# S-HOT: Scalable High-Order Tucker Decomposition

Jinoh Oh[§*], Kijung Shin[‡], Evangelos E. Papalexakis[¶], Christos Faloutsos[‡], Hwanjo Yu[§†]
[§]Dept. of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea
[‡]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[¶]Dept. of Computer Science and Engineering, University of California Riverside, CA, USA
{kurin, hwanjoyu}@postech.ac.kr, {kijungs, christos}@cs.cmu.edu, epapalex@cs.ucr.edu

## ABSTRACT

Multi-aspect data appear frequently in many web-related applications. For example, product reviews are quadruplets of (user, product, keyword, timestamp). How can we analyze such web-scale multi-aspect data? Can we analyze them on an off-the-shelf workstation with limited amount of memory?

Tucker decomposition has been widely used for discovering patterns in relationships among entities in multi-aspect data, naturally expressed as high-order tensors. However, existing algorithms for Tucker decomposition have limited scalability, and especially, fail to decompose high-order tensors since they *explicitly materialize* intermediate data, whose size rapidly grows as the order increases ($\geq 4$). We call this problem *M-Bottleneck* ("Materialization Bottleneck").

To avoid *M-Bottleneck*, we propose S-HOT, a scalable high-order tucker decomposition method that employs the *on-the-fly-computation* to minimize the materialized intermediate data. Moreover, S-HOT is designed for handling disk-resident tensors, too large to fit in memory, without loading them all in memory at once. We provide theoretical analysis on the amount of memory space and the number of scans of data required by S-HOT. In our experiments, S-HOT showed better scalability not only with the order but also with the dimensionality and the rank than baseline methods. In particular, S-HOT decomposed tensors **1000× larger** than baseline methods in terms dimensionality. S-HOT also successfully analyzed real-world tensors that are both large-scale and high-order on an off-the-shelf workstation with limited amount of memory, while baseline methods failed. The source code of S-HOT is publicly available at http://dm.postech.ac.kr/shot to encourage reproducibility.

## 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 [5, 17, 21, 22, 28, 31], including the following examples:

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

To analyze such multi-aspect data, several tensor decomposition methods have been proposed, and we refer interested readers to the excellent survey [16]; tensor decompositions have provided meaningful results in various domains [1, 16, 17, 19] as well as web [5, 11, 17, 22, 28, 32]. Especially, Tucker decomposition [40] has been successfully applied in many applications, such as web search [38], network forensics [37], social network analysis [6], and scientific data compression [2].

Developing a scalable Tucker decomposition method 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 method, 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 [16] for details). Previous studies [18, 14] pointed out that a huge amount of intermediate data are generated during the first step, and they proposed methods for reducing the intermediate data by carefully ordering computation.

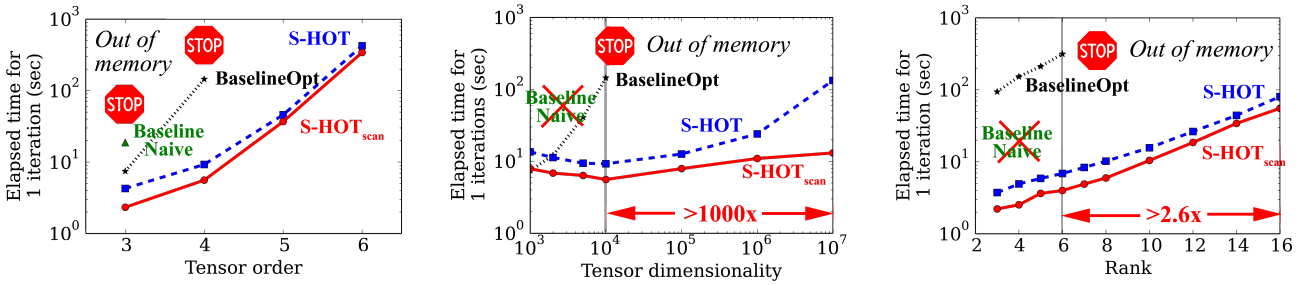
However, existing methods still have a scalability limitation, and easily run out of memory, particularly when dealing with very *high-order* tensors. Specifically, existing methods *explicitly materialize* $\mathcal{Y}$, but the amount of space required for storing $\mathcal{Y}$ grows rapidly with respect to the order, the dimensionality, and the rank of the input tensor. For example, the space required for $\mathcal{Y}$ is over *260 GBytes* for a 5-order tensor with 1 million dimensionality when the rank of Tucker decomposition is set to 16. We call this problem *M-Bottleneck*, which stands for Materialization Bottleneck. Due to *M-Bottleneck*, existing methods are not suitable for decomposing a tensor with high order, dimensionality, and/or rank. Our experimental results show that even state-of-the-art methods easily run out of memory as these factors increase (Figure 1).

To avoid *M-Bottleneck*, in this work, we propose S-HOT, a scalable Tucker decomposition method. S-HOT is designed

*This work is done while he is visiting CMU as a post-doc
†Corresponding author



*WSDM 2017, February 06-10, 2017, Cambridge, United Kingdom*
 ISBN 978-1-4503-4675-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018661.3018721




(a) S-HOT shows better order-scalability. (b) S-HOT decomposes $10^3\times$ higher dimension tensor (c) S-HOT shows better rank-scalability.

**Figure 1: S-HOT scales up. S-HOT successfully decomposes tensors with high order, dimensionality, and rank, while the baseline methods fail running out of memory as those three factors increase. Especially, S-HOT handles a tensor with 1000 times higher dimensionality. We use two baselines: 1) BaselineNaive: naive method for Tucker decomposition, and 2) BaselineOpt [18]: the state-of-the-art memory-efficient method for Tucker decomposition.**

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. Specifically, we utilize the *reverse communication interface* of a recent scalable eigensolver called Implicitly Restart Arnoldi Method (IRAM), which enables SVD computation without materializing $\mathcal{Y}$. Based on this *on-the-fly computation* idea, S-HOT performs Tucker decomposition by streaming nonzero tensor entries from the disk. Moreover, we offer two versions of S-HOT with distinct advantages: 1) S-HOT$_{\text{scan}}$: requiring multiple copies of the input tensor, but faster with half the number of data scans, and 2) S-HOT: slower but more space efficient requiring only one copy of the input tensor.

