학사 학위논문

Bachelor's Thesis

Scalable Methods for Random Walk with Restart and Tensor Factorization

신 기 정 (辛 基 禎 Shin, Kijung)

컴퓨터공학부

Department of Computer Science and Engineering

Seoul National University

2015

Scalable Methods for Random Walk with Restart and Tensor Factorization

ABSTRACT

"Big data" has received considerable interests from both academia and industry in the last decade. It turned out that mining large-scale data enables us to obtain machine learning models with higher accuracy and extend our knowledge about large complex systems such as Web and social media. However, the enormous volume of data prevents us from simply using previous machine learning or data mining algorithms and requires new approaches.

One approach for tackling big data is to exploit the structural properties of data. As the first part of this thesis, we utilize hub-and-spoke structure of real-world graphs to design scalable computation methods for random walk with restart (RWR), a widely-used measure of the relevance between two nodes in a graph. The hub-and-spoke structure enables us to divide a given RWR computation problem into smaller problems, and our methods significantly outperform state-of-the-art methods in terms of speed, space-efficiency, and accuracy.

Another approach is to exploit parallelism. As the second part of this thesis, we design tensor factorization methods for collaborative filtering that run in parallel in distributed environments. Owing to their high scalability with regard to all aspects of data (the dimension, mode length, and rank of tensors), our methods successfully solved tensor factorization with 50 billion parameters, while all other methods failed. We also discuss the limitations of MapReduce, the most widely-used programming model for distributed computing, we experienced during the development of the methods.

These limitations of previous programming models led to the last part of this thesis, the design of a programming model for distributed machine learning and data mining. Our multi-stage BSP model supports not only iteration and rich communication methods but also the retainment of computing resources and intermediate data. We show the expressiveness and convenience of our model through distributed machine learning examples and also discuss its implementation on a distributed system.

The algorithms and programming model proposed in this thesis will be available as parts of open-source projects, PEGASUS and Apache REEF, respectively.

Contents

Chapter 1. Introduction

During the last decade, "Big data" has sparked the considerable interests of academia, industry, and even governments. In machine learning, using large volumes of data is the simplest way to achieve high accuracy overcoming the bias-variance trade-off. It is also expected that we will be able to extend our knowledge about large complex systems, such as Web and social media, by mining large-scale data. However, it also has turned out that taming "Big data" involves significant challenges. One of the main challenges is that previous machine learning and data mining methods are hardly applicable to large-scale data due to their limited scalability. This thesis consists of three parts all of which are closely related to this challenge, which requires us to explore new approaches.

One major approach is to utilize the structural properties of real-world data. Especially, the structural properties real-world graphs, including the power-law degree distribution [28] and small diameter [44], have been widely explored and taken into consideration when designing algorithms and systems [41, 69, 73]. In Chapter 2, we utilize hub-and-spoke structure of real-world graphs to design scalable computation methods for random walk with restart (RWR) [80], a widely-used measure of the relevance between two nodes in a graph. The fact that real-world graphs are divided into smaller pieces by removing small number of hub nodes [5] enables us to divide a given RWR computation problem into smaller problems. We show that our methods significantly outperform state-of-the-art methods in terms of speed, space-efficiency, and accuracy.

Another widely explored approach is to exploit parallelism. Distributed machine learning algorithms [60, 39, 74], graph mining algorithms [65, 44] and systems [4, 45, 56] have been developed. In Chapter 3, we propose tensor factorization methods for context-aware collaborative filtering that run in parallel in distributed environments. Owing to their high scalability with regard to all aspects of data (the dimension, mode length, and rank of tensors) as well as machine scalability, our methods successfully analyzed a 5-dimensional tensor with 1 billion observable entries, 10M mode length, and 1K rank, while all other methods failed. We implemented our methods using MapReduce [24], the most widely-used programming model for distributed computing. However, the experience illustrated the limitations of the model in terms of iteration and communication methods.

These limitations of the MapReduce programming model are recognized repeatedly [4, 85, 77], and partially solved by recent systems [17, 27, 56]. However, room for improvements motivated us to design a new programming model for distributed machine learning and data mining described in Chapter 4. Our multi-stage BSP model supports not only iteration and rich communication methods but also the retainment of computing resources and intermediate data among parallel jobs. We show the expressiveness and convenience of our model through distributed machine learning examples and also discuss its implementation on a distributed system.

The binary codes and datasets used in Chapter 2 and Chapter 3 are available at `http://koreaskj.snucse.org/programs/`, and the source codes will be available as the part of the open-source PEGASUS project (`http://www.cs.cmu.edu/~pegasus/`). The proposed programming model and its runtime are being developed as a part of the Apache REEF project (`https://reef.incubator.apache.org/`).

Chapter 2. Block Elimination Approach for Random Walk with Restart on Large Graphs

Given a large graph, how can we calculate the relevance between nodes fast and accurately? Random walk with restart (RWR) provides a good measure for this purpose and has been applied to diverse data mining applications including ranking, community detection, link prediction, and anomaly detection. Since calculating RWR from scratch takes long, various preprocessing methods, most of which are related to inverting adjacency matrices, have been proposed to speed up the calculation. However, these methods do not scale to large graphs because they usually produce large and dense matrices which do not fit into memory.

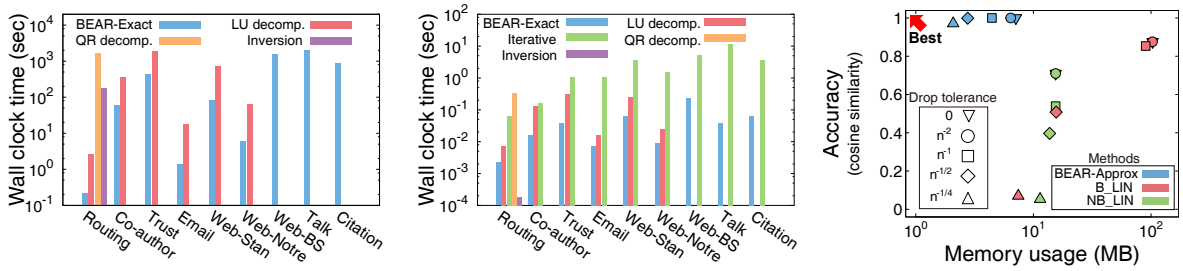
In this chapter, we propose BEAR, a fast, scalable, and accurate method for computing RWR on large graphs. BEAR comprises the preprocessing step and the query step. In the preprocessing step, BEAR reorders the adjacency matrix of a given graph so that it contains a large and easy-to-invert submatrix, and precomputes several matrices including the Schur complement of the submatrix. In the query step, BEAR computes the RWR scores for a given query node quickly using a block elimination approach with the matrices computed in the preprocessing step. Through extensive experiments, we show that BEAR significantly outperforms other state-of-the-art methods in terms of preprocessing and query speed, space efficiency, and accuracy.

2.1 Introduction

Measuring the relevance (proximity) between nodes in a graph becomes the base for various data mining tasks [7, 11, 33, 36, 52, 76, 80, 79, 83, 91] and has received much interest from the database research community [9, 20, 29, 84]. Among many methods [3, 37, 54, 64] to compute the relevance, random walk with restart (RWR) [64] has been popular due to its ability to account for the global network structure [36] and the multi-faceted relationship between nodes [79]. RWR has been used in many data mining applications including ranking [81], community detection [7, 33, 83, 91], link prediction [11], and anomaly detection [76].

However, existing methods for computing RWR are unsatisfactory in terms of speed, accuracy, functionality, or scalability. The iterative method, which naturally follows from the definition of RWR is not fast: it requires repeated matrix-vector multiplications whose computational cost is not acceptable in real world applications where RWR scores for different query nodes need to be computed. Several approximate methods [7, 32, 76, 81] have also been proposed; however, their accuracies or speed-ups are unsatisfactory. Although top-k methods [29, 84] improve efficiency by focusing on finding the k most relevant nodes and ignoring irrelevant ones, finding top-k is insufficient for many data mining applications [7, 11, 32, 36, 76, 80, 83, 91] which require the relevance scores of all nodes or the least relevant nodes. Existing preprocessing methods [29, 30], which achieve better performance by preprocessing the given graph, are not scalable due to their high memory requirements.

In this chapter, we propose BEAR, a fast, scalable, and accurate method for computing RWR on large graphs. BEAR comprises two steps: the preprocessing step and the query step. In the preprocessing



(a) Preprocessing time of exact methods

(b) Query time of exact methods

(c) Space for preprocessed data vs. Accuracy in approximate methods

Figure 2.1: Performance of BEAR. (a) and (b) compare exact methods, while (c) compares approximate methods. In (a) and (b), bars are omitted if and only if the corresponding experiments run out of memory. (a) In the preprocessing phase, BEAR-Exact is the fastest and the most scalable among all preprocessing methods. (b) BEAR-Exact is the fastest in the query phase except for the smallest Routing dataset. Due to its up to 300× faster query speed, BEAR-Exact outperforms the iterative method, despite the preprocessing cost, when RWR scores for different query nodes are required. (c) BEAR-Approx achieves both space efficiency and accuracy. Details of these experiments are explained in Section 2.4.

step, BEAR reorders the adjacency matrix of a given graph so that it contains a large and easy-to-invert submatrix, and precomputes several matrices including the Schur complement [16] of the submatrix. In the query step, BEAR computes the RWR scores for a given query node quickly using a block elimination approach with the matrices computed in the preprocessing step. BEAR has two versions: an exact method BEAR-Exact and an approximate method BEAR-Approx. The former provides accuracy assurance; the latter gives faster query speed and requires less space by allowing small accuracy loss.

Through extensive experiments with various real-world datasets, we demonstrate the superiority of BEAR over other state-of-the-art methods as seen in Figure 2.1. We also discuss how our method can be applied to other random-walk-based measures such as personalized PageRank [63], effective importance [15], and RWR with normalized graph Laplacian [81]. The main characteristics of our method are the followings:

- **Fast.** BEAR-Exact is faster up to 8× in the query phase and up to 12× in the preprocessing phase than other exact methods (Figures 2.1(a) and 2.1(b)); BEAR-Approx achieves a better time/accuracy trade-off in the query phase than other approximate methods (Figure 2.8(a)).
- **Space-efficient.** Compared with their respective competitors, BEAR-Exact requires up to 22× less memory space (Figure 2.5), and BEAR-Approx provides a better space/accuracy trade-off (Figure 2.1(c)).
- **Accurate.** BEAR-Exact guarantees exactness (Theorem 1); BEAR-Approx enjoys a better trade-off between accuracy, time, and space than other approximate methods (Figure 2.8).
- **Versatile.** BEAR can be applied to diverse RWR variants, including personalized PageRank, effective importance, and RWR with normalized graph Laplacian (Section 2.3.4).

The binary code of our method and several datasets are available at `http://koreaskj.snucse.org/programs/`. The rest of the chapter is organized as follows. Section 2.2 presents preliminaries on RWR. Our proposed BEAR is described in Section 2.3 followed by experimental results in Section 2.4. After providing a review on related work in Section 2.5, we make conclusion in Section 2.6.

Table 2.1: Table of symbols.

Symbol	Definition
G	input graph
n	number of nodes in G
m	number of edges in G
n_1	number of spokes in G
n_2	number of hubs in G
n_{1i}	number of nodes in the ith diagonal block of $\mathbf{H_{11}}$
b	number of diagonal blocks in $\mathbf{H_{11}}$
s	seed node (=query node)
c	restart probability
ξ	drop tolerance
$\tilde{\mathbf{A}}$	$(n \times n)$ row-normalized adjacency matrix of G
$\mathbf{H}$	$(n \times n)$ $\mathbf{H} = \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}$
$\mathbf{H_{ij}}$	$(n_i \times n_j)$ (i,j)th partition of $\mathbf{H}$
$\mathbf{S}$	$(n_2 \times n_2)$ Schur complement of $\mathbf{H_{11}}$
$\mathbf{L_1}, \mathbf{U_1}$	$(n_1 \times n_1)$ LU-decomposed matrices of $\mathbf{H_{11}}$
$\mathbf{L_2}, \mathbf{U_2}$	$(n_2 \times n_2)$ LU-decomposed matrices of $\mathbf{S}$
$\mathbf{q}$	$(n \times 1)$ starting vector
$\mathbf{q_i}$	$(n_i \times 1)$ ith partition of $\mathbf{q}$
$\mathbf{r}$	$(n \times 1)$ relevance vector
$\mathbf{r_i}$	$(n_i \times 1)$ ith partition of $\mathbf{r}$
T	number of SlashBurn iterations
k	number of hubs removed at a time in SlashBurn
t	rank in B_LIN and NB_LIN
ϵ	threshold to stop iteration in iterative methods
ϵ_b	threshold to expand nodes in RPPR and BRPPR

2.2 Preliminaries

In this section, we describe the preliminaries on random walk with restart and its algorithms. Table 2.1 lists the symbols used in this chapter. We denote matrices by boldface capitals, e.g., $\mathbf{A}$, and vectors by boldface lowercases, e.g., $\mathbf{q}$.

2.2.1 Random Walk with Restart

Random walk with restart (RWR) [81] measures each node's relevance w.r.t. a given seed node s in a given graph. It assumes a random surfer who occasionally gets bored with following the edges in the graph and restarts at node s. The surfer starts at node s, and at each current node, either restarts at node s (with probability c) or moves to a neighboring node along an edge (with probability $1-c$). The probability that each edge is chosen is proportional to its weight in the adjacency matrix. $\tilde{\mathbf{A}}$ denotes the row-normalized adjacency matrix, whose (u,v)th entry is the probability that a surfer at node u chooses the edge to node v among all edges from node u. The stationary probability of being at each

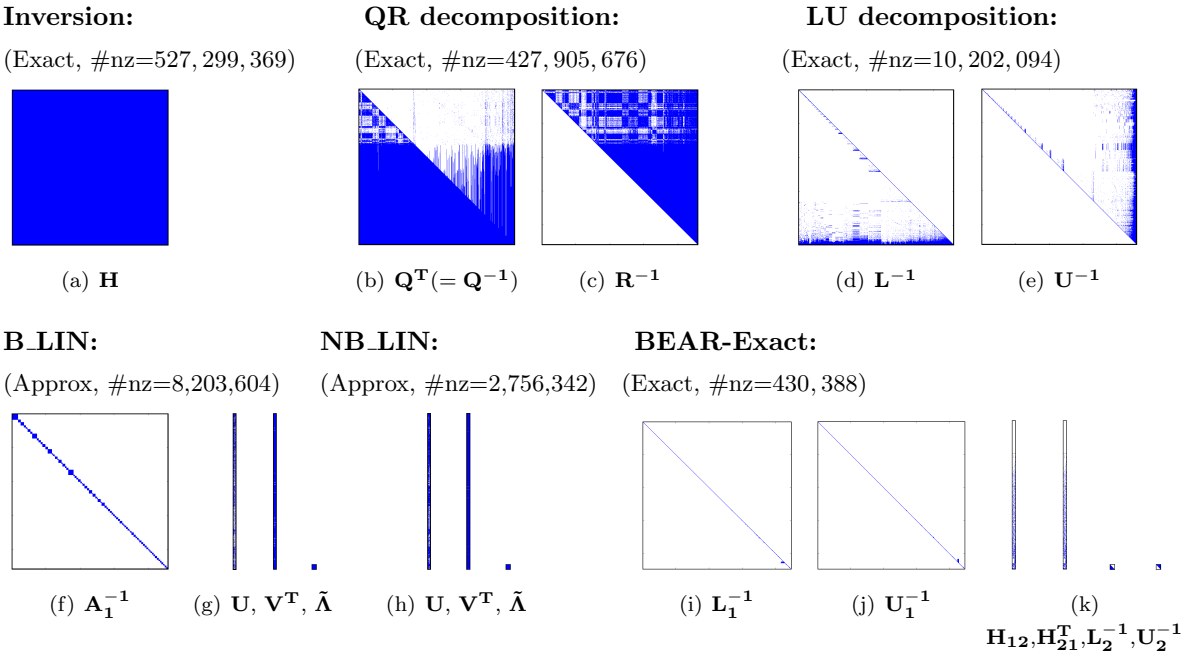


Figure 2.2: Sparsity patterns of the matrices resulted from different preprocessing methods on the Routing dataset (see Section 2.4.1 for the details of the dataset). Exact: exact method, Approx: approximate method, #nz: number of non-zero elements in the precomputed matrices. For B_LIN and NB_LIN, rank t is set to 500 and drop tolerance ξ is set to 0. Precomputed matrices are loaded into memory to speed up the query phase; for the reason, the number of non-zero entries in them determines the memory usage of each method and thus its scalability. Our exact method BEAR-Exact produces the smallest number of non-zero entries than all other methods including the approximate methods (1200× than Inversion and 6× than the second best method). Our approximate method BEAR-Approx further decreases the number of non-zero entries, depending on drop tolerance ξ.

node corresponds to its RWR score w.r.t. node s, and is denoted by $\mathbf{r}$, whose uth entry corresponds to node u's RWR score. The vector $\mathbf{r}$ satisfies the following equation:

$$\mathbf{r} = (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}\mathbf{r} + c\mathbf{q} \tag{2.1}$$

where $\mathbf{q}$ is the starting vector with the index of the seed node s is set to 1 and others to 0. It can be obtained by solving the following linear equation:

$$\begin{aligned}&(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})\mathbf{r} = c\mathbf{q}\\ &\Leftrightarrow \mathbf{H}\mathbf{r} = c\mathbf{q}.\end{aligned} \tag{2.2}$$

Personalized PageRank. Personalized PageRank (PPR) [63] is an extension of RWR. PPR calculates the relevance of nodes according to the preference of each user, and is widely used for personalized search. A random surfer in PPR either jumps to a random node according to the probability distribution (user preference distribution) given by $\mathbf{q}$ (with probability c) or moves to a neighboring node (with probability $1-c$). PPR can be viewed as a generalized version of RWR with multiple seed nodes. Equations (2.1) and (2.2) can be directly applied to PPR with the modified $\mathbf{q}$.

2.2.2 Algorithms for RWR

We review two basic methods for RWR computation and four recent methods for addressing the limitations of the basic methods. We also point out a need for improvement which we will address in the following section. Since most applications require RWR scores for different seed nodes, whether at once or on demand, we separate the preprocessing phase, which occurs once, from the query phase, which occurs for each seed node.

Iterative method. The iterative method repeats updating $\mathbf{r}$ until convergence ($|\mathbf{r}^{(i)} - \mathbf{r}^{(i-1)}| < \epsilon$) by the following update rule:

$$\mathbf{r}^{(i)} \leftarrow (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}\mathbf{r}^{(i-1)} + c\mathbf{q} \tag{2.3}$$

where the superscript i denotes the iteration number. If $0 < c < 1$, $\mathbf{r}^{(i)}$ is guaranteed to converge to a unique solution [50]. This method does not require preprocessing (one-time cost) but has expensive query cost which incurs a repeated matrix-vector multiplication. Thus, it is inefficient when RWR scores for many query nodes are required.

RPPR / BRPPR. Gleich et al. [32] propose restricted personalized PageRank (RPPR), which speeds up the iterative method by accessing only a part of a graph. This algorithm uses Equation 2.3 only for a subgraph and the nodes contained in it. The subgraph is initialized to a given seed node and grows as iteration proceeds. A node contained in the subgraph is on the boundary if its outgoing edges (in the original graph) are not contained in the subgraph, and the outgoing edges and outgoing neighbors of the node are added to the subgraph if the RWR score of the node (in the current iteration) is greater than a threshold ϵ_b. The algorithm repeats iterations until RWR scores converge. The RWR scores of nodes outside the subgraph are set to zero. Boundary-restricted personalized PageRank (BRPPR) [32] is a variant of the RPPR. It expands nodes on the boundary in decreasing order of their RWR scores (in the current iteration) until the sum of the RWR scores of nodes on the boundary becomes less than a threshold ϵ_b. Although these methods reduce the query cost of the iterative method significantly, they do not guarantee exactness.

Inversion. An algebraic method directly calculates $\mathbf{r}$ from Equation (2.2) as follows:

$$\mathbf{r} = c(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})^{-1}\mathbf{q} = c\mathbf{H}^{-\mathbf{1}}\mathbf{q}. \tag{2.4}$$

The matrix $\mathbf{H}$ is known to be invertible when $0 < c < 1$ [50]. Once $\mathbf{H}^{-\mathbf{1}}$ is computed in the preprocessing phase, $\mathbf{r}$ can be obtained efficiently in the query phase. However, this is again impractical for large graphs because calculating $\mathbf{H}^{-\mathbf{1}}$ is computationally expensive and $\mathbf{H}^{-\mathbf{1}}$ is usually too dense to fit into memory as seen in Figure 2.2(a).

QR decomposition. To avoid the problem regarding $\mathbf{H}^{-\mathbf{1}}$, Fujiwara et al. [30] decompose $\mathbf{H}$ using QR decomposition, and then use $\mathbf{Q}^{\mathbf{T}}(= \mathbf{Q}^{-\mathbf{1}})$ and $\mathbf{R}^{-\mathbf{1}}$ instead of $\mathbf{H}^{-\mathbf{1}}$ as follows:

$$\mathbf{r} = c\mathbf{H}^{-\mathbf{1}}\mathbf{q} = c\mathbf{R}^{-\mathbf{1}}(\mathbf{Q}^{\mathbf{T}}\mathbf{q})$$

where $\mathbf{H} = \mathbf{QR}$. They also propose a reordering rule for $\mathbf{H}$ which makes $\mathbf{Q}^{\mathbf{T}}$ and $\mathbf{R}^{-\mathbf{1}}$ sparser. However, on the most datasets used in our experiments, QR decomposition results in dense matrices as shown in Figures 2.2(b) and 2.2(c); thus, its scalability is limited. This fact agrees with the claim made by Boyd et al. [16] that it is difficult to exploit sparsity in QR decomposition.

LU decomposition. To replace $\mathbf{H}^{-\mathbf{1}}$, Fujiwara et al. [29] also exploit LU decomposition using the following rule:

$$\mathbf{r} = c\mathbf{H}^{-\mathbf{1}}\mathbf{q} = c\mathbf{U}^{-\mathbf{1}}(\mathbf{L}^{-\mathbf{1}}\mathbf{q})$$

where $\mathbf{H} = \mathbf{LU}$. Prior to the decomposition, $\mathbf{H}$ is reordered based on nodes' degrees and community structure. This makes matrices $\mathbf{L^{-1}}$ and $\mathbf{U^{-1}}$ sparser as seen in Figures 2.2(d) and 2.2(e). We incorporate their idea into our method to replace inverse terms, which will be explained in detail in Section 2.3.

B_LIN / NB_LIN. Tong et al. [81] partition a given graph and divide $\tilde{\mathbf{A}}^{\mathbf{T}}$ into $\mathbf{A_1}$ (inner-partition edges) and $\mathbf{A_2}$ (cross-partition edges). Then, they use a heuristic decomposition method with given rank t to approximate $\mathbf{A_2}$ with low-rank matrix $\mathbf{U\Sigma V}$ where $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ are $n \times t$, $t \times t$, and $t \times n$ matrices, respectively. In the query phase, they apply Sherman-Morrison lemma [66] to efficiently calculate $\mathbf{r}$ as follows:

$$\begin{aligned}\mathbf{r} &= c(\mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})^{-1}\mathbf{q} \\ &\approx c(\mathbf{I} - (1-c)\mathbf{A_1} - (1-c)\mathbf{U\Sigma V})^{-1}\mathbf{q} \\ &= c(\mathbf{A_1^{-1}q} + (1-c)\mathbf{A_1^{-1}U\tilde{\Lambda}VA_1^{-1}q})\end{aligned}$$

where $\tilde{\mathbf{\Lambda}} = (\mathbf{\Sigma}^{-1} - c\mathbf{VA_1^{-1}U})^{-1}$. To sparsify the precomputed matrices, near-zero entries whose absolute value is smaller than ξ are dropped. This method is called B_LIN, and its variant NB_LIN directly approximates $\tilde{\mathbf{A}}^{\mathbf{T}}$ without partitioning it. Both methods do not guarantee exactness.

As summarized in Figure 2.2, previous preprocessing methods require too much space for preprocessed data or do not guarantee accuracy. Our proposed BEAR, explained in the following section, achieves both space efficiency and accuracy as seen in Figures 2.2(i) through 2.2(k).

2.3 Proposed Method

In this section, we describe BEAR, our proposed method for fast, scalable, and accurate RWR computation. BEAR has two versions: BEAR-Exact for exact RWR, and BEAR-Approx for approximate RWR. BEAR-Exact guarantees accuracy, while BEAR-Approx improves speed and space efficiency by sacrificing little accuracy. A pictorial description of BEAR is provided in Figure 2.3. BEAR exploits the following ideas:

- The adjacency matrix of real-world graphs can be reordered so that it has a large but easy-to-invert submatrix, such as a block-diagonal matrix as shown in the upper left part of Figure 2.4(b).
- A linear equation like Equation (2.2) is easily solved by block elimination using Schur complement if the matrix contains a large and easy-to-invert submatrix such as a block-diagonal one.
- Compared with directly inverting an adjacency matrix, inverting its LU-decomposed matrices is more efficient in terms of time and space.

Algorithms 1 and 2 represent the procedure of BEAR. Since RWR scores are requested for different seed nodes in real world applications, we separate the preprocessing phase (Algorithm 1), which is run once, from the query phase (Algorithm 2), which is run for each seed node. The exact method BEAR-Exact and the approximate method BEAR-Approx differ only at line 9 of Algorithm 1; the detail is in Section 2.3.1. To exploit their sparsity, all matrices considered are saved in a sparse matrix format, such as the compressed sparse column format [67], which stores only non-zero entries.

2.3.1 Preprocessing Phase

The overall preprocessing phase of BEAR is shown in Algorithm 1 and the details in the following subsections.

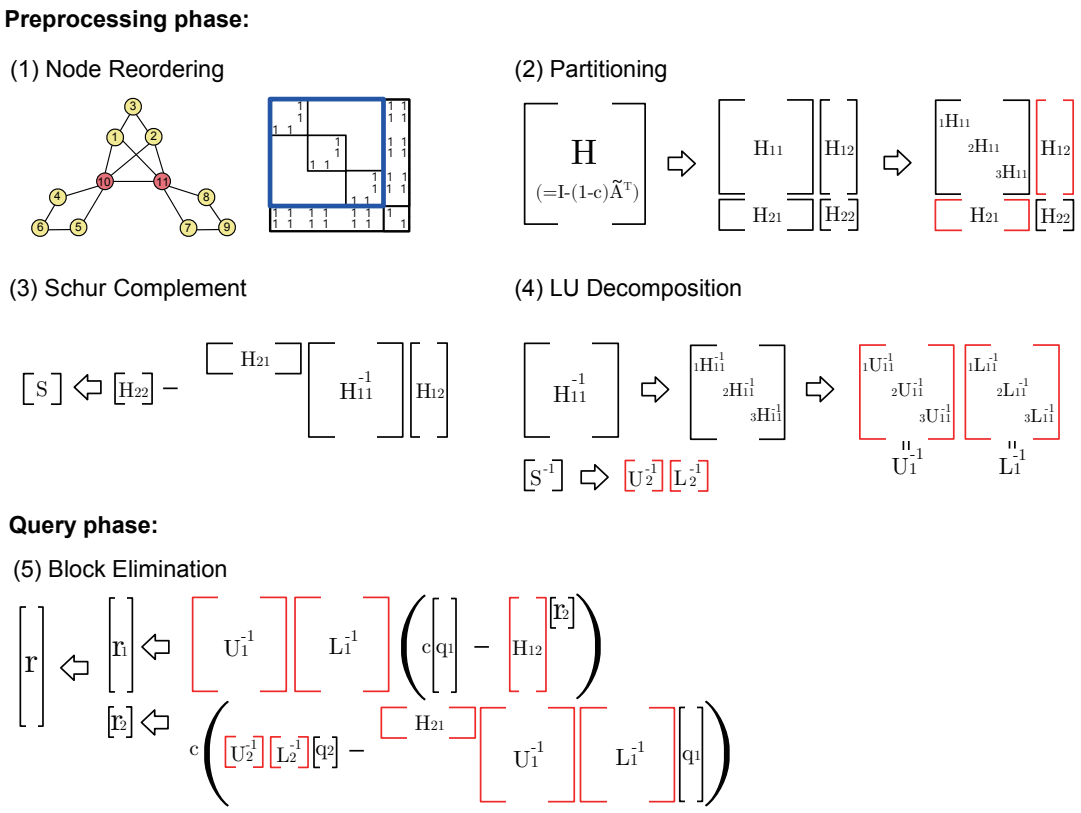


Figure 2.3: A pictorial description of BEAR. The output matrices of the preprocessing phase are red-bordered. (1) Reorder nodes so that the adjacency matrix has a large block-diagonal submatrix (the blue-bordered one). (2) Partition $\mathbf{H}$ into four blocks so that $\mathbf{H_{11}}$ corresponds to the submatrix. (3) Compute the Schur complement $\mathbf{S}$ of $\mathbf{H_{11}}$. (4) Since $\mathbf{S^{-1}}$ and the diagonal blocks of $\mathbf{H_{11}^{-1}}$ are likely to be dense, store the inverse of the LU decomposed matrices of $\mathbf{S}$ and $\mathbf{H_{11}}$ instead to save space. Notice that these can be computed efficiently in terms of time and space because $\mathbf{S}$ and the diagonal blocks of $\mathbf{H_{11}}$ are relatively small compared with $\mathbf{H}$. (5) Compute the RWR score vector $\mathbf{r}$ for a query vector $\mathbf{q}$ (the concatenation of $\mathbf{q_1}$ and $\mathbf{q_2}$) fast by utilizing the precomputed matrices.

