HaTen2: Billion-scale Tensor Decompositions

Inah Jeon †, Evangelos E. Papalexakis †, U Kang ‡, Christos Faloutsos ‡

† Department of Computer Science, KAIST
291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea

‡ Computer Science Department, Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213, United States

Abstract—How can we find useful patterns and anomalies in large scale real-world data with multiple attributes? For example, network intrusion logs, with (source-ip, target-ip, port-number, timestamp)? Tensors are suitable for modeling these multi-dimensional data, and widely used for the analysis of social networks, web data, network traffic, and in many other settings. However, current tensor decomposition methods do not scale for tensors with millions and billions of rows, columns and ‘fibres’, that often appear in real datasets. In this paper, we propose HaTen2, a scalable distributed suite of tensor decomposition algorithms running on the MapReduce platform. By carefully reordering the operations, and exploiting the sparsity of real world tensors, HaTen2 dramatically reduces the intermediate data, and the number of jobs. As a result, using HaTen2, we analyze big real-world tensors that can not be handled by the current state of the art, and discover hidden concepts.

I. INTRODUCTION

How can we find useful patterns and anomalies in large scale real-world data that has multiple attributes? For instance, network intrusion logs, where we record data of the form (source-ip, target-ip, port-number, timestamp)? Tensors are multi-dimensional arrays and are suitable for modeling multi-aspect data. Tensor decompositions are widely used tensor analysis tools for various real-world data such as knowledge bases [1], web data [2], network traffic data [3], and many others [4], [5]. Using tensor decompositions, we find latent factors (or relations) between the data. These latent factors can be roughly and informally seen as soft-clustering of the data; in the (source-ip, target-ip, port-number, timestamp) example, decomposing this tensor into $R$ latent factors corresponds to finding $R$ clusters of source-ip’s that talk to a set of target-ip’s on a set of port numbers and for a specific amount of time.

There are two major tensor decompositions: PARAFAC and Tucker. Since there is no single generalization of the Singular Value Decomposition (SVD) for tensors, both PARAFAC and Tucker can be thought of as extensions of SVD to higher dimensions. PARAFAC is mostly used when one is interested in decomposing a tensor into a sum of rank-one tensors (and is interested in the latent factors), while Tucker is more appropriate for tensor compression as well as detection of relations between latent factors.

Tensors have been used for various purposes of data mining and analysis over the years. However, recently, the size of real-world tensors size has started to become prohibitively large, approaching millions and billions of rows, columns, and ‘fibres’. The problem is that most of tensor decomposition algorithms do not scale to deal with those huge tensors due to the high computational costs and space. The main challenge is to overcome the intermediate data explosion problem where the amount of intermediate data of an operation exceeds the capacity of a single machine or even a cluster. For example, in the $n$-mode product in Tucker decomposition, a straightforward implementation using distributed systems would require 1 Exabytes ($=10^{18}$ bytes) of intermediate data, assuming the size of the input tensor $X \in \mathbb{R}^{I \times J \times K}$ is $I = J = K = 1$ millions. Thus, we need to develop scalable and distributed tensor decomposition algorithms.

In this paper, we propose HaTen2 (which stands for HADOOP Tensor method for 2 decompositions), a scalable tensor decomposition suite of methods for Tucker and PARAFAC decompositions on HADOOP [6], the open-source version of the MapReduce framework [7]. HaTen2 solves the intermediate data explosion problem by carefully reordering the operations, and exploiting the sparsity of real world tensors. Furthermore, HaTen2 integrates several redundant jobs to reduce the running time significantly. As a result, HaTen2 is able to analyze data that are several orders of magnitude larger than what the state of the art can handle.

Our main contributions are the following:

- **Algorithm.** HaTen2 unifies the large scale Tucker and PARAFAC tensor decomposition algorithms on MapReduce into a general framework, such that the intermediate data size and the running time are minimized.
- **Scalability.** HaTen2 decomposes $100 \times$ larger tensors compared to existing methods, as shown in Figure 1. Furthermore, HaTen2 enjoys near linear scalability on the number of machines.
- **Discovery.** We discover interesting concepts by applying HaTen2 on Freebase knowledge base tensor with 23 millions of entities and 99 millions of facts, which was hard to analyze by existing methods.

The rest of paper is organized as follows. Section II presents the preliminaries of the tensor and its decompositions. Sec-
In this section, we describe the preliminaries on tensor and its decompositions. Table I shows the definitions of symbols used in this paper. Matrices are denoted by boldface capitals (e.g. $\mathbf{B}$), and the $r$th row of the matrix $\mathbf{B}$ is denoted by $\mathbf{b}_r$. Vectors are denoted by boldface lowercases (e.g. $\mathbf{a}$).

### A. Tensor

Tensor is a multi-dimensional array. Each `dimension` of a tensor is called `mode` or `way`. An $N$-mode or $N$-way tensor is denoted by $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. $\text{bin}(\mathbf{X})$ denotes a function that converts non-zero elements in $\mathbf{X}$ to 1. $\text{nnz}(\mathbf{X})$ means the number of non-zero elements of $\mathbf{X}$, and $\text{idx}(\mathbf{X})$ means the set of indexes (e.g. $(i,j,k)$ for 3-mode tensor $\mathbf{X}$) of non-zero elements in $\mathbf{X}$. $i$-th slice of $\mathbf{X}$ is denoted by $\text{nnz}(\mathbf{X}_{i,:})$, and $ij$-th fiber of $\mathbf{X}$ is denoted by $\text{nnz}(\mathbf{X}_{ij,:})$. We refer the reader to [8] for further details.

### B. Tensor Decomposition

Tensor decomposition is a general tool for tensor analysis. Using tensor decomposition, we find latent factor or relations among data. In this paper, we focus on two major tensor decompositions, PARAFAC and Tucker.

1) **PARAFAC Decomposition**

PARAFAC (parallel factors) decomposition [9], also called CANDECOMP (canonical decomposition), decomposes a tensor into a sum of rank-one tensors. There has been rich literature on algorithms for the PARAFAC decomposition, a concise summary thereof can be found in [10].

![Fig. 2: Rank-$R$ PARAFAC decomposition of a three-way tensor.](image)

\[
\mathbf{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r
\]

where, $R$ is a positive integer, and $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$,

![Fig. 1: Data scalability of our proposed HATEN2-DRI compared to other methods, for Tucker decomposition. The datasets are explained in detail at Section IV-A. o.o.m: out of memory. Compared to the Tensor Toolbox, HATEN2-DRI decomposes 10 $\sim$ 100× larger data. Among the variants of HATEN2, HATEN2-DRI is the fastest, and analyzes 10× denser data than HATEN2-DNN.](image)
and $C \in \mathbb{R}^{K \times R}$ are the factor matrices. Figure 2 shows the 3-way PARAFAC tensor decomposition.

PARAFAC decomposition for $N$-way tensor: Given an $N$-way tensor $X \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and rank $R$, PARAFAC decomposition factorizes the tensor into $N$ factor matrices, $A^{(1)}, A^{(2)}, \ldots, A^{(N)}$, as follows:

$$X \approx [A^{(1)}, A^{(2)}, ..., A^{(N)}] = \sum_{r=1}^{R} a_r^{(1)} \circ a_r^{(2)} \circ \ldots \circ a_r^{(N)}.$$ 

where, $R$ is a positive integer, and $A^{(1)} \in \mathbb{R}^{I_1 \times R}$, $A^{(2)} \in \mathbb{R}^{I_2 \times R}$, ..., $A^{(N)} \in \mathbb{R}^{I_N \times R}$ are the factor matrices.

