

Fast and Memory-Efficient Algorithms for High-Order Tucker Decomposition

Jiyuan Zhang¹, Jinoh Oh², Kijung Shin³, Evangelos E. Papalexakis⁴, Christos Faloutsos⁵, Hwanjo Yu⁶

¹ Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA;

² Adobe Systems, San Jose, CA, USA;

³ School of Electrical Engineering, KAIST, Daejeon, South Korea;

⁴ School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA;

⁵ Dept. of Computer Science and Engineering, UC Riverside, Riverside, CA, USA;

⁶ Dept. of Computer Science and Engineering, POSTECH, Pohang, South Korea

Abstract. Multi-aspect data appear frequently in web-related applications. For example, product reviews are quadruplets of the form (user, product, keyword, timestamp), and search-engine logs are quadruplets of the form (user, keyword, location, timestamp). How can we analyze such web-scale multi-aspect data on an off-the-shelf workstation with a limited amount of memory?

Tucker decomposition has been used widely for discovering patterns in such multi-aspect data, which are naturally expressed as large but sparse tensors. However, existing Tucker decomposition algorithms have limited scalability, failing to decompose large-scale high-order (≥ 4) tensors, since they *explicitly materialize* intermediate data, whose size grows exponentially with the order.

To address this problem, which we call “Materialization Bottleneck”, we propose S-HOT, a scalable algorithm for high-order Tucker decomposition. S-HOT minimizes materialized intermediate data by using an *on-the-fly computation*, and it is optimized for disk-resident tensors that are too large to fit in memory. We theoretically analyze the amount of memory and the number of data scans required by S-HOT. Moreover, we empirically show that S-HOT handles tensors with higher order, dimensionality, and rank than baselines. For example, S-HOT successfully decomposes a real-world tensor from the Microsoft Academic Graph on an off-the-shelf workstation, while all baselines fail. Especially, in terms of dimensionality, S-HOT decomposes *1000× larger* tensors than baselines.

Keywords: Tensor, High-order Tensor, Tensor Decomposition, Tucker Decomposition, Out-of-core Algorithm

Received xxx

Revised xxx

Accepted xxx

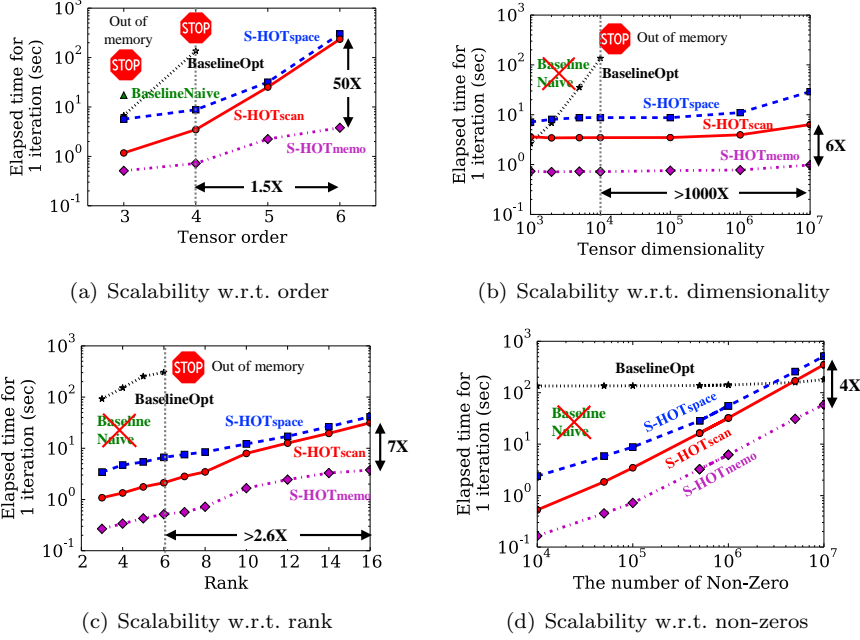


Fig. 1. **S-HOT scales up.** Every version of S-HOT successfully decomposes tensors with high order, dimensionality, and rank, while the baseline algorithms fail, running out of memory as those three factors increase. Especially, every version of S-HOT handles a tensor with 1000 times larger dimensionality. We use two baselines: 1) BaselineNaive (described in Section 3.1): naive algorithm for Tucker decomposition, and 2) BaselineOpt (Kolda and Sun, 2008) (described in Section 3.2): the state-of-the-art memory-efficient algorithm for Tucker decomposition. Note that all the methods have the same convergence properties (see Observation 1 in Section 4).

1. Introduction

Tensor decomposition is a widely-used technique for the analysis of multi-aspect data. Multi-aspect data, which are naturally modeled as high-order tensors, frequently appear in many applications (Cai et al, 2011; Kolda et al, 2005; Maruhashi et al, 2011; Moghaddam et al, 2012; Rendle and Schmidt-Thieme, 2010; Shin et al, 2017; Shin et al, 2018), including the following examples:

- Social media: 4-way tensor (sender, recipient, keyword, timestamp)
- Web search: 4-way tensor (user, keyword, location, timestamp)
- Internet security: 4-way tensor (source IP, destination IP, destination port, timestamp)
- Product reviews: 5-way tensor (user, product, keyword, rating, timestamp)

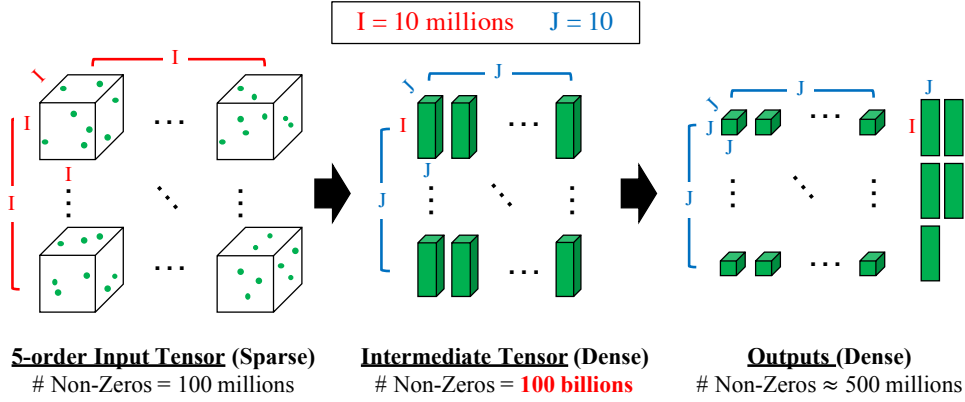


Fig. 2. **Illustration of M -Bottleneck.** For a high-order (≥ 4) sparse input tensor, the amount of space required for the intermediate tensor \mathcal{Y} can be much larger than that for the input tensor and the outputs. As in the figure, \mathcal{Y} is usually thinner but much denser than the input tensor. In such a case, materializing intermediate data becomes the scalability bottleneck of existing Tucker decomposition algorithms.

Most of these web-scale tensors are sparse (i.e., most of their entries are zero). For example, since typical customers buy and review a small fraction of products available at e-commerce sites, intermittently, most entries of the aforementioned product-review tensors are zero. To analyze such multi-aspect data, several tensor decomposition methods have been proposed, and we refer interested readers to an excellent survey (Kolda and Bader, 2009). Tensor decompositions have provided meaningful results in various domains (Acar et al, 2005; Kolda and Bader, 2009; Kolda et al, 2005; Lamba et al, 2016; Cai et al, 2011; Franz et al, 2009; Kolda et al, 2005; Moghaddam et al, 2012; Rendle and Schmidt-Thieme, 2010; Shin et al, 2017). Especially, Tucker decomposition (Tucker, 1966) has been successfully applied in many applications, including web search (Sun et al, 2005), network forensics (Sun et al, 2006), social network analysis (Chi et al, 2006), and scientific data compression (Austin et al, 2016).

Developing a scalable Tucker decomposition algorithm has been a challenge due to a huge amount of intermediate data generated during the computation. Briefly speaking, Alternating Least Square (ALS), the most widely-used Tucker decomposition algorithm, repeats two steps: 1) computing an intermediate tensor, denoted by \mathcal{Y} , and 2) computing the SVD of the matricized \mathcal{Y} (see Section 2 or (Kolda and Bader, 2009) for details). Previous studies (Kolda and Sun, 2008; Jeon et al, 2015) pointed out that a huge amount of intermediate data are generated during the first step, and they proposed algorithms for reducing the intermediate data by carefully ordering computation.

However, existing algorithms still have limited scalability and easily run out of memory, particularly when dealing with *high-order* (≥ 4) tensors. This is because existing algorithms *explicitly materialize* \mathcal{Y} , which is usually thinner but much denser than the input tensor, despite the fact that the amount of space required for storing \mathcal{Y} grows rapidly with respect to the order, dimensionality, and rank of the input tensor. For example, as illustrated in Fig. 2, the space required for \mathcal{Y} , is

about *400 Giga Bytes* for a 5-way tensor with 10 million dimensionality when the rank of Tucker decomposition is set to 10. We call this problem Materialization Bottleneck (or *M-Bottleneck* in short). Due to *M-Bottleneck*, existing algorithms are not suitable for decomposing tensors with high order, dimensionality, and/or rank. As seen in Fig. 1, even state-of-the-art algorithms (Kolda and Sun, 2008) easily run out of memory as these factors increase.

To avoid *M-Bottleneck*, in this work, we propose S-HOT, a scalable Tucker decomposition algorithm for large-scale, high-order, but sparse tensors. S-HOT is designed for decomposing high-order tensors on an off-the-shelf workstation. Our key idea is to compute \mathcal{Y} *on the fly*, without materialization, by combining both steps in ALS without changing its results. Specifically, we utilize the *reverse communication interface* of a recent scalable eigensolver called Implicitly Restart Arnoldi Method (IRAM) (Lehoucq et al, 1997), which enables SVD computation without materializing \mathcal{Y} . Moreover, S-HOT performs Tucker decomposition by streaming non-zero tensor entries from the disk, which enables it to handle disk-resident tensors that are too large to fit in memory. We offer the following versions of S-HOT with distinct advantages¹:

- S-HOT_{space}: the most space-efficient version that does not require additional copies the input tensor.
- S-HOT_{scan}: a faster version that requires multiple copies of the input tensor.
- S-HOT_{memo}: the fastest version that requires multiple copies of the input tensor and a buffer in main memory.

Our experimental results demonstrate that S-HOT outperforms baseline algorithms by providing significantly better scalability, as shown in Fig. 1. Specifically, all versions of S-HOT successfully decompose a 6-way tensor, while baselines fail to decompose even a 4-way tensor or a 5-way tensor due to their high memory requirements. The difference is more significant in terms of dimensionality. As seen in Fig. 1(b), S-HOT decomposes a tensor with **1000× larger dimensionality** than baselines.

Our contributions are summarized as follows.

- **Bottleneck Resolution:** We identify *M-Bottleneck* (Fig. 2), which limits the scalability of existing Tucker decomposition algorithms, and we avoid it by using an *on-the-fly computation*.
- **Scalable Algorithm Design:** We propose S-HOT, a scalable Tucker decomposition algorithm carefully designed for sparse high-order tensors that are too large to fit in memory. Compared to baselines, S-HOT scales up to *1000× bigger tensors* (Fig. 1) with identical convergence properties (Observation 1).
- **Theoretical Analyses:** We provide theoretical analyses on the amount of memory space and the number of data scans that S-HOT requires.

Reproducibility: The source code of S-HOT and the datasets used in the paper are available at <http://dm.postech.ac.kr/shot>.

In Section 2, we give preliminaries on tensors and Tucker decomposition. In Section 3, we review related work and introduce *M-Bottleneck*, which past algorithms commonly suffer from. In Section 4, we propose S-HOT, a scalable

¹ S-HOT_{space} and S-HOT_{scan} appeared in the conference version of this paper (Oh et al, 2017). This work extends (Oh et al, 2017) with S-HOT_{memo}, which is significantly faster than S-HOT_{space} and S-HOT_{scan}, space complexity analyses, and additional experimental results.

