On Density and Noise Challenges in Tensor-Based Data Analytics

by

Xinsheng Li

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2019 by the
Graduate Supervisory Committee:

K. Selçuk Candan, Chair
Hasan Davulcu
Maria Luisa Sapino
Hanghang Tong

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

Many real-world problems, such as model- and data-driven computer simulation analysis, social and collaborative network analysis, brain data analysis, and so on, benefit from jointly modeling and analyzing the underlying patterns associated with complex, multi-relational data. Tensor decomposition is an ideal mathematical tool for this joint modeling, due to its simultaneous analysis of such multi-relational data, which is made possible by the data's multidimensional, array-based nature.

A major challenge in tensor decomposition lies with its computational and space complexity, especially for dense datasets. While the process is comparatively faster for sparse tensors, decomposition is still a major bottleneck for many applications. The tensor decomposition process results in dense (hence, large) intermediate results, even when the input tensor is sparse (or small). Noise is another challenge for most data mining techniques, and many tensor decomposition schemes are sensitive to noisy datasets; this is an inevitable problem for real-world data, which can lead to false conclusions.

In this dissertation, I develop innovative tensor decomposition algorithms for mining both sparse and dense multi-relational data in a noise-resistant way. I present novel, scalable, parallelizable tensor decomposition algorithms, specifically tuned to be effective for dense, noisy tensors, and which maintain the quality of the resulting analysis. Furthermore, I present results on multi-relational data applications focusing on model- and data-driven computer simulation analysis, as well as social network and web mining, which demonstrate the effectiveness of these tensor decompositions.

DEDICATION

To my parents and my wife

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. K. Selçuk Candan. Without his mentorship and endless support, I can not finish this journey. He gives me an example of an excellent researcher, mentor, instructor. His feedback and insights were extremely critical to achieving my goal, and I will always be grateful to him for that.

I would like to thank my committee members, Dr. Maria Luisa Sapino, Dr. Hasan Davulcu, and Dr. Hanghang Tong. Thanks for all your guidance through this process; I would also like to thank the graduate advisors and technical staff members at CIDSE for always being very helpful.

I would like to thank all my excellent lab members of EmitLab, Parth Nagarkar, Jung Hyun Kim, Xilun Chen, Mijung Kim, Mithila Nagendra, Yash Garg, Sicong Liu, Shengyu Huang, Silvestro Poccia, Ashish Gadkari, Mao-Lin Li, and Hans Behrens. I am very happy to have this opportunity to work with such wonderful team members.

Lastly, I would especially like to thank my amazing family for the love, support, and constant encouragement I have gotten over the years. In particular, I would like to thank my parents, my wife, and my brother. Without your support, I definitely could not finish this.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Tensors or multi-way arrays (number of orders higher than two) are arrays indexed by three or more indices, which is a high-order generalization of matrices. Tensors are used in various disciplines. In the real world, most media and sensor data are multi-dimensional and multi-relational, such as social network, web graphs, sensor streams, and simulation ensembles. All multi-dimensional data can be modeled as tensor [23, 58, 35, 48, 39, 17] . Tensor decomposition is a process which rewrites an input tensor as a set of factor matrices and a core tensor(which describes the strength of latent clusters).

Many tensor decomposition schemes are sensitive to noisy data (i.e., due to the sensor fault, sensor streams is inconsistent). Noise pollution is an unavoidable problem in the data analysis domain. It has two major sources, one introduced by measurement tools. One introduced by the batch processing of expert, when the data is collected. The noisy data problem can be further compounded by overfitting, especially when the observed data is sparse. To address these challenges, I proposed the Noise-Profile Adaptive Tensor Decomposition (nTD) method, which leverages rough information about noise distribution in the data to improve tensor decomposition accuracy with block based tensor decomposition framework. Another approach, Noise-Profile Adaptive Tensor Train Decomposition (NTTD) method, is proposed, which is deployed on the Tensor Train format. NTTD aims to tackle the challenge, noise is rarely uniformly distributed in the data. Another key problem with tensor decomposition is its computational complexity and space requirements. As the relevant data sets get denser and larger, in-memory schemes for tensor decomposition become increasingly ineffective; therefore out-of-core (secondary-memory supported, potentially parallel) computing is necessitated. In this dissertation, I introduced 2PCP, a two-phase, block-based CP decomposition system with intelligent buffer sensitive task scheduling and buffer management mechanisms. 2PCP aims to reduce I/O costs in the analysis of relatively dense tensors common in scientific and engineering applications. Data- and model-driven computer simulations are increasingly critical in many application domains. However, obtaining and interpreting simulation ensembles to generate actionable results present difficulties, such as limited ensemble simulation budgets, need for post-simulation data processing, and inherent data sparsity of simulation ensembles. To handle those challenges, I propose an alternative ensemble creation strategy, which I refer to as the partition-stitch sampling, to increase the effective density of the ensemble with limited ensemble simulation budgets. Based on partition-stitch sampling, Multi-Task Tensor Decomposition (M2TD) scheme is proposed, which reduces the computational complexity of high-order tensor decomposition by (a) first cheaply decomposing the low-order partial tensors and (b) intelligently stitching back the decompositions of these partial tensors to obtain the decomposition for the whole system. Intuitively, M2TD leverages partial and imperfect simulation-based knowledge from the resulting partial dynamical systems to obtain a global view of the complex process being simulated.

1.1 Tensor Representation and Analysis

As a good representation of multi-dimensional and multi-relational data, the tensor representation is widely used in many applications include representations of RDF triples (subject-predicate-object) in knowledge bases, (venue-author-keywords) relationships in scientific digital libraries [18], and (movie-user-rating) relationships in movie recommendation [39]. Consequently, tensor decomposition operations (such as CP [23] and Tucker [58]) are showing an upward trend in many data analysis and knowledge discovery tasks, from clustering, trend detection, anomaly detection [18], to correlation analysis [56]. Tensor decomposition is commonly used techniques in the domain of tensor analysis, which is high order generation of matrix factorization. Similar to matrix factorization, it recovers latent features or clusters through decomposing tensor. Tensor decomposition is to decompose tensor into one small core tensor and a set of factor matrices. The small core tensor indicates the relational strength among latent clusters. The factor matrix represents the probability of the object belonging to latent clusters. CANDECOMP/PARAFAC (CP) proposed by Harshman [23] and Carrol [10] of the tensor concept is the fuse to make tensor popular. Tucker decomposition is another landmark in the tensor decomposition domain. The CP format can effectively avoid the curse of dimensionality. However, the disadvantage of CP format is numerical problem for very high-order tensors, which is caused by the intrinsic uncloseness of the CP format. TT tensor network [48] is proposed to overcome this problem, which have both good numerical properties and the ability to control the error of approximation. Those characteristic provide relatively easy opportunity to achieve desired approximation accuracy. Truncated singular value decompositions (tSVD) or adaptive cross-approximation [30, 47, 6] of TT format achieve stable quasi-optimal rank reduction, which is the disadvantage of CP decomposition. TT format is widely used in many domains for its stale and simple approaches, which separate latent variables in a sophisticated way. The associated TT decomposition algorithms provide full control over low-rank TN approximations

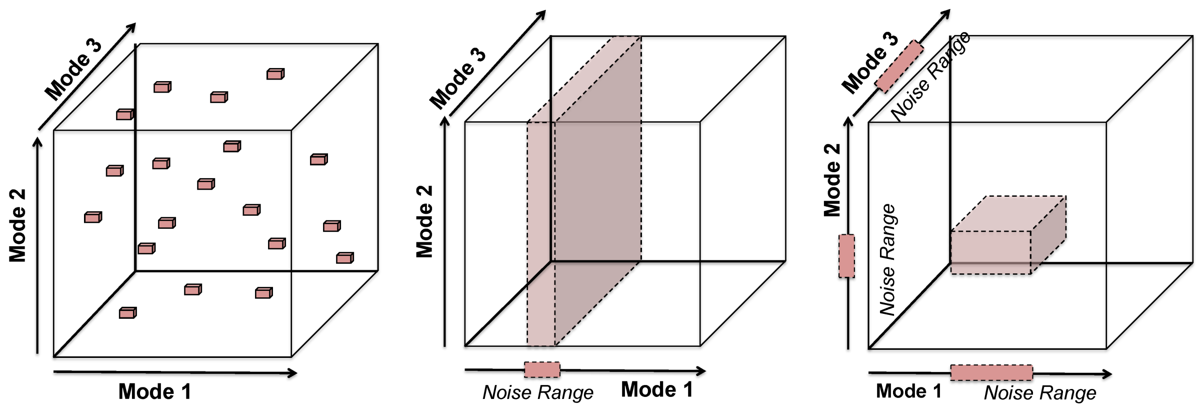
One problem with tensor decomposition, however, is its computational complexity especially for dense data sets, the decomposition process takes exponential time in the number of tensor modes. While the process is relatively faster for sparse tensors, decomposition is still a major bottleneck in many applications: decomposition algorithms have high computational costs and incur large memory overheads and, thus, are not suitable for large problems. Even more recent improvements distributed/parallel implementations, such as Grid PARAFAC [51] and GigaTensor [28], suffer from high computational costs. Due to the approximate nature of the tensors decomposition process, one way to reduce computational requirements might be to trade performance with accuracy. However, naturally, a drop in accuracy may not be acceptable in many applications. Therefore, this is not a feasible solution to tackle the computational cost.

1.2 Research Contributions

While block-based tensor decomposition techniques [51] and tensor network formats [14, 15, 48]provide potential opportunities to boost the accuracy/efficiency trade-off in the noisy data and dense data. Those solutions leave several open questions, including (a) how to partition the tensor and (b) how to most effectively combine results from these partitions (c) how to optimize for the out-of-core computing scenario (d) opportunities in inherent sparse simulation ensembles. In this dissertation, I introduce different algorithms to address different scenarios, which quantify how to leverage the inherent structure of different tensor decompositions to impact the overall tensor decomposition accuracy under different scenarios. I present four complementary algorithms that leverage the inherent structure of different tensor decomposition to address various key challenges in tensor decomposition, including noise, dense data, and sparse simulation ensembles scenario.

1.2.1 Dealing with Noisy Data

Many of the tensor decomposition schemes are sensitive to noisy data, an inevitable problem in the real world that can lead to false conclusions. Tensor decomposition faces are that the process can be negatively affected from the noise and low quality in the data, which is especially a concern for web-based user data in particular, especially for sparse data, avoiding over-fitting to the noisy data can be a significant challenge. Recent research has shown that it is possible to avoid such over-fitting by relying on probabilistic techniques [35], which introduces priors on the parameters, it can effectively average over various models and ease the pain of tuning parameters.



(a) uni noise (b) sc noise (c) mm noise

Figure 1.1: Alternative Noise Profiles of a Tensor

Parameters of Tensor Noise Profile

Noise distribution: Noise can be distributed in a tensor in several ways:

- In *uniform (uni) noise* (Figure 1.1(a)), there is no underlying pattern and noise is not clustered across any slice or region of the tensor.
- *Slice-concentrated (sc)* noise (Figure 1.1(b)) is clustered on one or more slices on the tensor across one or more modes. For example, a particular data source (represented by one or more slices) may be known to provide low quality, untrusted information.
- In *multi-modal (mm)* noise, again, the noise is clustered; however, in this case the noise is expected to occur when a combination of a subset of the values across two or more modes are considered together as in Figure 1.1(c).

Noise density: This is the ratio of the cells that are subject to noise. In this dissertation, without loss of generality, noise is on cells that have values (i.e., the observed values can be faulty, but there are no spurious observations) and, thus, noise density as *a ratio of the non-null cells.*

Dependent vs. independent noise: Noise may impact the observed values in the tensor in different ways: in *value-independent noise*, the correct data may be overwritten by a completely random new value, whereas in *value-correlated noise* existing values may be perturbed (often with a Gaussian noise, defined by a standard deviation, σ).

Block Based Tensor Decomposition Dealing with Noisy Data

Unfortunately, existing probabilistic approaches have two major deficiencies: (a) firstly, they assume that all the data and intermediary results can fit in the main memory and (b) they treat the entire tensor uniformly, ignoring possible non-uniformities in the distribution of noise in the given tensor.

To deal with the challenge, in this dissertation, I propose a *Noise Adaptive Tensor Decomposition (*`nTD`*)* method: `nTD` partitions the tensor into multiple sub-tensors and then decomposes each sub-tensor probabilistically through Bayesian factorization – the resulting decompositions are then recombined through an iterative refinement process to obtain the decomposition for the whole tensor. Simultaneously `nTD` develops a resource allocation strategy that accounts for the impact of the noise density of one sub-tensor on the decomposition accuracies of the other sub-tensors:

This provides several benefits:

- Firstly, the partitioning helps ensure that the memory footprint of the decomposition is kept low.
- Secondly, the probabilistic framework used in the first phase ensures that the decomposition is robust to the presence of noise in the sub-tensors.
- Thirdly,*a priori* knowledge about noise distribution among the sub-tensors is used to obtain a resource assignment strategy that best suits the noise profile of the given tensor.

Tensor Train Format Dealing with Noisy Data

Recent research has shown that several generalizations of higher order tensors' low-rank decompositions, such as hierarchical Tucker (HT) [20] and the Tensor Train (TT) [48] format, are effective solutions to this problem. Both Hierarchical Tucker and Tensor Train are designed to avoid the curse of dimensionality, in the form of the exposition of intermediary results, which plagues other tensor decomposition techniques. *Noise-Profile Adaptive Tensor Train Decomposition (*`NTTD`*)* method is proposed to leverages *rough* a prior information about noise in the data (which may be user provided or obtained through automated techniques [54, 13]) to improve decomposition accuracy in Tensor Train format. `NTTD` decomposes each mode matricization probabilistically through Bayesian factorization – the resulting factor matrix are then reconstructed to obtain the tensor approximations. Most importantly,

> `NTTD` *provides a resource allocation strategy, which accounts for the impacts of (a) the noise density of each mode and (b) inherent approximation error of the Tensor Train decomposition process, on the overall decomposition accuracy of the input tensor.*

In other words, *a priori* knowledge about noise distribution on the tensor and the inherently approximate nature of the tensor train decomposition process are both considered to obtain a decomposition strategy, which involves (a) the order of the modes and (b) the number of Gibbs samples allocated to each step of the decomposition process, that best suits the noise distribution of the given tensor.

1.2.2 Density Challenge in Tensor-Based Analytics

Tensor decomposition process results in dense (and hence large) intermediary data, even when the input tensor is sparse (and hence small). This is known as the intermediate memory blow-up problem and renders purely in-memory implementations of tensor-decomposition impractical, for both CP and Tucker decompositions

As the relevant data sets get large, existing in-memory schemes for tensor decomposition become increasingly ineffective and block-based solutions where some (possibly intermediate) data may be materialized on disks (instead of main memory) or other servers contributing to the decomposition process is necessitated. Several implementations of tensor decomposition operations on disk-resident data sets have been proposed, such as GridPARAFAC [51], TensorDB[33], HaTen2[26] and so on. In all these systems, I/O costs are an inevitable problem as they need I/O to fetch data either from disk or from the network. As experiments verified, the I/O or communication overhead of iterative algorithm (especially on a distributed platform like MapReduce) can be very expensive. In addition, naive implementations of the block-based iterative improvement algorithms can result in significant I/O, when the buffer memory is not large enough to hold the entire intermediary data. Consequently, reducing these I/O and communication costs, especially for dense tensors common in science and engineering, is a critical challenge.

In this dissertation, I propose `2PCP`, a two-phase CP tensor decomposition mechanism. Two-phase block-based tensor decomposition can help reduce the memory-blow-up problem as the first phase requires decomposition of much smaller tensors. However, the number of the (so-called factor) matrices that are produced in the first phase and the intermediary data generated while these are stitched together through an iterative process in the second phase may still be quite large. Consequently, the intermediary data may still take too much space to be fully memory-resident and may need to be brought to the memory on the on-demand basis. Consequently, the `2PCP` system, I present in this dissertation, complements the basic two-phase CP tensor decomposition approach with novel *data re-use promoting block scheduling* and *buffer management* mechanisms to address this difficulty:

In its first phase, `2PCP` partitions the input data to blocks (or sub-tensors), then conducts ALS on each sub-tensor (potentially in parallel using a MapReduce based platform) independently

In the second phase, which is executed on a single worker machine, `2PCP` leverages fine-grained block centric iterative refinement with a novel forward looking buffer replacement strategy that helps improve buffer utilization and reduce I/O:

- In particular, I extend the conventional mode-centric approach in a way that enables more flexible, fine-grained, block-centric scheduling of updates and the corresponding data accesses

- Given this *fine-grained* block-centric iterative improvement scheme, I then consider alternative scheduling techniques that can maximize the utility of the intermediary-data already in the buffer

- I then propose and study alternative buffer replacement policies complementing the different scheduling techniques considered above and develop a forward-looking, predictive buffer replacement strategy that matches the proposed scheduling techniques to further push the I/O costs down

1.2.3 Tensor-based analytics on Inherent Sparse Simulation Ensembles

Data- and model-driven computer simulations are increasingly critical in many application domains. For example, for predicting geo-temporal evolution of epidemics and assessing the impact of interventions, experts often rely on epidemic spread simulation software, such as STEM [40]. Consequently, obtaining and interpreting simulation ensembles to generate actionable results present difficulties: Limited ensemble simulation budgets: Since complex, inter-dependent parameters affected by complex dynamic processes have to be taken into account, *execution of simulation ensembles can be very costly.* This leads to *simulation budget constraints* that limit the number of simulations one can include in an ensemble. Need for post-simulation data processing: Because of the complexities of key processes and the varying scales at which they operate, experts often lack the means to drive conclusions from these ensembles. This leads to the need for *data analytics on simulation ensembles to discover broad, actionable patterns.* Inherent data sparsity of simulation ensembles: While the size and complexity of a simulation ensemble can indeed tax decision makers, I note that *a simulation ensemble is inherently sparse (relative to the space of potential simulations one could run)*, which constitutes a significant problem in simulation-based decision making. This leads to the following critical question: "*Given a parameter space and a fixed simulation budget, which simulation instances should be included in the ensemble?*

To address those challenges, Density Boosting Partition-Stitch Sampling is proposed. I propose an alternative ensemble creation strategy, which I refer to as the *partition-stitch sampling*: given an N-parameter simulation and an ensemble budget of B, instead of randomly allocating the B samples in the N-dimensional parameter space, I partition the simulation space into $\sim N/2$ dimensional sub-spaces and allocate $B/2$ simulations for each sub-space: note that, since the number of possible simulations for each sub-space reduced exponentially (in the number of excluded parameters), this corresponds to an exponential increase in the density of the samples for each sub-space: Note that neither of the two systems is perfect representations of the overall behavior of the whole system.

One important question is "*How to stitch back the results obtained from the individual sub-spaces?*" Here there might be several alternatives: In the simplest alternative, all the simulations from the two systems can be unioned into a single N-mode tensor and this N-mode tensor can be decomposed for analysis. This is potentially very expensive as the decomposition cost often increases exponentially with the number of modes of the input tensor. Once unioned into a single tensor, the overall density is still low and the accuracy gains will be very limited. Instead, I present a join-based scheme to increase the *effective density* of the ensemble. In particular, two approaches (join stitching and zero-join stitching) are presented to combine simulation results form the sub-systems and experimentally validate the effectiveness of these schemes.

Multi-Task Tensor Decomposition (M2TD): Naively joining the sub-ensembles would map the simulations back to an N-modal tensor and this would exponentially increase the tensor decomposition time. Instead, I propose a novel Multi-Task Tensor Decomposition (`M2TD`) scheme, which reduces the computational complexity of high-order tensor decomposition by (a) first cheaply decomposing the low-order partial tensors and (b) intelligently stitching back the decompositions of these partial tensors to obtain the decomposition for the whole system. Intuitively, M2TD leverages partial and imperfect simulation-based knowledge from the resulting partial dynamical systems to obtain a global view of the complex process being simulated. I study alternative ways one can stitch the tensor decompositions and propose an $\texttt{M2TD} - \texttt{SELECT}$ that provides better accuracy than the alternatives.

1.3 Dissertation Outline

The dissertation is organized in the following ways:

- In section 2,the background and related work about noise and density challenge for tensor-based analysis is presented
- In section 3, I describes the algorithm of nTD to handle the noise data scenario in block based tensor decomposition framework.
- in section 4, NTTD is proposed, which leverages a model-based noise adaptive tensor train decomposition strategy.
- In section 5, 2PCP framework to reduce I/O costs in the decomposition of relatively dense tensors is introduced
- in section 6, M2TD is illustrated, which rely on a tensor-based framework to represent and analyze patterns in large simulation ensemble data sets to obtain a high-level

understanding of the dynamic processes implied by a given ensemble of simulations

- in section 7, I conclude this dissertation.

Chapter 2

BACKGROUND AND RELATED WORKS

2.1 Tensors and Tensor Decompositions

In this section, the relevant background and notations are presented. Tensors are generalizations of matrices: while a matrix is essentially a 2-mode array, a tensor is an array of possibly larger number of modes. Intuitively, the tensor model maps a relational schema with N attributes to an N-modal array (where each potential tuple is a tensor cell).

The two most popular tensor decomposition algorithms are the Tucker [58] and the CANDECOMP/PARAFAC (or CP) [23] decompositions. Intuitively, both decompositions generalize singular value matrix decomposition (SVD) to tensor. CP decomposition, for example, decomposes the input tensor into a sum of component rank-one tensors.

2.2 CP Decomposition

More specifically, given a tensor $\boldsymbol{\mathcal{X}}$, the CP decomposition factorizes the tensor into F component matrices (where F is a user supplied non-zero integer value also referred to as the *rank* of the decomposition). For the simplicity of the discussion, let us consider a 3-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$. CP would decompose $\boldsymbol{\mathcal{X}}$ into three matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, such that

$$\boldsymbol{\mathcal{X}} \approx \tilde{\boldsymbol{\mathcal{X}}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}] \equiv \sum_{f=1}^{F} a_f \circ b_f \circ c_f,$$

where $a_f \in \mathbb{R}^I$, $b_f \in \mathbb{R}^J$ and $c_f \in \mathbb{R}^K$. The factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are the combinations of the rank-one component vectors into matrices; e.g., $\mathbf{A} = [\ a_1\ a_2\ \cdots\ a_F\ ]$. This is visualized in Figure 2.1.

Many algorithms for decomposing tensors are based on an iterative process that tries to

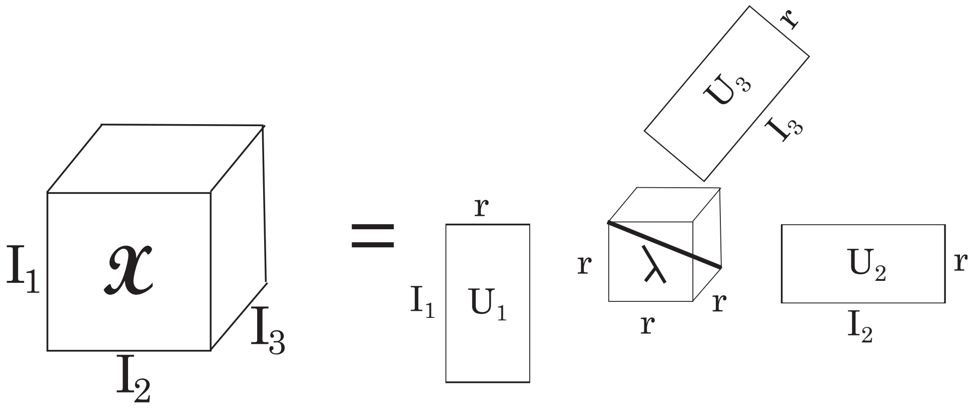


Figure 2.1: Illustration of CP Decomposition

improve the approximation until a convergence condition is reached through an alternating least squares (ALS) method: at its most basic form, ALS estimates, at each iteration, one factor matrix while maintaining other matrices fixed; this process is repeated for each factor matrix associated to the modes of the input tensor. Note that due to the approximate nature of tensor decomposition operation, given a decomposition $[\mathbf{A}, \mathbf{B}, \mathbf{C}]$ of $\boldsymbol{\mathcal{X}}$, the tensor $\tilde{\boldsymbol{\mathcal{X}}}$ that one would obtain by re-composing the tensor by combining the factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ is often different from the input tensor, $\boldsymbol{\mathcal{X}}$. The accuracy of the decomposition is often measured by considering the Frobenius norm of the difference tensor:

$$accuracy(\boldsymbol{\mathcal{X}}, \tilde{\boldsymbol{\mathcal{X}}}) = 1 - error(\boldsymbol{\mathcal{X}}, \tilde{\boldsymbol{\mathcal{X}}}) = 1 - \left(\frac{\|\tilde{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{X}}\|}{\|\boldsymbol{\mathcal{X}}\|} \right).$$

2.3 Tucker Decomposition

The Tucker decomposition is a form of higher-order principal component analysis. It decomposes a tensor into a core tensor multiplied (or transformed) by a matrix along each mode.

Given a tensor $\boldsymbol{\mathcal{X}}$, Tucker decomposition factorizes the tensor into factor matrices with different rows, which are referred to as the rank of the decomposition. For the simplicity of the discussion, let us consider a 3-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$. Tucker decomposition would

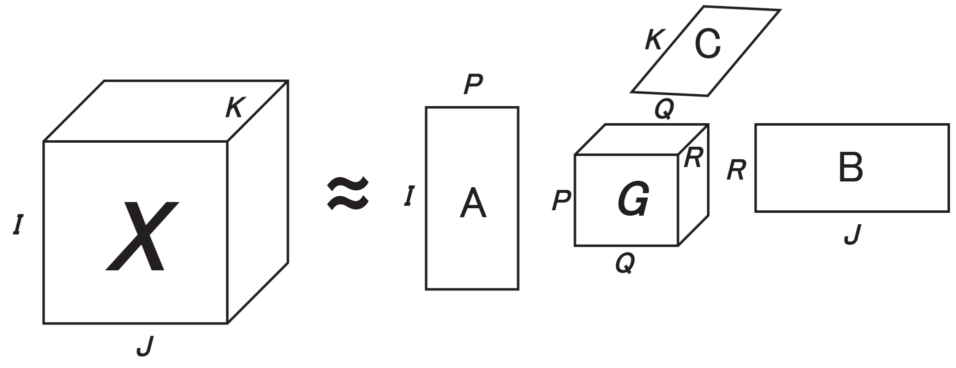


Figure 2.2: Illustration of Tucker Decomposition

decompose $\boldsymbol{\mathcal{X}}$ into three matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and one core dense tensor $\mathbf{g}$, such that

$$\boldsymbol{\mathcal{X}} \approx \tilde{\boldsymbol{\mathcal{X}}} = \mathbf{g} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} \equiv \sum_{p=1}^{P} \sum_{q=1}^{Q} \sum_{r=1}^{R} g_{pqr} a_p \circ b_q \circ c_r,$$

where $\mathbf{A} \in \mathbb{R}^{I \times P}$, $\mathbf{B} \in \mathbb{R}^{J \times Q}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, are the factor matrices and can be treated as the principal components in each mode. $\mathbf{g} \in \mathbb{R}^{P \times Q \times R}$dense core tensor, which indicates the strength of interaction between different component of each factor matrix. Since tensors may not always be exactly decomposed, the new tensor $\tilde{\boldsymbol{\mathcal{X}}}$ obtained by recomposing the factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and core tensor $\mathbf{g}$ is often different from the input tensor, $\boldsymbol{\mathcal{X}}$. The accuracy of the decomposition is often measured by considering the Frobenius norm of the difference tensor.

2.4 Tensor Train Decomposition

Intuitively, the tensor model maps a schema with N attributes to an N-modal array (where each potential tuple is a tensor cell). Tensor decomposition process generalizes the matrix decomposition process to tensors and rewrites the given tensor in the form of a set of factor matrices (one for each mode of the input tensor) and a core matrix (which, intuitively, describes the spectral structure of the given tensor). A common problem faced by tensor decomposition techniques, such as Tucker, which generates dense core tensors, is

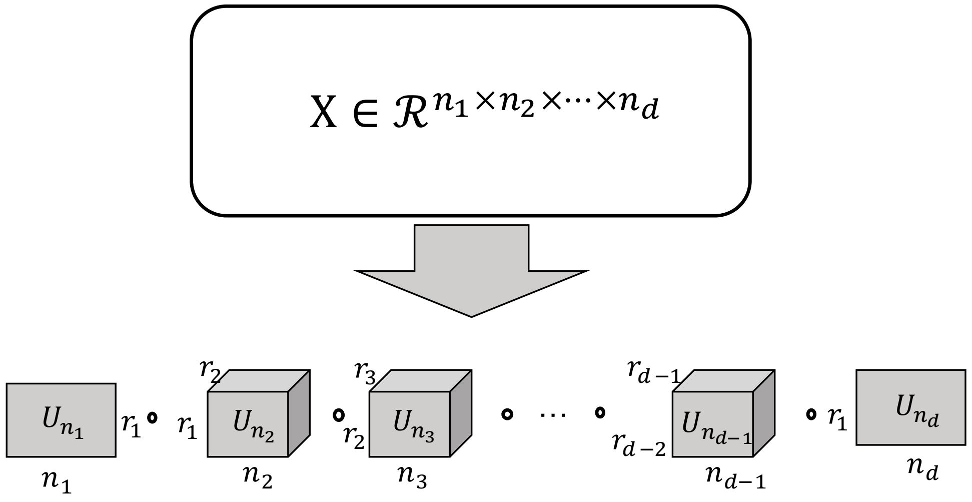


Figure 2.3: Illustration of Tensor Train (TT) Decomposition

that, even when the input is sparse, the intermediary and final steps in the decomposition may lead to very large datasets. Recent research has shown that several generalizations of higher order tensors' low-rank decompositions, such as hierarchical Tucker (HT) [20] and the Tensor Train (TT) [48] format, are effective solutions to this problem. Both Hierarchical Tucker and Tensor Train are designed to avoid the curse of dimensionality, in the form of the exposition of intermediary results, which plagues other tensor decomposition techniques.

Intuitively, the tensor train decomposition (which can be interpreted as a special case of HT, without a recursive formulation) avoids the creation of a high-modal dense core, by splitting the core into a sequence of low (3) modal cores (Figure 2.3). Since, the intermediary/final data size is exponential in the number of modes of the core tensor, this significantly reduces the computation and storage requirements of the decomposition process.

A major difficulty with the Tucker decomposition is that the dense core can be prohibitively large and expensive for high-modal tensors. As discussed in the introduction and related work, several tensor network approaches [48, 5, 4, 25] (where network edges correspond to contraction indices) have been proposed to avoid this large, dense core tensor. Unfortunately, many of these possess bad numerical properties. Due to its simpler structure, the tensor train (TT) format, which creates a linear tensor network (or a matrix product state, MPS [14, 15]) avoids the deficiencies of many other complex decomposition structures:

Definition 2.4.1 (Tensor Train (TT) Format) *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ be a tensor of order d. In Figure 2.3 (in Introduction) shown, the* tensor train decomposition *decomposes $\boldsymbol{\mathcal{X}}$ into d matrices $\mathbf{U}_{n_1}, \mathbf{U}_{n_2}, \ldots, \mathbf{U}_{n_d}$ such that,*

$$\boldsymbol{\mathcal{X}} \approx \tilde{\boldsymbol{\mathcal{X}}} = \mathbf{U}_{n_1} \circ \mathbf{U}_{n_2} \circ \cdots \circ \mathbf{U}_{n_d}, \tag{2.1}$$

where $\mathbf{U}_{n_1} \in \mathbb{R}^{n_1 \times r_1}$, $\mathbf{U}_{n_i} \in \mathbb{R}^{r_{i-1} \times n_i \times r_i} (i = 2, \ldots, d-1)$, and $\mathbf{U}_{n_d} \in \mathbb{R}^{r_{d-1} \times n_d}$.

Note that Equation 2.1, above, can also be written as follows:

$$\begin{aligned} &\boldsymbol{\mathcal{X}}[i_1 \ldots i_d] \approx \\ &\tilde{\boldsymbol{\mathcal{X}}}[i_1 \ldots i_d] = \sum_{j_1} \cdots \sum_{j_{d-1}} \mathbf{U}_{n_1}[i_1, j_1] \mathbf{U}_{n_2}[j_1, i_2, j_2] \ldots \mathbf{U}_{n_d}[j_{d-1}, i_d]. \end{aligned}$$

As in Tucker decomposition, a common way to obtain tensor train decomposition is to rely on singular value decompositions (SVD) of the matricizations of the tensor to obtain factors. Unlike HOSVD, however, instead of obtaining all matricizations across all modes simultaneously and then solving for a multi-modal dense core, tensor train decomposition proceeds matricization along one mode at a time: at each step, a factor matrix or a low-dimensional core, corresponding to the current mode, is obtained and the remainder of the data (which now has one less mode) is passed to the next step in the process [48].

2.5 Block-based CP Decomposition

block-based CP decomposition techniques[51] partition the given tensor into blocks, initially decompose each block independently, and then iteratively combine these decompositions into a final composition.

Let us consider an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\boldsymbol{\mathfrak{X}} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes. Without

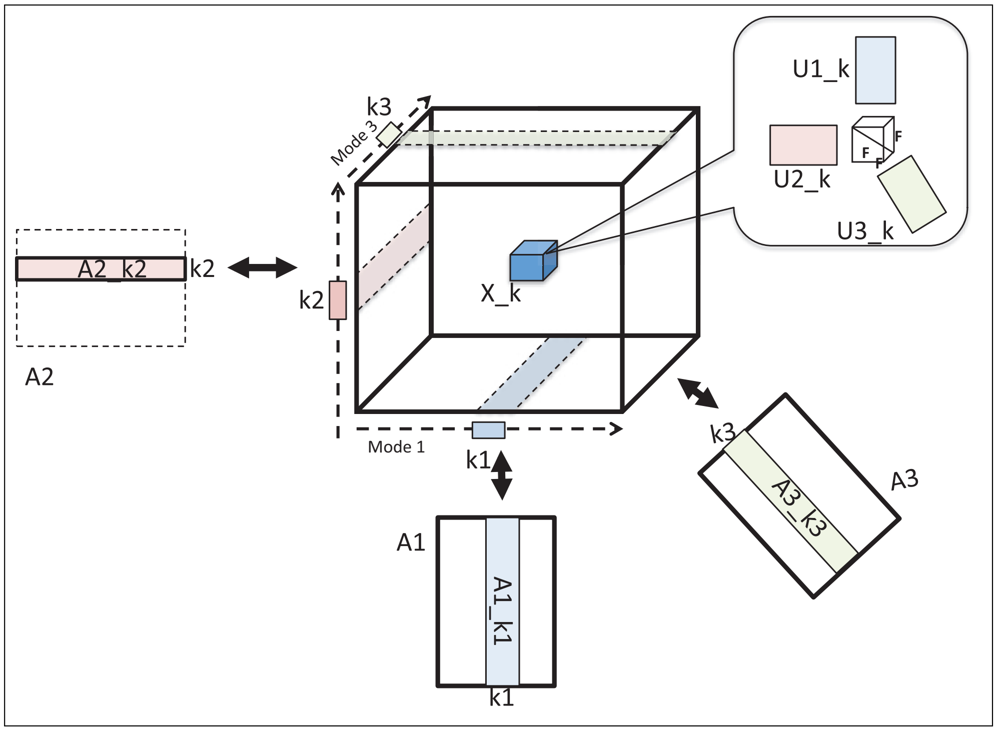


Figure 2.4: Each Sub-tensor (or Block) Can Be Described in Terms of The Corresponding Sub-factors

loss of generality, let us assume that $\mathcal{K}$ partitions the mode i into K_i equal partitions; i.e., $|\mathcal{K}| = \prod_{i=1}^{N} K_i$. Let us also assume that a target decomposition rank, F, is given for the tensor $\boldsymbol{\mathcal{X}}$. Let us further assume that each sub-tensor in $\mathfrak{X}$ has already been decomposed with target rank F and let $\mathfrak{U}^{(i)} = \{\boldsymbol{U}_{\vec{k}}^{(i)} \mid \vec{k} \in \mathcal{K}\}$ denote the set of F-rank sub-factors[1] corresponding to the sub-tensors in $\mathfrak{X}$ along mode i. In other words, for each $\boldsymbol{\mathcal{X}}_{\vec{k}}$, there is

$$\boldsymbol{\mathcal{X}}_{\vec{k}} \approx \boldsymbol{I} \times_1 \boldsymbol{U}_{\vec{k}}^{(1)} \times_2 \boldsymbol{U}_{\vec{k}}^{(2)} \cdots \times_N \boldsymbol{U}_{\vec{k}}^{(N)}, \tag{2.2}$$

where $\boldsymbol{I}$ is the N-mode $F \times F \times \ldots \times F$ identity tensor, where the diagonal entries are all 1s and the rest are all 0s.