PARAFAC-ALS. Algorithm 1 shows the alternating least squares algorithm for 3-way PARAFAC decomposition.

Algorithm 1: 3-way PARAFAC-ALS.

**Input:** Tensor $X \in \mathbb{R}^{I \times J \times K}$, rank $R$, maximum iterations $T$

**Output:** PARAFAC decomposition $\lambda \in \mathbb{R}^{R \times I}$, $A \in \mathbb{R}^{I \times K}$, 

$
\begin{align*}
B & \in \mathbb{R}^{J \times R},
C & \in \mathbb{R}^{K \times R}
\end{align*}
$

1: Initialize $A, B, C$;
2: for $t = 1, ..., T$ do
3: $A \leftarrow X_{(1)} (C \odot B) (C^T C + B^T B)^{\dagger}$;
4: Normalize columns of $A$ (storing norms in vector $\lambda$);
5: $B \leftarrow X_{(2)} (A \odot C) (C^T C + A^T A)^{\dagger}$;
6: Normalize columns of $B$ (storing norms in vector $\lambda$);
7: $C \leftarrow X_{(3)} (B \odot A) (B^T B + A^T A)^{\dagger}$;
8: Normalize columns of $C$ (storing norms in vector $\lambda$);
9: if convergence criterion is met then
10: break for loop;
11: end if
12: end for
13: return $A, B, C$;

2) Tucker Decomposition: In Tucker decomposition [11], called N-mode PCA or N-mode SVD, a tensor is decomposed into a core tensor and factor matrices of each mode. The factor matrices represent the principal components of each mode and the core tensor represents the interactions between the different components. Tucker decomposition is a more generalized version of PARAFAC decomposition, since the factors interact with all pairs of other factors.

**Tucker decomposition for 3-way tensor.** The 3-way tensor is decomposed as follows.

$$X \approx [\mathcal{G}; A, B, C] = \mathcal{G} \times_1 A \times_2 B \times_3 C$$

$$= \sum_{p=1}^{P} \sum_{q=1}^{Q} \sum_{r=1}^{R} g_{pq} a_p \circ b_q \circ c_r$$

where, $G \in \mathbb{R}^{P \times Q \times R}$ is the core tensor, and $A \in \mathbb{R}^{I \times P}$, $B \in \mathbb{R}^{J \times Q}$, and $C \in \mathbb{R}^{K \times R}$ are the factor matrices. Figure 3 shows the Tucker decomposition of a 3-way tensor.

**Tucker decomposition for N-way tensor.** The N-way tensor is decomposed as follows.

$$X \approx [\mathcal{G}; A^{(1)}, A^{(2)}, ..., A^{(N)}]$$

$$= \mathcal{G} \times_1 A^{(1)} \times_2 A^{(2)} \times_3 \ldots \times_N A^{(N)}$$

where, $G \in \mathbb{R}^{I_1 \times J_2 \times \ldots \times J_N}$ is the core tensor, and $A^{(1)} \in \mathbb{R}^{I_1 \times J_1}$, $A^{(2)} \in \mathbb{R}^{I_2 \times J_2}$, ..., and $A^{(N)} \in \mathbb{R}^{I_N \times J_N}$ are the factor matrices.

**Tucker-ALS.** Tucker-ALS algorithm uses an alternating least squares approach. For updating each factor, there are some approaches such as SVD, Bauer-Rutishauser, Gram-Schmidt and NIPALS [12]. Algorithm 2 shows the standard SVD-based algorithm for 3-way Tucker decomposition.

Algorithm 2: 3-way Tucker-ALS

**Input:** Tensor $X \in \mathbb{R}^{I \times J \times K}$, desired size: $P \times Q \times R$

**Output:** Core tensor $G \in \mathbb{R}^{P \times Q \times R}$ and orthogonal factor matrices $A \in \mathbb{R}^{I \times P}$, $B \in \mathbb{R}^{J \times Q}$, and $C \in \mathbb{R}^{K \times R}$

1: Initialize $B, C$;
2: repeat
3: $Y_1 \leftarrow X \times_2 B^T \times_3 C^T$;
4: $A \leftarrow P$ leading left singular vectors of $Y_1$;
5: $Y_2 \leftarrow X \times_3 A^T \times_1 C^T$;
6: $B \leftarrow Q$ leading left singular vectors of $Y_2$;
7: $Y_3 \leftarrow X \times_1 A^T \times_2 B^T$;
8: $C \leftarrow R$ leading left singular vectors of $Y_3$;
9: $G \leftarrow Y_3 C$;
10: until $\|G\|$ ceases to increase or the maximum number of outer iterations is exceeded.

### III. Proposed Method

In this section, we describe main ideas of HATEn2, the proposed distributed MapReduce algorithms for large scale tensor decompositions.

A. Overview

How can we design scalable and efficient PARAFAC/Tucker decomposition algorithms for very large tensors? The most challenging parts of those algorithms are $n$-mode matrix product $Y \leftarrow X \times_2 B^T \times_3 C^T$ (lines 3, 5, and 7 of Algorithm 2) in Tucker-ALS, and Khatri-rao product $Y \leftarrow X \times_{(1)} (C \odot B)$ (lines 3, 5, and 7 of Algorithm 1) in PARAFAC-ALS. There are several challenges in designing efficient distributed algorithms for these operations.

**TABLE III:** Summary of costs in all methods for computing $X \times_2 B \times_3 C$ in Tucker decomposition. We replace $nnz(X \times_2 B)$ with $nnz(X)Q$ according to the estimation of $nnz(X \times_2 B)$ in Lemma 3 of Appendix A.

<table>
<thead>
<tr>
<th>Method</th>
<th>Max. Intermediate Data</th>
<th>Total Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>HATEn2-Tucker-Naive</td>
<td>$nnz(X) + IJK$</td>
<td>$Q + R$</td>
</tr>
<tr>
<td>HATEn2-Tucker-DNN</td>
<td>$nnz(X)QR$</td>
<td>$Q + R + 2$</td>
</tr>
<tr>
<td>HATEn2-Tucker-DRN</td>
<td>$nnz(X)(Q + R)$</td>
<td>$Q + R + 1$</td>
</tr>
<tr>
<td>HATEn2-Tucker-DRI</td>
<td>$nnz(X)(Q + R)$</td>
<td>$2$</td>
</tr>
</tbody>
</table>

- Minimize intermediate data. During the computation, huge intermediate data are generated in the shuffle stage. How can we minimize the intermediate data?
**TABLE II**: Comparison of all methods experimented. Our proposed and recommended method is HATEN2-DRI (or just HATEN2) which incorporates all the proposed ideas.

<table>
<thead>
<tr>
<th>Method</th>
<th>Distributed?</th>
<th>Decoupling the Steps (D/N)</th>
<th>Remove Dependencies (R/N)</th>
<th>Integrating Jobs (I/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensor Toolbox</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HATEN2-Naive</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HATEN2-DNN</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HATEN2-DRN</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HATEN2-DRI (or just HATEN2)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **Minimize disk accesses.** How can we minimize the disk accesses to decrease the running time?
- **Minimize jobs.** How can we minimize the number of MAPREDUCE jobs to decrease the running time?