Our experimental results demonstrate that S-HOT outperforms baseline methods by providing significantly better scalability, as shown in Figure 1. Specifically, both S-HOT$_{\text{scan}}$ and S-HOT successfully decompose a 6-order tensor, while baselines fail to decompose even a 4-order tensor or a 5-order tensor due to their high memory requirements. The difference is more significant in terms of dimensionality. S-HOT decomposes a ***1000× larger tensor*** than baselines (see Figure 1(b)).

Our contributions are summarized as follows.

- **Handling Bottleneck:** We identify *M-Bottleneck*, which limits the scalability of existing Tucker decomposition methods, and we avoid it by an *on-the-fly computation*.
- **Algorithm Design:** We propose S-HOT, a scalable Tucker decomposition method carefully designed for high-order tensors too large to fit in memory. In our experiments, S-HOT offers up to *1000× better scalability* than baseline methods.
- **Theoretical analysis:** We provide a theoretical analysis on the amount of memory space and the number of scans of data that our methods require.

**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 the preliminaries on tensors and Tucker decomposition. In Section 3, we review related work, and introduce *M-Bottleneck*, which past methods commonly suffer. In Section 4, we propose S-HOT, a scalable high-order tucker decomposition, for addressing *M-Bottleneck*. After presenting experimental results in Section 5, we make a conclusion 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)

**Table 1: Table of Symbols**

| Symbol | Definition |
|---|---|
| $N$ | number of modes |
| $\mathcal{X}$ | $N$-order input tensor $\in \mathbb{R}^{I_1\times\cdots\times I_N}$ |
| $\mathcal{X}(i_1,...,i_N)$ | $(i_1,...,i_N)$-th entry of $\mathcal{X}$ (also denoted by $\mathcal{X}_{i_1 i_2 \ldots i_N}$) |
| $\Theta(\mathcal{X})$ | set of the indices of all non-zero entries in $\mathcal{X}$ |
| $\Theta^{(n)}_i(\mathcal{X})$ | subset of $\Theta(\mathcal{X})$ where the $n$-th mode index is $i$ |
| $\mathbf{X}_{(n)}$ | 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 $\in \mathbb{R}^{J_1\times\cdots\times J_N}$ |
| $\{\mathbf{A}\}$ | 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 $\mathcal{X}$ |
| $\bar{\mathbf{a}}^{(n)}_i$ | $i$-th row-vector of $\mathbf{A}^{(n)}$ |
| $\mathbf{a}^{(n)}_j$ | $j$-th column-vector of $\mathbf{A}^{(n)}$ |
| $\circ$ | outer product |
| $\bar{\times}_n$ | $n$-mode vector product |
| $\times_n$ | $n$-mode matrix product |

to higher orders. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be the input tensor, whose order is denoted by $N$. Like the rows and columns in a matrix, $\mathcal{X}$ has $N$ modes, whose lengths, also called dimensionality, are denoted by $I_1, \ldots, I_N \in \mathbb{N}$, respectively. We denote general $N$-order tensors by boldface Euler script letters e.g., by $\mathcal{X}$, 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, $\mathcal{X}(i_1, ..., i_N)$ indicates the $(i_1, ..., i_N)$th entry of $\mathcal{X}$. The similar notations is also 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 $\mathrm{a}_i$ in short).

**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.*

For example, in a three-order tensor $\mathcal{X}$, there are three 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.

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

For example, in a three-order tensor $\mathcal{X}$, there are three kinds of slices, $\mathcal{X}(i, :, :)$, $\mathcal{X}(:, j, :)$, and $\mathcal{X}(:, :, k)$. Table 1 lists the symbols frequently used in this paper.

## 2.2 Basic Tensor Operations

We review basic tensor operations, which are the building blocks of tucker decomposition, explained in the following section.

**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 $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is a matrix $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{q \neq n} I_q)}$ whose columns are the mode-n fibers.*

For example, the mode-1 unfolding of a three-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

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

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

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

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

**Definition 5** ($n$-mode Vector Product). *The n-mode vector product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with 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 $\mathbb{R}^{I_1 \times \ldots I_{n-1} \times I_{n+1} \times \cdots \times I_N}$. Elementwise, we have*

$$[\mathcal{X} \bar{\times}_n \mathbf{v}](i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N) = \sum_{i_n=1}^{I_n} \mathrm{x}_{i_1 i_2 \ldots i_N} \mathrm{v}_{i_n}.$$

**Definition 6** ($n$-mode Matrix Product). *The n-mode matrix product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with 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 $\mathbb{R}^{I_1 \times \ldots I_{n-1} \times J_n \times I_{n+1} \times \cdots \times I_N}$. Elementwise, we have*

$$[\mathcal{X} \times_n \mathbf{U}](i_1, \ldots, i_{n-1}, j_n, i_{n+1}, \ldots, i_N) = \sum_{i_n=1}^{I_n} \mathrm{x}_{i_1 i_2 \ldots i_N} \mathrm{u}_{j_n i_n}.$$

We adopt the shorthand notations in [18] for all-mode matrix product and matrix product in every mode but one:

$$\mathcal{X} \times \{\mathbf{U}\} \equiv \mathcal{X} \times_1 \mathbf{U}^{(1)} \ldots \times_N \mathbf{U}^{(N)}, \text{ and}$$

$$\mathcal{X} \times_{-n} \{\mathbf{U}\} \equiv \mathcal{X} \times_1 \mathbf{U}^{(1)} \ldots \times_{n-1} \mathbf{U}^{(n-1)} \times_{n+1} \mathbf{U}^{(n+1)} \ldots \times_N \mathbf{U}^{(N)}.$$