Given these, [51] presents an iterative improvement algorithm for composing these initial sub-factors into the full F-rank factors, $\boldsymbol{A}^{(i)}$ (each one along one mode), for the input tensor,

[1]If the sub-tensor is empty, then the factors are $\mathbf{0}$ matrices of the appropriate size.

$\boldsymbol{\mathcal{X}}$. The outline of this block based process is as follows: Let us partition each factor $\boldsymbol{A}^{(i)}$ into K_i parts corresponding to the block boundaries along mode i:

$$\boldsymbol{A}^{(i)} = [\boldsymbol{A}^{(i)T}_{(1)} \boldsymbol{A}^{(i)T}_{(2)} ... \boldsymbol{A}^{(i)T}_{(K_i)}]^T.$$

Given this partitioning, each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$, $\vec{k} = [k_1, \ldots, k_i, \ldots, k_N] \in \mathcal{K}$ can be described in terms of these sub-factors (Figure 2.4):

$$\boldsymbol{\mathcal{X}}_{\vec{k}} \approx \boldsymbol{I} \times_1 \boldsymbol{A}^{(1)}_{(k_1)} \times_2 \boldsymbol{A}^{(2)}_{(k_2)} \cdots \times_N \boldsymbol{A}^{(N)}_{(k_N)} \tag{2.3}$$

Moreover [51] shows that the current estimate of the sub-factor $\boldsymbol{A}^{(i)}_{(k_i)}$ can be revised using the update rule (for more details on the update rules please see [51]):

$$\boldsymbol{A}^{(i)}_{(k_i)} \longleftarrow \; \boldsymbol{T}^{(i)}_{(k_i)} \left(\boldsymbol{S}^{(i)}_{(k_i)}\right)^{-1} \tag{2.4}$$

where

$$\begin{aligned} \boldsymbol{T}^{(i)}_{(k_i)} &= \sum_{\vec{l} \in \{[*,\ldots,*,k_i,*,\ldots,*]\}} \boldsymbol{U}^{(i)}_{\vec{l}} \left(\boldsymbol{P}_{\vec{l}} \oslash (\boldsymbol{U}^{(i)T}_{\vec{l}} \boldsymbol{A}^{(i)}_{(k_i)})\right) \\ \boldsymbol{S}^{(i)}_{(k_i)} &= \sum_{\vec{l} \in \{[*,\ldots,*,k_i,*,\ldots,*]\}} \boldsymbol{Q}_{\vec{l}} \oslash \left(\boldsymbol{A}^{(i)T}_{(k_i)} \boldsymbol{A}^{(i)}_{(k_i)}\right) \end{aligned}$$

such that, given $\vec{l} = [l_1, l_2, \ldots, l_N]$, there is

- $\boldsymbol{P}_{\vec{l}} = \circledast_{h=1}^{N} (\boldsymbol{U}^{(h)T}_{\vec{l}} \boldsymbol{A}^{(h)}_{(l_h)})$ and
- $\boldsymbol{Q}_{\vec{l}} = \circledast_{h=1}^{N} (\boldsymbol{A}^{(h)T}_{(l_h)} \boldsymbol{A}^{(h)}_{(l_h)})$.

Above, $\circledast$ denotes the Hadamart product and $\oslash$ denotes the element-wise division operation.

While the precise derivation of the above update rule is not critical for the discussion (and is beyond the scope of this dissertation), it enables us to formulate the `2PCP` two-phase, block-based tensor decomposition process.

Chapter 3

NOISE-PROFILE ADAPTIVE TENSOR DECOMPOSITION IN BLOCK BASED FORMAT

3.1 Introduction

The problem tensor decomposition faces is that the process can be negatively affected from noise and low quality in the data, which is especially a concern for web-based user data [3, 60, 61, 12] – in particular, especially for sparse data, avoiding over-fitting to the noisy data can be a significant challenge. Recent research has shown that it is possible to avoid such over-fitting by relying on probabilistic techniques [59], which introduces priors on the parameters, it can effectively average over various models and ease the pain of tuning parameters. Unfortunately, existing probabilistic approaches have two major deficiencies: (a) firstly, they assume that all the data and intermediary results can fit in the main memory and (b) they treat the entire tensor uniformly, ignoring possible non-uniformities in the distribution of noise in the given tensor.

In this dissertation, I propose a *Noise-Profile Adaptive Tensor Decomposition (*`nTD`*)* method, which leverages a priori information about noise in the data (which may be user provided or obtained through automated techniques [54, 13]) to improve decomposition accuracy. nTD partitions the user data tensor into multiple sub-tensors (Figure 3.1) and then decomposes each sub-tensor probabilistically through Bayesian factorization – the resulting decompositions are then recombined to obtain the decomposition for the whole tensor. Most importantly, `nTD` *provides a resource allocation strategy that accounts for the impact of the noise density of one sub-tensor on the decomposition accuracies of the other sub-tensors.* In other words, *a priori* knowledge about noise distribution among the sub-tensors (noise pro-

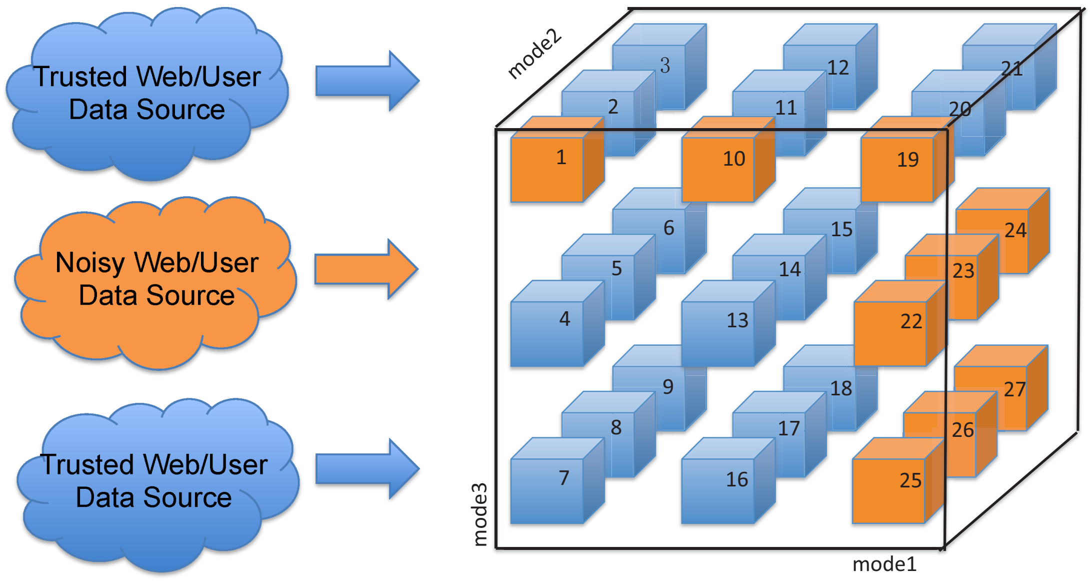

Trusted Web/User Data Source
Noisy Web/User Data Source
Trusted Web/User Data Source
mode1
mode2
mode3
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Figure 3.1: A Sample 3-mode Tensor, Partitioned into a Grid Of Sub-tensors, and Its Noise Profile: The Figure Highlights (in Orange) a Subset Of The Sub-tensors Which Are Noisy

files depicted in Figures 3.1 and 1.1) is used to obtain a resource assignment strategy that best suits the noise distribution of the given tensor.

3.2 Grid based Probabilistic Tensor Decomposition (GPTD)

Noise may not be uniformly distributed on a tensor. In order to take into account the underlying non-uniformities, I propose to partition the tensor into a grid and treat each grid partition differently based on its noise profile. In this section, I present a Grid Based Probabilistic Tensor Decomposition (GPTD) approach which extends the Probabilistic Tensor Decomposition (PTD [59]) into a grid-based framework. Note that, in and of itself, GPTD does not leverage **a priori** knowledge about noise distribution, but in Section 3.4, it provides a framework in which noise-profile based adaptation can be implemented.

Let us consider an N-mode tensor, $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\boldsymbol{\mathfrak{X}} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$, where $\mathcal{K}$ is the set of sub-tensor indexes. Without loss of generality, let us assume that $\mathcal{K}$ partitions the mode i into K_i equal partitions; i.e., $|\mathcal{K}| = \prod_{i=1}^{N} K_i$. Given a target decomposition rank, F, the first step of the proposed decomposition (`GPTD`) scheme is to decompose each sub-tensor in $\mathfrak{X}$ with target rank F, such that for each $\boldsymbol{\mathcal{X}}_{\vec{k}}$, there is $\boldsymbol{\mathcal{X}}_{\vec{k}} \approx \boldsymbol{I} \times_1 \boldsymbol{U}_{\vec{k}}^{(1)} \times_2 \boldsymbol{U}_{\vec{k}}^{(2)} \cdots \times_N \boldsymbol{U}_{\vec{k}}^{(N)}$, where $\mathfrak{U}^{(i)} = \{\boldsymbol{U}_{\vec{k}}^{(i)} \mid \vec{k} \in \mathcal{K}\}$ denotes the set of F-rank sub-factors[1] corresponding to the sub-tensors in $\mathfrak{X}$ along mode i and $\boldsymbol{I}$ is the N-mode $F \times F \times \ldots \times F$ identity tensor, where the diagonal entries are all 1s and the rest are all 0s. Intuitively, given a sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$, each entry $\boldsymbol{\mathcal{X}}_{\vec{k}(i_1,i_2,i_3,\ldots,i_N)}$ can be expressed as the inner-product of N F-dimensional vectors: $\boldsymbol{\mathcal{X}}_{\vec{k}(i_1,i_2,\ldots,i_N)} \approx [\boldsymbol{U}_{\vec{k}(i_1)}^{(1)}, \boldsymbol{U}_{\vec{k}(i_2)}^{(2)} \ldots, \boldsymbol{U}_{\vec{k}(i_N)}^{(N)}]$. I discuss the sub-tensor decomposition process next.

3.2.1 Phase 1: Monte Carlo based Bayesian Decomposition of Sub-tensors

For decomposing individual sub-tensors, the probabilistic approach proposed in [43, 59] is used: i.e., the fit between the observed data and the predicted latent factor matrices, probabilistically, is described as follows:

$$\begin{aligned}\boldsymbol{\mathcal{X}}_{\vec{k}(i_1,i_2,\ldots,i_N)} \Big| \boldsymbol{U}_{\vec{k}}^{(1)}, \boldsymbol{U}_{\vec{k}}^{(2)} \ldots, \boldsymbol{U}_{\vec{k}}^{(N)} \\ \sim \boldsymbol{\mathcal{N}}([\boldsymbol{U}_{\vec{k}(i_1)}^{(1)}, \boldsymbol{U}_{\vec{k}(i_2)}^{(2)} \ldots, \boldsymbol{U}_{\vec{k}(i_N)}^{(N)}], \alpha^{-1}),\end{aligned} \tag{3.1}$$

where the conditional distribution of $\boldsymbol{\mathcal{X}}_{\vec{k}(i_1,i_2,\ldots,i_N)}$ given $\boldsymbol{U}_{\vec{k}}^{(j)}$ $(1 \leq j \leq N)$ is a Gaussian distribution with mean $[\boldsymbol{U}_{\vec{k}(i_1)}^{(1)}, \boldsymbol{U}_{\vec{k}(i_2)}^{(2)}, \ldots, \boldsymbol{U}_{\vec{k}(i_N)}^{(N)}]$ and the observation precision α. independent Gaussian priors is imposed on the modes:

$$\boldsymbol{U}_{\vec{k}(i_j)}^{(j)} \sim \boldsymbol{\mathcal{N}}(\mu_{\boldsymbol{U}_{\vec{k}}^{(j)}}, \Lambda_{\boldsymbol{U}_{\vec{k}}^{(j)}}^{-1}) \quad i_j = 1 \ldots I_j \tag{3.2}$$

where I_j is the dimensionality of the j^{th} mode. Given this, one can estimate the latent features $\boldsymbol{U}_{\vec{k}}^{(j)}$ by maximizing the logarithm of the posterior distribution, $\log p(\boldsymbol{U}_{\vec{k}}^{(1)}, \boldsymbol{U}_{\vec{k}}^{(2)} \ldots, \boldsymbol{U}_{\vec{k}}^{(N)} | \boldsymbol{\mathcal{X}}_{\vec{k}})$. One difficulty with this approach, however, is the tuning of the hyper-parameters of the

[1]If the sub-tensor is empty, then the factors are $\mathbf{0}$ matrices of the appropriate size.

Algorithm 1 Phase 1: Monte Carlo based Bayesian decomposition of each sub-tensor(extension of [59] to more than 3 modes)

```
Input: Sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$, sampling number $L$
Output: Decomposed factors $\boldsymbol{U}_{\vec{k}}^{(1)}$, $\boldsymbol{U}_{\vec{k}}^{(2)}$, ..., $\boldsymbol{U}_{\vec{k}}^{(N)}$

1. Initialize model parameters $\boldsymbol{U}_{\vec{k}}^{(1)1}$, $\boldsymbol{U}_{\vec{k}}^{(2)1}$, ..., $\boldsymbol{U}_{\vec{k}}^{(N)1}$ .
2. For $l = 1, \dots, L$
    (a) Sample the hyper-parameter, $\alpha$:
        • $\alpha^l \sim p(\alpha^l | \boldsymbol{U}_{\vec{k}}^{(1)l}, \boldsymbol{U}_{\vec{k}}^{(2)l}, \dots, \boldsymbol{U}_{\vec{k}}^{(N)l}, \boldsymbol{\mathcal{X}}_{\vec{k}})$
    (b) For each mode $j = 1, \dots, N$,
        i. Sample the corresponding hyper-parameter, $\Theta$:
            • $\Theta_{\boldsymbol{U}_{\vec{k}}^{(j)l}} \sim p(\Theta_{\boldsymbol{U}_{\vec{k}}^{(j)l}} | \boldsymbol{U}_{\vec{k}}^{(j)l})$
        ii. For $i_j = 1, ..., I_j$, sample the mode (in parallel):
            $\boldsymbol{U}_{\vec{k}(i_j)}^{(j)(l+1)} \sim p\Big(\boldsymbol{U}_{\vec{k}(i_j)}^{(j)} \Big| \boldsymbol{U}_{\vec{k}}^{(1)l}, \dots, \boldsymbol{U}_{\vec{k}}^{(j-1)l}, \boldsymbol{U}_{\vec{k}}^{(j+1)l}, \dots, \boldsymbol{U}_{\vec{k}}^{(N)l}, \Theta^l_{\boldsymbol{U}_{\vec{k}}^{(j)}}, \alpha^l, \boldsymbol{\mathcal{X}}_{\vec{k}}\Big)$
3. For each mode $j = 1, \dots, N$,
    • $\boldsymbol{U}_{\vec{k}}^{(j)} = \frac{\sum_{i=1}^{L} \boldsymbol{U}_{\vec{k}}^{(j)i}}{L}$
```

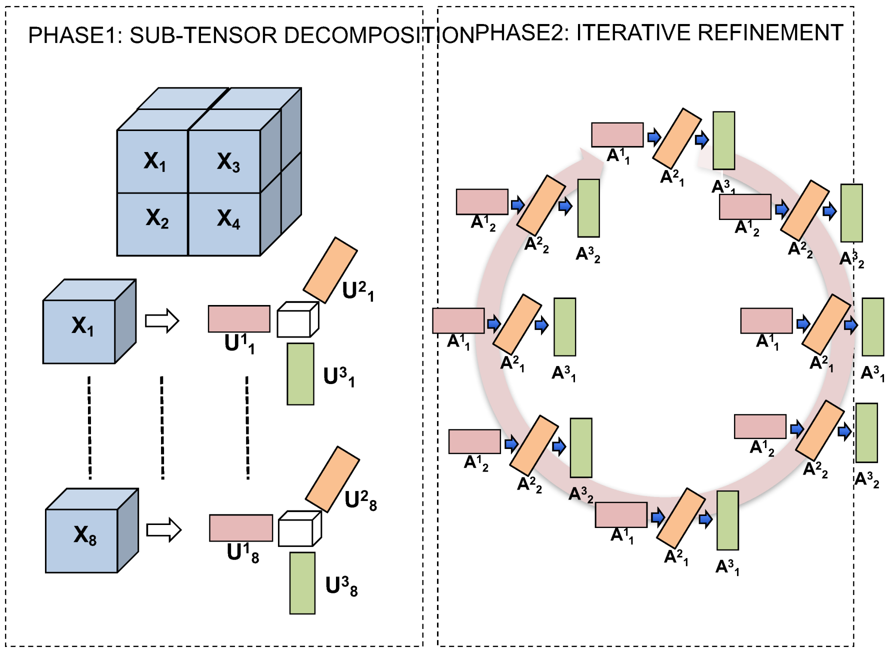

PHASE1: SUB-TENSOR DECOMPOSITION
PHASE2: ITERATIVE REFINEMENT
X1
X3
X2
X4
X8
U11
U21
U31
U18
U28
U38
A11
A21
A31
A12
A22
A32

Figure 3.2: Illustration of Sub-tensor Based Tensor Decomposition: The Input Tensor Is Partitioned into Smaller Blocks, Each Block Is Decomposed (Potentially in Parallel), and the Partial Decompositions Are Stitched Together Through an Iterative Improvement Process

model: α and $\Theta_{\boldsymbol{U}_{\vec{k}}^{(j)}} \equiv \{\mu_{\boldsymbol{U}_{\vec{k}}^{(j)}}, \Lambda_{\boldsymbol{U}_{\vec{k}}^{(j)}}\}$ for $1 \leq j \leq N$. [59] notes that one can avoid the difficulty underlying the estimation of these parameters through a fully Bayesian approach, complemented with a sampling-based Markov Chain Monte Carlo (MCMC) method to address the lack of the analytical solution. The process is visualized in Algorithm 1 in pseudo-code form.

3.2.2 Phase 2: Iterative Refinement

Once the individual sub-tensors are decomposed, the next step is to stitch the resulting sub-factors into the full F-rank factors, $\boldsymbol{A}^{(i)}$ (each one along one mode), for the input tensor, $\boldsymbol{\mathcal{X}}$. Let us partition each factor $\boldsymbol{A}^{(i)}$ into K_i parts corresponding to the block boundaries along mode i:

$$\boldsymbol{A}^{(i)} = [\boldsymbol{A}^{(i)T}_{(1)} \boldsymbol{A}^{(i)T}_{(2)} ... \boldsymbol{A}^{(i)T}_{(K_i)}]^T.$$

Given this partitioning, each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$, $\vec{k} = [k_1, \dots, k_i, \dots, k_N] \in \mathcal{K}$ can be described in terms of these sub-factors:

$$\boldsymbol{\mathcal{X}}_{\vec{k}} \approx \boldsymbol{I} \times_1 \boldsymbol{A}^{(1)}_{(k_1)} \times_2 \boldsymbol{A}^{(2)}_{(k_2)} \cdots \times_N \boldsymbol{A}^{(N)}_{(k_N)} \tag{3.3}$$

The current estimate of the sub-factor $\boldsymbol{A}^{(i)}_{(k_i)}$ can be revised using the following update rule [51]:

$$\boldsymbol{A}^{(i)}_{(k_i)} \longleftarrow \ \boldsymbol{T}^{(i)}_{(k_i)} \left(\boldsymbol{S}^{(i)}_{(k_i)} \right)^{-1} \tag{3.4}$$

where

$$\begin{aligned} \boldsymbol{T}^{(i)}_{(k_i)} &= \sum_{\vec{m} \in \{[*,\dots,*,k_i,*,\dots,*]\}} \boldsymbol{U}^{(i)}_{\vec{m}} \left(\boldsymbol{P}_{\vec{m}} \oslash (\boldsymbol{U}^{(i)T}_{\vec{m}} \boldsymbol{A}^{(i)}_{(k_i)}) \right) \\ \boldsymbol{S}^{(i)}_{(k_i)} &= \sum_{\vec{m} \in \{[*,\dots,*,k_i,*,\dots,*]\}} \boldsymbol{Q}_{\vec{m}} \oslash \left(\boldsymbol{A}^{(i)T}_{(k_i)} \boldsymbol{A}^{(i)}_{(k_i)} \right) \end{aligned}$$

such that, given $\vec{m} = [m_1, m_2, \dots, m_N]$, there is

- $\boldsymbol{P}_{\vec{m}} = \circledast_{h=1}^{N} (\boldsymbol{U}^{(h)T}_{\vec{m}} \boldsymbol{A}^{(h)}_{(m_h)})$ and
- $\boldsymbol{Q}_{\vec{m}} = \circledast_{h=1}^{N} (\boldsymbol{A}^{(h)T}_{(m_h)} \boldsymbol{A}^{(h)}_{(m_h)})$.

Above, $\circledast$ denotes the Hadamart product and $\oslash$ denotes the element-wise division operation.

3.3 Overview of GPTD

The two phases of the decomposition process are visualized in Algorithm 2 and Figure 3.2.

3.4 Noise-Profile Adaptive Tensor Decomposition

One crucial piece of information that the basic grid based decomposition process fails to account for is *potentially available knowledge* about the distribution of the noise across

Algorithm 2 The outline of the `GPTD` process

Input: Input tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, and decomposition rank, F, and per sub-tensor sample count, L

Output: Tensor decomposition $\mathring{\boldsymbol{\mathcal{X}}}$

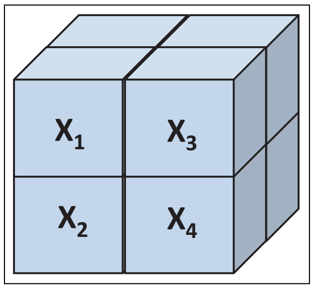
```
1. Phase 1: for all $\vec{k} \in \mathcal{K}$
     • decompose $\boldsymbol{\mathcal{X}}_{\vec{k}}$ into $\boldsymbol{U}^{(1)}_{\vec{k}}$, $\boldsymbol{U}^{(2)}_{\vec{k}}$, ..., $\boldsymbol{U}^{(N)}_{\vec{k}}$
   with sample count $L$ using Algorithm 1.
2. Phase 2: repeat
     (a) for each mode $i = 1$ to $N$
           i. for each modal partition, $k_i = 1$ to $K_i$,
                A. update $\boldsymbol{A}^{(i)}_{(k_i)}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$, for each block $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$; more specifically,
                   • compute $\boldsymbol{T}^{(i)}_{(k_i)}$, which involves the use of $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$)
                   • revise $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
                   • compute $\boldsymbol{S}^{(i)}_{(k_i)}$ using the above
                   • update $\boldsymbol{A}^{(i)}_{(k_i)}$ using the above
                   • for each $\vec{k} = [*, \ldots, *, k_i, *, \ldots, *]$
                     – update $\boldsymbol{P}_{\vec{k}}$ and $\boldsymbol{Q}_{\vec{k}}$ using
                     – $\boldsymbol{U}^{(i)}_{\vec{k}}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
   until stopping condition
3. Return $\mathring{\boldsymbol{\mathcal{X}}}$
```

the input tensor. Note that, in the second phase of the process, each $\boldsymbol{A}^{(i)}_{(k_i)}$ is maintained incrementally by using, for all $1 \leq j \leq N$, (a) the current estimates for $\boldsymbol{A}^{(j)}_{(k_j)}$ and (b) the decompositions in $\boldsymbol{\mathfrak{U}}^{(j)}$; i.e., the F-rank sub-factors of the sub-tensors in $\boldsymbol{\mathfrak{X}}$ along the different modes of the tensor. This implies that a sub-tensor which is poorly decomposed due to noise may negatively impact decomposition accuracies also for other parts of the tensor. Consequently, it is important to properly allocate resources to prevent a few noisy sub-tensors among all from negatively impacting the overall accuracy.

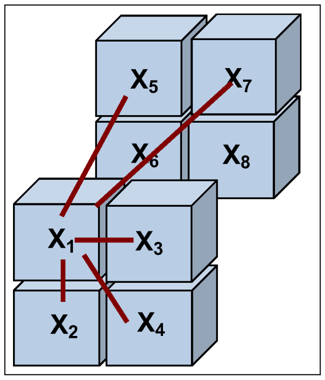
In [39],the approach of how to allocate resources is introduced, in a way that takes into account, user's non-uniform accuracy preferences for different parts of the tensor. In this dissertation, I develop a novel noise-profile adaptive tensor decomposition (`nTD`) scheme that focuses on resource allocation based on noise distribution. More specifically, user provided or automatically discovered [1, 60, 61] a priori knowledge about the noise profiles of the grid partitions enables us to develop a sample assignment strategy (or `s-strategy`) that best suits the noise distribution in a given tensor. In particular, `nTD` assigns the ranks and samples to different sub-tensors in a way that maximizes the overall decomposition accuracy of the whole tensor without negatively impacting the efficiency of the decomposition process. Since probabilistic decomposition can be costly, `nTD` considers *a priori* knowledge about each sub-tensor's noise density to decide the appropriate number of Gibbs samples to achieve good accuracy with the given number of samples.

3.4.1 Noise Sensitive Sample Assignment: First Naive Attempt

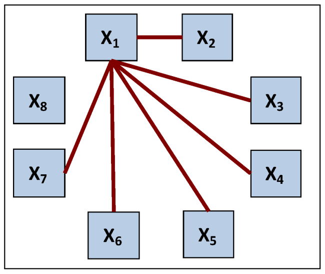
As experiments shown, there is a direct relationship between the amount of noise a (sub-)tensor has and the number of Gibbs samples it requires for accurate decomposition. On the other hand, the number of samples also directly impacts the cost of the probabilistic decomposition process. Consequently, given a set of sub-tensors, with different amounts of



(a) A $2 \times 2 \times 2$ grid



(b) Corresponding sub-tensors



(c) Pairwise refinement dependencies

Figure 3.3: A Sample Grid and the Corresponding Pairwise Refinement Dependencies among the Sub-tensors per Equation 3.4

noise, uniform assignment of the number of samples, $L = \left(\frac{L_{(total)}}{|\mathcal{K}|}\right)$, where $L_{(total)}$ is the total number of samples for the whole tensor and $|\mathcal{K}|$ is the number of sub-tensors, may not be the best choice.

In fact, the numbers of Gibbs samples allocated to different sub-tensors $\boldsymbol{\mathcal{X}}_{\vec{k}}$ in Algorithm 1 *do not need to be the same.* In Section 3.2.1, Phase 1 decomposition of each sub-tensor is independent from the others and, thus, the number of Gibbs samples of different sub-tensors can be different. This observation, along with observation that more samples can provide better accuracy for noisy sub-tensors, can be used to improve the overall decomposition accuracy for a given number of Gibbs samples. More specifically, the number of samples a noisy sub-tensor, $\boldsymbol{\mathcal{X}}_{\vec{k}}$, is allocated should be proportional to the density, $nd_{\vec{k}}$, of noise it contains:

$$L(\boldsymbol{\mathcal{X}}_{\vec{k}}) = \lceil \gamma \times nd_{\vec{k}} \rceil + L_{min}, \tag{3.5}$$

where L_{min} is the minimum number of samples a (non-noisy) tensor of the given size would need for accurate decomposition and γ is a control parameter. Note that the value of γ is selected such that the total number of samples needed is equal to the number, $L_{(total)}$, of samples allocated for the whole tensor:

$$L_{(total)} = \sum_{\vec{k} \in \mathcal{K}} L(\boldsymbol{\mathcal{X}}_{\vec{k}}). \tag{3.6}$$

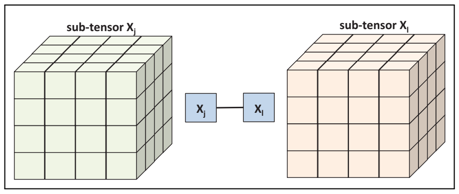
3.4.2 Noise Sensitive Sample Assignment: Second Naive Attempt

Equations 3.5 and 3.6, above, help allocate samples across sub-tensors based on their noise densities. However, they ignore the relationships among the sub-tensors. In Section 3.2.2, during the iterative refinement process of Phase 2, inaccuracies in decomposition of one sub-tensor can propagate across the rest of the sub-tensors. Therefore, a better approach could be to consider how errors can propagate across sub-tensors when allocating samples.

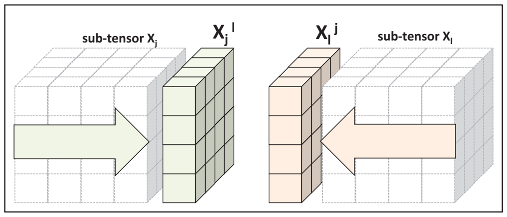
A. Accounting for Accuracy Inter-dependencies among Sub-Tensors More specifically, in this section, if a significance score is assigned to each sub-tensor, $\boldsymbol{\mathcal{X}}_{\vec{k}}$, that takes into account not only its noise density, but also the position of the sub-tensor relative to other sub-tensors, this information could be used to allocate samples.

Let $\boldsymbol{\mathcal{X}}$ be a tensor partitioned into a set (or grid) of sub-tensors $\boldsymbol{\mathfrak{X}} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$. According to the update rule (Equation 3.4) in Section 3.2.2, if two sub-tensors are lined up along one of the modes of the tensor, they can be used to revise each other's estimates. This means that the update rule ties each sub-tensor's accuracy directly to $\sum_{1 \leq i \leq N} K_i$ other sub-tensors (that line up with the given sub-tensor along one of the N modes – see Figure 3.3).

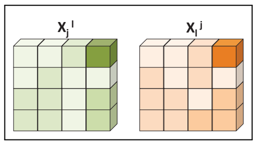
Moreover, if the two sub-tensors are similarly distributed along the modes that they share, then they are likely to have high impacts on each other's decomposition; in contrast, if they are dissimilar, their impacts on each other will also be minimal. In other words, given two sub-tensors $\boldsymbol{\mathcal{X}}_{\vec{j}}$ and $\boldsymbol{\mathcal{X}}_{\vec{l}}$, an *alignment* score can be computed, $align(\boldsymbol{\mathcal{X}}_{\vec{j}}, \boldsymbol{\mathcal{X}}_{\vec{l}})$, between $\boldsymbol{\mathcal{X}}_{\vec{j}}$ and $\boldsymbol{\mathcal{X}}_{\vec{l}}$ as $align(\boldsymbol{\mathcal{X}}_{\vec{j}}, \boldsymbol{\mathcal{X}}_{\vec{l}}) = cos(\boldsymbol{\mathcal{X}}_{\vec{j}}^{\vec{l}} \boldsymbol{\mathcal{X}}_{\vec{l}}^{\vec{j}})$, where $cos()$ is the cosine similarity function and $\boldsymbol{\mathcal{X}}_{\vec{a}}^{\vec{b}}$ is the version of the sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{a}}$ compressed, using the standard Frobenius norm, onto the modes along which the sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{a}}$ and $\boldsymbol{\mathcal{X}}_{\vec{b}}$ are aligned (Figure 3.4). Intuitively,

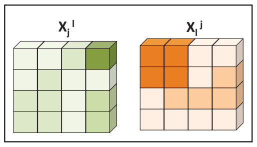


(a) Two sub-tensors with pairwise impact



(b) Their compressions onto shared modes





(c) Well-aligned sub-tensors (d) Poorly aligned sub-tensors

Figure 3.4: Measuring the Alignment of Two Sub-tensors: (A) The Sub-tensors with Pairwise Impact, (B) Their Compressions onto Their Shared Modes, (C) Well-aligned Tensors Have Similar Distributions On This Compressed Representation, Whereas (D) Poorly Aligned Tensors Have Dissimilar Distributions

this pairwise alignment score describes how the decomposition of one sub-tensor will impact another and also indicate the degree of numeric error propagation. A sub-tensor which is not aligned with the other sub-tensors is likely to have minimal impact on the accuracy of the overall decomposition even if it contains significant amount of noise. In contrast, a sub-tensor which is well-aligned with a larger portion of other sub-tensors may have a large impact on the other sub-tensors, and consequently, on the whole tensor. Consequently, while the former sub-tensor may not deserve a significant amount of resources, the accuracy of the latter sub-tensor is critical and hence that tensor should be allocated more resources to ensure better overall accuracy.

B. Sub-Tensor Centrality based Sample Assignment Therefore, given pairwise alignment scores among the sub-tensors, one option is to measure the significance of a sub-tensor relative to other sub-tensors using a centrality measure like PageRank (PR [8]), which computes the significance of each node in a (weighted) graph relative to the other nodes. More specifically, given a graph, $G(V, E)$, the PageRank score $\vec{p}[i]$, of a node $v_i \in V$ is obtained by solving $\vec{p} = (1-\beta)\mathbf{A}\ \vec{p} + \beta\vec{s}$, where $\mathbf{A}$ denotes the transition matrix, β is a parameter controlling the random walk likelihood , and $\vec{s}$ is a teleportation vector such that for $v_j \in V$, $\vec{s}[j] = \frac{1}{\|V\|}$. Therefore, given (a) the set (or grid) of sub-tensors $\mathfrak{X} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ and (b) their pairwise alignment scores, a significance score can be associated,

$$\tau_{\vec{k}} = \frac{\vec{p}[\vec{k}] - min_{\vec{j}\in\mathcal{K}}(\vec{p}[\vec{j}])}{max_{\vec{j}\in\mathcal{K}}(\vec{p}[\vec{j}]) - min_{\vec{j}\in\mathcal{K}}(\vec{p}[\vec{j}])},$$

to each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ by computing PageRank scores described by the vector $\vec{p}$. Given this score, Equation 3.5 can be rewritten as

$$L(\boldsymbol{\mathcal{X}}_{\vec{k}}) = \lceil \gamma \times \tau_{\vec{k}} \times nd_{\vec{k}} \rceil + L_{min}, \tag{3.7}$$

taking into account both the noise density of the sub-tensor along with its relationship to other sub-tensors.

