C}^{(n)}\right)^{-1} = \left(\sum_{i=S}^{E}(\mathbf{A}^{<i>})^{(n)}(\mathbf{B}^{<i>})^{(n)}\left(\mathbf{C}^{(n)}\right)^{-1}\right)$$

Note that $\tilde{\mathbf{A}}^{(N)}[i]$ is described in Lemma 5.4. □

#### 5.3.4.4 Proof of Lemma 5.4

*Proof.* From Equation (5.2), we decouple $\tilde{\mathbf{X}}_{(N)}$ for updating $N$-th factor matrix. We first re-express $\tilde{\mathbf{X}}_{(N)}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right)$ using temporal block tensors $\mathcal{X}^{<i>}$ for $i = S,..,E$ as follows:

$$\tilde{\mathbf{X}}_{(N)}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right) = \begin{bmatrix} \mathbf{X}_{(N)}^{<S>}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right) \\ \vdots \\ \mathbf{X}_{(N)}^{<E>}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right) \end{bmatrix}$$

Then, we replace $\mathbf{X}_{(N)}^{<i>}$ with the tucker results obtained at the preprocessing phase.

$$\tilde{\mathbf{X}}_{(N)}\left(\otimes_{k=1}^{N-1}\tilde{\mathbf{A}}^{(k)}\right) \approx \begin{bmatrix} (\mathbf{A}^{<S>})^{(N)}\mathbf{G}_{(N)}^{<S>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<S>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right) \\ \vdots \\ (\mathbf{A}^{<E>})^{(N)}\mathbf{G}_{(N)}^{<E>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<E>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right) \end{bmatrix} \tag{5.9}$$

Next, we obtain the following equation by inserting the right term of the above equation into Equation (5.2):

$$\tilde{\mathbf{A}}^{(N)} = \begin{bmatrix} (\mathbf{A}^{<S>})^{(N)}\mathbf{G}_{(N)}^{<S>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<S>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(N)}^{T} \\ \vdots \\ (\mathbf{A}^{<E>})^{(N)}\mathbf{G}_{(N)}^{<E>}\left(\otimes_{k=1}^{N-1}(\mathbf{A}^{<E>})^{(k)T}\tilde{\mathbf{A}}^{(k)}\right)\tilde{\mathbf{G}}_{(N)}^{T} \end{bmatrix} \left(\mathbf{C}^{(N)}\right)^{-1}$$

$$= \begin{bmatrix} (\mathbf{A}^{<S>})^{(N)}(\mathbf{B}^{<S>})^{(N)} \\ \vdots \\ (\mathbf{A}^{<E>})^{(N)}(\mathbf{B}^{<E>})^{(N)} \end{bmatrix} \left(\mathbf{C}^{(N)}\right)^{-1}$$

$(\mathbf{A}^{<S>})^{(N)}$ and $(\mathbf{A}^{<E>})^{(N)}$ are adjusted to fit to a range $[t_s, t_e]$. □

#### 5.3.4.5 Proof of Theorem 5.1

*Proof.* We split a tensor $\mathcal{X}$ into $B$ temporal block tensors $\mathcal{X}^{<i>}$, and then perform Tucker decomposition of $\mathcal{X}^{<i>}$ for $i = 1, \ldots, B$. Since we use Tucker-ALS in the preprocessing phase, the time complexity for each temporal block tensor $\mathcal{X}^{<i>}$ is $\mathcal{O}(MNI^{N-1}Jb)$. Therefore, the preprocessing phase takes $\mathcal{O}(MNI^{N-1}JbB)$ time. □

#### 5.3.4.6 Proof of Theorem 5.2

*Proof.* The time complexity of the query phase depends on updating factor matrices and core tensor. Updating a factor matrix or core tensor takes $\mathcal{O}\left(J^2 l_{[t_s,t_e]}\left(1 + \frac{NI}{b} + \frac{NJ^{N-1}}{b}\right)\right)$ time. Therefore, the total time complexity is $\mathcal{O}\left(MNJ^2 l_{[t_s,t_e]}\left(1 + \frac{NI}{b} + \frac{NJ^{N-1}}{b}\right)\right)$ which contains the time complexity of updating factor matrices and core tensor, the number of iterations, and the number of factor matrices. □

Table 5.3: Description of real-world tensor datasets.

| Dataset | Dimensionality | Length $l_{[t_s,t_e]}$ of Time Range | Summary |
|---|---|---|---|
| Boats[1] [92] | $320 \times 240 \times 7000$ | $(128, 2048)$ | Video |
| Walking Video [38] | $1080 \times 1980 \times 2400$ | $(128, 2048)$ | Video |
| Stock[3] | $3028 \times 54 \times 3050$ | $(128, 2048)$ | Time series |
| Traffic[4] [96] | $1084 \times 96 \times 2000$ | $(64, 1024)$ | Traffic volume |
| FMA[5] [94] | $7994 \times 1025 \times 700$ | $(32, 512)$ | Music |
| Absorb[6] | $192 \times 288 \times 30 \times 1200$ | $(64, 1024)$ | Climate |

#### 5.3.4.7 Proof of Theorem 5.3

*Proof.* For the mode-$N$, summing up the size of the factor matrices of the time dimension is equal to $I_N J$. For each mode $n \neq N$, there are $B$ factor matrices, for the $n$-th mode, of size $\mathcal{O}(IJ)$ where $B = \frac{I_N}{b}$ is the number of blocks. Then, the space complexity is $\mathcal{O}(NIJ(\lceil \frac{I_N}{b} \rceil) + I_N J)$. □

#### 5.3.4.8 Proof of Theorem 5.4

*Proof.* Given a time range $[t_s, t_e]$, summing up the size of the factor matrices of the time dimension is equal to $l_{[t_s,t_e]} \times J$; the size of the factor matrix of a non-temporal mode is $I \times J$, and the number of block is equal to $\lceil \frac{l_{[t_s,t_e]}}{b} \rceil$ or $(\lceil \frac{l_{[t_s,t_e]}}{b} \rceil) + 1$. The size of the block results used in the query phase is $\mathcal{O}(NIJ(\lceil \frac{l_{[t_s,t_e]}}{b} \rceil) + l_{[t_s,t_e]} J)$. By carefully stitching the block results, intermediate data are always smaller than the block results. Therefore, the space cost of Zoom-Tucker is $\mathcal{O}\left(NIJ(\lceil \frac{l_{[t_s,t_e]}}{b} \rceil) + J l_{[t_s,t_e]}\right)$ for a given time range $[t_s, t_e]$. □

## 5.4 Experiment

We present experimental results to answer the following questions.

Q1 **Performance Trade-off (Section 5.4.2).** Does Zoom-Tucker provide the best trade-off between query time and reconstruction error?