Likewise, for brevity, we also use the following shorthand notations for outer product:

$$\circ_{(i_1, \ldots, i_N)} \{\mathbf{A}\} = \bar{\mathbf{a}}^{(1)}_{i_1} \circ \cdots \circ \bar{\mathbf{a}}^{(N)}_{i_N}, \text{ and}$$

$$\circ^{-n}_{(i_1, \ldots, i_N)} \{\mathbf{A}\} = \bar{\mathbf{a}}^{(1)}_{i_1} \circ \cdots \circ \bar{\mathbf{a}}^{(n-1)}_{i_{n-1}} \circ [1] \circ \bar{\mathbf{a}}^{(n+1)}_{i_{n+1}} \circ \cdots \circ \bar{\mathbf{a}}^{(N)}_{i_N}.$$

## 2.3 Tucker decomposition

Tucker decomposition [40], which is also called $N$-mode PCA, decomposes a tensor into a core tensor and $N$ factor matrices so that the original tensor is approximated best. Specifically, $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is approximated by

$$\mathcal{X} \approx \mathcal{G} \times \{\mathbf{A}\},$$

where $\mathcal{G} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$, $J_n$ denotes the rank of $n$-th mode, and $\{\mathbf{A}\}$ is the set of factor matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, each of which is in $\mathbb{R}^{I_n \times J_n}$. Using $n$-mode matrix product and $N$-order outer product, Tucker is also presented as follows:

$$\begin{aligned} \mathcal{X} &\approx \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} \\ &= \sum_{j_1=1}^{J_1} \sum_{j_2=1}^{J_2} \cdots \sum_{j_N=1}^{J_N} g_{j_1 j_2 \ldots j_N} \mathbf{a}^{(1)}_{j_1} \circ \mathbf{a}^{(2)}_{j_2} \cdots \circ \mathbf{a}^{(N)}_{j_N}. \end{aligned} \quad (1)$$

Solving Tucker is to find the $\mathcal{G}$ and $\{\mathbf{A}\}$ that approximate $\mathcal{X}$ best. It is worth noting that the solution of Tucker is not unique. The most widely used way to solve Tucker is Tucker-ALS (or Higher Order Orthogonal Iteration (HOOI)), which assumes that all column vectors in $\mathbf{A}^{(n)}$ are orthonormal and solves Tucker by Alternating Least Squares (ALS). In addition, [18] found that $\mathcal{G}$ can be uniquely computed by $\mathcal{X} \times \{\mathbf{A}^T\}$ once $\{\mathbf{A}\}$ is determined, and simplified the objective function as follows (see [18] for details):

$$\max_{\{\mathbf{A}\}} ||\mathcal{X} \times \{\mathbf{A}^T\}||. \quad (2)$$

The details of the conventional Tucker-ALS is presented in Algorithm 1.

## 2.4 Implicitly Restarted Arnoldi Method (IRAM)

Computing the Eigendecomposition for large-scale datasets is important because it is an important foundation of various dimensionality reduction and low-rank approximation techniques. Vector iteration (or power method) is one of the fundamental algorithms for solving large-scale Eigenproblem [29]. Briefly speaking, for a given matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$, Vector iteration finds the leading eigenvector corresponding

**Algorithm 1:** Tucker-ALS

**Input :** $\mathcal{X}$, a $N$-order tensor of $\mathbb{R}^{I_1 \times \cdots \times I_N}$.
$J_1, \ldots, J_N$, the rank size of each mode.
$T$, the number of iterations.

**Output:** $\{\mathbf{A}\}$, a set of factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\}$ where $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$.
$\mathcal{G}$, a $N$-order core tensor of $\mathbb{R}^{J_1 \times \cdots \times J_N}$.

```
Initialize all A^(n)
for t ← 1..T do
    for n ← 1..N do
        Y_(n) ← [X ×_{-n} {A^T}]_(n)
        A^(n) ← top-J_n left singular vectors of Y_(n)
G ← Y_(N) ×_N A^(N)^T
return G, {A}
```

to the largest eigenvalue by repeating the following updating rule from a randomly initialized $\mathbf{v}^{(0)} \in \mathbb{R}^m$.

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

It is known that, as $k$ increases, $\mathbf{v}^{(k+1)}$ converges to the leading eigenvector [29].

Arnoldi iteration is one of subspace iteration methods that extend Vector iteration to find $k$ leading eigenvectors simultaneously. Specifically, it finds the $k$ eigenvectors from a subspace called Krylov space, which is spanned by $\{\mathbf{v}, \mathbf{U}\mathbf{v}, \ldots, \mathbf{U}^j\mathbf{v}\}$, where $j \geq k-1$. Implicitly Restarted Arnoldi Method (IRAM) is one of the most advanced techniques for Arnoldi [29]. Briefly speaking, IRAM only keeps $k$ orthonormal vectors which are a basis of the Krylov space, and updates the basis until it converges, then computes the $k$ leading eigenvectors from the basis. One virtue of IRAM is *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 returns them.

For details of IRAM and *reverse communication interface*, we refer interested readers to [20, 29].

# 3. RELATED WORKS

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 past methods still commonly suffer *M-Bottleneck*, which is explained in Section 3.3. Lastly, we briefly introduce scalable methods for other tensor decomposition methods in Section 3.4.

**Table 2: S-HOT is space efficient. It produces several orders of magnitude less intermediate data than state-of-the-art methods. For simplicity, we assume $N = 5$, $I_n = I = 1$ million, $J_n = J = 10$ for all $n$, $M$ =1 billion.**

| Method | Space Requirements for Intermediate Data | |
|---|---|---|
| | (in Theory) | (in Example) |
| BaselineNaive | $MJ^{N-1}$ | $\sim$ 40TB |
| BaselineOpt [18] | $IJ^{N-1}$ | $\sim$ 40GB |
| HATEN2 [14] | $\max(IJ^{N-1}, M(N-1)J)$ | $\sim$ 160GB |
| **S-HOT** | $\max(I, J^{N-1})$ | $\sim$ 4MB |
| **S-HOT$_{\text{scan}}$** | $J^{N-1}$ | $\sim$ 40KB |

## 3.1 Intermediate Data Explosion

The most important challenge in scaling Tucker decomposition is the "intermediate data explosion" problem which was first identified in [18] (Definition 7). It states that a naive implementation of Algorithm 1, especially the computation of $[\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$, 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) [18] . *Let $M$ be the number of non-zero entries in $\mathcal{X}$. In Algorithm 1, naively computing $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ requires $O(M \prod_{p \neq n} J_p)$ space for intermediate data.*

For example, if we assume a 5-order tensor with $M$=1 billion and $J_n$=10 for all $n$, $M \prod_{p \neq n} J_p$=10 trillions, which requires 40TB, which exceeds the capacity of a typical hard disk as well as RAM.