3.4.3 S-Strategy for Sample Assignment

The above formulation considers the position of each sub-tensor in the whole sub-tensor to compute its significance and then multiplies this with the corresponding noise density to decide how much resources to allocate to that sub-tensor. This, however, may not properly take into account the relationship among the noisy sub-tensors and the positioning of sub-tensors relative to the noisy ones.

In this dissertation, a better approach would be to consider the noise densities of the sub-tensors directly when evaluating the significance of each sub-tensor. More specifically, instead of relying on PageRank, I propose to use a measure like personalized PageRank (PPR [11]), which computes the significance of each node in a (weighted) graph relative to a given set of seed nodes. Given a graph, $G(V, E)$, and a set, $S \subseteq G(V, E)$, of seed nodes, the PPR score $\vec{p}[i]$, of a node $v_i \in G(V, E)$ is obtained by solving $\vec{p} = (1 - \beta)\mathbf{A}\ \vec{p} + \beta\vec{s}$, where $\mathbf{A}$ denotes the transition matrix, β is a parameter controlling the overall importance of the seeds, and $\vec{s}$ is a seeding vector such that if $v_i \in S$, then $\vec{s}[i] = \frac{1}{\|S\|}$ and $\vec{s}[i] = 0$, otherwise. Therefore, given (a) the set (or grid) of sub-tensors $\mathfrak{X} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$, (b) their pairwise alignment scores, and (c) a seeding vector

$$\vec{s}[\vec{k}] = \frac{nd_{\vec{k}}}{\sum_{\vec{j} \in \mathcal{K}} nd_{\vec{j}},}$$

A noise sensitive significance score is associated,

$$\eta_{\vec{k}} = \frac{\vec{p}[\vec{k}] - min_{\vec{j} \in \mathcal{K}}(\vec{p}[\vec{j}])}{max_{\vec{j} \in \mathcal{K}}(\vec{p}[\vec{j}]) - min_{\vec{j} \in \mathcal{K}}(\vec{p}[\vec{j}])},$$

to each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ based on the PPR scores, described by the vector $\vec{p}$, relative to the noisy tensors. Given this score, Equation 3.5 is rewritten as

$$L(\boldsymbol{\mathcal{X}}_{\vec{k}}) = \lceil \gamma \times \eta_{\vec{k}} \rceil + L_{min}. \tag{3.8}$$

3.5 Overview of nTD

The pseudo-code of the proposed noise adaptive tensor decomposition (`nTD`) process is visualized in Algorithm 3.

3.6 Experimental Evaluation

In this section, experiments is reported, which aim to assess the effectiveness of the proposed *noise adaptive tensor decomposition* approach. In particular, the proposed approach is compared against another grid based strategy, GridParafac.I further assess the proposed

Algorithm 3 Overview of `nTD`: noise adaptive decomposition (with noise based resource allocation)

```
Input: original tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, noisy sub-tensor $\mathcal{K}_P$, and decomposition rank,
    $F$ and total sampling number $L$
Output: tensor decomposition, $\hat{\boldsymbol{\mathcal{X}}}$
  1. obtain the noise profile of the sub-tensors of $\boldsymbol{\mathcal{X}}$,
  2. for sub-tensor $\vec{k} \in \mathcal{K}$, assign a decomposition rank $F_{\vec{k}} = F$ and a sampling number $L_{\vec{k}}$ based
     on noise-sensitive sample allocation strategy, described in Section 3.4.3.
  3. obtain the decomposition, $\hat{\boldsymbol{\mathcal{X}}}$, of $\boldsymbol{\mathcal{X}}$ using the GPTD algorithm (Algorithm 2), given parti-
     tioning pattern $\mathcal{K}$ and the initial decomposition ranks $\{F_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ and sampling number
     $\{L_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$,
  4. Return $\hat{\boldsymbol{\mathcal{X}}}$
```

noise-sensitive sample assignment strategy (`s-strategy`) by comparing the performance of `nTD`, which leverages this strategy, against `GPTD` with uniform sample assignment, on user-centered data.

3.6.1 Experiment Setup

Key parameters and their values are reported in Table 3.1.

Data Sets. In these experiments, I used three user centered datasets: *Epinions* [57], *Ciao* [57], and *MovieLens* [21, 22]. The first two of these are comparable in terms of their sizes and semantics: they are represented in the form of $5000 \times 5000 \times 999$ (density 1.4×10^{-6}) and $5000 \times 5000 \times 996$ (density 1.7×10^{-6}) tensors, respectively, and both have the schema $\langle user, item, time \rangle$. The *MovieLens* data set ($943 \times 1682 \times 2001$, density 3.15×10^{-5}) is denser and has a different schema, $\langle user, movie, time \rangle$. In all three data sets, the tensor cells contain

Parameters	*Alternative values*
Noise Density	**10**%; 30%; 50%; 80%
# partitions	$2 \times 2 \times 2$; $\mathbf{4 \times 4 \times 4}$
Per sub-tensor Gibbs sample count	1; **3**; 5; 10; 30; 80
Target Rank (F)	**10**

Table 3.1: Parameters – Default Values, Used Unless Otherwise Specified, Are Highlighted

rating values between 1 and 5 or (if the rating does not exist) a special “null” symbol.

Noise. In these experiments, uniform value-independent type of noise were introduced by modifying the existing ratings in the data set[2]. More specifically, given a uniform noise profile and density, I have selected a subset of the existing ratings (ignoring “null” values) and altered the existing values – by selecting a completely new rating (which I refer to as *value-independent noise*).

Evaluation Criteria. I use the *root mean squares error* (RMSE) inaccuracy measure to assess the decomposition effectiveness. I also report the decomposition times and memory consumptions. Unless otherwise reported, the execution time of the overall process is reported as if sub-tensor decompositions in Phase 1 and Phase 2 are all executed serially, without leveraging any sub-tensor parallelism. Each experiment was run 10 times with different random noise distributions and averages are reported.

Hardware and Software. I ran experiments on a quad-core CPU Nehalem Node with 12.00GB RAM. All codes were implemented in Matlab and run using Matlab R2015b. For conventional CP decomposition, I used MATLAB Tensor Toolbox Version 2.6 [2].

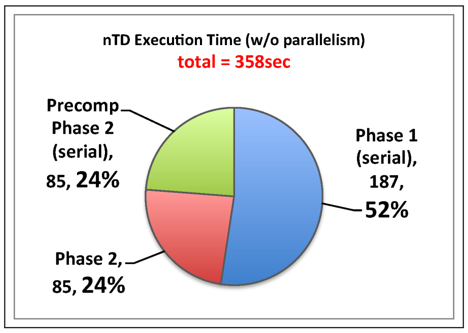
[2]Because of space limitations, I do not include results with slice-concentrated, multi-modal, and value-dependent noise; but the results for those types of results are similar to the results presented in this section.

3.6.2 Discussion of the Results

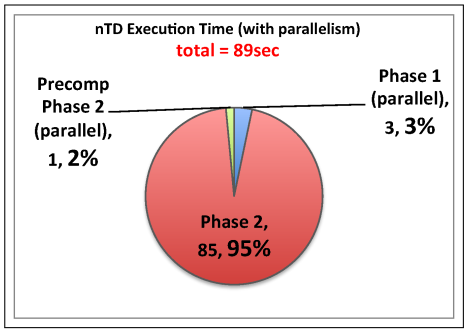
I start the discussion of the results by studying the impact of the `s-strategy` for leveraging noise profiles.

Impact of Leveraging Noise Profiles. In Figure 4.7, I compare the performance of `nTD` with noise-sensitive sample assignment (i.e., `s-strategy`) against `GPTD` with *uniform* sample assignment and the two naive noise adaptations, presented in Sections 3.4.1 and 3.4.2, respectively. Note that in the scenario considered in this figure, I have 640 total Gibbs samples for 64 sub-tensors, providing on the average 10 samples per sub-tensor. In these experiments, I set L_{min} to 9 (i.e. very close to this average), thus requiring that $576 (= 64 \times 9)$ samples are uniformly distributed across the sub-tensors – this leaves only 64 samples to be distributed adaptively across the sub-tensors based on the noise profiles of the sub-tensors and their relationships to other sub-tensors. In this figure, the proposed `nTD` is able to leverage these 64 uncommitted samples to significantly reduced RMSE relative to `GPTD` with *uniform* sample assignment. Moreover, I also see that naive noise adaptations can actually hurt the overall accuracy. These together show that the proposed `s-strategy` is highly effective in leveraging rough knowledge about noise distributions to better allocate the Gibbs samples across the tensor. Note that, as expected, `nTD` is costlier than `GPTD` as it requires additional preprocessing to compute sub-tensor alignments in Phase 2. However, the required pre-processing is trivially parallelizable as discussed next.

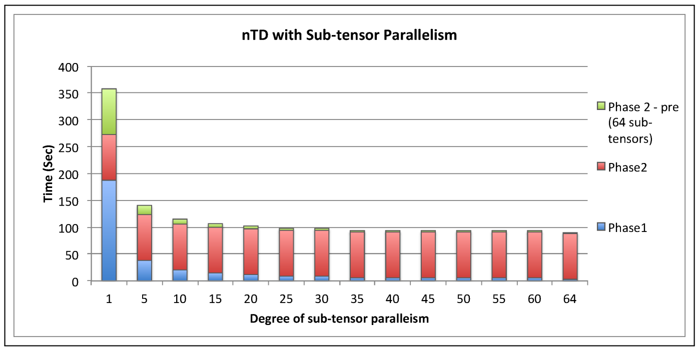
Impact of Sub-Tensor Parallelism. In Figure 3.5, Phase 1 of the `nTD` algorithm (Algorithm 1) is highly parallelizable as the sub-tensors resulting from grid-partitioning can be decomposed in parallel. Similarly, the pre-processing needed for computing the sample assignment strategy in Phase 2 is also highly parallelizable: the most expensive step of the process is the compression of the sub-tensors on modes shared with their neighbors (since the resulting sub-tensor graph is small, the PPR computation has negligible cost) and that work



(a) No sub-tensor parallelism



(b) sub-tensor parallelism

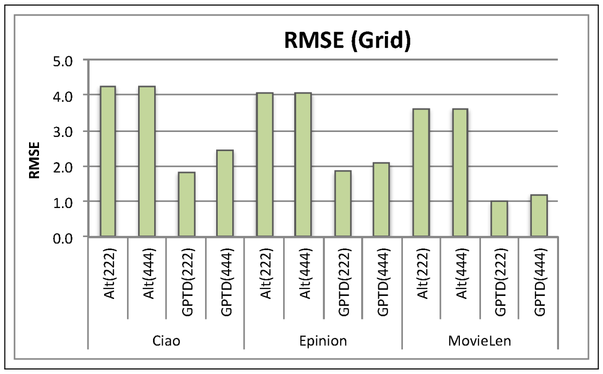


(c) Execution time with different degrees of parallelism

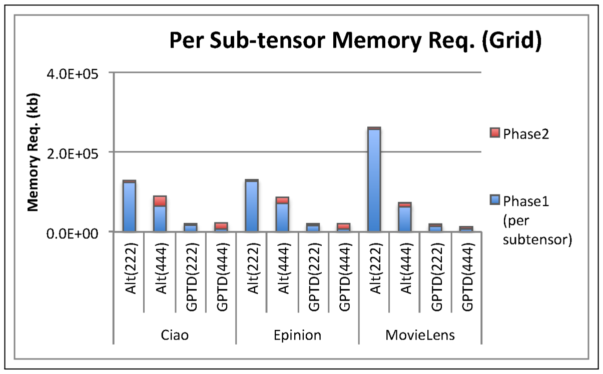
Figure 3.5: Impact of Sub-tensor Parallelism on `Ntd` ($4 \times 4 \times 4$ Grid; Uniform and Value Independent Noise; Noise Density 10%; $f = 10$; Num. Gibbs Samples per Sub-tensor = 3; Max. Num. Of P2 Iteration = 1000; 4 Sub-tensors with Noise; Ciao)

can be done in parallel for each sub-tensor or even for each cell in the resulting compressed representation. Unfortunately, Phase 2, involving incremental stitching and refinement of the factor matrices (see Algorithm 2) cannot be trivially parallelised by assigning different sub-tensors to different processors as the refinement rules need to simultaneously access data from multiple sub-tensors. While parallel execution of Phase 2 may be achieved by scheduling the refinement rules asynchronously in parallel, I leave the investigation of this as future work.

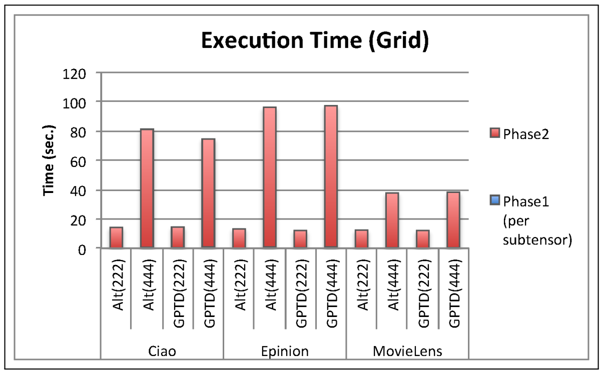
GPTD vs. GridParafac in the Presence of Noise In its Phase 1, `nTD` relies on grid based probabilistic decomposition strategy. I next compare this grid probabilistic tensor



(a) RMSE

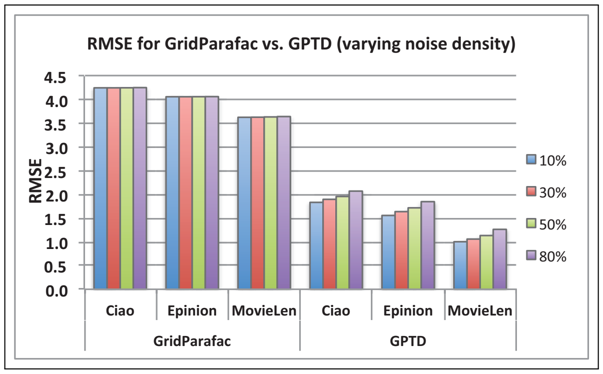


(b) Memory requirement

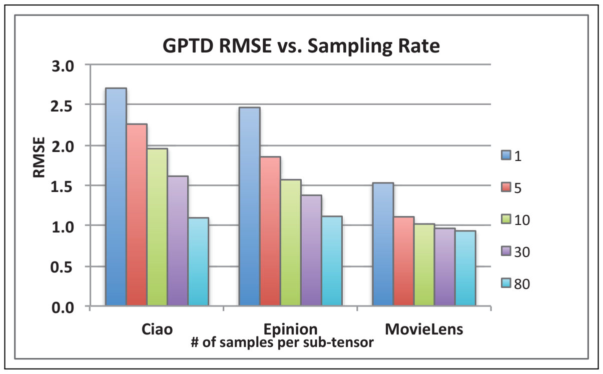


(c) Execution time

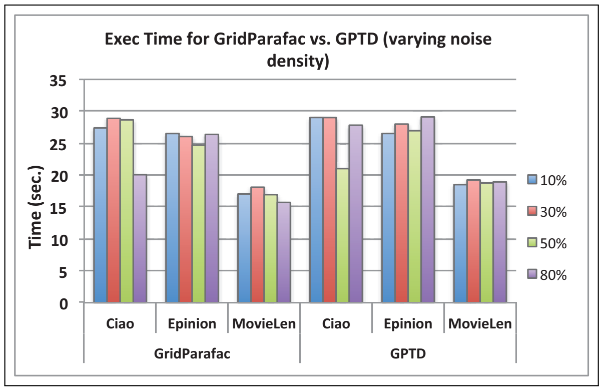
Figure 3.6: GPTD Vs. GridParafac Alternative (Denoted as "alt"); (Uniform Noise; Value Independent Noise; Noise Density 10%; f = 10; Num. Gibbs Samples per Sub-tensor = 3; Max. Num. Of P2 Iteration = 1000; 4 Sub-tensors with Noise)



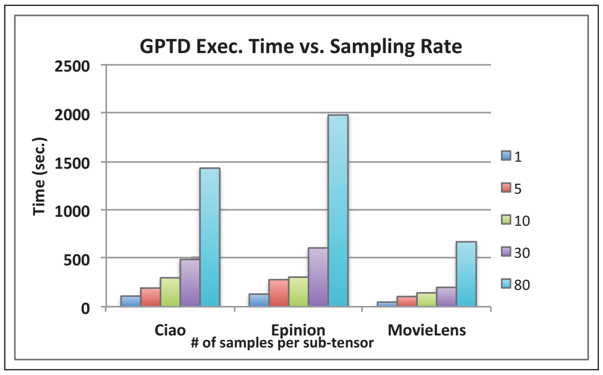
(a) RMSE



(b) RMSE



(c) Execution Time



(d) Execution Time

Figure 3.7: **(a)**`GPTD` vs. GridParafac ($2 \times 2 \times 2$ Grid; Varying Noise Density; Uniform Noise; Value Independent Noise; Num. Gibbs Samples per Sub-tensor $= 3$; $f = 10$; Max. Num. Of P2 Iteration $= 1000$); **(b)** `GPTD` with Different Num. Of Gibbs Samples ($4 \times 4 \times 4$ Grid; Uniform Noise; Value Independent Noise; Noise Density 10%; $f = 10$; Max. Num. Of P2 Iteration $= 1000$

decomposition (GPTD) against the more conventional GridParafac . In Figure 3.6, GPTD provides significantly better accuracy than the conventional approaches and also requires significantly lesser memory. As expected, I also see that increasing the number of sub-tensors results in a significant drop in the per-sub-tensor memory requirement (therefore improving the scalability of the tensor decomposition process) – though the execution time of the second phase of the process (where the initial decompositions of the sub-tensors are stitched together) increases due to the existence of more sub-tensors to consider.

An important observation in Figure 3.6 (b) is that the memory requirement for the conventional techniques is very sensitive to data density: While the *MovieLens* tensor has smaller dimensionality then the other two, it has a slightly higher density (3.15×10^{-5} vs. 1.7×10^{-6}). Consequently, for this data set, the memory consumptions of the conventional techniques (especially when the number of grid partitions used are low) are significantly higher than their memory consumptions for the other two data sets. In contrast, the results show that the probabilistic approach is not sensitive to data density and GPTD has similar memory usage for all three data sets.

Impact of Noise Density. These results are confirmed in Figure 3.7(a) & (c), where I vary the noise density between 10% and 80%: for all considered noise densities and for all three data sets, the RMSE provided by GPTD is significantly better than the RMSE provided by the conventional GridParafac and this RMSE gain does not come with a significant execution time penalty.

Impact of Numbers of Samples. A key parameter of the GPTD algorithm is the number of Gibbs samples used per

sub-tensor in Phase 1. In Figure 3.7(b)&(d), as expected, increasing the number of Gibbs samples helps reduce the decomposition error (measured using RMSE) ; however having more samples increases the execution time of the algorithm. It is important to note that, when the number of Gibbs samples is low, the algorithm is very fast, indicating that the worst case complexity of the Bayesian iterations arises only when the number of Gibbs samples is very high. Most critically, in Figures 3.6 and 3.7(a), the `GPTD` algorithm does not need too many Gibbs samples: using a few (in these experiments, even just 1) Gibbs samples per sub-tensor is sufficient to provide significantly better accuracy than GridParafac, as reported in Figures 3.6 (a), with similar or better time overhead, as reported in Figure 3.6 (c).

Chapter 4

NOISE ADAPTIVE TENSOR DECOMPOSITION IN TENSOR TRAIN FORMAT

4.1 Introduction

In this chapter, *Noise-Profile Adaptive Tensor Train Decomposition (*`NTTD`*)* method is proposed, which leverages *rough* a prior information about noise in the data (which may be user provided or obtained through automated techniques [54, 13]) to improve decomposition accuracy. `NTTD` decomposes each mode matricization probabilistically through Bayesian factorization – the resulting factor matrix are then reconstructed to obtain the tensor approximations. Most importantly,

> `NTTD` *provides a resource allocation strategy, which accounts for the impacts of (a) the noise density of each mode and (b) inherent approximation error of the Tensor Train decomposition process, on the overall decomposition accuracy of the input tensor.*

In other words, *a priori* knowledge about noise distribution on the tensor (noise profiles depicted in Figure 4.1 and Section 4.3.1) and the inherently approximate nature of the tensor train decomposition process are both considered to obtain a decomposition strategy, which involves (a) the order of the modes and (b) the number of Gibbs samples allocated to each step of the decomposition process, that best suits the noise distribution of the given tensor.

4.2 Probabilistic Tensor train Decomposition (PTTD)

In this section, I present a *probabilistic tensor train decomposition* (`PTTD`) approach. In particular, `PTTD` replaces the SVD decomposition step in tensor train decomposition with

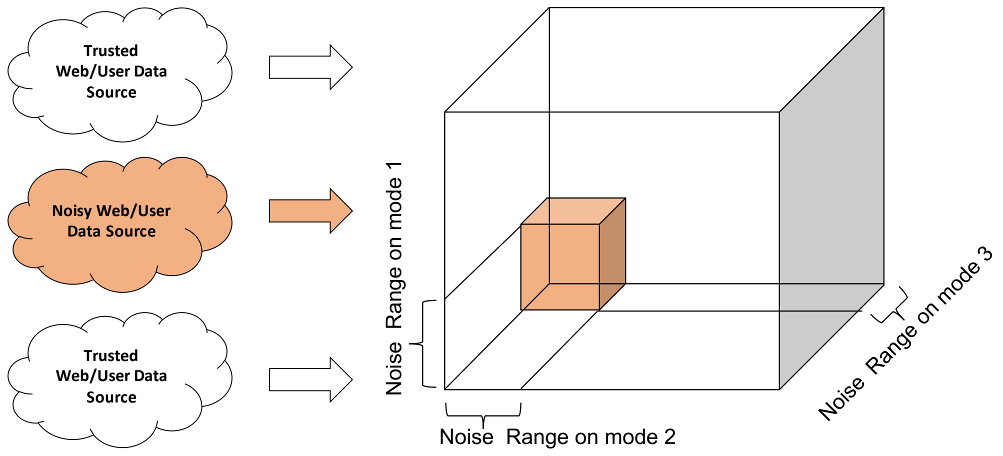

Trusted Web/User Data Source
Noisy Web/User Data Source
Trusted Web/User Data Source
Noise Range on mode 1
Noise Range on mode 2
Noise Range on mode 3

Figure 4.1: A Sample Noise Profile: The Figure Highlights (In Orange) the Sub-tensor Which Is Expected to Be More Noisy

probabilistic matrix factorization, in order to avoid over-fitting due to data sparsity and noise.

Note that, in and of itself, PTTD does not leverage **a priori** knowledge about noise distribution and internal decomposition interaction, but in Section 4.3, it provides the framework in which noise-profile based adaptation can be implemented.

4.2.1 Probabilistic Matrix Factorization

The singular value decomposition (SVD) that is commonly used in tensor train decomposition can be negatively affected from the noise and low data quality, which is especially a concern for web-based user data [3, 60, 61]. Especially for sparse data, avoiding over-fitting to the noisy data can be a significant challenge when using SVD. Recent research has shown that it may be possible to avoid such over-fitting by relying on probabilistic techniques [59], which introduce priors on the parameters, and can potentially average over various models and ease the difficulty of parameter tuning.

Probabilistic Matrix Factorization

Probabilistic Matrix Factorization (PMF) is a probabilistic linear model with Gaussian observation noise. Let us consider a matrix $\boldsymbol{M} \in \mathbb{R}^{n_1 \times n_2}$. The conditional distribution over the observed values $\boldsymbol{M} \in \mathbb{R}^{n_1 \times n_2}$ (the likelihood term) and the prior distributions over $U \in \mathbb{R}^{r \times n_2}$ and $V \in \mathbb{R}^{r \times n_1}$are given by

$$p(M|U, V, \alpha) = \prod_{i=1}^{n_1} \prod_{j=1}^{n_2} [\mathcal{N}(M_{ij}|U_i^T V_j, \alpha^{-1})] \tag{4.1}$$

$$p(U|\alpha_U) = \prod_{i=1}^{n_1} \mathcal{N}(U_i|0, \alpha_U^{-1}) \quad \text{and} \quad p(V|\alpha_V) = \prod_{j=1}^{n_2} \mathcal{N}(V_j|0, \alpha_V^{-1}),$$

where $\mathcal{N}(x|\mu, \alpha^{-1})$ denotes the Gaussian distribution with mean μ and precision α.

Bayesian Probabilistic Matrix Factorization

In the Bayesian formulation the prior distributions over U and V feature vectors are assumed to be Gaussian:

$$p(U|\mu_U, \Lambda_U) = \prod_{i=1}^{n_1} \mathcal{N}(U_i|\mu_U, \Lambda_U^{-1})$$

A similar formulation holds for V. Note that each row of factor matrices U and V follows a Gaussian distribution, as shown in Figure 4.2. In general, this Gaussian is related to the uncertainty in the corresponding element and, thus, provides an opportunity to discover the distribution of data noise across the tensor, as mentioned in Section 4.3. The Gaussian-Wishart priors on the U and V hyperparameters $\Theta_U = \{\mu_U, \Lambda_U\}$ and $\Theta_V = \{\mu_V, \Lambda_V\}$ are modeled as

$$p(\Theta_U|\Theta_0) = \mathcal{N}(\mu_U|\mu_0, (\beta\Lambda_U)^{-1})\mathcal{W}(\Lambda_U|W_0, v_0)$$

Again, A similar formulation hold for V.

$\mathcal{W}$ is the Wishart distribution with v_0 degrees of freedom and a D $\times$ D scale matrix, W_0:

$$\mathcal{W}(\Lambda|W_0, v_0) = \frac{1}{C}|\Lambda|^{(v_0-D-1)/2} exp(-\frac{1}{2}Tr(W_0^{-1}\Lambda)),$$

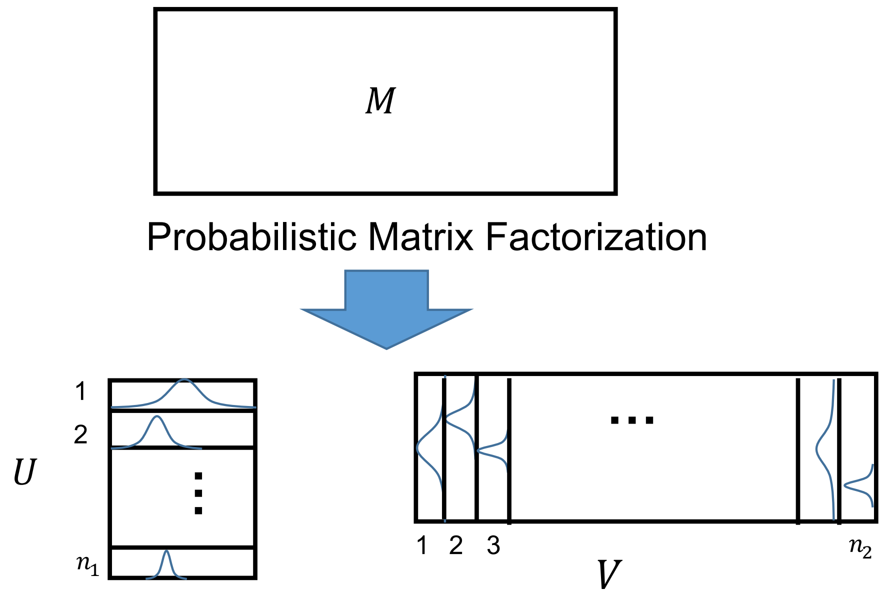


Figure 4.2: Illustration of PMF, Each Object of u and v Follows a Gaussian Distribution

where C is the normalizing constant. Here, $\Theta_0 = \mu_0, v_0, W_0, v_0 = D$ and W_0 are identity matrix for both U and V hyperparameters.

Given the above, $\widetilde{M_{ij}}$ can be predicted by marginalizing over model parameters and hyperparameters:

$$p(\widetilde{M_{ij}}|M,\Theta_0) = \iint p(\widetilde{M_{ij}}|U_i,V_j)p(U,V|R,\Theta_U,\Theta_V)$$
$$p(\Theta_U,\Theta_V|\Theta_0)d\{U,V\}d\{\Theta_U,\Theta_V\}.$$

Since exact evaluation of this predictive distribution is analytically intractable due to the complexity of the posterior I need to resort to approximate inference.

To provide deterministic approximation schemes for posteriors, variational methods [27] are one choice .The true posterior of factor matrix$p(U,V,\Theta_U,\Theta_V)$ parameters can be approximated by the distribution that factors. And each factor having a specific parametric form such as a Gaussian distribution. Equation 4.2.1 is leveraged to approximate the approximate posteriors. Variational methods have become the methodology of choice, since they typically scale well to large applications. However, they can produce inaccurate results because they tend to involve overly simple approximations to the posterior.

MCMC-based methods [45], for example, use Monte Carlo approximation to the predictive distribution of Equation 4.1, given by:

$$p(\widetilde{M}_{ij}|M, \Theta_0) \approx \frac{1}{K}\sum_{k=1}^{K} p(\widetilde{M}_{ij}|U_i^{(k)}, V_j^{(k)}) \tag{4.2}$$

More specifically, the factor matrix $\{U_i^{(k)}, V_j^{(k)}\}$ are sampled by running a Markov chain, which is stationary distribution over the model parameters and hyperparameters $\{U, V, \Theta_U, \Theta_V\}$. MCMC methods asymptotically approach the exact results.

Algorithm 4 PTTD

```
Input: d dimensional tensor $\boldsymbol{\mathcal{X}}$, Rank $R = \{r_1, ..., r_{d-1}\}$
Output: Decomposed factors $\boldsymbol{U}^{(1)}, \ldots, \boldsymbol{U}^{(d)}$ of
TT-approximation $\tilde{\boldsymbol{\mathcal{X}}}$

1. generate the appropriate sampling number for each mode ,$S = \{s_1, ...., s_{d_1}\}$ with intelligent sampling assignment strategy
2. Temporary Tensor: $M = \boldsymbol{\mathcal{X}}$
3. for k = 1 to d-1 do
    (a) $M := reshape(M, [r_{k-1}n_k, \frac{numel(M)}{r_{k-1}n_k}])$
    (b) apply the Probabilistic Matrix Factorization (PMF) on the matrix $M$ with pre-given rank $r_k$ and sampling number $s_k$ to get the $\boldsymbol{U}^{(k)} and \boldsymbol{V}^{(k)}$
    (c) New core: $\boldsymbol{U}^{(k)} = reshape(U, [r_{k-1}, n_k, r_k])$
    (d) $M := SV^{(k)T}$
4. $U_d = M$
5. Return tensor $\tilde{\boldsymbol{\mathcal{X}}}$ in TT-format with scores $\boldsymbol{U}^{(1)}, \ldots, \boldsymbol{U}^{(d)}$
```

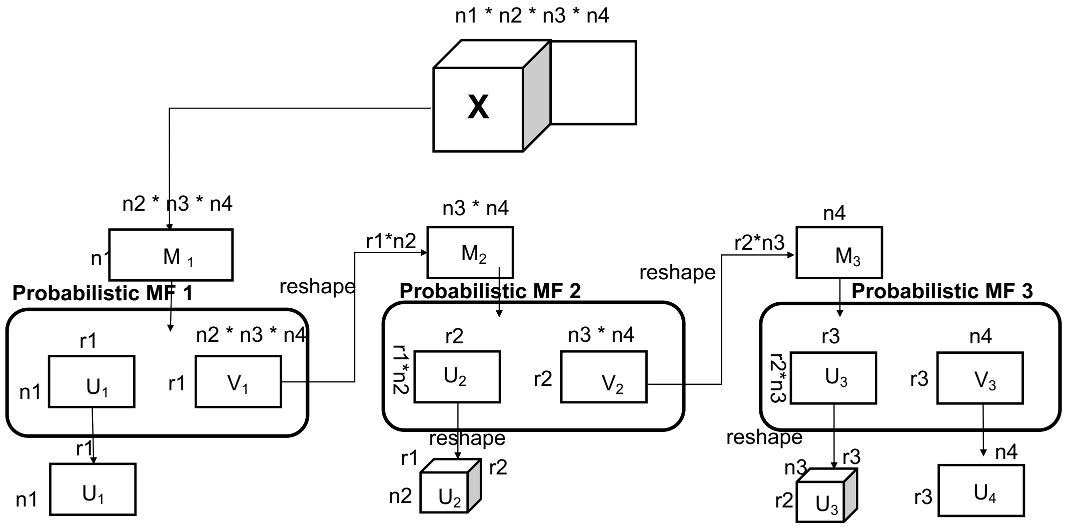


Figure 4.3: Probabilistic Tensor Train Decomposition (PTTD)

4.2.2 Overview of PTTD

Given the limitations of SVD on sparse and noisy tensors, the first step is to introduce a *probabilistic tensor train decomposition* scheme (PTTD), which extends tensor train decomposition framework with probabilistic matrix factorization [43].

In PTTD, the tensor $\boldsymbol{\mathcal{X}}$ is matricized as a matrix, M, and then I apply probabilistic matrix factorization on this matrix. The resulting factor matrix, U, is assigned as the first TT factor matrix. The matrix V is reshaped into the matrix M_{next} to be factorized in the next step. This probabilistic factorization and reshape processes are repeated until the decomposition is completed. The pseudo code of the algorithm is presented in Algorithm 4 and the process is visualized in Figure 4.3.

4.3 Noise Adaptive Probabilistic Tensor train Decomposition (NTTD)

One key advantage of the probabilistic decomposition framework presented above is that it can simultaneously uncover (Gaussian) noise while obtaining the decomposition[48]. Yet, it fails to account for the *potentially available (user-provided or automatically discovered)* *knowledge* about (a) the distribution of the *external* data noise across the tensor and (b) the noise generated *internally* due to the inherent imperfections in the decomposition process at the different steps of the tensor train network.

4.3.1 External and Internal Noise

External (Data) Noise

I define, (external data) *noise density* as the ratio of the cells that are subject to noise. Without loss of generality, I assume noise exists only on cells that have values (i.e., the observed values can be faulty, but there are no spurious observations) and, thus, I formalize noise density as the ratio of the non-null cells that are subject to noise. Note that noise may impact the observed values in the tensor in different ways: in *value-independent noise*, the correct data may be overwritten by a completely random new value, whereas in *value-correlated noise*, existing values may be perturbed (often with a Gaussian noise, defined by a standard deviation, σ). I refer to the amount of perturbation as the *noise intensity.*

In a tensor, noise can be distributed in many ways, depending on how data is collected. In *uniform noise*, there is no underlying pattern and noise is not clustered across any slice or region of the tensor. In general, however, noise is rarely uniformly distributed on a given tensor and may be clustered along rows, slices, or modes [37].