Our main ideas to address the challenges are as follows.

- **Decoupling the steps in n-mode vector product.** We decouple the multiplication and the addition steps in n-mode vector product by introducing a new operation called Hadamard-and-Merge which leads to decreasing the intermediate data size (Section III-B2).
- **Removing dependencies in sequential products.** We remove dependencies by carefully reordering the computations, and exploiting the sparsity of real world tensors. It leads to further decreasing the intermediate data size (Section III-B3).
- **Integrating jobs by increasing memory usage.** We integrate multiple MAPREDUCE jobs by increasing memory usage. The idea leads to minimizing the number of jobs and the disk accesses (Section III-B4).

Figure 4 shows the framework of our proposed HATEN2-DRI (or just HATEN2) method which contains all the above ideas. Note that although the computations for the two decompositions Tucker and PARAFAC are different, our HATEN2 unifies them into a general framework where the two methods differ only at the final merge step: HATEN2-Tucker uses CrossMerge, while HATEN2-PARAFAC uses PairwiseMerge (see Section III-B4 for details). In the next subsection, we describe the main ideas in detail.

**TABLE IV**: Summary of costs in all methods for computing $X_{(1)}(B \odot C)$ in PARAFAC decomposition.

<table>
<thead>
<tr>
<th>Method</th>
<th>Max. Intermediate Data</th>
<th>Total Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>HATEN2-PARAFAC-Naive</td>
<td>$\text{nnz}(X) + IJK$</td>
<td>$2R$</td>
</tr>
<tr>
<td>HATEN2-PARAFAC-DNN</td>
<td>$\text{nnz}(X) + J$</td>
<td>$4R$</td>
</tr>
<tr>
<td>HATEN2-PARAFAC-DRN</td>
<td>$2\text{nnz}(X)R$</td>
<td>$2R + 1$</td>
</tr>
<tr>
<td>HATEN2-PARAFAC-DRI</td>
<td>$2\text{nnz}(X)R$</td>
<td>$2$</td>
</tr>
</tbody>
</table>

**B. Method Details**

In the following, we start with a naive method, and improve the method gradually by adding several ideas one by one until we reach the final method HATEN2-DRI (or just HATEN2). Figure 5 and Table II summarize the differences between all methods. Table III and IV show the total costs of all the methods in terms of the maximum intermediate data size, and the number of total MAPREDUCE jobs.

1) **Naive Method**: The most naive method is the straightforward implementation of the idea in MET [5], the state-of-the-art single machine implementation which was adopted by Tensor Toolbox. The main idea is to perform each n-mode vector product separately. HATEN2-Tucker-Naive computes $T = X \times B^T$ first by performing $X \times_b b_q^T$ operation $Q$ times, and then computes $T \times C^T$ by performing $T \times C^T$ operation $R$ times, where $b_q$ and $c_r$ are the qth row of B, and the rth row of C, respectively. Algorithm 3 shows HATEN2-Tucker-Naive method.

**Algorithm 3**: HATEN2-Tucker-Naive for computing $Y = X \times_b b^T \times C^T$

**Input**: Tensor $X \in \mathbb{R}^{I \times J \times K}$, and factor matrices $B \in \mathbb{R}^{J \times Q}$, $C \in \mathbb{R}^{K \times R}$

**Output**: Tensor $Y \in \mathbb{R}^{I \times Q \times R}$

1. for $q=1...Q$ do
2. $T_q \leftarrow X \times_b b_q^T$
3. end for
4. for $r=1...R$ do
5. $Y_r \leftarrow T \times C^T$
6. end for

Similarly, HATEN2-PARAFAC-Naive computes $T_r = X \times_b b_{q_r}^T$ first, and then computes $Y_r = T_r \times C^T$. It computes $Y$ by performing these operations $R$ times. Algorithm 4 shows HATEN2-PARAFAC-Naive method.

**Algorithm 4**: HATEN2-PARAFAC-Naive for computing $Y = X_{(1)}(C \odot B)$

**Input**: Tensor $X \in \mathbb{R}^{I \times J \times K}$, and factor matrices $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$

**Output**: Tensor $Y \in \mathbb{R}^{I \times R}$

1. for $r=1...R$ do
2. $T_r \leftarrow X \times_b b_{q_r}^T$
3. $Y_r \leftarrow T_r \times C^T$
4. end for

**MAPREDUCE algorithm**: The MAPREDUCE algorithm of n-mode vector product $X \times_b b_q^T$ in HATEN2-Naive is as follows.

- **MAP**: map $<i, j, k, X(i, j, k) >$ on $(iK + k)$, such that tuples with the same key are shuffled to the same reducer in the form of $<key: (iK + k), values: \{(j, X(i, j, k)) | v(i, j, k) \in \text{idx}(X_{i,k})\} >$, and send $b_q^T$ to the all the reducers in the form of $<key: (iK + k), values: \{(j, b_q(j)) | v(j) \in \text{idx}(b_q)\} >$.
- **REDUCE**: take $<key: (iK + k), values: \{(j, b_q(j)) | v(j) \in \text{idx}(b_q)\}, \{(j, X(i, j, k)) | v(i, j, k) \in \text{idx}(X_{i,k})\} >$, and emit $<i, k, \sum_{j=1}^J X(i, j, k)b_q(j) >$.
The reducer processing the key $iK + k$ receives the non-zero elements of $X_{i,k}$ and $b_q$. Then it performs the inner product of the two vectors, and outputs an element of the result tensor $\mathcal{F}_{ij}$. The $\mathcal{F} \times _3 c^T_l$ operation is handled in the same manner. Although simple, this naive implementation has too much overhead because 1) the vector $b^T_q$ is copied $IK$ times which eventually generates too much intermediate data $(nnz(X) + IJK)$, and 2) the vector $b^T_q$ might not fit in the memory of a machine when $J$ is very large. How can we improve this naive method? In the following three subsections we incrementally improve the naive method.

2) Decoupling the Steps in n-mode Vector Product: The first idea to improve the naive method is to make the $n$-mode vector product $\times _n$ scalable. As we saw in the previous subsection, the naive algorithm which broadcasts the vector $b^T_q$ has too much overhead. Our idea, called Hadamard-and-Merge, is to decouple the product into two steps: the Hadamard product step where the element of the vector is multiplied with the corresponding element of the tensor, and the merge step where the multiplied values are summed. Hadamard-and-Merge operation comprises the two following operations: $n$-mode vector Hadamard product and Collapse.