Q2 **Space Cost (Section 5.4.3).** What is the space cost of Zoom-Tucker and competitors for preprocessed results?

Q3 **Query Cost (Section 5.4.4).** How quickly does Zoom-Tucker answer various time range queries?

Q4 **Effects of the block size $b$ (Section 5.4.5).** How does a block size $b$ affect query time and reconstruction error of Zoom-Tucker?

Q5 **Discovery (Section 5.4.6).** What pattern does Zoom-Tucker discover in different time ranges?

### 5.4.1 Experimental Settings

**Machine.** We run experiments on a workstation with a single CPU (Intel Xeon E5-2630 v4 @ 2.2GHz), and 512GB memory.

**Dataset.** We use six real-world dense tensors in Table 5.3. Boats[1] [92] and Walking Video[2] [38] datasets contain grayscale videos in the form of (height, width, time; value). Stock dataset[3] contains 5 basic features (open price, high price, low price, close price, trade volume) and 49 technical indicators features of Korea Stocks. Stock dataset has the form of (stock, features, date; value). The basic features are collected daily from *Jan. 2, 2008* to *May 6, 2020*. Traffic dataset[4] [96] contains traffic volume information in the form of (sensor, frequency, time; measurement). FMA dataset[5] [94] contains music information: (song, frequency, time; value). We convert a time series into an image of a log-power spectrogram for each song. Absorb dataset[6] is about absorption of aerosol in the form of (longitudes, latitudes, altitude, time; measurement).

[1] http://changedetection.net/
[2] https://github.com/OsmanMalik/tucker-tensorsketch
[3] https://datalab.snu.ac.kr/zoomtucker
[4] https://github.com/florinsch/BigTrafficData
[5] https://github.com/mdeff/fma
[6] https://www.earthsystemgrid.org/

**Competitors.** We compare Zoom-Tucker with 6 Tucker decomposition methods based on ALS approach. Zoom-Tucker and other methods are implemented in MATLAB (R2019b). We use the open sourced codes for 4 competitors: D-Tucker[7], Tucker-ALS [97], Tucker-ts[8], and Tucker-ttmts[8]. For MACH, we run Tucker-ALS in Tensor Toolbox [97] for a sampled tensor after sampling elements of a tensor; we use our implementation for a sampling scheme. We use the source code of RTD [36] provided by the authors.

**Parameters.** We use the following parameters for experiments:

- **Number of threads:** we use a single thread.
- **Max number of iterations:** the maximum number of iterations is set to 100.
- **Rank:** we set the dimensionality $J_n$ of each mode of core tensor to 10.
- **Choosing a time range query:** we randomly choose a start time $t_s$ of a time range, and compute $t_e = t_s + l_{[t_s,t_e]} - 1$ where $l_{[t_s,t_e]}$ is the length of the time range; we choose $l_{[t_s,t_e]}$ among the sets described in Table 5.3.
- **Block size $b$:** we set $b$ to 50 except in Section 5.4.5.
- **Tolerance:** the iteration stops when the variation of the error $\frac{\sqrt{\|\mathcal{X}\|_{\mathbf{F}}^2 - \|\mathcal{G}\|_{\mathbf{F}}^2}}{\|\mathcal{X}\|_{\mathbf{F}}}$ [29] is less than $\varepsilon = 10^{-4}$.

Other parameters for competitors are set to the values proposed in each paper. To compare the running time, we run each method 5 times, and report the average.

**Implementation details.** In the time range query problem, Zoom-Tucker, D-Tucker, and MACH preprocess a given tensor, and then perform Tucker decomposition for a time range query using preprocessed results included in the range. In contrast, Tucker-ALS and RTD perform Tucker decomposition using a sub-tensor

[7] https://datalab.snu.ac.kr/dtucker/
[8] https://github.com/OsmanMalik/tucker-tensorsketch

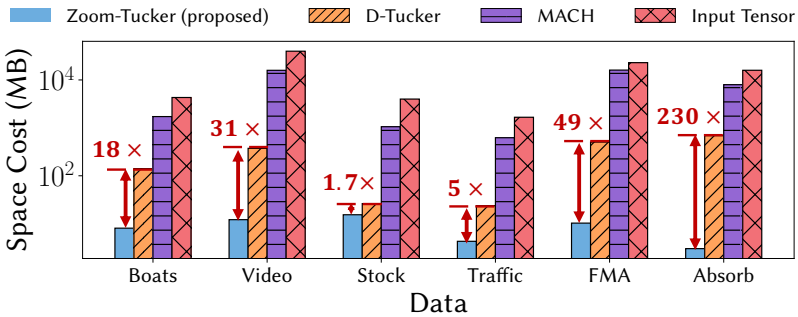



Figure 5.6: Space cost for storing preprocessed results. *Input Tensor* corresponds to the space cost of Tucker-ALS, Tucker-ts, Tucker-ttmts, and RTD. ZOOM-TUCKER requires up to 230× less space than competitors.

included in a time range query. Although Tucker-ts and Tucker-ttmts have a preprocessing phase, they also perform Tucker decomposition from scratch for a time range query since there is an inseparable preprocessed result along the time dimension.

**Reconstruction error.** Given an input tensor $\mathcal{X}$ and the reconstruction $\hat{\mathcal{X}}$ from the output of Tucker decomposition, reconstruction error is defined as $\frac{\|\mathcal{X}-\hat{\mathcal{X}}\|_F^2}{\|\mathcal{X}\|_F^2}$. Reconstruction error describes how well the reconstruction $\hat{\mathcal{X}}$ of Tucker decomposition represents an input tensor $\mathcal{X}$.

### 5.4.2 Trade-off between Query Time and Reconstruction Error

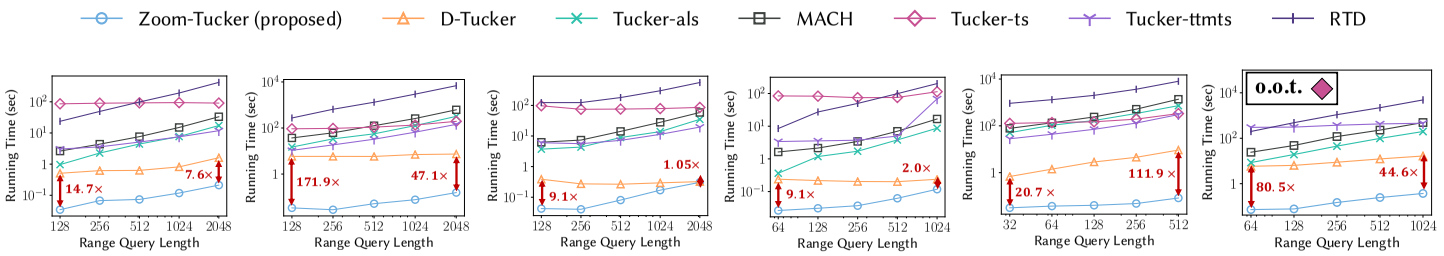
We compare the running time and reconstruction error of ZOOM-TUCKER with those of competitors for various time ranges. For each dataset, we use the narrowest and the widest time ranges among the ranges described in Table 5.3. Figure 5.2 shows that ZOOM-TUCKER is the closest method to the best point with the smallest error and running time. ZOOM-TUCKER is up to 171.9× and 111.9× faster than the second-fastest method, in narrow and wide time ranges, respectively, with similar errors.