Table 1. Table of Symbols

Symbol	Definition
N	number of modes
\mathcal{X}	N -order input tensor $\in \mathbb{R}^{I_1 \times \dots \times I_N}$
$\mathcal{X}(i_1, \dots, i_N)$	(i_1, \dots, i_N) -th entry of \mathcal{X} (also denoted by $x_{i_1 \dots i_N}$)
$\Theta(\mathcal{X})$	set of the indices of all non-zero entries in \mathcal{X}
$\Theta_i^{(n)}(\mathcal{X})$	subset of $\Theta(\mathcal{X})$ where the n -th mode index is i
$\mathbf{X}_{(n)}^i$	mode- n unfolding of \mathcal{X}
M	number of non-zero entries in \mathcal{X}
I_n	dimensionality of the n -th mode of \mathcal{X}
J_n	number of component (rank) for the n -th mode
\mathcal{G}	N -order core tensor $\in \mathbb{R}^{J_1 \times \dots \times J_N}$
$\{\mathbf{A}\}$	set of all the factor matrices of \mathcal{X}
$\mathbf{A}^{(n)}$	mode- n factor matrix ($\in \mathbb{R}^{I_n \times J_n}$) of \mathcal{X}
$\bar{\mathbf{a}}_i^{(n)}$	i -th row-vector of $\mathbf{A}^{(n)}$
$\mathbf{a}_j^{(n)}$	j -th column-vector of $\mathbf{A}^{(n)}$
\circ	outer product
$\bar{\times}_n$	n -mode vector product
\times_n	n -mode matrix product

algorithm for high-order tucker decomposition, to address *M-Bottleneck*. After presenting experimental results in Section 5, we make conclusions in Section 6.

2. Preliminaries

In this section, we give the preliminaries on tensors (Section 2.1), basic tensor operations (Section 2.2), Tucker decomposition (Section 2.3), and Implicitly Restarted Arnoldi Method (Section 2.4).

2.1. Tensors and Notations

A tensor is a multi-order array which generalizes a vector (an one-order tensor) and a matrix (a two-order tensor) to higher orders. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ be the input tensor, whose order is denoted by N . Like rows and columns in a matrix, \mathcal{X} has N modes, whose lengths, also called dimensionality, are denoted by $I_1, \dots, I_N \in \mathbb{N}$, respectively. We assume that most entries of \mathcal{X} are zero (i.e., \mathcal{X} is sparse), as in many real-world tensors (Papalexakis et al, 2015; Perros et al, 2015; Smith and Karypis, 2017).

We denote general N -order tensors by boldface Euler script letters e.g., \mathcal{X} , while matrices and vectors are denoted by boldface capitals, e.g., \mathbf{A} , and boldface lowercases, e.g., \mathbf{a} , respectively. We use the MATLAB-like notations to indicate the entries of tensors. For example, $\mathcal{X}(i_1, \dots, i_N)$ (or $x_{i_1 \dots i_N}$ in short) indicates the (i_1, \dots, i_N) -th entry of \mathcal{X} . Similar notations are used for matrices and vectors. $\mathbf{A}(i, :)$ and $\mathbf{A}(:, j)$ (or $\bar{\mathbf{a}}_i$ and \mathbf{a}_j in short) indicate the i -th row and the j -th column of \mathbf{A} . The i -th entry of a vector \mathbf{a} is denoted by $\mathbf{a}(i)$ (or a_i in short).

2.2. Basic Tensor Terminologies and Operations

We review basic tensor terminologies and operations, which are the building blocks of Tucker decomposition. Table 1 lists the symbols frequently used in this paper.

Definition 1 (Fiber). *A mode- n fiber is an one-order section of a tensor, obtained by fixing all indices except the n -th index.*

Definition 2 (Slice). *A slice is a two-order section of a tensor, obtained by fixing all indices but two.*

Definition 3 (Tensor Unfolding/Matricization). *Unfolding, also known as matricization, is the process of re-ordering the entries of an N -order tensor into a matrix. The mode- n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ is a matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{q \neq n} I_q)}$ whose columns are the mode- n fibers.*

Definition 4 (N -order Outer Product). *The N -order outer product of vectors $\mathbf{v}_1 \in \mathbb{R}^{I_1}, \mathbf{v}_2 \in \mathbb{R}^{I_2}, \dots, \mathbf{v}_N \in \mathbb{R}^{I_N}$ is denoted by $\mathbf{v}_1 \circ \mathbf{v}_2 \circ \dots \circ \mathbf{v}_N$ and is an N -order tensor in $\mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$. Elementwise, we have*

$$[\mathbf{v}_1 \circ \mathbf{v}_2 \circ \dots \circ \mathbf{v}_N](i_1, \dots, i_N) = \mathbf{v}_1(i_1) \mathbf{v}_2(i_2) \dots \mathbf{v}_N(i_N).$$

For brevity, we use the following shorthand notations for outer products:

$$\begin{aligned} \circ_{(i_1, \dots, i_N)} \{\mathbf{A}\} &= \bar{\mathbf{a}}_{i_1}^{(1)} \circ \dots \circ \bar{\mathbf{a}}_{i_N}^{(N)}, \text{ and} \\ \circ_{(i_1, \dots, i_N)}^{-n} \{\mathbf{A}\} &= \bar{\mathbf{a}}_{i_1}^{(1)} \circ \dots \circ \bar{\mathbf{a}}_{i_{n-1}}^{(n-1)} \circ [1] \circ \bar{\mathbf{a}}_{i_{n+1}}^{(n+1)} \circ \dots \circ \bar{\mathbf{a}}_{i_N}^{(N)}. \end{aligned}$$

Definition 5 (n -mode Vector Product). *The n -mode vector product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ and a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is denoted by $\mathcal{X} \bar{\times}_n \mathbf{v}$, and is an $(N-1)$ -order tensor in $\mathbb{R}^{I_1 \times \dots \times I_{n-1} \times I_{n+1} \times \dots \times I_N}$. Elementwise, we have*

$$[\mathcal{X} \bar{\times}_n \mathbf{v}](i_1, \dots, i_{n-1}, i_{n+1}, \dots, i_N) = \sum_{i_n=1}^{I_n} x_{i_1 \dots i_N} v_{i_n}.$$

Definition 6 (n -mode Matrix Product). *The n -mode matrix product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J_n \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$, and is an N -order tensor in $\mathbb{R}^{I_1 \times \dots \times I_{n-1} \times J_n \times I_{n+1} \times \dots \times I_N}$. Elementwise, we have*

$$[\mathcal{X} \times_n \mathbf{U}](i_1, \dots, i_{n-1}, j_n, i_{n+1}, \dots, i_N) = \sum_{i_n=1}^{I_n} x_{i_1 \dots i_N} u_{j_n i_n}.$$

We adopt the shorthand notations in (Kolda and Sun, 2008) for all-mode matrix product and matrix product in every mode but one:

$$\begin{aligned} \mathcal{X} \times \{\mathbf{U}\} &\equiv \mathcal{X} \times_1 \mathbf{U}^{(1)} \dots \times_N \mathbf{U}^{(N)}, \text{ and} \\ \mathcal{X} \times_{-n} \{\mathbf{U}\} &\equiv \mathcal{X} \times_1 \mathbf{U}^{(1)} \dots \times_{n-1} \mathbf{U}^{(n-1)} \times_{n+1} \mathbf{U}^{(n+1)} \dots \times_N \mathbf{U}^{(N)}. \end{aligned}$$

2.3. Tucker decomposition

Tucker decomposition (Tucker, 1966) decomposes a tensor into a core tensor and N factor matrices so that the original tensor is approximated best. Specifically,

Algorithm 1: Tucker-ALS (also known as HOOI)

Input : \mathcal{X} , an N -order tensor of $\mathbb{R}^{I_1 \times \dots \times I_N}$.
 J_1, \dots, J_N , rank in each mode.
 T , the number of iterations.
Output : $\{\mathbf{A}\}$, a set of factor matrices $\{\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}\}$ where $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$.
 \mathcal{G} , an N -order core tensor of $\mathbb{R}^{J_1 \times \dots \times J_N}$.

```

1 Initialize all  $\mathbf{A}^{(n)}$ 
2 for  $t \leftarrow 1..T$  do
3   for  $n \leftarrow 1..N$  do
4      $\mathbf{Y}_{(n)} \leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^{(T)}\}]_{(n)}$ 
5      $\mathbf{A}^{(n)} \leftarrow \text{top-}J_n \text{ left singular vectors of } \mathbf{Y}_{(n)}$ 
6  $\mathcal{G} \leftarrow \mathbf{Y}_{(N)} \times_N \mathbf{A}^{(N)T}$ 
7 return  $\mathcal{G}, \{\mathbf{A}\}$ 
  
```

$\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ is approximated by

$$\mathcal{X} \approx \mathcal{G} \times \{\mathbf{A}\} = \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \dots \times_N \mathbf{A}^{(N)},$$

where (a) $\mathcal{G} \in \mathbb{R}^{J_1 \times \dots \times J_N}$, (b) J_n denotes the rank of the n -th mode, and (c) $\{\mathbf{A}\}$ is the set of factor matrices $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$, each of which is in $\mathbb{R}^{I_n \times J_n}$.

The most widely used way to solve Tucker decomposition is Tucker-ALS (Algorithm 1), also known as Higher Order Orthogonal Iteration (HOOI) (De Lathauwer et al, 2000).² It finds factor matrices whose columns are orthonormal by Alternating Least Squares (ALS). Since \mathcal{G} is uniquely computed by $\mathcal{X} \times \{\mathbf{A}^T\}$ once $\{\mathbf{A}\}$ is determined (Kolda and Sun, 2008), the objective function is simplified as

$$\max_{\{\mathbf{A}\}} \|\mathcal{X} \times \{\mathbf{A}^T\}\|. \quad (1)$$

2.4. Implicitly Restarted Arnoldi Method (IRAM)

Vector iteration (or power method) is one of the fundamental algorithms for solving large-scale Eigenproblem (Saad, 2011). For a given matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$, vector iteration finds the leading eigenvector corresponding to the largest eigenvalue by repeating the following updating rule from a randomly initialized $\mathbf{v}^{(0)} \in \mathbb{R}^n$.

$$\mathbf{v}^{(k+1)} = \frac{\mathbf{U}\mathbf{v}^{(k)}}{\|\mathbf{U}\mathbf{v}^{(k)}\|}.$$

As k increases, $\mathbf{v}^{(k+1)}$ converges to the leading eigenvector (Saad, 2011).

Arnoldi, which is a subspace iteration method, extends vector iteration to find k leading eigenvectors simultaneously. Implicitly Restarted Arnoldi Method (IRAM) is one of the most advanced techniques for Arnoldi (Saad, 2011). Briefly speaking, IRAM only keeps k orthonormal vectors that are a basis of the Krylov space, updates the basis until it converges, and then computes the k leading

² Other methods include the truncated higher-order singular value decomposition (THOSVD) (De Lathauwer et al, 2000) and the sequentially THOSVD (?).

eigenvectors from the basis. One virtue of IRAM is the *reverse communication interface*, which enables users to compute Eigendecomposition by viewing Arnoldi as a black box. Specifically, the leading k eigenvectors of a square matrix \mathbf{U} are obtained as follows:

- (1) User initializes an instance of IRAM.
- (2) IRAM returns $\mathbf{v}^{(j)}$ (initially $\mathbf{v}^{(0)}$).
- (3) User computes $\mathbf{v}' \leftarrow \mathbf{U}\mathbf{v}^{(j)}$, and gives \mathbf{v}' to IRAM.
- (4) After an internal process, IRAM returns new vector $\mathbf{v}^{(j+1)}$.
- (5) Repeat steps (3)–(4) until the internal variables in IRAM converges.
- (6) IRAM computes eigenvalues and eigenvectors from its internal variables, and it returns them.

For details of IRAM and the *reverse communication interface*, we refer interested readers to (Lehoucq et al, 1997; Saad, 2011).

3. Related Work

We describe the major challenges in scaling Tucker decomposition in Section 3.1. Then, in Section 3.2, we briefly survey the literature on scalable Tucker decomposition to see how these challenges have been addressed. However, we notice that existing methods still commonly suffer from *M-Bottleneck*, which is described in Section 3.3. Lastly, we briefly introduce scalable methods for other tensor decomposition methods in Section 3.4.

3.1. Intermediate Data Explosion

The most important challenge in scaling Tucker decomposition is the intermediate data explosion problem which was first identified in (Kolda and Sun, 2008) (Definition 7). It states that a naive implementation of Algorithm 1, especially the computation of $[\mathbf{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$, can produce huge intermediate data that do not fit in memory or even on a disk. We shall refer to this naive method as BaselineNaive.

Definition 7 (Intermediate Data Explosion in BaselineNaive (Kolda and Sun, 2008)). *Let M be the number of non-zero entries in \mathbf{X} . In Algorithm 1, naively computing $\mathbf{X} \times_{-n} \{\mathbf{A}^T\}$ requires $O(M \prod_{p \neq n} J_p)$ space for intermediate data.*³

For example, if we assume a 5-order tensor with $M = 100$ millions and $J_n = 10$ for all n , $M \prod_{p \neq n} J_p = 1$ trillions. Thus, if single-precision floating-point numbers are used, computing $\mathbf{X} \times_{-n} \{\mathbf{A}^T\}$ requires about 4TB space, which exceeds the capacity of a typical hard disk as well as RAM.