**Definition 1 (n-mode vector Hadamard product):** The $n$-mode vector Hadamard product of a tensor $X \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n}$ and a vector $v \in \mathbb{R}^{I_n}$ is denoted by $X \times _n v$, and is of size $I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$. It is defined by

$$(X \times _n v)_{i_1,\ldots,i_{n-1},i_{n+1},\ldots,i_N} = x_{i_1,\ldots,i_{n-1},i_n \cdot v_{i_n}}.$$ 

**Definition 2 (Collapse($X$)$_n$):** The Collapse operation of a tensor $X \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ on mode $n$ is denoted by Collapse($X$)$_n$ and is of size $I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$. It is defined by

$$(\text{Collapse}(X)_n)_{i_1,\ldots,i_{n-1},i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} x_{i_1,\ldots,i_n,\ldots,i_N}.$$ 

Intuitively, the $n$-mode vector Hadamard product is a generalization of Hadamard product of two vectors. The Collapse($X$)$_n$ operation sums up all the values of a tensor $X$ across the mode $n$. With these definitions, HATEN2-DNN expresses the original $n$-mode vector product $X \times _n b^T_q$ by Collapse($X \times _n b^T_q$)$_2$. By decoupling the $n$-mode vector product into two steps, HATEN2-DNN greatly decreases the intermediate data size of HATEN2-Naive from $nnz(X) + IJK$ to $nnz(X)QIR$ for Tucker, and from $nnz(X) + IJK$ to $nnz(X) + J$ for PARAFAC. Algorithms 5 and 6 show HATEN2-DNN for Tucker and PARAFAC decompositions, respectively. In HATEN2-Tucker-DNN, we compute $\mathcal{F} = X \times _3 b^T \in \mathbb{R}^{I \times Q \times K}$ by iteratively performing $\mathcal{F}_q = X \times _n b^T_q$ for $Q$ times, and then merging them using the operation Collapse($\mathcal{F}$)$_2$. \mathcal{F} \times _3 C^T$ operation is handled in the same manner. In HATEN2-PARAFAC-DNN, Collapse is applied right after the individual $n$-mode vector Hadamard product.

**Algorithm 5:** HATEN2-Tucker-DNN for computing $Y = X \times _2 b^T \times _3 C^T$.

**Input:** Tensor $X \in \mathbb{R}^{I \times J \times K}$, and factor matrices $B \in \mathbb{R}^{J \times Q}$, $C \in \mathbb{R}^{K \times R}$.

**Output:** Tensor $Y \in \mathbb{R}^{I \times Q \times R}$.

1. for $q=1\ldots Q$ do
2. $ \mathcal{F}_q \leftarrow X \times _n b^T_q$;
3. end for
4. $ \mathcal{F} \leftarrow \text{Collapse}(\mathcal{F})_2$;
5. for $r=1\ldots R$ do
6. $ \mathcal{F}_r \leftarrow \mathcal{F} \times _3 C^T$;
7. end for
8. $Y \leftarrow \text{Collapse}(\mathcal{F})_3$;

**MapReduce algorithm.** The MapReduce algorithms of $n$-mode vector Hadamard product and Collapse($X$)$_n$ in HATEN2-DNN are expressed as follows.

$\langle X \times _n b^T_q \rangle$:
- **MAP:** map $\langle i, j, k, X(i,j,k) \rangle$ on $j$, and $\langle j, q, b_q(j) \rangle$ on $j$ such that tuples with the same key are shuffled to the same reducer in the form of $\langle \text{key}; j, \text{values:} (q, b_q(j)), \{(i, k, X(i,j,k))|\forall(i,k) \in \text{idx}(X_{i,k})\} \rangle$. 

Fig. 4: General computational framework in HATEN2 for Tucker and PARAFAC decompositions ($Q = R$ in PARAFAC). Although the two decompositions are different, our HATEN2 unifies them into a general framework where the two methods differ only at the final merge step: CrossMerge for HATEN2-Tucker, and PairwiseMerge for HATEN2-PARAFAC (see Section III-B4 for details).
have a significant problem: they have dependencies in their computation sequences. In Tucker decomposition, to compute \( Y ← X ×_2 B^T ×_3 C^T \), both of the previous methods first compute \( T = X ×_2 B^T \) by multiplying \( X \) and columns of \( B \), and then multiply \( T \) with the columns of \( C \). That is, the second step cannot be initiated until the first step is finished. Similarly, in PARAFAC decomposition, computing \( Y ← X(1) (C ⊙ B) \) has a dependency: \( X \) is multiplied with \( b^T \) first, and the result is multiplied with \( c^T \). These dependencies in the computation have the following problems.

- Too large intermediate data: in Tucker decomposition, the number \( nnz(T) \) of nonzero elements in \( T = X ×_2 B^T \) is estimated to be \( nnz(X)Q \) for a sparse tensor \( X \), as described in Lemma 3 of Appendix A. Thus, multiplying \( T \) with \( C^T \) would require intermediate data of size \( nnz(X)QR \) which is prohibitively large.
- Too many MapReduce jobs: in PARAFAC decomposition, HATEN2-PARAFAC-DNN requires \( 4R \) MapReduce jobs, since the first multiplication with \( b^T \), and the second multiplication with \( c^T \) are performed in sequence.

Our idea to solve the problems is to remove the dependencies by carefully reordering the computations, and exploiting the sparsity of real world tensors. Before describing the details, we introduce two new operations CrossMerge and PairwiseMerge as follows.

**Definition 3 (CrossMerge):** The CrossMerge operation of \( N = 1 \) tensors \( X_1 ∈ \mathbb{R}^{I_1 × J_1 × J_N} \), ..., \( X_{n-1}, X_{n+1}, ..., X_N ∈ \mathbb{R}^{I_1 × J_1 × J_N} \) on the mode \( n \) is denoted by \( \text{CrossMerge}(X_1, ..., X_{n-1}, X_{n+1}, ..., X_N) \). It is defined by 

\[
(\text{CrossMerge}(X_1, ..., X_{n-1}, X_{n+1}, ..., X_N))_{n,j_1,...,j_N} = \]

Fig 5: Comparison of all HATEN2 variants for Tucker decomposition. Areas with the same color are sent to the same reducer in the MapReduce jobs for \( n \)-mode (Hadamard) product.

**Algorithm 6: HATEN2-PARAFAC-DNN** for computing \( Y ← X(1)(C ⊙ B) \)

**Input:** Tensor \( X ∈ \mathbb{R}^{I × J × K} \), and factor matrices \( B ∈ \mathbb{R}^{J × R} \), \( C ∈ \mathbb{R}^{K × R} \).

**Output:** Tensor \( Y ∈ \mathbb{R}^{I × R} \)

1: for \( r = 1,...,R \) do
2: \( T'_r ← Xb^T_r ; \)
3: \( T_r ← \text{Collapse}(T'_r)_2 ; \)
4: \( Y'_r ← T_r⊙3c^T_r ; \)
5: \( Y_r ← \text{Collapse}(Y'_r)_3 ; \)
6: end for

- **REDUCE:** take \(<\text{key}:j \), \( values: (q, b_q(j)), \{(i, k, X(i,j,k))|v(i,k) ∈ id x(X_{i;k})\}> \) and emit \( i,j,k,q,X(i,j,k)b_q(j) \) for each \( (i,k) ∈ id x(X_{i;k}) \).

**<Collapse(\( T \))>**

- **MAP:** map \( <i,j,k,q,X(i,j,k)B(j,q)> \) on \((iK + k) \) such that tuples with the same key are shuffled to the same reducer in the form of \(<\text{key}: (iK + k) \), \( values: \{(q, X(i,j,k)B(j,q))|v(i,k,q) ∈ id x(T_{i;k})\}> \).