Internal (Decomposition) Noise

In Section 2.4, in the tensor train format, the network structure acts as a "train" or "chain" of tensors: the core tensors only interact with their neighboring cores as illustrated in Figure 2.3. The corresponding tensor train decomposition relies on sequential projections (formulated as sequential matrix factorizations) and

the decomposition accuracy of the intermediate matrix, M_k, depends on the ac-

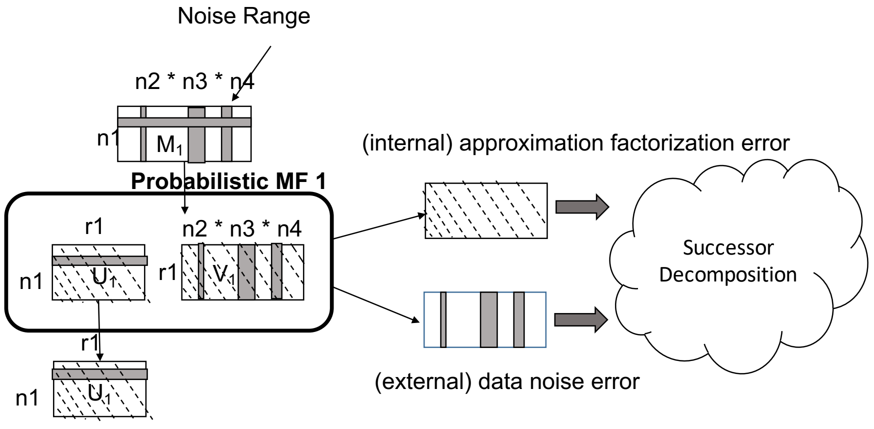


Figure 4.4: Two Types of Errors Propagate to Downstream Matricizations in Tensor Train Decomposition: (*Internal*) Approximate Factorization Error and (*External*) Data Noise Error

curacy of the previous matrix M_{k-1}'s (approximate) decomposition; similarly, the factorization error of M_k propagates to the following sequence of (intermediate) matrices, $M_{k+1}, ..., M_N$, of the chain.

This implies that a predecessor matrix which is poorly decomposed due to data noise or approximation error may negatively impact decomposition accuracies also for the rest of the successor matrices.

4.3.2 Noise Adaptation through Sample Assignment

Consequently, the inaccuracies resulting from each intermediate decomposition along the chain (whether due to data noise or factorization approximation error, Figure 4.4) need to be carefully considered during planning and resource allocation. The proposed *noise-profile adaptive tensor decomposition* (NTTD) algorithm adapts to (user provided or automatically discovered) a priori knowledge about noise by selecting a resource assignment strategy that best suits to the internal and external noise profiles. More specifically, NTTD assigns Gibbs samples to the decompositions of the various individual matricizations in a way that maximizes the overall decomposition accuracy of the whole tensor, In particular, it considers noise density along various modes and internal interaction between neighboring matrices to decide the appropriate number of samples to allocate for each matricization to achieve good overall accuracy with a given sample budget.

Table 4.1 provides a motivating example for the non-uniform sampling strategy. Uniform sampling strategy, which would distribute the available sampling budget, L_{total}, among the various steps in the decomposition chain gives the worst accuracy performance in term of RMSE. We, therefore, need a strategy to allocate the Gibbs samples across the various steps in the tensor train. In the rest of this section, I carefully study the two (internal and external) types of errors and how they propagate during the overall decomposition and how they impact the overall decomposition accuracy.

4.3.3 Gibbs Sampling and (Internal) Decomposition Error

As discussed in Section 4.2.2, the probabilistic tensor train decomposition process consists of several sequential probabilistic matrix decompositions. Consequently, any inaccuracies generated in any of the upstream decompositions will *propagate* to the downstream matrix decompositions along the "train" structure. In this section, I ignore the external data noise and focus on the impact of this internal noise generated due to decomposition inaccuracies. More specifically, I aim to investigate how to allocate Gibbs samples in a way that is sensitive to (a) the internal noise generated by the individual matrix factorizations, (b) the downstream (internal) noise propagation, and (c) their impacts to the accuracy of the overall tensor train decomposition.

As discussed in Section 4.2.1, Gibbs sampling is used for tackling the challenge of evaluating the predictive distribution of the posterior by approximating the expectation by an average of samples drawn from the posterior distribution through a Markov Chain Monte Carlo (MCMC) technique. Gibbs sampling cycles through the latent variables, sampling each one from its distribution, conditioned on the current values of all other variables. In

	Uniform Sampling	Non-Uniform ($L_{i\_err}(\boldsymbol{M})$)			
		$L_{min} = 5$	$L_{min} = 10$	$L_{min} = 15$	$L_{min} = 20$
RMSE	1.1	0.5783	0.4846	**0.4455**	0.6524

Table 4.1: Example of Tensor Train Decomposition with Different Sample Distribution Strategies: The Input Tensor Has Dimensions $143 \times 200 \times 12 \times 4$ (Density 5.68e-04), with Schema < User, Product, Category, Helpfullness>; In This Example, the Total Sampling Number Is Set to 90; Rmse (Matching Error) Is Presented to Illustrate the Decomposition Performance (Here l Stands for the Number of Gibbs Sample Allocated; Details Of the Parameters Are Described In Section 4.4)

Figure 4.4, for each intermediate matrix decomposition in `PTTD`, two factor matrices are generated: The U factor matrix is used to construct the core tensor corresponding to the current mode, whereas the V factor matrix is re-shaped as an input matrix for the successor decomposition step. Therefore,

- the accuracy of the U_k matrix has *direct* impact on the accuracy of one of the cores, whereas
- the accuracy of the V_k matrix *indirectly* influences accuracies of all downstream cores,

This observation, along with the observation that more samples can help provide better accuracy (*to certain degree*) in matrix factorization, can be used to improve the overall decomposition accuracy, to help allocate Gibbs samples to the different steps in tensor decomposition. More specifically, the number of samples for an intermediate matrix, M_k, should be allocated proportional to the size of the factor matrix, $size(U_k) + size(V_k)$, which reflects the number of unknowns to be discovered during the factorization of matrix M_k. In other words, the *internal decomposition error* sensitive sampling number, $L_{i\_err}(M_k)$, for matrix M_k can

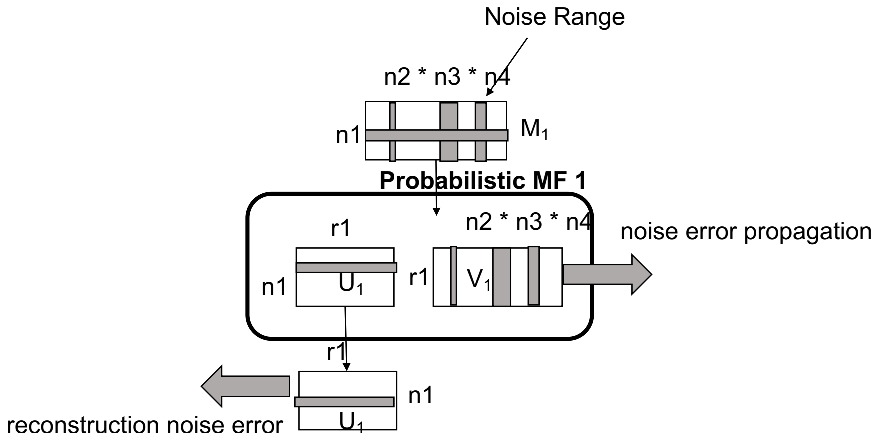


Figure 4.5: Illustration of Decomposition Noise Error Propagation and Reconstruction Noise Error for the First Decomposition Step

be computed as

$$L_{i\_err}(M_k) = L_{min}(M_k) + \lceil \gamma_{i\_err} \times (size(U_k) + size(V_k)) \rceil,$$

where γ_{i_err} is a scaling parameter such that the sum of all the sample counts is equal to the total number, $L_{i\_err(total)}$, of samples allocated for dealing with *internal decomposition errors* for the whole tensor decomposition:

$$L_{i\_err(total)} = \sum_{k=1}^{d-1} L_{i\_err}(M_k) \tag{4.3}$$

In Table 4.1, internal error informed sample assignment provides significant gains over uniform sample assignment. While the parameter L_{min} has some impact on the overall accuracy, its impact is minor relative to the impressive gains I obtain by considering the impact of the number of samples allocated on the individual matrices on the overall decomposition. I will further experimentally study this in Section 4.4.

4.3.4 Gibbs Sampling and (External) Noise

Equation 4.7, above, helps allocate samples across intermediate decomposition phases. However, it ignores one crucial piece of information that may be available: *distribution of*

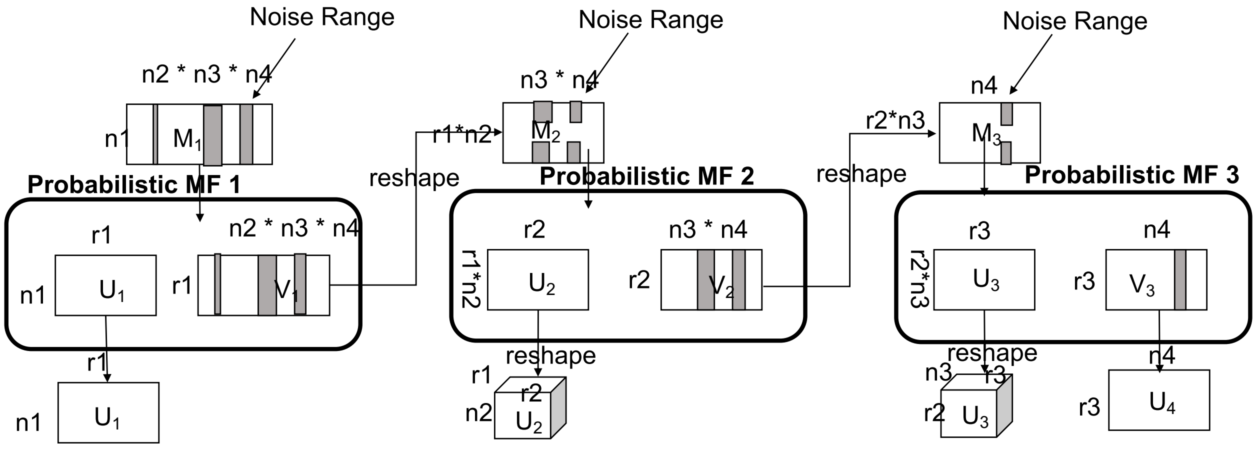


Figure 4.6: Illustration of Noise Error Propagation

the noise across the input tensor.

The basic probabilistic tensor train decomposition (Section 4.2) assumes the noise is uniformly distributed across the tensor. In the real world, however, noise is rarely *uniformly* distributed along the entire tensor. More often, I would expect that noise would be clustered across slices of the tensor (corresponding, for example, to unreliable information sources or difficult to obtain data). In many cases, even if I do not have precise knowledge about the cells that are subject to such noise or the amount of noise they contain, I may have a rough idea about the distribution of noise across the different modes [37]. As I experimentally show in Section 4.4, there is a direct relationship between the noise distribution across the tensor and the number of Gibbs samples it requires for accurate decomposition. Consequently, given a tensor with non-uniform noise distribution across different modes, uniform assignment of the number samples, $L_{n\_err}(M_k) = \frac{L_{n\_err(total)}}{d-1}$ (where $L_{n\_err(total)}$ is the total number of Gibbs samples for tackling the impact of noise) becomes ineffective. Therefore, in this section, I aim to answer the question

> *can we leverage rough information that may be available about noise distribution in improving the accuracy of the overall tensor train decomposition?*

Noise taints accuracy through two distinct mechanisms: (a) impact of noise during decomposition and, for applications (such as recommendation and prediction) that involve the recovery of missing entries in the tensor, (b) impact of noise during reconstruction .In Figure 4.5, the noise in the input matrix partitions itself into the resulting factor matrices U and V. The factor matrix V_i is reshaped as input matrix for the following tensor train decomposition steps, therefore is involved in the propagation of the noise to downstream steps during the tensor train decomposition process (Figure 4.6). The matrix, U_i, however, is separated into a factor matrix (for U_1) or more generally to a core tensor for factor $i > 1$, and thus impacts accuracy during reconstruction. I discuss these next.

Impact of Noise During Reconstruction

The noise reconstruction error taints the overall accuracy in the reconstruction process due to the matrix tensor multiplication operations involved in the recomposition of the (approximate) tensor. For example, if the ith object of U_k is polluted by the noise, after the reconstruction process, the complete slice $\tilde{\boldsymbol{\mathcal{X}}}_{*,\ldots,*,k(i),*,\ldots,*}$ will be tainted by the noise pollution from column $U_{k(i)}$ due to the matrix and tensor multiplication. Consequently, to account for the noise reconstruction error, the number of Gibbs samples should be proportional to the mode noise density, nd_k.

Impact of Noise During Decomposition

A naive approach to allocate the number of samples for a noisy matrix, M_k, is to allocate it proportional to its noise density, nd_k. However, since the probabilistic tensor train decomposition process follows a "train" structure, errors propagate downstream as shown in Figure 4.6. Consequently, allocating sampling number proportional to the noise density maybe not the best strategy.

As mentioned earlier, the Gibbs sampling algorithm cycles through the latent variables,

Algorithm 5 Gibbs sampling for Bayesian PMF

```
1. Initialize model parameters $\{U^1, V^1\}$
2. For t = 1,...,T
    (a) Sample the hyperparameters
        $\Theta_U^t \approx p(\Theta_U | U^t, \Theta_0)$
        $\Theta_V^t \approx p(\Theta_V | V^t, \Theta_0)$
    (b) For each decomposition= 1,...,N sample $U$ features in parallel
        $U_i^{t+1} \approx p(U_i | R, V^t, \Theta_U^t)$
    (c) For each j = 1,...,M sample $V$ features in parallel
        $V_j^{t+1} \approx p(U_j | R, V^t, \Theta_U^t)$
```

sampling each one from its distribution conditional on the current values of all other variables. Due to the use of conjugate priors for the parameters and hyperparameters in the Bayesian PMF model, the conditional distributions derived from the posterior distribution are easy to sample from. In particular, the conditional distribution over the feature vector U_i, conditioned on the other features V_i, observed matrix cell value M_i, and the values of the hyperparameters are Gaussian:

$$\begin{aligned} p(U_i | M, V, \Theta_U, \alpha) &= \mathcal{N}(U_i | \mu_i, \Lambda_i^{-1}) \\ &\approx \prod_{i=1}^{n_1} [\mathcal{N}(\widetilde{M}_{i,j} | U_i^T, V_i, \alpha^{(-1)})]^{n_{i,j}} p(U_i | \mu_U, \Lambda_U^{-1}) \end{aligned} \tag{4.4}$$

Here,

$$\Lambda_i = \Lambda_U + \alpha \sum_{j=1}^{M} [V_j * V_j^T]^{n_{ij}}$$

and

$$\mu_i = [\Lambda_i]^{-1} (\alpha \sum_{j=1}^{M} [V_j * \widetilde{M}_{i,j}]^{n_{ij}} + \mu_U \Lambda_U).$$

Note that the conditional distribution over the latent feature matrix U factorizes into the product of conditional distributions over the individual feature vector:

$$p(U|M, V, \Theta_U) = \prod_{i=1}^{n_1} p(U_i|M, V, \Theta_U) \tag{4.5}$$

The conditional distribution over the U feature matrix hyperparameters conditioned on the feature matrix U is given by the Gaussian-Wishart distribution

$$p(\mu_U, \Theta_U) =$$

$$\mathcal{N}(\mu_U|\mu_0^*, (\beta^*\Theta_U)^{-1})\mathcal{W}(\Theta_U|W_0^*, V_0^*)$$

where

$$\mu_0^* = \frac{\beta_0\mu_0 + n_1\overline{U}}{\beta_0 + n_1}, \beta_0^* = \beta_0 + n_1; v_0^* = v0 + n_1;$$

$$[W_0^*]^{-1} = W_0^{-1} + n_1\overline{S} + \frac{\beta_0 n_1}{\beta_0 + n_1}(\mu_0 - \overline{U})(\mu_0 - \overline{U})^T$$

$$\overline{U} = \frac{1}{n_1}\sum_{i=1}^{n_1} U_i, \overline{S} = \frac{1}{n_1}\sum_{i=1}^{n_1} U_i U_i^T$$

The conditional distributions over the V feature vectors and the V mode hyperparameters have exactly the same form. Consequently, the Gibbs sampling algorithm takes the form in Algorithm 5.

Equations 4.4 and 4.5, along with figure 4.6, indicate how errors propagate downstream. In particular, in Figure 4.6, red columns of the first matricization, M_1, show the columns that are noise polluted. During the decomposition, the corresponding columns of resulting factor matrix V_1 (highlighted also in red) are also tainted with stronger noise than other columns of V_1. This tainting process flows downstream (*subject to matrix re-shape operations*) as shown in the figure 4.6. Consequently, for decomposition phase k, the number of samples should be proportional to

$$\sum_{j=k}^{d-1} \prod_{i=j+1}^{d} nd_i, \tag{4.6}$$

where, $\prod_{i=j+1}^{d} nd_i$ is the noise density of matricization of V_k on mode k . The $\sum_{j=k}^{d-1}$ operation, above, takes into account the accumulation process of the noise on all downstream decomposition steps.

Combining Decomposition and Reconstruction Impacts of Noise

Assuming that the decomposed tensor will be utilized for an application (such as recommendation) which necessitates reconstruction of the approximate tensor, I need to consider reconstruction and decomposition errors together when assigning the number of Gibbs samples. In other words, for an intermediate matrix, M_k, the number of samples must be allocated proportional to the sum of reconstruction and decomposition errors, i.e, $nd_k + \sum_{j=k}^{d-1} \prod_{i=j+1}^{d} nd_i$. This leads to the following formula for the number $L_{n\_err}(M_k)$ of samples:

$$L_{min}(M_k) + \lceil \gamma_{n\_err} \times (\sum_{j=k}^{d-1} \prod_{i=j+1}^{d} nd_i + nd_k) \rceil \times L_{n\_err(total)},$$

where γ_{n_err} is a scaling parameter such that the sum of all the sample counts is equal to the total number, $L_{n\_err(total)}$, of samples allocated for dealing with *noise errors*:

$$L_{n\_err(total)} = \sum_{k=1}^{d-1} L_{n\_err}(M_k). \tag{4.7}$$

4.3.5 Overall Sample Assignment

As seen above, while considering the error propagation, both internal decomposition error (Section 4.3.3) and external noise error (Section 4.3.4) need to be accounted for. Therefore, the combined sample assignment equation, for matricization, M_k, in the tensor train decomposition process, can be written as

$$\begin{aligned} L(M_k) = & \lceil \gamma_{n\_err} \times (\sum_{j=k}^{d-1} \prod_{i=j+1}^{d} nd_i + nd_k) \rceil \times L_{n\_err(total)} \\ & + \lceil \gamma_{i\_err} \times (size(U_k) + size(V_k)) \rceil \times L_{i\_err(total)} \\ & + L_{min}(M_k) \end{aligned} \tag{4.8}$$

Parameters	*Alternative values*
Dataset	Ciao; BxCrossing; MovieLens
Noise Density	**10**%; 20%; 30%;
Noise Intensity (σ)	**1**,3,5
Total Samples (L_{total})	**90**; 135; 180;
Min. Samples ($L_{min}(M_k)$)	$L_{total}/(3 \times (d-1)) = L_{total}/9$

Table 4.2: Parameters – Default Values, Used Unless Otherwise Specified, Are Highlighted

where $L_{min}(M_k)$ is the minimum number of samples a (non-noisy) tensor of the given size would need for accurate decomposition and γ_{n_err} and γ_{i_err} are two scaling parameters, selected such that the total number of samples is equal to the number, L_{total}, of samples allocated for the whole tensor; i.e.,

$$L_{total} = \sum_{k=1}^{d-1} L(M_k).$$

The two parameters, γ_{n_err} and γ_{i_err}, also control the relative impacts of the internal and external noise on sample allocation. In the experiments below, they are set such that the number of samples allocated to handle internal and external noise are the same.

4.4 Experimental Evaluation

In this section, I report experiments that aim to assess the effectiveness of the proposed noise adaptive tensor train decomposition approach.

4.4.1 Experiment Setup

Key parameters and their values are reported in Table 4.2

Data Sets

In these experiments, I used three user-centered datasets: Ciao [57], MovieLens [21] and BxCrossing [62]. Ciao dataset is represented in the form of $143 \times 200 \times 12 \times 4$ (density 5.68E-04), which has the schema <user, product, category, helpfullness>. BxCrossing dataset is represented in the form of $2599 \times 34 \times 16 \times 76$ (density 2.48E-0.5), which has the schema <user, book, published year, user age>. The MovieLens dataset is represented in the form of $247 \times 112 \times 48 \times 21$ (density 8.86E-06), which has the schema <user, movie, age, location>. In Ciao and MovieLens data sets, the tensor cells contain rating values between 1 and 5 or (if the rating does not exist) a special "null" symbol. And for the BxCrossing dataset, the tensor cells contain rating values between 1 and 10.

Noise

In order to observe the different degrees of noise, I selected a random portion of the non-null cells (based on the noise density) and randomly perturbed the value (based on the noise intensity). Note that this is a worst-case scenario for `NTTD`, where the noise is distributed *uniformly on the tensor*; but the experiments show that even in this case, `NTTD` can take into account the noise density difference across the various data modes, implied by the difference in corresponding data densities. Therefore, in the experiments, the noise density for different modes is approximated by the corresponding data density.

Alternative Strategies

I compare the proposed approach against other sampling strategies: uniform, internal-noise only, and external-noise only sample assignment:

- In *uniform* strategy (`UNI`), L_{total} is uniformly divided among the three matricizations involved in the tensor train decomposition and default `PTTD` is used for decomposition.

- In *internal-noise only* strategy (`I_ERR`), γ_{n_err} is set to zero in Equation 4.8, focusing the assignment to only internal decomposition error.
- In *external-noise only* strategy (`N_ERR`), γ_{i_err} is set to zero in Equation 4.8, focusing sample assignment to the impact of noise and its propagation during the decomposition and reconstruction.

Decomposition Order and Rank

For tensor train decomposition, as default decomposition orders, I used the following:

- Ciao: $user \rightarrow product \rightarrow category \rightarrow helpfulness$,
- BxCrossing: $user \rightarrow book \rightarrow year \rightarrow age$.
- MovieLens: $user \rightarrow movie \rightarrow age \rightarrow location$.

To pick the decomposition ranks for the different modes, I used the recommended ranks as computed by the TT-SVD strategy [48].

Evaluation Criterion

I use the root mean squares error (RMSE) inaccuracy measure to assess the decomposition effectiveness. Each experiment was run 10 times with different random noise distributions and averages are reported.

Hardware and Software

I ran experiments on an eight-core CPU Nehalem Node with 16.00GB RAM. All codes were implemented in Matlab and run using Matlab R2016b. For tensor decomposition, I used MATLAB Tensor Toolbox Version 2.6

4.4.2 Discussion of the Results

Overview

In Figure 4.7, I compare the performance of `NTTD` with noise-adaptive sample assignments against other strategies for different noise densities.

In this figure, the proposed `NTTD` strategy is able to allocate Gibbs samples effectively to significantly reduce RMSE relative to `PTTD` with uniform sample assignment for all data sets and noise densities.

Moreover, I also see that internal- and external-only strategies that ignore part of the error, can actually hurt the overall accuracy and perform worse than the uniform strategy.

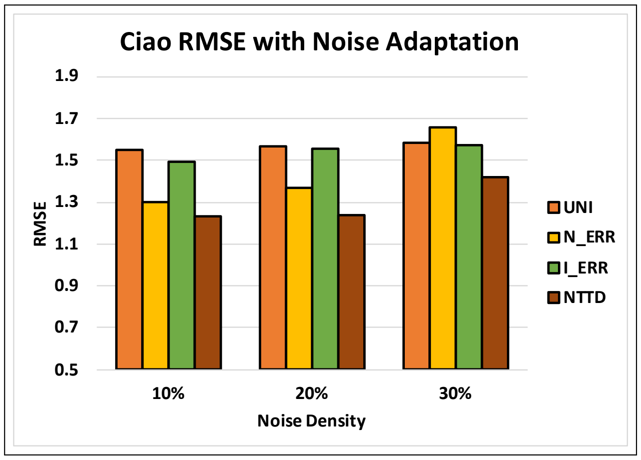
These, together, show that the proposed noise-adaptive strategy is highly effective in leveraging rough knowledge about external noise distributions and internal decomposition errors to better allocate the Gibbs samples across the tensor.

Impact of the total number of samples.

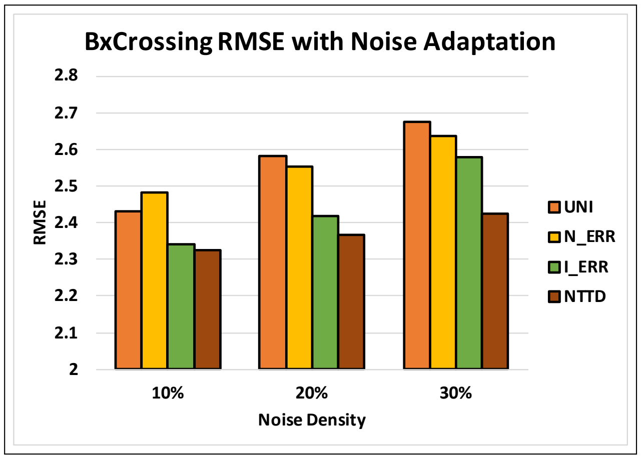
A key parameter of the `NTTD` algorithm is the number of total Gibbs samples. In Figure 4.8, as expected, increasing the number of Gibbs samples helps reduce the overall decomposition error. It is important to note that, among the four strategies, `NTTD` strategy is the one that provides most consistent and quickest drop in error. The figure shows the result for the MovieLens data; the results are similar also for the other data sets.

Impact of the noise intensity.

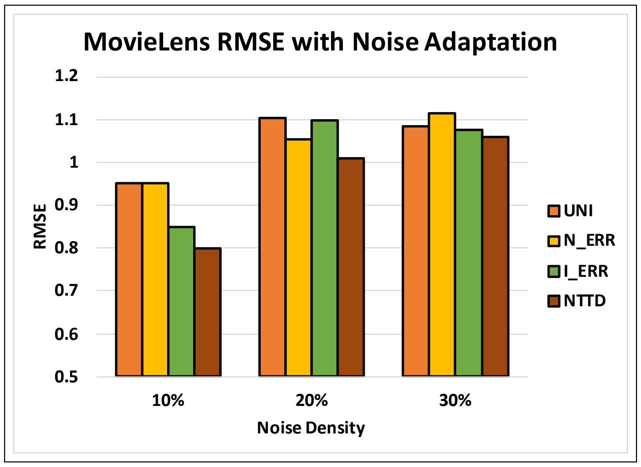
In Figure 4.9, I consider the MovieLens data set with different noise intensities. As expected, in this data set, increased noise corresponds to increased RMSE. However, `NTTD` provides the best results for all noise intensities considered. `NTTD` is also the best strategy for the other two data sets.



(a) RMSE for Ciao Dataset



(b)RMSE for BxCrossing Dataset



(c) RMSE for MovieLens Dataset

Figure 4.7: RMSE with Different Data Sets and Noise Densities ($L_{total} = 90$))

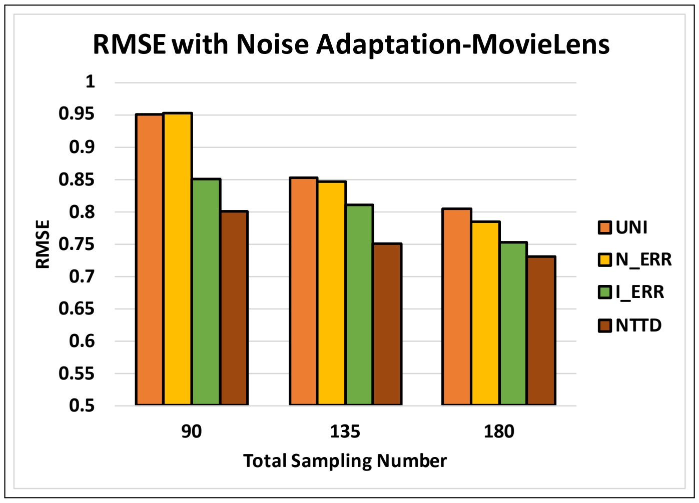


Figure 4.8: RMSE with Different Num. Of Samples; I.E. l_{Total} Is 90, 135, or 180 (Noise Density 10%, Noise Intensity 1)

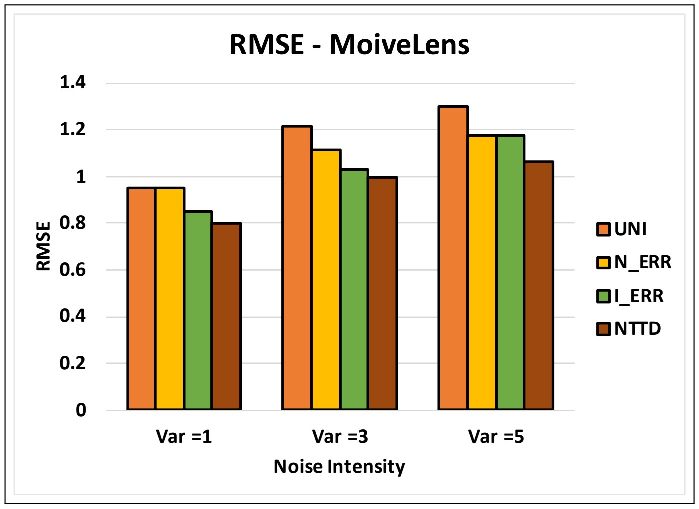


Figure 4.9: RMSE with Different Noise Intensities; I.E., σ Is 1, 3, or 5 (Noise Density 10%)

	RMSE			
	`UNI`	`I_ERR`	`N_ERR`	`NTTD`
Ciao	1.551	1.30	1.494	**1.233**
Ciao (alterate)	1.507	**1.299**	1.320	1.329
BxCrossing	2.4326	2.4838	2.3414	**2.3251**
BxCrossing (alternate)	2.3689	2.3445	2.311	**2.2835**
MovieLens	0.9507	0.9515	0.8497	**0.8007**
MovieLens (alternate)	0.9272	0.8952	0.8546	**0.8032**

Table 4.3: RMSE with Different Decomposition Orders (Default Parameters)

Impact of decomposition order

In the above experiments reported, I considered the default tensor train "chain" order. In Table 4.3, I study the impact of different decomposition orders. In particular, I swap the second and third steps in the decomposition to obtain the following alternative chains:

- Ciao: $user \rightarrow category \rightarrow product \rightarrow helpfulness$,
- BxCrossing: $user \rightarrow year \rightarrow book \rightarrow age.$
- MovieLens: $user \rightarrow age \rightarrow movie \rightarrow location.$

As seen in the table, the overall decomposition accuracy depends on the order [1]. Nevertheless, in most cases, `NTTD` is still providing the best sample assignment strategy for all orders considered. The only exception is the Ciao dataset, where for the alternative chain order, the uniform strategy provides the best overall accuracy, with the internal-error based strategy

[1]The decomposition order in tensor train decomposition is usually assumed to be an input provided by the user. The study of the impact of this order on the overall accuracy and optimization of orders is a future work.

being a close second. Note that the Ciao dataset is also the only data set where the alternative order actually provides a significantly lower error than the default. This indicates that, in the alternative order, the decomposition process is simpler and mostly error free and thus NTTD strategy cannot bring advantages over the naive sample assignment approach. However, NTTD is especially advantageous in the more likely cases where the decomposition process is subject large errors.

Chapter 5

TENSOR DECOMPOSITION FOR BILLION-SCALE DENSE TENSOR

5.1 Introduction

A major challenge with tensor decomposition is its computational and space complexities – especially for *dense* data sets[1]. While the process is relatively faster for *sparse* tensors, decomposition is still a major bottleneck in many applications [16]: Tensor decomposition process results in dense (and hence large) intermediary data, even when the input tensor is sparse (and hence small). This is known as the *intermediate memory blow-up problem* [28] and renders purely in-memory implementations of tensor-decomposition impractical, for both CP and Tucker decompositions [50]. As the relevant data sets get large, existing in-memory schemes for tensor decomposition become increasingly ineffective and block-based solutions where some (possibly intermediate) data may be materialized on disks (instead of main memory) or other servers contributing to the decomposition process are necessitated. Several implementations of tensor decomposition operations on disk-resident data sets have been proposed, such as GirdPARAFAC [51], TensorDB [31, 32], HaTen2 [26]. In all these systems, I/O costs is an inevitable problem as they need I/O to fetch data either from disk or from the network. Consequently, reducing these I/O and communication costs, especially for dense tensors common in science and engineering, is a critical challenge.

In this dissertation, I propose 2PCP, a two-phase CP tensor decomposition mechanism. In Figure 5.1, two- phase block-based tensor decomposition can help reduce the memory-blow-

[1]Such dense tensors are common in science and engineering: ensemble simulations, for example, are created by sampling the domains of the relevant input parameters, and recording simulation results for each configuration.

up problem as the first phase requires decomposition of much smaller tensors. However, the number of the (so called factor) matrices that are produced in the first phase and the intermediary data generated while these are stitched together through an iterative process in the second phase may still be quite large. Consequently, the intermediary data may still take too much space to be fully memory-resident and may need to be brought to the memory on on-demand basis. Consequently, the 2PCP system I present in this dissertation complements the basic two-phase CP tensor decomposition approach with novel data re-use promoting block scheduling and buffer management mechanisms to address this difficulty:

5.2 Overview of `2PCP`

The outline of the proposed two-phase, block-based tensor decomposition algorithm, `2PCP` is presented in Algorithm 6 and visualized in Figures 5.1 and 5.2.