### 5.4.3 Space Cost

We compare the storage cost of Zoom-Tucker with those of competitors for storing preprocessed results. Note that memory requirements for a time range query are proportional to the storage cost since preprocessed results or an input tensor is the dominant term in the space cost. Figure 5.6 shows that Zoom-Tucker requires the lowest space; Zoom-Tucker requires up to 230× less space than the second-best method D-Tucker. Zoom-Tucker has more compression rate on the 4-order tensor, Absorb dataset.

### 5.4.4 Query Cost

Figure 5.7 shows that Zoom-Tucker outperforms competitors for all time ranges; Zoom-Tucker is up to 171.9× faster than the second-fastest method for the narrow time ranges. Zoom-Tucker is up to 111.9× faster than competitors for the wide time ranges. In addition, Zoom-Tucker exhibits near-linear scalability in terms of the length of a time range.

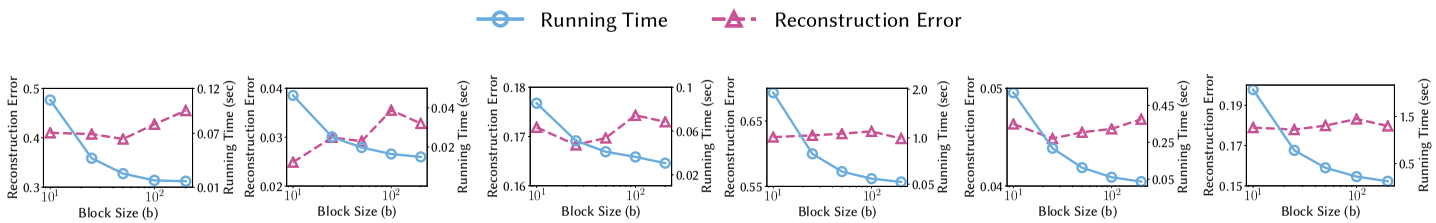



(a) Boat data (b) Video data (c) Stock data (d) Traffic data (e) FMA data (f) Absorb data

Figure 5.7: Query time for various time range queries. Zoom-Tucker is up to 171.9× faster than competitors while providing linear scalability in terms of the length of a time range. o.o.t.: out of time (takes more than 20,000 seconds).

### 5.4.5 Effects of Block Size $b$

We investigate the effects of block size $b$ on running time and reconstruction error of Zoom-Tucker. We use block sizes 10, 25, 50, 100, and 200 on Stock, Traffic, and Absorb datasets. As shown in Figures 5.8(a) to 5.8(c), there are trade-off relationships between running time and reconstruction error for narrow time range queries. In Figures 5.8(d) to 5.8(f), the running time of Zoom-Tucker is inversely proportional to $b$ for a wide range query while the reconstruction error is not sensitive to $b$. A large $b$ prevents the preprocessing phase from capturing local information so that it is challenging to serve narrow time range queries. For wide time range queries, local information has little effect on reconstruction errors since capturing widespread patterns is more beneficial in reducing errors. Therefore, we select 50, which is the largest value providing small errors for narrow time range queries, for the default block size to preprocess all datasets in other experimental sections.




(a) Stock data (128) (b) Traffic data (64) (c) Absorb data (64) (d) Stock data (2048) (e) Traffic data (1024) (f) Absorb data (1024)

Figure 5.8: Sensitivity with respect to block size $b$ on Stock, Traffic, and Absorb datasets. Numbers after the data name represent the length of time ranges; e.g., (128) means the length of time range is $t_e - t_s + 1 = 128$ timesteps. (a,b,c) There are trade-off relationships between running time and reconstruction error for narrow time range queries. (d,e,f) For wide time range queries, the running times decrease while the errors do not change much, as block size increases.

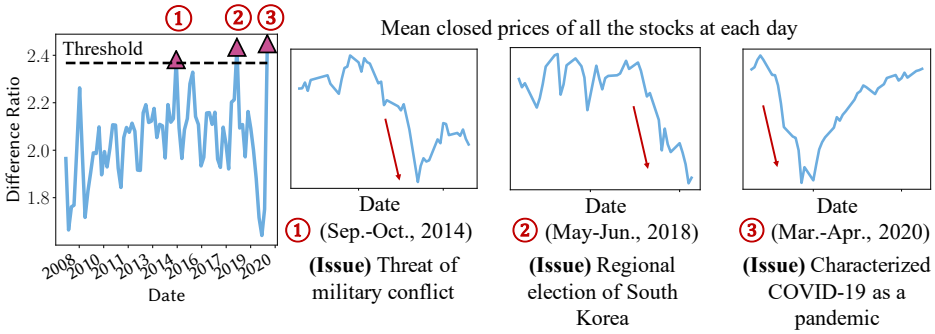



Figure 5.9: Anomalous two-month ranges and their related events, found by Zoom-Tucker.

### 5.4.6 Discovery

On Stock dataset, we discover interesting results by answering various time range queries with Zoom-Tucker.

**Finding anomalous ranges.** The goal is to find narrow time ranges that are anomalous, compared to the entire time range. For the goal, we select every consecutive two-month interval from Jan. 1, 2008 to Apr. 30, 2020, perform Tucker decomposition for each of the intervals using Zoom-Tucker, and find anomalous ranges that deviate the most from the entire ranges. Given a two-month range $r$, and its corresponding sub-tensor $\tilde{\mathcal{X}}$, we compute the anomaly score for $r$ using the difference ratio $\frac{\|\tilde{\mathcal{X}}-\hat{\mathcal{Y}}\|_F^2}{\|\tilde{\mathcal{X}}-\hat{\mathcal{Z}}\|_F^2}$ where $\hat{\mathcal{Y}}$ and $\hat{\mathcal{Z}}$ are the sub-tensors for $r$ reconstructed from the Tucker results of 1) the entire range query, and 2) the two-month range query, respectively.

The leftmost plot of Figure 5.9 shows the difference ratios and the top three anomalous ranges where the threshold indicates 2 standard deviations from the mean. The right three plots of Figure 5.9 show that the three anomalies follow the similar plunging pattern of prices from issues affecting the stock market.