## 3.2 Scalable Tucker decomposition

**Memory Efficient Tucker (MET) [18]:** 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 MET$^{\text{fiber}}$ is the most space-efficient one.

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

$$\mathcal{Y}_{:j_2 j_3} \leftarrow \overbrace{\mathcal{X} \bar{\times}_2 \mathbf{a}^{(2)}_{j_2} \bar{\times}_3 \mathbf{a}^{(3)}_{j_3}}^{I_1}. \quad (3)$$

The amount of intermediate data produced during the computation of a fiber in $\mathcal{Y}$ by Equation (3) is only $O(I_1)$. This amount is the same for general $N$-order tensors. MET$^{\text{fiber}}$ is one of the most space-optimized tensor decomposition methods, and we shall refer to MET$^{\text{fiber}}$ as BaselineOpt from now on.

**Hadoop Tensor Method (HaTen2) [14]:** 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 $O(M \sum_{p \neq n} J_p)$

space for intermediate data, which is larger than $O(I_n)$ space that BaselineOpt requires.

**Other Work Related to Scalable Tucker Decomposition:** Several methods were proposed for the case when the input tensor $\mathcal{X}$ is dense so that it cannot fit in memory. Specifically, [39] uses random sampling of non-zero entries to sparsify $\mathcal{X}$, and [2] distributes the entries of $\mathcal{X}$ across multiple machines. However, in this work, our method stores $\mathcal{X}$ in disk, thus, the memory requirement does not depend on the number of non-zero entries (i.e., $M$). In addition, we assume that $\mathcal{X}$ is a large but sparse tensor, which is more common in real-world application, and thus its non-zero entries fit on a disk.

Related to our work are more recent variations of the Tucker model, such as Hierarchical Tucker [12, 26] which are specifically tailored for high-order tensor analysis, however such models require additional knowledge that may be application dependent. Note that our method is a scalable algorithm for the original Tucker model with no modification. We leave exploration of such variations for future work.

## 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 methods 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, if we assume a 5-order tensor with $I_n$=1 million and $I_n$=10 for all $n$, $I_n \prod_{p \neq n} J_p$ =10 billions. Thus, if single-precision floating-point numbers are used, materializing $\mathbf{Y}_{(n)}$ in a dense matrix format requires about 40GB 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 method in Table 2. Our proposed S-HOT$_{\text{scan}}$ and S-HOT methods, which are discussed in detail in the following section, require several orders of magnitude less space for intermediate data by avoiding the materialization of $\mathbf{Y}_{(n)}$.

## 3.4 Scalable Algorithms for Other Tensor Decomposition Methods

Comprehensive surveys on scalable algorithms for various tensor decomposition methods can be found in [25, 30]. Among other methods 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 in many applications including chemometrics [36] and signal processing [34]. The memory explosion problem in PARAFAC decomposition was studied in [15], and recently, many scalable PARAFAC decomposition methods have been proposed to reduce memory requirements and/or computation. These approaches can be grouped into the following categories:

- Optimized standard approaches: Standard optimization algorithms including ALS and its variants [14, 15, 7, 13, 27], Gradient Descent [7], Stochastic Gradient Descent [4], Coordinate Descent and its variants [32, 33], are optimized for PARAFAC decomposition to reduce intermediate data and computation.
- Sampling or Subdivision: Smaller tensors are obtained by sampling [23, 24] or subdivision [9, 10], and each subtensor is factorized. Then, the factor matrices of the entire tensor are reconstructed from those of subtensors. In [41], a partially observable tensor, obtained by randomly sampling entries in the input tensor, is factorized instead of the original tensor.
- Compression: In [34, 8], tensor is compressed before being factorized.

Above approaches focused only on PARAFAC decomposition. Although PARAFAC decomposition is simple, it has a limitation in capturing nontrilinear variation in a tensor. In contrast, Tucker decomposition successfully captures nontrilinear variation and also compresses a tensor optimally [25]. In this work, we focus on Tucker decomposition and propose a scalable algorithm for it.

# 4. PROPOSED METHOD: S-HOT

In this section, we develop a novel method called S-HOT, which avoids *M-Bottleneck* caused by the materialization of $\mathbf{Y}_{(n)}$. S-HOT enables high-order Tucker decomposition to be performed even in an off-the-shelf workstation. In Table 3, two versions of S-HOT are compared with baseline methods 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):

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

Our key idea is to tightly integrate the above two steps, and compute the singular vectors through IRAM directly from $\mathcal{X}$ without materializing the entire $\mathcal{Y}$ 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 $\mathbf{Y}_{(n)}{\mathbf{Y}_{(n)}}^T \in \mathbb{R}^{I_n \times I_n}$. Specifically, if we use *reverse communication interface* of IRAM, the above two steps are computed by simply updating $\mathbf{v}'$ repeatedly as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)}{\mathbf{Y}_{(n)}}^T \mathbf{v}, \tag{4}$$

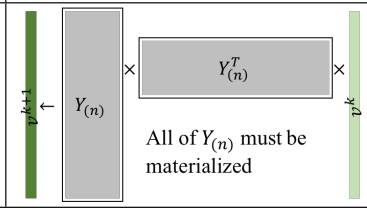
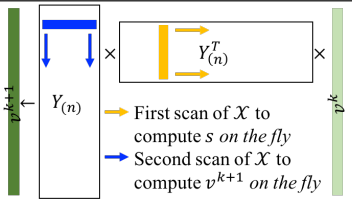
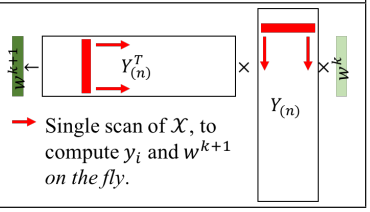
where we do not need to materialize $\mathbf{Y}_{(n)}$ (and thus we can avoid *M-Bottleneck*) if we are able to update $\mathbf{v}'$ directly from the $\mathcal{X}$. Note that, using IRAM does not change the result of the above two steps. Thus, final results of Tucker decomposition are also not changed, while space requirements are reduced drastically, as summarized in Table 3.