Example 5.2.1 *In Figure 3.2, the given tensor $\boldsymbol{\mathcal{X}}$ is partitioned into two sub-tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$:*

- **Phase 1:** *In phase 1, each sub-tensor is decomposed with a standard PARAFAC algorithm. In the example, sub-factors $\boldsymbol{U}^{(1)}_{(1)}$, $\boldsymbol{U}^{(2)}_{(1)}$, $\boldsymbol{U}^{(3)}_{(1)}$ of sub-tensor $\boldsymbol{\mathcal{X}}_1$ and sub-factors $\boldsymbol{U}^{(1)}_{(2)}$, $\boldsymbol{U}^{(2)}_{(2)}$, $\boldsymbol{U}^{(3)}_{(2)}$ of sub-tensor $\boldsymbol{\mathcal{X}}_2$ are generated in the first stage.*
- **Phase 2:** *In the second phase, the sub-factors $\boldsymbol{U}^{(i)}_{\vec{k}}$ of each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ are used for iteratively refining the sub-factors $\boldsymbol{A}^{(i)}_{(k_i)}$ of the input tensor $\boldsymbol{\mathcal{X}}$.*

5.2.1 Key Observations

The pseudo code presented in Algorithm 6 supports the following key observations:

• **Observation #1 (Independent/Parallel Sub-tensor Decomposition in Phase 1):** Each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ can be decomposed (in parallel) independently from the others. This ensures that the proposed system can handle very large tensors as long as the input tensor

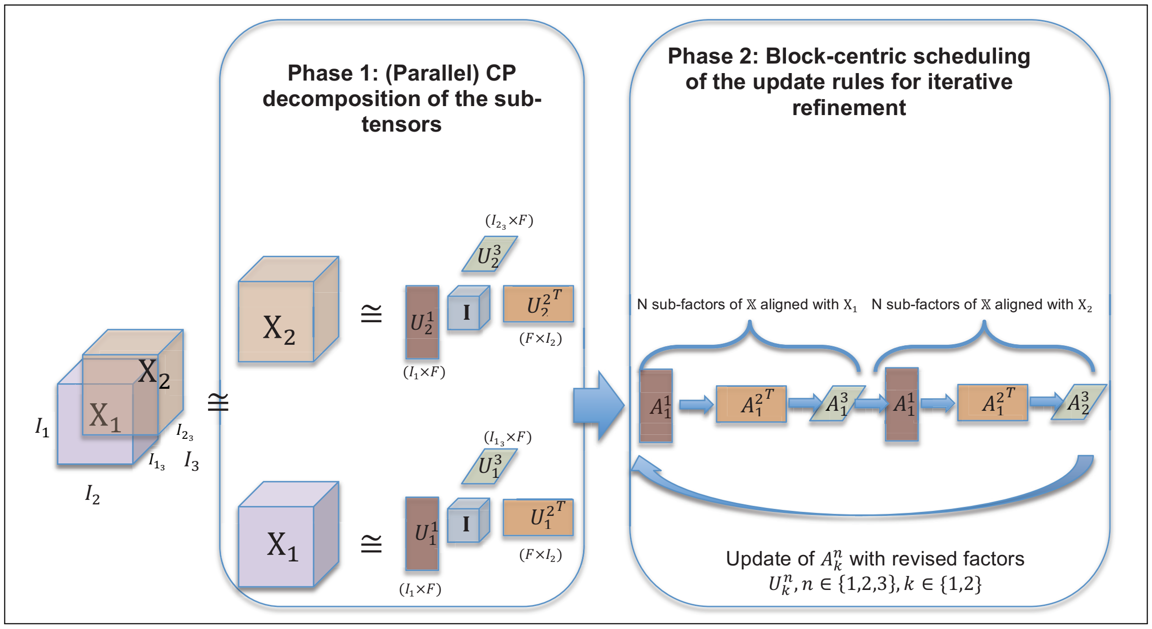

Phase 1: (Parallel) CP decomposition of the sub-tensors
Phase 2: Block-centric scheduling of the update rules for iterative refinement
N sub-factors of X aligned with X1
N sub-factors of X aligned with X2
Update of A_k^n with revised factors $U_k^n, n \in \{1,2,3\}, k \in \{1,2\}$
X_1
X_2
I_1
I_2
I_3
I_{1_3}
I_{2_3}
$(I_1 \times F)$
$(F \times I_2)$
$(I_{1_3} \times F)$
$(I_{2_3} \times F)$
U_1^1
$U_1^{2^T}$
U_1^3
U_2^1
$U_2^{2^T}$
U_2^3
I
A_1^1
$A_1^{2^T}$
A_1^3
A_2^3

Figure 5.1: Illustration of the Two-phase, Block-based Tensor Decomposition: The Input Tensor Is Partitioned into Smaller Blocks, Each Block Is Decomposed (Potentially in Parallel), and the Partial Decompositions Are Stitched Together Through an Iterative Improvement Process

is partitioned into several blocks in such a way that each block can be decomposed with the available memory. Moreover, this phase is easy to parallelize using, for example, the popular distributed computing framework, MapReduce, using the following `map` and `reduce` operators (assuming a three-mode input tensor):

- `map`: $\langle \boldsymbol{b}, i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$ on $\boldsymbol{b}$. Here, $\boldsymbol{b}$ is the sub-tensor id, i, j, k together give the coordinate of sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{\boldsymbol{k}}}$. Tuples with the same b are shuffled to the same reducer in the form of $\langle key : \boldsymbol{b}, values : i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$.

- `reduce` $\langle key : \boldsymbol{b}, values : i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$: The reducer processing the key $\boldsymbol{b}$ receives the non-zero elements of sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{\boldsymbol{k}}}$. It recomposes the sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{\boldsymbol{k}}}$. Then sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{\boldsymbol{k}}}$ is decomposed into sub-factor $\boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}$, where $\boldsymbol{n}$ is the mode id, by using PARAFAC. Finally, reducer emits each sub-factor $\boldsymbol{U_b^n}$ as an independent file, with content $\langle$ *key* : $\boldsymbol{U_b^n}$, *value* : $i, j, \boldsymbol{U_b^n}(i,j)\rangle$. Here, i, j are the coordinates of sub-factor $\boldsymbol{U_b^n}$.

.

• **Observation** #2 **(In-place Iterative Refinement in Phase 2):** As shown in Algorithm 6, once Phase 1 is completed, in the second phase, each $\boldsymbol{A}^{(i)}_{(k_i)}$ can be maintained by computing and revising $\boldsymbol{T}^{(i)}_{(k_i)}$ and $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ incrementally. Note that this incremental update process presented in Algorithm 6 is logically equivalent to the one presented in [51], but includes a significant structural difference: in [51], P and Q are updated using a separate loop for each mode to optimize for parallelism, whereas Algorithm 6 updates P and Q in-place to significantly reduce the amount of disk accesses.

The total space requirement during this iterative refinement process is governed by the sizes of the F-rank partial factors, $\boldsymbol{A}^{(i)}_{(k_i)}$, and the corresponding, $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$, for each mode i of the given tensor $\boldsymbol{\mathcal{X}}$; i.e., memory requirement $mem_{total}(\boldsymbol{\mathcal{X}})$ can be computed as

$$\sum_{i=1}^{N} K_i \times \left(\underbrace{\left(\frac{I_i}{K_i} \times F\right)}_{\boldsymbol{A}^{(i)}_{(k_i)}} + \underbrace{\left(\left(\prod_{j\neq i} K_j\right) \times \frac{I_i}{K_i} \times F\right)}_{total\ for\ \boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}} \right)$$

or equivalently as

$$mem_{total}(\boldsymbol{\mathcal{X}}) \sim \sum_{i=1}^{N} \left(\left(1 + \prod_{j\neq i} K_j\right) \times I_i \times F \right).$$

This implies that the total memory requirement increases quickly with the number of partitions considered. Since, when large tensors are considered, the number of partitions themselves may need to be large (to ensure that each resulting block can be decomposed using the available memory), $mem_{total}(\boldsymbol{\mathcal{X}})$ can go beyond the available memory.

• **Observation** #3 **(On-demand Per Mode-Partition (MP) Data Access in Phase 2):** Fortunately, during this iterative refinement process I do not need all this data in the

Algorithm 6 The Outline of the `2PCP` Block-based Iterative Improvement Process

Input: original tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, and decomposition rank, F

Output: CP tensor decomposition $\mathring{\boldsymbol{\mathcal{X}}}$

```
1. Phase 1: for all $\vec{k} \in \mathcal{K}$
    • decompose $\boldsymbol{\mathcal{X}}_{\vec{k}}$ into $\boldsymbol{U}^{(1)}_{\vec{k}}, \boldsymbol{U}^{(2)}_{\vec{k}}, \ldots, \boldsymbol{U}^{(N)}_{\vec{k}}$
2. Phase 2: repeat
    (a) for each mode $i = 1$ to $N$
        i. for each modal partition, $k_i = 1$ to $K_i$,
            A. update $\boldsymbol{A}^{(i)}_{(k_i)}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$, for each block $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$; more specifically,
                • compute $\boldsymbol{T}^{(i)}_{(k_i)}$, which involves the use of $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$)
                • revise $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
                • compute $\boldsymbol{S}^{(i)}_{(k_i)}$ using the above
                • update $\boldsymbol{A}^{(i)}_{(k_i)}$ using the above
                • for each $\vec{k} = [*, *, \ldots, k_i, \ldots, *, *]$
                    – update $\boldsymbol{P}_{\vec{k}}$ and $\boldsymbol{Q}_{\vec{k}}$ using
                    – $\boldsymbol{U}^{(i)}_{\vec{k}}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
   until stopping condition
3. Return $\mathring{\boldsymbol{\mathcal{X}}}$
```

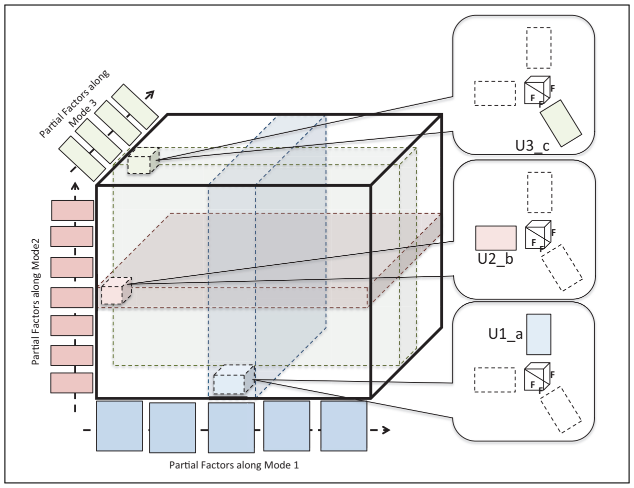

Partial Factors along Mode 3
Partial Factors along Mode2
Partial Factors along Mode 1
U3_c
U2_b
U1_a
F
F
F
F
F
F

Figure 5.2: The Outline of the Mode-centric Iterative Improvement Algorithm Proposed In [51]: The k^{th} Partial Factor along Mode i Is Updated Using the Mode i Partial Factors Of the Blocks Aligned with the k^{th} Mode Partition along Mode i

memory simultaneously. Incrementally maintaining $\boldsymbol{T}^{(i)}_{(k_i)}$ and $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ require bringing <u>only</u> $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ (i.e. the mode-i factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$) and (the old value of) $\boldsymbol{A}^{(i)}_{(k_i)}$ to the main memory. In fact, Phase 2 can be executed by considering each mode-partition individually and bringing to the memory only the relevant partial factors. In other words, for maintaining each $\boldsymbol{A}^{(i)}_{(k_i)}$, I need

$$mem_{MP}(\boldsymbol{\mathcal{X}}, \boldsymbol{A}^{(i)}_{(k_i)}) \sim \left(1 + \prod_{j \neq i} K_j\right) \times \frac{I_i}{K_i} \times F$$

units of memory.

• **Observation #4 (Challenge - Naive Execution Increases I/O):** While the above observations show that using the proposed two phase Algorithm 6, I am able to significantly reduce the amount of memory needed to decompose large tensors. A naive implementation of this strategy, however, may require significant amount of I/O: After each mode partition is processed, to open space for the data needed to process the next mode partition, (a) all updated data need to be written back to the disk and (b) data relevant to the next mode partition need to be read from the disk into the memory, resulting in $\sum_{i=1}^{N} K_i$ data swap operations for a single iteration of the algorithm. A more reasonable alternative would be to use the memory as a cache for mode-partition data and only swap data in and out of the buffer if the cache is full.

5.2.2 Problem Statement: Re-Use Promoting Data Access and Buffer Management During Iterative Refinement Phase

In the second phase of `2PCP`, the overall efficiency of the process depends highly on the effectiveness of the buffer utilization. Therefore, the key problem that needs to be addressed to improve the efficiency of `2PCP` is to improve the utilization of the buffer through *re-use promoting data access and buffer management* during the iterative improvement process.

In the rest of the section, I introduce how `2PCP` addresses this problem: I first (a) present an alternative fine-grained block-centric iterative refinement scheme, next (b) I consider alternative block-scheduling techniques leveraging this block-centric update process, and then (c) I investigate corresponding buffer replacement strategies to help improve the buffer utilization and reduce I/O costs.

5.3 Block-Centric Scheduling of Iterative Improvement Process

In the previous section (Algorithm 6 and visualized in Figure 5.2), the conventional way to perform the iterative refinement process of the block-based CP involves considering each mode, i, separately. In this *mode-centric* scheme, for the k_i^{th} partition of mode i, I then maintain $\boldsymbol{A}_{(k_i)}^{(i)}$ using, for all $1 \leq j \leq N$, the current estimates for $\boldsymbol{A}_{(k_j)}^{(j)}$ and the decompositions in $\boldsymbol{\mathfrak{U}}^{(j)}$; i.e., the F-rank sub-factors of the sub-tensors in $\boldsymbol{\mathcal{X}}$ along different modes.

As ny experiments validated, however, this conventional scheme may result in a significant amount of I/O (for swapping data in and out of memory) if the available memory is not sufficient to buffer all the data. Intuitively, this is because the order in which the update

Algorithm 7 The outline of the fine-grained decomposition algorithm, used by `2PCP`, which schedules update rules in a block-centric manner

```
Input: original tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, and decomposition rank, $F$
Output: CP tensor decomposition $\mathring{\boldsymbol{\mathcal{X}}}$

1. for all $\vec{k} \in \mathcal{K}$
     • decompose $\boldsymbol{\mathcal{X}}_{\vec{k}}$ into $\boldsymbol{U}^{(1)}_{\vec{k}}$, $\boldsymbol{U}^{(2)}_{\vec{k}}$, ..., $\boldsymbol{U}^{(N)}_{\vec{k}}$
2. repeat for each $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$
     (a) for each mode $i = 1$ to $N$
           i. update $\boldsymbol{A}^{(i)}_{(k_i)}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$, for each block $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$; more specifically,
                • compute $\boldsymbol{T}^{(i)}_{(k_i)}$, which involves the use of $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$)
                • revise $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ using $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
                • compute $\boldsymbol{S}^{(i)}_{(k_i)}$ using the above
                • update $\boldsymbol{A}^{(i)}_{(k_i)}$ using the above
           ii. for all $\vec{l} = [*, \ldots, *, k_i, *, \ldots, *] \in \mathcal{K}$
                • update $\boldsymbol{P}_{\vec{l}}$ and $\boldsymbol{Q}_{\vec{l}}$ using
                    – $\boldsymbol{U}^{(i)}_{\vec{l}}$ and $\boldsymbol{A}^{(i)}_{(k_i)}$
   until stopping condition
3. Return $\mathring{\boldsymbol{\mathcal{X}}}$
```

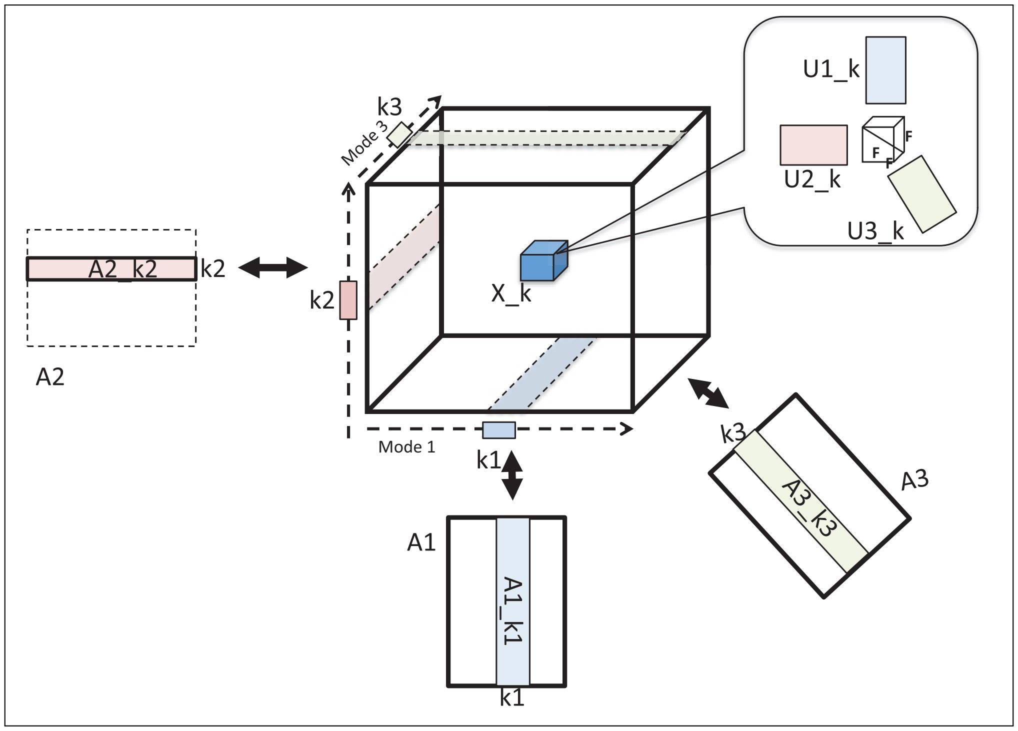
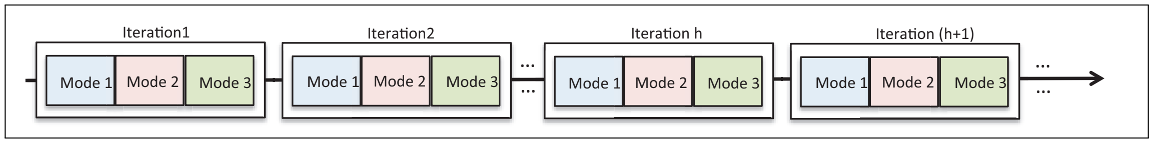


Figure 5.3: For Any Block $\boldsymbol{\mathcal{X}}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$, Its Factors $\boldsymbol{U}^{(1)}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$ Through $\boldsymbol{U}^{(N)}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$ Can Be Used For Maintaining N Sub-factors of $\boldsymbol{\mathcal{X}}$, One Along Each of the N Modes.

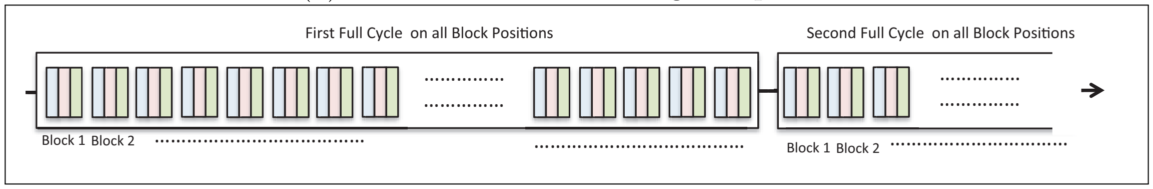
rules are applied does not lend itself into significant amount of data sharing. In this section, I present an alternative *block-centric* way to implement the process of iterative refinement; this alternative process lends itself to better data sharing and memory utilization, thereby helping reduce the overall I/O costs.

5.3.1 Block-centric Scheduling of the Update Rules for Iterative Refinement

The key observation that forms the basis of this alternative iterative refinement process is visualized in Figure 5.3: here, for any block $\boldsymbol{\mathcal{X}}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$, its factors $\boldsymbol{U}^{(1)}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$ through $\boldsymbol{U}^{(N)}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$ can be used for maintaining N sub-factors of $\boldsymbol{\mathcal{X}}$, one along each of the N modes. Therefore an alternative way to implement the iterative refinement process



(a) Mode-centric Scheduling of Updates


First Full Cycle on all Block Positions
Second Full Cycle on all Block Positions
Block 1 Block 2
Block 1 Block 2

(b) Block-centric Scheduling of Updates

Figure 5.4: (a) Mode-centric vs. (b) Block-centric Scheduling of Updates

is to schedule the update rules in a *block-centeric manner* as opposed to the *mode-centeric* manner of the conventional scheme.

The outline of the proposed block-centric iterative refinement process is visualized in Algorithm 7. While the core update-rule is identical to that of the conventional, mode-centric decomposition process [51] (detailed in Algorithm 6) and while the two algorithms have the *same time and space complexities*, these two algorithms differ significantly in the ways the update-rules are scheduled:

- as visualized in Figure 5.4(a), the mode-centric Algorithm 6 considers each mode one at a time, and for each mode it schedules the update rule for all the partitions of that mode; whereas
- as visualized in Figure 5.4 (b), the block-centric Algorithm 7, considers the individual block positions one at a time, and for each block index, $\vec{k}$, it schedules update rules for all N modes together.

One way to see the key difference between these two algorithms is to consider their outer repeat-loops: as also visualized in Figure 5.4, in the mode-centric Algorithm 6, for each cycle of the repeat rule, each sub-factor, $\boldsymbol{A}^{(i)}_{(k_i)}$, along each mode, i, is updated exactly once. In the block-centric Algorithm 7, however, when all the block indexes, $\vec{k} \in \mathcal{K}$, are considered once, the sub-factor $\boldsymbol{A}^{(i)}_{(k_i)}$ along mode i is updated once for each block along the partition k_i; i.e., $\boldsymbol{A}^{(i)}_{(k_i)}$ has been updated $\prod_{j \neq i} K_j$ times.

Definition 5.3.1 (Block-Centric Update Schedule) *Let us consider an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\boldsymbol{\mathfrak{X}} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes. An update schedule, $S = \langle u_1, u_2, \ldots \rangle$, is a sequence, such that each u_j belongs to the set, $\mathcal{K}$, of block positions.*

In other words, a block-centric update schedule drives the order in which Algorithm 7 applies its updates through the order in which the blocks of the tensor are considered. In this dissertation, I consider tensor-filling, cyclic update schedules.

Definition 5.3.2 (Tensor-Filling Schedules) *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ be an N-mode tensor partitioned into a grid of blocks and $\mathcal{K}$ be the indexes of the resulting blocks. A schedule, S, is said to be tensor-filling if S is of the form $C : C : C : \ldots : C'$ (i.e., S can be thought as a repeated concatenation of a cycle sequence, C), such that*

- *the length of the cycle sequence C is equal to $|\mathcal{K}|$,*
- *there exists a one-to-one mapping between $u_j \in C$, and $\vec{k}_i \in \mathcal{K}$,*
- *the last (potentially partial) sequence, C', is a prefix of the cycle sequence C.*

Intuitively, a tensor-filling schedule consists of cycle sequences, each traversing all the blocks of the given tensor – possibly except the last cycle, which may be partial if the termination condition is satisfied before the cycle sequence is completed. A tensor-filling cycle sequence

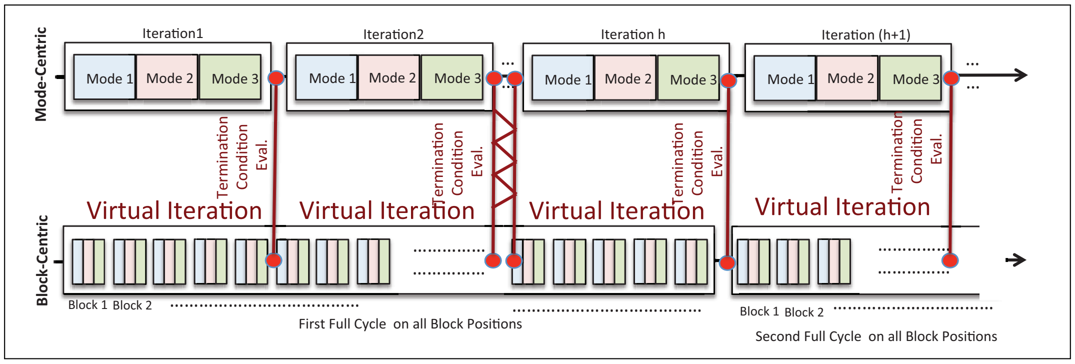


Figure 5.5: Virtual Iterations Are Equal in Length to the Length Of The Iterations of the Mode-centric Update Process and The Block-centric Process Checks for Termination Once for Each Virtual Iteration.

would ensure that all sub-factors of $\boldsymbol{\mathcal{X}}$ are updated using all the data available from the decompositions of its blocks.

5.3.2 Virtual Iterations

In the mode-centric Algorithm 6, the stopping condition is checked once for each iteration of the outer repeat-loop; in other words, each sub-factor, $\boldsymbol{A}^{(i)}_{(k_i)}$, along each mode, i, is updated once. On the other hand, the outer-cycle of the block-centered Algorithm 7 is not necessarily aligned with individual iterations of the outer repeat-loop Algorithm 6. In fact, as visualized in Figure 5.4, each cycle of Algorithm 7 potentially involves many more updates than a single iteration of Algorithm 6. This naturally raises the question of when to check for the termination condition. While the termination condition can be checked at the end of one full cycle, this might result in redundant updates if the termination condition is reached early in a cycle.

Therefore, instead of using cycle boundaries as positions for termination condition evaluation, *virtual iterations* is introduced, which are equal in length to the length of the iterations of the mode-centric update process.

Definition 5.3.3 (Virtual Iteration) *Given an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\mathfrak{X} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes, partitioning each mode i into K_i equal partitions, the length of each virtual iteration is*

$$length\_virtual\_iteration(\mathcal{K}) = \sum_{i=1}^{N} K_i$$

updates of the sub-factors of $\mathfrak{X}$.

Given a tensor-filling, cyclic update schedule S with cycle C, the update schedule C is split into $\frac{\prod_{i=1}^{N} K_i}{\sum_{i=1}^{N} K_i}$ virtual iterations, each of length $length\_virtual\_iteration(\mathcal{K})$.

As visualized in Figure 5.5, I check for termination once for each virtual iteration.

5.4 I/O Reducing Update Schedules

Intuitively, when the available buffer is not sufficient to hold the entire data needed to support the decomposition process, the efficiency of the decomposition will necessarily depend on the effectiveness of the utilization of the buffer. In this section, I consider alternative block-centric update scheduling techniques, which set the order in which blocks are considered in a way to boost data reuse and reduce I/O needed to obtain tensor decomposition.

In Section 5.2.1, I had formalized the amount of data needed for each iteration of the mode-centric iterative improvement algorithm. I now formalize the amount of data needed for each step of the block-centric update process.

Definition 5.4.1 (Unit of Data Access) *In Algorithm 7, for each $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$, I need to bring into the memory, for each mode $i = 1$ to N, (a) the sub-factor, $\boldsymbol{A}^{(i)}_{(k_i)}$, of the corresponding mode partition, and (b) the i^{th} mode factors of all blocks corresponding to the mode partition k_i; i.e., $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$. Therefore, for a given a block position, $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$,*

$$data(\vec{k}, i) = \left\{ \boldsymbol{A}^{(i)}_{(k_i)};\;\; \boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]} \right\}, i \in \{1 \ldots N\}$$

are the N units of data needed for implementing the update corresponding to this block position.

This is visualized in Figure 5.6. Therefore the data in the buffer can be organized in terms of mode-partition pairs,

$$\langle i, k_i \rangle = \left\{ \boldsymbol{A}^{(i)}_{(k_i)};\ \ \boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]} \right\},$$

of size (assuming 8-byte double precision representation)

$$\left(\underbrace{\left(\frac{I_i}{K_i} \times F\right)}_{\boldsymbol{A}^{(i)}_{(k_i)}} + \underbrace{\left(\left(\prod_{j \neq i} K_j\right) \times \left(\frac{I_i}{K_i} \times F\right)\right)}_{\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}} \times 8 \right) \quad \text{bytes}$$

as also discussed in Section 5.2.1. Once brought into memory, such a pair can be used for implementing the factor matrix revision updates at any block position $\vec{k}$ for which the i^{th} mode partition is equal to k_i.

5.4.1 Re-Use Promoting Schedules

As seen above, the update process corresponding to two distinct block positions, $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$ and $\vec{l} = [l_1, \ldots, l_N] \in \mathcal{K}$, can share the same $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$ matrices, if $k_i = l_i$ for some mode i, $1 \leq i \leq N$ (i.e., they are along the same mode partition on mode i). The larger the number of common mode partitions between $\vec{k} = [k_1, \ldots, k_N]$ and $\vec{l} = [l_1, \ldots, l_N]$, the larger will be the sharing of $\boldsymbol{U}$ matrices. This leads me to my primary desideratum.

Desideratum 1 (Reuse-Promoting Schedules) *The cycle sequence, $C = \langle u_1, u_2, \ldots, u_m \rangle$ of the update schedule S should be such that, the closer u_a and u_b are to each other (i.e., the smaller $|a - b|$ is), the larger the intersection between the corresponding block partitions, $\vec{k_a}, \vec{k_b} \in \mathcal{K}$.*

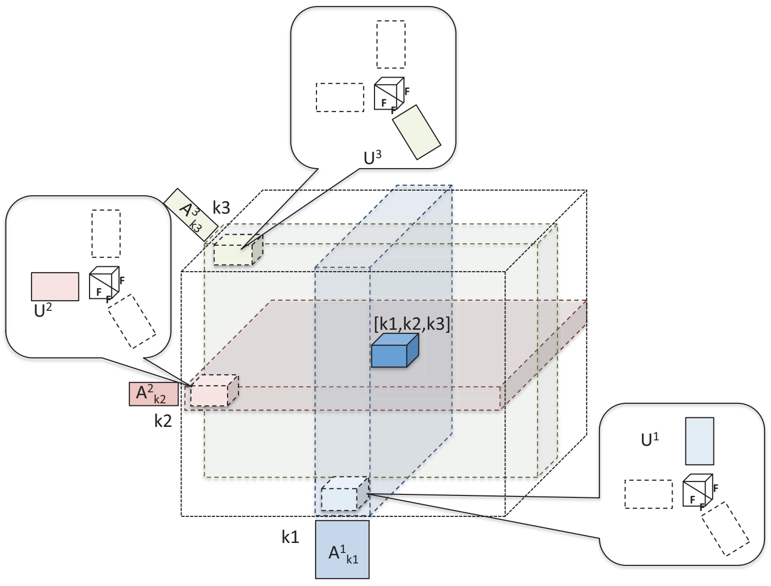


Figure 5.6: For a Given Block Position $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$, I Need To Bring into the Memory, for Each Mode $i = 1$ to N, The Data Unit $data(\vec{k}, i)$ Consisting of the Sub-factor, $\boldsymbol{A}^{(i)}_{(k_i)}$ And $\boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]}$.

As experiments validated, the more reuse-promoting a schedule is, the lower is the number of accesses to the disk. I next consider alternative update schedules, with different traversal patterns of the tensor blocks.

5.4.2 Fiber-Order Update Schedules

The first, straightforward, alternative is to traverse the tensor blocks one fiber at a time as shown in Figure 5.7(a). This strategy is very simple to implement in the form of a set of nested loops, with one loop for each mode.

Fiber-order update schedules support significant amount of data re-use. To see why, consider two consecutive block positions $\vec{k}_h = [k_{h,1}, \ldots, k_{h,N}]$ and $\vec{k}_{h+1} = [k_{h+1,1}, \ldots, k_{h+1,N}]$, visited as the schedule traverses along a single fiber. It is easy to see that, in most cases, the only difference between these two positions will be in their N^{th} position; i.e., $\forall_{i<N}$ $(k_{h,i} =$

$k_{h+1,i}$) and ($k_{h+1,i} = k_{h,i} + 1$). Consequently, $\forall_{i<N}\ data(\vec{k}_h, i) = data(\vec{k}_{h+1}, i)$. Therefore (assuming that N data units fit into memory), once all the data needed for the block position, $\vec{k}_h$, are brought into memory, the only new data unit that may need to be fetched from the disk for the block position, $\vec{k}_{h+1}$, is $data(\vec{k}_{h+1}, N)$.

5.4.3 Fractal-based Update Schedules

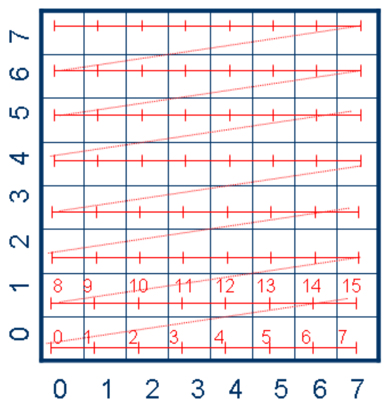
We, however, note that I can have even better data reuse by using fractal-structured block traversals, instead of relying on this simple fiber-order scheduling strategy. A fractal curve, thus, is a curve that looks similar when one zooms-in or zooms-out in the space that contains it. Fractals that are space-filling, such as Z-order curve [44] and Hilbert curve [24] are known to show good clustering properties in the sense that these curves tend to completely traverse a *neighborhood* of the space before moving to another neighborhood.

My intuition is that, if two block locations $\vec{k}_a = [k_{a,1}, \ldots, k_{a,N}]$ and $\vec{k}_b = [k_{b,1}, \ldots, k_{b,N}]$, share significant amount of data, then traversals based on space-filling fractal curves are likely to visit them close to each other. Based on this intuition, I propose two update scheduling techniques, based on *Z-order* and *Hilbert-order traversal of tensor blocks*.

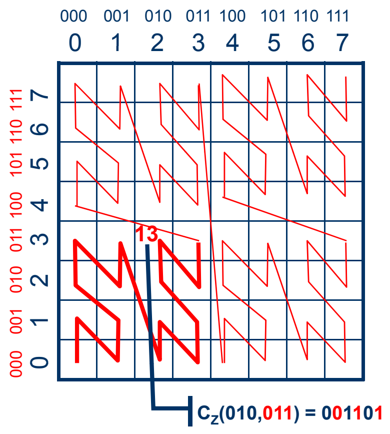
5.4.4 Z-Order Update Schedules

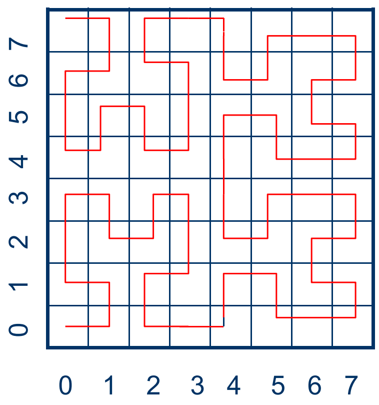
Z-order (or Morton-order) curve [44] is a fractal-based curve that fills the space more effectively than the fiber-order traversal described above. In particular, as mentioned above its fractal nature ensures that it clusters nearby block positions (i.e., nearby block positions are visited close to each other during traversal).

Let us consider an N-mode tensor where each mode i is partitioned into 2^m partitions for some $m > 0$ and let $\vec{k} = [k_1, k_2, \ldots, k_N]$ be a block position. Then, the Z-value corresponding



(a) fiber-order schedule





(b) z-order (c) Hilbert-order

Figure 5.7: Alternative Update Schedules for the Blocks of a 2-mode Tensor (the Numbers along a given Mode Denote the Block Indexes Along That Mode)

to $\vec{k}$ is an integer, $zvalue(\vec{k})$, defined as follows: $\forall_{1\leq i\leq N}\forall_{1\leq j\leq m}$

$$zvalue(\vec{k}).\mathtt{base2}((m-j)N+i) = k_i.\mathtt{base2}(j),$$

where, given an integer a $(0 \leq a \leq 2^m - 1)$, $a.\mathtt{base2}(j) \in \{0,1\}$ denotes the value of the j^{th} least significant bit of a.