³ When the input tensor is sparse, a straightforward way of computing $\mathbf{X} \times_{-n} \{\mathbf{A}^T\}$ is to (1) compute $\mathbf{X}(i_1, \dots, i_N) \left[\circ_{i_1, \dots, i_N}^{-n} \{\mathbf{A}\} \right]_{(n)}$ for each non-zero element $\mathbf{X}(i_1, \dots, i_N)$ and (2) combine the results together. The result of $\mathbf{X}(i_1, \dots, i_N) \left[\circ_{i_1, \dots, i_N}^{-n} \{\mathbf{A}\} \right]_{(n)}$ takes $O(\prod_{p \neq n} J_p)$ space. Since there are M non-zero elements, $O(M \prod_{p \neq n} J_p)$ space is required in total.

Table 2. **The S-HOT family is space efficient.** The S-HOT family requires orders of magnitude less space than state-of-the-art methods. As an example, we assume a tensor where $N = 5$, $I_n = I = 10$ millions for every mode n , and $M = 100$ millions. We also assume that $J_n = J = 10$ for every mode n , and $B = 40\text{MB}$, where B is the memory budget for memoization in S-HOT_{memo}. The space required by IRAM is included in the space for output data. Note that all the methods have the same convergence properties (see Observation 1 in Section 4).

Methods	Locations	Input Data (in Theory)	Output Data (in Theory)	Intermediate Data (in Theory)
BaselineNaive	Memory	$O(NM)$	$O(NIJ + J^N)$	$O(MJ^{N-1})$
BaselineOpt (Kolda and Sun, 2008)	Memory	$O(NM)$	$O(NIJ + J^N)$	$O(IJ^{N-1})$
HATen2 (Jeon et al, 2015)	Memory	-	-	$O(J^{N-1} + \text{max. degree}^*)$
	Disk	$O(NM)$	$O(NIJ + J^N)$	$O(IJ^{N-1} + MN^2J)$
S-HOT_{space}	Memory	-	$O(NIJ + J^N)$	$O(I + J^{N-1})$
	Disk	$O(NM)$	-	-
S-HOT_{scan}	Memory	-	$O(NIJ + J^N)$	$O(J^{N-1})$
	Disk	$O(N^2M)$	-	-
S-HOT_{memo}	Memory	-	$O(NIJ + J^N)$	$O(B + J^{N-1})$
	Disk	$O(N^2M)$	-	-

* the degree of an n -th mode index is the number of non-zero entries with the index (see Definition 8 for a formal definition of degree)

Methods	Locations	Input Data (in Example)	Output Data (in Example)	Intermediate Data (in Example)
BaselineNaive	Memory	$\sim 2\text{GB}$	$\sim 2\text{GB}$	$\sim 4\text{TB}$
BaselineOpt (Kolda and Sun, 2008)	Memory	$\sim 2\text{GB}$	$\sim 2\text{GB}$	$\sim 400\text{GB}$
HATen2 (Jeon et al, 2015)	Memory	-	-	$\gtrsim 40\text{KB}$
	Disk	$\sim 2\text{GB}$	$\sim 2\text{GB}$	$\sim 500\text{GB}$
S-HOT_{space}	Memory	-	$\sim 2\text{GB}$	$\sim 40\text{MB}$
	Disk	$\sim 2\text{GB}$	-	-
S-HOT_{scan}	Memory	-	$\sim 2\text{GB}$	$\sim 40\text{KB}$
	Disk	$\sim 10\text{GB}$	-	-
S-HOT_{memo}	Memory	-	$\sim 2\text{GB}$	$\sim 40\text{MB}$
	Disk	$\sim 10\text{GB}$	-	-

3.2. Scalable Tucker decomposition

Memory Efficient Tucker (MET) (Kolda and Sun, 2008): MET carefully orders the computation of $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ in Algorithm 1 so that space required for intermediate data is reduced. Let $\mathcal{Y} = \mathcal{X} \times_{-n} \{\mathbf{A}^T\}$. Instead of computing entire \mathcal{Y} at a time, MET computes a part of it at a time. Depending on the unit computed at a time, MET has various versions, and $\text{MET}^{\text{fiber}}$ is the most space-efficient one.

In $\text{MET}^{\text{fiber}}$, each fiber (Definition 1) of \mathcal{X} is computed at a time. The specific equation when \mathcal{X} is a 3-way tensor is as follows:

$$\mathcal{Y}(:, j_2, j_3) \leftarrow \overbrace{\mathcal{X} \times_2 \mathbf{a}_{j_2}^{(2)} \times_3 \mathbf{a}_{j_3}^{(3)}}^{I_1}. \quad (2)$$

The amount of intermediate data produced during the computation of a fiber in \mathcal{Y} by Eq. (2) is only $O(I_1)$, and this amount is the same for general N -order tensors. However, since entire (matricized) \mathcal{Y} still needs to be materialized, $\text{MET}^{\text{fiber}}$ suffers from *M-Bottleneck*, which is discussed in Section 3.3. $\text{MET}^{\text{fiber}}$ is one of the most space-optimized tensor decomposition methods, and we shall refer to $\text{MET}^{\text{fiber}}$ as BaselineOpt from now on.

Hadoop Tensor Method (HaTen2) (Jeon et al, 2015): HATEN2, in the same spirit as MET, carefully orders the computation of $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ in Algorithm 1 on MapReduce so that the amount of intermediate data and the number of MapReduce jobs are reduced. Specifically, HATEN2 first computes $\mathcal{X} \times_p (\mathbf{A}^{(p)})^T$ for each $p \neq n$, then combines the results to obtain $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$. However, HATEN2 requires $O(MN \sum_{p \neq n} J_p)$ disk space for intermediate data, which is much larger than $O(I_n)$ space, which BaselineOpt requires.

Other Work Related to Scalable Tucker Decomposition: Several algorithms were proposed for the case when the input tensor \mathcal{X} is dense so that it cannot fit in memory. Specifically, (Tsourakakis, 2010) uses random sampling of non-zero entries to sparsify \mathcal{X} , and (Austin et al, 2016) distributes the entries of \mathcal{X} across multiple machines. However, in this chapter, we assume that \mathcal{X} is a large but sparse tensor, which is more common in real-world applications. Moreover, our method stores \mathcal{X} in disk, and thus the memory requirement does not depend on the number of non-zero entries (i.e., M).

Another line of research focused on reducing redundant computations that occur during a tensor-times-matrix chain operation (TTMc) (i.e., $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ in line 4 of Algorithm 2), which is the dominant computation in Tucker-ALS. It was observed in (Baskaran et al, 2012) that partial computations of TTMCs can be reused. For example, $\mathcal{X} \times_3 \mathbf{A}^{(3)T} \dots \times_N \mathbf{A}^{(N)T}$, which is a partial computation of $\mathcal{X} \times_{-1} \{\mathbf{A}^T\}$, can be reused when computing $\mathcal{X} \times_{-2} \{\mathbf{A}^T\}$. To exploit this, in (Baskaran et al, 2012), the N modes are partitioned into two groups: $N_1 := \{1, \dots, \lceil N/2 \rceil\}$ and $N_2 := \{\lceil N/2 \rceil + 1, \dots, N\}$. Then, $\mathcal{X} \times_1 \mathbf{A}^{(1)T} \dots \times_{\lceil N/2 \rceil} \mathbf{A}^{(\lceil N/2 \rceil)T}$ is stored and used for computing $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ for each $n \in N_2$. Similarly, $\mathcal{X} \times_{\lceil N/2 \rceil + 1} \mathbf{A}^{(\lceil N/2 \rceil + 1)T} \dots \times_N \mathbf{A}^{(N)T}$ is stored and used for computing $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ for each $n \in N_1$. In (Kaya and Ucar, 2016), partial computations of TTMCs are stored in the nodes of a binary tree and reused so that the number of n -mode products is limited to $\log(N)$ per TTMc. It was shown in (Smith and Karypis, 2017) that the partial computations can be reused “on the fly” and

faster without having to be stored, while an additional amount of user-specified memory can be used for further reducing the number of n -mode products. To this end, the input tensor is stored in the *compressed sparse fiber* (CSF) format (Smith and Karypis, 2015), where a tensor is stored as a forest of I_n trees with N levels so that each path from a root to a leaf encodes a non-zero entry. All these algorithms are parallelized in shared-memory (Baskaran et al, 2012; Smith and Karypis, 2017; Kaya and Uçar, 2016) and/or distributed-memory (Kaya and Uçar, 2016) environments, and a lightweight but near-optimal scheme for distributing the input tensor among processors was proposed in (?). Note that these algorithms do not suffer from intermediate data explosion (see Definition 7), which may occur in TTMcs (line 4 of Algorithm 1), by computing $\mathbf{Y}_{(n)}$ row by row. However, they suffer from *M-Bottleneck*, which is described in the following subsection, since they compute (truncated) singular value decomposition (as in line 5 of Algorithm 1) on materialized $\mathbf{Y}_{(n)}$.⁴ To minimize memory requirements, our proposed methods, described in Section 4, store the input tensor on disk in the coordinate format and stream its non-zero entries one by one. Alternatively, the CSF format (Smith and Karypis, 2015) can be used to reduce redundant computations that occur during TTMcs, as in (Smith and Karypis, 2017), while it requires additional memory space.

In (Oh et al, 2017; Choi et al, 2017), several algorithms were proposed for (coupled) Tucker decomposition when most entries of the input tensor are unobserved (or missing), and they were extended to heterogeneous platforms (?). However, since they have time complexities proportional to the number of observed entries, they are inefficient for fully observable tensors (i.e., tensors without missing entries), which our algorithms assume. A fully-observable tensor has $I_1 \times \dots \times I_N$ observed entries.

3.3. Limitation: M-bottleneck

Although BaselineOpt and HATen2 successfully reduce the space required for intermediate data produced while $\mathbf{Y}_{(n)} \leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$ is computed, they have an important limitation. Both algorithms materialize $\mathbf{Y}_{(n)}$, but its size $O(I_n \prod_{p \neq n} J_p)$ is usually huge, mainly due to I_n , and more seriously, it grows rapidly as N , I_n or $\{J_n\}_{n=1}^N$ increases. For example, as illustrated in Fig. 2 in Section 1, if we assume a 5-order tensor with $I_n = 10$ millions and $J_p = 10$ for every $p \neq n$, then $I_n \prod_{p \neq n} J_p = 100$ billions. Thus, if single-precision floating-point numbers are used, materializing $\mathbf{Y}_{(n)}$ in a dense matrix format requires about 400GB space, which exceeds the capacity of typical RAM. Note that simply storing $\mathbf{Y}_{(n)}$ in a sparse matrix format does not solve the problem since $\mathbf{Y}_{(n)}$ is usually *dense*.

Considering this fact and the results in Section 3.2, we summarize the amount of intermediate data required during the whole process of tucker decomposition in each algorithm in Table 2. Our proposed S-HOT algorithms, which are discussed in detail in the following section, require several orders of magnitude less space for intermediate data.

⁴ For example, see line 13 of Algorithm 4 in (Kaya and Uçar, 2016).

3.4. Scalable Algorithms for Other Tensor Decomposition Models

Comprehensive surveys on scalable algorithms for various tensor decomposition models can be found in (Papalexakis et al, 2016; Sael et al, 2015). Among other models except Tucker decomposition, PARAFAC decomposition, which can be seen as a special case of Tucker decomposition where the core tensor has only super-diagonal entries, has been widely used. Below, we summarize previous approaches for scalable PARAFAC decomposition:

- **Parallelize standard approaches:** Standard optimization algorithms including ALS, (stochastic) gradient descent, and coordinate descent, are optimized and parallelized in distributed-memory (Choi and Vishwanathan, 2014; Kaya and Uçar, 2015) and MAPREDUCE (Kang et al, 2012; Beutel et al, 2014; Jeon et al, 2015; Jeon et al, 2016; Shin et al, 2017) settings.
- **Sampling or Subdivision:** Smaller subtensors of the input tensor are obtained by sampling (Papalexakis et al, 2015) or subdivision (De Almeida and Kibangou, 2013; De Almeida and Kibangou, 2014). Then, each subtensor is factorized. After that, the factor matrices of the entire tensor are reconstructed from those of subtensors.
- **Compression:** In (Sidiropoulos and Kyrillidis, 2012; Cohen et al, 2015), the input tensor is compressed before being factorized.
- **Concise Representation:** Several data structures, including compressed sparse fiber (CSF) (Smith and Karypis, 2015), flagged coordinate (F-COO) (Li et al, 2018), hierarchical coordinate (HiCOO) (Li et al, 2018), have been developed for concisely representing tensors and accelerating tensor decomposition.
- **Memoization:** A sequence of matricized tensor times Khatri-Rao products (MTTKRPs) is the dominant computation in PARAFAC decomposition. In (Li et al, 2017), partial computations of MTTKRPs are memoized and reused to reduce redundant computations that occur during MTTKRPs.