The remaining problem is how to update $\mathbf{v}'$ directly from $\mathcal{X}$, which is stored in disk, without materializing $\mathbf{Y}_{(n)}$. For addressing this problem, we first examine a naive method extending BaselineOpt and then eventually propose two algorithms, S-HOT and S-HOT$_{\text{scan}}$.

## 4.1 First step: "Naive S-HOT"

NAIVE S-HOT is a straight-forward extension of BaselineOpt, which computes $\mathcal{Y}$ fiber by fiber, for computing Equation (4). Thus, NAIVE S-HOT computes $\mathbf{v}'$ progressively on the basis of each column vector of $\mathbf{Y}_{(n)}$, which

**Table 3: S-HOT is nimble. Key differences in the objectives, update equations, materialized data of methods; and figures illustrating how they work. In the illustrating figures, colored regions need to be explicitly materialized in memory.**

| Method | BaselineNaive and BaselineOpt [18] | S-HOT | S-HOT$_{\text{scan}}$ |
|---|---|---|---|
| Objective | Left singular vectors of $\mathbf{Y}_{(n)}$ | | Right singular vectors of $\mathbf{Y}_{(n)}$ |
| Update equation | $\mathbf{v}^{k+1} \leftarrow \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}{}^T\mathbf{v}^k$ | $\mathbf{s} \leftarrow \sum_{p\in\Theta(\mathcal{X})} \mathrm{v}_{i_n}^k \mathcal{X}(p) \left[\circ_p^{-n}\{\mathbf{A}\}\right]_{(n)}$<br>$\mathrm{v}_{i_n}^{k+1} \leftarrow \sum_{p\in\Theta_{i_n}^{(n)}(\mathcal{X})} \mathbf{s}^T\mathcal{X}(p) \left[\circ_p^{-n}\{\mathbf{A}\}\right]_{(n)}$ | $\mathbf{y_i} \leftarrow \sum_{p\in\Theta_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \left[\circ_p^{-n}\{\mathbf{A}\}\right]_{(n)}$<br>$\mathbf{w^{k+1}} \leftarrow \sum_{i=1}^{I_n} (\mathbf{y_i}{}^T\mathbf{w}^k)\,\mathbf{y_i}$ |
| Materialization | $\mathbf{Y}_{(n)} \in \mathbb{R}^{I_n \times \prod_{p\neq n} J_p}$ | $\mathbf{s} \in \mathbb{R}^{\prod_{p\neq n} J_p}$ | $\mathbf{y_i} \in \mathbb{R}^{\prod_{p\neq n} J_p}$ |
| Illustration |  |  |  |

corresponds to a fiber in $\mathcal{Y}$, as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}{}^T\mathbf{v} = \sum_c \mathbf{y}_c \left(\mathbf{y}_c{}^T\mathbf{v}\right), \tag{5}$$

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

This equation can be reformulated by $\mathcal{X}$ and $\{\mathbf{A}^T\}$. For ease of explanation, let $\mathcal{X}$ be a 3-order tensor and let a fiber $\mathcal{Y}_{:j_2j_3}$ correspond a column vector $\mathbf{y}_c$. By plugging Equation (3) into Equation (5), we obtain

$$\mathbf{v}' \leftarrow \sum_c \mathbf{y}_c \left(\mathbf{y}_c{}^T\mathbf{v}\right) = \sum_{\forall(j_2,j_3)} \mathcal{Y}_{:j_2j_3} \left(\mathcal{Y}_{:j_2j_3}{}^T\mathbf{v}\right)$$
$$= \sum_{\forall(j_2,j_3)} \left(\mathcal{X}\bar{\times}_2\mathbf{a}_{j_2}^{(n)}\bar{\times}_3\mathbf{a}_{j_3}^{(n)}\right)\left(\left(\mathcal{X}\bar{\times}_2\mathbf{a}_{j_2}^{(n)}\bar{\times}_3\mathbf{a}_{j_3}^{(n)}\right)^T\mathbf{v}\right).$$

As clarified in Equation (3), $\mathcal{X}\bar{\times}_2\mathbf{a}_{j_2}^{(n)}\bar{\times}_3\mathbf{a}_{j_3}^{(n)}$ 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 $\mathcal{X}$ increases explosively, as Lemmas 1 and 2 state.

**Lemma 1** (Scan cost of computing a fiber)**.** *Computing a fiber* on the fly *requires a complete scan of* $\mathcal{X}$.

*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:

$$\mathcal{X}\bar{\times}_n\mathbf{v} = \sum_{i_n=1}^{I_n} \mathcal{X}(\underbrace{:,\ldots,:}_{n-1}, i_n, \underbrace{:,\ldots,:}_{N-n})\mathrm{v}_{i_n}. \tag{6}$$

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

**Lemma 2** (Minimum scan cost of Naive S-HOT)**.** *Let $M_b$ 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 $\frac{I_n}{M_b}\prod_{p\neq n} J_p$ scans of $\mathcal{X}$ for computing Equation* (5).