- **REDUCE:** take \(<\text{key}: (iK + k) \), \( values: \{(q, X(i,j,k)B(j,q))|v(i,k,q) ∈ id x(T_{i;k})\}> \) and emit \( i,q,k,∑_j X(i,j,k)B(j,q) \) for each \( (i,j,k,q) ∈ id x(T_{i;k}) \).

In \( n \)-mode vector Hadamard product, the mappers send \( nnz(X_{j;i}) \) of \( j \)-th slice and an element of \( b_q \) to reducers using \( j \) as the key. The reducers multiply the vector element with the tensor elements. In Collapse operation, mappers send \( nnz(X_{i;k})Q \) elements to reducers using \( iK + k \) as the key. The reducers aggregate the values.

3) Removing Dependencies in Sequential Products : The previous two methods HATEN2-Naive and HATEN2-DNN
where $\bar{R} = 1 + \sum_{j=1}^{J} R_j$.

Our crucial observation is that these two operations can be used for removing the dependencies in the computations of $X \times_2 B^T \times_3 C^T$ and $X(1) (C \odot B)$, respectively, as shown in the following lemmas.

**Lemma 1 (CrossMerge):** Given $X \in \mathbb{R}^{I \times J \times K}$, $B \in \mathbb{R}^{J \times Q}$, and $C \in \mathbb{R}^{K \times R}$,

$$X \times_2 B^T \times_3 C^T \Leftrightarrow \text{CrossMerge}((\mathcal{T}', \mathcal{T}'')(1))$$

where $\mathcal{T}' \in \mathbb{R}^{I \times J \times K \times Q}$ is a tensor whose $q$th subtensor $\mathcal{T}'_{::q}$ is given by $X_{2:b_q} B_q^T$, and $\mathcal{T}'' \in \mathbb{R}^{I \times J \times K \times R}$ is a tensor whose $r$th subtensor $\mathcal{T}''_{::r}$ is given by $\text{bin}(X) \odot C_r^T$.

**Proof:** See the supplementary material [8].

**Lemma 2 (PairwiseMerge):** Given $X \in \mathbb{R}^{I \times J \times K}$, $B \in \mathbb{R}^{J \times R}$, and $C \in \mathbb{R}^{K \times R}$,

$$X(1) (C \odot B) \Leftrightarrow \text{PairwiseMerge}(\mathcal{T}', \mathcal{T}'')(1)$$

where $\mathcal{T}' \in \mathbb{R}^{I \times J \times K \times R}$ is a tensor whose $r$th subtensor $\mathcal{T}'_{::r}$ is given by $X_{2:b_r} B_r^T$, and $\mathcal{T}'' \in \mathbb{R}^{I \times J \times K \times R}$ is a tensor whose $r$th subtensor $\mathcal{T}''_{::r}$ is given by $\text{bin}(X) \odot C_r^T$.

**Proof:** See the supplementary material [8].

Using these two operations, HaTen2-Tucker-DRN, shown in Algorithm 8, decreases the number of MapReduce jobs of HaTen2-PARAFAC-DNN from $4R$ to $2R + 1$.

**Algorithm 7:** HaTen2-Tucker-DRN for computing $Y \leftarrow X \times_2 B^T \times_3 C^T$.

Input: Tensor $X \in \mathbb{R}^{I \times J \times K}$, and factor matrices $B \in \mathbb{R}^{J \times Q}$, $C \in \mathbb{R}^{K \times R}$.

Output: Tensor $Y \in \mathbb{R}^{I \times Q \times R}$.

1. for $q=1,...,Q$ do
2. $\mathcal{T}'_q \leftarrow X_{2:b_q} B_q^T$;
3. end for
4. for $r=1,...,R$ do
5. $\mathcal{T}''_r \leftarrow \text{bin}(X) C_r^T$;
6. end for
7. $Y \leftarrow \text{PairwiseMerge}(\mathcal{T}', \mathcal{T}'')(1)$;

**Algorithm 8:** HaTen2-PARAFAC-DNN for computing $Y \leftarrow X(1) (C \odot B)$.

Input: Tensor $X \in \mathbb{R}^{I \times J \times K}$, and factor matrices $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$.

Output: Tensor $Y \in \mathbb{R}^{I \times Q \times R}$.

1. for $r=1,...,R$ do
2. $\mathcal{T}'_r \leftarrow X_{2:b_r} B_r^T$;
3. end for
4. for $q=1,...,Q$ do
5. $\mathcal{T}''_q \leftarrow \text{bin}(X) C_q^T$;
6. end for
7. $Y \leftarrow \text{PairwiseMerge}(\mathcal{T}', \mathcal{T}'')(1)$;

**MAPREDUCE algorithm.** The MAPREDUCE algorithms of $\text{CrossMerge}(\mathcal{T}', \mathcal{T}'')(1)$ and $\text{PairwiseMerge}(\mathcal{T}', \mathcal{T}'')(1)$ operations are as follows.

- **MAP:** map $< i, j, k, q, X(i,j,k)b_q(j) >$ on $(i, rQ + q)$ for all $r = 1,...,R$, and $< i, j, k, r, c_r(k) >$ on $(i, rQ + q)$ for all $q = 1,...,Q$ such that tuples with the same key are shuffled to the same reducer in the form of $\text{key}: (i, r \ast Q + q)$, values: $(j, k, X(i,j,k)b_q(j))$ if $(i, j, k) \in \text{id}(X(1))$. Otherwise, emit $< i, j, k, \sum_{q=1}^{Q} X(i,j,k)b_q(j)c_r(k) >$ for all $q=1,...,Q$. Then, for each $(i, j, k) \in \text{id}(X(1))$.

- **REDUCE:** take $< \text{key}: (i, r \ast Q + q) >$ values: $(j, k, X(i,j,k)b_q(j))$ if $(i, j, k) \in \text{id}(X(1))$. Otherwise, emit $< i, j, k, \sum_{q=1}^{Q} X(i,j,k)b_q(j)c_r(k) >$ for all $q=1,...,Q$. Then, for each $(i, j, k) \in \text{id}(X(1))$.

- **MAP:** map $< i, j, k, r, X(i,j,k)b_r(j) >$ on $(i, r)$, and $< i, j, k, r, c_r(k) >$ on $(i, r)$ such that tuples with the same key are shuffled to the same reducer in the form of $\text{key}: (i, r \ast Q + q)$, values: $(j, k, X(i,j,k)b_r(j))$ if $(i, j, k) \in \text{id}(X(1))$. Otherwise, emit $< i, j, k, \sum_{r=1}^{R} X(i,j,k)b_r(j)c_r(k) >$ for all $r=1,...,R$. Then, for each $(i, j, k) \in \text{id}(X(1))$.

- **REDUCE:** take $< \text{key}: (i, r) >$ values: $(j, k, X(i,j,k)b_r(j))$ if $(i, j, k) \in \text{id}(X(1))$. Otherwise, emit $< i, j, k, \sum_{r=1}^{R} X(i,j,k)b_r(j)c_r(k) >$ for all $r=1,...,R$. Then, for each $(i, j, k) \in \text{id}(X(1))$. 