**Analyzing trend change.** We analyze the change of yearly trend of *Samsung Electronics* in the years 2013 and 2018. For each of the range (year 2013 or 2018), we perform Zoom-Tucker and get the feature matrix $\tilde{\mathbf{A}}^{(1)}$ each of whose rows contain

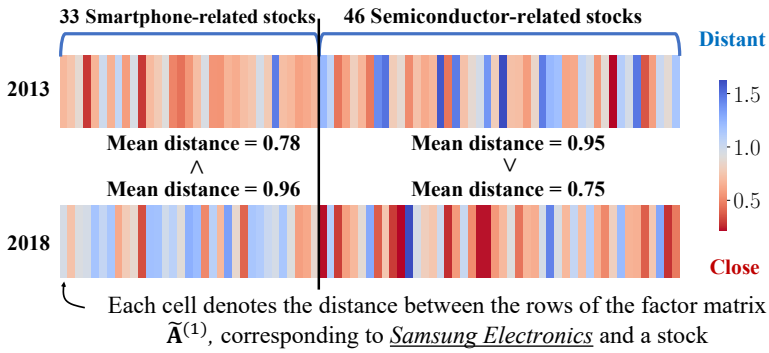



Figure 5.10: Cosine distance between feature vectors of *Samsung Electronics* and other stocks related to smartphones or semiconductors in 2013 and 2018. Zoom-Tucker helps capture the clear change of the trend, where *Samsung Electronics* is closer to smartphone-related stocks in 2013, but to semiconductor-related stocks in 2018.

the latent features of a stock. We also manually pick 33 smartphone-related stocks and 46 semiconductor-related stocks, and compare the cosine distance between the latent feature vectors of each stock and *Samsung Electronics*.

Figure 5.10 shows the result. Note that there is a clear change of the distances between year 2013 and 2018: *Samsung Electronics* is closer to smartphone-related stocks in 2013, but to semiconductor-related stocks in 2018. This result exactly reflects the sales trend of *Samsung Electronics*; the annual sales of its smartphone division are 3.7× larger than those of its semiconductor division in 2013, while in 2018 the annual sales of its semiconductor division are 30% larger than those of its smartphone division. Zoom-Tucker enables us to quickly and accurately capture this trend change.

## 5.5 Summary

In this work, we propose Zoom-Tucker, an efficient Tucker decomposition method to discover latent factors in a given time range from a temporal tensor. Zoom-Tucker efficiently answers diverse time range queries with the preprocessing phase and the query phase. In the preprocessing phase, Zoom-Tucker lays the groundwork for an

efficient time range query by compressing sub-tensors along time dimension block by block. Given a time range query in the query phase, Zoom-Tucker elaborately stitches compressed results reducing computational cost and space cost. Experiments show that Zoom-Tucker is up to 171.9× faster and requires up to 230× less space than existing methods, with comparable accuracy to competitors. With Zoom-Tucker, we discover interesting patterns including anomalous ranges and trend changes in a real-world stock dataset. Future research includes extending the method for sparse tensors.

# Chapter 6

# Future Works

In this section, I describe research plans that extend my works of this thesis. My future works are to 1) devise an efficient method for an irregular tensor in an online streaming setting and 2) propose tensor algorithms integrated with deep learning techniques. With the following plan, I look forward to understanding complex phenomena inherent in the tensors.

## 6.1 Efficient Online Streaming Method for an Irregular Tensor

How can we efficiently analyze an irregular tensor in an online streaming setting? Many real-world irregular tensors are dynamically collected in nature where an irregular tensor consists of matrices whose columns have the same size but rows have different sizes. Specifically, the row sizes of existing matrices and the number of matrices grow over time. However, many existing methods are inefficient in the online streaming setting, and the main challenge is to avoid computations involved with old data accumulated over time. To address this problem, my future research plan is to devise an efficient method for an irregular tensor in an online streaming setting.

## 6.2 Novel Tensor Method with Deep Learning Techniques

How can we effectively analyze real-world tensors? How can we find accurate factor matrices of tensor decomposition? Recent representative deep learning architectures have improved the performance in various applications. However, tensor data are relatively far from deep learning, compared to graph, image, and text data processed using Graph Convolutional Network (GCN), Convolutional Neural Network (CNN), and Transformer, respectively. Although a few tensor algorithms [117, 118, 119, 120] use deep learning techniques, there are still a lot of problems to be addressed for integrating a tensor algorithm with deep learning architectures. Since they focus only on using a deep learning technique to fuse factor vectors, they fail to extract factor vectors of high quality. Yang [121] et al. extract factor vectors by contrastive learning with data augmentation, but there is room for extracting better factor vectors by further improving the performance of data augmentation. My research direction is to find the characteristics inherent in real-world tensors and then develop a tensor algorithm integrated with deep learning techniques which fully employ the characteristics.

# Chapter 7

# Conclusion

In this dissertation, I develop efficient tensor decomposition methods for regular and irregular tensors in real-world settings. In addition, my proposed method successfully deals with diverse time ranges in a temporal tensor.

First, I propose D-Tucker, a fast and memory-efficient Tucker decomposition method for regular tensors. D-Tucker approximates a given tensor and computes Tucker decomposition only using the approximated results. I experimentally show that D-Tucker is much faster than existing methods while requiring less space than them.

Second, I propose DPar2, a fast and scalable PARAFAC2 decomposition method for irregular tensors. Similar to D-Tucker, DPar2 reduces numerical computations and intermediate data by approximating an irregular tensor and computing PARAFAC2 decomposition with the approximated results. In addition, it maximizes multi-core parallelism by considering irregularity. With these ideas, DPar2 achieves much faster and more scalable than existing PARAFAC2 decomposition methods.

Finally, I propose Zoom-Tucker, a fast and memory-efficient Tucker decomposition method for diverse time ranges Zoom-Tucker effectively approximates a given tensor before time range queries are given. Then, it answers diverse time range queries quickly and memory-efficiently by carefully dealing with the approximated results and exploiting fruitful mathematical techniques. Through extensive experiments, I show that Zoom-Tucker answers diverse time range queries more efficiently

than existing Tucker decomposition methods.

In future works, I will develop more powerful tensor algorithms based on the knowledge found in previous works. My future directions are to develop an efficient method for an irregular tensor in an online streaming setting, develop tensor algorithms with deep learning techniques (e.g., contrastive learning), and generalize tensor decomposition.

# References


[1] J. Jang and U. Kang, "Dpar2: Fast and scalable PARAFAC2 decomposition for irregular dense tensors," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pp. 2454–2467, IEEE, 2022.

[2] I. Perros, E. E. Papalexakis, F. Wang, R. W. Vuduc, E. Searles, M. Thompson, and J. Sun, "Spartan: Scalable PARAFAC2 for large & sparse data," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pp. 375–384, ACM, 2017.