*Proof.* Since we compute $\mathbf{y}_c\left(\mathbf{y}_c^T\mathbf{v}\right)$, $\mathbf{y}_c$ 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 $\frac{M_b}{I_n}$ fibers at the same time within one scan of $\mathcal{X}$. Therefore, Naive S-HOT requires at least $\frac{I_n}{M_b}\prod_{p\neq n} J_p$ scans of $\mathcal{X}$ to compute Equation (5). ■

## 4.2 Proposed: "S-HOT"

To avoid the explosion in the number of scans of $\mathcal{X}$ required, we propose S-HOT, which computes Equation (4) within two scans of $\mathcal{X}$. S-HOT progressively computes $\mathbf{v}'$ from each row vector of $\mathbf{Y}_{(n)}$. Specifically, $\mathbf{v}'$ is computed by:

$$1 \le \forall i \le I_n,\ \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i\mathbf{Y}_{(n)}\mathbf{v} = \bar{\mathbf{y}}_i\sum_{i=1}^{I_n}\mathbf{v}(i)\bar{\mathbf{y}}_i^T \tag{7}$$

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, Equation (7) should be computed in the following two steps:

$$\mathbf{s} \leftarrow \sum_{i=1}^{I_n}\mathbf{v}(i)\bar{\mathbf{y}}_i^T \tag{8}$$
$$1 \le \forall i \le I_n,\ \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i\mathbf{s}. \tag{9}$$

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. Algorithm 2 gives a formal description of S-HOT, and Lemma 3 states the number of scans required in the algorithm

**Lemma 3** (Scan cost of S-HOT)**.** S-HOT *requires only two scans of $\mathcal{X}$ for computing Equation* (4).

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

$$\bar{\mathbf{y}}_i \leftarrow [\mathcal{X}\times_{-n}\{\mathbf{A}^T\}]_{(n)}(i,:) = \sum_{p\in\Theta_i^{(n)}(\mathcal{X})}\mathcal{X}(p)\times_{-n}\{\mathbf{A}^T\}$$
$$= \sum_{p\in\Theta_i^{(n)}(\mathcal{X})}\mathcal{X}(p)\left[\circ_p^{-n}\{\mathbf{A}\}\right]_{(n)}, \tag{10}$$

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

**Algorithm 2:** Formal description for S-HOT.

**Input :** $N$-order Tensor: $\mathcal{X}$,
The size of the core tensor: $J_1 \times \cdots \times J_N$
**Output:** Core tensor: $\mathcal{G} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$,
Factor matrices: $\{\mathbf{A}\}$

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

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

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

23 **Subroutine** `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  $\mathbf{y_i} \leftarrow \sum_{p \in \Theta^{(n)}_i(\mathcal{X})} \mathcal{X}(p)\left[\circ^{-n}_p\{\mathbf{A}\}\right]_{(n)}$
27  $\mathbf{w}' \leftarrow \mathbf{w}' + (\mathbf{y}_i^T\mathbf{w})\mathbf{y}_i$
28  Deallocate $\mathbf{y}_i$
29  **return** $\mathbf{w}'$

Equation (8) can be computed progressively as follows:

$$\mathbf{s} \leftarrow \sum_{i=1}^{I_n} \mathbf{v}(i)\bar{\mathbf{y}}_i^T = \sum_{i=1}^{I_n} \mathbf{v}(i) \sum_{p \in \Theta^{(n)}_i(\mathcal{X})} \mathcal{X}(p)\left[\circ^{-n}_p\{\mathbf{A}\}\right]_{(n)}$$
$$= \sum_{p \in \Theta(\mathcal{X})} \mathbf{v}(i_n)\mathcal{X}(p)\left[\circ^{-n}_p\{\mathbf{A}\}\right]_{(n)}. \quad (11)$$

Thus, computing Equation (8) requires only one scan of $\mathcal{X}$. Similarly, Equation (9) also can be computed within one scan of $\mathcal{X}$. Therefore, Equation (7), consisting of Equation (8) and Equation (9), can be computed within two scans of $\mathcal{X}$. ■

## 4.3 Even faster: "S-HOT$_{scan}$"

We propose S-HOT$_{\text{scan}}$, which further reduces the number of scans of $\mathcal{X}$ at the expense of requiring multiple (disk-resident) copies of $\mathcal{X}$ sorted by different mode indices. In effect, S-HOT$_{\text{scan}}$ 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,

$$\mathbf{Y}_{(n)}\mathbf{V}\mathbf{\Sigma}^{-1} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}^{-1} = \mathbf{U}. \quad (12)$$

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

S-HOT$_{\text{scan}}$ 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:

$$\mathbf{w}' \leftarrow {\mathbf{Y}_{(n)}}^T\mathbf{Y}_{(n)}\mathbf{w} = \sum_{i=1}^{I_n}(\bar{\mathbf{y}}_i^T\mathbf{w})\bar{\mathbf{y}}_i. \quad (13)$$

The virtue of S-HOT$_{\text{scan}}$ is that it requires only one scan of $\mathcal{X}$ for calculating Equation (13), as Lemma 5 states.

**Lemma 4** (Scan cost for computing $\bar{\mathbf{y}}_i$)**.** *$\bar{\mathbf{y}}_i$ can be computed from by scanning the entries of $\mathcal{X}$ whose $n$-mode index is $i$.*

*Proof.* Proven by Equation (10). ■

**Lemma 5** (Scan cost of S-HOT$_{\text{scan}}$)**.** S-HOT$_{scan}$ *computes Equation* (13) *within one scan of $\mathcal{X}$ when $\mathcal{X}$ is sorted by the $n$-mode index.*

*Proof.* By Lemma 4, 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, Equation (13) 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.

The formal description for S-HOT$_{\text{scan}}$ 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 (Equation (12)), S-HOT$_{\text{scan}}$ still requires fewer scans of $\mathcal{X}$ than S-HOT since it saves one scan during $\mathbf{w}'$ computation, which is repeated more frequently.

Table 3 summaries the key differences of BaselineOpt, S-HOT, and S-HOT$_{\text{scan}}$ 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 1) the dimensionality, 2) the rank, 3) the order, and 4) the number of non-zero entries?

–**Q2.** Can S-HOT decompose real-world tensors that are both large-scale and high-order?

## 5.1 Experimental setting

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

(1) BaselineNaive: naive method computing $\mathcal{X} \times_{-n} \{\mathbf{A}\}$ in a straight-forward way.
(2) BaselineOpt: the state-of-the-art Memory Efficient Tucker decomposition which computes $\mathcal{Y}$ fiber by fiber.
(3) S-HOT: the proposed method that avoids *M-Bottleneck* by *on-the-fly* computation.
(4) S-HOT$_{\text{scan}}$: the scan-optimized variant of S-HOT.

For BaselineOpt and BaselineNaive, we use the implementation in MATLAB Tensor Toolbox 2.6 [3]. 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$_{\text{scan}}$.