Fig. 6: Comparison of HaTen2-Tucker-DNN and HaTen2-Tucker-DRN.
4) Integrating Jobs by Increasing Memory Usage: Although HATEN2-DRN decreased the intermediate data size and the number of MAPREDUCE jobs, the number of jobs is still significant: it is \( Q + R + 1 \) for HATEN2-Tucker-DRN and \( 2R + 1 \) for HATEN2-PARAFAC-DRN. In this subsection, we propose HATEN2-DRI to further decrease the number of jobs to 2, thereby decreasing the disk accesses and the running time. Algorithms 9 and 10 show HATEN2-Tucker-DRI, and HATEN2-PARAFAC-DRI, respectively. HATEN2-DRI has two main ideas: integrating 1) vector products into a matrix product, and 2) products for different factor matrices.

Algorithm 9: HATEN2-Tucker-DRI for computing \( Y \leftarrow X \times_2 B^T \times_3 C^T \)

Input: Tensor \( X \in \mathbb{R}^{I \times J \times K} \), and factor matrices \( B \in \mathbb{R}^{J \times Q} \), \( C \in \mathbb{R}^{K \times R} \)

Output: Tensor \( Y \in \mathbb{R}^{I \times Q \times R} \)

1. \( (T', T'') \leftarrow \text{IMHP}(X, B, C) \);
2. \( Y \leftarrow \text{CrossMerge}(T', T'') \);

Algorithm 10: HATEN2-PARAFAC-DRI for computing \( Y \leftarrow X_{(1)} (C \circ B) \)

Input: Tensor \( X \in \mathbb{R}^{I \times J \times K} \), and factor matrices \( B \in \mathbb{R}^{J \times R} \), \( C \in \mathbb{R}^{K \times R} \)

Output: Tensor \( Y \in \mathbb{R}^{I \times R} \)

1. \( (T', T'') \leftarrow \text{IMHP}(X, B, C) \);
2. \( Y \leftarrow \text{PairwiseMerge}(T', T''(1)) \);

Integrating vector products into a matrix product.

HATEN2-DRI performs several \( n \)-mode vector Hadamard products together in one MAPREDUCE job, instead of multiple jobs, using the \( n \)-mode matrix Hadamard product which we define as follows.

Definition 5 (\( n \)-mode matrix Hadamard product): The \( n \)-mode matrix Hadamard product of a tensor \( X \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N} \) with a matrix \( U \in \mathbb{R}^{Q \times I_n} \) is denoted by \( X *_{\overline{U}} U \) and is of size \( I_1 \times I_2 \times \cdots \times I_{N-1} \times Q \). It is defined by

\[
(X *_{\overline{U}} U)_{i_1,i_2,\ldots,i_{N-1},q} = (X_{i_1,i_2,\ldots,i_{N-1}} U_{1,q})_{i_1,i_2,\ldots,i_{N-1}}.
\]

MAPREDUCE algorithm. The MAPREDUCE algorithm of an \( n \)-mode matrix Hadamard product is as follows:

\[
< X *_{\overline{2}} B >
\]

- MAP: map \( < i, j, k, X(i, j, k) > \) on \( j \), and \( < j, q, B(j, q) > \) on \( j \) such that tuples with the same key are shuffled to the same reducer in the form of

\[
\text{<key: } j, \text{ values: } \{(q, B(j, q)) | q < i, j, k > in idx(X_{j,j}) >}\.
\]

- REDUCE: take \( < k > \), values: \( \{(q, B(j, q)) | q < i, j, k > in idx(X_{j,j}) > \) and emit \( < i, j, k, q, X(i, j, k)B(j, q) > \) for each \( (i, j, k) in idx(X_{j,j}) \) and \( q in \{1, \ldots, Q\} \).

The previous HATEN2-DRN method performs the \( n \)-mode vector Hadamard product \( X *_{\overline{2}} B_q^T \) for \( Q \) times using \( nnz(X_{j,:}) + 1 \) of memory space per reducer; however, our new method HATEN2-DRI performs \( X *_{\overline{2}} B_q^T \) only once using \( nnz(X_{j,:}) + Q \) of memory space per reducer where \( Q \) is used for storing a column of \( B^T \). HATEN2-DRI decreases the number of jobs significantly without introducing too much overhead since \( Q \) is very small (e.g., \( 10 \) or \( 20 \)).

Integrating products for different factor matrices. HATEN2-DRI also integrates \( X *_{\overline{2}} B^T \) and \( bin(X) *_{\overline{3}} C^T \) computations into one MAPREDUCE job. This integration is possible since the dependency of the two operations is removed in HATEN2-DRN (and, hence in HATEN2-DRI). Thanks to the integration, the original tensor data \( X \) needs to be read from disks only once (not twice as in previous methods), and thus we further decrease the running time.

MAPREDUCE algorithm. The integrating operation, denoted by \( IMHP(X, B, C) \), is as follows.

\[
IMHP(X, B, C)
\]

- MAP: map \( < i, j, k, X(i, j, k) > on j, < j, q, B(j, q) on j, < i, j, k, X(i, j, k) > on k, and < k, r, C(k, r) > on k such that tuples with the same key are shuffled to the same reducer in the form of

\[
\text{<key: } j, \text{ values: } \{(q, B(j, q)) | q < i, j, k > in idx(X_{j,j}) >, and <key: } k, \text{ values: } \{(r, C(k, r)) | r < i, j, k > in idx(X_{j,k}) >.}
\]

- REDUCE: take <key: > j, values: \( \{(q, B(j, q)) | q < i, j, k > in idx(X_{j,j}) > \) and emit \( < i, j, k, q, X(i, j, k)B(j, q) > \) for each \( (i, j, k) in idx(X_{j,j}) \) and \( q in \{1, \ldots, Q\} \), and take

\[
\text{<key: } k, \text{ values: } \{(r, C(k, r)) | r < i, j, k > in idx(X_{j,k}) >, and emit < i, j, k, r, C(k, r) > for each (i, j, k) in idx(X_{j,k}) \) and \( r in \{1, \ldots, R\} \).
\]

Finally, we note that although the two decompositions Tucker and PARAFAC are different, our HATEN2-DRI unifies them in a general framework of IMHP and merge. As seen in Algorithms 9 and 10, as well as in Figure 4, HATEN2-Tucker-DRI and HATEN2-PARAFAC-DRI differ only in the merge function (CrossMerge for HATEN2-Tucker-DRI, and PairwiseMerge for HATEN2-PARAFAC-DRI). This general framework allows easier extension of the method for other algorithms, as well as simple maintenance of the code.

IV. Experiment

In this section, we present experimental results to answer the following questions.

Q1 What is the performance of HATEN2-DRI compared with other methods?

Q2 How well does HATEN2-DRI scale up with various factors (nonzeros, dimensionality, density, core tensor size, and machines)?

Q3 What are the discoveries on real world tensors?

Q4 What are the differences between PARAFAC and Tucker decompositions on real world tensors?

After describing the experimental settings in Section IV-A, we present the scalability results in Section IV-B to answer Q1 and Q2. Then we present the discovery results to answer Q3 and Q4.
A. Experimental Settings

We compare our final method HATEN2-DRI with other methods (HATEN2-Naive, HATEN2-DNN, HATEN2-DRN) as well as the Tensor Toolbox [13], the state of the art tensor computation package for single machine.

1) Machines: HATEN2 is run on a HADOOP cluster with 40 machines where each machine has a quad-core Intel Xeon E3 1230v3 3.3Ghz CPU, 32 GB RAM, and 12 Terabytes disk. The Tensor Toolbox is run on a machine from the HADOOP cluster.