Node Reordering (lines 2 through 4)

In this part, we reorder $\mathbf{H}(= \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}})$ and partition it. Our objective is to reorder $\mathbf{H}$ so that it has a large but easy-to-invert submatrix such as a block-diagonal one. Any node reordering method (e.g., spectral clustering [61], cross association [18], shingle [22], etc.) can be used for this purpose; in this paper, we use a method which improves on SlashBurn [41] since it is the state-of-the-art method in concentrating the nonzeros of adjacency matrices of graphs (more details in Appendix A.1). We first run SlashBurn on a given graph, to decompose the graph into hubs (high-degree nodes), and spokes (low-degree nodes which get disconnected from the giant connected component if the hubs are removed). Within each connected component containing spokes, we reorder nodes in the ascending order of degrees within the component. As a result, we get an adjacency matrix whose upper-left area (e.g., $\mathbf{H_{11}}$ in Figure 2.4(a)) is a large and sparse block diagonal matrix which is easily inverted, while the lower-right area (e.g., $\mathbf{H_{22}}$ in Figure 2.4(a)) is a small but dense matrix. Let n_1 denote the number of spokes and n_2 denote the number of hubs. After the reordering, we partition the matrix $\mathbf{H}$ into four pieces: $\mathbf{H_{11}}$ ($n_1 \times n_1$ matrix), $\mathbf{H_{12}}$ ($n_1 \times n_2$ matrix), $\mathbf{H_{21}}$ ($n_2 \times n_1$ matrix), and $\mathbf{H_{22}}$ ($n_2 \times n_2$ matrix), which

Algorithm 1: Preprocessing phase in BEAR

Input : graph: G, restart probability: c, drop tolerance: ξ
Output: precomputed matrices: $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$

1 compute $\tilde{\mathbf{A}}$, and $\mathbf{H} = \mathbf{I} - (1-c)\tilde{\mathbf{A}}^{\mathbf{T}}$
2 find hubs using SlashBurn [41], and divide spokes into disconnected components by removing the hubs
3 reorder nodes and $\mathbf{H}$ so that the disconnected components form a block-diagonal submatrix $\mathbf{H_{11}}$ where each block is ordered in the ascending order of degrees within the component
4 partition $\mathbf{H}$ into $\mathbf{H_{11}}, \mathbf{H_{12}}, \mathbf{H_{21}}, \mathbf{H_{22}}$
5 decompose $\mathbf{H_{11}}$ into $\mathbf{L_1}$ and $\mathbf{U_1}$ using LU decomposition and compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$
6 compute the Schur complement of $\mathbf{H_{11}}$, $\mathbf{S} = \mathbf{H_{22}} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(\mathbf{H_{12}})))$
7 reorder the hubs in the ascending order of their degrees in $\mathbf{S}$ and reorder $\mathbf{S}$, $\mathbf{H_{21}}$ and $\mathbf{H_{12}}$ according to it
8 decompose $\mathbf{S}$ into $\mathbf{L_2}$ and $\mathbf{U_2}$ using LU decomposition and compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$
9 (BEAR-Approx only) drop entries whose absolute value is smaller than ξ in $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$
10 return $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$

Algorithm 2: Query phase in BEAR

Input : seed node: s, precomputed matrices: $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$
Output: relevance scores: $\mathbf{r}$

1 create $\mathbf{q}$ whose sth entry is 1 and the others are 0
2 partition $\mathbf{q}$ into $\mathbf{q_1}$ and $\mathbf{q_2}$
3 compute $\mathbf{r_2} = c(\mathbf{U_2^{-1}}(\mathbf{L_2^{-1}}(\mathbf{q_2} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{q_1})))))$
4 compute $\mathbf{r_1} = \mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}}\mathbf{r_2}))$
5 create $\mathbf{r}$ by concatenating $\mathbf{r_1}$ and $\mathbf{r_2}$
6 return $\mathbf{r}$

correspond to the adjacency matrix representation of edges between spokes, from spokes to hubs, from hubs to spokes, and between hubs, respectively.

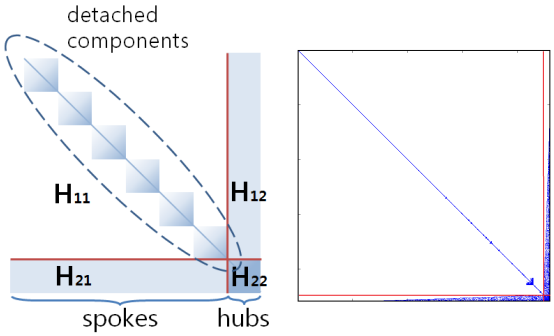
Schur Complement (lines 5 through 6)

In this part, we compute the Schur complement of $\mathbf{H_{11}}$, whose inverse is required in the query phase.

Definition 1 (Schur Complement [16])**.** *Suppose a square matrix* $\mathbf{A}$ *is partitioned into* $\mathbf{A_{11}}$*,* $\mathbf{A_{12}}$*,* $\mathbf{A_{21}}$*, and* $\mathbf{A_{22}}$*, which are* $p \times p$*,* $p \times q$*,* $q \times p$*, and* $q \times q$ *matrices, resp., and* $\mathbf{A_{11}}$ *is invertible. The Schur complement* $\mathbf{S}$ *of the block* $\mathbf{A_{11}}$ *of the matrix* $\mathbf{A}$ *is defined by*

$$\mathbf{S} = \mathbf{A_{22}} - \mathbf{A_{21}}\mathbf{A_{11}^{-1}}\mathbf{A_{12}}.$$

According to the definition, computing the Schur complement $\mathbf{S}$ ($n_2 \times n_2$ matrix) of $\mathbf{H_{11}}$ requires $\mathbf{H_{11}^{-1}}$. Instead of directly inverting $\mathbf{H_{11}}$, we LU decompose it into $\mathbf{L_1}$ and $\mathbf{U_1}$, then compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ instead (the reason will be explained in Section 2.3.1). Consequently, $\mathbf{S}$ is computed by the following



(a) Node reordering method (b) Result on the Routing datset

Figure 2.4: The node reordering method of BEAR and its result on the Routing dataset. BEAR reorders nodes so that edges between spokes form a large but sparse block-diagonal submatrix ($\mathbf{H_{11}}$). Diagonal blocks correspond to the connected components detached from the giant connected component when hubs are removed. Nodes in each block are sorted in the ascending order of their degrees within the component.

rule:

$$\mathbf{S} = \mathbf{H_{22}} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{H_{12}})). \tag{2.5}$$

LU Decomposition (lines 7 through 8)

The block elimination method, which will be explained in Section 2.3.2, requires $\mathbf{H_{11}^{-1}}$ and $\mathbf{S^{-1}}$ to solve Equation (2.2). Directly inverting $\mathbf{H_{11}}$ and $\mathbf{S}$, however, is inefficient since $\mathbf{S^{-1}}$ as well as each diagonal block of $\mathbf{H_{11}^{-1}}$ is likely to be dense. We avoid this problem by replacing $\mathbf{H_{11}^{-1}}$ with $\mathbf{U_1^{-1}}\mathbf{L_1^{-1}}$ as in Equation (2.5) and replacing $\mathbf{S^{-1}}$ with $\mathbf{U_2^{-1}}\mathbf{L_2^{-1}}$ where $\mathbf{L_2}$ and $\mathbf{U_2}$ denote the LU-decomposed matrices of $\mathbf{S}$. To compute $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, and $\mathbf{U_2^{-1}}$ efficiently and make them sparser, we exploit the following observation [29] and lemma.

Observation 1. *Reordering nodes in the ascending order of their degrees speeds up the LU decomposition of an adjacency matrix and makes the inverse of the LU-decomposed matrices sparser.*

Lemma 1. *Suppose* $\mathbf{A}$ *is a non-singular block-diagonal matrix that consists of diagonal blocks,* $\mathbf{A_1}$ *through* $\mathbf{A_b}$*. Let* $\mathbf{L}$ *and* $\mathbf{U}$ *denote the LU-decomposed matrices of* $\mathbf{A}$*, and let* $\mathbf{L_i}$ *and* $\mathbf{U_i}$ *denote those of* $\mathbf{A_i}$ *for all* i $(1 \leq i \leq b)$*. Then,* $\mathbf{L^{-1}}$ *and* $\mathbf{U^{-1}}$ *are the block-diagonal matrices that consist of* $\mathbf{L_1^{-1}}$ *through* $\mathbf{L_b^{-1}}$ *and* $\mathbf{U_1^{-1}}$ *through* $\mathbf{U_b^{-1}}$*, respectively.*

Proof. See Appendix A.2. □

These observation and lemma suggest for $\mathbf{H_{11}^{-1}}$ the reordering method we already used in Section 2.3.1. Since we computed $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ in Section 2.3.1, we only need to process $\mathbf{S}$. We first rearrange the hubs in the ascending order of their degrees within $\mathbf{S}$, which reorders $\mathbf{H_{12}}$, $\mathbf{H_{21}}$, and $\mathbf{H_{22}}$ as well as $\mathbf{S}$. Note that computing $\mathbf{S}$ after reordering the hubs, and reordering the hubs after computing $\mathbf{S}$ produce the same result. After decomposing $\mathbf{S}$ into $\mathbf{L_2}$ and $\mathbf{U_2}$, we compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$.

Dropping Near-zero Entries (line 9, BEAR-Approx only)

The running time and the memory usage in the query phase of our method largely depend on the number of non-zero entries in $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, $\mathbf{U_2^{-1}}$, $\mathbf{H_{12}}$, and $\mathbf{H_{21}}$. Thus, dropping near-zero entries

in them saves time and memory space in the query phase although it sacrifices little accuracy of $\mathbf{r}$. BEAR-Approx drops entries whose absolute value is smaller than the drop tolerance ξ. The effects of different ξ values on accuracy, query time, and memory usage are empirically analyzed in Section 2.4. Note that, contrary to BEAR-Approx, BEAR-Exact guarantees the exactness of $\mathbf{r}$, which will be proved in Section 2.3.2, and still outperforms other exact methods in terms of time and space, which will be shown in Section 2.4.

2.3.2 Query Phase

In the query phase, BEAR computes the RWR score vector $\mathbf{r}$ w.r.t. a given seed node s by exploiting the results of the preprocessing phase. Algorithm 2 describes the overall procedure of the query phase.

The vector $\mathbf{q} = \begin{bmatrix} \mathbf{q_1} \\ \mathbf{q_2} \end{bmatrix}$ denotes the length-n starting vector whose entry at the index of the seed node s is 1 and otherwise 0. It is partitioned into the length-n_1 vector $\mathbf{q_1}$ and the length-n_2 vector $\mathbf{q_2}$. The exact RWR score vector $\mathbf{r}$ is computed by the following equation:

$$\mathbf{r} = \begin{bmatrix} \mathbf{r_1} \\ \mathbf{r_2} \end{bmatrix} = \begin{bmatrix} \mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}}\mathbf{r_2})) \\ c(\mathbf{U_2^{-1}}(\mathbf{L_2^{-1}}(\mathbf{q_2} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{q_1}))))) \end{bmatrix}. \tag{2.6}$$

To prove the correctness of the above equation, we use the block elimination method.

Lemma 2 (Block elimination [16])**.** *Suppose a linear equation* $\mathbf{Ax} = \mathbf{b}$ *is partitioned as*

$$\begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \end{bmatrix} = \begin{bmatrix} \mathbf{b_1} \\ \mathbf{b_2} \end{bmatrix}$$

where $\mathbf{A_{11}}$ *and* $\mathbf{A_{22}}$ *are square matrices. If the submatrix* $\mathbf{A_{11}}$ *is invertible, and* $\mathbf{S}$ *is the Schur complement of the submatrix* $\mathbf{A_{11}}$ *in* $\mathbf{A}$*,*

$$\mathbf{x} = \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}^{-1}}(\mathbf{b_1} - \mathbf{A_{12}}\mathbf{x_2}) \\ \mathbf{S^{-1}}(\mathbf{b_2} - \mathbf{A_{21}}\mathbf{A_{11}^{-1}}\mathbf{b_1}) \end{bmatrix}. \tag{2.7}$$

Theorem 1 (The correctness of BEAR-Exact)**.** *The* $\mathbf{r}$ *in Equation* (2.6) *is equal to the* $\mathbf{r}$ *in Equation* (2.2)*.*

Proof. $\mathbf{H_{11}}$ is invertible because its transpose is a strictly diagonally dominant matrix for $0 < c < 1$ [13]. Thus, by Lemma 2, the following holds for Equation (2.2):

$$\mathbf{r} = \begin{bmatrix} \mathbf{r_1} \\ \mathbf{r_2} \end{bmatrix} = \begin{bmatrix} \mathbf{H_{11}^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}}\mathbf{r_2}) \\ \mathbf{S^{-1}}(c\mathbf{q_2} - \mathbf{H_{21}}\mathbf{H_{11}^{-1}}(c\mathbf{q_1})) \end{bmatrix}.$$

Equation (2.6) only replaces $\mathbf{H_{11}^{-1}}$ with $\mathbf{U_1^{-1}}\mathbf{L_1^{-1}}$ and $\mathbf{S^{-1}}$ with $\mathbf{U_2^{-1}}\mathbf{L_2^{-1}}$ where $\mathbf{H_{11}} = \mathbf{L_1}\mathbf{U_1}$ and $\mathbf{S} = \mathbf{L_2}\mathbf{U_2}$. □

2.3.3 Complexity Analysis

In this section, we analyze the time and space complexity of BEAR. We assume that all the matrices considered are saved in a sparse format, such as the compressed sparse column format [67], which stores only non-zero entries, and that all the matrix operations exploit such sparsity by only considering non-zero entries. We also assume that the number of edges is greater than that of nodes (i.e., $m > n$) for simplicity. The maximum number of non-zero entries in each precomputed matrix is summarized in

Table 2.2: Maximum number of non-zero entries in the precomputed matrices.

Matrix	Max nonzeros
$\mathbf{H_{12}}$ & $\mathbf{H_{21}}$	$O(\min(n_1 n_2, m))$
$\mathbf{L_1^{-1}}$ & $\mathbf{U_1^{-1}}$	$O(\sum_{i=1}^{b} n_{1i}^2)$
$\mathbf{L_2^{-1}}$ & $\mathbf{U_2^{-1}}$	$O(n_2^2)$

Table 2.3: Time complexity of each step of BEAR.

Line	Task	Time complexity
	Preprocessing phase (Algorithm 1)	
2	run SlashBurn	$O(T(m + n \log n))$ [41]
3	reorder $\mathbf{H}$	$O(m + n + \sum_{i=1}^{b} n_{1i} \log n_{1i})$
5	compute $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$	$O(\sum_{i=1}^{b} n_{1i}^3)$
6	compute $\mathbf{S}$	$O(n_2 \sum_{i=1}^{b} n_{1i}^2 + \min(n_1 n_2^2, n_2 m))$
8	compute $\mathbf{L_2^{-1}}$ and $\mathbf{U_2^{-1}}$	$O(n_2^3)$
9	drop near-zero entries	$O(\sum_{i=1}^{b} n_{1i}^2 + n_2^2 + min(n_1 n_2, m))$
Total		$O(T(m + n \log n) + \sum_{i=1}^{b} n_{1i}^3 + n_2 \sum_{i=1}^{b} n_{1i}^2 + n_2^3 + \min(n_2^2 n_1, n_2 m))$
	Query phase (Algorithm 2)	
3	compute $\mathbf{r_2}$	$O(\sum_{i=1}^{b} n_{1i}^2 + n_2^2 + \min(n_1 n_2, m))$
4	compute $\mathbf{r_1}$	$O(\sum_{i=1}^{b} n_{1i}^2 + \min(n_1 n_2, m))$
Total		$O(\sum_{i=1}^{b} n_{1i}^2 + n_2^2 + \min(n_1 n_2, m))$

Table 2.2 where b denotes the number of diagonal blocks in $\mathbf{H_{11}}$, and n_{1i} denotes the number of nodes in the ith diagonal block. The maximum numbers of non-zero entries in $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ depend on the size of diagonal blocks because the LU-decomposed matrices and inverse of a block-diagonal matrix are also block-diagonal matrices with the same block sizes (see Lemma 1).

Time Complexity

The time complexity of each step of BEAR is summarized in Table 2.3. In this section, we provide proofs for nontrivial analysis results starting from the following lemma:

Lemma 3 (Sparse matrix multiplication)**.** *Suppose* $\mathbf{A}$ *and* $\mathbf{B}$ *are* $p \times q$ *and* $q \times r$ *matrices, respectively, and* $\mathbf{A}$ *has* $|\mathbf{A}|$ $(> p, q)$ *non-zero entries. Calculating* $\mathbf{C} = \mathbf{AB}$ *using sparse matrix multiplication takes* $O(|\mathbf{A}|r)$.

Proof. Each non-zero entries in $\mathbf{A}$ is multiplied and then added up to r times. □

Lemma 4. *It takes* $O(\sum_{i=1}^{b} n_{1i}^3)$ *to compute* $\mathbf{L_1^{-1}}$ *and* $\mathbf{U_1^{-1}}$.

Proof. By Lemma 1, each block of $\mathbf{H_{11}}$ is processed separately. Since the LU decomposition of $p \times p$ matrix takes $O(p^3)$ [16], it takes $O(\sum_{i=1}^{b} n_{1i}^3)$ to LU decompose the diagonal blocks of $\mathbf{H_{11}}$. Since the inversion of $p \times p$ matrix takes $O(p^3)$ [16], it takes $O(\sum_{i=1}^{b} n_{1i}^3)$ to invert the decomposed blocks. □

Lemma 5. *It takes* $O(n_2 \sum_{i=1}^{b} n_{1i}^2 + \min(n_1 n_2^2, n_2 m))$ *to compute* $\mathbf{S} = \mathbf{H_{22}} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{H_{12}}))$.

Proof. By Lemma 3 and Table 2.2, it takes $O(n_2 \sum_{i=1}^{b} n_{1i}^2)$ to compute $\mathbf{R_1} = \mathbf{L_1^{-1}}\mathbf{H_{12}}$, $O(n_2 \sum_{i=1}^{b} n_{1i}^2)$ to compute $\mathbf{R_2} = \mathbf{U_1^{-1}}\mathbf{R_1}$, $O(\min(n_1 n_2^2, n_2 m))$ to compute $\mathbf{R_3} = \mathbf{H_{21}}\mathbf{R_2}$, and $O(n_2^2)$ to compute $\mathbf{S} = \mathbf{H_{22}} - \mathbf{R_3}$. □

Theorem 2. *Preprocessing phase in* BEAR *takes* $O(T(m + n \log n) + \sum_{i=1}^{b} n_{1i}^3 + n_2 \sum_{i=1}^{b} n_{1i}^2 + n_2^3 + \min(n_2^2 n_1, n_2 m))$

Proof. See Lemma 4, Lemma 5, and Table 2.3. □

Theorem 3. *Query phase in* BEAR *takes* $O(\sum_{i=1}^{b} n_{1i}^2 + n_2^2 + \min(n_1 n_2, m))$

Proof. Apply Lemma 3 and the results in Table 2.2 to each steps in $\mathbf{r_2} = c(\mathbf{U_2^{-1}}(\mathbf{L_2^{-1}}(\mathbf{q_2} - \mathbf{H_{21}}(\mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}\mathbf{q_1})))))$ and $\mathbf{r_1} = \mathbf{U_1^{-1}}(\mathbf{L_1^{-1}}(c\mathbf{q_1} - \mathbf{H_{12}}\mathbf{r_2}))$ as in Lemma 5. □

In real-world graphs, $\sum_{i=1}^{b} n_{1i}^2$ in the above results can be replaced by m since the number of non-zero entries in $\mathbf{L_1^{-1}}$ and $\mathbf{U_1^{-1}}$ is closer to $O(m)$ than $O(\sum_{i=1}^{b} n_{1i}^2)$ as seen in Table 2.4.

Space Complexity

Theorem 4. BEAR *requires* $O(\sum_{i=1}^{b} n_{1i}^2 + \min(n_1 n_2, m) + n_2^2)$ *memory space for precomputed matrices:* $\mathbf{H_{12}}$, $\mathbf{H_{21}}$, $\mathbf{L_1^{-1}}$, $\mathbf{U_1^{-1}}$, $\mathbf{L_2^{-1}}$, *and* $\mathbf{U_2^{-1}}$.

Proof. See Table 2.2. □

For the same reason, as in the time complexity, $\sum_{i=1}^{b} n_{1i}^2$ in the above result can be replaced by m in real-world graphs.

Theorems 2, 3, and 4 imply that BEAR works efficiently when the given graph is divided into small pieces (small $\sum_{i=1}^{b} n_{1i}^2$ and $\sum_{i=1}^{b} n_{1i}^3$) by removing a small number of hubs (small n_2), which is true in many real world graphs [6, 41].

2.3.4 Application to RWR Variants

Our BEAR method is easily applicable to various RWR variants since BEAR does not assume any unique property of RWR contrary to other methods [29, 84]. In this section, we show how BEAR can be applied to three of such variants.

Personalized PageRank. As explained in Section 2.2.1, personalized PageRank (PPR) selects a restart node according to given probability distribution. PPR can be computed by replacing $\mathbf{q}$ in Algorithm 2 with the probability distribution.

Effective importance. Effective importance (EI) [15] is the degree-normalized version of RWR. It captures the local community structure and adjusts RWR's preference towards high-degree nodes. We can compute EI by dividing each entry of $\mathbf{r}$ in Algorithm 2 by the degree of the corresponding node.

RWR with normalized graph Laplacian. Instead of row-normalized adjacency matrix, Tong et al. [81] use the normalized graph Laplacian. It outputs symmetric relevance scores for undirected graphs, which are desirable for some applications. This score can be computed by replacing $\tilde{\mathbf{A}}$ in Algorithm 1 with $\mathbf{D^{-1/2}}\mathbf{A}\mathbf{D^{-1/2}}$ where $\mathbf{A}$ is an unnormalized adjacency matrix and $\mathbf{D}$ is a diagonal matrix whose (i, i)th entry is the degree of ith node.

2.4 Experiments

Table 2.4: Summary of real-world and synthetic datasets. $|\mathbf{M}|$ denotes the number of non-zero entries in the matrix $\mathbf{M}$. A brief description of each real-world dataset is presented in Appendix A.3.

dataset	n	m	n_2	$\sum_{i=1}^{b} n_{1i}^2$	$\lvert\mathbf{H}\rvert$	$\lvert\mathbf{H_{12}}\rvert + \lvert\mathbf{H_{21}}\rvert$	$\lvert\mathbf{L_1^{-1}}\rvert + \lvert\mathbf{U_1^{-1}}\rvert$	$\lvert\mathbf{L_2^{-1}}\rvert + \lvert\mathbf{U_2^{-1}}\rvert$
Routing	22,963	48,436	572	678,097	119,835	72,168	86,972	271,248
Co-author	31,163	120,029	4,464	1,364,443	271,221	118,012	202,482	18,526,862
Trust	131,828	841,372	10,087	3,493,773	972,627	317,874	318,461	81,039,727
Email	265,214	420,045	1,590	566,435	684,170	358,458	554,681	1,149,462
Web-Stan	281,903	2,312,497	16,017	420,658,754	2,594,400	1,423,993	26,191,040	55,450,105
Web-Notre	325,729	1,497,134	12,350	77,441,937	1,795,408	611,408	5,912,673	12,105,579
Web-BS	685,230	7,600,595	50,005	717,727,201	8,285,825	4,725,657	25,990,336	547,936,698
Talk	2,394,385	5,021,410	19,152	3,272,265	7,415,795	3,996,546	4,841,878	246,235,838
Citation	3,774,768	16,518,948	1,143,522	71,206,030	20,293,715	9,738,225	6,709,469	408,178,662
R-MAT (0.5)	99,982	500,000	17,721	1,513,855	599,982	184,812	204,585	260,247,890
R-MAT (0.6)	99,956	500,000	11,088	1,258,518	599,956	145,281	205,837	104,153,333
R-MAT (0.7)	99,707	500,000	7,029	705,042	599,707	116,382	202,922	43,250,097
R-MAT (0.8)	99,267	500,000	4,653	313,848	599,267	104,415	199,576	19,300,458
R-MAT (0.9)	98,438	500,000	3,038	244,204	598,438	107,770	196,873	8,633,841

To evaluate the effectiveness of our exact method BEAR-Exact, we design and conduct experiments which answer the following questions:

- **Q1. Preprocessing cost (Section 2.4.2).** How much memory space do BEAR-Exact and its competitors require for their precomputed results? How long does this preprocessing phase take?
- **Q2. Query cost (Section 2.4.3).** How quickly does BEAR-Exact answer an RWR query compared with other methods?
- **Q3. Effects of network structure (Section 2.4.4).** How does network structure affect the preprocessing time, query time, and space requirements of BEAR-Exact?

For our approximate method BEAR-Approx, our experiments answer the following questions:

- **Q4. Effects of drop tolerance (Section 2.4.5).** How does drop tolerance ξ affect the accuracy, query time, and space requirements of BEAR-Approx?
- **Q5. Comparison with approximate methods (Section 2.4.6).** Does BEAR-Approx provide a better trade-off between accuracy, time, and space compared with its competitors?

2.4.1 Experimental Settings

Machine. All the experiments are conducted on a PC with a 4-core Intel i5-4570 3.2GHz CPU and 16GB memory.

Data. The graph data used in our experiments are summarized in Table 2.4. A brief description of each real-world dataset is presented in Appendix A.3. For synthetic graphs, we use R-MAT [19] with different p_{ul}, the probability of an edge falling into the upper-left partition. The probabilities for the other partitions are set to $(1 - p_{ul})/3$, respectively.