[3] A. Afshar, I. Perros, E. E. Papalexakis, E. Searles, J. C. Ho, and J. Sun, "COPA: constrained PARAFAC2 for sparse & large datasets," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pp. 793–802, ACM, 2018.

[4] K. Yin, W. K. Cheung, B. C. M. Fung, and J. Poon, "Tedpar: Temporally dependent PARAFAC2 factorization for phenotype-based disease progression modeling," in *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM 2021, Virtual Event, April 29 - May 1, 2021* (C. Demeniconi and I. Davidson, eds.), pp. 594–602, SIAM, 2021.

[5] J. Oh, K. Shin, E. E. Papalexakis, C. Faloutsos, and H. Yu, "S-HOT: scalable high-order tucker decomposition," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, pp. 761–770, ACM, 2017.

[6] H. Tan, G. Feng, J. Feng, W. Wang, Y.-J. Zhang, and F. Li, "A tensor-based method for missing traffic data completion," *Transportation Research Part C: Emerging Technologies*, vol. 28, pp. 15–27, 2013.

[7] S. Oh, N. Park, L. Sael, and U. Kang, "Scalable tucker factorization for sparse tensors - algorithms and discoveries," in *34th IEEE International Conference on*

*Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 1120–1131, IEEE Computer Society, 2018.

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

[9] J. Liu, P. Musialski, P. Wonka, and J. Ye, "Tensor completion for estimating missing values in visual data," in *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pp. 2114–2121, IEEE Computer Society, 2009.

[10] N. Zheng, Q. Li, S. Liao, and L. Zhang, "Flickr group recommendation based on tensor decomposition," in *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2010, Geneva, Switzerland, July 19-23, 2010* (F. Crestani, S. Marchand-Maillet, H. Chen, E. N. Efthimiadis, and J. Savoy, eds.), pp. 737–738, ACM, 2010.

[11] D. Choi, J.-G. Jang, and U. Kang, "S3cmtf: Fast, accurate, and scalable method for incomplete coupled matrix-tensor factorization," *PloS one*, vol. 14, no. 6, p. e0217316, 2019.

[12] L. Xiong, X. Chen, T. Huang, J. G. Schneider, and J. G. Carbonell, "Temporal collaborative filtering with bayesian probabilistic tensor factorization," in *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pp. 211–222, SIAM, 2010.

[13] P. Bhargava, T. Phan, J. Zhou, and J. Lee, "Who, what, when, and where: Multi-dimensional collaborative recommendations using tensor factorization on sparse user-generated data," in *Proceedings of the 24th international conference on world wide web*, pp. 130–140, 2015.

[14] Z. Chen, Z. Xu, and D. Wang, "Deep transfer tensor decomposition with orthogonal constraint for recommender systems," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative*

*Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 4010–4018, AAAI Press, 2021.

[15] D. Koutra, E. E. Papalexakis, and C. Faloutsos, "Tensorsplat: Spotting latent anomalies in time," in *16th Panhellenic Conference on Informatics, PCI 2012, Piraeus, Greece, October 5-7, 2012*, pp. 144–149, IEEE Computer Society, 2012.

[16] T. Kwon, I. Park, D. Lee, and K. Shin, "Slicenstitch: Continuous CP decomposition of sparse tensor streams," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pp. 816–827, IEEE, 2021.

[17] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: dynamic tensor analysis," in *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pp. 374–383, ACM, 2006.

[18] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Parcube: Sparse parallelizable tensor decompositions," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 521–536, Springer, 2012.

[19] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *ICLR*, 2016.

[20] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[21] A. H. Phan, K. Sobolev, K. Sozykin, D. Ermilov, J. Gusak, P. Tichavský, V. Glukhov, I. V. Oseledets, and A. Cichocki, "Stable low-rank tensor decomposition for compression of convolutional neural network," in *Computer Vision -*

*ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXIX*, vol. 12374 of *Lecture Notes in Computer Science*, pp. 522–539, Springer, 2020.

[22] M. Yin, S. Liao, X. Liu, X. Wang, and B. Yuan, "Towards extremely compact rnns for video recognition with fully decomposed hierarchical tucker structure," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pp. 12085–12094, Computer Vision Foundation / IEEE, 2021.

[23] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 3891–3900, PMLR, 2017.

[24] M. Yin, H. Phan, X. Zang, S. Liao, and B. Yuan, "BATUDE: budget-aware neural network compression based on tucker decomposition," in *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pp. 8874–8882, AAAI Press, 2022.

[25] J. Jang and U. Kang, "Fast and memory-efficient tucker decomposition for answering diverse time range queries," in *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pp. 725–735, ACM, 2021.

[26] J. Jang and U. Kang, "D-tucker: Fast and memory-efficient tucker decomposition for dense tensors," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pp. 1850–1853, IEEE, 2020.

[27] J.-G. Jang and U. Kang, "Static and streaming tucker decomposition for dense tensors," *ACM Transactions on Knowledge Discovery from Data*, 2022.

[28] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors," *SIAM J. Matrix Analysis Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.

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

[30] R. A. Harshman, "Parafac2: Mathematical and technical notes," *UCLA working papers in phonetics*, vol. 22, no. 3044, p. 122215, 1972.

[31] H. A. Kiers, J. M. Ten Berge, and R. Bro, "Parafac2—part i. a direct fitting algorithm for the parafac2 model," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 13, no. 3-4, pp. 275–294, 1999.

[32] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU press, 2013.

[33] Y. Cheng and M. Haardt, "Efficient computation of the PARAFAC2 decomposition," in *53rd Asilomar Conference on Signals, Systems, and Computers, ACSCC 2019, Pacific Grove, CA, USA, November 3-6, 2019* (M. B. Matthews, ed.), pp. 1626–1630, IEEE, 2019.

[34] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Transactions on Mathematical Software*, vol. 32, pp. 635–653, Dec. 2006.

[35] J. Li, C. Battaglino, I. Perros, J. Sun, and R. W. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pp. 76:1–76:12, ACM, 2015.

[36] M. Che and Y. Wei, "Randomized algorithms for the approximations of tucker and the tensor train decompositions," *Adv. Comput. Math.*, vol. 45, no. 1, pp. 395–428, 2019.

[37] C. E. Tsourakakis, "MACH: fast randomized tensor decompositions," in *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pp. 689–700, SIAM, 2010.

[38] O. A. Malik and S. Becker, "Low-rank tucker decomposition of large tensors using tensorsketch," in *Advances in Neural Information Processing Systems 31:*

*Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 10117–10127, 2018.

[39] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries," in *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012* (Q. Yang, D. Agarwal, and J. Pei, eds.), pp. 316–324, ACM, 2012.

[40] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pp. 61–70, IEEE Computer Society, 2015.