**Dataset:** We use both of synthetic and real-world datasets. Specifically, synthetic datasets are mainly used 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. In addition, we also decompose a high-order real-world dataset called Microsoft Academic Graph [35], which cannot be decomposed by the baselines.

**Synthetic tensor:** To synthesize a random $N$-order tensor, we randomly sample $M$ tuples where each $n$-mode index is sampled from $[1, I_n]$. Once the indices of an tuple is sampled, the value of the tuple is set to 1. However, this does not mean that S-HOT is limited to binary tensors or our implementation is optimized for binary tensors. We choose binary tensors for simplicity because generating realistic values, while we control each factor, is not a trivial task.

As a default parameter setting, we used $N = 4$, $M = 10^5$; $I_n = 10^3$; and $J_n = 8$ for each $n$. These default values are chosen to compare the scalability of competitors. If we increase these values, both BaselineNaive and BaselineOpt run out of memory. 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. 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 [20], 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 (7) and (13))

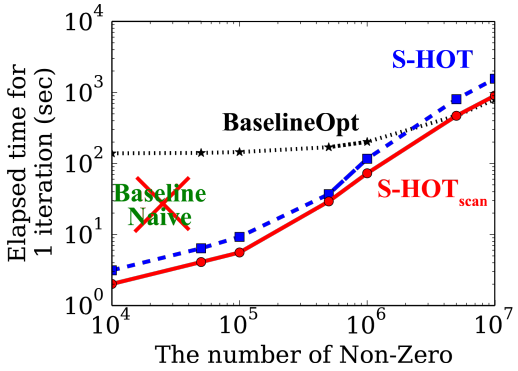



**Figure 2: S-HOT scales near linearly with respect to the number of non-zero entries.**

## 5.2 Evaluation

### 5.2.1 S-HOT scales up

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 elapsed wall clock time for a single iteration on synthetic tensors.

**Order:** First, we investigate the scalability of competitors with respect to the order by controlling the order of tensors from three to six while fixing the other factors to the default value. As shown in Figure 1(a), **S-HOT outperforms baselines**. BaselineNaive fails to decompose the 4-order tensor because it suffers from *intermediate explosion problem*. BaselineOpt is more efficient than BaselineNaive since it avoids the intermediate explosion problem, however, it fails to decompose a tensor whose order is higher than four due to *M-Bottleneck*. On the contrary, both S-HOT and S-HOT$_{\text{scan}}$ successfully decompose even the six-order tensor.

**Dimensionality:** Second, we investigate the scalability of the competitors with respect to the dimensionality. Specifically, we increase the dimensionality $I_n$ from $10^3$ to $10^7$, where $I_n = 10^3$ means that the tensor has size $10^3 \times 10^3 \times 10^3 \times 10^3$ since the default order is four. As shown in Figure 1(b), **S-HOT is orders-of-magnitude scalable than the baselines**. Specifically, BaselineNaive fails to decompose any 4-order tensors, and thus does not appear in the plot. In the case of BaselineOpt, the elapsed time for an iteration rapidly increases as the dimensionality grows, and eventually fails to decompose tensors with dimensionality larger than $10^4$. This is since the space for $\mathcal{Y}$ increases rapidly with respect to the size of dimensionality (*M-Bottleneck*). On the contrary, both S-HOT and S-HOT$_{\text{scan}}$ show better scalability with respect to the dimensionality. Specifically, S-HOT$_{\text{scan}}$ shows almost constant time, since it solves the transposed problem, whose size is only affected by rank and order. In the case of S-HOT, the elapsed time

**Table 4: Sample venue clustering results on Microsoft Academic Graph dataset**

| | |
|---|---|
| *CS-related* | 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, ... |
| *Nanotech.* | Nature Nanotechnology, PLOS ONE, Journal of Experimental Nanoscience, Journal of Nanoscience and Nanotechnology, Journal of Semiconductors, Trends in Biotechnology, ... |
| *Clinical* | European Journal of Cancer, PLOS Biology, Clinical and Applied Thrombosis-Hemostasis, Journal of Infection Prevention, RBMC Clinical Pharmacology, Regional Anesthesia and Pain ... |

is affected by dimensionality, and increases after $10^6$. Before that, the effect of dimensionality is negligible because the cost for outer-product in lines 19 and 21 of Algorithm 2 is the major factor within that range.

**Rank:** We also investigate the scalability with respect to the rank. In this experiment, we set dimensionality to $20,000$ because it is better to experimentally show the difference in the scalability of competitors, but the overall trends do not change with the other parameter settings. As shown in Figure 1(c), **S-HOT outperforms baselines.** Similar to the other evaluation results, BaselineNaive does not appear in this plot, and BaselineOpt fails for rank larger than 6. On the contrary, both S-HOT and S-HOT$_{\text{scan}}$ successfully perform Tucker decomposition with larger rank. In most cases, S-HOT$_{\text{scan}}$ is faster than S-HOT, but the difference between two methods decreases as rank increases. This is because, as the rank increases, the cost for outer product becomes the major bottleneck, which is common in BaselineOpt, S-HOT, and S-HOT$_{\text{scan}}$.

**Nonzero entries:** We investigate the scalability of the competitors with respect to the number of non-zero entries by evaluating the methods using tensors with $10^4$ to $10^7$ non-zero entries. As shown in Figure 2, **both S-HOT and S-HOT$_{\text{scan}}$ show near linear scalability** with respect to the number of non-zero entries. This is because both methods scan all the non-zero entries, and processing each non-zero entry takes almost same time, which is decided by outer products and inner products. With respect to the number of non-zeros, BaselineOpt shows better scalability, since it *explicitly materializes* $\mathcal{Y}$. Once $\mathcal{Y}$ is materialized, since its size does not depend on the number of non-zero entries, elapsed time is less affected by the number of non-zero entries.

### 5.2.2 S-HOT at work

We test the scalability of S-HOT on Microsoft Academic Graph dataset [35] (a snapshot on Feb 5, 2016). The dataset 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. We model this as a 4-order tensor whose modes are (Author, Venue, Year, Keyword). Since papers having missing attributes are ignored, the final tensor is of size $9380418 \times 18894 \times 2016 \times 37000$.