Implementation. We compare our methods with the iterative method, RPPR [32], BRPPR [32], inversion, LU decomposition [29], QR decomposition [30], B_LIN [81], and NB_LIN [81], all of which are explained in Section 2.2.2. Methods only applicable to undirected graphs [7] or top-k search [35, 84] are excluded. All the methods including BEAR-Exact and BEAR-Approx are implemented using MATLAB, which provides a state-of-the-art linear algebra package. In particular, our implementation of NB_LIN and that of RPPR optimize their open-sourced implementations[1,2] in terms of preprocessing speed and query speed, respectively. The binary code of our method and several datasets are available at `http://koreaskj.snucse.org/programs/`.

Parameters. We set the restart probability c to 0.05 as in the previous work [81][3]. We set k of SlashBurn to $0.001n$ (see Appendix A.1 for the meaning of k), which achieves a good trade-off between running time and reordering quality. The convergence threshold ϵ of the iterative method is set to 10^{-8}, which gives accurate results. For B_LIN and NB_LIN, we use the heuristic decomposition method proposed in their paper, which is much faster with little difference in accuracy compared with SVD in our experiments. The number of partitions in B_LIN, the rank in B_LIN and NB_LIN, and the convergence threshold of RPPR and BRPPR are tuned for each dataset, which are summarized in Table A.1 in Appendix A.4.

[1] http://www.cs.cmu.edu/~htong/pdfs/FastRWR_20080319.tgz

[2] http://www.mathworks.co.kr/matlabcentral/fileexchange/11613-pagerank

[3] In this work, c denotes (1 - restart probability)

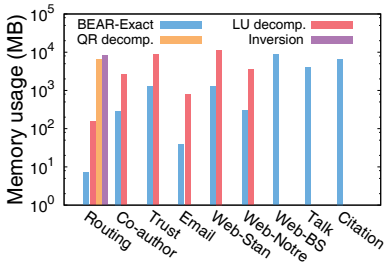


Figure 2.5: Space for preprocessed data. BEAR-Exact requires the least amount of space for preprocessed data on all the datasets. Due to its space efficiency, only BEAR-Exact successfully scales to the largest Citation dataset (with 3.8M nodes) without running out of memory.

2.4.2 Preprocessing Cost

We compare the preprocessing cost of BEAR-Exact with that of other exact methods. Figures 2.1(a) and 2.5 present the preprocessing time and space requirements of the methods except the iterative method, which does not require preprocessing. Only BEAR-Exact successfully preprocesses all the datasets, while others fail due to their high memory requirements.

Preprocessing time is measured in wall-clock time and includes time for SlashBurn (in BEAR-Exact) and community detection (in LU decomposition). BEAR-Exact requires the least amount of time, which is less than an hour, for all the datasets as seen in Figure 2.1(a). Especially, in the graphs with distinct hub-and-spoke structure, BEAR-Exact is up to 12× faster than the second best one. This fast preprocessing of BEAR-Exact implies that it is better at handling frequently changing graphs.

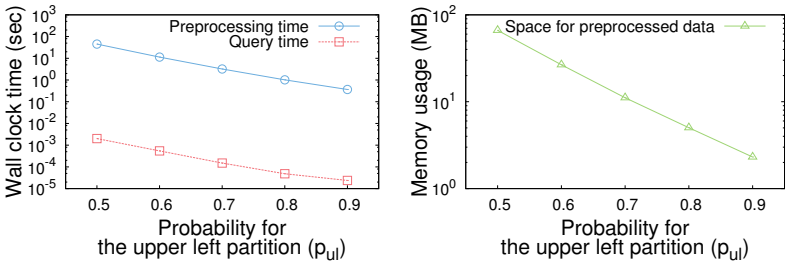
To compare space efficiency, we measure the amount of memory required for precomputed matrices of each method. The precomputed matrices are saved in the compressed sparse column format [67], which requires memory space proportional to the number of non-zero entries. As seen in Figure 2.5, BEAR-Exact requires up to 22× less memory space than its competitors in all the datasets, which results in the superior scalability of BEAR-Exact compared with the competitors.

2.4.3 Query Cost

We compare BEAR-Exact with other exact methods in terms of query time, which means time taken to compute $\mathbf{r}$ for a given seed node. Although time taken for a single query is small compared with preprocessing time, reducing query time is important because many applications require RWR scores for different query nodes (e.g., all nodes) or require the real-time computation of RWR scores for a query node.

Figure 2.1(b) shows the result where y axis represents average query time for 1000 random seed nodes. Only BEAR-Exact and the iterative method run successfully on all the graphs, while the others fail on large graphs due to their high space requirements. BEAR-Exact outperforms its competitors in all the datasets except the smallest one, which is the only dataset that the inversion method can scale to. BEAR-Exact is up to 8× faster than LU decomposition, the second best one, and in the Talk dataset, it is almost 300× faster than the iterative method, which is the only competitor. Although BEAR-Exact requires a preprocessing step which is not needed by the iterative method, for real world applications where RWR scores for many query nodes are required, BEAR-Exact outperforms the iterative method in terms of total running time.

Furthermore, BEAR-Exact also provides the best performance in personalized PageRank, where



(a) Preprocessing time and Query time (b) Space for preprocessed data

Figure 2.6: Effects of network structure. BEAR-Exact becomes fast and space-efficient on a graph with distinct hub-and-spoke structure (a graph with high p_{ul}).

the number of seeds is greater than one. See Figure A.3 of Appendix A.5.1 for the query time of BEAR-Exact with different number of seeds, and see Figure A.2 of Appendix A.5.1 for the comparison of the query time of BEAR-Exact with that of others in personalized PageRank.

2.4.4 Effects of Network Structure

The complexity analysis in Section 2.3.3 indicates that the performance of BEAR-Exact depends on the structure of a given graph. Specifically, the analysis implies that BEAR-Exact is fast and space-efficient on a graph with strong hub-and-spoke structure where the graph is divided into small pieces by removing a small number of hubs. In this experiment, we empirically support this claim using graphs with similar sizes but different structures.

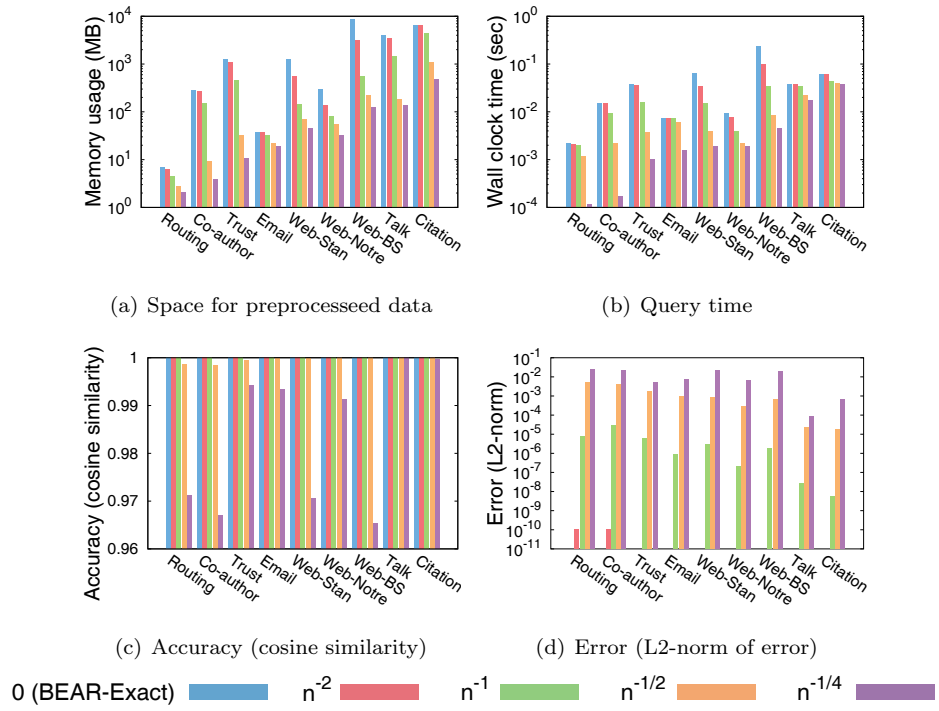
Table 2.4 summarizes four synthetic graphs generated using R-MAT [19] with different p_{ul}, the probability of an edge falling into the upper-left partition. As p_{ul} increases, hub-and-spoke structure becomes stronger as seen from the number of hubs (n_2) and the size of partitions ($\sum_{i=1}^{b} n_{1i}^2$) of each graph. Figure 2.6 shows the performance of BEAR-Exact on theses graphs. Preprocessing time, query time, and space required for preprocessed data decline rapidly with regard to p_{ul}, which is coherent with what the complexity analysis implies.

2.4.5 Effects of Drop Tolerance

As explained in Section 2.3.1, BEAR-Approx improves the speed and space efficiency of BEAR-Exact by dropping near-zero entries in the precomputed matrices although it loses the guarantee of accuracy. In this experiment, we measure the effects of different drop tolerance values on the query time, space requirements, and accuracy of BEAR-Approx. We change the drop tolerance, ξ, from 0 to $n^{-1/4}$ and measure the accuracy using cosine similarity[4] and L2-norm of error[5] between $\mathbf{r}$ computed by BEAR-Exact and $\hat{\mathbf{r}}$ computed by BEAR-Approx with the given drop tolerance.

Figure 2.7 summarizes the results. We observe that both the space required for preprocessed data and the query time of BEAR-Approx significantly decrease compared with those of BEAR-Exact, while the accuracy remains high. When ξ is set to n^{-1}, BEAR-Approx requires up to 16× less space and 7× less query time than BEAR-Exact, while cosine similarity remains higher than 0.999 and L2-norm remains less than 10^{-4}. Similarly, when ξ is set to $n^{-1/4}$, BEAR-Approx requires up to 117×

[4] $(\mathbf{r} \cdot \hat{\mathbf{r}})/(||\mathbf{r}||\,||\hat{\mathbf{r}}||)$, ranging from -1 (dissimilar) to 1 (similar)
[5] $||\hat{\mathbf{r}} - \mathbf{r}||$, ranging from 0 (no error) to $||\hat{\mathbf{r}}|| + ||\mathbf{r}||$ (max error bound)



(a) Space for preprocesseed data (b) Query time

(c) Accuracy (cosine similarity) (d) Error (L2-norm of error)

Figure 2.7: Effects of drop tolerance on the performance of BEAR-Approx. Drop tolerance changes from 0 to $n^{-1/4}$. As drop tolerance increases, space requirements and query time are significantly improved, while accuracy, measured by cosine similarity and L2-norm of error, remains high.

less space and 90× less query time than BEAR-Exact, while cosine similarity remains higher than 0.96 and L2-norm remains less than 0.03.

2.4.6 Comparison with Approximate Methods

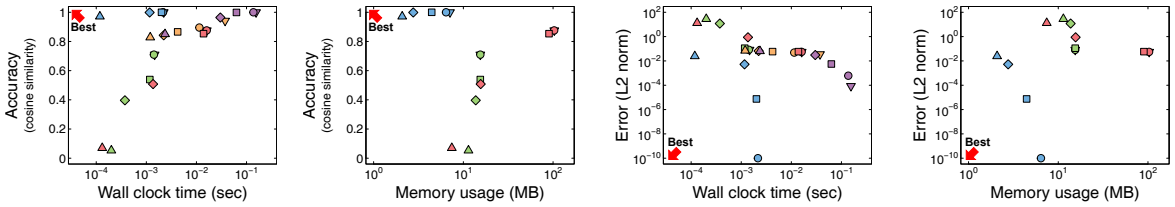
We conduct performance comparison among BEAR-Approx and other state-of-the-art approximate methods. Dropping near-zero entries of precomputed matrices is commonly applied to BEAR-Approx, B_LIN, and NB_LIN, and provides a trade-off between accuracy, query time, and storage cost. We analyze this trade-off by changing drop tolerance from 0 to $n^{-1/4}$. Likewise, we analyze the trade-off between accuracy and time provided by RPPR and BRPPR by changing the threshold to expand nodes, θ_b, from 10^{-4} to 0.5. RPPR and BRPPR do not require space for preprocessed data. Accuracy is measured using cosine similarity and L2-norm of error as in Section 2.4.5.

Figure 2.8 summarizes the result on two datasets. In Figure 2.8(a), the points corresponding to BEAR-Approx lie in the upper-left zone, indicating that it provides a better trade-off between accuracy and time. BEAR-Approx with $\xi = n^{-1/4}$ achieves 254× speedup in query time compared with other methods with the similar accuracy. It also preserves accuracy (> 0.97), while other methods with similar query time produce almost meaningless results.

BEAR-Approx also provides the best trade-off between accuracy and space requirements among preprocessing methods as seen in Figure 2.8(b). BEAR-Approx with $\xi = n^{-1/4}$ saves 50× on storage compared with B_LIN with $\xi = 0$ while providing higher accuracy (> 0.97). NB_LIN with $\xi = 0$ requires 7× more space compared with BEAR-Approx with $\xi = n^{-1/4}$ despite its lower accuracy (< 0.71).

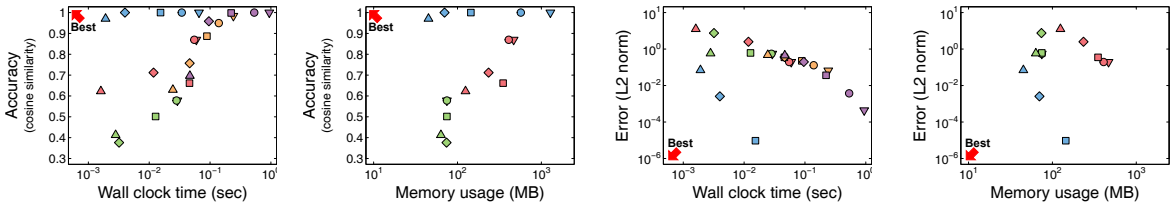
Experiments using L2-norm (Figures 2.8(c) and 2.8(d)) show similar tendencies. BEAR-Approx lies in the lower-left zone, which indicates that it provides the best trade-off. The results on other datasets

Routing:



(a) Query time vs. Cosine similarity (b) Space for preprocessed data vs. Cosine similarity (c) Query time vs. L2-norm of error (d) Space for preprocessed data vs. L2-norm of error

Web-Stan:



(e) Query time vs. Cosine similarity (f) Space for preprocessed data vs. Cosine similarity (g) Query time vs. L2-norm of error (h) Space for preprocessed data vs. L2-norm of error

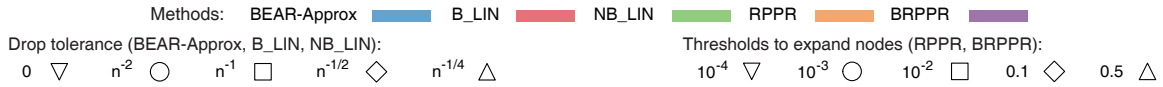


Figure 2.8: Trade-off between time, space, and accuracy provided by approximate methods. The above four subfigures show the results on the Routing dataset, and the below ones show the results on the Web-Stan dataset. The colors distinguish the methods, and the shapes distinguish the drop tolerance (for BEAR-Approx, B_LIN, and NB_LIN) or the threshold to expand nodes (for RPPR and BRPPR). RPPR and BRPPR do not appear in the subfigures in the second and fourth columns because they do not require space for preprocessed data. In the left four subfigures, upper-left region indicates better performance, while in the right ones, lower-left region indicates better performance. Notice that BEAR-Approx provides 1) the best trade-off between accuracy and query speed, and 2) the best trade-off between accuracy and space requirements, among preprocessing methods.

are given in Figure A.5 of Appendix A.5.3. The preprocessing times of the approximate methods are also compared in Figure A.4 of Appendix A.5.2.

2.5 Related Work

In this section, we review related work, which can be categorized into four parts: (1) relevance measures and applications, (2) approximate methods for RWR, (3) top-k search for RWR, and (4) preprocessing methods for RWR.

Relevance measures and applications. There are various relevance (proximity) measures based on random walk, e.g., penalized hitting probability (PHP) [88], effective importance (EI) [15], discounted hitting time (DHT) [71], truncated hitting time (THT) [70], random walk with restart (RWR) [64], effective conductance (EC) [25], ObjectRank [12], SimRank [37], and personalized PageRank (PPR) [63]. Among these measures, RWR has received much interests and has been applied to many applications including community detection [7, 33, 83, 91], ranking [81], link prediction [11], graph matching [80, 43], knowledge discovery [47], anomaly detection [76], content based image retrieval [36], and cross modal

correlation discovery [64]. Andersen et al. [7] proposed a local community detection algorithm which utilizes RWR to find a cut with small conductance near a seed node. This algorithm was used to explore the properties of communities in large graphs since it is fast and returns tight communities [51]. Considerable improvements of the algorithm have been made regarding seed finding [33, 83] and inner connectivity [91]. Backstrom et al. [11] proposed a link prediction algorithm called supervised random walk, which is a variant of RWR. In the algorithm, transition probabilities are determined as a function of the attributes of nodes and edges, and the function is adjusted through supervised learning. Tong et al. [80] proposed G-Ray, which finds best-effort subgraph matches for a given query graph in a large and attributed graph. They formulated a goodness function using RWR to estimate the proximity between a node and a subgraph. Kasneci et al. [47] exploited RWR as a measure of node-based informativeness to extract an informative subgraph for given query nodes. Sun et al. [76] utilized RWR for neighborhood formulation and abnormal node detection in bipartite graphs. He et al. [36] employed RWR to rank retrieved images in their manifold ranking algorithm. Pan et al. [64] proposed a method adopting RWR to compute the correlations between a query image node and caption nodes.

Approximate methods for RWR. The iterative method, which comes from the definition of RWR, is not fast enough in real world applications where RWR scores for different query nodes need to be computed. To overcome this obstacle, approximate approaches have been proposed. Sun et al. [76] observed that the relevance scores for a seed node are highly skewed, and many real-world graphs have a block-wise structure. Based on these observations, they performed random walks only on the partition containing the seed node and assigned the proximities of zero to the other nodes outside the partition. Instead of using a precomputed partition, Gleich et al. [32] proposed methods which adaptively determine the part of a graph used for RWR computation in the query phase as explained in Section 2.2.2. Andersen et al. [7] also proposed an approximate method based on local information. This method, however, is only applicable to undirected graphs. Tong et al. [81] proposed approximate approaches called B_LIN and NB_LIN. They achieved higher accuracy than previous methods by applying a low-rank approximation to cross-partition edges instead of ignoring them. However, our proposed BEAR-Approx outperforms them by providing a better trade-off between accuracy, time, and space.

Top-k search for RWR. Several recent works focus on finding the k most relevant nodes of a seed node instead of calculating the RWR scores of every node. Gupta et al. [35] proposed the basic push algorithm (BPA), which finds top-k nodes for personalized PageRank (PPR) in an efficient way. BPA precomputes relevance score vectors w.r.t. hub nodes and uses them to obtain the upper bounds of PPR scores. Fujiwara et al. [29] proposed K-dash, which computes the RWR scores of top-k nodes exactly. It computes the RWR scores efficiently by exploiting precomputed sparse matrices and pruning unnecessary computations while searching for the top-k nodes. Wu et al. [84] showed that many random walk based measures have the no-local-minimum (or no-local-maximum) property, which means that each node except for a given query node has at least one neighboring node having lower (or higher) proximity. Based on this property, they developed a unified local search method called Fast Local Search (FLoS), which exactly finds top-k relevant nodes in terms of measures satisfying the no-local-optimum property. Furthermore, Wu et al. showed that FLoS can be applied to RWR, which does not have the no-local-optimum property, by utilizing its relationship with other measures. However, top-k RWR computation is insufficient for many data mining applications [7, 11, 33, 36, 76, 80, 83, 91]; on the other hand, BEAR computes the RWR scores of all nodes.

Preprocessing methods for RWR. As seen from Section 2.2.2, the computational cost of RWR can be significantly reduced by precomputing $\mathbf{H}^{-1}$. However, matrix inversion does not scale up. That is,

for a large graph, it often results in a dense matrix that cannot fit to memory. For this reason, alternative methods have been proposed. NB_LIN, proposed by Tong et al. [80], decomposes the adjacency matrix using low-rank approximation in the preprocessing phase and approximates $\mathbf{H}^{-1}$ from these decomposed matrices in the query phase using Sherman-Morrison Lemma [66]. Its variant B_LIN uses this technique only for cross-partition edges. These methods require less space but do not guarantee accuracy. Fujiwara et al. inverted the results of LU decomposition [29] or QR decomposition [30] of $\mathbf{H}$ after carefully reordering nodes. Their methods produce sparser matrices that can be used in place of $\mathbf{H^{-1}}$ in the query phase but still have limited scalability. Contrary to the earlier methods, our proposed BEAR-Exact offers both accuracy and space efficiency.

In addition to the approaches described above, distributed computing is another promising approach. Andersen et al. [8] proposed a distributed platform to solve linear systems including RWR. In order to reduce communication cost, they partitioned a graph into overlapping clusters and assigned each cluster to a distinct processor.

2.6 Conclusion

In this chapter, we propose BEAR, a novel algorithm for fast, scalable, and accurate random walk with restart computation on large graphs. BEAR preprocesses the given graph by reordering and partitioning the adjacency matrix and uses block elimination to compute RWR from the preprocessed results. We discuss the two versions of BEAR: BEAR-Exact and BEAR-Approx. The former guarantees accuracy, while the latter is faster and more space-efficient with little loss of accuracy. We experimentally show that the preprocessing phase of the exact method BEAR-Exact takes up to 12× less time and requires up to 22× less memory space than that of other preprocessing methods guaranteeing accuracy, which makes BEAR-Exact enjoy the superior scalability. BEAR-Exact also outperforms the competitors in the query phase: it is up to 8× faster than other preprocessing methods, and in large graphs where other preprocessing methods run out of memory, is almost 300× faster than its only competitor, the iterative method. The approximate method BEAR-Approx consistently provides a better trade-off between accuracy, time, and storage cost compared with other approximate methods. Future research directions include extending BEAR to support frequently changing graphs [40].

Chapter 3. Distributed Methods for High-dimensional and Large-scale Tensor Factorization

Given a high-dimensional large-scale tensor, how can we decompose it into latent factors? Can we process it on commodity computers with limited memory? These questions are closely related to recommender systems, which have modeled rating data not as a matrix but as a tensor to utilize contextual information such as time and location. This increase in the dimension requires tensor factorization methods scalable with both the dimension and size of a tensor. In this chapter, we propose two distributed tensor factorization methods, CDTF and SALS. Both methods are scalable with all aspects of data and show a trade-off between convergence speed and memory requirements. CDTF, based on coordinate descent, updates one parameter at a time, while SALS generalizes on the number of parameters updated at a time. In our experiments, only our methods factorized a 5-dimension tensor with 1 billion observable entries, 10M mode length, and 1K rank, while all other state-of-the-art methods failed. Moreover, our methods required several orders of magnitude less memory than our competitors. We implemented our methods on MapReduce with two widely-applicable optimization techniques: local disk caching and greedy row assignment. They speeded up our methods up to 98.2× and also the competitors up to 5.9×.

3.1 Introduction

Recommendation problems can be viewed as completing a partially observable user-item matrix whose entries are ratings. Matrix factorization (MF), which decomposes the input matrix into a user factor matrix and an item factor matrix such that their multiplication approximates the input matrix, is one of the most widely-used methods for matrix completion [21, 49, 90]. To handle web-scale data, efforts were made to find distributed ways for MF, including ALS [90], DSGD [31], and CCD++ [86].

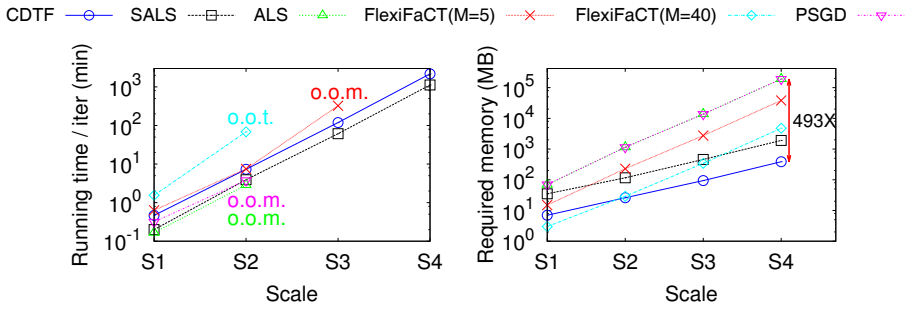
On the other hand, there have been attempts to improve the accuracy of recommendation by using additional contextual information such as time and location. A straightforward way to utilize such extra factors is to model rating data as a partially observable tensor where additional dimensions correspond to the extra factors. Similar to the matrix completion, tensor factorization (TF), which decomposes the input tensor into multiple factor matrices and a core tensor, has been used for tensor completion [46, 59, 89].

As the dimension of web-scale recommendation problems increases, a necessity for TF algorithms scalable with the dimension as well as the size of data has arisen. A promising way to find such algorithms is to extend distributed MF algorithms to higher dimensions. However, the scalability of existing methods including ALS [90], PSGD [58], and FlexiFaCT [14] is limited as we will explain in Section 3.2.3.

In this chapter, we propose Coordinate Descent for Tensor Factorization (CDTF) and Subset Alternating Least Square (SALS), distributed tensor factorization methods scalable with all aspects of data. CDTF applies coordinate descent, which updates one parameter at a time, to TF. SALS, which includes CDTF and ALS as special cases, generalizes on the number of parameters updated at a time.

Table 3.1: Summary of scalability results. The factors which each method is scalable with are checked. Our proposed CDTF and SALS are the only methods scalable with all the factors.

	CDTF	**SALS**	ALS	PSGD	FLEXIFACT
Dimension	✓	✓	✓	✓	
Observations	✓	✓	✓	✓	✓
Mode Length	✓	✓			✓
Rank	✓	✓			✓
Machines	✓	✓	✓		



(a) Running time (b) Memory requirements

Figure 3.1: Scalability comparison of tensor factorization methods on the Hadoop cluster. o.o.m. : out of memory, o.o.t. : out of time (takes more than a week). Only our proposed CDTF and SALS scaled up to the largest scale S4 (details are in Table 3.4), and they showed a trade-off: CDTF was more memory-efficient, and SALS ran faster. The scalability of ALS, PSGD, and FLEXIFACT with five reducers was limited due to their high memory requirements. In particular, ALS and PSGD required 186GB for S4, which is 493× of 387MB that CDTF required. The scalability of FLEXIFACT with 40 reducers was limited because of its rapidly increasing communication cost.

These two methods have distinct advantages: CDTF is more memory-efficient, and SALS converges faster. Our methods can also be applied to any applications handling large-scale partially observable tensors, including social network analysis [26] and Web search [75].

The main contributions of our study are as follows:

- **Algorithm.** We proposed CDTF and SALS, scalable tensor factorization algorithms. Their distributed versions are the only methods scalable with all the following factors: the dimension and size of data, the number of parameters, and the number of machines (Table 3.1).
- **Analysis.** We analyzed our methods and the competitors in a general N-dimensional setting in the following aspects: computational complexity, communication complexity, memory requirements, and convergence speed (Table 3.3).
- **Optimization.** We implemented our methods on MapReduce with two widely-applicable optimization techniques: local disk caching and greedy row assignment. They speeded up not only our methods (up to 98.2×) but also the competitors (up to 5.9×) (Figure 3.10).
- **Experiment.** We empirically confirmed the superior scalability of our methods and their several orders of magnitude less memory requirements than the competitors. Only our methods analyzed a 5-dimensional tensor with 1B observable entries, 10M mode length, and 1K rank, while all others failed (Figure 3.1).

The binary codes of our methods and datasets are available at `http://koreaskj.snucse.org/programs/`. The rest of this chapter is organized as follows. Section 3.2 presents preliminaries for tensor factorization and its distributed algorithms. Section 3.3 describes our proposed CDTF and SALS methods. Section 3.4 presents the optimization techniques for them on MapReduce. Section 3.5 provides experimental results. After reviewing related work in Section 3.6, we conclude in Section 3.7.

3.2 Preliminaries: Tensor Factorization

Table 3.2: Table of symbols.