[41] S. Oh, N. Park, J. Jang, L. Sael, and U. Kang, "High-performance tucker factorization on heterogeneous platforms," *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 10, pp. 2237–2248, 2019.

[42] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, vol. 10417 of *Lecture Notes in Computer Science*, pp. 653–668, Springer, 2017.

[43] F. Yang, F. Shang, Y. Huang, J. Cheng, J. Li, Y. Zhao, and R. Zhao, "LFTF: A framework for efficient tensor analytics at scale," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 745–756, 2017.

[44] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pp. 1047–1058, IEEE Computer Society, 2015.

[45] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *2014 IEEE International Conference on Data Min-*

*ing, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pp. 989–994, IEEE Computer Society, 2014.

[46] 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.

[47] N. Park, B. Jeon, J. Lee, and U. Kang, "Bigtensor: Mining billion-scale tensor made easy," in *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pp. 2457–2460, ACM, 2016.

[48] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pp. 1038–1047, IEEE Computer Society, 2017.

[49] J. W. Choi, X. Liu, and V. T. Chakaravarthy, "High-performance dense tucker decomposition on GPU clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pp. 42:1–42:11, IEEE / ACM, 2018.

[50] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 912–922, IEEE Computer Society, 2016.

[51] G. Ballard, A. Klinvex, and T. G. Kolda, "Tuckermpi: A parallel c++/mpi software package for large-scale data compression via the tucker tensor decomposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 2, pp. 1–31, 2020.

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

[53] X. Li, S. Huang, K. S. Candan, and M. L. Sapino, "2pcp: Two-phase CP decomposition for billion-scale dense tensors," in *32nd IEEE International Conference*

*on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pp. 835–846, 2016.

[54] D. Chen, Y. Hu, L. Wang, A. Y. Zomaya, and X. Li, "H-PARAFAC: hierarchical parallel factor analysis of multidimensional big data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1091–1104, 2017.

[55] N. Park, S. Oh, and U. Kang, "Fast and scalable distributed boolean tensor factorization," in *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pp. 1071–1082, IEEE Computer Society, 2017.

[56] N. Park, S. Oh, and U. Kang, "Fast and scalable method for distributed boolean tensor factorization," *VLDB J.*, vol. 28, no. 4, pp. 549–574, 2019.

[57] I. Jeon, E. E. Papalexakis, C. Faloutsos, L. Sael, and U. Kang, "Mining billion-scale tensors: algorithms and discoveries," *VLDB J.*, vol. 25, no. 4, pp. 519–544, 2016.

[58] N. D. Sidiropoulos, L. D. Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Trans. Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[59] S. Rendle and L. Schmidt-Thieme, "Pairwise interaction tensor factorization for personalized tag recommendation," in *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010, New York, NY, USA, February 4-6, 2010* (B. D. Davison, T. Suel, N. Craswell, and B. Liu, eds.), pp. 81–90, ACM, 2010.

[60] X. Cao, X. Wei, Y. Han, and D. Lin, "Robust face clustering via tensor decomposition," *IEEE Trans. Cybernetics*, vol. 45, no. 11, pp. 2546–2557, 2015.

[61] H. Huang, C. H. Q. Ding, D. Luo, and T. Li, "Simultaneous tensor subspace selection and clustering: the equivalence of high order svd and k-means clustering," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pp. 327–335, ACM, 2008.

[62] J. Tang, X. Shu, G. Qi, Z. Li, M. Wang, S. Yan, and R. C. Jain, "Tri-clustered tensor completion for social-aware image tag refinement," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 8, pp. 1662–1674, 2017.

[63] J. Tang, X. Shu, Z. Li, Y. Jiang, and Q. Tian, "Social anchor-unit graph regularized tensor completion for large-scale image retagging," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 8, pp. 2027–2034, 2019.

[64] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pp. 363–372, IEEE Computer Society, 2008.

[65] I. Perros, R. Chen, R. W. Vuduc, and J. Sun, "Sparse hierarchical tucker factorization and its application to healthcare," in *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015* (C. C. Aggarwal, Z. Zhou, A. Tuzhilin, H. Xiong, and X. Wu, eds.), pp. 943–948, IEEE Computer Society, 2015.

[66] Y. Ren, J. Lou, L. Xiong, and J. C. Ho, "Robust irregular tensor factorization and completion for temporal health data analysis," in *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020* (M. d'Aquin, S. Dietze, C. Hauff, E. Curry, and P. Cudré-Mauroux, eds.), pp. 1295–1304, ACM, 2020.

[67] A. Afshar, I. Perros, H. Park, C. Defilippi, X. Yan, W. Stewart, J. Ho, and J. Sun, "Taste: Temporal and static tensor factorization for phenotyping electronic health records," in *Proceedings of the ACM Conference on Health, Inference, and Learning*, pp. 193–203, 2020.

[68] K. Yin, A. Afshar, J. C. Ho, W. K. Cheung, C. Zhang, and J. Sun, "Logpar: Logistic PARAFAC2 factorization for temporal binary data with missing values," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pp. 1625–1635, ACM, 2020.

[69] E. Gujral, G. Theocharous, and E. E. Papalexakis, "SPADE: streaming PARAFAC2 decomposition for large datasets," in *Proceedings of the 2020 SIAM*

*International Conference on Data Mining, SDM 2020, Cincinnati, Ohio, USA, May 7-9, 2020* (C. Demeniconi and N. V. Chawla, eds.), pp. 577–585, SIAM, 2020.

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

[71] S. Zhou, X. V. Nguyen, J. Bailey, Y. Jia, and I. Davidson, "Accelerating online CP decompositions for higher order tensors," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 1375–1384, ACM, 2016.

[72] Q. Song, X. Huang, H. Ge, J. Caverlee, and X. Hu, "Multi-aspect streaming tensor completion," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pp. 435–443, ACM, 2017.

[73] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis, "Streaming tensor factorization for infinite data sources," in *Proceedings of the 2018 SIAM International Conference on Data Mining, SDM 2018, May 3-5, 2018, San Diego Marriott Mission Valley, San Diego, CA, USA*, pp. 81–89, SIAM, 2018.

[74] E. Gujral, R. Pasricha, and E. E. Papalexakis, "Sambaten: Sampling-based batch incremental tensor decomposition," in *Proceedings of the 2018 SIAM International Conference on Data Mining, SDM 2018, May 3-5, 2018, San Diego Marriott Mission Valley, San Diego, CA, USA*, pp. 387–395, SIAM, 2018.

[75] S. Zhou, S. M. Erfani, and J. Bailey, "Online CP decomposition for sparse tensors," in *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, pp. 1458–1463, IEEE Computer Society, 2018.