2) Dataset: The tensor dataset used in our experiments are summarized in Table V, with the following details.

- Freebase-music: RDF dataset containing music-related (subject entity, object entity, relation) triples from Freebase [14].
- NELL: real world knowledge base data containing (noun phrase 1, noun phrase 2, context) triples (e.g. ‘George Harrison’, ‘guitars’, ‘plays’) from the ‘Read the Web’ project [1].
- Random: synthetic random tensor of size \( I \times I \times I \). The size \( I \) varies from \( 10^3 \) to \( 10^9 \), the number of nonzeros varies from \( 10^4 \) to \( 10^{10} \), and the density varies from \( 10^{-15} \) to \( 10^{-5} \).

B. Scalability

To answer the questions Q1 and Q2, we compare the machine and the data scalabilities of HATEN2-DRI with other methods.

1) Data Scalability: Data scalability is measured for the following three aspects: number of nonzeros and dimensionality, density, and core tensor size. We change the input tensor in terms of each aspect one by one, while fixing other aspects, and measure the running time using all the 40 machines in the cluster. Since the HATEN2-Naive method cannot process even a \( 10^4 \) scale tensor (Figures 1(a) and 7(a)), HATEN2-Naive is omitted from the density and core scalability experiments (Figures 1(b,c) and 7(b,c)).

   **Nonzeros and Dimensionality.** We increase the dimensionality \( I = J = K \) of modes from \( 10^3 \) to \( 10^6 \). The number of nonzeros is set to dimensionality \( \times 10 \). For Tucker decomposition, the size \( P \times Q \times R \) of the core tensor is fixed to \( 10 \times 10 \times 10 \). For PARAFAC decomposition, the rank \( R \) is set to 10. As shown in Figures 1(a) and 7(a), our best method HATEN2-DRI shows the best result: HATEN2-DRI analyzes \( 10^8 \) scale tensor most quickly. HATEN2-Naive and HATEN2-DNN failed for tensors with size beyond \( 10^3 \) and \( 10^7 \), respectively. Although HATEN2-DRN also analyzes \( 10^8 \) scale tensor, the running time is 1.3 times slower than that of HATEN2-DRI.

   **Density.** We increase the density of input tensor from \( 10^{-9} \) to \( 10^{-5} \); accordingly, the number of nonzeros becomes 1 billion to 1 trillion, and they take 20MB to 196GB disk space. The dimensionality of each mode is set to \( 10^5 \) (\( I = J = K \)). For Tucker decomposition, the size \( P \times Q \times R \) of the core tensor is fixed to \( 10 \times 10 \times 10 \). For PARAFAC decomposition, the rank \( R \) is set to 10. As shown in Figures 1(b) and 7(b), HATEN2-DRN decomposes 100× denser data than the Tensor Toolbox, and is the fastest among the variants of HATEN2.

   **Core Tensor Size.** We increase the core size of a random tensor of size \( 10^6 \times 10^6 \times 10^6 \) with \( 10^7 \) nonzeros, and measure the running time. For Tucker decomposition, the core tensor size increases from \( 10 \times 10 \times 10 \) to \( 80 \times 80 \times 80 \); for PARAFAC, the rank \( R \) increases from 10 to 80. As shown in Figures 1(c) and 7(c), HATEN2-DRI scales well, providing the best performance for all the core sizes. When the core size is 80, HATEN2-DRI outperforms the second best method

---

**TABLE V:** Summary of the tensor data used. B: billion, M: million, K: thousand.

<table>
<thead>
<tr>
<th>Data</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>Nonzeros</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freebase-music</td>
<td>23 M</td>
<td>23 M</td>
<td>166</td>
<td>99 M</td>
<td>( 1.12 \times 10^{-9} )</td>
</tr>
<tr>
<td>NELL</td>
<td>26 M</td>
<td>26 M</td>
<td>48 M</td>
<td>144 M</td>
<td>( 4.387 \times 10^{-15} )</td>
</tr>
<tr>
<td>Random</td>
<td>1 K</td>
<td>1~100 M</td>
<td>1 K~100 M</td>
<td>10 K~10 B</td>
<td>( 10^{-15} \sim 10^{-5} )</td>
</tr>
</tbody>
</table>

**Fig. 7:** Data scalability of our proposed HATEN2-DRI compared to other methods, for PARAFAC decomposition. The datasets are explained in detail at Section IV-A. o.o.m: out of memory. Compared to the Tensor Toolbox, HATEN2-DRI decomposes 10~100× larger data. Among the variants of HATEN2, HATEN2-DRI is the fastest, and analyzes 10× larger data than HATEN2-DNN.
we normalize the value by dividing it with the sum of all PARAFAC on it. For each element in an output factor matrix, we add the term's contribution to the core tensor. For example, for a tensor of size $26 \times 26 \times 26$, we use the core tensor data to alleviate the general term's domination [15]: we reweight the elements of the triples whose predicates appear only once in the data, as well as those that contain rare links. Finally, we reweight the elements of the relations corresponding to the least frequent predicate groups. To further mitigate the effects of dominant terms. Then, we choose top-$k$ highest valued elements from each column of the factors. Table VI shows the results from HATEN2-PARAFAC where we use rank 10. We see several concepts (e.g., “Classic Album”, “Pop/Rock Music”, and “Instrumentalist”) each of which contains groups of subjects, objects, and relations. Note that each subject group is tightly coupled with a relation group, due to the diagonal core tensor of PARAFAC. On the other hand, Tucker decomposition gives more diverse concepts from various groups. Table VII shows the factors from HATEN2-Tucker where we use the core size $10 \times 10 \times 10$. We found several groups for each mode: e.g., for the ‘subject’ mode, we found the groups “Classic Orchestra”, “Instrument”, and “Record Labels”. Table VIII shows the concepts each of which combines the groups from the subject, the object, and the relation factors. The first concept “Instrumental Pieces” contains the subject group S2 (“Instruments”), the object group O1 (“Classic Music”), and the relation group (“Instrumentalists”); the second concept “Classics” contains the subject group S1 (“Classic Composers and Performers”), the object group O1 (“Classic Music”), and the relation group (“Albums”). Note that the object group O1 appears in both of the concepts, exemplifying the Tucker’s ability to find concepts from various, possibly overlapping groups. Similarly, the second and the third concepts share the relation group R3 (“Albums”).

### V. RELATED WORK

#### A. CP/PARAFAC

1) **CP/PARAFAC at work**: Acar et al. [16] use the PARAFAC decomposition in order to detect epilepsy in brain measurements. In [2], Kolda and Bader extend the popular HITS algorithm for ranking web-pages, by incorporating anchor text information to the hyperlinks, and using PARAFAC in order to derive hubs and authorities from the data. PARAFAC has also been used in anomaly detection; [3] and [17] detect network anomalies in computer network connection logs and specifically [17] spots anomalies in time-evolving social networks as well. Last but not least, the PARAFAC decomposition has been used in community detection, where we have different views of the same network of people [18], or the network evolves over time and we are interested in identifying communities over time [19].