Symbol	Definition
$\mathcal{X}$	input tensor ($\in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$)
$x_{i_1 \cdots i_N}$	$(i_1, ..., i_N)$th entry of $\mathcal{X}$
N	dimension of $\mathcal{X}$
I_n	length of the nth mode of $\mathcal{X}$
$\mathbf{A}^{(n)}$	nth factor matrix ($\in \mathbb{R}^{I_n \times K}$)
$a^{(n)}_{i_n k}$	(i_n, k)th entry of $\mathbf{A}^{(n)}$
K	rank of $\mathcal{X}$
Ω	set of indices of observable entries of $\mathcal{X}$
$\Omega^{(n)}_{i_n}$	subset of Ω whose nth mode's index is equal to i_n
${}_mS_n$	set of rows of $\mathbf{A}^{(n)}$ assigned to machine m
$\mathcal{R}$	residual tensor ($\in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$)
$r_{i_1 \cdots i_N}$	$(i_1, ..., i_N)$th entry of $\mathcal{R}$
M	number of machines (reducers on MapReduce)
T_{out}	number of outer iterations
T_{in}	number of inner iterations
λ	regularization parameter
C	number of parameters updated at a time
η_0	initial learning rate

In this section, we describe the preliminaries of tensor factorization and its distributed algorithms.

3.2.1 Tensor and the Notations

Tensors are multi-dimensional arrays that generalize vectors (1-dimensional tensors) and matrices (2-dimensional tensors) to higher dimensions. Like rows and columns in a matrix, an N-dimensional tensor has N modes whose lengths are I_1 through I_N, respectively. We denote tensors with variable dimension N by boldface Euler script letters, e.g., $\mathcal{X}$. Matrices and vectors are denoted by boldface capitals, e.g., $\mathbf{A}$, and boldface lowercases, e.g., $\mathbf{a}$, respectively. We denote the entry of a tensor by the symbolic name of the tensor with its indices in subscript. For example, the (i_1, i_2)th entry of $\mathbf{A}$ is denoted by $a_{i_1 i_2}$, and the $(i_1, ..., i_N)$th entry of $\mathcal{X}$ is denoted by $x_{i_1 \cdots i_N}$. The i_1th row of $\mathbf{A}$ is denoted by $\mathbf{a}_{i_1 *}$, and the i_2th column of $\mathbf{A}$ is denoted by $\mathbf{a}_{* i_2}$. Table 3.2 lists the symbols used in this chapter.

Table 3.3: Summary of distributed tensor factorization algorithms. The performance bottlenecks that prevent each algorithm from handling web-scale data are colored red. Only our proposed SALS and CDTF methods have no bottleneck. Communication complexity is measured by the number of parameters that each machine exchanges with the others. For simplicity, we assume that workload of each algorithm is equally distributed across machines, that the length of every mode is equal to I, and that T_{in} of SALS and CDTF is set to one.

Algorithm	**Computational complexity** (per iteration)	**Communication complexity** (per iteration)	**Memory requirements**	**Convergence speed**
CDTF	$O(\|\Omega\|N^2K/M)$	$O(NIK)$	$O(NI)$	Fast
SALS	$O(\|\Omega\|NK(N+C)/M$ $+NIKC^2/M)$	$O(NIK)$	$O(NIC)$	Fastest
ALS [90]	$O(\|\Omega\|NK(N+K)/M$ $+NIK^3/M)$	$O(NIK)$	$O(NIK)$	Fastest
PSGD [58]	$O(\|\Omega\|NK/M)$	$O(NIK)$	$O(NIK)$	Slow
FLEXIFACT [14]	$O(\|\Omega\|NK/M)$	$O(M^{N-2}NIK)$	$O(NIK/M)$	Fast

3.2.2 Tensor Factorization

There are several ways to define a tensor factorization. Our definition is based on the PARAFAC decomposition [48], which is one of the most popular decomposition methods, and the nonzero squared loss with L_2 regularization, whose weighted form has been successfully used in many recommendation systems [21, 49, 90]. Other forms of decomposition and loss functions are considered in Section 3.3.5.

Definition 2 (Tensor Factorization)**.**
Given an N-dimensional tensor $\mathcal{X}(\in \mathbb{R}^{I_1 \times I_2 \dots \times I_N})$ with observable entries $\{x_{i_1 \dots i_N} | (i_1, \dots, i_N) \in \Omega\}$, the rank K factorization of $\mathcal{X}$ is to find factor matrices $\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times K} | 1 \leq n \leq N\}$ which minimize the following loss function:

$$L(\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}) = \sum_{(i_1, \dots, i_N) \in \Omega} \left(x_{i_1 \dots i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a_{i_n k}^{(n)} \right)^2 + \lambda \sum_{n=1}^{N} \|\mathbf{A}^{(n)}\|_F^2 \tag{3.1}$$

Note that the loss function depends only on the observable entries. Each factor matrix $\mathbf{A}^{(n)}$ corresponds to the latent feature vectors of nth mode instances, and $\sum_{k=1}^{K} \prod_{n=1}^{N} a_{i_n k}^{(n)}$ corresponds to the interaction between the features.

3.2.3 Distributed Methods for Tensor Factorization

In this section, we explain how widely-used distributed optimization methods are applied to tensor factorization. Their performances are summarized in Table 3.3. Note that only our proposed CDTF and SALS methods, which will be described in Sections 3.3 and 3.4, have no bottleneck in any aspects. The preliminary version of CDTF and SALS appeared in [74].

Alternating Least Square (ALS)

Using ALS [90], we update factor matrices one by one while keeping all other matrices fixed. When all other factor matrices are fixed, (3.1) is analytically solvable in terms of the updated matrix, which

can be updated row by row due to the independence between rows. The update rule for each row of $\mathbf{A}^{(n)}$ is as follows:

$$[a_{i_n 1}^{(n)}, ..., a_{i_n K}^{(n)}]^T \leftarrow \underset{[a_{i_n 1}^{(n)}, ..., a_{i_n K}^{(n)}]^T}{\arg\min} L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = (\mathbf{B}_{i_n}^{(n)} + \lambda \mathbf{I}_K)^{-1} \mathbf{c}_{i_n}^{(n)}$$

where the (k_1, k_2)th entry of $\mathbf{B}_{i_n}^{(n)} (\in \mathbb{R}^{K \times K})$ is

$$\sum_{(i_1, ..., i_N) \in \Omega_{i_n}^{(n)}} \left(\prod_{l \neq n} a_{i_l k_1}^{(l)} \prod_{l \neq n} a_{i_l k_2}^{(l)} \right),$$

the kth entry of $\mathbf{c}_{i_n}^{(n)} (\in \mathbb{R}^K)$ is

$$\sum_{(i_1, ..., i_N) \in \Omega_{i_n}^{(n)}} \left(x_{i_1 ... i_N} \prod_{l \neq n} a_{i_l k}^{(l)} \right),$$

and $\mathbf{I}_K$ is a K by K identity matrix. $\Omega_{i_n}^{(n)}$ denotes the subset of Ω whose nth mode's index is i_n. This update rule can be proved as in Theorem 5 in Section 3.3.3 since ALS is a special case of SALS.

Updating a row, $\mathbf{a}_{i_n *}^{(n)}$ for example, using (3.2) takes $O(|\Omega_{i_n}|K(N+K) + K^3)$, which consists of $O(|\Omega_{i_n}^{(n)}|NK)$ to calculate $\prod_{l \neq n} a_{i_l 1}^{(l)}$ through $\prod_{l \neq n} a_{i_l K}^{(l)}$ for all the entries in $\Omega_{i_n}^{(n)}$, $O(|\Omega_{i_n}^{(n)}|K^2)$ to build $\mathbf{B}_{i_n}^{(n)}$, $O(|\Omega_{i_n}^{(n)}|K)$ to build $\mathbf{c}_{i_n}^{(n)}$, and $O(K^3)$ to invert $\mathbf{B}_{i_n}^{(n)}$. Thus, updating every row of every factor matrix once, which corresponds to a full ALS iteration, takes $O(|\Omega|NK(N+K) + K^3 \sum_{n=1}^N I_n)$.

In distributed environments, updating each factor matrix can be parallelized without affecting the correctness of ALS by distributing the rows of the factor matrix across machines and updating them simultaneously. The parameters updated by each machine are broadcast to all other machines. The number of parameters each machine exchanges with the others is $O(KI_n)$ for each factor matrix $\mathbf{A}^{(n)}$ and $O(K \sum_{n=1}^N I_n)$ per iteration. The memory requirements of ALS, however, cannot be distributed. Since the update rule (3.2) possibly depends on any entry of any fixed factor matrix, every machine is required to load all the fixed matrices into its memory. This high memory requirements of ALS, requiring $O(K \sum_{n=1}^N I_n)$ memory space per machine, has been noted as a scalability bottleneck even in matrix factorization [31, 85].

Parallelized Stochastic Gradient Descent (PSGD)

PSGD [58] is a distributed algorithm based on stochastic gradient descent (SGD). Using PSGD, we randomly divide the observable entries of $\mathcal{X}$ into M machines which run SGD independently using the assigned entries. The updated parameters are averaged after each iteration. For each observable entry $x_{i_1 ... i_N}$, $a_{i_n k}^{(n)}$ for all n and k, whose number is NK, are updated simultaneously by the following rule :

$$a_{i_n k}^{(n)} \leftarrow a_{i_n k}^{(n)} - 2\eta \left(\frac{\lambda a_{i_n k}^{(n)}}{|\Omega_{i_n}^{(n)}|} - r_{i_1 ... i_N} \left(\prod_{l \neq n} a_{i_l k}^{(l)} \right) \right) \tag{3.2}$$

where $r_{i_1 ... i_N} = x_{i_1 ... i_N} - \sum_{s=1}^K \prod_{l=1}^N a_{i_l s}^{(l)}$. It takes $O(NK)$ to calculate $r_{i_1 ... i_N}$ and $\prod_{l=1}^N a_{i_l k}^{(l)}$ for all k. Once they are calculated, since $\prod_{l \neq n} a_{i_l k}^{(l)}$ can be calculated as $\left(\prod_{l=1}^N a_{i_l k}^{(l)} \right) / a_{i_n k}^{(n)}$, calculating (3.2) takes $O(1)$, and thus updating all the NK parameters takes $O(NK)$. If we assume that $\mathcal{X}$ entries are equally distributed across machines, the computational complexity per iteration is $O(|\Omega|NK/M)$. Averaging parameters can also be distributed, which incurs exchanging $O(K \sum_{n=1}^N I_n)$ parameters per machine.

Like ALS, the memory requirements of PSGD are not distributed, i.e., all the machines are required to load all the factor matrices into their memory, which requires $O(K\sum_{n=1}^{N} I_n)$ memory space per machine. Moreover, PSGD tends to converge slowly due to the non-identifiability of (3.1) [31].

Flexible Factorization of Coupled Tensors (FlexiFaCT)

FlexiFaCT [14] is another SGD-based algorithm that improves on the high memory requirements and slow convergence of PSGD. FlexiFaCT divides $\mathcal{X}$ into M^N blocks. M disjoint blocks of the M^N blocks that do not share common fibers (rows in a general nth mode) compose a stratum. FlexiFaCT processes $\mathcal{X}$ one stratum at a time in which the M blocks composing a stratum are distributed across machines and processed independently. The update rule is the same as (3.2), and the computational complexity per iteration is $O(|\Omega|NK/M)$ as in PSGD. However, contrary to PSGD, averaging is unnecessary because a set of parameters updated by each machine are disjoint with those updated by the 'other machines. In addition, the memory requirements of FlexiFaCT are distributed among the machines. Each machine only needs to load the parameters related to the block it processes, whose number is $(K\sum_{n=1}^{N} I_n)/M$, into its memory at a time. However, FlexiFaCT suffers from high communication cost. After processing one stratum, each machine sends the updated parameters to the machine which updates them using the next stratum. Each machine exchanges at most $(K\sum_{n=2}^{N} I_n)/M$ parameters per stratum and $M^{N-2}K\sum_{n=2}^{N} I_n$ per iteration where M^{N-1} is the number of strata. Thus, the communication cost increases exponentially with the dimension of $\mathcal{X}$ and polynomially with the number of machines.

3.3 Proposed Methods

In this section, we propose two scalable tensor factorization algorithms: CDTF (Section 3.3.1) and SALS (Section 3.3.2). After analyzing their theoretical properties (Section 3.3.3), we discuss how these methods are parallelized in distributed environments (Section 3.3.4) and applied to diverse loss functions (Section 3.3.5).

3.3.1 Coordinate Descent for Tensor Factorization

Update Rule

Coordinate descent for tensor factorization (CDTF) is a tensor factorization algorithm based on coordinate descent and extends CCD++ [86] to higher dimensions. Coordinate descent is a widely-used optimization technique which updates one parameter at a time while keeping all others fixed. This method monotonically decreases the loss function until convergence. CDTF applies coordinate descent to tensor factorization, whose parameters are the entries of factor matrices (i.e., $\mathbf{A}^{(1)}$ through $\mathbf{A}^{(N)}$) and loss function is (3.1). CDTF updates an entry of a factor matrix at a time by assigning the optimal value minimizing (3.1). Equation (3.1) becomes a quadratic equation in terms of the updated parameter when all other parameters are fixed. This leads to the following update rule for each parameter $a_{i_n k}^{(n)}$:

$$a_{i_n k}^{(n)} \leftarrow \arg\min_{a_{i_n k}^{(n)}} L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \frac{\sum_{(i_1,...,i_N)\in\Omega_{i_n}^{(n)}} \left(\left(r_{i_1 \dots i_N} + \prod_{l=1}^{N} a_{i_l k}^{(l)} \right) \prod_{l\neq n} a_{i_l k}^{(l)} \right)}{\lambda + \sum_{(i_1,...,i_N)\in\Omega_{i_n}^{(n)}} \prod_{l\neq n} (a_{i_l k}^{(l)})^2} \tag{3.3}$$

Algorithm 3: Serial version of CDTF

Input : $\mathcal{X}$, K, λ

Output: $\mathbf{A}^{(n)}$ for all n

```
initialize R and A^(n) for all n
for outer iter = 1..T_out do
    for k = 1..K do
        compute R̂ using (3.4)
        for inner iter = 1..T_in do
            for n = 1..N do
                for i_n = 1..I_n do
                    update a^(n)_{i_n k} using (3.5)
        update R using (3.6)
```

where $r_{i_1 \dots i_N} = x_{i_1 \dots i_N} - \sum_{s=1}^{K} \prod_{l=1}^{N} a_{i_l s}^{(l)}$. Computing (3.3) from the beginning takes $O(|\Omega_{i_n}^{(n)}|NK)$, but we can reduce it to $O(|\Omega_{i_n}^{(n)}|N)$ by maintaining a residual tensor $\mathcal{R}$ up-to-date instead of calculating its entries every time. To maintain $\mathcal{R}$ up-to-date, after updating each parameter $a_{i_n k}^{(n)}$, we update the entries of $\mathcal{R}$ in $\{r_{i_1 \dots i_N} | (i_1, ..., i_N) \in \Omega_{i_n}^{(n)}\}$ by the following rule:

$$r_{i_1 \dots i_N} \leftarrow r_{i_1 \dots i_N} + \left((a_{i_n k}^{(n)})^{old} - a_{i_n k}^{(n)} \right) \prod_{l \neq n} a_{i_l k}^{(l)}$$

where $(a_{i_n k}^{(n)})^{old}$ is the parameter value before the update.

Update Sequence

Column-wise Order. Deciding the update sequence of parameters is another important part of coordinate descent. CDTF adopts the following column-wise order:

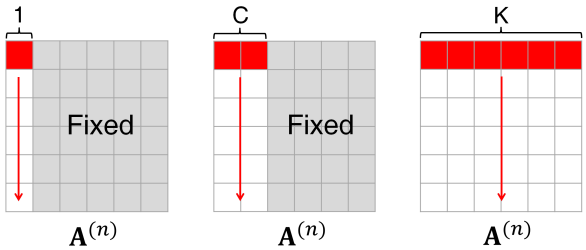
$$(\underbrace{(a_{11}^{(1)} ... a_{I_1 1}^{(1)})}_{\mathbf{a}_{*1}^{(1)}} ... \underbrace{(a_{11}^{(N)} ... a_{I_N 1}^{(N)})}_{\mathbf{a}_{*1}^{(N)}}) ... (\underbrace{(a_{1K}^{(1)} ... a_{I_1 K}^{(1)})}_{\mathbf{a}_{*K}^{(1)}} ... \underbrace{(a_{1K}^{(N)} ... a_{I_N K}^{(N)})}_{\mathbf{a}_{*K}^{(N)}})$$

where we update the entries in, for example, the kth column of a factor matrix and then move on to the kth column of the next factor matrix. After the kth columns of all the factor matrices are updated, the $(k+1)$th columns of the matrices are updated.

In this column-wise order, we can reduce the number of $\mathcal{R}$ updates. Updating, for example, the kth columns of all the factor matrices is equivalent to the rank-one factorization of tensor $\hat{\mathcal{R}}$ whose $(i_1, ..., i_N)$th entry is $\hat{r}_{i_1 \dots i_N} = x_{i_1 \dots i_N} - \sum_{s \neq k} \prod_{l=1}^{N} a_{i_l s}^{(l)}$. $\hat{\mathcal{R}}$ can be computed from $\mathcal{R}$ by the following rule:

$$\hat{r}_{i_1 \dots i_N} \leftarrow r_{i_1 \dots i_N} + \prod_{n=1}^{N} a_{i_n k}^{(n)}, \tag{3.4}$$

which takes $O(N)$ for each entry and $O(|\Omega|N)$ for entire $\hat{\mathcal{R}}$. Once $\hat{\mathcal{R}}$ is computed, we can compute the rank-one factorization (i.e., updating $a_{i_n k}^{(n)}$ for all n and i_n) without updating $\hat{\mathcal{R}}$. This is because the entries of $\hat{\mathcal{R}}$ do not depend on the parameters updated during the rank-one factorization. Each parameter



(a) CDTF (b) SALS (c) ALS

Figure 3.2: Update rules of CDTF, SALS, and ALS. CDTF updates each column of factor matrices entry by entry, SALS updates each C ($1 \leq C \leq K$) columns row by row, and ALS updates all K columns row by row.

$a^{(n)}_{i_n k}$ is updated in $O(|\Omega^{(n)}_{i_n}|N)$ by the following rule:

$$a^{(n)}_{i_n k} \leftarrow \frac{\sum_{(i_1,\ldots,i_N)\in\Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1\ldots i_N} \prod_{l\neq n} a^{(l)}_{i_l k} \right)}{\lambda + \sum_{(i_1,\ldots,i_N)\in\Omega^{(n)}_{i_n}} \prod_{l\neq n} (a^{(l)}_{i_l k})^2}. \tag{3.5}$$

We need to update $\mathcal{R}$ only once per rank-one factorization by the following rule:

$$r_{i_1\ldots i_N} \leftarrow \hat{r}_{i_1\ldots i_N} - \prod_{n=1}^{N} a^{(n)}_{i_n k}, \tag{3.6}$$

which takes $O(|\Omega|N)$ for entire $\mathcal{R}$ as in (3.4).

Inner Iterations. The number of updates on $\mathcal{R}$ can be reduced further by repeating the update of, for example, the kth columns multiple times before moving on to the $(k+1)$th columns. Since the repeated computation (i.e., rank-one factorization) does not require to update $\mathcal{R}$, if the number of the repetitions is T_{in}, $\mathcal{R}$ needs to be updated only once per T_{in} times of rank-one factorization. With T_{in} times of iteration, the update sequence at column level changes as follows:

$$(\mathbf{a}^{(1)}_{*1}\ldots\mathbf{a}^{(N)}_{*1})^{T_{in}} (\mathbf{a}^{(1)}_{*2}\ldots\mathbf{a}^{(N)}_{*2})^{T_{in}} \ldots (\mathbf{a}^{(1)}_{*K}\ldots\mathbf{a}^{(N)}_{*K})^{T_{in}}.$$

Algorithm 3 describes the serial version of CDTF with T_{out} times of outer iteration. We initialize the entries of $\mathbf{A}^{(1)}$ to zero and those of all other factor matrices to random values, which makes the initial value of $\mathcal{R}$ equal to $\mathcal{X}$. Instead of computing $\hat{\mathcal{R}}$ (line 4) before rank-one factorization, the entries of $\hat{\mathcal{R}}$ can be computed while computing (3.5) and (3.6). This can result in better performance on a disk-based system like MapReduce by eliminating disk I/O operations required to compute and store $\hat{\mathcal{R}}$.

3.3.2 Subset Alternating Least Square

Subset alternating least square (SALS), another scalable tensor factorization algorithm, generalizes CDTF in terms of the number of parameters updated at a time. Figure 3.2 depicts the difference among CDTF, SALS, and ALS. Unlike CDTF, which updates each column of factor matrices entry by entry, and ALS, which updates all K columns row by row, SALS updates each C ($1 \leq C \leq K$) columns row by row. SALS contains CDTF ($C = 1$) and ALS ($C = K$) as special cases. As we show in Section 3.5, with proper C, SALS enjoys both the high scalability of CDTF and the fast convergence of ALS although SALS requires more memory space than CDTF.

Algorithm 4: Serial version of SALS

Input : $\mathcal{X}$, K, λ

Output: $\mathbf{A}^{(n)}$ for all n

```
1  initialize R and A^(n) for all n
2  for outer iter = 1..T_out do
3      for split iter = 1..⌈K/C⌉ do
4          choose k_1, ..., k_C (from columns not updated yet)
5          compute R̂ using (3.7)
6          for inner iter = 1..T_in do
7              for n = 1..N do
8                  for i_n = 1..I_n do
9                      update a^(n)_{i_n k_1}, ..., a^(n)_{i_n k_C} using (3.8)
10         update R using (3.9)
```

Algorithm 4 describes the procedure of SALS. Line 4, where C columns are randomly chosen, and line 9, where C parameters are updated simultaneously, are the major differences from CDTF.

Updating C columns $(k_1, ..., k_C)$ of all the factor matrices (lines 7 through 9) is equivalent to the rank-C factorization of $\hat{\mathcal{R}}$ whose $(i_1, ..., i_N)$th entry is $\hat{r}_{i_1...i_N} = x_{i_1...i_N} - \sum_{k \notin \{k_1,...,k_C\}} \prod_{n=1}^{N} a^{(n)}_{i_n k}$. $\hat{\mathcal{R}}$ can be computed from $\mathcal{R}$ by the following rule (line 5):

$$\hat{r}_{i_1...i_N} = r_{i_1...i_N} + \sum_{c=1}^{C} \prod_{n=1}^{N} a^{(n)}_{i_n k_c}, \tag{3.7}$$

which takes $O(NC)$ for each entry and $O(|\Omega|NC)$ for entire $\hat{\mathcal{R}}$. After computing $\hat{\mathcal{R}}$, the parameters in the C columns are updated row by row by the following rule (line 9):

$$[a^{(n)}_{i_n k_1}, ..., a^{(n)}_{i_n k_C}]^T \leftarrow \underset{[a^{(n)}_{i_n k_1}, ..., a^{(n)}_{i_n k_C}]^T}{\arg\min} L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = (\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_C)^{-1} \mathbf{c}^{(n)}_{i_n} \tag{3.8}$$

where the (c_1, c_2)th entry of $\mathbf{B}^{(n)}_{i_n} (\in \mathbb{R}^{C \times C})$ is

$$\sum_{(i_1,...,i_N) \in \Omega^{(n)}_{i_n}} \left(\prod_{l \neq n} a^{(l)}_{i_l k_{c_1}} \prod_{l \neq n} a^{(l)}_{i_l k_{c_2}} \right),$$

the cth entry of $\mathbf{c}^{(n)}_{i_n} (\in \mathbb{R}^C)$ is

$$\sum_{(i_1,...,i_N) \in \Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1...i_N} \prod_{l \neq n} a^{(l)}_{i_l k_c} \right),$$

and $\mathbf{I}_C$ is a C by C identity matrix. The correctness of this update rule is proved in Section 3.3.3. Computing (3.8) takes $O(|\Omega^{(n)}_{i_n}|C(N + C) + C^3)$, which consists of $O(|\Omega^{(n)}_{i_n}|NC)$ to compute $\prod_{l \neq n} a^{(l)}_{i_l k_1}$ through $\prod_{l \neq n} a^{(l)}_{i_l k_C}$ for all the entries in $\Omega^{(n)}_{i_n}$, $O(|\Omega^{(n)}_{i_n}|C^2)$ to build $\mathbf{B}^{(n)}_{i_n}$, $O(|\Omega^{(n)}_{i_n}|C)$ to build $\mathbf{c}^{(n)}_{i_n}$, and $O(C^3)$ to compute the inverse. Instead of computing the inverse, Cholesky decomposition, which also takes $O(C^3)$, can be used. After repeating this rank-C factorization T_{in} times (line 6), $\mathcal{R}$ is updated by

the following rule (line 10):

$$r_{i_1 \dots i_N} \leftarrow \hat{r}_{i_1 \dots i_N} - \sum_{c=1}^{C} \prod_{n=1}^{N} a_{i_n k_c}^{(n)}, \tag{3.9}$$

which takes $O(|\Omega|NC)$ for the entire $\boldsymbol{\mathcal{R}}$ as in (3.7).

3.3.3 Theoretical Analysis

Convergence Analysis

In this section, we prove that CDTF and SALS monotonically decrease the loss function (3.1) until convergence.

Theorem 5 (Correctness of SALS)**.** *The update rule* (3.8) *is correct. That is,*

$$\underset{[a_{i_n k_1}^{(n)}, \dots, a_{i_n k_C}^{(n)}]^T}{\arg\min} L(\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}) = (\mathbf{B}_{i_n}^{(n)} + \lambda \mathbf{I}_C)^{-1} \mathbf{c}_{i_n}^{(n)}.$$

Proof.

$$\begin{aligned}
&\frac{\partial L}{\partial a_{i_n k_c}^{(n)}} = 0, \forall c, 1 \le c \le C \\
&\Leftrightarrow \sum_{(i_1, \dots, i_N) \in \Omega_{i_n}^n} \left(\left(\sum_{s=1}^{C} \prod_{l=1}^{N} a_{i_l k_s}^{(l)} - \hat{r}_{i_1 \dots i_N} \right) \prod_{l \neq n} a_{i_l k_c}^{(l)} \right) + \lambda a_{i_n k_c}^{(n)} = 0, \forall c \\
&\Leftrightarrow \sum_{(i_1, \dots, i_N) \in \Omega_{i_n}^n} \left(\sum_{s=1}^{C} \left(a_{i_n k_s}^{(n)} \prod_{l \neq n} a_{i_l k_s}^{(l)} \right) \prod_{l \neq n} a_{i_l k_c}^{(l)} \right) + \lambda a_{i_n k_c}^{(n)} = \sum_{(i_1, \dots, i_N) \in \Omega_{i_n}^n} \left(\hat{r}_{i_1 \dots i_N} \prod_{l \neq n} a_{i_l k_c}^{(l)} \right), \forall c \\
&\Leftrightarrow (\mathbf{B}_{i_n}^{(n)} + \lambda \mathbf{I}_c)[a_{i_n k_1}^{(n)}, \dots, a_{i_n k_C}^{(n)}]^T = \mathbf{c}_{i_n}^{(n)}.
\end{aligned}$$

□

This theorem also proves the correctness of CDTF, which is a special case of SALS, and leads to the following convergence property.