Figure 5.7(b) provides an example. In this example, the block position $[2,3]$, has the corresponding Z-order value, 001101_2 $(= 13_{10})$, which can be obtained by shuffling the bits of the inputs, 010_2 $(= 2_{10})$ and 011_2 $(= 3_{10})$ as specified by the above formula. This example also shows that the Z-order traversal of the space is self-similar (i.e., composed of "Z"s at multiple scales) and clustered: block-positions that are closer to each other on the grid tend to be visited closer to each other also in traversal sequence. I argue that this property of the Z-order traversal should help provide a more re-use promoting schedule than the fiber-order traversal.

5.4.5 Hilbert-Order Update Schedules

A second look at Figure 5.7(b), however, points to a potential weakness of the Z-order traversal: the Z-order traversal of the blocks contains a few relatively large jumps and, at these points, the Z-order based schedule may result in a large amount of data to be brought into the memory from the disk. Therefore, the Hilbert (or Peano-Hilbert) curve [24], which tends to have smaller jumps, may provide a higher degree of reuse-promotion. Figure 5.7(c) shows a sample Hilbert traversal of blocks assuming a 2 mode tensor. As I see here, Hilbert traversal relies on "`U`" shaped curve-segments (as opposed to the "`Z`" shaped curve-segments of the Z-order traversal) and this helps better preserve the adjacency property (i.e., avoiding discontinuity - which would require undesirable jumps).

As experiments verify, Hilbert-order based traversal indeed provides lower I/O costs than Z-order based scheduling of the updates. However, one difficulty with the Hilbert-order traversal is that, unlike the Z-order traversal, there does not exist an efficient way to map from the block positions to the positions on the Hilbert curve (and vice versa). Existing algorithms are not practical for tensors with large numbers (10s or 100s) of modes as they may require large amounts of memory. Therefore, in those cases, Z-order traversal, which (as described above) has very efficient mapping implementations, may be preferred over Hilbert-order traversals of block positions.

5.5 Update Schedule Aware Buffer Replacement

So far, I focused on the problem of selecting an update schedule for tensor decomposition refinement such that the total amount of data that need to be brought from disk to the buffer is minimized. In this section, I argue that the I/O needed to perform the iterative refinement can be further reduced by using buffer replacement policies that complement the traversal order, driving the update scheduling process. In particular, I argue that the common *least-recently used (LRU)* buffer replacement strategy[2], which relies on the *temporal-locality* principle (i.e., assumes that data brought to the memory recently is likely to be used also in the near future) and drops the data which have been used furthest in the past, is not likely to be effective.

5.5.1 Fiber-Order Schedules and MRU

Let us first consider the fiber-order strategy described in Section 5.4.2 and visualized in Figure 5.7(a). It is easy to see that data brought to the memory for a given block of a fiber will not be accessed again during the traversal of that fiber. This implies that temporal locality principle does not hold during fiber-order traversal. In contrast, due to the looping characteristic of the traversal, a *temporal-$\underline{a}$locality* principle (which states that data brought to the memory for a block $\underline{\text{is not}}$ likely to be used in the near future) holds and this implies that a *most-recently used (MRU)* buffer replacement strategy may be more suitable for fiber-order schedules.

5.5.2 Forward-Looking Buffer Replacement

More importantly, though, the definition of the unit of data access (Definition 5.4.1 in Section 5.4), along with the proposed update scheduling techniques, enable more precise

[2]For example, SciDB[9] array database underlying TensorDB [31, 32] implements LRU-based buffer management.

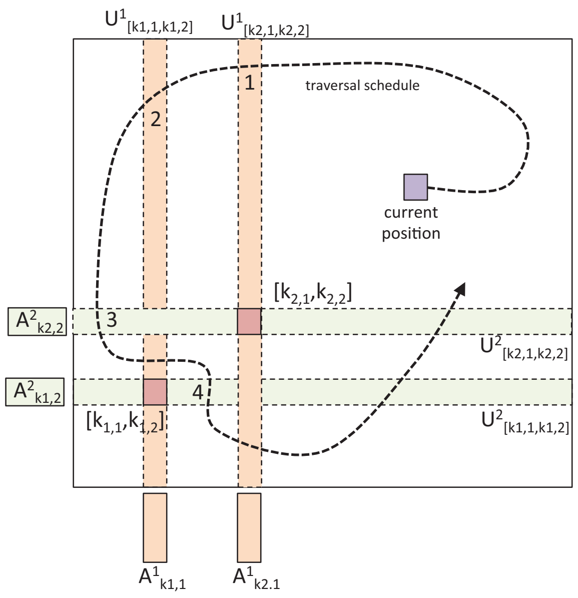


Figure 5.8: Forward-looking, Schedule-aware Buffer Replacement: Let Us Assume That the Buffer Currently Contains the 4 Shown Data Units, 2 for Each Of $[k_{1,1}, k_{1,2}]$ and $[k_{2,1}, k_{2,2}]$. Since, According to the Current Traversal Plan, $data([k_{1,1}, k_{1,2}], 2) = \{\boldsymbol{A}^{(2)}_{(k_{1,2})};\ \ \boldsymbol{U}^{(2)}_{[k_{1,1},k_{1,2}]}\}$ is the Last Data Unit To Be Needed, It Will Be the One Selected For Replacement

forward-looking, and traversal order aware, predictive buffer replacement strategies as opposed to the *backward-looking* strategies, such as LRU and MRU. In particular, the data in the buffer are organized in the form of mode-partition pairs, $\langle i, k_i \rangle = \left\{ \boldsymbol{A}^{(i)}_{(k_i)};\ \ \boldsymbol{U}^{(i)}_{[*,\ldots,*,k_i,*,\ldots,*]} \right\}$. Once brought into memory, a $\langle i, k_i \rangle$ pair is used for updates at any block position $\vec{k}$ for which the i^{th} mode partition is equal to k_i. Consequently, given the current position in an update schedule, if I can compute for each mode-partition pair $\langle i, k_i \rangle$ in the buffer, how far in the future the traversal will cross that mode-partition pair, then I can select for replacement the pair that will be crossed furthest in the future (Figure 5.8).

While such forward-looking replacement policies are difficult to implement when the data accesses are irregular and unpredictable, thanks to the regular natures of fiber-, Z-, and Hilbert-order traversals, it is possible to compute in advance precisely how far in the future (i.e., how *urgently*) a given data unit that is brought into the buffer will be needed again. This enables us to maintain the data units in an order of *urgency* and, if needed, replace the least urgent data unit. As expriments shown, this forward-looking, traversal-order aware replacement policy significantly reduces the I/O cost of the decomposition process.

5.6 Experimental Evaluation

In this section, I report experiments that aim to assess the effectiveness of the proposed disk and buffer sensitive update scheduling and buffer management techniques underlying `2PCP`. In particular, I aim to observe and report the amount of I/O (i.e., data swaps) necessitated by different update schedules under different partitioning strategies, memory availabilities, and buffer replacement strategies.

5.6.1 Experiments with Strong Configuration

In these experiments, I considered billion-scale (with $\sim$ 1 billion non-zero entries) dense tensors and compared the execution time performance of `2PCP` to that of another billion-scale tensor decomposition platform, HaTen2 [26]. Note that unlike `2PCP` designed for scientific applications, HaTen2 is designed for handling sparse tensors, commonly found in social media applications.

Hardware. For these experiments, I used EC2 platform with R3.xlarge configurations: I deployed `2PCP` and HaTen2 each on 8 Intel Xeon E5-2670 v2 (Ivy Bridge) Processors (4 CPUs, each with 30.5GB and 80 GB SSD storage).

Software. Distributed version of `2PCP` was implemented in Java 7, over Hadoop 0.20.2. The binary code for HaTen2 was obtained from `http://datalab.snu.ac.kr/haten2`.

In Table 5.1, I compare the execution time performance of `2PCP` to that of HaTen2 [26] for dense tensors of different sizes. As I mentioned earlier, HaTen2 is designed for handling

Tensor size	2PCP (sec.)	HaTen2 (sec.)
$500 \times 500 \times 500$ (0.025B non-zeros)	92.9	2380.2
$1000 \times 1000 \times 1000$ (0.2B non-zeros)	441.5	11764.9
$1500 \times 1500 \times 1500$ (0.7B non-zeros)	1513.9	FAILS

Table 5.1: Comparison of Execution times on Billion-scale Dense Tensors (Density 0.2; Target Rank 10; Results Reported Here Use a $2 \times 2 \times 2$ Partitioning Strategy for `2pcp`; Due to the Large Execution Time of Haten2, I Only Report Execution Time for 1 Iteration)

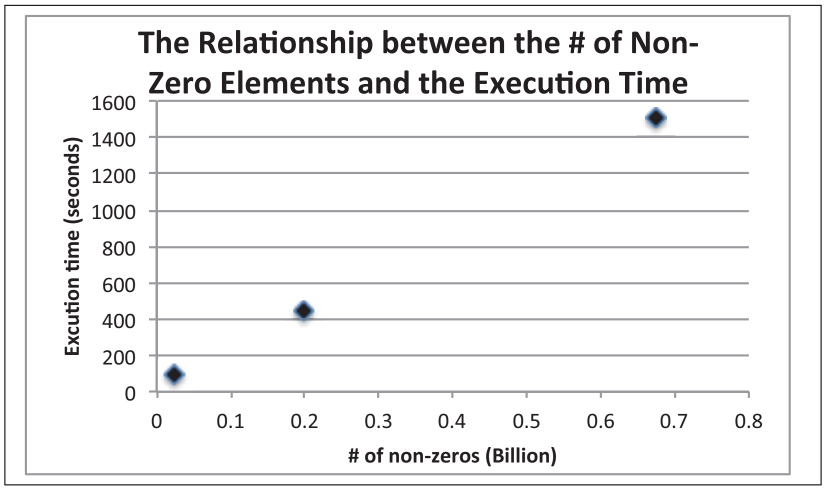


Figure 5.9: `2PCP` Scales Well as the Tensor Size Grows (Data From Table 5.1)

sparse tensors, commonly found in social media applications, whereas `2PCP` (motivated by scientific and engineering applications) does not make this assumption. The results reported in Table 5.1 confirm this: in this table and Figure 5.9, while `2PCP` scales well as the tensor size grows, HaTen2 requires significantly more time and memory and soon fails to run with the available resources.

It is important to note that this execution time gain does not come with any loss in accuracy. In fact, `2PCP` provides significantly higher accuracy than that of HaTen2. For example, for the case with $0.025B$ non-zeros, the fit measure (described in Section 2.2) for

# Part.	Phase I	Phase II		Total	
	BD (per block)	LRU	FOR	LRU	FOR
Naive CP	>12 hours	N/A	N/A	N/A	N/A
$2 \times 2 \times 2$	79.1	10.6	9.6	89.7	88.7
$4 \times 4 \times 4$	9.8	64.3	54.5	74.1	**64.4**

Table 5.2: Execution Times (in Minutes)

`2PCP` is 0.077 whereas HaTen2's fit for the same configuration is only 0.0011.

5.6.2 Experiments with Weak Configuration

In addition to the configuration considered above, I also ran experiments with a weaker configuration, consisting of quad-core Intel(R) Core(TM)i5-2400 CPU @ 3.10GHz machines with 8.00GB RAM. Since in this configuration the main-memory is much smaller, `2PCP` is implemented on top TensorDB (obtained from `https://github.com/mkim48/TensorDB` and installed on SciDB 12.12 [9], using Python and C++) to enable out-of-core CP-ALS computations in Phase 1.

In Table 5.2, I compare Naive CP-ALS against `2PCP` with LRU and the forward-looking (FOR) strategies (both under the proposed Z-order update scheduling scheme) for different partition scenarios, for a $1000 \times 1000 \times 1000$ of high density (0.49). The target rank was set to 100. Here, the execution time for the first phase includes the time for obtaining and decomposing each block. The second phase was ran until convergence. The table also includes times for the conventional CP-decomposition (i.e., default TensorDB with no partitioning [31, 32]).

The first thing to note in this table is that, compared to conventional CP tensor decomposition (without partitioning), the fine-grained decomposition strategies that operate in a block-centric manner, significantly improve the execution time of tensor decomposition

Parameter	*Alternative values*
# partitions	$2 \times 2 \times 2$; $4 \times 4 \times 4$; $8 \times 8 \times 8$
Buffer size (portion of the total space requirement)	1/3; 1/2; 2/3
# (virtual) iterations	100; 200
Schedules	Mode-centric (MC); Fiber-order (FO); Z-order (ZO); Hilbert-order (HO)
Replacement	LRU; MRU; Forward (FOR)

Table 5.3: Parameter Settings (Unless Otherwise Specified)

process. Secondly, I see that, as expected, the forward-looking buffer replacement (FOR) outperforms the LRU buffer replacement strategy. The fastest execution time was obtained using the $4 \times 4 \times 4$ strategy, where the forward strategy (FOR) completed in 64.4 minutes, against 74.1 minutes for the LRU strategy, a $\sim 15\%$ gain. These results are especially significant when considering that (a) this tensor cannot be decomposed using a fully in-memory Matlab based strategy and (b) a naive secondary-storage supported CP tensor decomposition using TensorDB runs more than 12 hours for the same configuration.

5.6.3 Parameter Analysis (Stand-Alone Configuration)

In the next set of experiments I study the impact of various parameters on the data swap and accuracy performance of `2PCP`. These experiments are running on a stand alone version of `2PCP` developed to support system-independent evaluation of the algorithms underlying `2PCP`: in order to count data swaps precisely, this version is implemented and run using Matlab 7.11.0 (2010b) and Tensor Toolbox Version 2.5 [2].

Evaluation Criteria - Data Swaps. Since (a) the wall-clock execution times depend on the particular hardware/software setting (including the disk page read/write times and data compression/decompression costs – based on whether the data is compressed on the disk) and (b) since the block-based decomposition process is I/O-bound[3] I observe and report the amount of I/O (i.e., data swaps) between the disk and memory buffer for different scenarios, as is common in the buffer/cache management literature.

Evaluation Criteria - Accuracy. I use the measure reported in Section 2.2 to assess decomposition accuracy. Each experiment is ran 10 times and I report median results.

Data. I used four real datasets with different characteristics: *Epinions* [57], *Ciao* [57], *Enron* [52], and Face [19]. The first two of these are comparable in terms of their sizes and semantics: they are represented in the form of $170 \times 1000 \times 18$ (density 2.4×10^{-4}) and $167 \times 967 \times 18$ (density 2.2×10^{-4}) tensors, respectively, and both have the schema $\langle user, item, category \rangle$. The *Enron* email data set, however, is larger ($5632 \times 184 \times 184$, density 1.8×10^{-4}) and has a different schema, $\langle time, from, to \rangle$. The *Face* data set, a benchmark for research of face recognition, is a **dense** $480 \times 640 \times 100$ tensor with schema $\langle$*x-coord*, *y-coord*, *image*$\rangle$ and density 1.0. For these experiments, I set the stopping condition to an accuracy improvement of less than 10^{-2} per iteration; but, I also set a maximum number of (virtual) iterations to help observe how quickly iterative improvement converges under different strategies. In these experiments, I set the target decomposition rank to 100.

Parameter Settings. The number of data swaps necessary for iterative improvement is not a function of the absolute data size, but the number of partitions and the *size of the buffer relative to the total space requirement* (see Section 5.2.1). Therefore, as I report in Table 5.3, in these experiments, I primarily vary the number of partitions of the tensors and the size of the memory buffer, relative to the total space requirement for the decomposition process.

[3]I observed that, on the average, swapping a block takes ~ 3 times more than the time needed to perform the in-memory operations on the block.

Amount of I/O (Data Swaps)

In Figure 5.10, I see per-(virtual)iteration data swaps for different scheduling and replacement algorithms, and different buffer sizes. Since the number of per-iteration swaps is not a function of the data, but the number of partitions and the *size of the buffer relative to the total space requirement*, I have the same result for all data sets. To see the impact of scheduling on convergence, the scheduling process was run without any bound on iterations. In this figure 5.10, for all configurations the conventional mode-centric (MC) update plans result in the highest amount of I/O. In contrast, the proposed block-centric schedules require significantly lesser I/O, especially when combined with the forward-looking, schedule aware buffer replacement strategies. The worst strategy is the mode-centric (MC) schedules with LRU buffer replacement, with up to $\sim$ 24 swaps per iteration, for $8 \times 8 \times 8$ partitions, independent of the buffer availability; while MRU based replacement brings the number of swaps down, MC is overall the worst strategy. In contrast, block-centric Hilbert-order schedules (HO) with forward-looking (FOR) buffer replacement has as low as ~ 1.1 swaps per iteration for $8 \times 8 \times 8$ partitions with $\frac{1}{3}$ buffer availability and ~ 0.22 swaps per iteration for the same partition with $\frac{2}{3}$ buffer availability.

To give context, let us consider a $100K \times 100K \times 100K$ tensor partitioned into $8 \times 8 \times 8$ blocks. Let me also assume that my target rank is 100. The best case for MC is on the average ~ 8.32 swaps per iteration with MRU, corresponding to (assuming *double-precision* number representation)

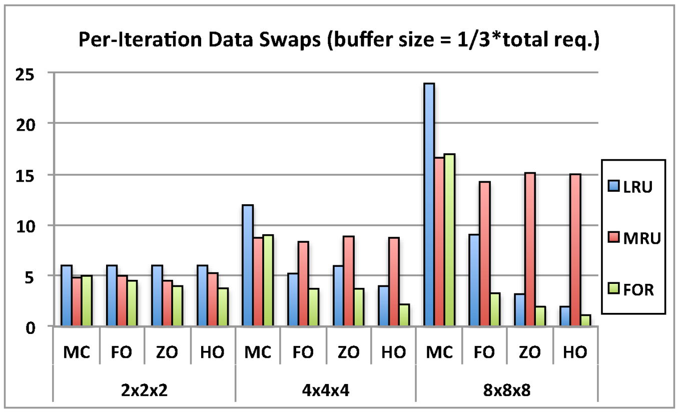
$$8.32 \times \left(\left(\frac{10^5}{8} \times 100 \right) + \left((8 \times 8) \times \left(\frac{10^5}{8} \times 100 \right) \right) \times 8 \right)$$

$\simeq 6GB$ data exchange per iteration (Section 5.4). In contrast, for the same configuration, Hilbert-order (HO) schedule with forward-looking replacement (FOR) requires only ~ 0.22 swaps per iteration, corresponding to only $\sim 160MB$ data exchange per iteration.

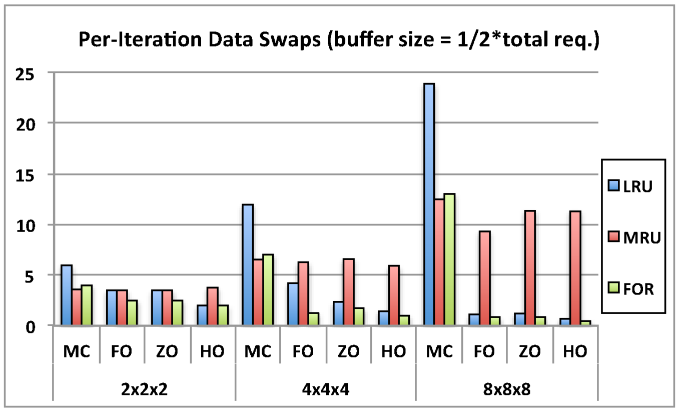
Accuracy Results

In Figure 5.11, I see the accuracy results for different data sets, scheduling algorithms, and partition configurations The charts in the figure plot the *relative accuracy difference* between the block-centric algorithms (fiber-order, FO, Z-order, ZO, and Hilbert-order, HO) and conventional mode-centric scheduling (MO). Positive relative difference indicates cases where buffer-centric approach outperforms mode-centric approach.

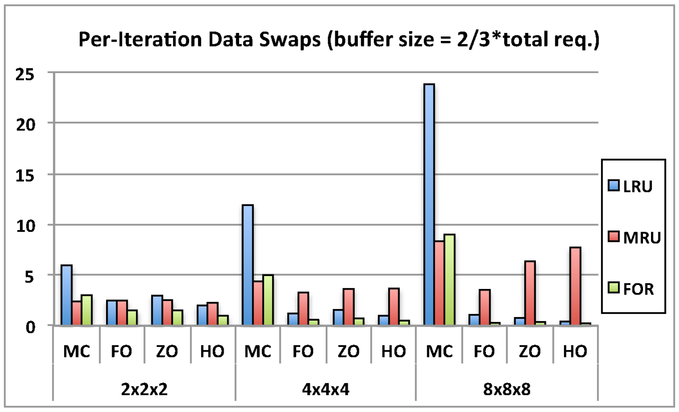
Except for a few instances (specifically Enron data set, $2 \times 2 \times 2$ partitions), the accuracies of the block-centric algorithms (especially Hilbert-order, HO) do match or exceed the accuracies of the mode-centric algorithm. As expected, the variability is higher for the sparse data sets as the accuracy of the block-based iterative improvement strategy depends highly on the densities of the blocks and, on sparse data sets, densities of the blocks can vary significantly. For the dense Face data set, accuracies for the mode- and block-centric algorithms are virtually identical.



(a) Buffer size is $\frac{1}{3}$ of total data req.

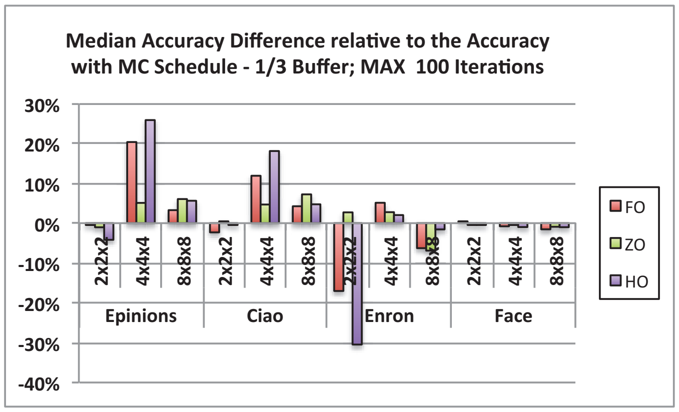


(b) Buffer size is $\frac{1}{2}$ of total data req.

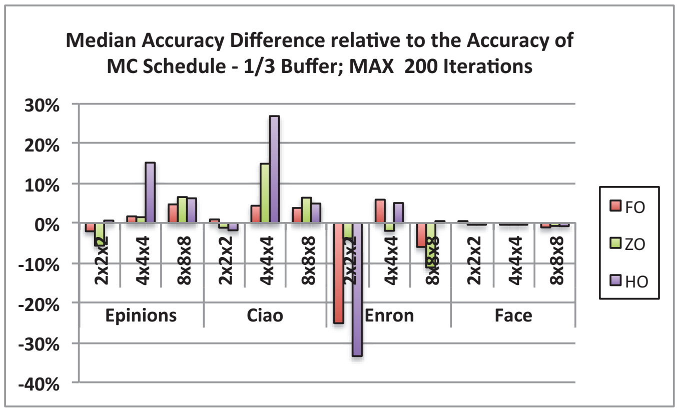


(c) Buffer size is $\frac{2}{3}$ of total data req.

Figure 5.10: Per-(Virtual)Iteration Number of Data Swaps for Different Configurations (since the Per-iteration Number of Swaps Is Not A Function of the Data, but the Number of Partitions and the *Size Of the Buffer Relative to the Total Space Requirement*, I Have The Same Result for All Data Sets)



(a) Max. 100 iterations; buffer size is $\frac{1}{3}$ of total data req.



(b) Max. 200 iterations; buffer size is $\frac{1}{3}$ of total data req.

Figure 5.11: Relative Accuracy Difference: Positive Values Indicate Cases Where Buffer-centric Approach Outperforms Mode-centric Approach.

Chapter 6

MULTI-TASK TENSOR DECOMPOSITION FOR SPARSE ENSEMBLE SIMULATION

6.1 Introduction

Data- and model-driven computer simulations are increasingly critical in many application domains. For example, for predicting geo-temporal evolution of epidemics and assessing the impact of interventions, experts often rely on epidemic spread simulation software, such as STEM [40]. Simulation-based decision making, however, introduces several fundamental data challenges [42, 55]:

- Many complex processes (such as disasters [46]) involve various distinct, yet inter-dependent, sub-processes. Consequently, in order to be useful, these simulations may track 100s of parameters, spanning multiple layers and spatial-temporal frames, affected by complex inter-dependent dynamic processes (Figure 6.1).
- Moreover, due to large number of unknowns, decision makers usually need to generate an ensemble of stochastic realizations, requiring 1000s of individual simulation instances, each with different parameter settings corresponding to different, but plausible, scenarios.

Consequently, obtaining and interpreting simulation ensembles to generate actionable results present difficulties:

- **Limited ensemble simulation budgets:** Since complex, inter-dependent parameters affected by complex dynamic processes have to be taken into account, *execution of simulation ensembles can be very costly.* This leads to *simulation budget constraints* that limit the number of simulations one can include in an ensemble.

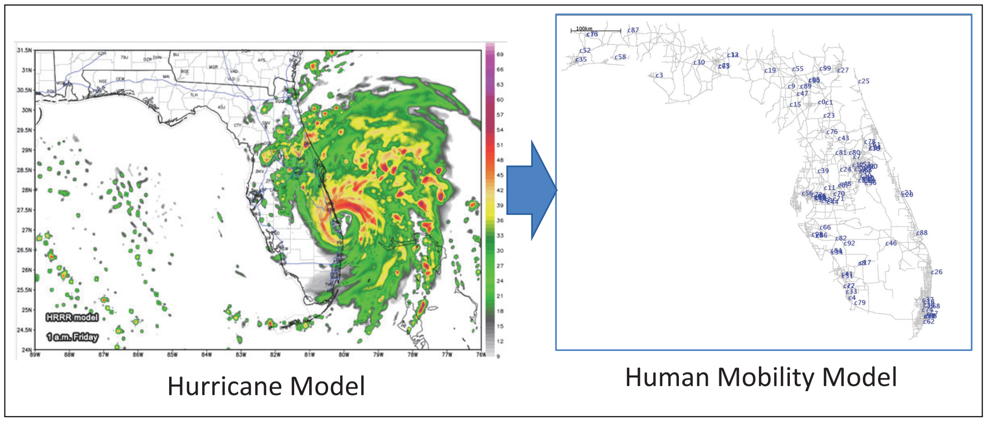


Figure 6.1: Coupled Simulation of a Hurricane and Human Mobility

- **Need for post-simulation data processing:** Because of the complexities of key processes and the varying scales at which they operate, experts often lack the means to drive conclusions from these ensembles. This leads to the need for *data analytics on simulation ensembles to discover broad, actionable patterns.*

- **Inherent data sparsity of simulation ensembles:** While the size and complexity of a simulation ensemble can indeed tax decision makers, I note that *a simulation ensemble is inherently sparse (relative to the space of potential simulations one could run)*, which constitutes a significant problem in simulation-based decision making. This leads to the following critical question: "*Given a parameter space and a fixed simulation budget, which simulation instances I should include in the ensemble?*"

6.2 Background and Notation

6.2.1 Tensor Representation of Simulation Ensembles

I propose a tensor-based framework to represent and analyze large simulation ensembles. Intuitively, the tensor model maps a multi-attribute schema to a multi-modal array (where each potential tuple is a tensor cell). Consequently, I can map a given simulation ensemble onto a tensor such that each simulation parameter corresponds to a mode of a tensor and the non-null entries in the tensor represent results of the simulations I have executed.

Tensor decomposition [23, 58, 36] (which generalizes matrix decomposition to tensors) has been successfully used in various applications, such as social networks, sensor streams, and others [35]. Intuitively, the tensor decomposition process rewrites the given tensor in the form of a set of *factor* matrices (one for each mode of the input tensor) and a core matrix (which, intuitively, describes the spectral structure of the given tensor).

As such, tensor decomposition has also been used for the analysis of dynamical systems: [53] proposed a tensor-based model for time series and [34] proposed a dynamic mode decomposition (DMD) scheme for the analysis of the behavior of complex dynamical systems.

6.2.2 Inherent Sparsity of Ensembles

While, as discussed above, tensors have been successfully used for understanding dynamic systems, I note that when the data is sparse, tensor decomposition is less effective in extracting meaningful information – which is a significant challenge when I am attempting to learn about dynamic processes through an *inherently sparse* ensemble of simulations. To see why, note that as the number of input parameters of a simulation increases, the number of potential situations one can simulate increases exponentially. Consider for example, the simple dynamical system, *double equal-length pendulum*, depicted in Figure 6.2: in this system there are five parameters that one can control: (a) the initial angle of the first pendulum ϕ_1, (b) the initial angle of the second pendulum ϕ_2, (c) the weight of the first bob m_1, (d) the weight of the second bob m_2, and (b) the gravity, g. For each combination of parameter values, the system can be viewed as a two-variate time series consisting of the angles of the pendulums at each time step. It is easy to see that the number of potential simulations of this *double equal-length pendulum* system is a function of the resolution of each of these four parameters – if I simply assume that for each parameter I consider, say, 20 distinct values, this would lead to $20^5 = 3200000$ possible simulations to potentially consider. Assuming that I have a simulation budget, $B = 1000$, this would lead to a simulation density of only $1000/3200000 \sim 0.0003125$. Therefore, even for a relatively small number of parameters, any realistic simulation budget is likely to be much smaller than the possible space of all simulations – consequently, the naive approach of randomly sampling the simulation space is likely to lead to sparse tensors that are difficult to accurately analyze.

6.2.3 Tensor Representation of a Complex System

Let us be given a complex dynamic system, S, with N input parameters, such that the i^{th} input parameter can take I_i distinct values. For simplicity of the discussion, let us further assume that for each input parameter combination $\langle v_1, \ldots v_N \rangle$, the complex dynamic system S generates a single value $S(v_1, \ldots, v_n)$. Let, further, Y be the set of all simulations of the system S one can execute and the corresponding results; i.e., $Y = \{y_i = \langle \langle v_{i,1}, \ldots, v_{i,N} \rangle, S(v_{i,1}, \ldots, v_{i,N}) \rangle \parallel 1 \leq i \leq I_1 \times I_2 \times \ldots \times I_N\}$. It is easy to see that Y can be encoded as a tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, where for all $y_i \in Y$, the tensor cell $\boldsymbol{\mathcal{Y}}(v_{i,1}, \ldots, v_{i,N})$ has the value $S(v_{i,1}, \ldots, v_{i,n})$.

6.2.4 Tensor Representation of a Simulation Ensemble

The number, $I_1 \times \ldots \times I_N$, of simulations of the system, S, one can run can be very large. Instead, as discussed in the introduction, a much smaller subset (or ensemble) of the simulations is executed to get an idea about S. Given an ensemble of, $B \ll I_2 \times \ldots \times I_N$ simulations, let X be the set of simulations that have been selected to be executed as well as the corresponding system outputs; i.e., $X = \{x_i = \langle \langle v_{i,1}, \ldots, v_{i,N} \rangle, S(v_{i,1}, \ldots, v_{i,N}) \rangle \parallel 1 \leq i \leq B\}$. It is easy to see that X can be encoded as a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, where for all $x_i \in X$, the tensor cell $\boldsymbol{\mathcal{X}}(v_{i,1}, \ldots, v_{i,N})$ has the value $S(v_{i,1}, \ldots, v_{i,N})$ and all other cells have null values (indicating simulations that could potentially have been run, but have not

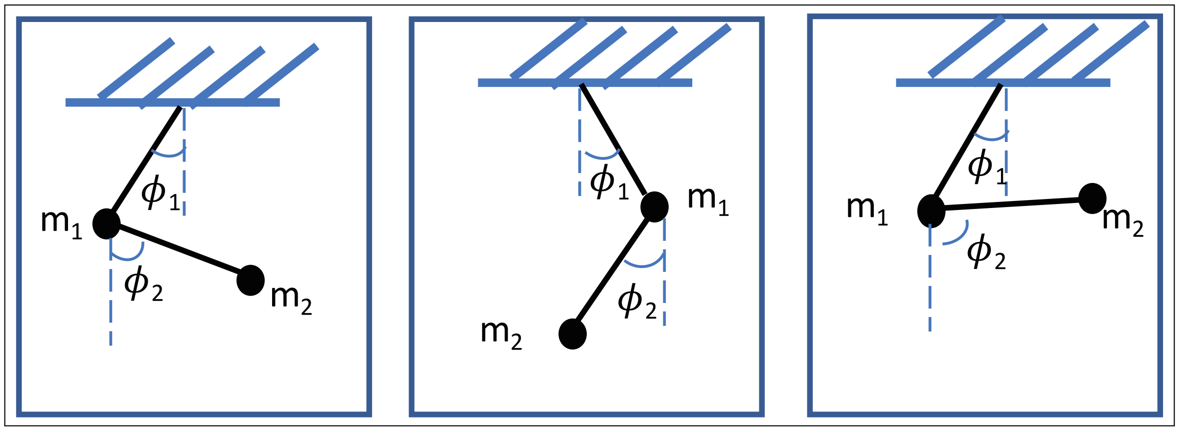


Figure 6.2: States of a Multi-pendulum System

been included in the ensemble). Since $B \ll I_1 \times I_2 \times \ldots \times I_N$, the tensor $\boldsymbol{\mathcal{X}}$ is very sparse, meaning that there will be many more null-valued cells than the cells recording real-valued simulation results; i.e., $B \ll I_1 \times I_2 \times \ldots \times I_N$.

6.3 Contribution

6.3.1 Contribution 1: Density Boosting Partition-Stitch Sampling

I propose an alternative ensemble creation strategy, which I refer to as the *partition-stitch sampling* (Figure 6.3): given an N-parameter simulation and an ensemble budget of B, instead of randomly allocating the B samples in the N-dimensional parameter space, I partition the simulation space into $\sim N/2$ dimensional sub-spaces and allocate $B/2$ simulations for each sub-space: note that, since the number of possible simulations for each sub-space reduced exponentially (in the number of excluded parameters), this corresponds to an exponential increase in the density of the samples for each sub-space: let us re-consider the *double equal-length pendulum* system in Figure 6.2: instead of considering the original 5-parameter system, I can divide the simulation space into simulations for two 3-parameter systems:

- **System 1:** In this system, I am allowed to *vary* the initial angle, ϕ_1, and weight, m_1, of the first pendulum as well as the gravity, g; but the initial angle, ϕ_2, and weight, m_2, of the second pendulum are *fixed*.

- **System 2:** In the second system, I can *vary* the initial angle, ϕ_2, and weight, m_2, of the second pendulum as well as the gravity, g; in this case, the initial angle, ϕ_1, and weight, m_1, of the first pendulum are *fixed*.