We note that, since this tensor is high-order and large, *both baselines* fail to handle it running out of memory. However, S-HOT successfully computes Tucker decomposition of the tensor.

To interpret the result of Tucker decomposition better, we runs k-means clustering treating the factor matrices as the low-rank embeddings of each mode, as suggested in [18]. Specifically, we perform Tucker decomposition with a core tensor of size $8 \times 8 \times 8 \times 8$ and 30 iterations, and k-means by setting the number of clusters to 400 and max iteration to 100.

Table 4 shows sample clusters in the venue mode. The first cluster seems to be related to Computer Science. The second one contains many nano-technology-related venues such as Nautre Nanotechnology, Journal of Experimental Nanoscience. The third one have 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.

## 6. CONCLUSIONS

In this paper, we propose S-HOT, a scalable Tucker decomposition method. We offer two version of S-HOT, which optimize space and time, resp., and show that they successfully decompose large tensors that cannot be decomposed by baseline methods. Furthermore, we provide a theoretical analysis on the number of scans of data required in the methods. To sum up, our contributions are as follows.

- **Handling Bottleneck:** *M-Bottleneck*, the scalability bottleneck of existing Tucker decomposition methods, is identified (Section 3.3) and avoided by a novel approach based on an *on-the-fly computation* (Section 4).
- **Algorithm Design:** We propose S-HOT, which employs the *on-the-fly computation* approach. Moreover, S-HOT is carefully optimized for large-scale disk-resident tensors. This enables S-HOT to offer up to *1000× better scalability* than baseline methods (Section 5.2.1).
- **Theoretical analysis:** We prove the amount of memory space and the number of scans of $\mathcal{X}$ that S-HOT requires (Table 2 and Lemma 3). We also show that the number of scans can be reduced by half if we keep multiple copies of the tensor which are sorted by different modes (Lemma 5).

For reproducibility and extensibility of our work, we make the source code of S-HOT publicly available at http://dm.postech.ac.kr/shot.

## Acknowledgement

This material is based upon work supported by ICT R&D program of MSIP/IITP under Grant No. [14-824-09-014, Basic Software Research in Human-level Lifelong Machine Learning (Machine Learning Center)].

# 7. REFERENCES


[1] E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener. Modeling and multiway analysis of chatroom tensors. In *ISI*, 2005.

[2] W. Austin, G. Ballard, and T. G. Kolda. Parallel tensor compression for large-scale scientific data. In *IPDPS*, 2016.

[3] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6. Available online, February 2015.

[4] 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.

[5] Y. Cai, M. Zhang, D. Luo, C. Ding, and S. Chakravarthy. Low-order tensor decompositions for social tagging recommendation. In *WSDM*, 2011.

[6] Y. Chi, B. L. Tseng, and J. Tatemura. Eigen-trend: Trend analysis in the blogosphere based on singular value decompositions. In *CIKM*, 2006.

[7] J. H. Choi and S. Vishwanathan. Dfacto: Distributed factorization of tensors. In *NIPS*, 2014.

[8] J. E. Cohen, R. C. Farias, and P. Comon. Fast decomposition of large nonnegative tensors. *IEEE Signal Processing Letters*, 22(7):862–866, 2015.

[9] A. L. de Almeida and A. Y. Kibangou. Distributed computation of tensor decompositions in collaborative networks. In *CAMSAP*, pages 232–235, 2013.

[10] A. L. De Almeida and A. Y. Kibangou. Distributed large-scale tensor decomposition. In *ICASSP*, 2014.

[11] T. Franz, A. Schultz, S. Sizov, and S. Staab. Triplerank: Ranking semantic web data by tensor decomposition. In *ISWC*, 2009.

[12] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.

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

[14] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. Haten2: Billion-scale tensor decompositions. In *ICDE*, 2015.

[15] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *KDD*, 2012.

[16] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[17] T. G. Kolda, B. W. Bader, and J. P. Kenny. Higher-order web link analysis using multilinear algebra. In *ICDM*, 2005.

[18] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, 2008.

[19] H. Lamba, V. Nagarajan, K. Shin, and N. Shajarisales. Incorporating side information in tensor completion. In *WWW Companion*, 2016.

[20] 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.

[21] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, 2011.

[22] S. Moghaddam, M. Jamali, and M. Ester. Etf: extended tensor factorization model for personalizing prediction of review helpfulness. In *WSDM*, 2012.

[23] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *ECML/PKDD*, pages 521–536, 2012.

[24] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Parcube: Sparse parallelizable candecomp-parafac tensor decomposition. *TKDD*, 10(1):3, 2015.

[25] 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.

[26] I. Perros, R. Chen, R. Vuduc, and J. Sun. Sparse hierarchical tucker factorization and its application to healthcare. In *ICDM*, 2015.

[27] A. H. Phan and A. Cichocki. Parafac algorithms for large-scale problems. *Neurocomputing*, 74(11):1970–1984, 2011.

[28] S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *WSDM*, 2010.

[29] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011.

[30] L. Sael, I. Jeon, and U. Kang. Scalable tensor mining. *Big Data Research*, 2(2):82–86, 2015.

[31] K. Shin, B. Hooi, and C. Faloutsos. M-zoom: Fast dense-block detection in tensors with quality guarantees. In *ECML/PKDD*, 2016.

[32] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. In *ICDM*, 2014.

[33] K. Shin, S. Lee, and U. Kang. Fully scalable methods for distributed tensor factorization. *TKDE*, PP(99):1–1, 2016.

[34] N. D. Sidiropoulos and A. Kyrillidis. Multi-way compressed sensing for sparse low-rank tensors. *IEEE Signal Processing Letters*, 19(11):757–760, 2012.

[35] 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.

[36] A. Smilde, R. Bro, and P. Geladi. *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.

[37] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: Dynamic tensor analysis. In *KDD*, 2006.

[38] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: A novel approach to personalized web search. In *WWW*, 2005.

[39] C. E. Tsourakakis. Mach: Fast randomized tensor decompositions. In *SDM*, pages 689–700. SIAM, 2010.

[40] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[41] 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.