Theorem 6 (Convergence of CDTF and SALS)**.** CDTF *and* SALS *monotonically decrease the loss function* (3.1)*.*

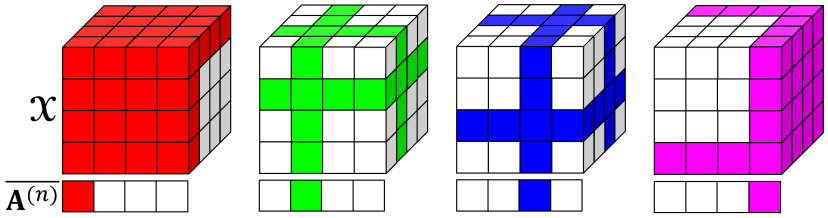
Proof. From Theorem 5, every update in CDTF and SALS minimizes the loss function (3.1) in terms of updated parameters. Thus, the loss function never increases. □

Complexity Analysis

In this section, we analyze the computational and space complexity of CDTF and SALS.

Theorem 7 (Computational complexity of SALS)**.** *The computational complexity of the serial version of* SALS *(Algorithm 4) is* $O(T_{out}|\Omega|NT_{in}K(N+C) + T_{out}T_{in}KC^2 \sum_{n=1}^{N} I_n)$.

Proof. As explained in Section 3.3.2, updating C parameters, $(a_{i_n k_1}^{(n)}, \dots, a_{i_n k_C}^{(n)})$ for example, using (3.8) (line 9 of Algorithm 4) takes $O(|\Omega_{i_n}^{(n)}|C(C+N) + C^3)$; and both computing $\hat{\boldsymbol{\mathcal{R}}}$ (line 5) and updating $\boldsymbol{\mathcal{R}}$ (line 10) take $O(|\Omega|NC)$. Thus, updating all the entries in C columns (lines 8 through 9) takes $O(|\Omega|C(C+N) + I_n C^3)$, and the rank C factorization (lines 7 through 9) takes $O(|\Omega|NC(N+C) + C^3 \sum_{n=1}^{N} I_n)$. As a result, an outer iteration, which repeats the rank C factorization $T_{in}K/C$ times and both $\hat{\boldsymbol{\mathcal{R}}}$ and $\boldsymbol{\mathcal{R}}$ updates K/C times, takes $O(|\Omega|NT_{in}K(N+C) + T_{in}KC^2 \sum_{n=1}^{N} I_n) + O(|\Omega|NK)$, where the second term is dominated. □



(a) Machine 1 (b) Machine 2 (c) Machine 3 (d) Machine 4

Figure 3.3: Work and data distribution of CDTF and SALS in distributed environments when an input tensor is a three-dimensional tensor and the number of machines is four. The rows of the factor matrices are assigned to the machines sequentially. The colored region of $\overline{\mathbf{A}^{(n)}}$ (the transpose of $\mathbf{A}^{(n)}$) in each sub-figure corresponds to the parameters updated by each machine, resp., and that of $\mathcal{X}$ corresponds to the data distributed to each machine.

Theorem 8 (Space complexity of SALS)**.** *The memory requirement of the serial version of* SALS *(Algorithm 4) is* $O(C\sum_{n=1}^{N} I_n)$.

Proof. Since $\hat{\mathcal{R}}$ computation (line 5 of Algorithm 4), rank C factorization (lines 7 through 9), and $\mathcal{R}$ update (line 10) all depend only on the C columns of the factor matrices, the number of whose entries is $C\sum_{n=1}^{N} I_n$, the other $(K-C)$ columns need not be loaded into the memory. Thus, the columns of the factor matrices can be loaded by turns depending on $(k_1, ..., k_C)$ values. Moreover, updating C columns (lines 8 through 9) can be processed by streaming the entries of $\hat{\mathcal{R}}$ from disk and processing them one by one instead of loading them all at once because the entries of $\mathbf{B}_{i_n}^{(n)}$ and $\mathbf{c}_{i_n}^{(n)}$ in (3.8) are the sum of the values calculated independently from each $\hat{\mathcal{R}}$ entry. Likewise, $\hat{\mathcal{R}}$ computation and $\mathcal{R}$ update can also be processed by streaming $\mathcal{R}$ and $\hat{\mathcal{R}}$, respectively. □

These theorems are applied to CDTF, which is a special case of SALS.

Theorem 9 (Computational complexity of CDTF)**.** *The computational complexity of the serial version of* CDTF *(Algorithm 3) is* $O(T_{out}|\Omega|N^2T_{in}K)$.

Theorem 10 (Space complexity of CDTF)**.** *The memory requirement of the serial version of* CDTF *(Algorithm 3) is* $O(\sum_{n=1}^{N} I_n)$.

3.3.4 Parallelization in Distributed Environments

In this section, we describe the distributed versions of CDTF and SALS. We assume a distributed environment where machines do not share memory, such as in MapReduce. The extension to shared-memory systems, where the only difference is that we do not need to broadcast updated parameters, is straightforward. The method to assign the rows of the factor matrices to machines (i.e., to decide ${}_mS_n$ for all n and m) is explained in Section 12, and until then, we assume that the row assignments are given.

Distributed Version of CDTF

Work and Data Distribution. Since the update rule (3.5) for each parameter $a_{i_nk}^{(n)}$ does not depend on the other parameters in its column $\mathbf{a}_{*k}^{(n)}$, updating the parameters in a column can be distributed across multiple machines and processed simultaneously without affecting the correctness of

Algorithm 5: Distributed version of CDTF

Input : $\mathcal{X}$, K, λ, ${}_mS_n$ for all m and n

Output: $\mathbf{A}^{(n)}$ for all n

```
1  distribute the _mΩ entries of X to each machine m
2  Parallel (P): initialize the _mΩ entries of R
3  P: initialize A^(n) for all n
4  for outer iter = 1..T_out do
5      for k = 1..K do
6          P: compute the _mΩ entries of R̂
7          for inner iter = 1..T_in do
8              for n = 1..N do
9                  P: update _m a^(n)_*k using (3.5)
10                 P: broadcast _m a^(n)_*k
11         P: update the _mΩ entries of R using (3.6)
```

CDTF. For each column $\mathbf{a}^{(n)}_{*k}$, machine m updates the assigned parameters, ${}_m\mathbf{a}^{(n)}_{*k} = \{a^{(n)}_{i_n k} | i_n \in {}_mS_n\}$. At matrix level, ${}_mS_n$ rows of each factor matrix $\mathbf{A}^{(n)}$ are updated by machine m. The entries of $\mathcal{X}$ necessary to update the assigned parameters are distributed to each machine. In other words, $\mathcal{X}$ entries in ${}_m\Omega = \bigcup_{n=1}^{N} \left(\bigcup_{i_n \in {}_mS_n} \Omega^{(n)}_{i_n} \right)$ are distributed to each machine m, and the machine maintains and updates $\mathcal{R}$ entries in ${}_m\Omega$.

The sets of $\mathcal{X}$ entries sent to the machines are not disjoint (i.e., ${}_{m_1}\Omega \cap {}_{m_2}\Omega \neq \emptyset$), and each entry of $\mathcal{X}$ can be sent to N machines at most. Thus, the computational cost for computing $\hat{\mathcal{R}}$ and updating $\mathcal{R}$ increases up to N times in distributed environments. However, since this cost is dominated by the others, the overall asymptotic complexity remains the same. Figure 3.3 shows an example of work and data distribution.

Communication. In order to update the parameters, for example, in the kth column of a factor matrix using (3.5), the kth columns of all other factor matrices are required. Therefore, after updating the kth column of a factor matrix, the updated parameters are broadcast to all other machines so that they can be used to update the kth column of the next factor matrix. The broadcast parameters are also used to update the entries of $\mathcal{R}$ using (3.6). Therefore, after updating the assigned parameters in $\mathbf{a}^{(n)}_{*k}$, each machine m broadcasts $|{}_mS_n|$ parameters and receives $(I_n - |{}_mS_n|)$ parameters from the other machines. The number of parameters each machine exchanges with the other machines is $\sum_{n=1}^{N} I_n$ per rank-one factorization and $KT_{in} \sum_{n=1}^{N} I_n$ per outer iteration. Algorithm 5 depicts the distributed version of CDTF.

Distributed Version of SALS

SALS is also parallelized in distributed environments without affecting its correctness. The work and data distribution of SALS is exactly the same as that of CDTF, and the only difference between two methods is that in SALS each machine updates and broadcasts C columns at a time.

Algorithm 6 describes the distributed version of SALS. After updating the assigned parameters in C columns of a factor matrix $\mathbf{A}^{(n)}$ (line 10), each machine m sends $C|{}_mS_n|$ parameters and receives $C(I_n - |{}_mS_n|)$ parameters (line 11), and thus each machine exchanges $C \sum_{n=1}^{N} I_n$ parameters during

Algorithm 6: Distributed version of SALS

Input : $\mathcal{X}$, K, λ, ${}_mS_n$ for all m and n

Output: $\mathbf{A}^{(n)}$ for all n

1 distribute the ${}_m\Omega$ entries of $\mathcal{X}$ to each machine m
2 **Parallel (P):** initialize the ${}_m\Omega$ entries of $\mathcal{R}$
3 **P:** initialize $\mathbf{A}^{(n)}$ for all n
4 **for** *outer iter = 1..*T_{out} **do**
5 **for** *split iter = 1..*$\lceil \frac{K}{C} \rceil$ **do**
6 choose $(k_1, ..., k_C)$ (from columns not updated yet)
7 **P:** compute the ${}_m\Omega$ entries of $\hat{\mathcal{R}}$
8 **for** *inner iter = 1..*T_{in} **do**
9 **for** n = *1..*N **do**
10 **P:** update $\{a^{(n)}_{i_n k_c} | i_n \in {}_mS_n, 1 \leq c \leq C\}$ using (3.8)
11 **P:** broadcast $\{a^{(n)}_{i_n k_c} | i_n \in {}_mS_n, 1 \leq c \leq C\}$
12 **P:** update the ${}_m\Omega$ entries of $\mathcal{R}$ using (3.9)

rank C factorization (lines 9 through 11). Since this factorization is repeated $T_{in}K/C$ times, the total number of parameters each machine exchanges is $KT_{in}\sum_{n=1}^{N} I_n$ per outer iteration, which is the same as that of CDTF.

Row Assignment

The running time of the parallel steps in the distributed versions of CDTF and SALS depends on the longest running time among all machines. Specifically, the running time of lines 6, 9, and 11 in Algorithm 5 and lines 7, 10, and 12 in Algorithm 6 are proportional to $\max_m |{}_m\Omega^{(n)}|$ where ${}_m\Omega^{(n)} = \bigcup_{i_n \in {}_mS_n} \Omega^{(n)}_{i_n}$. Line 10 in Algorithm 5 and line 11 in Algorithm 6 are proportional to $\max_m |{}_mS_n|$. Therefore, it is important to assign the rows of the factor matrices to the machines (i.e., to decide ${}_mS_n$) such that $|{}_m\Omega^{(n)}|$ and $|{}_mS_n|$ are evenly distributed among all the machines. For this purpose, we design a greedy assignment algorithm which aims to minimize $\max_m |{}_m\Omega^{(n)}|$ under the condition that $|{}_mS_n|$ is completely even (i.e., $|{}_mS_n| = I_n/M$ for all n where M is the number of machines). For each factor matrix $\mathbf{A}^{(n)}$, we sort its rows in the decreasing order of $|\Omega^{(n)}_{i_n}|$ and assign the rows one by one to the machine m that satisfies $|{}_mS_n| < \lceil I_n/M \rceil$ and has the smallest $|{}_m\Omega^{(n)}|$ currently. Algorithm 7 describes the details of the algorithm. The effects of greedy row assignment on actual running times are described in Section 3.5.5.

3.3.5 Loss Functions and Updates

In this section, we discuss how CDTF and SALS are applied to minimize various loss functions. We consider the PARAFAC decomposition with L_1 and weighted-L_2 regularization and non-negativity constraints. Coupled tensor factorization and factorization using the bias model are also considered.

Algorithm 7: Greedy row assignment in CDTF and SALS

Input : $\mathcal{X}$, M

Output: ${}_mS_n$ for all m and n

initialize $|{}_m\Omega|$ to 0 for all m
for $n = 1..N$ **do**
  initialize ${}_mS_n$ to $\emptyset$ for all m
  initialize $|{}_m\Omega^{(n)}|$ to 0 for all m
  calculate $|\Omega^{(n)}_{i_n}|$ for all i_n
  foreach i_n (in decreasing order of $|\Omega^{(n)}_{i_n}|$) **do**
    find m with $|{}_mS_n| < \lceil \frac{I_n}{M} \rceil$ and the smallest $|{}_m\Omega^{(n)}|$
    (in case of a tie, choose the machine with smaller $|{}_mS_n|$, and if still a tie, choose the one with smaller $|{}_m\Omega|$)
    add i_n to ${}_mS_n$
    add $|\Omega^{(n)}_{i_n}|$ to $|{}_m\Omega^{(n)}|$ and $|{}_m\Omega|$

PARAFAC with L_1 Regularization

L_1 regularization, which is known to lead to sparse factor matrices, can be used in the PARAFAC decomposition (Section 3.2.2) instead of L_2 regularization. With L_1 regularization, the loss function (3.1) is changed as follows:

$$L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega} \left(x_{i_1...i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_n k} \right)^2 + \lambda \sum_{n=1}^{N} \|\mathbf{A}^{(n)}\|_1. \tag{3.10}$$

In CDTF, (3.10) becomes the sum of a quadratic function and the absolute value function in terms of an updated parameter if we keep all other parameters fixed. This sum can be minimized by the following update rule, which replaces (3.5):

$$a^{(n)}_{i_n k} \leftarrow \begin{cases} (\lambda - g)/d & \text{if g} > \lambda \\ -(\lambda + g)/d & \text{if g} < \text{-}\lambda \\ 0 & \text{otherwise} \end{cases}$$

where $g = -2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \left(\hat{r}_{i_1...i_N} \prod_{l\neq n} a^{(l)}_{i_l k} \right)$ and $d = 2\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} \prod_{l\neq n} (a^{(l)}_{i_l k})^2$.

However, a closed form solution for minimizing the loss function dose not exist in SALS when $C \geq 2$. In SALS, therefore, an iterative or suboptimal update rule should be used instead of (3.8).

PARAFAC with Weighted L_2 Regularization

Weighted L_2 regularization [90] has been used in many recommendation systems based on matrix factorization [21, 49, 90]. This regularization method can be applied to the PARAFAC decomposition and results in the following loss function:

$$L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{(i_1,...,i_N)\in\Omega} \left(x_{i_1...i_N} - \sum_{k=1}^{K} \prod_{n=1}^{N} a^{(n)}_{i_n k} \right)^2 + \lambda \sum_{n=1}^{N} \sum_{i_n=1}^{I_n} |\Omega^{(n)}_{i_n}| \cdot \|\mathbf{a}^{(\mathbf{n})}_{\mathbf{i_n *}}\|_2^2. \tag{3.11}$$

CDTF and SALS can be used to minimize (3.11) by replacing λ in (3.5) and (3.8) with $\lambda|\Omega^{(n)}_{i_n}|$.

PARAFAC with Non-negativity Constraint

The constraint that factor matrices have only non-negative entries has been adopted in many applications [14, 23] because the resulting factor matrices are easier to inspect. If CDTF is used with this constraint, each parameter is updated to the maximum value between zero and the value obtained by (3.5). This way of update minimizes (3.1) under the non-negativity constraint in terms of an updated parameter. In SALS, however, a closed form solution for minimizing (3.1) does not exist when $C \geq 2$ under the non-negativity constraint. Thus, an iterative or suboptimal update rule should be used instead of (3.8) in SALS.

Coupled Tensor Factorization

The joint factorization of tensors sharing common modes (e.g., user-movie rating data and a social network among users) can provide a better understanding of latent factors [14, 2]. Suppose that an N_x-dimensional tensor $\mathcal{X}$ and an N_y-dimensional tensor $\mathcal{Y}$ share their first mode without loss of generality. The coupled factorization of them decomposes $\mathcal{X}$ into factor matrices ${}_x\mathbf{A}^{(1)}$ through ${}_x\mathbf{A}^{(N_x)}$ and $\mathcal{Y}$ into ${}_y\mathbf{A}^{(1)}$ through ${}_y\mathbf{A}^{(N_y)}$ under the condition that ${}_x\mathbf{A}^{(1)} = {}_y\mathbf{A}^{(1)}$. The loss function of the coupled factorization is the sum of the loss function of each separate factorization (Equation (3.1)) as follows:

$$L_{Coupled}({}_x\mathbf{A}^{(1)}, ..., {}_x\mathbf{A}^{(N_x)}, {}_y\mathbf{A}^{(1)}, ..., {}_y\mathbf{A}^{(N_y)}) = \\ L({}_x\mathbf{A}^{(1)}, ..., {}_x\mathbf{A}^{(N_x)}) + L({}_y\mathbf{A}^{(1)}, ..., {}_y\mathbf{A}^{(N_y)}) - \lambda\|{}_x\mathbf{A}^{(1)}\|_F^2 \quad (3.12)$$

where $-\lambda\|{}_x\mathbf{A}^{(1)}\|_F^2$ prevents the shared matrix from being regularized twice.

The update rule for the entries in the non-shared factor matrices are the same as that of separate factorization. When updating the entries in the shared matrix, however, we should consider both tensors. In SALS, if $[a^{(1)}_{i_1k_1}, ..., a^{(1)}_{i_1k_C}]^T$ is updated to $({}_x\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}{}_x\mathbf{c}^{(1)}_{i_1}$ in the separate factorization of $\mathcal{X}$ and $({}_y\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}{}_y\mathbf{c}^{(1)}_{i_1}$ in that of $\mathcal{Y}$ (see (3.8) for the notations), $[a^{(1)}_{i_1k_1}, ..., a^{(1)}_{i_1k_C}]^T$ is updated to $({}_x\mathbf{B}^{(1)}_{i_1} + {}_y\mathbf{B}^{(1)}_{i_1} + \lambda\mathbf{I}_C)^{-1}({}_x\mathbf{c}^{(1)}_{i_1} + {}_y\mathbf{c}^{(1)}_{i_1})$ in the coupled factorization. This update process minimizes (3.12) in terms of the updated parameters. In a similar way, SALS can be used for coupled tensor factorization problems with more than two tensors sharing two or more modes. It is also true for CDTF, a special case of SALS.

Bias Model

The PARAFAC decomposition (Section 3.2.2) captures interactions among modes that lead to different entry values in $\mathcal{X}$. In recommendation problems, however, much of the variation in entry values (i.e., rating values) is explained by each mode solely. For this reason, Zhou et al. [90] added bias terms, which capture the effect of each mode, in their matrix factorization model. Likewise, we can add bias vectors $\{\mathbf{b}^{(n)} \in \mathbb{R}^{I_n} | 1 \leq n \leq N\}$ to (3.1) which leads to the following loss function:

$$L(\mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}, \mathbf{b}^{(1)}, ..., \mathbf{b}^{(N)}) = \\ \sum_{(i_1,...,i_N)\in\Omega} \left(x_{i_1...i_N} - \mu - \sum_{n=1}^{N} b^{(n)}_{i_n} - \sum_{k=1}^{K}\prod_{n=1}^{N} a^{(n)}_{i_nk} \right)^2 + \lambda\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 + \lambda\sum_{n=1}^{N}\|\mathbf{b}^{(n)}\|_2^2 \quad (3.13)$$

where μ denotes the average entry value of $\mathcal{X}$.

With (3.13), each $(i_1, ...i_N)$th entry of the residual tensor $\mathcal{R}$ is redefined as $r_{i_1...i_n} = x_{i_1...i_N} - \mu - \sum_{n=1}^{N} b^{(n)}_{i_n} - \sum_{k=1}^{K}\prod_{n=1}^{N} a^{(n)}_{i_nk}$. At each outer iteration, we first update factor matrices by either CDTF

Algorithm 8: Parameter update in CDTF without local disk caching

Given : n, k, ${}_mS_n$ for all m, $\mathbf{a}^{(l)}_{*k}$ for all l
Input : $\hat{\mathcal{R}}$
Output: updated $a^{(n)}_{*k}$

```
Map(Key k, Value v)
begin
    ((i_1, ..., i_N), r̂_{i_1...i_N}) ← v
    find m where i_n ∈ _mS_n
    emit < (m, i_n), ((i_1, ..., i_N), r̂_{i_1...i_N}) >
Partitioner(Key k, Value v)
begin
    (m, i_n) ← k
    assign < k, v > to machine m
Reduce(Key k, Value v[1..r])
begin
    (m, i_n) ← k
    Ω^(n)_{i_n} entries of R̂ ← v
    update and emit a^(n)_{i_n k}
```

or SALS and then update bias vectors. Each parameter $b^{(n)}_{in}$ is updated by the following rule which minimizes (3.13) in terms of the updated parameter:

$$b^{(n)}_{i_n} \leftarrow \frac{\sum_{(i_1,...,i_N)\in\Omega^{(n)}_{i_n}} (r_{i_1...i_N} + b^{(n)}_{i_n})}{\lambda + |\Omega^{(n)}_{i_n}|}.$$

After updating $b^{(n)}_{i_n}$, we update the entries of $\mathcal{R}$ in $\{r_{i_1...i_N} | (i_1, ..., i_N) \in \Omega^{(n)}_{i_n}\}$ by the following rule:

$$r_{i_1...i_N} \leftarrow r_{i_1...i_N} + (b^{(n)}_{i_n})^{old} - b^{(n)}_{i_n}$$

where $(b^{(n)}_{i_n})^{old}$ is the parameter value before the update.

3.4 Optimization on MapReduce

In this section, we describe the optimization techniques used to implement CDTF and SALS on MapReduce, which is one of the most widely used distributed platforms. Theses techniques can also be applied to diverse distributed algorithms, including ALS, FlexiFaCT, and PSGD.

3.4.1 Local Disk Caching

The typical MapReduce implementations of CDTF and SALS without local disk caching run each parallel step as a separate MapReduce job. Algorithm 8 describes the MapReduce implementations of parameter update (update of $a^{(n)}_{*k}$); $\mathcal{R}$ update and $\hat{\mathcal{R}}$ computation can be implemented similarly as separate jobs. In this implementation, the updated parameters are stored in the distributed file system and read in the next job. Since CDTF repeats both the $\mathcal{R}$ update and the $\hat{\mathcal{R}}$ computation K times and

Algorithm 9: Data distribution in CDTF and SALS with local disk caching

Input : $\mathcal{X}$, ${}_mS_n$ for all m and n

Output: ${}_m\Omega^{(n)}$ entries of $\mathcal{R}(=\mathcal{X})$ for all m and n

```
Map(Key k, Value v)
begin
    ((i_1, ..., i_N), x_{i_1...i_N}) ← v
    for n = 1,...,N do
        find m where i_n ∈ _mS_n
        emit < (m, n), ((i_1, ..., i_N), x_{i_1...i_N}) >
Partitioner(Key k, Value v)
begin
    (m, n) ← k
    assign < k, v > to machine m
Reduce(Key k, Value v[1..|v|])
begin
    (m, n) ← k
    create a file on the local disk to cache _mΩ^(n) entries of R
    foreach ((i_1, ..., i_N), x_{i_1...i_N}) ∈ v do
        write ((i_1, ..., i_N), x_{i_1...i_N}) to the file
```

the parameter update $KT_{in}N$ times at every outer iteration, this implementation repeats distributing $\mathcal{R}$ or $\hat{\mathcal{R}}$ across machines (the mapper stage of Algorithm 8) $T_{out}K(T_{in}N+2)$ times ($T_{out}K$ times less if the entries of $\hat{\mathcal{R}}$ are computed while computing (3.5) and (3.6)). Likewise, the typical MapReduce implementations of SALS repeat distributing $\mathcal{R}$ or $\hat{\mathcal{R}}$ $T_{out}K(T_{in}N+2)/C$ times.

Our implementation avoids these unnecessary repetitions by caching data to local disk once they are distributed. In our implementations of CDTF and SALS with the local disk caching, $\mathcal{X}$ entries are distributed across machines and cached in the local disk during the map and reduce stage (Algorithm 9); and the rest parts of CDTF and SALS run in the close stage (cleanup stage in Hadoop) using the cached data. Our implementation streams the cached data from the local disk instead of distributing entire $\mathcal{R}$ or $\hat{\mathcal{R}}$ from the distributed file system when updating factor matrices. For example, ${}_m\Omega^{(n)}$ entries of $\hat{\mathcal{R}}$ are streamed from the local disk when the columns of $\mathbf{A}^{(n)}$ are updated. The effect of this local disk caching on the actual running time is described in Section 3.5.5.

3.4.2 Direct Communication

In MapReduce, it is generally assumed that reducers run independently and do not communicate directly with each other. However, CDTF and SALS require updated parameters to be broadcast to other reducers. We circumvent this problem by adapting the direct communication method used in FlexiFaCT [14]. Each reducer writes the parameters that it updated to the distributed file system, and the other reducers read these parameters from the distributed file system.

3.4.3 Greedy Row Assignment

The greedy row assignment algorithm, explained in Section 12, can also be implemented on MapReduce as follows. We assume that $\mathcal{X}$ is stored on the distributed file system. At the first stage, $|\Omega^{(n)}_{i_n}|$ for all n and i_n is computed. Specifically, mappers output $< (n, i_n), 1 >$ for all n for each entry $x_{i_1 \dots i_N}$, and reducers output $< (n, i_n), |\Omega^{(n)}_{i_n}| >$ for all n and i_n by counting the number of values for each key. At the second stage, the outputs are aggregated to a single reducer which runs the rest of Algorithm 7. The effect of greedy row assignment on the actual running time is described in Section 3.5.5.

3.5 Experiments

To evaluate CDTF and SALS, we designed and conducted experiments to answer the following questions:

- **Q1: Data scalability (Section 3.5.2).** How do CDTF, SALS, and their competitors scale with regard to the following properties of an input tensor: dimension, the number of observations, mode length, and rank?
- **Q2: Machine scalability (Section 3.5.3).** How do CDTF, SALS, and their competitors scale with regard to the number of machines?
- **Q3: Convergence (Section 3.5.4).** How quickly and accurately do CDTF, SALS, and their competitors factorize real-world tensors?
- **Q4: Optimization (Section 3.5.5).** How much do local disk caching and greedy row assignment improve the speed of CDTF and SALS? Can these optimization techniques be applied to other methods?
- **Q5: Effects of $\mathbf{T_{in}}$ (Section 3.5.6)** How do different numbers of inner iterations (T_{in}) affect the convergence of CDTF?
- **Q6: Effects of C (Section 3.5.7)** How do different numbers of columns updated at a time (C) affect the convergence of SALS?

All experiments are focused on distributed methods, which are the most suitable to achieve our purpose of handling large-scale data. We compared our methods with ALS, FlexiFaCT, and PSGD, all of which can be parallelized in distributed environments as explained in Section 3.2.3. Serial methods (e.g., [1, 78]) and methods not applicable to partially observable tensors (e.g., [38, 42]) were not considered.

3.5.1 Experimental Settings

We ran experiments on a 40-node Hadoop cluster. Each node had an Intel Xeon E5620 2.4GHz CPU. The maximum heap size per reducer was set to 8GB.

We used both synthetic (Table 3.4) and real-world (Table 3.5) datasets most of which are available at `http://koreaskj.snucse.org/programs/`. Synthetic tensors were created by the procedure used in [62] to create Jumbo dataset, and the real-world tensor data are preprocessed as follows:

- **Movielens$_4$[1]:** Movie rating data from MovieLens, an online movie recommender service. We converted them into a four-dimensional tensor where the third and fourth modes correspond to (year, month) and hour-of-day when the movie was rated, respectively. The rates range from 1 to 5.

[1] http://grouplens.org/datasets/movielens

Table 3.4: Scale of synthetic datasets. B: billion, M: million, K: thousand. The length of every mode is equal to I.

	S1	**S2** (default)	**S3**	**S4**
N	2	3	4	5
I	300K	1M	3M	10M
$\|\Omega\|$	30M	100M	300M	1B
K	30	100	300	1K

Table 3.5: Summary of real-world datasets.