Note that neither of the two systems are perfect representations of the overall behavior of the whole system as, in both cases, two out of the five parameters are fixed to some default values. However, the simulation densities of both systems are now much higher than the simulation density of the original system: using the numbers considered earlier, each sub-system has 3 parameters with 20 distinct values, leading to a parameter space of $20^3 = 8000$ simulations. If I allocate 500 (=1000/2) simulations to each sub-space, this leads to a simulation density of $500/8000 = 0.0625$, which is 200 time denser than the original simulation space. There, however, remain several important questions:

- The first important question is "*How do I stitch back the results obtained from the individual sub-spaces?*"

Here I may have several alternatives: In the simplest alternative, all the simulations from the two systems can be unioned into a single 5-mode tensor and this 5-mode tensor can be decomposed for analysis. This is potentially very expensive as the decomposition cost often increases exponentially with the number of modes of the input tensor [33, 41]. I will also see that, once unioned into a single tensor, the overall density is still low and the accuracy gains will be very limited.

Instead, I present a join-based scheme to increase the *effective density* of the ensemble. In particular, I will present two approaches (join stitching and zero-join stitching) to combine simulation results form the sub-systems and experimentally validate the effectiveness of these schemes. Several questions, however, remain:

- *How do I select the parameter to be shared across the two sub-spaces?*: experiments verify that the significant gains in accuracy due to the increase in simulation densities of the sub-systems reduces the need to be particularly careful in selecting the shared parameter.
- *What about the fact that both partial systems use some* default *values to fix some of the parameters? Doesn't this negatively affect accuracy?* the gains obtained in accuracy due to the significant jump in simulation densities will overcome any disadvantages associated with fixing some of the parameters.
- *If I am joining the sub-ensembles back to the original N-parameter space, wouldn't this negatively effect the tensor decomposition cost?* If done naively, yes; and I discuss this in the next sub-section.

6.3.2 Contribution 2: Multi-Task Tensor Decomposition (M2TD)

Naively joining the sub-ensembles would map the simulations back to an N-modal tensor and this would exponentially increase the tensor decomposition time. Instead, I propose a novel **Multi-Task Tensor Decomposition** (`M2TD`) scheme, which reduces the computational complexity of high-order tensor decomposition by (a) first cheaply decomposing the low-order partial tensors and (b) intelligently stitching back the decompositions of these partial tensors to obtain the decomposition for the whole system. Intuitively, M2TD leverages partial and imperfect simulation-based knowledge from the resulting partial dynamical systems to obtain a global view of the complex process being simulated. I study alternative ways one can stitch the tensor decompositions and propose an $\texttt{M2TD} - \texttt{SELECT}$ that provides better accuracy than the alternatives.

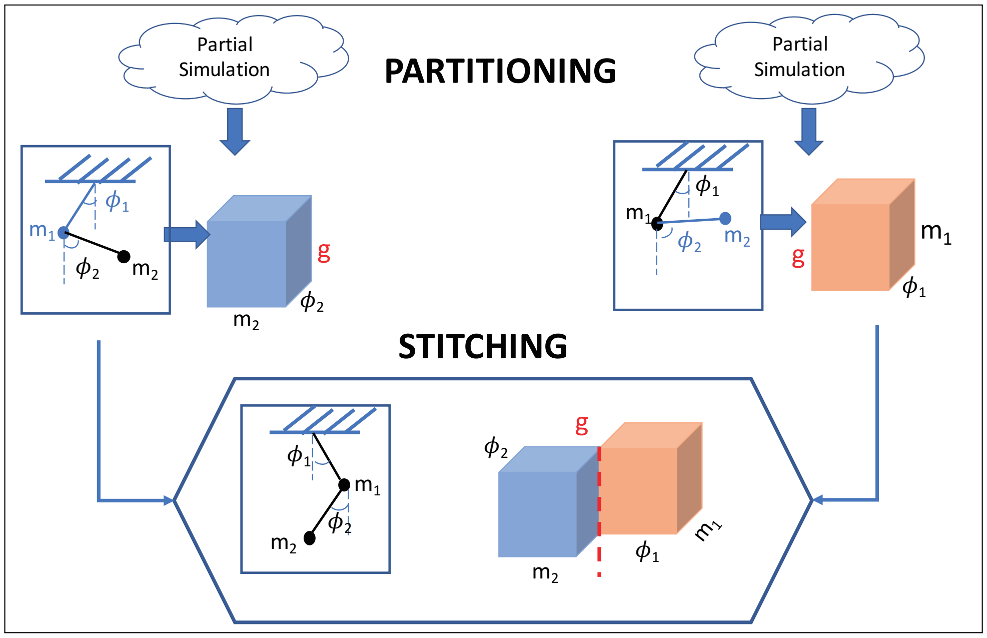


Figure 6.3: Partition-Stitch Sampling

6.4 Problem Definition

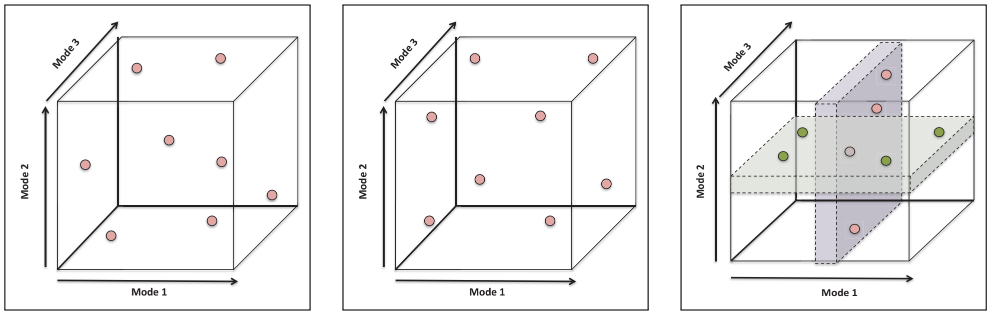
Ideally, to study the system, S, i would construct a complete tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, and given target rank values, r_1 through r_N, I would obtain its corresponding Tucker decomposition $[\boldsymbol{\mathcal{H}}, \mathbf{V^{(1)}}, \mathbf{V^{(2)}}, \mathbf{V^{(3)}}, \ldots, \mathbf{V^{(N)}}]$, where

$$\tilde{\boldsymbol{\mathcal{Y}}} = \boldsymbol{\mathcal{H}} \times_1 \mathbf{V^{(1)}} \times_2 \mathbf{V^{(2)}} \times_3 \mathbf{V^{(3)}} \ldots \times_N \mathbf{V^{(N)}} \approx \boldsymbol{\mathcal{Y}}.$$

However, this would be prohibitively costly:

- Firstly, this would require $I_1 \times \ldots \times I_N$ simulations, which can be computationally overwhelming.
- Even if this many simulations can be obtained, the analysis of the resulting tensor may be prohibitively expensive.

Instead, given a budget $B \ll I_1 \times \ldots \times I_N$ of simulations, the problem is to identify a set, $X = \{x_i = \langle \langle v_{i,1}, \ldots, v_{i,N} \rangle, S(v_{i,1}, \ldots, v_{i,N}) \rangle \parallel 1 \leq i \leq B\}$ of B simulations to execute, such



(a) Random (b) Grid (c) Slice

Figure 6.4: Conventional Solutions for Ensemble Generation

that the Tucker decomposition $[\boldsymbol{\mathcal{G}}, \mathbf{U^{(1)}}, \mathbf{U^{(2)}}, \mathbf{U^{(3)}}, \ldots \mathbf{U^{(N)}}]$ of the corresponding tensor $\boldsymbol{\mathcal{X}}$ has the following property:

$$\tilde{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{G}} \times_1 \mathbf{U^{(1)}} \times_2 \mathbf{U^{(2)}} \times_3 \mathbf{U^{(3)}} \ldots \times_N \mathbf{U^{(N)}} \approx \boldsymbol{\mathcal{X}},$$

and the Frobenius norm, $\|\boldsymbol{\mathcal{Y}} - \tilde{\boldsymbol{\mathcal{X}}}\|_F$, of the difference (from the full simulation ensemble, $\boldsymbol{\mathcal{Y}}$) is small.

6.5 Conventional Ensemble Sampling Strategies

6.5.1 Strategy #1: Random Sampling

The first approach to creating a budget constrained ensemble of simulations for the system S is to uniformly randomly sample $B \ll I_1 \times \ldots \times I_N$ parameter value configurations in the parameter space and execute those B randomly sampled simulations to obtain the ensemble, X_{rs} (Figure 6.4(a)).

6.5.2 Strategy #2: Grid Sampling

The second approach to creating a budgeted ensemble of simulations for S is to sample B parameter value configurations at positions defined by a regularly spaced grid and execute those B sampled simulations to obtain the ensemble, X_{gs} (Figure 6.4(b)).

6.5.3 Strategy #3: Slice Sampling

From Figures 6.4(a) and (b), the major difference between random sampling and grid sampling is that in grid-based ensemble construction, the subsets of the selected simulation samples are aligned on vertical and horizontal directions (or slices) of the underlying tensor and these vertical and horizontal slices cover the tensor regularly. Alternatively, these slices and the samples within each slice can be randomly selected. Intuitively, each slice fixes one of the parameters, therefore, the samples within each slice are denser (whereas the density of the overall tensor remains the same). I refer to the resulting ensemble as X_{ss}.

6.6 Partition-Stitch Sampling

The three alternatives presented in the previous section cover the underlying parameter space in different ways using the same number of simulation instances. Consequently, while the local sub-space densities may differ, the overall simulation density is identically low for all three cases.

In this section, it shows that, while executing the same number (B) of simulation instances as before, I can *increase the effective simulation density* of the ensemble by carefully partitioning the simulations to run into two groups and, then, by carefully stitching them, relying on *shared information* among these groups to *transfer* knowledge among them.

6.6.1 Key Observation

The key observation is that most complex processes can be partitioned such that, while each partition captures different sub-processes, these nevertheless relate to each other and, hence, reflect the footprints of the same underlying global pattern. Therefore, at least in theory, it should be possible to partition the given system S into two sub-systems S_1 and S_2, analyze them independently, and then *transferring* what I independently learned from the analysis of S_1 and S_2 back-and-forth, I should be able to gather information regarding the original global system, S. To leverage this observation, however, I need to answer two major questions: (a) "*How do I partition the system, S, into two sub-systems?*" and (b) "*How do I stitch the outcomes of these two sub-systems, S_1 and S_2, back to learn about S?*"

6.6.2 PF-Partitioning of a Parameter Space

It turns out that the answer to the first question is relatively straightforward: Given a system S with N input parameters, I will partition the system into two sub-systems S_1 and S_2, each with $\frac{N-k}{2} + k$ input parameters, such that

- the two systems share k of their input parameters as *pivot parameters*, and
- for each system, the remaining $\frac{N-k}{2}$ parameters will be set to a default value, referred to as *fixing constants*.

I will refer to this as the *Pivoted/Fixed (PF)-partitioning* of a parameter space. Intuitively, S_1 and S_2 correspond to two *constrained sub-spaces*: they have lesser free parameters than the original system S as each one is generated by fixing $\frac{N-k}{2}$ of the input parameters. Once the two sub-systems are obtained through PF-partitioning, I can then create two sets, X_1 and X_2, of ensembles (through random, grid, or slice sampling), each with $B/2$ simulations – these simulations are created with common values for shared pivot parameters. Consequently, the pivot parameters can be used for stitching the two ensembles together. More formally, let $\rho_1, \ldots, \rho_i, \ldots \rho_N$ denote the N input parameters of S, each with a domain with I_i distinct values. Without loss of generality, I refer to

- ρ_1 through ρ_k as the pivot parameters,

  - I select $P \leq I_1 \times \ldots \times I_k$ possible configurations for the pivot parameters for ensemble generation,

- ρ_{k+1} through $\rho_{k+(N-k)/2}$ will serve as the free input parameters of system S_1 and fixed parameters of S_2,

  - I select $E \leq I_{k+1} \times \ldots \times I_{k+(N-k)/2}$ possible configurations for the free parameters for ensemble generation for system S_1,

- $\rho_{k+(N-k)/2+1}$ through ρ_N will serve as the free input parameters of system S_2 and fixed parameters of S_1.

  - I select $E \leq I_{k+(N-k)/2+1} \times \ldots \times I_N$ possible configurations for the free parameters for ensemble generation for system S_2.

Note that, given the input budget B, I have $P \times E = B/2$. In the next sub-section, I discuss how to stitch these sub-ensembles to increase the overall effective density.

6.6.3 JE-Stitching

As I mentioned above, the goal of the stitching process is to increase the effective density of the ensemble. *Join-Ensemble (JE)-Stitching* achieves this by *joining* or *zero-joining* the two sub-systems along the shared modes:

Join-based Stitching Let $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ denote the two tensors representing the simulation ensembles, X_1 and X_2, for the two sub-systems $S_1(\rho_{1,1}, \ldots, \rho_{1,k+(N-k)/2})$ and $S_2(\rho_{2,1}, \ldots, \rho_{2,k+(N-k)/2})$, respectively. For simplicity, let the first k parameters of both sub-systems denote the set of parameters shared between the two sub-systems. I construct a new join ensemble, J, as follows: for all pairs of simulations in the two ensembles that agree on the parameter values for the k shared parameters (i.e., $(\rho_{1,1} = \rho_{2,1}) \wedge \ldots \wedge (\rho_{1,k} = \rho_{2,k})$), I compute the average of

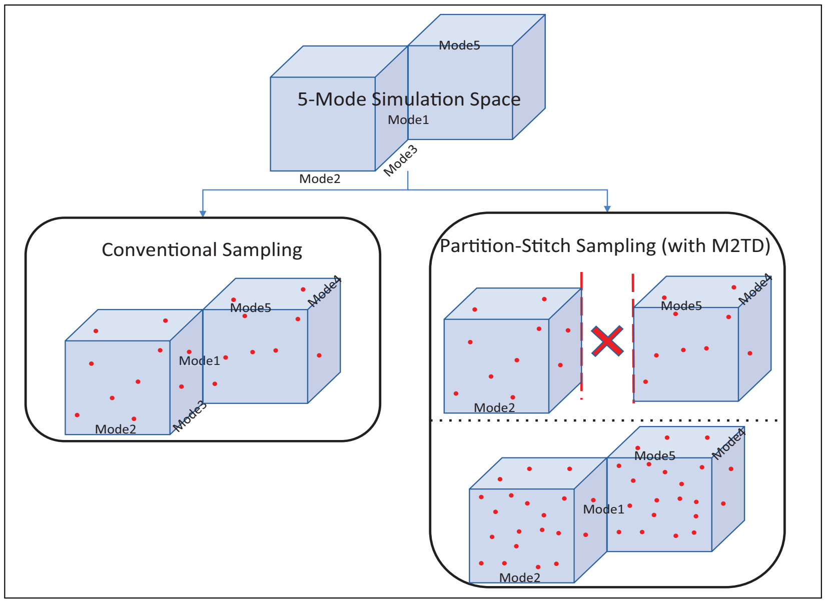


Figure 6.5: Ensemble Creation Through Pf-partitioning, Followed By Je-stitching Provides a Higher Effective Density than The Convention Sampling of the Original Parameter Space

the terms

$$x_1 = X_1(\rho_{1,1}, \ldots, \rho_{1,k}, \rho_{1,k+1}, \ldots, \rho_{1,k+(N-k)/2})$$

$$x_2 = X_2(\rho_{2,1}, \ldots, \rho_{2,k}, \rho_{2,k+1}, \ldots, \rho_{2,k+(N-k)/2})$$

and the resulting average, $\frac{x_1+x_2}{2}$, as the value for the corresponding join ensemble entry - $J(\rho_{1,1}, \ldots, \rho_{1,k+(N-k)/2}, \rho_{2,k+1}, \ldots, \rho_{2,k+(N-k)/2})$.

Note that, since for each one of the P unique combinations selected for the shared pivot parameters, there are E ensemble simulations in both sub-systems, the resulting join ensemble tensor, $\boldsymbol{\mathcal{J}}$, represents $P \times E^2$ joined simulations – since, in the previous section, I have $P \times E = B/2$, this gives us $B^2/(4P)$ simulation entries, (and assuming that $B \gg (4P)$) *effectively squaring the simulation density* (Figure 6.5). As experiment verified, (due to this increase effective density) the decomposition of $\boldsymbol{\mathcal{J}}$ will be a far better approximation for the original system S then the decomposition of the tensor $\boldsymbol{\mathcal{X}}$ which represents the original set of simulations, $X = X_1 \cup X_2$. In fact, the accuracy gains associated with this density increase

- prevents any disadvantages associated with eliminating some of the free parameters, and
- leads to significant overall accuracy gains, even without precise *a priori* knowledge about parameters to use as pivot and/or values for *fixing constants*.

Zero-Join based Stitching Note, however, that when E (i.e., sub-system densities) is small, the overall join ensemble density may still be too low to provide accurate analysis. In such a case, I can further boost the overall ensemble density by using *zero-join* (as opposed to simple join) to stitch the sub-ensembles: when constructing the join ensemble, J, for all pairs of simulations in the two sub-ensembles that agree on the parameter values for the k shared parameters (i.e., $(\rho_{1,1} = \rho_{2,1}) \wedge \ldots \wedge (\rho_{1,k} = \rho_{2,k})$), I still compute the average of the terms as described above. But, in this case, if there is a simulation instance,

$$x_1 = X_1(\rho_{1,1}, \ldots, \rho_{1,k}, \rho_{1,k+1}, \ldots, \rho_{1,k+(N-k)/2})$$

but the simulation instance

$$X_2(\rho_{1,1}, \ldots, \rho_{1,k}, \rho_{2,k+1}, \ldots, \rho_{2,k+(N-k)/2})$$

does not exist; then I treat the *missing* simulation instance as if it exists with 0 value, and I construct the corresponding join ensemble entry $J(\rho_{1,1}, \ldots, \rho_{1,k+(N-k)/2}, \rho_{2,k+1}, \ldots, \rho_{2,k+(N-k)/2})$ with value $\frac{x_1+0}{2}$. I similarly handle simulation instances in X_2.

Note that zero-joining increases the effective density of the simulation ensemble to $2 \times (P \times E^2) \times E^2$, and as experiments verified, it significantly boosts accuracy in cases where sub-ensemble simulation densities are too low to for basic join-based stitching be effective.

6.7 Multi-Task Tensor Decomposition (M2TD)

The difficulty with JE-stitching, of course, is that tensor $\boldsymbol{\mathcal{J}}$ has almost double the number of modes as the tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$. Consequently, its decomposition is likely to be

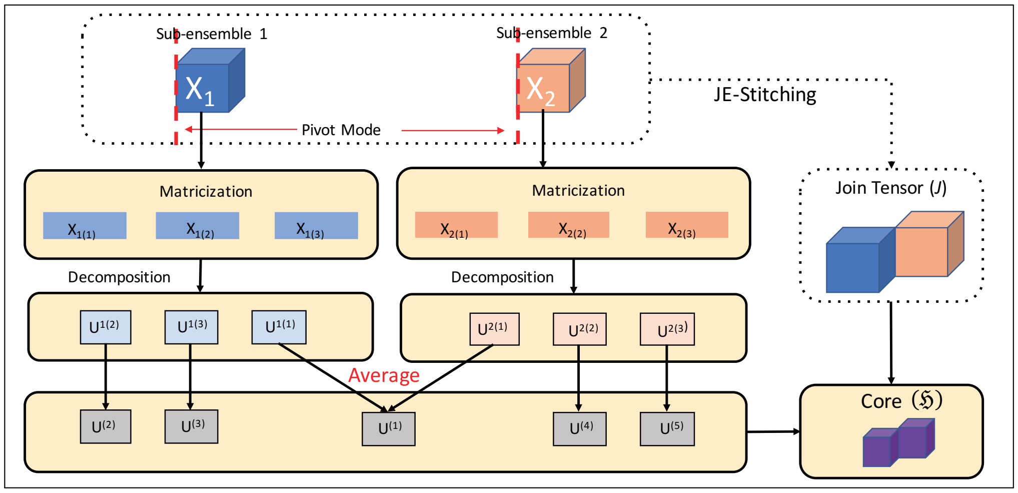


Figure 6.6: Overview of M2TD-AVG

significantly more expensive than the decomposition of these two pre-join tensors. What remains to be shown is that I can, in fact, obtain the decomposition of $\mathcal{J}$ directly from the decompositions of $\mathcal{X}_1$ and $\mathcal{X}_2$. I discuss this in this section.

Let $\mathcal{X}_1$ and $\mathcal{X}_2$ be two sub-ensemble tensors corresponding to sub-systems constructed through PF-partitioning of an N-parameter system, S. Let J be the join ensemble and $\mathcal{J}$ be the corresponding join tensor one could obtain through JE-stitching. In this section, I introduce three alternative *multi-task tensor decomposition* (M2TD) schemes to obtain the decomposition of $\mathcal{J}$ from the decompositions of $\mathcal{X}_1$ and $\mathcal{X}_2$.

6.7.1 M2TD-Average (M2TD-AVG)

Remember from the earlier sections that both $\mathcal{X}_1$ and $\mathcal{X}_2$ are M-model tensors, where $M = k + (N - k)/2$, and that the first k modes are shared. I modify the HOSVD algorithm, to obtain the proposed M2TD-AVG algorithm (Algorithm 8). Intuitively, M2TD-AVG takes the first k factor matrix pairs, $(U^{1(n)}, U^{2(n)})$, corresponding to the shared pivot tensors of the independently decomposed tensors, $\mathcal{X}_1$ and $\mathcal{X}_2$, and averages each pair to obtain a *common* factor matrix representing both tensors: since factor matrices, $U^{1(n)}$ and $U^{2(n)}$, both map

Algorithm 8 M2TD-AVG

```
Input: Tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$, Rank for each mode $r_1, r_2, ..., r_N$
Output: Decomposed factors $\boldsymbol{U}^{(1)}$, $\boldsymbol{U}^{(2)}$, ..., $\boldsymbol{U}^{(N)}$ and core tensor $\boldsymbol{\mathcal{G}}$ for the join tensor $\boldsymbol{\mathcal{J}}$
for $m = 1, ..., M$ do
    matricize $\boldsymbol{\mathcal{X}}_1$ into matrix $X_{1(m)}$
    matricize $\boldsymbol{\mathcal{X}}_2$ into matrix $X_{2(m)}$
end
for $n = 1, ..., k$ do
    $U^{1(n)} \leftarrow r_n$ leading left singular vectors of $X_{1(n)}$
    $U^{2(n)} \leftarrow r_n$ leading left singular vectors of $X_{2(n)}$
    $U^{(n)} \leftarrow average(U^{1(n)}, U^{2(n)})$
end
for $n = k+1, ..., M$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{1(n)}$
end
for $n = M+1, ..., 2M-k$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{2(n-M+k)}$
end
$\boldsymbol{\mathcal{J}}$ = join_tensor($\boldsymbol{\mathcal{X}}_1$,$\boldsymbol{\mathcal{X}}_2$)
$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{J}} \times_1 U^{(1)T} \times_2 U^{(2)T}, ..., \times_N U^{(N)T}$
return $\boldsymbol{\mathcal{G}}$, $U^{(1)}$, $U^{(2)}$,..., $U^{(N)}$
```

the domain of the corresponding factor to a vector space represented by r_n singular factors (sorted in decreasing order of significance), the operation $average(U^{1(n)}, U^{2(n)})$ essentially constructs a new vector space, where each element of the domain is represented by the average vector from the two input vector spaces (Figure 6.9(a)). Remaining factor matrices are then combined to obtain the core tensor, $\boldsymbol{\mathcal{G}}$ (see Figure 6.6). As experiment verify, this leads to a better approximation of the original system than any of the naive ensemble

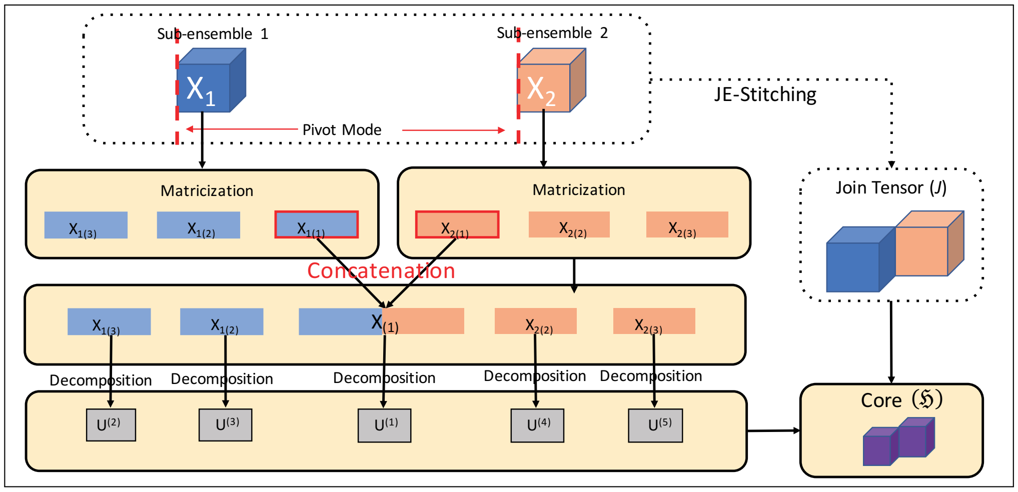

Sub-ensemble 1
Sub-ensemble 2
X_1
X_2
Pivot Mode
JE-Stitching
Matricization
$X_{1(3)}$ $X_{1(2)}$ $X_{1(1)}$
Matricization
$X_{2(1)}$ $X_{2(2)}$ $X_{2(3)}$
Join Tensor (J)
Concatenation
$X_{1(3)}$ $X_{1(2)}$ $X_{(1)}$ $X_{2(2)}$ $X_{2(3)}$
Decomposition
$U^{(2)}$ $U^{(3)}$ $U^{(1)}$ $U^{(4)}$ $U^{(5)}$
Core ($\mathfrak{H}$)

Figure 6.7: Overview of M2TD-CONCAT

sampling schemes.

6.7.2 *M2TD-Concatenate (M2TD-CONCAT)*

M2TD-AVG, presented in the previous section, recovers the factor matrices for pivot parameters (modes) by averaging the corresponding factor matrices; i.e., by first obtaining the singular vectors of the matricizations and then averaging these singular vectors. However, there is nothing that guarantees that these averages will act as singular vectors themselves.

Instead, the alternative M2TD-CONCAT algorithm (detailed in Algorithm 9 and visualized in Figure 6.7) avoids this potential issue by first constructing a concatenated matricization for each pivot mode pair and then seeking the left singular vectors of this combined matricization. Intuitively, M2TD-CONCAT maps the matricizations along the shared/pivot modes back into the higher-modal space and seeks the singular vectors that best represent this higher modal space.

Algorithm 9 M2TD-CONCAT

```
Input: Tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$, Rank for each mode $r_1, r_2, ..., r_N$
Output: Decomposed factors $\boldsymbol{U}^{(1)}$, $\boldsymbol{U}^{(2)}$, ..., $\boldsymbol{U}^{(N)}$ and core tensor $\boldsymbol{\mathcal{G}}$ for the join tensor $\boldsymbol{\mathcal{J}}$
for $n = 1, ..., k$ do
    matricize $\boldsymbol{\mathcal{X}}_1$ into matrix $X_{1(n)}$
    matricize $\boldsymbol{\mathcal{X}}_2$ into matrix $X_{2(n)}$
    $X_{(n)} \leftarrow concatenate(X_{1(n)}, X_{2(n)})$
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{(n)}$
end
for $m = k+1, ..., M$ do
    matricize $\boldsymbol{\mathcal{X}}_1$ into matrix $X_{1(m)}$
    matricize $\boldsymbol{\mathcal{X}}_2$ into matrix $X_{2(m)}$
end
for $n = k+1, ..., M$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{1(n)}$
end
for $n = M+1, ..., 2M-k$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{2(n-M+k)}$
end
$\boldsymbol{\mathcal{J}}$ = join_tensor($\boldsymbol{\mathcal{X}}_1$,$\boldsymbol{\mathcal{X}}_2$)
$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{J}} \times_1 U^{(1)T} \times_2 U^{(2)T}, ..., \times_N U^{(N)T}$
return $\boldsymbol{\mathcal{G}}$, $U^{(1)}$, $U^{(2)}$,..., $U^{(N)}$
```

Algorithm 10 M2TD-SELECT

```
Input: Tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$, Rank for each mode $r_1, r_2, ..., r_N$
Output: Decomposed factors $\boldsymbol{U}^{(1)}$, $\boldsymbol{U}^{(2)}$, ..., $\boldsymbol{U}^{(N)}$ and core tensor $\boldsymbol{\mathcal{G}}$ for the join tensor $\boldsymbol{\mathcal{J}}$
for $m = 1, ..., M$ do
    matricize $\boldsymbol{\mathcal{X}}_1$ into matrix $X_{1(m)}$
    matricize $\boldsymbol{\mathcal{X}}_2$ into matrix $X_{2(m)}$
end
for $n = 1, ..., k$ do
    $U^{1(n)} \leftarrow r_n$ leading left singular vectors of $X_{1(n)}$
    $U^{2(n)} \leftarrow r_n$ leading left singular vectors of $X_{2(n)}$
    $U^{(n)} \leftarrow row\_select(U^{1(n)}, U^{2(n)})$
end
for $n = k+1, ..., M$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{1(n)}$
end
for $n = M+1, ..., 2M-k$ do
    $U^{(n)} \leftarrow r_n$ leading left singular vectors of $X_{2(n-M+k)}$
end
$\boldsymbol{\mathcal{J}}$ = join_tensor($\boldsymbol{\mathcal{X}}_1$,$\boldsymbol{\mathcal{X}}_2$)
$\boldsymbol{\mathcal{G}} = \boldsymbol{\mathcal{J}} \times_1 U^{(1)T} \times_2 U^{(2)T}, ..., \times_N U^{(N)T}$
return $\boldsymbol{\mathcal{G}}$, $U^{(1)}$, $U^{(2)}$,..., $U^{(N)}$
```

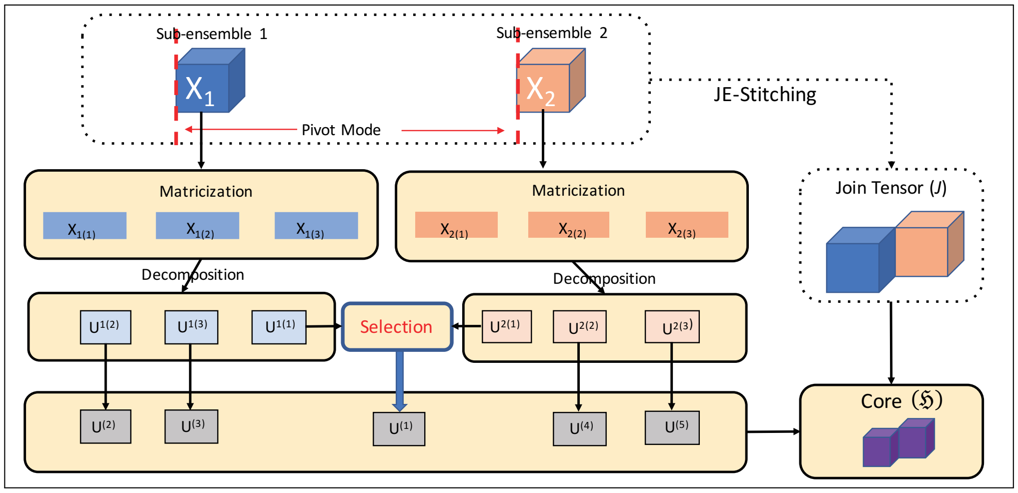


Figure 6.8: Overview of M2TD-SELECT

6.7.3 M2TD-Selection (M2TD-SELECT)

The M2TD-CONCAT algorithm presented above tries to improve the vector averaging scheme of M2TD-AVG through row-by-row concatenation of the pivot matricizations before the corresponding factor matrices are computed. In this section, I note that there is an alternative, and potentially more effective, way to improve the M2TD-AVG scheme: once the factor matrices for the pivots are obtained, instead of averaging them, I can carefully select between the individual rows of the corresponding factor matrices and use these selected rows to construct more effective combined factor matrices.

The pseudocode for the process is shown in Algorithm 10 and visualized in Figure 6.8. Note that the major difference between this algorithm and M2TD-AVG is the line

$$U^{(n)} \leftarrow row\_select(U^{1(n)}, U^{2(n)}),$$

where the factor matrix $U^{(n)}$ is constructed by selecting the appropriate rows from $U^{1(n)}$ or $U^{2(n)}$, instead of simply averaging them. This row selection process is further detailed in Algorithm 11 and visualized in Figure 6.9(b).The key idea is to consider the energies (captured by the 2-norm function) of each row, i, in U_1 and U_2, and identify which of the two

Algorithm 11 ROW_SELECT

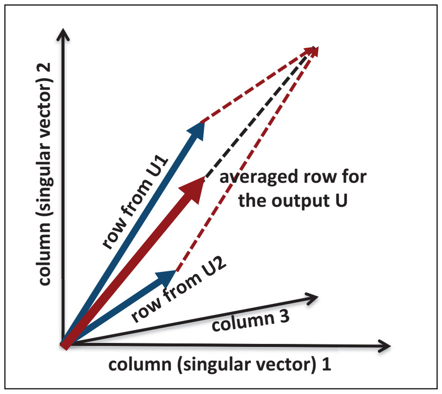
```
Input: Factor matrices $U_1$ and $U_2$
Output: Row-selected Factor Matrix $U$
$I \leftarrow num\_rows(U_1)$
for $1 \leq i \leq I$ do
    if $\|row(U_1, i)\|_2 \geq \|row(U_2, i)\|_2$ then
        $row(U, i) \leftarrow row(U_1, i)$
    else
        $row(U, i) \leftarrow row(U_2, i)$
    end
end
return $U$
```

factor matrices provides a higher energy for that particular row. Intuitively, this enables us to identify which of the two factor matrices better represent the entity corresponding to row, i, and, given this information, I can construct the row i of the output factor matrix, U, by selecting the corresponding row from the factor matrix, U_1 or U_2, with a higher representation power for that entity.

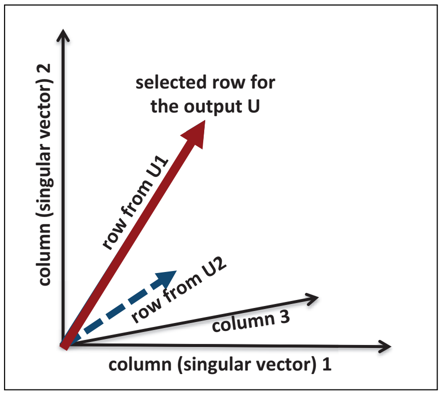
As experiment verified, this selection strategy prevents the row with the lesser energy to act as *noise* on the description of the corresponding entity and, thus, leads to significantly higher decomposition accuracies. Moreover, as the experiments show, the accuracy gains gets higher as I target higher ranking decompositions that maintain more details by seeking a larger number of patterns in the data.