4. Proposed Method: S-HOT

In this section, we develop a novel algorithm called S-HOT, which avoids *M-Bottleneck* caused by the materialization of \mathbf{Y} . S-HOT enables high-order Tucker decomposition to be performed even in an off-the-shelf workstation. In Table 3, the different versions of S-HOT are compared with baseline algorithms in terms of objectives, update equations, and materialized data.

Specifically, we focus on the memory-efficient computation of the following two steps (lines 4 and 5 of Algorithm 1):

$$\begin{aligned}\mathbf{Y}_{(n)} &\leftarrow [\mathbf{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)} \quad (\in \mathbb{R}^{I_n \times (\prod_{p \neq n} J_p)}) \\ \mathbf{A}^{(n)} &\leftarrow \text{top-}J_n \text{ left singular vectors of } \mathbf{Y}_{(n)}.\end{aligned}$$

Our key idea is to tightly integrate the above two steps, and compute the singular vectors through IRAM directly from \mathbf{X} without materializing the entire \mathbf{Y} at once. We also use the fact that top- J_n left singular vectors of $\mathbf{Y}_{(n)}$ are equivalent to the top- J_n eigenvectors of $\mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T \in \mathbb{R}^{I_n \times I_n}$.⁵ Specifically, if

⁵ Instead of computing the eigenvectors of $\mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T$, we can use directly obtain the sin-

Table 3. **Summary of the algorithms.** We show the key differences in the objectives, update equations, materialized data of the algorithms. The figures, where the colored regions need to be explicitly materialized in memory at once, illustrate how the algorithms work.

Method	BaselineNaive and BaselineOpt (Kolda and Sun, 2008)	S-HOT _{space}
Objective	Left singular vectors of $\mathbf{Y}_{(n)}$	
Update equations	$\mathbf{v}^{k+1} \leftarrow \mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T \mathbf{v}^k$	$\mathbf{s} \leftarrow \sum_{p \in \Theta(\mathcal{X})} v_{i_n}^k \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\} \right]_{(n)}$ $v_{i_n}^{k+1} \leftarrow \sum_{p \in \Theta_{i_n}^{(n)}(\mathcal{X})} \mathbf{s}^T \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\} \right]_{(n)}$
Materialization	$\mathbf{Y}_{(n)} \in \mathbb{R}^{I_n \times \prod_{p \neq n} J_p}$	$\mathbf{s} \in \mathbb{R}^{\prod_{p \neq n} J_p}$
Illustration		
Method	S-HOT _{scan}	S-HOT _{memo}
Objective	Right singular vectors of $\mathbf{Y}_{(n)}$	
Update equations	$\mathbf{y}_i \leftarrow \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\} \right]_{(n)}$	$\mathbf{w}^{k+1} \leftarrow \sum_{i=1}^{I_n} (\mathbf{y}_i^T \mathbf{w}^k) \mathbf{y}_i$
Materialization	$\mathbf{y}_i \in \mathbb{R}^{\prod_{p \neq n} J_p}$	
Illustration		

we use the *reverse communication interface* of IRAM, the above two steps are computed by simply updating \mathbf{v}' repeatedly as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T \mathbf{v}, \quad (3)$$

where we do not need to materialize $\mathbf{Y}_{(n)}$ (and thus we can avoid *M-Bottleneck*) if we are able to update \mathbf{v}' directly from the \mathcal{X} . Note that, using IRAM does not

gular vectors of $\mathbf{Y}_{(n)}$ using, for example, Lanczos bidiagonalization (Berry, 1992). We leave exploration of such variation for future work.

change the result of the above two steps. Thus, final results of Tucker decomposition are also not changed, while space requirements are reduced drastically, as summarized in Table 3.

The remaining problem is how to update \mathbf{v}' directly from \mathfrak{X} , which is stored in disk, without materializing $\mathbf{Y}_{(n)}$. To address this problem, we first examine a naive method extending BaselineOpt and then eventually propose S-HOT_{space}, S-HOT_{scan}, and S-HOT_{memo}, which are the three versions of S-HOT with distinct advantages.

Note that all our ideas described in this section do not change the outputs of BaselineNaive and BaselineOpt. Thus, all versions of S-HOT have the same convergence properties of BaselineNaive and BaselineOpt, as described in Observation 1.

Observation 1 (Convergence Property of S-HOT). *When all initial conditions are identical, S-HOT_{space}, S-HOT_{scan}, and S-HOT_{memo} give the same result of BaselineNaive and BaselineOpt after the same number of iterations.*

4.1. First step: “Naive S-HOT”

How can we avoid *M-Bottleneck*? In other words, how can we compute Eq. (3) without materializing the entire \mathbf{Y} ? We describe NAIVE S-HOT, which computes \mathbf{Y} fiber by fiber, for computing Eq. (3). Thus, NAIVE S-HOT computes \mathbf{v}' progressively on the basis of each column vector of $\mathbf{Y}_{(n)}$, which corresponds to a fiber in \mathbf{Y} , as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T \mathbf{v} = \sum_c \mathbf{y}_c (\mathbf{y}_c^T \mathbf{v}), \quad (4)$$

where $\mathbf{y}_c \in \mathbb{R}^{I_n}$ is a column vector of $\mathbf{Y}_{(n)}$.

This equation can be reformulated by \mathfrak{X} and $\{\mathbf{A}^T\}$. For ease of explanation, let \mathfrak{X} be a 3-order tensor. For each column vector \mathbf{y}_c , there exists a fiber $\mathbf{y}(:, j_2, j_3)$ corresponding to \mathbf{y}_c . By plugging Eq. (2) into Eq. (4), we obtain

$$\begin{aligned} \mathbf{v}' &\leftarrow \sum_c \mathbf{y}_c (\mathbf{y}_c^T \mathbf{v}) = \sum_{\forall(j_2, j_3)} \mathbf{y}(:, j_2, j_3) \left(\mathbf{y}(:, j_2, j_3)^T \mathbf{v} \right) \\ &= \sum_{\forall(j_2, j_3)} \left(\mathfrak{X} \bar{\times}_2 \mathbf{a}_{j_2}^{(n)} \bar{\times}_3 \mathbf{a}_{j_3}^{(n)} \right) \left(\left(\mathfrak{X} \bar{\times}_2 \mathbf{a}_{j_2}^{(n)} \bar{\times}_3 \mathbf{a}_{j_3}^{(n)} \right)^T \mathbf{v} \right). \end{aligned}$$

As clarified in Eq. (2), $\mathfrak{X} \bar{\times}_2 \mathbf{a}_{j_2}^{(n)} \bar{\times}_3 \mathbf{a}_{j_3}^{(n)}$ is computed within $O(I_1)$ space, which is significantly smaller than space required for $\mathbf{Y}_{(n)}$.

However, NAIVE S-HOT is impractical because the number of scans of \mathfrak{X} increases explosively, as stated in Lemma 1 and Lemma 2.

Lemma 1 (Scan Cost of Computing a Fiber). *Computing a fiber on the fly requires a complete scan of \mathfrak{X} .*

Proof. Computing a fiber consists of multiple n -mode vector products. Each n -mode vector product is considered as a weighted sum of $(N - 1)$ -order section

of \mathcal{X} as follows:

$$\mathcal{X}_{\bar{\times}_n} \mathbf{v} = \sum_{i_n=1}^{I_n} \mathcal{X}(\underbrace{:, \dots, :}_{n-1}, i_n, \underbrace{:, \dots, :}_{N-n}) \mathbf{v}_{i_n}. \quad (5)$$

Thus, a complete scan of \mathcal{X} is required to compute a fiber. \blacksquare

Lemma 2 (Minimum Scan Cost of NAIVE S-HOT). *Let B be the memory budget, i.e., the number of floating-point numbers that can be stored in memory at once. Then, NAIVE S-HOT requires at least $\frac{I_n}{B} \prod_{p \neq n} J_p$ scans of \mathcal{X} for computing Eq. (4).*

Proof. Since we compute $\mathbf{y}_c (\mathbf{y}_c^T \mathbf{v})$, \mathbf{y}_c should be stored in memory requiring I_n space, until the computation of $\mathbf{y}_c^T \mathbf{v}$ finishes. Thus, we can compute at most $\frac{B}{I_n}$ fibers at the same time within one scan of \mathcal{X} . Therefore, NAIVE S-HOT requires at least $\frac{I_n}{B} \prod_{p \neq n} J_p$ scans of \mathcal{X} to compute Eq. (4). \blacksquare

4.2. Proposed: “S-HOT_{space}”

How can we avoid the explosion in the number of scans of the input tensor required in NAIVE S-HOT? We propose S-HOT_{space}, which computes Eq. (3) within two scans of \mathcal{X} . S-HOT_{space} progressively computes \mathbf{v}' from each row vector of $\mathbf{Y}_{(n)}$. Specifically, \mathbf{v}' is computed by:

$$1 \leq \forall i \leq I_n, \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i \mathbf{Y}_{(n)}^T \mathbf{v} = \bar{\mathbf{y}}_i \sum_{k=1}^{I_n} \mathbf{v}(k) \bar{\mathbf{y}}_k^T \quad (6)$$

where $\bar{\mathbf{y}}_i$ is the i th row vector of $\mathbf{Y}_{(n)}$, which corresponds to an $(N-1)$ -order segment of \mathcal{Y} where the n -th mode index is fixed to i . When entire \mathcal{Y} does not fit in memory, Eq. (6) should be computed in the following two steps:

$$\mathbf{s} \leftarrow \sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T \quad (7)$$

$$1 \leq \forall i \leq I_n, \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i \mathbf{s}. \quad (8)$$

This is since we cannot store all $\bar{\mathbf{y}}_i$ in memory until the computation of $\sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T$ finishes.

A pictorial description and a formal description of S-HOT_{space} are provided in Fig. 3 and Algorithm 2, respectively. As shown in Lemma 3, S-HOT_{space} requires two scans of \mathcal{X} for computing Eq. (3).

Lemma 3 (Scan Cost of S-HOT_{space}). *S-HOT_{space} requires two scans of \mathcal{X} for computing Eq. (3).*

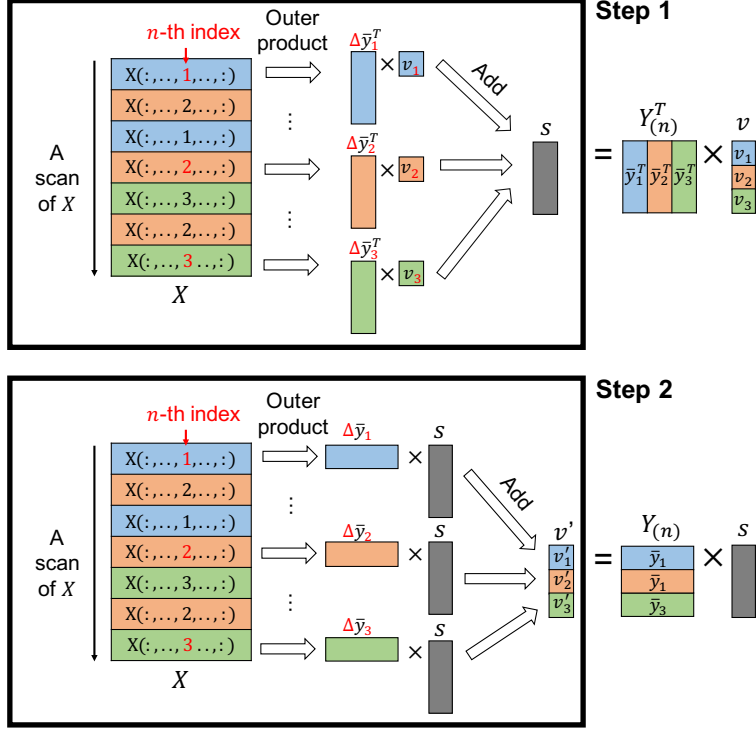


Fig. 3. Illustration of the two steps of S-HOT_{space} for computing Eq. (6). Note that we scan the non-zero entries of \mathcal{X} once during each step. (a) **First step for Eq. (7)**: For each non-zero element, we add its contribution to s . To compute the contribution to s , we first compute the contribution to the corresponding row of $Y_{(n)}$ (i.e., $\Delta \bar{y}_i$ where i is the n -th mode index of the element) by outer products and then multiply it and the corresponding element of v (i.e., $v(i)$). (b) **Second step for Eq. (8)**: For each non-zero element, we add its contribution to the corresponding entry of v' (i.e., $v'(i)$ where i is the n -th mode index of the element). To compute the contribution to $v'(i)$, we first compute the contribution to the corresponding row of $Y_{(n)}$ (i.e., $\Delta \bar{y}_i$) by outer products and then multiply it and s , which is obtained in the first step.