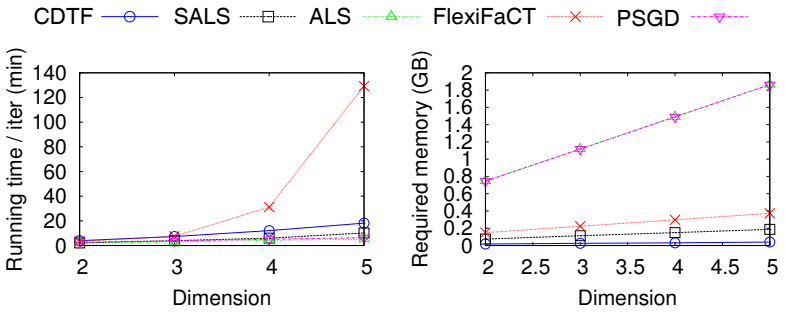
	Movielens$_4$	**Netflix**$_3$	**Yahoo-music**$_4$
N	4	3	4
I_1	71,567	2,649,429	1,000,990
I_2	65,133	17,770	624,961
I_3	169	74	133
I_4	24	-	24
$\|\Omega\|$	9,301,274	99,072,112	252,800,275
$\|\Omega\|_{test}$	698,780	1,408,395	4,003,960
K	20	40	80
λ	0.01	0.02	1.0
η_0	0.01	0.01	10^{-5} (FLEXIFACT), 10^{-4} (PSGD)

- **Netflix**$_3$[2]**:** Movie rating data used in Netflix prize. We regarded them as a three-dimensional tensor where the third mode corresponds to (year, month) when the movie was rated. The rates range from 1 to 5.
- **Yahoo-music**$_4$[3]**:** Music rating data used in KDD CUP 2011. We converted them into a four-dimensional tensor in the same way as we did for Movielens$_4$. Since the exact year and month are not provided, we used the values obtained by dividing the provided data (the number of days passed from an unrevealed date) by 30. The rates range from 0 to 100.

All the methods were implemented in Java with Hadoop 1.0.3. The local disk caching, the direct communication, and the greedy row assignment, all of which are explained in Section 3.4, were applied to all the methods if possible. All our implementations used weighted L_2 regularization as explained in Section 3.3.5. For CDTF and SALS, T_{in} was set to 1 and C was set to 10, unless otherwise stated. The learning rate of FLEXIFACT and PSGD at tth iteration was set to $2\eta_0/(1+t)$, which follows the open-sourced FLEXIFACT (http://alexbeutel.com/l/flexifact/). The number of reducers was set to 5 for FLEXIFACT, 20 for PSGD, and 40 for the other methods, each of which leaded to the best performance on the machine scalability test in Section 3.5.3.

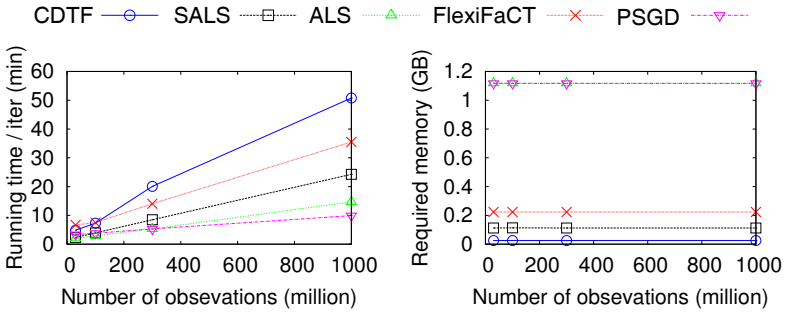
[2]http://www.netflixprize.com

[3]http://webscope.sandbox.yahoo.com/catalog.php?datatype=c



(a) Running time (b) Memory requirements

Figure 3.4: Scalability w.r.t. dimension. FLEXIFACT was not scalable with dimension because of its communication cost, which increases exponentially with dimension. In contrast, the other methods, including CDTF and SALS, were scalable with dimension.



(a) Running time (b) Memory requirements

Figure 3.5: Scalability w.r.t. the number of observations. All methods, including CDTF and SALS, were scalable with the number of observable entries.

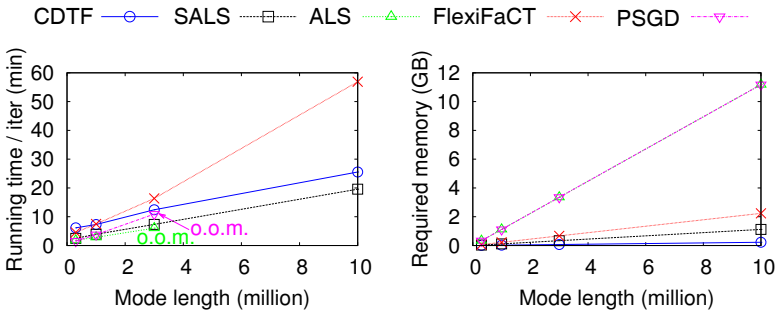
3.5.2 Data Scalability

Scalability with Each Factor (Figures 3.4∼3.7)

We measured the scalability of CDTF, SALS, and the competitors with regard to the dimension, number of observations, mode length, and rank of an input tensor. When measuring the scalability with regard to a factor, the factor was scaled up from S1 to S4 while all other factors were fixed at S2 in Table 3.4.

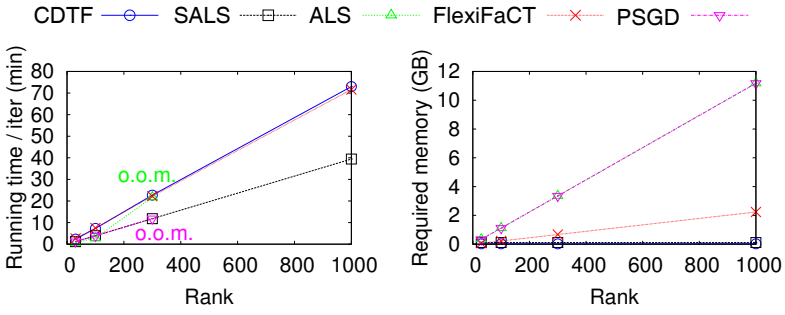
As seen in Figure 3.4, FLEXIFACT did not scale with the dimension because of its communication cost, which increases exponentially with the dimension. ALS and PSGD were not scalable with the mode length and the rank due to their high memory requirements as Figures 3.6 and 3.7 show. They required up to 11.2GB, which is 48× of 234MB that CDTF required and 10× of 1,147MB that SALS required. Moreover, the running time of ALS increased rapidly with rank owing to its cubically increasing computational cost.

Only SALS and CDTF were scalable with all the factors as summarized in Table 3.1. Their running times increased linearly with all the factors except the dimension, with which they increased slightly faster due to the quadratically increasing computational cost.



(a) Running time (b) Memory requirements

Figure 3.6: Scalability w.r.t. mode length. o.o.m.: out of memory. ALS and PSGD were not scalable with mode length because of their high memory requirements. In contrast, CDTF, SALS, and FlexiFaCT were scalable with mode length.



(a) Running time (b) Memory requirements

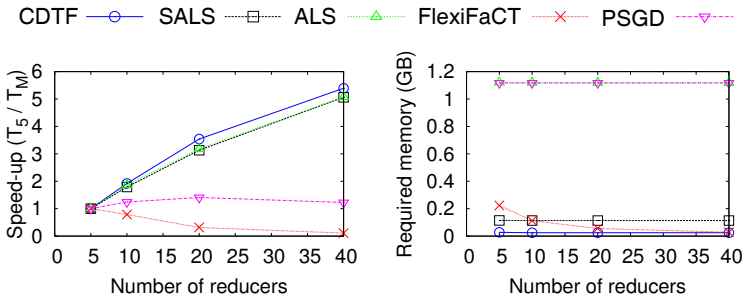
Figure 3.7: Scalability w.r.t. rank. o.o.m.: out of memory. ALS and PSGD were not scalable with rank because of their high memory requirements. In contrast, CDTF, SALS, and FlexiFaCT were scalable with rank.

Overall scalability (Figure 3.1 in Section 3.1)

We measured the scalability of the methods by scaling up all the factors simultaneously from S1 to S4. The scalability of ALS and PSGD, and FlexiFaCT with five machines was limited owing to their high memory requirements. ALS and PSGD required about 186GB to handle S4, which is 493× of 387MB that CDTF required and 100× of 1,912MB that SALS required. FlexiFaCT with 40 machines did not scale over S2 due to its rapidly increasing communication cost. Only CDTF and SALS scaled up to S4, and there was a trade-off between them: CDTF was more memory-efficient, and SALS ran faster.

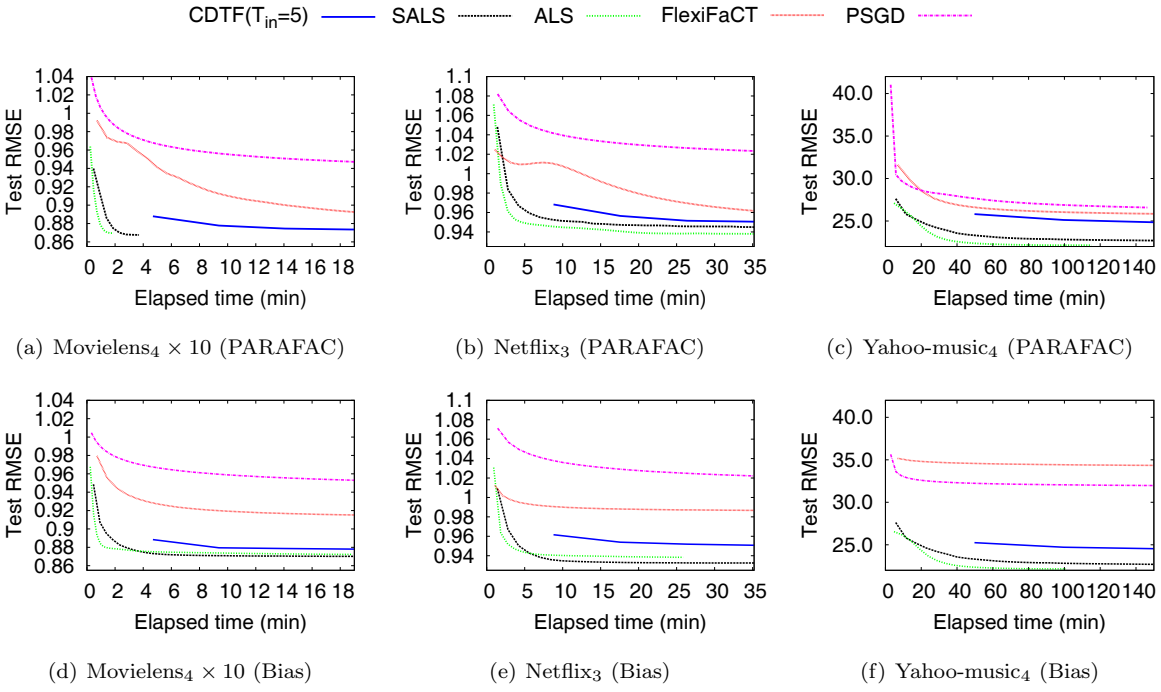
3.5.3 Machine Scalability (Figure 3.8)

We measured the speed-ups (T_5/T_M where T_M is the running time with M reducers) and memory requirements of the methods on the S2 scale dataset by increasing the number of reducers. The speed-ups of CDTF, SALS, and ALS increased linearly at the beginning and then flattened out slowly owing to their fixed communication cost which does not depend on the number of reducers (see Table 3.3). The speed-up of PSGD flattened out fast, and PSGD even slightly slowed down at 40 reducers because of the increased overhead. FlexiFaCT slowed down as the number of reducers increased because of its rapidly increasing communication cost. The memory requirements of FlexiFaCT decreased as the number of



(a) Speed-up (b) Memory requirements

Figure 3.8: Machine scalability on the S2 scale dataset. FLEXIFACT and PSGD were not scalable with the number of machines. Especially, FLEXIFACT slowed down as the number of reducers increased because of its rapidly increasing communication cost. In contrast, the speed-ups of CDTF, SALS, and ALS increased almost linearly at the beginning and then flattened out gradually.



(a) Movielens$_4$ × 10 (PARAFAC) (b) Netflix$_3$ (PARAFAC) (c) Yahoo-music$_4$ (PARAFAC)

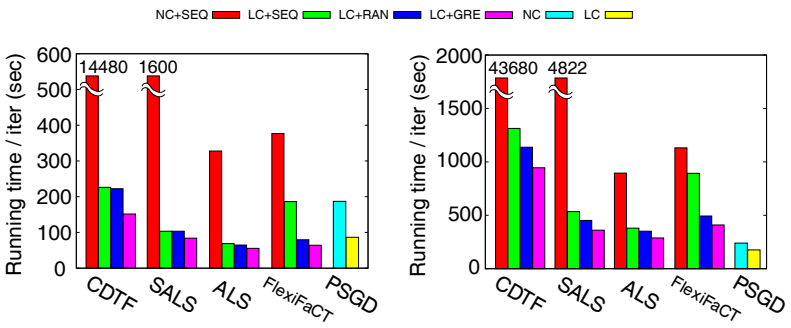
(d) Movielens$_4$ × 10 (Bias) (e) Netflix$_3$ (Bias) (f) Yahoo-music$_4$ (Bias)

Figure 3.9: Convergence speed on real-world datasets. (a)∼(c) show the results with the PARAFAC model, and (d)∼(e) show the results with the bias model (Section 3.3.5). SALS and ALS converged fastest to the best solution, and CDTF followed them. CDTF and SALS, however, have much better scalability than ALS as shown in Section 3.5.2.

reducers increased, while the other methods required the fixed amounts of memory.

3.5.4 Convergence (Figure 3.9)

We compared how quickly and accurately each method factorizes real-world tensors using the bias model (Section 3.3.5) as well as the PARAFAC model (Section 3.2.2). Accuracies were calculated at each iteration by root mean square error (RMSE) on a held-out test set, which is commonly used by recommendation systems. Table 3.5 lists K, λ, and η_0 values used for each dataset. They were determined



(a) Netflix$_3$ (b) Yahoo-music$_4$

Figure 3.10: Effects of optimization techniques on running times. NC: no caching, LC: local disk caching, SEQ: sequential row assignment[4], RAN: random row assignment, GRE: greedy row assignment. Our proposed optimization techniques (LC+GRE) significantly speeded up CDTF, SALS, and also their competitors.

by cross validation. Since the Movielens$_4$ dataset (183MB) is too small to run on Hadoop, we increased its size by duplicating each user 10 times. Owing to the non-convexity of (3.1), each algorithm may converge to local minima with different accuracies.

In all datasets, SALS was comparable with ALS, which converged fastest to the best solution, and CDTF followed them. CDTF and SALS, however, have much better scalability than ALS as shown in Section 3.5.2. PSGD converged the slowest to the worst solution because of the non-identifiability of (3.1) [31].

3.5.5 Optimization (Figure 3.10)

We measured how our proposed optimization techniques, the local disk caching and the greedy row assignment, affect the running time of CDTF, SALS, and the competitors on real-world datasets. The direct communication method explained in Section 3.4.2 was applied to all the implementations if necessary.

The local disk caching speeded up CDTF up to 65.7×, SALS up to 15.5×, and the competitors up to 4.8×. The speed-ups of SALS and CDTF were the most significant because of their highly iterative nature. Additionally, the greedy row assignment speeded up CDTF up to 1.5×; SALS up to 1.3×; and the competitors up to 1.2× compared with the second best one. It is not applicable to PSGD, which does not distribute parameters row by row.

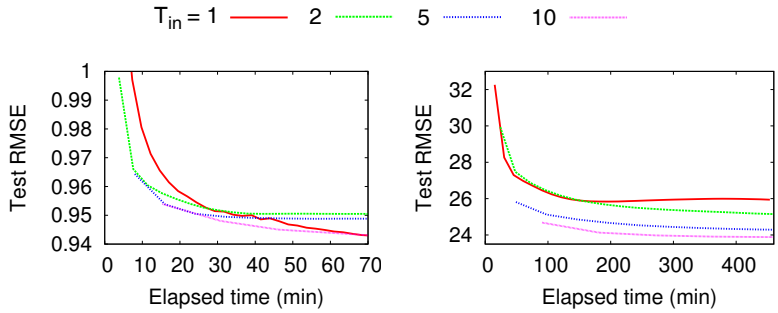
3.5.6 Effects of Inner Iterations (Figure 3.11)

We compared the convergence properties of CDTF with different T_{in} values. As T_{in} increased, CDTF tended to converge more stably to better solutions (with lower test RMSE). However, there was an exception; $T_{in} = 1$ in the Netflix$_3$ dataset converged to the best solution.

3.5.7 Effects of the Number of Columns Updated at a Time (Figure 3.12)

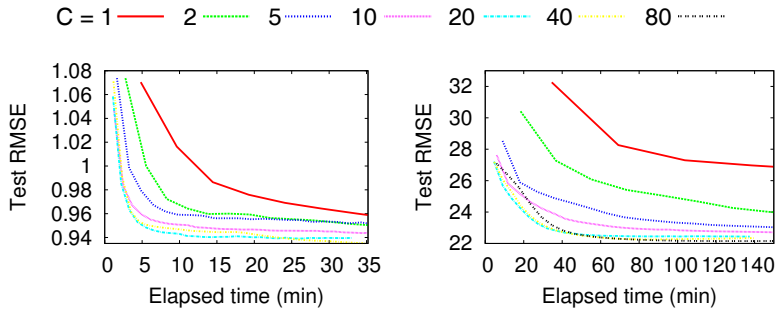
We compared the convergence properties of SALS with different C values. T_{in} was fixed to one in all cases. As C increased, SALS tended to converge faster to better solutions (with lower test RMSE) although it required more memory as explained in Theorem 8. With C above 20, however, convergence

[4] ${}_mS_n = \{i_n \in \mathbb{N} | \frac{I_n \times (m-1)}{M} < i_n \leq \frac{I_n \times m}{M}\}$



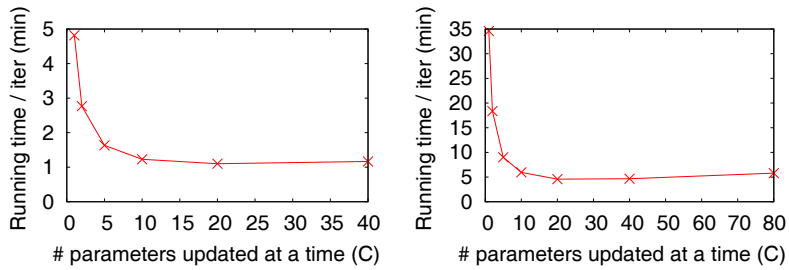
(a) Netflix$_3$ (b) Yahoo-music$_4$

Figure 3.11: Effects of inner iterations (T_{in}) on the convergence of CDTF. CDTF tended to converge stably to better solutions as T_{in} increased.



(a) Netflix$_3$ (b) Yahoo-music$_4$

Figure 3.12: Effects of the number of columns updated at a time (C) on the convergence of SALS. SALS tended to converge faster to better solutions as C increased. However, its convergence speed decreased at C above 20.



(a) Netflix$_3$ (b) Yahoo-music$_4$

Figure 3.13: Effects of the number of columns updated at a time (C) on the running time of SALS. Running time per iteration decreased until $C = 20$, then started to increase.

speed started to decrease. This trend is related to the running time per iteration, which showed the same trend as seen in Figure 3.13. As C increases, the amount of disk I/O declines since it depends on the number of times that the entries of $\mathcal{R}$ or $\hat{\mathcal{R}}$ are streamed from disk, which is inversely proportional to C. Conversely, computational cost increases quadratically with regard to C. At low C values, the decrease in the amount of disk I/O was greater and leaded to a downward trend of running time per iteration. The opposite happened at high C values.

3.6 Related Work

Matrix Factorization. Matrix factorization (MF) has been successfully used in many recommender systems [21, 49, 90]. The underlying intuition of MF for recommendation can be found in [49]. Two major approaches for large-scale MF are alternating least squares (ALS) and stochastic gradient descent (SGD). ALS is inherently parallelizable [90] but has high memory requirements and computational cost [86]. Efforts were made to parallelize SGD [31, 58, 92], including distributed stochastic gradient descent (DSGD) [31] which divides a matrix into disjoint blocks and processes them simultaneously. Recently, coordinate descent were also applied to large-scale MF [85, 86].

Tensor Factorization. Comprehensive survey on tensor factorization can be found in [48, 68]. The factorization of partially observable tensors has been used in many fields such as computer vision [55], chemometrics [78], social network analysis [26], and Web search [75]. Recently, it also has been used in recommender systems to utilize additional contextual information such as time and location [46, 59, 89]. For the PARAFAC decomposition, distributed algorithms [14] as well as serial ones [1, 78] were proposed.

MapReduce and Hadoop. MapReduce [24] is a distributed computing framework for processing Web-scale data, and Hadoop is a widely-used open-source implementation of MapReduce. A variety of data mining algorithms, including matrix and tensor factorization [31, 14], have been implemented on MapReduce. Nevertheless, its limited computational model and inefficiency because of the repeated distribution of data have been pointed out as problems [86, 77, 17, 27]. To tackle these problems, extensions of MapReduce [17, 27], as well as systems with different programming models [56, 87] have been proposed. In contrast to these approaches, our approach, which includes the local disk caching and the direct communication method, does not require modification to MapReduce.

3.7 Conclusion

In this chapter, we propose two distributed algorithms for high-dimensional and large-scale tensor factorization: CDTF and SALS. They decompose a tensor with a given rank through a series of lower rank factorization. CDTF and SALS have advantages over each other in terms of memory usage and convergence speed, respectively. We compared our methods with other state-of-the-art distributed methods both analytically and experimentally. Only CDTF and SALS are scalable with all aspects of data (i.e., dimension, the number of observable entries, mode length, and rank) and successfully factorized a 5-dimensional tensor with 1B observable entries, 10M mode length, and 1K rank with up to 493× less memory requirement. We implemented our methods on top of MapReduce with two widely-applicable optimization techniques, which accelerated not only CDTF and SALS (up to 98.2×), but also the competitors (up to 5.9×).

Chapter 4. Multi-stage BSP Model for Distributed Machine Learning

The MapReduce programming model provides a simple way to express parallel algorithms and has been used in many application. However, its limited expressive power, especially in terms of iteration and communication methods, results in inefficiency in distributed machine learning and data mining cases. To overcome this inefficiency, enhanced MapReduce models and models based on graph structure were proposed. Even with these models, though, it is difficult to fully utilize distributed environments where computing resources and intermediate data can be retained and shared among multiple parallel steps that together compose a machine learning or data mining job.

Taking such environments into account, we propose a multi-stage BSP programming model where a job consists of multiple stages among which computing resources and intermediate data can be shared. Moreover, each stage supports iteration and rich communication methods. We show the expressiveness and convenience of the model by implementing three distributed machine learning algorithms using the model. In addition, we briefly explain how the model can be implemented on top of REEF and enjoy in-memory processing as well as resource retainment provided by REEF.

4.1 Introduction and Related Work

The MapReduce programming model [24] and distributed systems supporting the model provide an easy way to implement and run distributed algorithms. By simply implementing two functions, Map and Reduce, various algorithms in many domains have enjoyed massive parallelism provided by computer clusters. When in comes to machine learning and data mining algorithms, however, the excessive simplicity of the MapReduce model results in inefficiency. Although the MapReduce model assumes a single parallel step consisting of Map and Reduce, machine learning algorithms consist of several parallel steps executed repeatedly. Likewise, the MapReduce model allows only one way of communication among machines (i.e., shuffle), while machine learning algorithms make good use of various communication methods for simplicity and efficiency. Due to these limitations, many distributed machine learning algorithms have had to put up with inefficiency or hack distributed systems for their extended usage [4, 39, 74]

To solve this problem, extended MapReduce models have been proposed by some systems. HaLoop [17] supports a programming model which can express a MapReduce job executed repeatedly until termination conditions are met. Termination conditions are expressed using fixed points and the maximum number of iterations. The programming model of Twister [27] also supports the iterative execution of a MapReduce job. The model also supports broadcast among machines in addition to shuffle. These extended MapReduce models enable the simpler programming and more efficient execution of distributed machine learning algorithms.

On the other hand, other systems including Pregel [57], GraphLab [56], and PowerGraph [34] adopt the node-centric model [57] as an alternative of the MapReduce model. In the node-centric model, a job consists of three steps on a graph which are executed repeatedly: (1) computation at each node, (2)

message passing between neighboring nodes, and (3) the aggregation of the messages at each node. These systems show that the simpler expression and more efficient execution of distributed machine learning are possible with the node-centric model compared with the MapReduce model.

The extended MapReduce models and the node-centric model, however, assume that a single parallel step or independent parallel steps are repeated, while many distributed machine learning algorithms consist of a series of tightly related parallel steps. Even when a distributed machine learning algorithm is composed of a single parallel step, it is a part of knowledge discovery process which consists of a series of related parallel steps. Allowing a programming model to express multiple related steps as a single job not only simplifies programming but also provides systems with an opportunity to retain computing resources and intermediate data between parallel steps. By doing so, systems can reduce disk I/O necessary to store intermediate data and redistribute the data among machines.

In this chapter, we propose a multi-stage BSP programming model. In the model, a job is composed of several stages. Each stage can be run iteratively and supports rich communication methods: broadcast, scatter, gather, reduce, and shuffle. In addition, each stage can hand intermediate results over the following stage using key-value stores. To prove the expressiveness of the model and the conciseness of programs expressed in the model, we implement three representative machine learning algorithms (K-means, Linear Regression, and ALS) using the model. In addition, we briefly describe how such model can be implemented on top of REEF [82]. REEF enables our model to utilize in-memory processing and retain computing resources and intermediate results among stages.

Our programming model and its runtime are being developed as a part of the Apache REEF project (`http://reef.incubator.apache.org/`), an open-source project. The rest of the chapter is organized as follows. Section 4.2 presents preliminaries on BSP and REEF. Our proposed multi-stage BSP model is described in Section 4.3. Applications in distributed machine learning are provided in Section 4.4, followed by the implementation of the model on REEF in Section 4.5. We make conclusion in Section 4.6.

4.2 Preliminaries

In this section, we describe the preliminaries on Bulk Synchronous Parallel (BSP) and Retainable Evaluator Execution Framework (REEF).

4.2.1 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel (BSP) is a computational model for parallel computing. In the model, a parallel program is run across multiple processes (within a machine or across machines) as a sequence of supersteps. Each superstep consists of the following three components:

- **Local Computation.** Each process performs local computations using data stored in local storages.
- **Global Communication.** The processes exchange data with each other.
- **Barrier Synchronization.** The processes reaching a barrier wait until all other processes reach the same barrier.

The first and second components do not need to be ordered in time. Due to its simplicity and expressiveness, BSP has been adopted in many distributed systems [10, 72]. Also in our multi-stage model, explained in Section 4.3, each stage follows the BSP model.

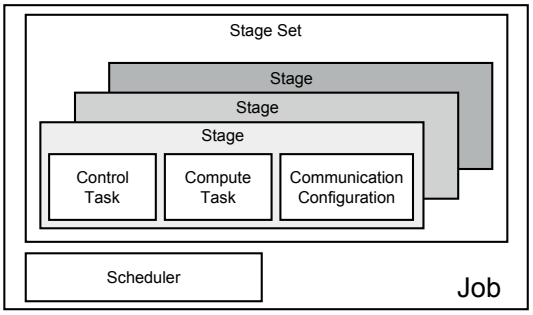


Figure 4.1: Components of our multi-stage BSP model

4.2.2 Retainable Evaluator Execution Framework (REEF)

Retainable Evaluator Execution Framework (REEF) [82] runs on top of resource managers and enables easy implementations of a range of computational models in distributed environments. Especially, REEF facilitates retaining computing resources across tasks and jobs. The key abstractions upon which REEF is structured are as follows:

- **Driver.** User-defined control logic that requests resource managers for computing resources, initiates evaluators on the resources, and assigns tasks and services to the evaluators.
- **Task.** User-defined computation logic that runs on the assigned evaluator. Tasks can access the data and functions provided by services.
- **Evaluator.** Runtime for tasks. An evaluator executes one task at a time but many tasks throughout its life time. Evaluators also retain services.
- **Service.** Daemons and objects retained in an evaluator across tasks run on the evaluator. Sharing services among tasks reduces overhead to initiate tasks and enables handing over intermediate data.