#### B. Tucker

1) **Tucker at work**: In [20], apart from a highly memory efficient Tucker decomposition algorithm, there is an overview of the various aspects of the Tucker decomposition as a data mining tool. The authors of [21] use a tensor in order to represent multiple semantic relations (such as “synonym” or “antonym”) and use Tucker as a higher order generalization of SVD, in order to perform Latent Semantic Analysis. One of the most widely used Tucker variation is the so called Higher Order Singular Value Decomposition (HOSVD) [22] which is a Tucker3 model with additional orthonormality constraints on the factor matrices. An exemplary work of employing HOSVD...
TABLE VI: Concept discovery result from HATEN2-PARAFAC on Freebase-music dataset.

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Subject Entity</th>
<th>Object Entity</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compilation Album</td>
<td>Symphony No. 7 in E minor: IVa. Nachtmusik II: Andante amoroso</td>
<td>ns:music.artist.track</td>
</tr>
<tr>
<td></td>
<td>Live Album</td>
<td>Fantasie a quatre mains sur Don Juan, op. 26: V. Variation 3</td>
<td>ns:music.performance-role.track-performances</td>
</tr>
<tr>
<td>Concept2: “Pop/Rock music”</td>
<td>Rock music</td>
<td>Our Album! (Pop album)</td>
<td>ns:music.album-release-type.albums</td>
</tr>
<tr>
<td></td>
<td>Pop music</td>
<td>Plastic Parachute (Rock band)</td>
<td>ns:music.genre.albums</td>
</tr>
<tr>
<td></td>
<td>Alternative rock</td>
<td>Since the Accident (Rock album)</td>
<td>ns:music.voice.singers</td>
</tr>
<tr>
<td>Concept3: “Instrumentalist”</td>
<td>Guitar</td>
<td>Klezmator (Violinist)</td>
<td>ns:music.performance-role.regular-performances</td>
</tr>
<tr>
<td></td>
<td>Cor anglais</td>
<td>George Baquet (Jazz clarinetist)</td>
<td>ns:music.instrument.instrumentalists</td>
</tr>
<tr>
<td></td>
<td>Flute</td>
<td>Manuel Staropoli (Recorder player)</td>
<td>ns:music.genre.artists</td>
</tr>
</tbody>
</table>

TABLE VII: Discovered factors from HATEN2-Tucker on Freebase-music dataset.

<table>
<thead>
<tr>
<th>Subject S1: Classic Composers and Performers</th>
<th>Subject S2: Instruments</th>
<th>Subject S3: Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>London Symphony Orchestra</td>
<td>Guitar</td>
<td>EMI</td>
</tr>
<tr>
<td>Wolfgang Amadeus Mozart</td>
<td>Keyboard</td>
<td>Atlantic Records</td>
</tr>
<tr>
<td>Ludwig van Beethoven</td>
<td>Drums</td>
<td>Universal Music Group</td>
</tr>
<tr>
<td>New York Philharmonic</td>
<td>Bass guitar</td>
<td>Warner Bros. Records</td>
</tr>
</tbody>
</table>

TABLE VIII: Concept discovery result from HATEN2-Tucker on Freebase-music dataset.

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Subject Entity</th>
<th>Object Entity</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept1: “Instrumental Pieces”</td>
<td>Guitar</td>
<td>Faust: Soldatenchor</td>
<td>'ns:music.instrument.instrumentalists'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 Trios for Flute, Violin, Cello, no. 1: IV, Rondo: Allegro</td>
<td>'ns:music.instrument.variation'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Piano Concerto in A minor, op. 54: III. Allegro vivace</td>
<td>'ns:music.album.artist'</td>
</tr>
<tr>
<td>Concept2: “Classics”</td>
<td>London Symphony Orchestra</td>
<td>Faust: Soldatenchor</td>
<td>'ns:music.release.region'</td>
</tr>
<tr>
<td></td>
<td>Wolfgang Amadeus Mozart</td>
<td>3 Trios for Flute, Violin, Cello, no. 1: IV, Rondo: Allegro</td>
<td>'ns:music.record-label.artist'</td>
</tr>
<tr>
<td></td>
<td>Ludwig van Beethoven</td>
<td>Piano Concerto in A minor, op. 54: III. Allegro vivace</td>
<td>'ns:music.album.artist'</td>
</tr>
<tr>
<td>Concept3: “Musics from Labels”</td>
<td>Columbia</td>
<td>Sikdim (by Warner Music Group)</td>
<td>'ns:music.release.region'</td>
</tr>
<tr>
<td></td>
<td>EMI</td>
<td>Terrifying Tales (by EMI)</td>
<td>'ns:record-label.artist'</td>
</tr>
<tr>
<td></td>
<td>Atlantic Records</td>
<td>Rose of Tralee (by EMI)</td>
<td>'ns:music.album.artists'</td>
</tr>
</tbody>
</table>

is [23] where the authors provide web search recommendations to users. HOSVD has been extensively used in Computer Vision applications [24][25].

C. Scalable Algorithms for Tensor Analysis

The pioneering work [26] of Bader and Kolda develop efficient algorithms for sparse tensors, where they avoid the materialization of very large, unnecessary intermediate Khatri-Rao products. Kang et al. proposed GigaTensor [27], [28] that first uses a distributed system for PARAFAC decomposition. GigaTensor is similar to HATEN2-PARAFAC-DRN in this paper, however in our work we provide a significant improvement upon [27]. It can be shown that the ways that GigaTensor [27] and [26] avoid the intermediate data explosion are equivalent, however, GigaTensor [27] provides an algorithm which is optimized for the distributed setting. In [29] Beutel et al. propose FlexiFaCT, a MAPREDUCE algorithm based on Distributed Stochastic Gradient Descent for PARAFAC and coupled PARAFAC decompositions. In [30], Bro and Sidiropoulos use Tucker to compress a tensor, then do the PARAFAC decomposition on the compressed tensor and finally decompress the factors, thus speeding up the PARAFAC decomposition. An alternative approach, DBN, is introduced in [31] where the authors use Relational Algebra to break down the tensor into smaller tensors, using relational decomposition, and thus achieving scalability. Furthermore, [17], introduces ParCube, an approximate and highly parallelizable algorithm for sparse PARAFAC decomposition. For scalable Tucker decomposition, there exists several previous works. Kolda and Sun [20] propose MET (Memory-Efficient Tucker) for scalable Tucker decomposition algorithm running on Matlab. Finally, Erdos and Miettinen introduce a scalable boolean tensor decomposition using random walks [32].

VI. Conclusion

In this paper we propose HATEN2, a scalable method for distributed tensor decompositions on MAPREDUCE. HATEN2 unifies the two major tensor decomposition algorithms, Tucker and PARAFAC, into a general framework such that the intermediate data size and the running times are minimized. Due to
the careful design, HATEN2 decomposes $100 \times$ larger tensors compared to existing methods. Furthermore, HATEN2 enjoys near linear scalability on the number of machines. Applying HATEN2 on a knowledge base tensor with 23 millions of entities and 99 millions of facts, we discover interesting concepts of entities and relations.

Future research directions include extending our framework to other settings such as tensor decompositions with missing values or nonnegative tensor decompositions.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) Grant funded by the Korean Government(MSIP)(No. 2013R1A1A1064409), and by the National Science Foundation under Grant No. IIS-1247489. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties.

REFERENCES