Proof. Each \bar{y}_i can be computed as follows.

$$\begin{aligned}
 \bar{y}_i &\leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}(i, :) = \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \times_{-n} \{\mathbf{A}^T\} \\
 &= \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) [\text{o}_p^{-n} \{\mathbf{A}\}]_{(n)}, \tag{9}
 \end{aligned}$$

where p is a tuple (i_1, \dots, i_N) whose n -th mode index is fixed to i ; $\mathcal{X}(p)$ is an entry specified by p . Based on each \bar{y}_i , Eq. (7) can be computed progressively

as follows:

$$\begin{aligned} \mathbf{s} &\leftarrow \sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T = \sum_{i=1}^{I_n} \mathbf{v}(i) \sum_{p \in \Theta_i^{(n)}(\mathbf{X})} \mathbf{X}(p) [\circ_p^{-n} \{\mathbf{A}\}]_{(n)} \\ &= \sum_{p \in \Theta(\mathbf{X})} \mathbf{v}(i_n) \mathbf{X}(p) [\circ_p^{-n} \{\mathbf{A}\}]_{(n)}. \end{aligned} \quad (10)$$

Thus, computing Eq. (7) requires only one scan of \mathbf{X} . Similarly, Eq. (8) also can be computed within one scan of \mathbf{X} . Therefore, Eq. (6), which consists of Eq. (7) and Eq. (8), can be computed within two scans of \mathbf{X} . ■

In Lemma 4, we prove the amount of space required by S-HOT_{scan} for intermediate data.

Lemma 4 (Space Complexity in S-HOT_{space}). *The update step of S-HOT_{space} (lines 16-22 of Algorithm 2) requires*

$$O\left(\max_{1 \leq n \leq N} (I_n + \prod_{p=1}^N J_p/J_n)\right)$$

memory space for intermediate data.

Proof. S-HOT_{space} maintains \mathbf{v} , \mathbf{v}' , and \mathbf{s} in its update step. When each factor matrix $\mathbf{A}^{(n)}$ is updated, \mathbf{v} and \mathbf{v}' are I_n by 1 vectors, and \mathbf{s} is a $\prod_{p=1}^N J_p/J_n$ by 1 vector. Thus, $O(I_n + \prod_{p=1}^N J_p/J_n)$ space is required in the update step for each factor matrix $\mathbf{A}^{(n)}$. Since the factor matrices are update one by one, $O(\max_{1 \leq n \leq N} (I_n + \prod_{p=1}^N J_p/J_n))$ space is required at a time. ■

4.3. Faster: “S-HOT_{scan}”

How can we further reduce the number of required scans of the input tensor? We propose S-HOT_{scan}, which halves the number of scans of \mathbf{X} at the expense of requiring multiple (disk-resident) copies of \mathbf{X} sorted by different mode indices. In effect, S-HOT_{scan} trades off disk space for speed.

Our key idea for the further optimization is to compute J_n right leading singular vectors of $\mathbf{Y}_{(n)}$, which are eigenvectors of $\mathbf{Y}_{(n)}^T \mathbf{Y}_{(n)}$, and use the result to compute the left singular vectors. Let $\mathbf{Y}_{(n)} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T$ be the SVD of $\mathbf{Y}_{(n)}$. Then,

$$\mathbf{Y}_{(n)} \mathbf{V} \mathbf{\Sigma}^{-1} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T \mathbf{V} \mathbf{\Sigma}^{-1} = \mathbf{U}. \quad (11)$$

Thus, left singular vectors are obtained from right singular vectors.

S-HOT_{scan} computes top- J_n right singular vectors of $\mathbf{Y}_{(n)}$ by updating the vector $\mathbf{w} \in \mathbb{R}^{\prod_{p \neq n} J_p}$ as follows:

$$\mathbf{w}' \leftarrow \mathbf{Y}_{(n)}^T \mathbf{Y}_{(n)} \mathbf{w} = \sum_{i=1}^{I_n} (\bar{\mathbf{y}}_i^T \mathbf{w}) \bar{\mathbf{y}}_i. \quad (12)$$

The virtue of S-HOT_{scan} is that it requires only one scan of \mathbf{X} for calculating Eq. (12), as stated in Lemma 6.

Algorithm 2: Formal description for S-HOT.

Input : N -order tensor: \mathcal{X} ,
Rank in each mode: $J_1 \times \dots \times J_N$
Output: Core tensor: $\mathcal{G} \in \mathbb{R}^{J_1 \times \dots \times J_N}$,
Factor matrices: $\{\mathbf{A}\}$

- 1 Initialize $\{\mathbf{A}\}$
- 2 **repeat**
- 3 **for** $n \leftarrow 1 \dots N$ **do**
- 4 $\mathbf{v} \leftarrow \text{IRAM_init}(I_n, J_n)$
- 5 **repeat**
- 6 $\mathbf{v}' \leftarrow \text{UpdateMethod}(\mathcal{X}, n, \mathbf{v})$
- 7 $\mathbf{v} \leftarrow \text{IRAM_doIter}(\mathbf{v}')$
- 8 **until** $\text{IRAM_isconv}()$;
- 9 $\mathbf{A}^{(n)} \leftarrow \text{getSingularVec}()$
- 10 **until** *terminal condition*;
- 11 $\mathcal{G} \leftarrow \mathcal{X} \times \{\mathbf{A}^T\}$
- 12 **return** $\mathcal{G}, \{\mathbf{A}\}$

- 13 **Subroutine** $\text{NaiveTucker}(\mathcal{X}, n, \mathbf{v})$
- 14 Materialize $\mathbf{Y}_{(n)} = [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$ if it does not exist.
- 15 **return** $\mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^T \mathbf{v}$

- 16 **Subroutine** $\text{S-HOT}_{\text{space}}(\mathcal{X}, n, \mathbf{v})$
- 17 $\mathbf{s}, \mathbf{v}' \leftarrow \mathbf{0}$
- 18 **forall** $(i_1, \dots, i_N) \in \Theta(\mathcal{X})$ **do**
- 19 $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{v}_{i_n} \mathcal{X}(i_1, \dots, i_N) [\circ_{(i_1, \dots, i_N)}^{-n} \{\mathbf{A}\}]_{(n)}$
- 20 **forall** $(i_1, \dots, i_N) \in \Theta(\mathcal{X})$ **do**
- 21 $\mathbf{v}'_{i_n} \leftarrow \mathbf{v}'_{i_n} + \mathbf{s}^T \mathcal{X}(i_1, \dots, i_N) [\circ_{(i_1, \dots, i_N)}^{-n} \{\mathbf{A}\}]_{(n)}$
- 22 **return** \mathbf{v}'

- 23 **Subroutine** $\text{S-HOT}_{\text{scan}}(\mathcal{X}, n, \mathbf{w})$
- 24 $\mathbf{w}' \leftarrow \mathbf{0}$
- 25 **for** $i \leftarrow 1 \dots I_n$ **do**
- 26 $\mathbf{y}_i \leftarrow \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) [\circ_p^{-n} \{\mathbf{A}\}]_{(n)}$
- 27 $\mathbf{w}' \leftarrow \mathbf{w}' + (\mathbf{y}_i^T \mathbf{w}) \mathbf{y}_i$
- 28 Deallocate \mathbf{y}_i
- 29 **return** \mathbf{w}'

Lemma 5 (Scan Cost for Computing $\bar{\mathbf{y}}_i$). $\bar{\mathbf{y}}_i$ can be computed by scanning only the entries of \mathcal{X} whose n -mode index is i .

Proof. Proven by Eq. (9). ■

Lemma 6 (Scan Cost of S-HOT_{scan}). S-HOT_{scan} computes Eq. (12) within one scan of \mathcal{X} when \mathcal{X} is sorted by the n -mode index.

Proof. By Lemma 5, only a section of tensor whose n -mode index is i is required for computing $\bar{\mathbf{y}}_i$. If \mathcal{X} is sorted by the n th mode index, we can sequentially compute each \mathbf{y}_i on the fly. Moreover, once $\bar{\mathbf{y}}_i$ is computed, we can immediately compute $(\bar{\mathbf{y}}_i^T \mathbf{w}) \bar{\mathbf{y}}_i$. After that, we do not need $\bar{\mathbf{y}}_i$ anymore, and can discard it. Thus, Eq. (12) can be computed on the fly within only a single scan of \mathcal{X} . ■

In this paper, we satisfy the sort constraint for all modes by simply keeping N copies of \mathcal{X} sorted by each mode index.

A formal description for S-HOT_{scan} is in Algorithm 2. It is assumed that \mathbf{w} is initialized by passing $(\prod_{p \neq n} J_p, J_n)$ instead of (I_n, J_n) at Line 4. Although one additional scan of \mathcal{X} is required for computing left singular vectors from the obtained right singular vectors (Eq. (11)), S-HOT_{scan} still requires fewer scans of \mathcal{X} than S-HOT_{space} since it saves one scan during \mathbf{w}' computation, which is repeated more frequently.

In Lemma 7, we prove the amount of space required by S-HOT_{scan} for intermediate data.

Lemma 7 (Space Complexity of S-HOT_{scan}). The update step of S-HOT_{scan} (lines 23-29 of Algorithm 2) requires

$$O\left(\max_{1 \leq n \leq N} \left(\prod_{p=1}^N J_p / J_n\right)\right)$$

memory space for intermediate data.

Proof. In its update step for each factor matrix $\mathbf{A}^{(n)}$, S-HOT_{scan} maintains \mathbf{w} , \mathbf{w}' , and \mathbf{y}_i at a time. All of them are $\prod_{p=1}^N J_p / J_n$ by 1 vectors. Thus, $O(\prod_{p=1}^N J_p / J_n)$ space is required in the update step for each factor matrix $\mathbf{A}^{(n)}$. Since the factor matrices are updated one by one, $O(\max_{1 \leq n \leq N} (\prod_{p=1}^N J_p / J_n))$ space is required at a time. ■

4.4. Even Faster: “S-HOT_{memo}”

How can we make good use of remaining memory when memory is underutilized by S-HOT_{scan}, which requires little space for intermediate data (see Table 2)? We propose S-HOT_{memo}, which improves S-HOT_{scan} in terms of speed by introducing the memoization technique. The memoization technique leverages the spare memory space, which we call *memo*, to store a part of intermediate data (i.e., some rows of $\mathbf{Y}_{(n)}$) in memory instead of computing all of them on-the-fly. Especially, within a given memory budget, S-HOT_{memo} carefully decides the rows of $\mathbf{Y}_{(n)}$ to be memoized so that the speed gain is maximized. A formal description of S-HOT_{memo} is given in Algorithm 3, where the steps added for memoization (i.e., lines 5, 14-19, 23-24) are in red.

Given a memory budget B , let k_n be the maximum number of rows of $\mathbf{Y}_{(n)}$ that can be memoized within B . When updating each factor matrix $\mathbf{A}^{(n)}$, S-HOT_{memo} memoizes the k_n rows that are most expensive to compute. Such rows

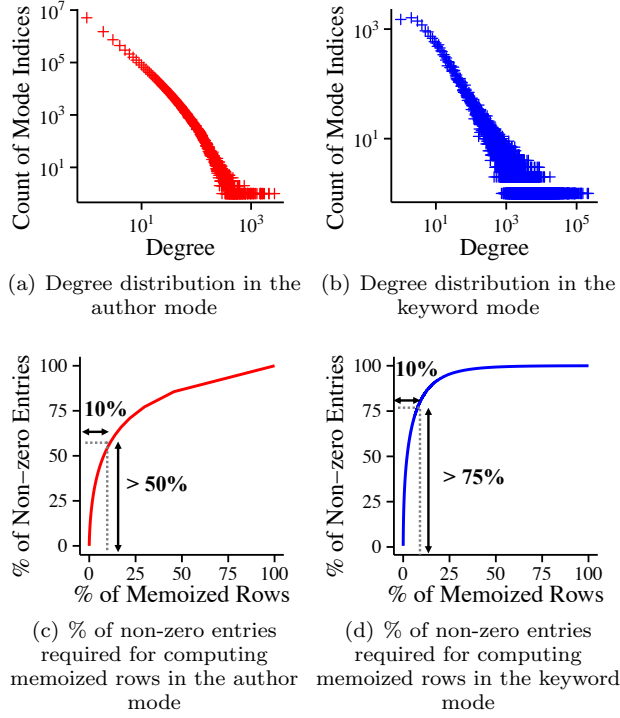


Fig. 4. Power-law degree distributions in the Microsoft Academic Graph dataset (see Section 5.3 for a description of the dataset). (a) and (b) show the skewed degree distributions in the author and keyword modes, which are exploited by S-HOT_{memo} for speed-up. (c) and (d) show that S-HOT_{memo} can avoid accessing many (e.g., 50-75%) non-zero entries by memoizing a small percentage (e.g., 10%) of rows.

can be found by comparing the degrees of the n -th mode indices (see Definition 8 for the definition of degree), as described in lines 14-19.