Our multi-stage BSP model, explained in Section 4.3, is being developed on top of REEF. Our model make a good use of retained computing resources and data for efficiency. In addition, our model utilizes building blocks provided by REEF such as the data loading service and the group communication service.

4.3 Multi-stage BSP Model

In this section, we describe our proposed multi-stage BSP model. We first present the components of the model then present the control flow of the model both on job level and stage level.

4.3.1 Components

Our multi-stage BSP model consists of the components depicted in Figure 4.1

Job

A *job* is a unit of computation consisting of one or more *stages* and a *scheduler*. The *scheduler* decides the execution order of the *stages*. A distributed machine learning algorithm consisting of several parallel steps or the entire knowledge discovery process can be modeled as a *job*. We assume the situation where computing resources and intermediate data are retained and shared among *stages* composing a *job*.

Algorithm 10: Interface for Control Tasks

```
1  interface control_task {
2     void initialize();
3     void control();
4     boolean is_terminate();
5     int close();
6     List<S> send_scatter(); /*optional*/
7     B send_broadcast(); /*optional*/
8     void receive_gather(List<G>); /*optional*/
9     void receive_reduce(R); /*optional*/
10 }
```

Scheduler

A *scheduler* decides the execution order of *stages* composing a *job*. The *scheduler* is a user-defined function that takes the current *stage* and the output of the current *stage* as inputs and returns the next *stage*.

Stage

A *stage* corresponds to a parallel step. A single MapReduce job in the MapReduce model corresponds to a *stage* in our model. In our model, multiple *stages* compose a *job*. Each *stage* consists of a *compute task*, a *control task*, and *communication configuration* which together describe the three components of a BSP model, explained in Section 4.2.1.

Control Task

A *control task* class is a user-defined class whose interface is described in Algorithm 10. A *control task* (instance) uniquely exists and its functions are executed on a single machine (or process). A *control task*, which is a part of a *stage*, (1) specifies execution flow within a *stage*, (2) receives data from *compute tasks* (3) processes the data, and (4) sends the processed data back to *compute tasks*.

The details of the functions composing a *control task* are as follows:

- **initialize**. This function initializes the control task before starting the iteration phase (see Section 4.3.2). Especially, it loads retained intermediate data if exist.
- **control.** This function, which is a part of the iteration phase, processes data received from compute tasks.
- **is_terminate.** This function, which is a part of the iteration phase, decides whether to stop the iteration phase.
- **close**. This function cleans up the control task after the iteration phase terminates. Especially, it stores intermediate data that should be retained. The return value of this function can be used by the scheduler to determine the next stage.
- **send_scatter** (optional). This function, which is a part of the iteration phase, returns data to scatter over compute tasks.
- **send_broadcast** (optional). This function, which is a part of the iteration phase, returns data to broadcast over compute tasks.

Algorithm 11: Interface for Compute Tasks

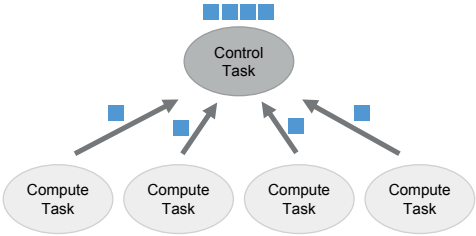
```
1  interface compute_task {
2     void initialize();
3     void compute();
4     int close();
5     List<G> send_gather(); /*optional*/
6     R send_reduce(); /*optional*/
7     List<K,V> send_shuffle(); /*optional*/
8     void receive_shuffle(List<K,List<V>>); /*optional*/
9     void receive_scatter(List<S>); /*optional*/
10    void receive_broadcast(B); /*optional*/
11 }
```

- **receive_gather** (optional). This function, which is a part of the iteration phase, receives data gathered from compute tasks.
- **receive_reducer** (optional). This function, which is a part of the iteration phase, receives reduced data from compute tasks.

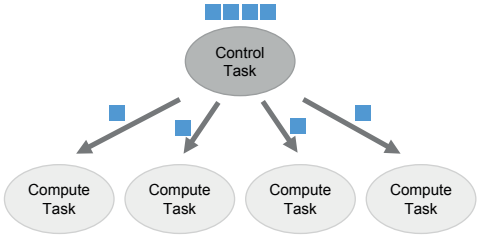
Compute Task

A *compute task* class is a user-defined class whose interface is described in Algorithm 11. Multiple *compute tasks* (instances) are created and their functions are executed on multiple machines (or processes). A *compute task*, which is a part of a *stage*, specifies local computation using data stored in local storages. The details of the functions composing a *compute task* are as follows:

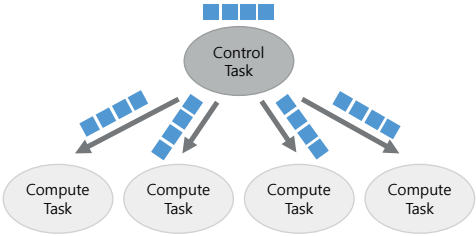
- **initialize**. This function initializes the compute task before starting the iteration phase (see Section 4.3.2). Especially, it loads retained intermediate data if exist.
- **compute.** This function, which is a part of the iteration phase, carries out local computation using data stored in local storages.
- **close**. This function cleans up the compute task after the iteration phase terminates. Especially, it stores intermediate data that should be retained. The return value of this function can be used by the scheduler to determine the next stage.
- **send_gather** (optional). This function, which is a part of the iteration phase, returns data to be sent to the control task.
- **send_broadcast** (optional). This function, which is a part of the iteration phase, returns data to be reduced then sent to the control task.
- **send_shuffle** (optional). This function, which is a part of the iteration phase, returns key-value pairs to be shuffled among compute tasks.
- **receive_shuffle** (optional). This function, which is a part of the iteration phase, receives key-value pairs shuffled among compute tasks.
- **receive_scatter** (optional). This function, which is a part of the iteration phase, receives data scattered by the control task.
- **receive_broadcast** (optional). This function, which is a part of the iteration phase, receives data broadcast by the control task.



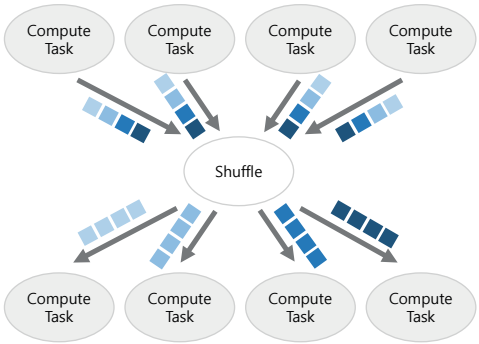
(a) Gather



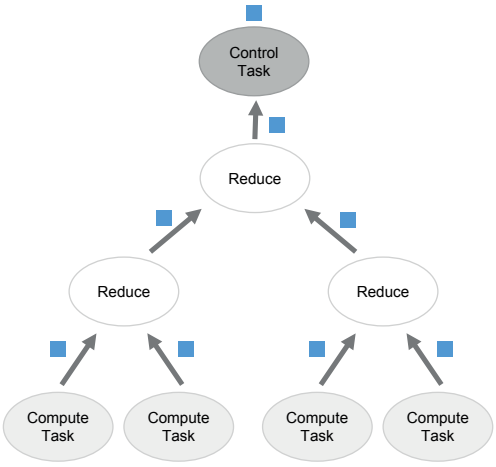
(b) Scatter



(c) Broadcast



(d) Shuffle



(e) Reduce

Figure 4.2: Topologies of the group communication methods supported by our multi-stage BSP model.

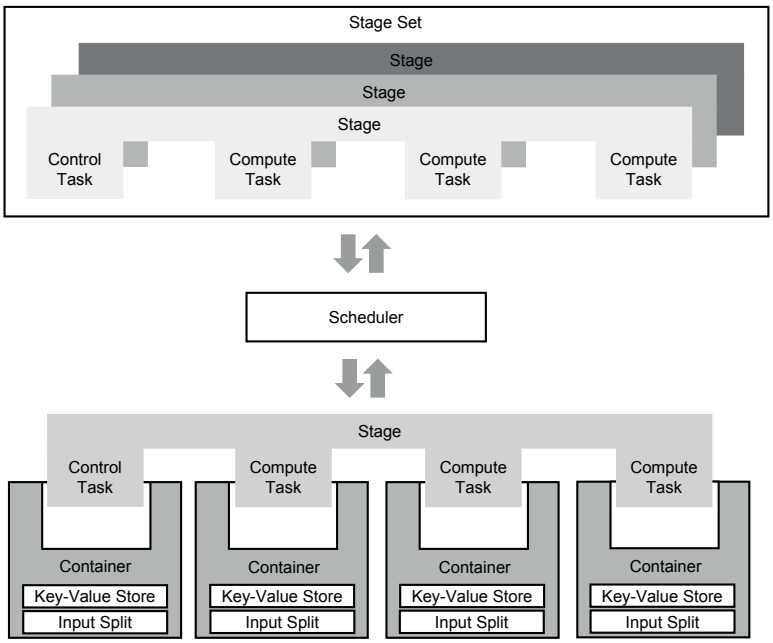


Figure 4.3: Job-level execution flow of our model.

Communication Configuration

Communication configuration specifies communication methods among tasks in a stage. *Communication configuration* includes (1) topologies of communication, (2) types of communicated data, and (3) additional necessary logics (e.g., how to reduce data and how to partition keys). Our model provides the following group communication methods, which are also described in Figure 4.2.

- **Gather.** Data from compute tasks are gathered and sent to the control task.
- **Reduce.** Data from compute tasks are reduced and sent to the control task. A *reduce function* specifies how to reduce data.
- **Scatter.** Data from the control task are scattered over compute tasks.
- **Broadcast.** Data from the control task are broadcast over compute tasks.
- **Shuffle.** List of key-value pairs from the compute tasks are shuffled among compute tasks. Values with the same key are sent to the same machine. A *partition function* specifies which key is sent to which machine.

4.3.2 Execution Flow

Job-level Execution Flow

Figure 4.3 presents the execution flow of our model on job level. We assume that containers, computing resources that our model runs on, are retained among stages until a job terminates. Input data split and intermediate data stored in a key-value store are also retained in the containers. The execution order of stages is decided by the scheduler. The control task of every stage is assigned to the same container (the first container in Figure 4.3) so that intermediate data can be haded over among control tasks.

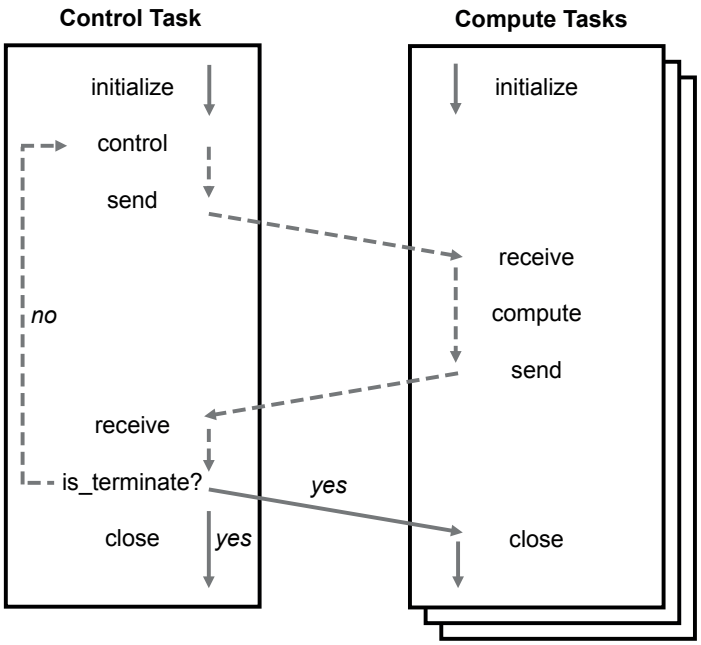


Figure 4.4: Stage-level execution flow of our model.

Table 4.1: Table of symbols for K-means.

Symbol	Definition
$\mathbf{V}$	input vectors
$\mathbf{V}_{split}$	input vectors split to each compute task
k	number of clusters
$\mathbf{c}_c$	centroid vector of the cth cluster
n_c	number of input vectors assigned to the cth cluster
$\mathbf{s}_c$	sum of input vectors assigned to the cth cluster
T	number of compute tasks

Stage-level Execution Flow

Figure 4.4 presents the execution flow of our model on stage level. The execution flow in each task consists of initialization phase (initialize function), iteration phase, and close phase (close function). The dotted arrows in Figure 4.4 shows the iteration phase, which is repeated until termination conditions are met. Notice that the execution flow of compute tasks is also controlled by the control task. Data communication from compute tasks to the control task is synchronous. The execution flow continues to the next function after receiving data from all compute tasks.

4.4 ML Applications

In this section, we implement three widely-used machine learning algorithms using our multi-stage BSP model. By doing so, we show the expressiveness and convenience of our model.

Algorithm 12: Initialization Stage of K-means

Input : Input vectors: $\mathbf{V}$, Number of clusters: k

Output: Initial centroids of clusters: $\mathbf{c}_1, ..., \mathbf{c}_k$

```
1 class control_task {
2    boolean is_terminate() { return true }
3    int close() { store [c_1, ..., c_k] to the local key-value store }
4    void receive_gather(List<Vector> list) { [c_1, ..., c_k] ← list }
5 }
6 class compute_task {
7    void compute() { sample k/T instances from V_split }
8    List<Vector> send_gather() { return sampled vectors }
9 }
```

4.4.1 K-means Clustering

K-means is a classic clustering algorithm still widely used in many fields. Given a set of vectors $\mathbf{V} = \{\mathbf{v}_i\}_{i=1}^n$, the algorithm finds k clusters so that vectors in the same cluster are close, while vectors in different clusters and distant. After initializing the centroids with k randomly sampled vectors, the algorithm repeats the following two phases until convergence:

- **Assignment Phase.** In this phase, each vector is assigned to the closest cluster.
- **Update Phase.** In this phase, the centroid of each cluster is recomputed.

With our multi-stage BSP model, K-means consists of the initialization stage and the clustering stage, which are executed in order by the scheduler. Table 4.1 summarizes the symbols used in the algorithm.

Initialization Stage (Algorithm 12)

Each compute task samples k/T vectors from $\mathbf{V}_{split}$ (*compute*) and sends them to the control task (*send_gather*). The control task gathers them (*receive_gather*) and stores them in the local key-value store (*close*). The entire process is executed only once without repetition (*is_termination*). This stage only uses gather with the *Vector* type; therefore, the fact is specified in the communication configuration.

Clustering Stage (Algorithm 13)

The control task loads the initial centroids (*initialize*) from the local key-value store and broadcasts them over compute tasks (*send_broadcast*). Each compute task assigns the vectors in $\mathbf{V}_{split}$ to their closest clusters, and computes a part of n_c and s_c (*compute*). The parts of n_c and s_c computed by the compute tasks are summed during the reduce communication (*reduce*) and sent to the control task (*send_reduce*). The control task updates the centroids of the clusters from the reduced data (*receive_reduce*) and broadcasts the updated centroids. This process is repeated until the centroids are converged (*is_terminate*).

4.4.2 Linear Regression

Linear regression is a classic regression algorithm which find a relationship between independent variables $\mathbf{X}$ and dependent variable $\mathbf{y}$. Given data $\mathbf{D} = \{y_i, x_{i1}, x_{i2}, ..., x_{ip}\}_{i=1}^n$, the algorithm finds

Algorithm 13: Clustering stage of K-means

Input : Input vectors: $\mathbf{V}$, Number of clusters: k

Output: Centroids of clusters: $\mathbf{c}_1, ..., \mathbf{c}_k$

```
1  class control_task {
2     void initialize() { load [c_1, ..., c_k] from the local key-value store }
3     boolean is_terminate() { return whether [c_1, ..., c_k] is converged or not }
4     List<Vector> send_broadcast() { return [c_1, ..., c_k] }
5     void receive_reduce(List<Int, Vector> list) {
6        foreach (n_c, s_c) ∈ list { c_c ← s_c / n_c }
7     }
8  }
9  class compute_task {
10    void compute() {
11       initialize n_c to 0 and s_c to 0 for all c
12       foreach v ∈ V_split {
13          find the closest cluster c of v
14          n_c ← n_c + 1
15          s_c ← s_c + v
16       }
17    }
18    void receive_broadcast(List<Vector>) list) { [c_1, ..., c_k] ← list }
19    List<Int, Vector> send_reduce() { return [(n_1, s_1), ..., (n_k, s_k)] }
20 }
21 List<Int, Vector> reduce(List<Int, Vector> list1, List<Int, Vector> list2) {
22    [(1n_1, 1s_1), ..., (1n_k, 1s_k)] ← list1
23    [(2n_1, 2s_1), ..., (2n_k, 2s_k)] ← list2
24    return [(1n_1 + 2n_1, 1s_1 + 2s_1), ..., (1n_k + 2n_k, 1s_k + 2s_k)]
25 }
```

Algorithm 14: Stage of Linear Regression

Input : Input data: $\mathbf{D} = (\mathbf{X}, \mathbf{y})$
Output: Coefficient vector : β

```
1  class control_task {
2     void initialize() { initialize β randomly }
3     boolean is_terminate() { return whether β is converged or not }
4     Vector send_broadcast() { return β }
5     void receive_reduce(Vector vector) {
6        s ← vector
7        β ← s/T
8     }
9  }
10 class compute_task {
11    void compute() {
12       foreach (x, y) ∈ D_split { update β using SGD on x and y }
13    }
14    void receive_broadcast(Vector vector) { β ← vector }
15    Vector send_reduce() { return β }
16 }
17 Vector reduce(Vector vector1, Vector vector2) {
18    1β ← vector1
19    2β ← vector2
20    return 1β + 2β
21 }
```

coefficient vector β minimizing $||\mathbf{y} - \mathbf{X}\beta||_2$. Among many methods to compute linear regression, we use Parallel Stochastic Gradient Descent (PSGD) [92], which fits in with distributed environments, for our implementation using the multi-stage BSP model. In PSGD, β updated by each task are averaged and redistributed across tasks at every iteration.

Our implementation of linear regression consists of a single stage depicted in Algorithm 14. The control task initializes β (*initialize*) and broadcasts it over compute tasks (*send_broadcast*). Each task updates β using Stochastic Gradient Descent (SGD) on $\mathbf{D}_{split}$ (*compute*). β updated by the compute tasks are summed during reduce communication (*reduce*) and averaged by the control task (*receive_reduce*). Again, the averaged β is broadcast over compute tasks. This process is repeated until β is converged (*is_terminate*).

4.4.3 Alternating Least Square (ALS)

Alternating Least Square (ALS) [90] is an optimization technique widely used for feature-based collaborative filtering. Let n the number of users and m the number of items. The objective of ALS is to solve the following problem:

- GIVEN: rating matrix $\mathbf{X}(\in \mathbb{R}^{n \times m})$, where x_{ij} is the feedback score of user i about item j, and the number of features d

Table 4.2: Table of symbols for ALS.

Symbol	Definition
n	number of users
m	number of items
d	number of features
$\mathbf{X}(\in \mathbb{R}^{n \times m})$	rating matrix
$\mathbf{X}_{split}$	entries of $\mathbf{X}$ originally split to each compute task
$\mathbf{X}'_{split}$	entries of $\mathbf{X}$ repartitioned to each compute task
$\mathbf{U}(\in \mathbb{R}^{n \times d})$	user feature matrix
$\mathbf{U}_{split}$	entries of $\mathbf{U}$ assigned to each compute task
$\mathbf{V}(\in \mathbb{R}^{m \times d})$	item feature matrix
$\mathbf{V}_{split}$	entries of $\mathbf{V}$ assigned to each compute task
T	number of compute tasks

- FIND: user feature matrix $\mathbf{U}(\in \mathbb{R}^{n \times d})$ and item feature matrix $\mathbf{V}(\in \mathbb{R}^{m \times d})$
- to MINIMIZE: $||\mathbf{X} - \mathbf{U}\mathbf{V}^T||_F^2$

For this purpose, ALS computes $\mathbf{U}$ and $\mathbf{V}$ one at a time while fixing the other, and repeats this computation until $\mathbf{U}$ and $\mathbf{V}$ are converged. In distributed environments, $\mathbf{U}$ and $\mathbf{V}$ are split into parts and computed in parallel by multiple machines. With our multi-stage BSP model, ALS consists of the repartitioning stage, the $\mathbf{U}$ stage, and the $\mathbf{V}$ stage. The scheduler runs the repartitioning stage once then runs the $\mathbf{U}$ stage and the $\mathbf{V}$ stage alternatively until $\mathbf{U}$ and $\mathbf{V}$ are converged. Table 4.2 summarizes the symbols used in the algorithm.

Repartitioning Stage (Algorithm 15)

In this stage, we repartition $\mathbf{X}$ entries so that entires in the same row and entries in the same column are assigned to the same container. For that purpose, compute tasks shuffle $\mathbf{X}$ entries (*send_shuffle* and *receive_shuffle*) using the row number and the column number as a key of each entry (*compute*). The modulo function ($f(k) = k\%T$) can be used as the partition function of the shuffle communication. Repartitioned data are stored in the local key-value store (*close*).

U Stage (Algorithm 16)

In this stage, we update the user feature matrix $\mathbf{U}$ while fixing the item feature matrix $\mathbf{V}$. If this is the first time that the $\mathbf{U}$ stage is executed, the control task randomly initializes $\mathbf{V}$. Otherwise, the control task load $\mathbf{V}$ computed in the previous stage from the local key-value store (*initialize*). Then, $\mathbf{V}$ is broadcast over compute tasks (*send_broadcast* and *receive_broadcast*). Each compute task loads $\mathbf{X}'_{split}$ from the local key-value store (*initialize*) and computes $\mathbf{U}_{split}$ using $\mathbf{X}'_{split}$ and $\mathbf{V}$ (*compute*). Computed $\mathbf{U}_{split}$ are sent to the control task (*send_gather*), and entire $\mathbf{U}$ is gathered (*receive_gather*) and stored in the local key-value store (*close*) by the compute task.

Algorithm 15: Repartitioning stage of ALS

Input : Rating matrix: $\mathbf{X}$

Output: Partitioned rating matrix : $\mathbf{X}'_{split}$ for each container

```
1  class control_task {
2     boolean is_terminate() { return true }
3  }
4  class compute_task {
5     void compute() {
6        key-value list ← []
7        foreach (i, j, x_ij) ∈ X_split {
8           add (i, (i, j, x_ij)) and (j, (i, j, x_ij)) to key-value list
9        }
10    }
11    List<Integer,<Integer, Integer, Float>> send_shuffle() {
12       return key-value list
13    }
14    void receive_shuffle(List<Integer, List<Integer, Integer, Float>> list) {
15       X'_split ← φ
16       foreach (key, values) ∈ list {
17          X'_split ← X'_split ∪ values
18       }
19    }
20    void close() { store X'_split to the local key-value store }
21 }
```

Algorithm 16: **U** stage of ALS

Input : Partitioned rating matrix: $\mathbf{X}'_{split}$, Item feature matrix: $\mathbf{V}$
Output: User feature matrix: $\mathbf{U}$

```
1  class control_task {
2    void initialize() {
3      load V from the local key-value store if exists
4      otherwise, initialize V randomly
5    }
6    boolean is_terminate() { return true }
7    List<Vector> send_broadcast() { return V }
8    void receive_gather(List<Vector> list) { U ← list }
9    void close() { store U in the local key-value store }
10 }
11 class compute_task {
12   void initialize() { load X'_split from the local key-value store }
13   void compute() { compute U_split using X'_split and V }
14   void receive_broadcast(List<Vector>) list) { V ← list }
15   List<Vector> send_gather() { return U_split }
16 }
```

V Stage

The **V** stage is symmetric to the **U** stage. The item feature matrix **V** is computed by compute tasks and stored in the local key-value store by the control task while the user feature matrix **U** is fixed.

4.5 Implementation on REEF

Our multi-stage BSP model are being developed as a part of the Apache REEF project (`http://reef.incubator.apache.org/`). On top of REEF, our model can utilize functionalities provided by REEF such as resource retainment and in-memory processing. In this section, we describe how our model is divided into and implemented on each component of REEF.

4.5.1 Driver

The scheduler in our model is implemented as a part of the driver in REEF. The driver allocates evaluators, assigns services, and assigns tasks composing stages to the evaluators according to the job-level execution order provided by the scheduler.

4.5.2 Service

The data loading service provided by REEF can be used to split input data across compute tasks. Likewise, the group communication service can be used for communication between tasks. Currently, all communication methods used in our model are provided by REEF except shuffle, which is being

developed. A local key-value store that our model uses can be implemented as a service. This allows intermediate data stored in the key-value store are retained across stages.

4.5.3 Task

The control task and compute task are implemented as tasks in REEF. We can separate common execution flow (described in Figure 4.4) from algorithm-specific logics using the strategy design pattern.

4.6 Conclusion

In this chapter, we propose the multi-stage BSP model, a programming model for distributed machine learning. In the model, a job is composed of a series of tightly related stages so that computing resources and intermediate date can be shared among stages. Our model has the following properties:

- **Expressiveness.** Our model can express distributed machine learning algorithms consisting of multiple parallel steps. The entire stages of knowledge discovery process can also be expressed using our model.
- **Concise Programming.** Our model provides rich communication methods among machines as well as iterative execution of stages, which enable concise expression of distributed algorithms.
- **Efficient Implementation.** Our model is being developed on top of REEF, which enables in-memory processing as well as the retainment of computing resources and intermediate results.

Chapter 5. Conclusion

In this thesis, we propose scalable methods for random walk with restart and tensor factorization; and an expressive programming model for distributed machine learning. Our main contributions are as follows:

- **Algorithms.** We propose BEAR, a scalable method for random-walk-with-restart computation. In addition, we propose CDTF and SALS, distributed tensor factorization methods.
- **Analyses.** We analyze the computation and space complexity of BEAR. CDTF and SALS are also analyzed in terms of convergence property as well as computation, communication, and space complexity.
- **Experiments.** Through extensive experiments with many real-world datasets, we show the excellence of BEAR in terms of speed, space-efficiency, and accuracy. Likewise, the superior scalability of CDTF and SALS are experimentally supported.
- **Programming Model.** We propose the multi-stage BSP model, an expressive and convenient programming model for distributed machine learning.
- **Open-source Implementations.** The proposed algorithms and programming model will be available as parts of open-source projects, PEGASUS and Apache REEF, respectively.

References

[1] Evrim Acar, Daniel M Dunlavy, Tamara G Kolda, and Morten Mørup. Scalable tensor factorizations for incomplete data. *Chemometr. Intell. Lab. Syst.*, 106(1):41–56, 2011.

[2] Evrim Acar, Tamara G Kolda, and Daniel M Dunlavy. All-at-once optimization for coupled matrix and tensor factorizations. In *MLG*, 2011.

[3] Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[4] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *JMLR*, 15(1):1111–1133, 2014.

[5] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *nature*, 406(6794):378–382, 2000.

[6] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, 2000.

[7] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

[8] Reid Andersen, David F Gleich, and Vahab Mirrokni. Overlapping clusters for distributed computation. In *WSDM*, pages 273–282, 2012.

[9] Ioannis Antonellis, Hector Garcia Molina, and Chi Chao Chang. Simrank++: Query rewriting through link analysis of the click graph. *PVLDB*, 1(1):408–421, 2008.