[76] D. Ahn, S. Kim, and U. Kang, "Accurate online tensor factorization for temporal tensor streams with missing values," in *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pp. 2822–2826, ACM, 2021.

[77] D. Lee and K. Shin, "Robust factorization of real-world tensor streams with patterns, missing values, and outliers," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pp. 840–851, IEEE, 2021.

[78] D. Ahn, J. Jang, and U. Kang, "Time-aware tensor decomposition for sparse tensors," *Mach. Learn.*, vol. 111, no. 4, pp. 1409–1430, 2022.

[79] S. Son, Y.-c. Park, M. Cho, and U. Kang, "Dao-cp: Data-adaptive online cp decomposition for tensor stream," *PLOS ONE*, vol. 17, pp. 1–18, 04 2022.

[80] Y. Sun, Y. Guo, C. Luo, J. A. Tropp, and M. Udell, "Low-rank tucker approximation of a tensor from streaming data," *SIAM J. Math. Data Sci.*, vol. 2, no. 4, pp. 1123–1150, 2020.

[81] J. Jang, D. Choi, J. Jung, and U. Kang, "Zoom-svd: Fast and memory efficient method for extracting key patterns in an arbitrary time range," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pp. 1083–1092, ACM, 2018.

[82] R. Minster, A. K. Saibaba, and M. E. Kilmer, "Randomized algorithms for low-rank tensor decompositions in the tucker format," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 1, pp. 189–215, 2020.

[83] N. Halko, P. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011.

[84] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, "A fast randomized algorithm for the approximation of matrices," *Applied and Computational Harmonic Analysis*, vol. 25, no. 3, pp. 335–366, 2008.

[85] K. L. Clarkson and D. P. Woodruff, "Low-rank approximation and regression in input sparsity time," *Journal of the ACM (JACM)*, vol. 63, no. 6, pp. 1–45, 2017.

[86] J. Baglama and L. Reichel, "Augmented implicitly restarted lanczos bidiagonalization methods," *SIAM J. Scientific Computing*, vol. 27, no. 1, pp. 19–42, 2005.

[87] S. Papadimitriou, J. Sun, and C. Faloutsos, "Streaming pattern discovery in multiple time-series," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pp. 697–708, ACM, 2005.

[88] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, "A new truncation strategy for the higher-order singular value decomposition," *SIAM J. Scientific Computing*, vol. 34, no. 2, 2012.

[89] M. Iwen and B. Ong, "A distributed and incremental svd algorithm for agglomerative data analysis on large networks," *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 4, pp. 1699–1718, 2016.

[90] 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.

[91] T. M. Mitchell, S. V. Shinkareva, A. Carlson, K.-M. Chang, V. L. Malave, R. A. Mason, and M. A. Just, "Predicting human brain activity associated with the meanings of nouns," *Science*, vol. 320, pp. 1191–1195, May 2008.

[92] Y. Wang, P. Jodoin, F. M. Porikli, J. Konrad, Y. Benezeth, and P. Ishwar, "Cdnet 2014: An expanded change detection benchmark dataset," in *IEEE Conference on Computer Vision and Pattern Recognition CVPR Workshops*, pp. 393–400, 2014.

[93] D. Foster, K. Amano, S. Nascimento, and M. Foster, "Frequency of metamerism in natural scenes," *Optical Society of America. Journal A: Optics, Image Science, and Vision*, vol. 23, pp. 2359–2372, 10 2006.

[94] M. Defferrard, K. Benzi, P. Vandergheynst, and X. Bresson, "FMA: A dataset for music analysis," in *18th International Society for Music Information Retrieval Conference (ISMIR)*, 2017.

[95] M. Defferrard, S. P. Mohanty, S. F. Carroll, and M. Salathé, "Learning to recognize musical genre from audio," in *The 2018 Web Conference Companion*, ACM Press, 2018.

[96] F. Schimbinschi, X. V. Nguyen, J. Bailey, C. Leckie, H. L. Vu, and R. Kotagiri, "Traffic forecasting in complex urban networks: Leveraging big data and machine learning," in *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*, pp. 1019–1024, IEEE Computer Society, 2015.

[97] B. W. Bader, T. G. Kolda, *et al.*, "Matlab tensor toolbox version 3.0-dev." Available online, Oct. 2017.

[98] Y. Lin, J. Sun, P. C. Castro, R. B. Konuru, H. Sundaram, and A. Kelliher, "Metafac: community discovery via relational hypergraph factorization," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, pp. 527–536, ACM, 2009.

[99] S. Spiegel, J. H. Clausen, S. Albayrak, and J. Kunegis, "Link prediction on evolving data using tensor factorization," in *New Frontiers in Applied Data Mining - PAKDD 2011 International Workshops, Shenzhen, China, May 24-27, 2011, Revised Selected Papers*, vol. 7104 of *Lecture Notes in Computer Science*, pp. 100–110, Springer, 2011.

[100] D. Ahn, S. Son, and U. Kang, "Gtensor: Fast and accurate tensor analysis system using gpus," in *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020* (M. d'Aquin, S. Dietze, C. Hauff, E. Curry, and P. Cudré-Mauroux, eds.), pp. 3361–3364, ACM, 2020.

[101] N. E. Helwig, "Estimating latent trends in multivariate longitudinal data via parafac2 with functional and structural constraints," *Biometrical Journal*, vol. 59, no. 4, pp. 783–803, 2017.

[102] B. M. Wise, N. B. Gallagher, and E. B. Martin, "Application of parafac2 to fault detection and diagnosis in semiconductor etch," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 15, no. 4, pp. 285–298, 2001.

[103] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pp. 1041–1044, ACM, 2014.

[104] J. Wang, Z. Liu, Y. Wu, and J. Yuan, "Mining actionlet ensemble for action recognition with depth cameras," in *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*, pp. 1290–1297, IEEE Computer Society, 2012.

[105] F. Karim, S. Majumdar, H. Darabi, and S. Harford, "Multivariate lstm-fcns for time series classification," 2018.

[106] J. Jung, J. Yoo, and U. Kang, "Signed random walk diffusion for effective representation learning in signed graphs," *PLOS ONE*, vol. 17, pp. 1–19, 03 2022.

[107] J. Jung, W. Jin, H. Park, and U. Kang, "Accurate relational reasoning in edge-labeled graphs by multi-labeled random walk with restart," *World Wide Web*, vol. 24, no. 4, pp. 1369–1393, 2021.

[108] J. Jung, W. Jin, and U. Kang, "Random walk-based ranking in signed social networks: model and algorithms," *Knowl. Inf. Syst.*, vol. 62, no. 2, pp. 571–610, 2020.

[109] W. Jin, J. Jung, and U. Kang, "Supervised and extended restart in random walks for ranking and link prediction in networks," *PLOS ONE*, vol. 14, pp. 1–23, 03 2019.