Definition 8 (Degree of Mode Indices). *The degree of each n -th mode index i is defined as $|\Theta_i^{(n)}(\mathcal{X})|$, i.e., the number of non-zero entries whose n -th mode index is i .*

This is because computing each row \mathbf{y}_i of $\mathbf{Y}_{(n)}$ takes time proportional the degree of n -th mode index i (i.e., $|\Theta_i^{(n)}(\mathcal{X})|$), as shown in Eq. (9). The remaining steps for updating $\mathbf{A}^{(n)}$ are the same as those of S-HOT_{scan} except for that the memoized rows are used, as described in lines 20-29.

This careful choice of the rows of $\mathbf{Y}_{(n)}$ to be memoized in memory is crucial to speed up the algorithm. This is because, in real-world tensors, the degree of mode indices often follows a power-law distribution (Clauset et al, 2017), and thus there exist indices with extremely high degree (see Figs. 4(a) and 4(b) for examples). By memoizing the rows of $\mathbf{Y}_{(n)}$ corresponding to such high-degree indices, S-HOT_{memo} avoids accessing many non-zero entries (see Figs. 4(c) and 4(d) for

Algorithm 3: Formal description for S-HOT_{memo}. The steps added for memoization (i.e., lines 5, 14-19, 23-24) are in **red**.

Input : N -order tensor: \mathcal{X} ,
Rank in each mode: $J_1 \times \dots \times J_N$,
Memory budget for memoization: B (or equivalently k_1, \dots, k_N)

Output: Core tensor: $\mathcal{G} \in \mathbb{R}^{J_1 \times \dots \times J_N}$,
Factor matrices: $\{\mathbf{A}\}$

```

1 Initialize  $\{\mathbf{A}\}$ 
2 repeat
3   for  $n \leftarrow 1 \dots N$  do
4      $\mathbf{v} \leftarrow \text{IRAM\_init}(I_n, J_n)$ 
5      $\text{Map} \leftarrow \text{Memoization}(\mathcal{X}, n, k_n)$ 
6     repeat
7        $\mathbf{v}' \leftarrow \text{Update}(\mathcal{X}, n, \mathbf{v}, \text{Map})$ 
8        $\mathbf{v} \leftarrow \text{IRAM\_doIter}(\mathbf{v}')$ 
9     until  $\text{IRAM\_isconv}()$ ;
10     $\mathbf{A}^{(n)} \leftarrow \text{getSingularVec}()$ 
11 until terminal condition;
12  $\mathcal{G} \leftarrow \mathcal{X} \times \{\mathbf{A}^T\}$ 
13 return  $\mathcal{G}, \{\mathbf{A}\}$ 

14 Subroutine Memoization( $\mathcal{X}, n, k_n$ )
15    $\text{Top} \leftarrow$  top- $k_n$  highest-degree  $n$ -th mode indices
16    $\text{Map} \leftarrow$  an empty map
17   forall  $i \in \text{Top}$  do
18      $\text{Map.put}(i, \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) [\circ_p^{-n} \{\mathbf{A}\}]_{(n)})$ 
19   return  $\text{Map}$ 

20 Subroutine Update( $\mathcal{X}, n, \mathbf{w}, \text{Map}$ )
21    $\mathbf{w}' \leftarrow \mathbf{0}$ 
22   for  $i \leftarrow 1 \dots I_n$  do
23     if  $i \in \text{Map.keys}()$  then
24        $\mathbf{y}_i \leftarrow \text{Map.get}(i)$ 
25     else
26        $\mathbf{y}_i \leftarrow \sum_{p \in \Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) [\circ_p^{-n} \{\mathbf{A}\}]_{(n)}$ 
27      $\mathbf{w}' \leftarrow \mathbf{w}' + (\mathbf{y}_i^T \mathbf{w}) \mathbf{y}_i$ 
28   Deallocate  $\mathbf{y}_i$ 
29   return  $\mathbf{w}'$ 

```

examples) and thus saves considerable computation time, as shown empirically in Section 5.5.

We prove the scan cost of S-HOT_{memo} in Lemma 8 and the space complexity of S-HOT_{memo} in Lemma 9.

Lemma 8 (Scan Cost of S-HOT_{memo}). S-HOT_{memo} computes Eq. (12) within one scan of \mathcal{X} when \mathcal{X} is sorted by the n -mode index.

Proof. Given memoized rows, S-HOT_{memo} computes Eq. (12) in the same way as does S-HOT_{scan} only except for that S-HOT_{memo} uses the memoized rows. Thus, S-HOT_{memo} and S-HOT_{scan} require the same number of scans of \mathcal{X} , which is one, as shown in Lemma 6. We do not need an additional scan of \mathcal{X} for the memoization step if it is done while Eq. (12) is first computed. ■

Lemma 9 (Space Complexity of S-HOT_{memo}). *The update step of S-HOT_{memo} (lines 20-29 of Algorithm 3) requires*

$$O\left(B + \max_{1 \leq n \leq N} \left(\prod_{p=1}^N J_p/J_n\right)\right)$$

memory space for intermediate data.

Proof. In addition to those maintained in S-HOT_{scan}, which require $O\left(\max_{1 \leq n \leq N} \left(\prod_{p=1}^N J_p/J_n\right)\right)$ space at a time (see Lemma 7), S-HOT_{memo} maintains the memoized rows, whose size is within the given budget B . Thus, $O(B + \max_{1 \leq n \leq N} \left(\prod_{p=1}^N J_p/J_n\right))$ space is required at a time. ■

Table 3 summaries the key differences of BaselineOpt, S-HOT, S-HOT_{scan}, S-HOT_{memo} in terms of objective, update equations, and materialized data of methods. The table also presents the figures illustrating how the methods work.

5. Experiments

In this section, we present experimental results supporting our claim that S-HOT outperforms state-of-the-art baselines. Specifically, our experiments are designed to answer the following two questions:

- **Q1.** How scalable is S-HOT compared to the state-of-the-art competitors with respect to the dimensionality, the rank, the order, and the number of non-zero entries?
- **Q2.** Can S-HOT decompose real-world tensors that are both large-scale and high-order?
- **Q3.** How does the memory budget affect the speed of S-HOT_{memo}?
- **Q4.** How does the degree distribution of the input tensor affect the speed of S-HOT_{memo}?

5.1. Experimental setting

Competitors: Throughout all experiments, we use two baseline methods and three versions of our proposed method:

- (1) BaselineNaive: a naive method computing $\mathcal{X} \times_{-n} \{\mathbf{A}\}$ in a straight-forward way.
- (2) BaselineOpt (Kolda and Sun, 2008): the state-of-the-art Memory Efficient Tucker decomposition which computes \mathcal{Y} fiber by fiber.
- (3) S-HOT_{space} (Section 4.2): the most space-efficient version of S-HOT.

Table 4. Statistics of the real-world tensors used in our experiments.

Tensor	Non-zeros	Order	Dimensionality
LBNL	1,698,825	5	$1,605 \times 4,198 \times 1,631 \times 4,209 \times 868,131$
MS Academic Graph	35,400,035	4	$9,380,418 \times 18,894 \times 2,016 \times 37,000$
Enron	54,202,099	4	$6,066 \times 5,699 \times 244,268 \times 1,176$

- (4) S-HOT_{scan} (Section 4.3): a faster version of S-HOT.
- (5) S-HOT_{memo} (Section 4.4): the fastest version of S-HOT with the memoization technique. We set the size of memo so that we can memoize up to 30 rows of $\mathbf{Y}_{(n)}$ for each n -th mode unless otherwise stated.⁶

For BaselineOpt and BaselineNaive, we use the implementation in MATLAB Tensor Toolbox 2.6 (Bader and Kolda, 2015). We exclude HATEN2 because HATEN2 is designed for Hadoop, and thus it takes too much time in a single machine. For example, in order to decompose a synthetic tensor with default parameters, HATEN2 takes 10,700 seconds for an iteration, which is almost **100× slower** than S-HOT_{scan}.

Real-world Datasets: We use the following high-order real-world tensors, whose statistics are given in Table 4:

- LBNL (Smith et al, 2017): This dataset, which was collected during the Traces project (Pang, 2006), contains information on internal network traffics from LBNL/ICSI. It is modeled as a 5-order tensor whose modes are sender IPs, sender ports, destination IPs, destination ports, and timestamps.
- MS Academic Graph (Sinha et al, 2015): This dataset is a snapshot of the Microsoft Academic Graph on Feb 5, 2016. It contains 42 million papers; 1,283 conferences and 23,404 journals; 115 million authors; and 53,834 keywords used to annotate the topics of the papers. It is modeled as a 4-order tensor whose modes are authors, venues, years, and keywords. Since the papers with missing attributes are ignored, the final tensor is of size $9,380,418 \times 18,894 \times 2,016 \times 37,000$.
- Enron (Smith et al, 2017): This dataset, which was collected for an investigation by the federal energy regulatory commission, contains information on emails from or to the employees of Enron Corporation (Shetty et al, 2004). It is modeled as a 4-order tensor whose modes are senders, receivers, words, and dates, respectively.

Note that both BaselineNaive and BaselineOpt methods fail to run on the three real-life tensors due to “out-of-memory” errors, while our S-HOT family successfully decompose them.

Synthetic Datasets: We also use synthetic tensors mainly to evaluate the scalability of methods with respect to various factors (i.e., the dimensionality, rank, order, and the number of non-zero entries) by controlling each factor while fixing the others. We generate synthetic tensors where the degree of their mode indices follows a Zipf distribution, which is common in real-world data (Powers, 2017; Adamic and Huberman, 2017). Specifically, to create an N -order

⁶ The size of the memo never exceeds 200KB unless otherwise stated.

tensor, we sample M entries where each n -th mode index is sampled from the following probability density function:

$$p(x) = \frac{x^{-\alpha}}{\sum_{k=1}^{\infty} \frac{1}{k^x}},$$

where α is a parameter determining the skewness of the distribution. We set the value of each entry to 1.⁷ As default parameter values, we use $N = 4$, $M = 10^5$, $I_n = 10^3$ for every n , $J_n = 8$ for every n , and $\alpha = 1.5$. These default values are chosen to effectively compare the scalability of competitors. We show that baselines (i.e., BaselineNaive and BaselineOpt) run out of memory if we increase these values. All experiments using synthetic datasets are repeated nine times (three times for each of three randomly generated tensors), and reported values are the average of the multiple trials.

Equipment: All experiments are conducted on a machine with Intel Core i7 4700@3.4GHz (4 cores), 32GB RAM, and Ubuntu 14.04 trusty. Every version of S-HOT is implemented in C++ with OpenMP library and AVX instruction set; and the source code is available at <http://dm.postech.ac.kr/shot>. We used ARPACK (Lehoucq et al, 1997), which implements IRAM supporting *reverse communication interface*. It is worth noting that ARPACK is an underlying package for a built-in function called `eigs()`, which is provided in many popular numerical computing environments including SCIPY, GNU OCTAVE, and MATLAB. Therefore, S-HOT is numerically stable and has the similar reconstruction error with `eigs()` function in the above mentioned numerical computing environments.

For fairness, we must note that, a fully optimized C++ implementation could potentially be faster than that of MATLAB, (although that is unlikely, since MATLAB is extremely well optimized for matrix operations). But in any case, our main contribution still holds: regardless of programming languages, S-HOT scales to much larger settings, thanks to our proposed “on-the-fly” computation (Equations (6) and (12)).

5.2. Q1: Scalability of S-HOT

We evaluate the scalability of the competing methods with respect to various factors: (1) the order, (2) the dimensionality, (3) the number of non-zero entries, and (4) the rank. Specifically, we measure the wall-clock time of a single iteration of each algorithm on synthetic tensors. Note that all the methods have the same convergence properties, as described in Observation 1 in Section 4.

Order: First, we investigate the scalability of the considered methods with respect to the order by controlling the order of the input tensor from 3 to 6 while fixing the other factors to their default values. As shown in Fig. 1(a), **S-HOT outperforms baselines**. BaselineNaive fails to decompose the 4-order tensor because it suffers from the *intermediate explosion problem*. BaselineOpt, which avoids the problem, is more memory-efficient than BaselineNaive. However, it

⁷ However, this does not mean that S-HOT is limited to binary tensors nor our implementation is optimized for binary tensors. We choose binary tensors for simplicity. Generating realistic values, while we control each factor, is not a trivial task.

fails to decompose a tensor whose order is higher than 4 due to *M-Bottleneck*. On the contrary, every version of S-HOT successfully decomposes even the 6-order tensor. Especially, S-HOT_{memo} is up to $50\times$ faster than S-HOT_{space} and S-HOT_{scan}.