[10] Ching Avery. Large-scale graph processing infrastructure on hadoop. In *Hadoop Summit*, 2011.

[11] Lars Backstrom and Jure Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.

[12] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[13] Sudipto Banerjee and Anindya Roy. *Linear Algebra and Matrix Analysis for Statistics*. CRC Press, 2014.

[14] Alex Beutel, Abhimanu Kumar, Evangelos E. Papalexakis, Partha Pratim Talukdar, Christos Faloutsos, and Eric P. Xing. Flexifact: Scalable flexible factorization of coupled tensors on hadoop. In *SDM*, 2014.

[15] Petko Bogdanov and Ambuj Singh. Accurate and scalable nearest neighbors in large networks based on effective importance. In *CIKM*, pages 1009–1018, 2013.

[16] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2009.

[17] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[18] Deepayan Chakrabarti, Spiros Papadimitriou, Dharmendra S. Modha, and Christos Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88, 2004.

[19] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446, 2004.

[20] Soumen Chakrabarti, Amit Pathak, and Manish Gupta. Index design and query processing for graph conductance search. *PVLDB*, 20(3):445–470, 2011.

[21] Po-Lung Chen, Chen-Tse Tsai, Yao-Nan Chen, Ku-Chun Chou, Chun-Liang Li, Cheng-Hao Tsai, Kuan-Wei Wu, Yu-Cheng Chou, Chung-Yi Li, Wei-Shih Lin, et al. A linear ensemble of individual and blended models for music rating prediction. *KDDCup 2011 Workshop*, 2011.

[22] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

[23] Andrzej Cichocki and PHAN Anh-Huy. Fast local algorithms for large scale nonnegative matrix and tensor factorizations. *IEICE Trans. Fundamentals*, 92(3):708–721, 2009.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[25] Peter G Doyle and J Laurie Snell. Random walks and electric networks. *AMC*, 10:12, 1984.

[26] Daniel M Dunlavy, Tamara G Kolda, and Evrim Acar. Temporal link prediction using matrix and tensor factorizations. *ACM TKDD*, 5(2):10, 2011.

[27] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[28] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, volume 29, pages 251–262, 1999.

[29] Yasuhiro Fujiwara, Makoto Nakatsuji, Makoto Onizuka, and Masaru Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5):442–453, 2012.

[30] Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka. Efficient personalized pagerank with accuracy assurance. In *KDD*, pages 15–23, 2012.

[31] R. Gemulla, E. Nijkamp, P.J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[32] David Gleich and Marzia Polito. Approximating personalized pagerank with minimal use of web graph data. *Internet Mathematics*, 3(3):257–294, 2006.

[33] David F Gleich and C Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *KDD*, pages 597–605, 2012.

[34] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[35] Manish Gupta, Amit Pathak, and Soumen Chakrabarti. Fast algorithms for topk personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.

[36] Jingrui He, Mingjing Li, Hong-Jiang Zhang, Hanghang Tong, and Changshui Zhang. Manifold-ranking based image retrieval. In *ACM Multimedia*, pages 9–16, 2004.

[37] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. In *KDD*, pages 538–543, 2002.

[38] Inah Jeon, Evangelos E. Papalexakis, U. Kang, and Christos Faloutsos. Haten2: Billion-scale tensor decompositions. In *ICDE*, 2015.

[39] Dongyeop Kang, Woosang Lim, Kijung Shin, Lee Sael, and U Kang. Data/feature distributed stochastic coordinate descent for logistic regression. In *CIKM*, 2014.

[40] U. Kang and C. Faloutsos. Big graph mining: algorithms and discoveries. *SIGKDD Explorations*, 14(2):29–36, 2012.

[41] U Kang and Christos Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309, 2011.

[42] U. Kang, Evangelos E. Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries. In *KDD*, 2012.

[43] U. Kang, H. Tong, and J. Sun. Fast random walk graph kernel. 2012.

[44] U Kang, C.E Tsourakakis, Ana Paula Appel, C Faloutsos, and Jure Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. *SIAM International Conference on Data Mining*, 2010.

[45] U Kang, C.E Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.

[46] Alexandros Karatzoglou, Xavier Amatriain, Linas Baltrunas, and Nuria Oliver. Multiverse recommendation: N-dimensional tensor factorization for context-aware collaborative filtering. In *RecSys*, 2010.

[47] Gjergji Kasneci, Shady Elbassuoni, and Gerhard Weikum. Ming: mining informative entity relationship subgraphs. In *CIKM*, pages 1653–1656, 2009.

[48] T.G. Kolda and B.W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3), 2009.

[49] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[50] Amy N Langville and Carl D Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2011.

[51] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[52] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *J. Amer. Soc. Inf. Sci. and Tech*, 58(7):1019–1031, 2007.

[53] Yongsub Lim, U. Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng.*, 26(12):3077–3089, 2014.

[54] Zhenjiang Lin, Michael R Lyu, and Irwin King. Matchsim: a novel neighbor-based similarity measure with maximum neighborhood matching. In *CIKM*, pages 1613–1616, 2009.

[55] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. Tensor completion for estimating missing values in visual data. *IEEE TPAMI*, 35(1):208–220, 2013.

[56] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[57] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[58] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *HLT-NAACL*, 2010.

[59] Alexandros Nanopoulos, Dimitrios Rafailidis, Panagiotis Symeonidis, and Yannis Manolopoulos. Musicbox: Personalized music recommendation based on cubic analysis of social tags. *IEEE TASLP*, 18(2):407–412, 2010.

[60] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. Distributed algorithms for topic models. *JMLR*, 10:1801–1828, 2009.

[61] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *NIPS*, 2002.

[62] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *NIPS*, 2011.

[63] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Technical report, Stanford University*, 1999.

[64] Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. Automatic multimedia cross-modal correlation discovery. In *KDD*, pages 653–658, 2004.

[65] Spiros Papadimitriou and Jimeng Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008.

[66] Walter W Piegorsch and George Casella. Inverting a sum of matrices. *SIAM Review*, 32(3):470–470, 1990.

[67] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[68] Lee Sael, Inah Jeon, and U Kang. Scalable tensor mining. *Big Data Research*, (0):–, 2015.

[69] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *SSDBM*, 2013.

[70] Purnamrita Sarkar and Andrew W. Moore. A tractable approach to finding closest truncated-commute-time neighbors in large graphs. In *UAI*, pages 335–343, 2007.

[71] Purnamrita Sarkar and Andrew W Moore. Fast nearest-neighbor search in disk-resident graphs. In *KDD*, pages 513–522, 2010.

[72] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, 2010.

[73] Kijung Shin, Jinhong Jung, Lee Sael, and U Kang. Bear: Block elimination approach for random walk with restart on large graphs. In *SIGMOD*, 2015.

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

[75] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. Cubesvd: a novel approach to personalized web search. In *WWW*, 2005.

[76] Jimeng Sun, Huiming Qu, Deepayan Chakrabarti, and Christos Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005.

[77] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. In *ICDM*, 2012.

[78] Giorgio Tomasi and Rasmus Bro. Parafac and missing values. *Chemometr. Intell. Lab. Syst.*, 75(2):163–180, 2005.

[79] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[80] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, pages 737–746, 2007.

[81] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Random walk with restart: fast solutions and applications. *KAIS*, 14(3):327–346, 2008.

[82] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, et al. Reef: Retainable evaluator execution framework. In *SIGMOD*, 2015.

[83] Joyce Jiyoung Whang, David F Gleich, and Inderjit S Dhillon. Overlapping community detection using seed set expansion. In *CIKM*, pages 2099–2108, 2013.

[84] Yubao Wu, Ruoming Jin, and Xiang Zhang. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *SIGMOD*, pages 1139–1150, 2014.

[85] Cho-Jui Hsieh Si Si Yu, Hsiang-Fu and Inderjit S. Dhillon. Parallel matrix factorization for recommender systems. *Knowl. Inf. Syst.*, pages 1–27, 2013.

[86] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, 2012.

[87] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *USENIX*, 2010.

[88] Chao Zhang, Lidan Shou, Ke Chen, Gang Chen, and Yijun Bei. Evaluating geo-social influence in location-based social networks. In *CIKM*, pages 1442–1451, 2012.

[89] Vincent Wenchen Zheng, Bin Cao, Yu Zheng, Xing Xie, and Qiang Yang. Collaborative filtering meets mobile recommendation: A user-centered approach. In *AAAI*, 2010.

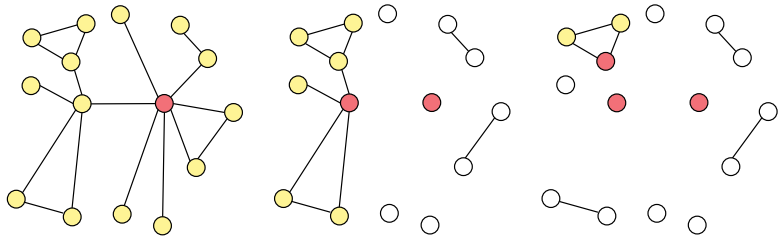
[90] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pages 337–348. 2008.

[91] Zeyuan A Zhu, Silvio Lattanzi, and Vahab Mirrokni. A local algorithm for finding well-connected clusters. In *ICML*, pages 396–404, 2013.

[92] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.

Chapter A. Appendix

A.1 Details of SlashBurn



(a) Step 1 (b) Step 2 (c) Step 3

Figure A.1: Hub selection in SlashBurn when $k = 1$. Hub nodes are colored red, nodes in the giant connected component are colored yellow, and nodes in disconnected components are colored white.

SlashBurn [41, 53] is a node reordering method for a graph so that the nonzeros of the resulting adjacency matrix are concentrated. For the purpose, SlashBurn first removes k hub nodes (high-degree nodes) from the graph so that the graph is divided into the giant connected component (GCC) and the remaining disconnected components. Then, SlashBurn reorders nodes such that the hub nodes get the highest node ids, the nodes in the disconnected components get the lowest node ids, and the nodes in the GCC get the ids in the middle. The above procedure repeats on the GCC until the size of GCC becomes smaller than k. When SlashBurn finishes, the adjacency matrix of the graph contains a large and sparse block diagonal matrix in the upper left area, as shown in Figure 2.4(b). Figure A.1 illustrates this process when $k = 1$.

A.2 The proof of Lemma 1

Proof. Let $\mathbf{L}'$ and $\mathbf{U}'$ be the block-diagonal matrices that consist of $\mathbf{L_1}$ through $\mathbf{L_b}$ and $\mathbf{U_1}$ through $\mathbf{U_b}$, respectively. Then, $\mathbf{L} = \mathbf{L}'$ and $\mathbf{U} = \mathbf{U}'$ because: (1) $\mathbf{L}'$ is a unit lower triangular matrix; (2) $\mathbf{U}'$ is an upper triangular matrix; (3) $\mathbf{L}'\mathbf{U}'$ is the block-diagonal matrix that consists of $\mathbf{L_1U_1}$ through $\mathbf{L_bU_b}$ which is equal to $\mathbf{A}$; and (4) $\mathbf{L}'$ and $\mathbf{U}'$ satisfying (1)$\sim$(3) are the unique LU decomposition of $\mathbf{A}$ [13]. The multiplication of $\mathbf{L}$ and $\mathbf{L^{-1}}$ defined in Lemma 1 results in the block-diagonal matrix that consist of $\mathbf{L_1L_1^{-1}}$ through $\mathbf{L_bL_b^{-1}}$ which is an identical matrix. Likewise, the multiplication of $\mathbf{U}$ and $\mathbf{U^{-1}}$ defined in Lemma 1 results in an identical matrix. □

A.3 Experimental Datasets

Below, we provide a short description of the real-world datasets used in Section 2.4.

- **Routing**[1]**.** The structure of the Internet at the level of autonomous systems. This data was reconstructed from BGP tables collected by the University of Oregon Route Views Project[2].
- **Co-author**[3]**.** The co-authorship network of scientists who posted preprints on the Condensed Matter E-Print Archive[4]. This data include all preprints posted between Jan. 1, 1995 and June. 30, 2003.
- **Trust**[5]**.** A who-trust-whom online social network taken from Epinions.com[6], a general consumer review site. Members of the site decide whether to trust each other, and this affects which reviews are shown to them.
- **Email**[7]**.** An email network taken from a large European research institution. This data include all incoming and outgoing emails of the research institution from October 2003 to May 2005.
- **Web-Notre**[8]**.** The hyperlink network of web pages from University of Notre Dame in 1999.
- **Web-Stan**[9]**.** The hyperlink network of web pages from Stanford University in 2002.
- **Web-BS**[10]**.** The hyperlink network of web pages from University of California, Berkeley and Stanford University in 2002.
- **Talk**[11]**.** A who-talks to-whom network taken from Wiki- pedia[12], a free encyclopedia written collaboratively by volunteers around the world. This data include all the users and discussion (talk) from the inception of Wikipedia to January 2008.
- **Citation**[13]**.** The citation network of utility patents granted in U.S. between 1975 and 1999.

A.4 Parameter settings for B_LIN, NB_LIN, RPPR, and BRPPR

For each dataset, we determine the number of partitions ($\#p$) and the rank (t) of B_LIN among $\{100, 200, 500, 1000, 2000\}$ and $\{100, 200, 500, 1000\}$, resp., so that their pair provides the best trade-off between query time, space for preprocessed data, and accuracy. Likewise, the rank (t) of NB_LIN is determined among $\{100, 200, 500, 1000\}$, and the convergence threshold (ϵ) of RPPR and BRPPR is determined among $\{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$. The parameter values used for each dataset are summarized in Table A.1. B_LIN runs out of memory on the Talk dataset and the Citation dataset regardless of parameter values used.

[1] http://www-personal.umich.edu/~mejn/netdata/as-22july06.zip
[2] http://www.routeviews.org/
[3] http://www-personal.umich.edu/~mejn/netdata/cond-mat-2003.zip
[4] http://arxiv.org/archive/cond-mat
[5] http://snap.stanford.edu/data/soc-sign-epinions.html
[6] http://www.epinions.com/
[7] http://snap.stanford.edu/data/email-EuAll.html
[8] http://snap.stanford.edu/data/web-NotreDame.html
[9] http://snap.stanford.edu/data/web-Stanford.html
[10] http://snap.stanford.edu/data/web-BerkStan.html
[11] http://snap.stanford.edu/data/wiki-Talk.html
[12] http://www.wikipedia.org/
[13] http://snap.stanford.edu/data/cit-Patents.html

Table A.1: Parameter values of B_LIN, NB_LIN, RPPR, and BRPPR used for each dataset. $\#p$ denotes the number of partitions, t denotes the rank, and ϵ denotes the convergence threshold.

dataset	B_LIN		NB_LIN	RPPR	BRPPR
	$\#p$	t	t	ϵ	ϵ
Routing	200	200	100	10^{-4}	10^{-4}
Co-author	200	500	1,000	10^{-4}	10^{-4}
Trust	100	200	1,000	10^{-4}	10^{-5}
Email	1,000	100	200	10^{-3}	10^{-5}
Web-Stan	1,000	100	100	10^{-3}	10^{-4}
Web-Notre	500	100	200	10^{-4}	10^{-5}
Web-BS	2,000	100	100	10^{-3}	10^{-5}
Talk	-	-	200	10^{-3}	10^{-6}
Citation	-	-	100	10^{-4}	10^{-5}

A.5 Additional Experiments

A.5.1 Query Time in Personalized PageRank

We measure the query time of exact methods when the number of seeds is greater than one, which corresponds to personalized PageRank. We change the number of seeds (non-zero entries in $\mathbf{q}$) from 1 to 1000. As seen in Figure A.2, BEAR-Exact is the fastest method regardless of the number of seeds in all the datasets except the smallest Routing dataset The query time of the inverse method increases rapidly as the number of seeds increases. It increases by 210× on the Routing dataset, while the query time of BEAR-Exact increases by 3×. Figure A.3 summarizes the effect of the number of seeds on the query time of BEAR-Exact. In most datasets, the effect of the number of seeds diminishes as the number of seeds increases. The query time increases by up to 16× depending on the number of seeds.

A.5.2 Preprocessing Time of Approximate Methods

Figure A.4 presents the preprocessing time of approximate methods. Preprocessing time is measured in wall-clock time and includes time for SlashBurn (in BEAR-Approx) and community detection (in B_LIN). B_LIN cannot scale to the Talk dataset and the Citation dataset because it runs out of memory while inverting the block diagonal matrices. The relative performances among the methods depend on the structure of graphs, as summarized in Table 2.4. BEAR-Approx tends to be faster than the competitors on graphs with a small number of hubs (such as the Routing and Email datasets) and slower on those with a large number of hubs (such as the Web-BS and Citation datasets).

A.5.3 Comparison with Approximate Methods

Figure A.5 shows the result of the experiments described in Section 2.4.6 on other datasets. In most of the datasets, BEAR-Approx gives the best trade-off between accuracy and time. It also provides the best trade-off between accuracy and storage among all preprocessing methods.

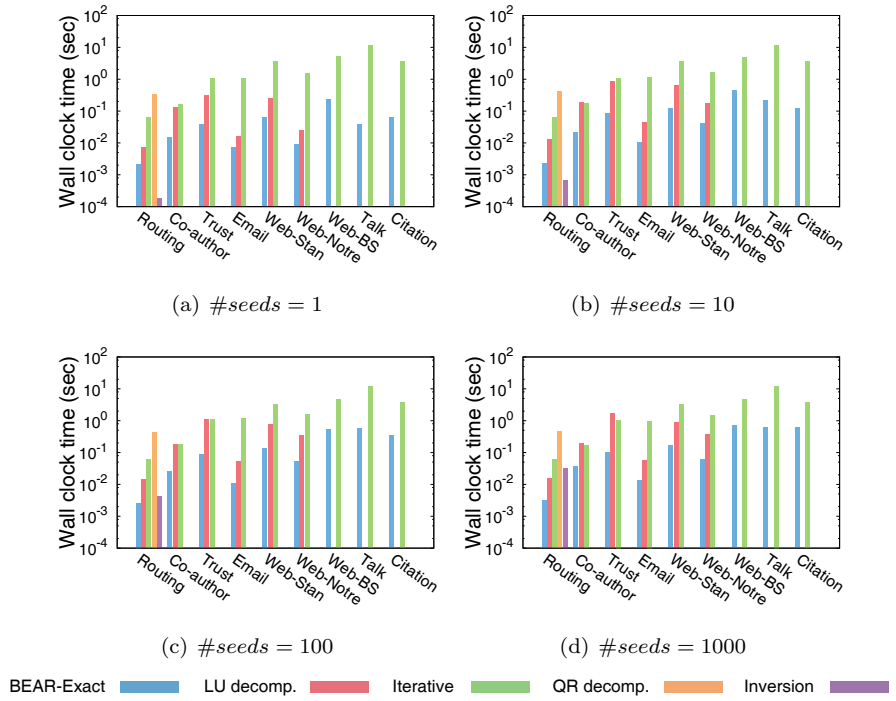


Figure A.2: Query time of exact methods with different number of seeds. If a method cannot scale to a dataset, the corresponding bar is omitted in the graphs. BEAR-Exact is the fastest method regardless of the number of seeds in all the datasets except the smallest Routing dataset.

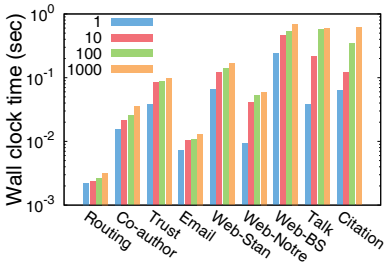


Figure A.3: Effect of the number of seeds on the query time of BEAR-Exact. Query time increases as the number of seeds increases, but the rate of increase slows down.

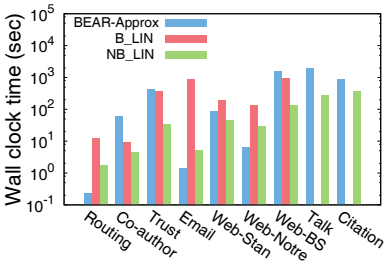
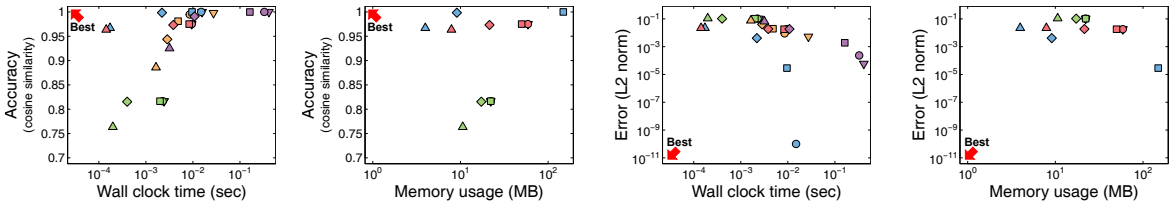
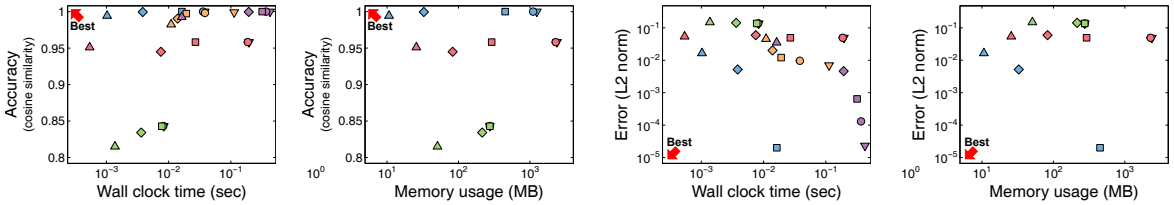


Figure A.4: Preprocessing time of approximate methods. The relative performances among the methods depend on the characteristics of data.

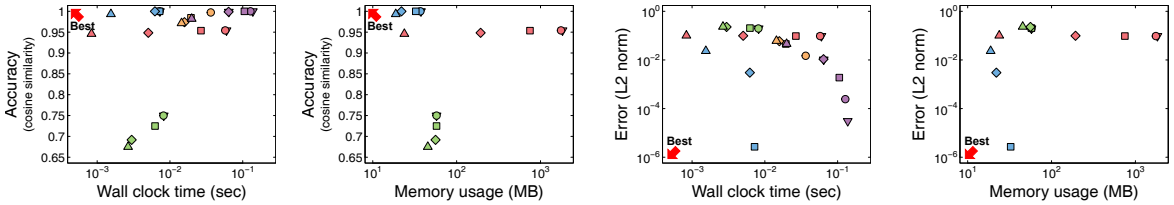
Co-author:



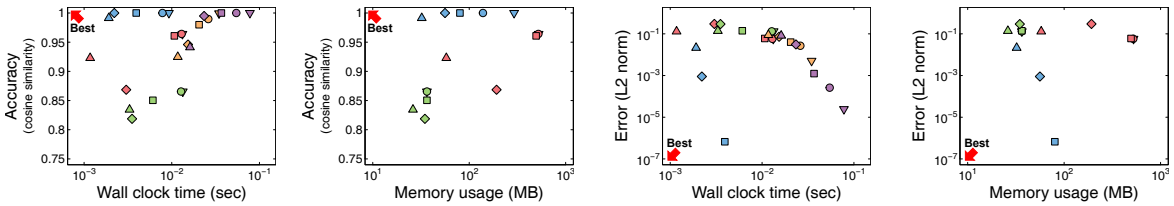
Trust:



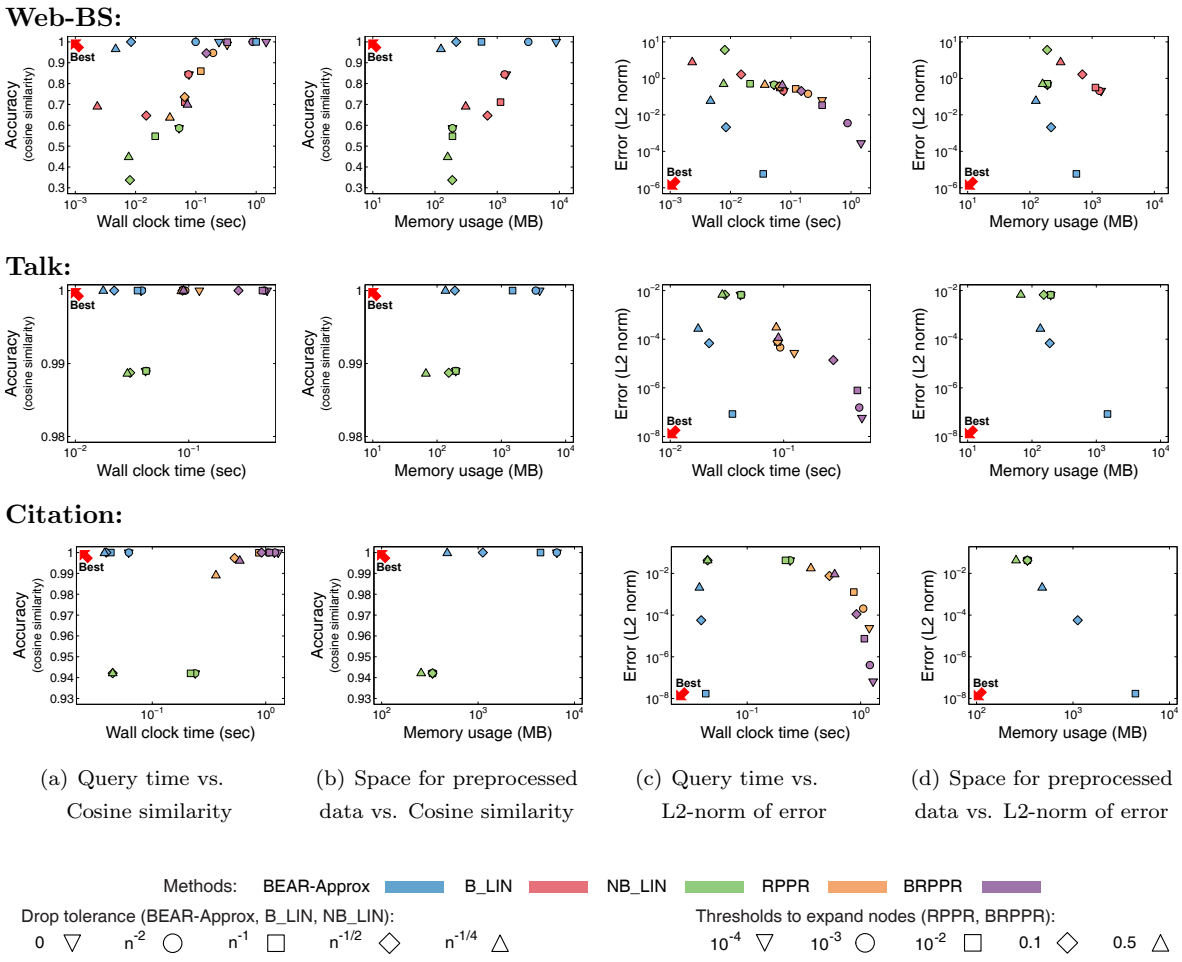
Email:



Web-Notre:



(continues on the next page)



(a) Query time vs. Cosine similarity

(b) Space for preprocessed data vs. Cosine similarity

(c) Query time vs. L2-norm of error

(d) Space for preprocessed data vs. L2-norm of error

Figure A.5: Trade-off between time, space, and accuracy provided by approximate methods. The subfigures in each row show the results on the corresponding dataset. The colors distinguish the methods, and the shapes distinguish the drop tolerance (for BEAR-Approx, B_LIN, and NB_LIN) or the threshold to expand nodes (for RPPR and BRPPR). RPPR and BRPPR do not appear in the subfigures in the second and fourth columns because they do not require space for preprocessed data. In the subfigures in the left two columns, upper-left region indicates better performance, while in the subfigures in the right two columns, lower-left region indicates better performance. BEAR-Approx is located more closely to those regions than the competitors in most of the datasets.