[110] K. Shin, J. Jung, L. Sael, and U. Kang, "BEAR: block elimination approach for random walk with restart on large graphs," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (T. K. Sellis, S. B. Davidson, and Z. G. Ives, eds.), pp. 1571–1585, ACM, 2015.

[111] J. Jung, N. Park, L. Sael, and U. Kang, "Bepi: Fast and memory-efficient method for billion-scale random walk with restart," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, eds.), pp. 789–804, ACM, 2017.

[112] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[113] B. Jeon, I. Jeon, L. Sael, and U. Kang, "Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pp. 811–822, IEEE Computer Society, 2016.

[114] H. Wang and N. Ahuja, "A tensor approximation approach to dimensionality reduction," *International Journal of Computer Vision*, vol. 76, no. 3, pp. 217–229, 2008.

[115] Y. Liu, Q. Yao, and Y. Li, "Generalizing tensor decomposition for n-ary relational knowledge bases," in *WWW*, pp. 1104–1114, ACM / IW3C2, 2020.

[116] T. Lacroix, G. Obozinski, and N. Usunier, "Tensor decompositions for temporal knowledge base completion," in *ICLR*, OpenReview.net, 2020.

[117] H. Liu, Y. Li, M. Tsang, and Y. Liu, "Costco: A neural tensor completion model for sparse tensors," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 324–334, 2019.

[118] H. Chen and J. Li, "Neural tensor model for learning multi-aspect factors in recommender systems," in *International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 2020, 2020.

[119] X. Wu, B. Shi, Y. Dong, C. Huang, and N. V. Chawla, "Neural tensor factorization for temporal interaction learning," in *Proceedings of the Twelfth ACM international conference on web search and data mining*, pp. 537–545, 2019.

[120] B. Jing, H. Tong, and Y. Zhu, "Network of tensor time series," in *Proceedings of the Web Conference 2021*, pp. 2425–2437, 2021.

[121] C. Yang, C. Qian, N. Singh, C. Xiao, M. B. Westover, E. Solomonik, and J. Sun, "Atd: Augmenting cp tensor decomposition by self supervision," in *Advances in Neural Information Processing Systems*.

# 요 약

실세계에 존재하는 다양한 다차원 데이터가 텐서로 표현된다. 텐서는 1차원의 벡터, 2차원의 행렬, 3차원 이상의 고차원 텐서를 포함하는 개념이다. 예를 들어, 주식 데이터, 헬스케어 데이터, 동영상 데이터, 센서 데이터, 영화 등급 데이터 등이 텐서로 표현될 수 있다. 실세계 텐서들에는 중요한 지식 및 정보들이 내재 되어있기 때문에 텐서를 분석하는 도구를 개발하는 것은 매우 중요하다. 하지만, 전통적인 텐서 분석 방법들은 대다수의 규모가 매우 큰 실세계 텐서를 분석하는데 있어 많은 시간과 공간 자원을 필요로 하기 때문에, 텐서들로부터 중요한 지식 및 정보들을 탐색하는데 많은 어려움을 겪어왔다.

본 학위 논문에서는 텐서 분해 기반 대규모 실세계 텐서 분석 기술의 한계를 극복하고자 한다. 기존 텐서 분해 방법들은 주어진 대규모 입력 텐서와의 계산이 많기 때문에 이를 줄이는 것이 효율성을 높이는데 매우 중요하다. 따라서, 텐서 데이터의 실세계 구조적 특징을 활용하여 대규모 입력 텐서에 대한 계산량을 줄임으로써 정확도 손실을 최소화하면서도 빠르고 확장성 있는 기법을 제안한다. 또한, 다양한 시간 범위에 숨겨진 지식 정보들을 효율적으로 탐색할 수 있는 텐서 분해 기반 텐서 분석 기술을 제안한다.

본 학위 논문에서 제안하는 방법들이 실세계 텐서를 매우 효율적으로 분석할 수 있다는 것을 실험을 통해 보여준다. 제안하는 방법들은 비슷한 에러를 가지면서도 매우 높은 효율성을 달성한다. 제안하는 방법들은 기존 방법들과 비교하여 주어진 규칙 텐서와 불규칙 텐서를 각각 최대 38.4배, 6배 빠르게 분해한다. 또한, 제안하는 방법은 임의의 시간 범위에 대해 최대 171.9배 빠르게 분석하는 것을 가능하게 해준다. 제안하는 방법들은 다양한 실세계 텐서들로부터 유의미한 지식을 효율적으로

탐구하는 것을 가능하게 한다.

**주요어 :** 텐서 마이닝, 텐서 분해, 터커 분해, PARAFAC2 분해, 실세계 규칙 텐서, 실세계 불규칙 텐서

**학번 :** 2017-23528

# 감사의 글

먼저 박사과정 동안 저를 성심성의껏 지도해주신 강유 교수님께 감사하다는 말씀드립니다. 교수님께 논문을 쓰고 연구를 하는 것 뿐만 아니라 실패에 좌절하지 않고 꾸준히 노력하는 교수님의 모습을 보며 세상을 살아가는 지혜를 배울 수 있었습니다. 이러한 배움을 통해 저는 박사과정을 포기하지 않고 무사히 마무리할 수 있었고, 앞으로도 나아갈 수 있는 힘이 생겼습니다.

졸업논문 심사에 참여하여 귀중한 의견을 주신 문봉기 교수님, 박근수 교수님, 황승원 교수님, 신기정 교수님께도 감사의 말씀을 드립니다.

저의 박사과정동안 함께 연구했던 뛰어난 연구자들에게도 감사하다는 말씀드립니다. 전북대학교 정진홍 교수님, 데이터 마이닝 연구실에서 함께 했던 전현식, 권순, 박용찬, 김태훈, 이전경, 박지원, 최동진, 오세준, 박남용, 안다원, 박채흠, 박문정, 이현동, 엔씨소프트의 김건수, 장창원, 아주대학교 이슬 교수님께 감사드립니다. 그리고, 비록 긴밀한 연구를 수행하지는 못했지만 박사과정 동안 많은 시간을 함께 했던 데이터 마이닝 구성원들께도 감사의 말씀 전합니다.

마지막으로, 저를 항상 믿어주시고 응원해준 가족들에게 이 모든 영광을 바칩니다. 가족들이 없었다면 박사학위를 무사히 마치지 못했을 것입니다. 저를 항상 무한한 사랑으로 응원해준 나의 가족 엄마, 아빠, 동생, 할머니, 할아버지에게 깊은 감사의 말씀을 전합니다. 또 이제는 가족이 될 장모님, 장인 어른, 그리고 기쁠 때나 슬플 때 항상 저와 함께 해준 소희에게 깊은 감사를 전합니다.