Dimensionality: Second, we investigate the scalability of the considered methods with respect to the dimensionality. Specifically, we increase the dimensionality I_n of every n from 10^3 to 10^7 . That is, since the default order is 4, we increase the tensor from $10^3 \times 10^3 \times 10^3 \times 10^3$ to $10^7 \times 10^7 \times 10^7 \times 10^7$. As shown in Fig. 1(b), **S-HOT is several orders of magnitude scalable than the baselines**. Specifically, BaselineNaive fails to decompose any 4-order tensor, and thus it does not appear in the plot. BaselineOpt fails to decompose tensors with dimensionality larger than 10^4 since the space for storing \mathcal{Y} increases rapidly with respect to the size of dimensionality (*M-Bottleneck*). On the contrary, every version of S-HOT successfully decomposes the largest tensor of size $10^7 \times 10^7 \times 10^7 \times 10^7$. Moreover, the running times of S-HOT_{scan} and S-HOT_{memo} are almost constant since they solve the transposed problem, whose size is independent of the dimensionality. Between them, S-HOT_{memo} is up to $6\times$ faster than S-HOT_{scan}. On the other hand, the running time of S-HOT_{space} depends on dimensionality and increases as the dimensionality becomes greater than 10^6 . For smaller dimensionalities, however, the effect of dimensionality on its running time is negligible because the outer products (i.e., lines 19 and 21 of Algorithm 2) are the major bottleneck.

Rank: Third, we investigate the scalability of the considered methods with respect to the rank. To show the difference between the competitors clearly, we set the dimensionality of the input tensor to 20,000 in this experiment. However, the overall trends do not depend on the parameter values. As shown in Fig. 1(c), **the S-HOT has better scalability than baselines**. Specifically, BaselineNaive fails to decompose any tensor and does not appear in this plot. BaselineOpt fails to decompose the tensors with rank larger than 6. On the contrary, every version of S-HOT successfully decompose the tensors with larger ranks. Among them, S-HOT_{memo} is up to $7\times$ faster than S-HOT_{scan} and S-HOT_{space}. S-HOT_{scan} is faster than S-HOT_{space} but the difference between them decreases as the rank increases. This is because, as the rank increases, the outer products (i.e., lines 19 and 21 of Algorithm 2) become the major bottleneck, which are common in S-HOT_{space} and S-HOT_{scan}.

Nonzero entries: Lastly, we investigate the scalability of the considered methods with respect to the number of non-zero entries. We increase the number of non-zero entries in the input tensor from 10^4 to 10^7 . As shown in Fig. 1(d), **every version of S-HOT scales near linearly** with respect to the number of non-zero entries. This is because the S-HOT family scans most non-zero entries (especially, S-HOT_{space} and S-HOT_{scan} scan all the non-zero entries), and processing each non-zero entry takes almost the same time. Among them, S-HOT_{memo} is $4\times$ faster than S-HOT_{scan} and S-HOT_{space}. With respect to the number of non-zero entries, BaselineOpt shows better scalability than S-HOT since it *explicitly materializes* \mathcal{Y} . Once \mathcal{Y} is materialized, since its size does not depend on the number of non-zero entries, the remaining tasks of BaselineOpt are not affected by the number of non-zero entries.

Table 5. Sample clusters of venues in the Microsoft Academic Graph dataset.

<u>CS-related</u>	International Conference on Networking(ICN), Wired/Wireless Internet Communications(WWIC), Database and Expert Systems Applications(DEXA), Data Mining and Knowledge Discovery, IEEE Transactions on Robotics, ...
<u>Nanotech.</u>	Nature Nanotechnology, PLOS ONE, Journal of Experimental Nanoscience, Journal of Nanoscience and Nanotechnology, Journal of Semiconductors, Trends in Biotechnology, ...
<u>Clinical</u>	European Journal of Cancer, PLOS Biology, Clinical and Applied Thrombosis-Hemostasis, Journal of Infection Prevention, RBMC Clinical Pharmacology, Regional Anesthesia and Pain ...

5.3. Q2: S-HOT at work

We test the scalability of S-HOT on the MS Academic Graphdataset. We note that, since this tensor is high-order and large, both baseline algorithms fail to handle it running out of memory. However, every version of S-HOT successfully decomposes the tensor.

To better interpret the result of Tucker decomposition, we runs k-means clustering (Arthur and Vassilvitskii, 2007) where we treat each factor matrix as the low-rank embedding of the entities in the corresponding mode, as suggested in (Kolda and Sun, 2008). Specifically, for Tucker decomposition, we set the rank of each mode to 8 and run 30 iterations. For k-means clustering, we set the number of clusters to 400 and run 100 iterations.

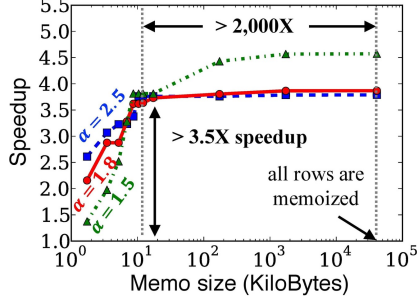
Table 5 shows sample clusters in the venue mode. The first cluster contains many venues related to Computer Science. The second cluster contains many nano-technology-related venues such as Nature Nanotechnology, Journal of Experimental Nanoscience. The third one have many venues related to Medical Science and Diseases. This result indicates that Tucker decomposition discovers meaningful concepts and groups entities related to each other. However, there is a vast array of methods for multi-aspect data analysis, and we leave a comparative study as to which one performs the best for future work.

5.4. Q3: Effect of the memory budget on the speed of S-HOT_{memo}

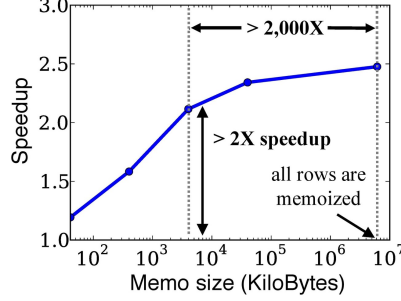
We measure the effect of memory budget B for memoization on the speed of S-HOT_{memo} using synthetic and real-world tensors. We use three 4-order tensors with dimensionality 20,000 for each mode. All the tensors have 10^6 non-zero entries, while they have different degree distributions characterized by the skewness α of the Zipf distribution. We also use the real-world datasets listed in Table 4.

Fig. 5(a) shows the result with the synthetic tensors where we set the rank of each mode to 6. S-HOT_{memo} tends to be faster as we use more memory for memoization. However, the speed-up slows down because we prioritize rows to be memoized by the degree of the corresponding mode indices, as described in Section 4.4. As the memory budget increases, S-HOT_{memo} memoizes rows corresponding to mode indices with smaller degree, which saves less computation. Notice that, with only the 10KB memory budget, S-HOT_{memo} becomes over $3.5\times$ faster than S-HOT_{scan}, which does not use memoization.

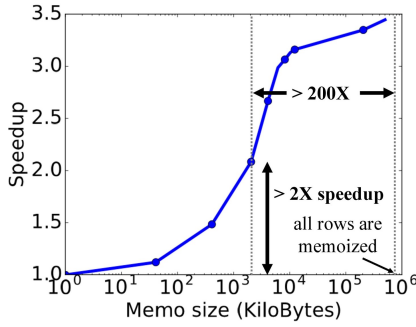
As shown in Figs. 5(b)-5(d), we obtain the same trend with the real-world

Speedup under different memoization budgets:

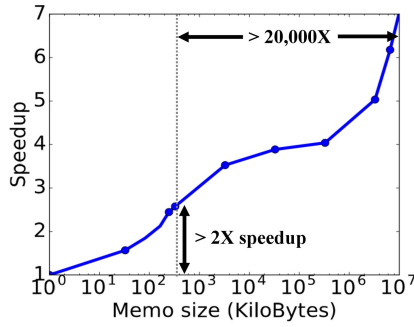
(a) Synthetic



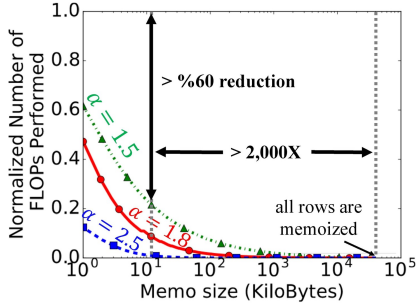
(b) MS Academic Graph



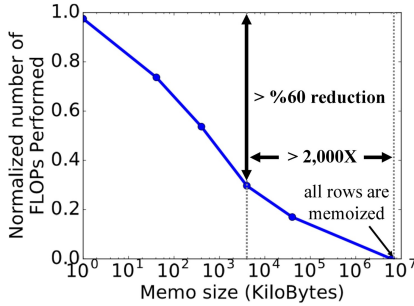
(c) Enron



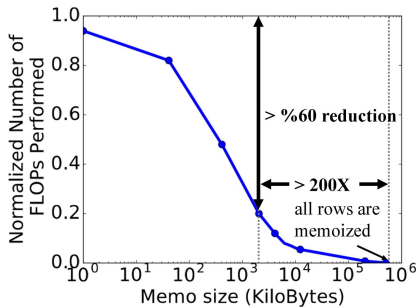
(d) Speed-up in the LBNL tensor

FLOPS under different memoization budgets:

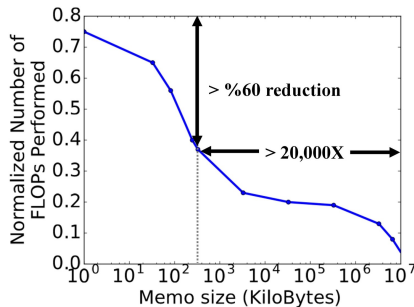
(e) Synthetic



(f) MS Academic Graph



(g) Enron



(h) LBNL

Fig. 5. **S-HOT_{memo}** significantly reduces computational cost on both synthetic and real-world tensors. In the synthetic tensors, S-HOT_{memo} achieves over $3.5\times$ speed-up by memoizing less than 0.05% of rows with a 10KB memory budget. In the MS Academic Graph dataset, S-HOT_{memo} achieves over $2\times$ speed-up by memoizing less than 0.05% of the rows with a 5MB memory budget. In (e)-(h), the y-axis represents the number of floating-point operations (FLOPs) for recomputing ‘unmemoized’ rows of $\mathbf{Y}_{(n)}$.

tensors. Notice that, in the MS Academic Graphdataset, $\text{S-HOT}_{\text{memo}}$ becomes over $2\times$ faster than $\text{S-HOT}_{\text{scan}}$, which does not use memoization, with only a 4MB memory budget. $\text{S-HOT}_{\text{scan}}$ saves much computation by memoizing a small number of rows due to the power-law degree distributions, shown in Fig. 4.

Additionally, Figs. 5(e)-5(h) show how the normalized number of floating-point operations (FLOPs) required for recomputing the ‘unmemoized’ rows of $\mathbf{Y}_{(n)}$ changes depending on the memory budget B for memoization. The computation costs are significantly reduced even when a small fraction of rows are memoized, due to our prioritization scheme.

5.5. Q5: Effect of the skewness of data on the speed of $\text{S-HOT}_{\text{memo}}$

We measure the effect of the skewness of degree distribution on the speed of $\text{S-HOT}_{\text{memo}}$. To this end, we use three 4-order tensors with different degree distributions characterized by the skewness α of the Zipf distribution. All of them have 10^6 non-zero entries, and their dimensionality for each mode is 20,000.

Fig. 5(a) shows how rapidly the speed-up of $\text{S-HOT}_{\text{memo}}$ increases depending on the skewness α . In tensors with larger α , the speed-up of $\text{S-HOT}_{\text{memo}}$ tends to increase faster. For example, with a 2KB memory budget, $\text{S-HOT}_{\text{memo}}$ achieves over $2.5\times$ speed-up in the tensor with $\alpha = 2.5$, while it achieves less than $1.5\times$ speed up in the tensor with $\alpha = 1.5$. This is because, with larger α , more non-zero entries are concentrated in few mode indices.

For every realistic degree distribution with $\alpha > 1$, $\text{S-HOT}_{\text{memo}}$ achieves over $3.5\times$ speed-up with a 10KB memory budget. $\text{S-HOT}_{\text{memo}}$ reverts to $\text{S-HOT}_{\text{scan}}$ if the input tensor has an unrealistic uniform degree distribution with $\alpha = 0$.

6. Conclusions

In this paper, we propose S-HOT, a scalable algorithm for high-order Tucker decomposition. S-HOT solves *M-bottleneck*, which existing algorithms suffer from, by using an *on-the-fly* computation. We provide three versions of S-HOT: $\text{S-HOT}_{\text{space}}$, $\text{S-HOT}_{\text{scan}}$, and $\text{S-HOT}_{\text{memo}}$, which provide an interesting trade-off between time and space. We theoretically and empirically show that all versions of S-HOT have better scalability than baselines.

In summary, our contributions are as follows.

- **Bottleneck Resolution:** We identify *M-Bottleneck* (Fig. 2), the scalability bottleneck of existing Tucker decomposition algorithms and avoid it by a novel approach based on an *on-the-fly* computation.
- **Scalable Algorithm Design:** We propose S-HOT, a Tucker decomposition algorithm that is carefully optimized for large-scale high-order tensors. S-HOT successfully decomposes $1000\times$ larger tensors than baselines algorithms (Fig. 1) with identical convergence properties (Observation 1).
- **Theoretical Analyses:** We prove the amount of memory space and the number of data scans that the different versions of S-HOT require (Table 2 and Lemmas 3-9).

Future work includes reducing redundant computations that occur during TTMcs,

which is a subroutine of S-HOT, using advanced data structures (e.g., compressed sparse fiber (Smith and Karypis, 2015)), as suggested in (Smith and Karypis, 2017). For reproducibility and extensibility of our work, we make the source code of S-HOT publicly available at <http://dm.postech.ac.kr/shot>.

Acknowledgements. This research was supported by Disaster-Safety Platform Technology Development Program of the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (Grant Number: 2019M3D7A1094364) and the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MSIP) (No. 2016R1E1A1A01942642).

A. Examples for Basic Tensor Terminologies and Operations

We provide examples for basic tensor terminologies and operations defined in Section 2.2.

Fiber: In a 3-order tensor \mathcal{X} , there are 3 kinds of fibers, $\mathcal{X}(:, j, k)$ (mode-1), $\mathcal{X}(i, :, k)$ (mode-2), and $\mathcal{X}(i, j, :)$ (mode-3) depending on fixed indices.

Slice: In a 3-order tensor \mathcal{X} , there are 3 kinds of slices, $\mathcal{X}(i, :, :)$, $\mathcal{X}(:, j, :)$, and $\mathcal{X}(:, :, k)$ depending on fixed indices.

Tensor Unfolding/Matricization: The mode-1 unfolding of a 3-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is denoted by $\mathbf{X}_{(1)} \in \mathbb{R}^{I_1 \times (I_2 I_3)}$. Note that, there are multiple ways to unfold a tensor in terms of the order that the entries of each slice are stacked. For example, the followings are two different ways of mode-1 unfolding

$$\mathbf{X}_{(1)}(i, j + (k - 1)I_2) = \mathcal{X}(i, j, k) \text{ and } \mathbf{X}_{(1)}(i, k + (j - 1)I_3) = \mathcal{X}(i, j, k).$$

However, specific orders do not have an impact on our algorithm as long as an order is used consistently.

N-order Outer Product: The 3-order outer product of vectors $\mathbf{a} \in \mathbb{R}^I$, $\mathbf{b} \in \mathbb{R}^J$, and $\mathbf{c} \in \mathbb{R}^K$ is a 3-order tensor of $\mathbb{R}^{I \times J \times K}$ where each (i, j, k) -th entry is

$$[\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}](i, j, k) = \mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k).$$

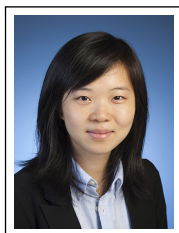
References

- E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener. Modeling and multiway analysis of chatroom tensors. In *ISI*, 2005.
- L. A. Adamic and B. A. Huberman. Zipf’s law and the Internet. *Glottometrics*, 3(1):143–150, 2002.
- D. Arthur, S. Vassilvitskii. k-means++: The advantages of careful seeding. *SODA*, 2007
- W. Austin, G. Ballard, and T. G. Kolda. Parallel tensor compression for large-scale scientific data. In *IPDPS*, 2016.
- B. W. Bader, and T. G. Kolda. Matlab tensor toolbox version 2.6. <http://www.sandia.gov/~tgkolda/TensorToolbox/>
- M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and Scalable Computations with Sparse Tensors *HPEC*, 2012

- M. W. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputing Applications* 6(1):13–49, 1992.
- A. Beutel, A. Kumar, E. E. Papalexakis, P. P. Talukdar, C. Faloutsos, and E. P. Xing. Flexifact: Scalable flexible factorization of coupled tensors on hadoop. In *SDM*, 2014.
- Y. Cai, M. Zhang, D. Luo, C. Ding, and S. Chakravarthy. Low-order tensor decompositions for social tagging recommendation. In *WSDM*, 2011.
- Y. Chi, B. L. Tseng, and J. Tatemura. Eigen-trend: Trend analysis in the blogosphere based on singular value decompositions. In *CIKM*, 2006.
- D. Choi, J. G. Jang, and U Kang. Fast, Accurate, and Scalable Method for Sparse Coupled Matrix-Tensor Factorization. *arXiv preprint arXiv:1708.08640*, 2017.
- J. Choi, X. Liu, V. Chakaravarthy. High-performance dense tucker decomposition on GPU clusters. *SC*, 2018.
- J. H. Choi and S. Vishwanathan. Dfacto: Distributed factorization of tensors. In *NIPS*, 2014.
- A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.
- J. E. Cohen, R. C. Farias, and P. Comon. Fast decomposition of large nonnegative tensors. *IEEE Signal Processing Letters*, 22(7):862–866, 2015.
- A. L. de Almeida and A. Y. Kibangou. Distributed computation of tensor decompositions in collaborative networks. In *CAMSAP*, pages 232–235, 2013.
- A. L. de Almeida and A. Y. Kibangou. Distributed large-scale tensor decomposition. In *ICASSP*, 2014.
- T. Franz, A. Schultz, S. Sizov, and S. Staab. Triplerank: Ranking semantic web data by tensor decomposition. In *ISWC*, 2009.
- L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.
- B. Jeon, I. Jeon, L. Sael, and U. Kang. Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries. In *ICDE*, pages 811–822, 2016.
- I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. Haten2: Billion-scale tensor decompositions. In *ICDE*, 2015.
- U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *KDD*, 2012.
- O. Kaya, and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. *SC*, 2015.
- O. Kaya, and B. Uçar. High performance parallel algorithms for the tucker decomposition of sparse tensors. *ICCP*, 2016.
- T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.
- T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, 2008.
- T. G. Kolda, B. W. Bader, and J. P. Kenny. Higher-order web link analysis using multilinear algebra. In *ICDM*, 2005.
- H. Lamba, V. Nagarajan, K. Shin, and N. Shajarisales. Incorporating side information in tensor completion. In *WWW Companion*, 2016.
- L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-(R_1, R_2, \dots, R_n) approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications* 21(4):1324–1342, 2000.
- L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications* 21(4):1253–1278, 2000.
- R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. Available from netlib@ornl.gov, 1997.
- J. Li, J. Choi, I. Perros, J. Sun and R. Vuduc. Model-driven sparse CP decomposition for higher-order tensors. In *IPDPS*, 2017.
- J. Li, J. Sun, and R. Vuduc. HiCOO: hierarchical storage of sparse tensors. In *SC*, 2018.
- B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *CLUSTER*, 2017.
- K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, 2011.
- S. Moghaddam, M. Jamali, and M. Ester. Etf: extended tensor factorization model for personalizing prediction of review helpfulness. In *WSDM*, 2012.
- J. Oh, K. Shin, E. E. Papalexakis, C. Faloutsos, and H. Yu. S-hot: Scalable high-order tucker decomposition. In *WSDM*, pages 761–770, 2017.

- S. Oh, N. Park, L. Sael and U. Kang. Scalable Tucker Factorization for Sparse Tensors - Algorithms and Discoveries In *ICDE*, pages 1120–1131, 2018.
- E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Parcube: Sparse parallelizable candecomp-parafac tensor decomposition. *TKDD*, 10(1):3, 2015.
- E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *TIST*, 8(2):16, 2016.
- I. Perros, R. Chen, R. Vuduc, and J. Sun. Sparse hierarchical tucker factorization and its application to healthcare. In *ICDM*, 2015.
- R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *ACM SIGCOMM Computer Communication Review*, 36(1):29–38, 2014.
- D. M. Powers. Applications and explanations of Zipf’s law. In *NeMLaP/CoNLL*, 1998.
- S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *WSDM*, 2010.
- Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011.
- L. Sael, I. Jeon, and U. Kang. Scalable tensor mining. *Big Data Research*, 2(2):82–86, 2015.
- S. Smith and G. Karypis. Tensor-matrix products with a compressed sparse tensor. *IA3*, 2015.
- S. Smith and G. Karypis. Accelerating the tucker decomposition with compressed sparse tensors. *Euro-Par*, 2017.
- S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools. <http://frostdt.io/>.
- K. Shin, B. Hooi, and C. Faloutsos. Fast, Accurate, and Flexible Algorithms for Dense Sub-tensor Mining. *TKDD*, 12(3):28:1–28:30, 2018.
- K. Shin, B. Hooi, J. Kim, and C. Faloutsos. DenseAlert: Incremental Dense-Subtensor Detection in Tensor Streams. In *KDD*, 2017.
- K. Shin, S. Lee, and U. Kang. Fully scalable methods for distributed tensor factorization. *TKDE*, 29(1):100–113, 2017.
- N. D. Sidiropoulos and A. Kyriallidis. Multi-way compressed sensing for sparse low-rank tensors. *IEEE Signal Processing Letters*, 19(11):757–760, 2012.
- A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. P. Hsu, and K. Wang. An overview of microsoft academic service (mas) and applications. In *WWW Companion*, 2015.
- A. Smilde, R. Bro, and P. Geladi. *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.
- J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: Dynamic tensor analysis. In *KDD*, 2006.
- J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: A novel approach to personalized web search. In *WWW*, 2005.
- C. E. Tsourakakis. Mach: Fast randomized tensor decompositions. In *SDM*, pages 689–700. SIAM, 2010.
- L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.
- N. Vervliet, O. Debals, L. Sorber, and L. De Lathauwer. Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis. *IEEE Signal Processing Magazine*, 31(5):71–79, 2014.
- J. Shetty and J. Adibi. The Enron email dataset database schema and brief statistical report, *Information sciences institute technical report, University of Southern California*.

Author Biographies



Jiyuan Zhang is a Ph.D. student in Electrical and Computer Engineering Department of Carnegie Mellon University. Her research interests lie in high performance computing with applications in machine learning and data mining.



Jinoh Oh is a computer scientist at Adobe Systems Inc. He received his Ph.D. in Computer Science and Engineering from Pohang University of Science and Technology. His research interests include recommendation and scalable machine learning.



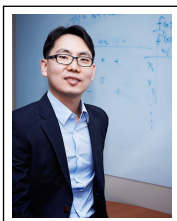
Kijung Shin is an Assistant Professor of the School of Electrical Engineering at KAIST. He received his Ph.D. in Computer Science from Carnegie Mellon University in 2019. He received his B.S. in Computer Science and Engineering from Seoul National University in 2015. His research interests include graph mining and scalable machine learning.



Evangelos E. Papalexakis is an Assistant Professor of the CSE Dept. at University of California Riverside. He received his PhD degree at the School of Computer Science at Carnegie Mellon University (CMU). Prior to CMU, he obtained his Diploma and MSc in Electronic & Computer Engineering at the Technical University of Crete, in Greece. Broadly, his research interests span the fields of Data Mining, Machine Learning, and Signal Processing. His research involves designing scalable algorithms for mining large multi-aspect datasets, with specific emphasis on tensor factorization models, and applying those algorithms to a variety of real world multi-aspect data problems. His work has appeared in KDD, ICDM, SDM, ECML-PKDD, WWW, PAKDD, ICDE, ICASSP, IEEE Transactions of Signal Processing, and ACM TKDD.



Christos Faloutsos is a Professor at Carnegie Mellon University. He has received the Presidential Young Investigator Award by the National Science Foundation (1989), the Research Contributions Award in ICDM 2006, the SIGKDD Innovations Award (2010), 24 “best paper” awards (including 5 “test of time” awards), and four teaching awards. Eight of his advisees have attracted KDD or SCS dissertation awards. He is an ACM Fellow, he has served as a member of the executive committee of SIGKDD; he has published over 350 refereed articles, 17 book chapters and two monographs. He holds seven patents (and 2 pending), and he has given over 40 tutorials and over 20 invited distinguished lectures. His research interests include large-scale data mining with emphasis on graphs and time sequences; anomaly detection, tensors, and fractals.



Hwanjo Yu is a Professor in the Department of Computer Science and Engineering at POSTECH, South Korea since 2008. He received his PhD under the supervision of Prof. Jiawei Han at the University of Illinois at Urbana-Champaign at 2004, and worked as an assistant professor at the University of Iowa until 2007. He has a broad background in data mining and machine learning and published over 100 papers in top-tier conferences and journals. His recent research interests include scalable machine learning and recommender system.