6.7.4 Distributed M2TD (D-M2TD)

A major challenge with tensor decomposition is its computational and space complexity. This is especially true for the Tucker decomposition with a dense core. In this section, relying on several key properties of the M2TD algorithm, I propose a 3-phase distributed version of


column (singular vector) 2
row from U1
averaged row for the output U
row from U2
column 3
column (singular vector) 1

(a) M2TD-AVG


column (singular vector) 2
selected row for the output U
row from U1
row from U2
column 3
column (singular vector) 1

(b) M2TD-SELECT

Figure 6.9: Comparison of the Row Construction Processes Between M2TD-AVG and M2TD-SELECT

M2TD that can be efficiently and scalably executed on MapReduce or Spark based platforms (see Algorithm 12):

• **Phase 1: Parallel Sub-Tensor Decomposition:** Consider the M2TD-SELECT pseudocode in Algorithm 10. Here, $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ are two sub-tensors corresponding to two sub-systems constructed through PF-partitioning. These low-order sub-tensors can be decomposed (in parallel) independently from each other. Therefore, this phase can be parallelized using, for example, the popular distributed computing framework, MapReduce, using the following `map` and `reduce` operators:

- **map**: $\langle \boldsymbol{\kappa}, \rho_1, \rho_2, \ldots, \rho_M, \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M) \rangle$ on $\boldsymbol{\kappa}$. Here, $\boldsymbol{\kappa}$ is the low-order tensor id; i.e., $\boldsymbol{\kappa} \in \{1, 2\}$. $\rho_1, \rho_2, \ldots, \rho_M$ together give the coordinate of a cell in the low-order tensor $\boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}$. Key-value pairs with the same κ are shuffled to the same reducer in the form of $\langle key : \boldsymbol{\kappa}, val : \rho_1, \rho_2, \ldots, \rho_M, \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M) \rangle$.

- **reduce**: $\langle key{:}\boldsymbol{\kappa},\ val{:}\rho_1, \rho_2, \ldots, \rho_M,\ \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M) \rangle$. The reducer processing the key, $\boldsymbol{\kappa}$, receives the non-zero elements of sub-tensor $\boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}$ and decomposes it into sub-factor $\boldsymbol{U}^{\boldsymbol{\kappa}(\boldsymbol{n})}$, where $\boldsymbol{n}$ is the mode id, by using SVD. Finally, reducer appropriately

Algorithm 12 The outline of the Distributed Multi-Task Tensor Decomposition, $\mathtt{D-M2TD}$, process

```
Input: Tensor $\boldsymbol{\mathcal{X}}_1$, $\boldsymbol{\mathcal{X}}_2$, Rank for each mode $r_1, r_2, \ldots, r_N$
Output: Factor Matrices $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$, ..., $\mathbf{U}^{(N)}$ and core tensor $\boldsymbol{\mathcal{G}}$
for the join tensor $\boldsymbol{\mathcal{J}}$

  1. Phase 1: Parallel decomposition of $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ to generate $U^{1(n)}, U^{2(n)}, n \in \{1, \ldots, N\}$
  2. Phase 2: Parallel JE-Stitching $\boldsymbol{\mathcal{X}}_1$, $\boldsymbol{\mathcal{X}}_2$ to obtain the decomposition of the joined tensor $\boldsymbol{\mathcal{J}}$
  3. Phase 3: for $1 \leq n \leq N$
      (a) Parallel tensor matrix mutiplication- $\boldsymbol{\mathcal{G}}_n = \boldsymbol{\mathcal{J}} \times_n \mathbf{U}^{(n)}$
  4. Return Factor Matrices $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$, ..., $\mathbf{U}^{(N)}$ and core $\boldsymbol{\mathcal{G}}$
```

relabels each $\boldsymbol{U^{\kappa(n)}}$ as $\boldsymbol{U^{(n)}}$ and emits each sub-factor as an independent file, with content $\langle$ *key* : n, *value* : $i, j, \boldsymbol{U^{(n)}}(i,j)\rangle$. Here, i, j are the coordinates of sub-factor $\boldsymbol{U^{(n)}}$.

Note that this step can be further parallelized by leveraging parallel Tucker decomposition techniques, such as [49, 26].

• **Phase 2: Parallel JE-Stitching to Obtain Join Tensor, $\boldsymbol{\mathcal{J}}$:** The goal of the stitching process is to increase the effective density of the ensemble. JE-stitching achieves this by joining the two sub-systems along their shared pivot modes to obtain the $\boldsymbol{\mathcal{J}}$ tensor. This process can be parallelized as follows:

- **map**: $\langle \boldsymbol{\kappa}, \rho_1, \rho_2, \ldots, \rho_M, \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M)\rangle$. Key-value pairs with the same pivot mode index $(\rho_1, \rho_2, \ldots, \rho_k)$are shuffled to the same reducer in the form of $\langle$*key*:$(\rho_1, \rho_2, \ldots, \rho_k)$, *val*:$\rho_1, \rho_2, \ldots, \rho_M, \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M)\rangle$.

- **reduce:** $\langle$*key*:$(\rho_1, \rho_2, \ldots, \rho_k)$, *val*:$\rho_1, \rho_2, \ldots, \rho_M, \boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}(\rho_1, \rho_2, \ldots, \rho_M)\rangle$. The join ensemble $\boldsymbol{\mathcal{J}}(\rho_1, \rho_2, \ldots, \rho_k, \ldots)$ is constructed for all pairs of $\boldsymbol{\mathcal{X}}_{\boldsymbol{\kappa}}$, that agree on the parameter values for the k pivot parameters.

.

• **Phase 3: Parallel Tensor-Matrix Multiplication to Recover the Core Tensor:** The costliest part of the M2TD algorithm is the final step where the join tensor $\boldsymbol{\mathcal{J}}$ is multiplied by the transposes of the factor matrices to recover the dense core tensor. I parallelize this as follows:

- **map**: $\langle \rho_1, \rho_2, \ldots, \rho_N, \boldsymbol{\mathcal{J}}(\rho_1, \rho_2, \ldots, \rho_N)\rangle$, $\langle \boldsymbol{n}, i, j, \boldsymbol{U}^{(n)}(i,j)\rangle$. Cells of $\boldsymbol{\mathcal{J}}$ (from Phase 2) with index $(\rho_1, \rho_2, \ldots, \rho_{n-1}, \rho_{n+1}, \ldots, \rho_N)$ are shuffled to the same reducer in the form of $\langle$*key*:$(\rho_1, \rho_2, \ldots, \rho_{n-1}, \rho_{n+1}, \ldots, \rho_N)$, *val*:$\boldsymbol{\mathcal{J}}(\rho_1, \rho_2, \ldots, \rho_N)\rangle$
- **map**: $\langle \boldsymbol{n}, i, j, \boldsymbol{U}^{(n)}(i,j)\rangle$. Outputs of Phase 1 $\langle \boldsymbol{n}, i, j, \boldsymbol{U}^{(n)}(i,j)\rangle$ are shuffled to the same reducer based on mode id n in the form of $\langle$*key*:$(\rho_1, \rho_2, \ldots, \rho_{n-1}, \rho_{n+1}, \ldots, \rho_N)$, *val*:$\boldsymbol{n}, i, j, \boldsymbol{U}^{(n)}(i,j)\rangle$
- **reduce:** The reducer takes

  $$\langle key : (\rho_1, .., \rho_{n-1}, \rho_{n+1}, .., \rho_N), val : \boldsymbol{\mathcal{J}}(\rho_1, , \ldots, \rho_N)\rangle$$

  and

  $$\langle key : (\rho_1, .., \rho_{n-1}, \rho_{n+1}, .., \rho_N), val : \boldsymbol{n}, i, j, \boldsymbol{U}^{(n)}(i,j)\rangle$$

  and performs vector-matrix multiplication to emit
  $\langle(\rho_1, \rho_2, \ldots, \rho_{n-1}, j, \rho_{n+1}, \ldots, \rho_N), \sum_{\rho_n=1}^{I_n} \boldsymbol{\mathcal{J}}(\rho_1, , \ldots, \rho_n, \ldots, \rho_N) * \boldsymbol{U}^{(n)}(\rho_n, j)\rangle$.

6.8 Experiments

In this section, I report results of the experiments that aim to assess the effectiveness and efficiency of the proposed partition-stitch ensemble sampling strategy and the novel *multi-task tensor decomposition (M2TD)* scheme. For these experiments, I used the Chameleon

	Alternative values		
Dynamic systems	**Double Pend.**;	Triple Pend.	Lorenz System
Parameter resolution	60 ;	**70**;	80
Size of the corresponding simulation space (S)	$60^5(8 \times 10^8)$;	$70^5(2 \times 10^9)$;	$80^5(3 \times 10^9)$
Pivot density (P)	10%;	**100%**	
Sub-system density (E)	10%;	**100%**	
Ensemble budget	4×10^4,	7×10^4,	1×10^5,
($B = 2 \times P \times E \times S$)	4×10^5,	7×10^5,	1×10^6
Target decomposition rank (r)	5;	**10**;	20
Stitching technique	**Join**;	Zero-Join	
Number of servers	2, 6, 10, 14, **18**		

Table 6.1: Experiment Setup – Default Values, Used Unless Otherwise Specified, Are Highlighted

cloud platform [29]: I deployed all algorithms on 18 xxlarge instances, with 8-core vCPU, 32GB memory, 160GB disk space. Distributed versions were implemented in Java 8, over Hadoop 2.7.3. The key system parameters and their value ranges are reported in Table 6.1 and explained below.

6.9 Dynamic Systems

In these experiments, I consider three dynamic processes: *double pendulum*, *triple pendulum*, *lorenz system* [7]. The code for these systems was obtained from [38]. These dynamic processes are selected for their varying complexities:

The ***double pendulum*** system has four parameters: initial angle, ϕ_1, and weight, $m1$, of the first pendulum as well as the initial angle, ϕ_2, and weight, $m2$, of the second pendulum.

The ***triple pendulum (with variable friction)*** system is similar, but more complex due to the addition of a third pendulum. Moreover, the system has a different set of initial parameters: the angle ϕ_1 of the first pendulum, the initial angle ϕ_2 of the second pendulum, the initial angle ϕ_3 of the third pendulum, and the friction f of whole system. Intuitively, unlike the double pendulum system, in the triple pendulum system the friction is considered as a simulation parameter.

The ***Lorenz system*** is notable for having chaotic solutions for certain initial conditions [7]. The system has four variable parameters: the coordinate of the initial position, z, and three other system parameters, σ, β, ρ.

6.10 Simulation Ensembles

For the above systems, I construct 5-mode simulation ensembles. Each cell of the 5-mode ensemble simulation tensor encodes the Euclidean distance between the states of the resulting simulated system and the observed system parameters at a given time stamp, for a given quadruple of simulation parameters. Intuitively, each cell encodes the relationship between a given simulation instance to a configuration observed in the real-world.

In Table 6.1, in the experiments, the size of the simulation space varied between $60^5 \sim 8 \times 10^8$ to $80^5 \sim 3 \times 10^9$ simulation instances. In contrast, the simulation instance budgets were on the order of 10^4 to 10^5, indicating that, despite the large number of simulations included in the ensembles, the resulting ensemble tensors were very sparse (densities on the order of $\sim 10^{-4}$). Despite this sparsity, for the different configurations considered in Table 6.1, the simulation ensemble required from 25GB to 105GB data storage.

6.11 Alternative Ensemble Construction Schemes

In this section, I evaluated the M2TD-AVG, -CONCAT, and -SELECT strategies and compared them against the conventional (RANDOM, GRID, and SLICE) ensemble sampling approaches (Section 6.5). For M2TD-based schemes, I considered the case with a single pivot parameter and, to analyze worst case behavior, I sampled the sub-systems randomly.

6.12 Evaluation Criteria

I compared accuracy and efficiency of alternative schemes, for different target decomposition ranks, different parameter space resolutions, and simulation budgets (see Table 6.1). To measure accuracy, I use the Frobenius norm of the difference tensor:

$$accuracy(\tilde{\boldsymbol{\mathcal{X}}}, \boldsymbol{\mathcal{Y}}) = 1 - \left(\frac{\|\tilde{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{Y}}\|}{\|\boldsymbol{\mathcal{Y}}\|_F} \right),$$

where $\tilde{\boldsymbol{\mathcal{X}}}$ is the reconstructed tensor (after sampling and decomposition), while $\boldsymbol{\mathcal{Y}}$ is the tensor corresponding to the full simulation space. I also report the decomposition times.

6.13 Discussions of the Results

6.13.1 General Overview

Table 6.2 focuses on the double pendulum system and compares accuracies and decomposition times for various approaches considered in this section for the different target ranks and for different parameter resolutions. In the table, the M2TD-based algorithms provide *several orders better accuracy* than the conventional approaches, with the same number of simulation instances. As expected, among the conventional schemes, the Random strategy provides the worst and the Grid strategy provides the best accuracy; however, even Grid is $\sim 1000\times$ worse than the proposed M2TD-SELECT algorithm. As also expected, among the M2TD-based algorithms, M2TD-SELECT provides the best overall accuracy: moreover, the relative performance gains of M2TD-SELECT algorithm further increases for larger decomposition ranks, indicating that as I seek more detailed patterns in the ensemble, M2TD-SELECT better captures these underlying patterns in the data.

In the Table, M2TD-based algorithms are somewhat more expensive than the conventional sampling strategies; but the gains in accuracy are several orders higher than the decomposition time overheads of M2TD-based techniques. This is because, as highlighted in Section 6.6.3, the proposed partition-stitch technique increases the *effective density* of the join ensemble. Consequently, the increase in the decomposition is *well amortized* by the increase in the effective simulation density. In these experiments, each double pendulum simulation took roughly 0.66ms. Given this, obtaining an ensemble simulation with density $70^4 (= 70^2 \times 70^2)$ would require roughly 16000 seconds (ignoring the additional time to decompose). In contrast, the proposed M2TD based techniques are able to achieve the same *effective density* by running only 2×70^2 simulations in just 46 seconds and obtain the ensemble decomposition in an additional $\sim$ 1600 seconds. This points to the impressive performance gains provided by the proposed multi-task tensor decomposition (M2TD) technique.

One question that remains is whether I could have joined the sub-ensembles directly into tensor $\mathcal{J}$ to decompose instead of relying on the M2TD techniques: the answer to this question is a *strong no*: for the experiments reported in Table 6.2, with the configuration of 18 xxlarge servers, direct decomposition of the resulting dense tensor was not feasible due to memory limitations.

6.13.2 Decomposition Time Distribution

Table 6.3 presents how the decomposition time is split among the three phases of the map-reduce process described in Section 6.7.4. The table also shows how the execution time varies as I change the number of servers allocated for the decomposition process. In this table, as expected, the third phase where I recover the core tensor of the decomposition is the costliest step of the process. Allocating more servers indeed helps bring the cost of this phase down; however, there are diminishing returns due to data communication overheads.

Accuracy for Double Pendulum System							
Res.	Rank	M2TD			Random	Grid	Slice
		AVG	CONCAT	SELECT			
60	5	0.49	0.49	**0.54**	1E-8	3E-4	2E-4
	10	0.50	0.50	**0.62**	2E-7	3E-4	2E-4
	20	0.52	0.53	**0.56**	5E-6	3E-4	2E-4
70	5	0.46	0.46	**0.51**	7E-9	2E-4	2E-4
	10	0.47	0.48	**0.57**	9E-8	2E-4	2E-4
	20	0.49	0.50	**0.73**	2E-6	2E-4	2E-4
80	50	0.46	0.46	**0.50**	4E-9	1E-4	1E-4
	10	0.47	0.47	**0.49**	4E-8	1E-4	1E-4
	20	0.48	0.49	**0.59**	1E-6	2E-4	1E-4

(a) Accuracy

Decomposition Time for Double Pendulum System (sec.)							
Res.	Rank	M2TD			Random	Grid	Slice
		AVG	CONCAT	SELECT			
60	5	808	797	785	203	144	167
	10	808	819	849	234	148	186
	20	1034	929	935	348	456	258
70	5	1508	1581	1594	312	209	193
	10	1696	1645	1576	379	201	244
	20	1866	1914	1995	575	744	381
80	5	3990	3591	4907	414	227	336
	10	5232	5979	6068	514	239	410
	20	5341	5707	5439	860	883	606

(b) Time (sec.)

Table 6.2: Results for Double Pendulum System (Pivot=t, $p = 100\%$, $e = 100\%$)

Decomposition Time using Different Numbers of Servers (sec.)						
Num. Servers	M2TD-SELECT			Random	Grid	Slice
	Phase 1	Phase 2	Phase 3			
2	52	817	4167	670	420	488
6	62	383	1802	464	275	318
10	61	371	1318	415	237	280
14	65	354	1279	381	214	253
18	67	363	1118	379	201	244

Table 6.3: Different Number of Servers (Double Pendulum, Resolution=70, Rank = 10, Pivot=t, $p = 100\%$, $e = 100\%$)

6.13.3 Varying Data Sets

In Table 6.4, I study the accuracy and decomposition time results for different dynamic systems. As seen here, also for the triple pendulum and Lorenz systems, I observe the very same pattern: M2TD-SELECT provides the best accuracy among all alternatives, providing several orders of magnitude gain in accuracy relative to the conventional schemes.

6.13.4 Varying Budgets and Zero-Joins

In the default experiments considered above, the budget was selected such that the sub-ensembles would have a perfect density of 1.0. In the first row of Table 6.5, I reduced the ensemble budget by taking $1/10^{th}$ of the samples I considered in the previous examples. Naturally, this results in a drop in accuracy for all approaches. However, M2TD-based schemes remain several orders better than the conventional approaches.

The table also shows that when the budgets (thus sub-ensemble densities) are low, I can boost the overall accuracy by leveraging zero-joins (introduced in Section 6.6.3), rather than

Accuracy for Different Systems						
Dyn.System	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
D.P.	0.47	0.48	**0.57**	9E-8	2E-4	2E-4
T.P.	0.25	0.25	**0.31**	6E-8	2E-4	1E-4
L.S.	0.31	0.32	**0.36**	4E-8	2E-4	1E-4

(a) Accuracy

Decomposition Time for Different Systems (sec.)						
Dyn.System	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
D.P.	1696	1645	1576	379	201	244
T.P.	992	1422	1106	221	180	166
L.S.	1728	1850	1705	444	230	211

(b) Time (sec.)

Table 6.4: Results for Different Dynamical Systems (Resolution=70, Rank = 10, Pivot=t, $p = 100\%$, $e = 100\%$)

using simple joins when implementing JE-stitching.

6.13.5 Varying Pivot/Sub-Ensemble Densities

Tables 6.6 and 6.7 show the impact of reduced pivot and sub-ensemble densities (i.e., P and E) respectively. As seen here, the overall pattern is as before: reduction in the simulation budget reduces the overall accuracy; however, M2TD-based schemes provide significantly higher accuracy overall.

An interesting observation, however, is that (while the total number of simulations is the same) reduction in the pivot sub-ensemble density has a significantly higher impact than the

Accuracy for Different Ensemble Budgets (B)						
Budget	M2TD			Random	Grid	Slice
	AVG	CONCAT	SEL.			
4×10^4 (join)	3.5E-5	3.4E-5	**4.1E-5**	9E-9	2E-5	2E-6
4×10^4 (zero-join)	3.3E-3	3.2E-3	**3.9E-3**	9E-9	2E-5	2E-6
4×10^5	0.47	0.48	**0.57**	9E-8	2E-4	2E-4

(a) Accuracy

Decomposition Time for Different Ensemble Budgets (B) (sec.)						
Budget	M2TD			Random	Grid	Slice
	AVG	CONCAT	SEL.			
4×10^4 (join)	200	201	200	190	175	183
4×10^4 (zero-join)	596	598	592	190	175	183
4×10^5	1696	1645	1576	379	201	244

(b) Time (sec.)

Table 6.5: Results for Different Ensemble Budgets (Double Pendulum, Resolution=70, Rank = 10, Pivot=t; Note That $b = 4 \times 10^5$ Corresponds to the Case Where Both Pivot, p, and Sub-systems, e, Have 100% Densities)

reduction in the pivot density: this is because, as discussed in Section 6.6.3, the effective density of a stitched simulation ensemble is proportional to $P \times E^2$, and thus reductions in E have a more significant impact than reductions in P: this further confirms our initial hypothesis that maintaining sub-ensemble densities high is important for accurate characterization of the system being studied.

Accuracy for Different Pivot Densities (P)						
P. Density	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
10%	3.5E-2	7.6E-3	**3.6E-2**	9E-9	2E-5	2E-6
100%	0.47	0.48	**0.57**	9E-8	2E-4	2E-4

(a) Accuracy

Decomposition Time for Different Pivot Densities (P) (sec.)						
P.Density	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
10%	606	597	607	190	175	183
100%	1696	1645	1576	379	201	244

(b) Time (sec.)

Table 6.6: Results for Different Pivot Densities (Double Pendulum, Resolution=70, Rank = 10, Pivot=t, $e = 100\%$)

6.13.6 Selection of the Pivot Parameter

In Table 6.8, I vary the pivot parameter[1]: as expected, which parameter is selected as the pivot has some impact on the accuracy of the proposed partition-stitch scheme. However, whichever pivot is selected, the overall accuracy is several orders of magnitude better than that of conventional schemes, indicating that I do not need very precise information about the system being studied to decide how to partition the system.

[1]Due to space constraints, I omit experiments where I keep the same pivot parameter, but vary the groupings of free parameters. The results are similar to the results of pivot parameter selection.

Accuracy for Different Sub-system Densities (E)						
E. Density	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
10%($join$)	4E-5	4E-5	**4.5E-5**	9E-9	2E-5	2E-6
10% (zero-join)	3.4E-3	3.3E-3	**3.8E-3**	9E-9	2E-5	2E-6
100%	0.47	0.48	**0.57**	9E-8	2E-4	2E-4

(a) Accuracy

Decomposition Time for Different Sub-system Densities (E) (sec.)						
E. Density	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
10%($join$)	207	202	201	190	175	183
10% (zero-join)	602	640	617	190	175	183
100%	1696	1645	1576	379	201	244

(b) Time (sec.)

Table 6.7: Results for Different Sub-system Densities (Double Pendulum, Resolution=70, Rank = 10, Pivot=t, $p = 100\%$)

Accuracy for Different Pivot Parameters						
Pivot	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
t	0.47	0.48	**0.57**	9E-8	2E-4	2E-4
ϕ_1	0.35	0.36	**0.40**			
ϕ_2	0.40	0.41	**0.56**			
m_1	0.58	0.59	**0.71**			
m_2	0.41	0.40	**0.42**			

(a) Accuracy

Decomposition Time for Different Pivot Parameters (sec.)						
Pivot	M2TD			Random	Grid	Slice
	AVG	CONCAT	SELECT			
t	1696	1645	1576	379	201	244
ϕ_1	1607	1673	1673			
ϕ_2	1694	1677	1571			
m_1	1661	1512	1697			
m_2	1556	1602	1538			

(b) Time (sec.)

Table 6.8: Results for Different Pivots (Double Pendulum, Resolution=70, Rank = 10, $p =$ 100%, $e =$ 100%; 3-mode Sub-systems Are Created in Such a Way That Free Parameters of the Same Pendulum Are Kept in the Same Sub-system)

Chapter 7

CONCLUSION

The main goal of this dissertation is to optimized the tensor decomposition for two major challenges - Density and Noise. I particularly look at those two different kinds of challenges for different scenarios. Each of these challenges for tensor decomposition under different scenario has their own unique propertywhich need to be investigated separately. In this dissertation, I proposed different algorithms that tackled these challenges for each of the above mentioned different scenarios.

7.1 Noise-Profile Adaptive Tensor Decomposition

Web-based user data can be noisy. Recent research has shown that it is possible to improve the resilience of the tensor decomposition process to overfitting (an important challenge in the presence of noisy data) by relying on probabilistic techniques. However, existing techniques assume that all the data and intermediary results can fit in the main memory and (more critically) they treat the entire tensor uniformly, ignoring potential non-uniformities in the noise distribution. In chapter 3, I proposed a novel noise-adaptive decomposition (nTD) technique that leverages rough information about noise distribution to improve the tensor decomposition performance. nTD partitions the tensor into multiple sub-tensors and then decomposes each sub-tensor probabilistically through Bayesian factorization. The noise profiles of the grid partitions and their alignments are then leveraged to develop a sample assignment strategy (or s-strategy) that best suits the noise profile of a given tensor. Experiments with user-centered web data show that nTD is significantly better than conventional CP decomposition on noisy tensors.

7.2 Noise Profile Adaptive Tensor Train Decomposition

Tensor train decomposition, one of the widely used tensor decomposition techniques, is designed to avoid the curse of dimensionality, in the form of the exposition of intermediary results, which plagues other tensor decomposition techniques.However, many tensor decomposition schemes, including tensor train decomposition is sensitive to noise in the input data streams: this is especially true for relatively sparse web and social network datasets, where incorrect or inconsistent data, an inevitable problem in the real world, can lead to false conclusions and recommendations. The problem is compounded by over-fitting as the web and user data are sparse. These techniques have a major deficiency: they treat the entire tensor uniformly, ignoring potential non-uniformities in the noise distribution. The noise is rarely uniformly distributed in the data. In chapter 4, Noise-Profile Adaptive Tensor Train Decomposition (NTTD) method is proposed, which aims to tackle this challenge. In particular, NTTD leverages a model-based noise adaptive tensor train decomposition strategy for the purposes: any rough priori knowledge about the noise profiles of the tensor enable us to develop a sample assignment

7.3 Tensor Decomposition for Billion - Scale Dense Tensor

One key problem with tensor decomposition is its computational complexity and space requirements. Especially, as the relevant data sets get denser, in-memory schemes for tensor decomposition become increasingly ineffective; therefore out-of-core (secondary-memory supported, potentially parallel) computing is necessitated. However, existing techniques do not consider the I/O and network data exchange costs that out of core execution of the tensor decomposition operation will incur. In chapter 5, I note that when this operation is implemented with the help of secondary-memory and/or multiple servers to tackle the memory limitations, I would need intelligent buffer-management and task-scheduling techniques which take into account the cost of bringing the relevant blocks into the buffer to minimize I/O in the system. I introduce 2PCP, a two-phase, block-based CP decomposition system with intelligent buffer sensitive task scheduling and buffer management mechanisms. 2PCP aims to reduce I/O costs in the analysis of relatively dense tensors common in scientific and engineering applications. Experiment results compare with current state of art tensor decomposition algorithms and show that our algorithms can significantly reduce the amount of I/O and execution time while maintaining decomposition accuracy.

7.4 Multi-Task Tensor Decomposition for Sparse Ensemble Simulation

Data- and model-driven computer simulations are increasingly critical in many application domains. These simulations may track 10s or 100s of parameters, affected by complex interdependent dynamic processes. Moreover, decision makers usually need to run large simulation ensembles, containing 1000s of simulations. In this chapter 6, a tensor-based framework is proposed to represent and analyze patterns in large simulation ensemble data sets to obtain a high-level understanding of the dynamic processes implied by a given ensemble of simulations. The inherent sparsity of the simulation ensembles (relative to the space of potential simulations one can run) constitutes a significant problem in discovering these underlying patterns. To address this challenge, a partition-stitch sampling scheme is proposed, which divides the parameter space into subspaces to collect several lower modal ensembles, and complement this with a novel Multi-Task Tensor Decomposition (M2TD) technique which helps effectively and efficiently stitch these sub ensembles back. Experiments showed that, for a given budget of simulations, the proposed structured sampling scheme leads to significantly better overall accuracy relative to traditional sampling approaches, even when the user does not have a perfect information to help guide the structured partitioning process.

REFERENCES

[1] Abedjan, Z., X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker and N. Tang, “Detecting data errors: Where are we and what needs to be done?”, Proceedings of the VLDB Endowment **9**, 12, 993–1004 (2016).

[2] Bader, B. W., T. G. Kolda *et al.*, “Matlab tensor toolbox version 2.6”, Available online, URL `http://www.sandia.gov/~tgkolda/TensorToolbox/` (2015).

[3] Balakrishnan, R. and S. Kambhampati, “Sourcerank: relevance and trust assessment for deep web sources based on inter-source agreement”, in “Proceedings of the 20th international conference on World wide web”, pp. 227–236 (ACM, 2011).

[4] Ballani, J. and L. Grasedyck, “A projection method to solve linear systems in tensor format”, Numerical linear algebra with applications **20**, 1, 27–43 (2013).

[5] Ballani, J., L. Grasedyck and M. Kluge, “Black box approximation of tensors in hierarchical tucker format”, Linear algebra and its applications **438**, 2, 639–657 (2013).

[6] Bebendorf, M., “Adaptive cross approximation of multivariate functions”, Constructive approximation **34**, 2, 149–179 (2011).

[7] Bergé, P., Y. Pomeau and C. Vidal, “Order within chaos: towards a deterministic approach to turbulence. 1986”, New York: Wiley Google Scholar (1986).

[8] Brin, S. and L. Page, “The anatomy of a large-scale hypertextual web search engine”, Computer networks and ISDN systems **30**, 1-7, 107–117 (1998).

[9] Brown, P. G., “Overview of scidb: large scale array storage, processing and analysis”, in “Proceedings of the 2010 ACM SIGMOD International Conference on Management of data”, pp. 963–968 (ACM, 2010).

[10] Carroll, J. D. and J.-J. Chang, “Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition”, Psychometrika **35**, 3, 283–319 (1970).

[11] Chakrabarti, S., “Dynamic personalized pagerank in entity-relation graphs”, in “Proceedings of the 16th international conference on World Wide Web”, pp. 571–580 (ACM, 2007).

[12] Chen, X. and K. S. Candan, “Lwi-svd: low-rank, windowed, incremental singular value decompositions on time-evolving data sets”, in “Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining”, pp. 987–996 (ACM, 2014).

[13] Chu, X., I. F. Ilyas, P. Papotti and Y. Ye, “Ruleminer: Data quality rules discovery”, in “2014 IEEE 30th International Conference on Data Engineering (ICDE)”, pp. 1222–1225 (IEEE, 2014).

[14] Cichocki, A., N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, D. P. Mandic *et al.*, “Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions”, Foundations and Trends® in Machine Learning **9**, 4-5, 249–429 (2016).

[15] Cichocki, A., A.-H. Phan, Q. Zhao, N. Lee, I. Oseledets, M. Sugiyama, D. P. Mandic *et al.*, “Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives”, Foundations and Trends® in Machine Learning **9**, 6, 431–673 (2017).

[16] Cohen, J., B. Dolan, M. Dunlap, J. M. Hellerstein and C. Welton, “Mad skills: new analysis practices for big data”, Proceedings of the VLDB Endowment **2**, 2, 1481–1492 (2009).

[17] Davidson, I., S. Gilpin, O. Carmichael and P. Walker, “Network discovery via constrained tensor analysis of fmri data”, in “Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining”, pp. 194–202 (ACM, 2013).

[18] Drumond, L., S. Rendle and L. Schmidt-Thieme, “Predicting rdf triples in incomplete knowledge bases with tensor factorization”, in “Proceedings of the 27th Annual ACM Symposium on Applied Computing”, pp. 326–331 (ACM, 2012).

[19] Georghiades, A., P. Belhumeur and D. Kriegman, “From few to many: Illumination cone models for face recognition under variable lighting and pose”, IEEE Trans. Pattern Anal. Mach. Intelligence **23**, 6, 643–660 (2001).

[20] Grasedyck, L. and W. Hackbusch, “An introduction to hierarchical (h-) rank and tt-rank of tensors with examples”, Computational Methods in Applied Mathematics Comput. Methods Appl. Math. **11**, 3, 291–304 (2011).

[21] Harper, F. M. and J. A. Konstan, “The movielens datasets: History and context”, Acm transactions on interactive intelligent systems (tiis) **5**, 4, 19 (2016).

[22] Harper, F. M. and J. A. Konstan, “The movielens datasets: History and context”, Acm transactions on interactive intelligent systems (tiis) **5**, 4, 19 (2016).

[23] Harshman, R. A., “Foundations of the parafac procedure: Models and conditions for an” explanatory” multimodal factor analysis”, (1970).

[24] Hilbert, D., *Über die stetige Abbildung einer Linie auf ein Flächenstück*, pp. 1–2 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1935), URL `https://doi.org/10.1007/978-3-662-38452-7_1`.

[25] Holtz, S., T. Rohwedder and R. Schneider, “The alternating linear scheme for tensor optimization in the tensor train format”, SIAM Journal on Scientific Computing **34**, 2, A683–A713 (2012).

[26] Jeon, I., E. E. Papalexakis, U. Kang and C. Faloutsos, “Haten2: Billion-scale tensor decompositions”, in “Data Engineering (ICDE), 2015 IEEE 31st International Conference on”, pp. 1047–1058 (IEEE, 2015).

[27] Jordan, M. I., Z. Ghahramani, T. S. Jaakkola and L. K. Saul, "An introduction to variational methods for graphical models", Machine learning **37**, 2, 183–233 (1999).

[28] Kang, U., E. Papalexakis, A. Harpale and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries", in "Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining", pp. 316–324 (ACM, 2012).

[29] Keahey, K., J. Mambretti, D. Panda, P. Rad, W. Smith and D. Stanzione, "Nsf chameleon cloud", website, November (2014).

[30] Khoromskij, B. and A. Veit, "Efficient computation of highly oscillatory integrals by using qtt tensor approximation", Computational Methods in Applied Mathematics **16**, 1, 145–159 (2016).

[31] Kim, M. and K. S. Candan, "Efficient static and dynamic in-database tensor decompositions on chunk-based array stores", in "Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management", pp. 969–978 (ACM, 2014).

[32] Kim, M. and K. S. Candan, "Tensordb: in-database tensor manipulation with tensor-relational query plans", in "Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management", pp. 2039–2041 (ACM, 2014).

[33] Kim, M. and K. S. Candan, "Decomposition-by-normalization (dbn): leveraging approximate functional dependencies for efficient cp and tucker decompositions", Data mining and knowledge discovery **30**, 1, 1–46 (2016).

[34] Klus, S., P. Gelß, S. Peitz and C. Schütte, "Tensor-based dynamic mode decomposition", Nonlinearity **31**, 7, 3359 (2018).

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

[36] Kolda, T. G. and J. Sun, "Scalable tensor decompositions for multi-aspect data mining", in "Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on", pp. 363–372 (IEEE, 2008).

[37] Li, X., K. S. Candan and M. L. Sapino, "ntd: Noise-profile adaptive tensor decomposition", in "Proceedings of the 26th International Conference on World Wide Web", pp. 243–252 (International World Wide Web Conferences Steering Committee, 2017).

[38] Li, X., K. S. Candan and M. L. Sapino, "M2td: multi-task tensor decomposition for sparse ensemble simulations", in "2018 IEEE 34th International Conference on Data Engineering (ICDE)", pp. 1144–1155 (IEEE, 2018).

[39] Li, X., S. Huang, K. S. Candan and M. L. Sapino, "Focusing decomposition accuracy by personalizing tensor decomposition (ptd)", in "Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management", pp. 689–698 (ACM, 2014).

[40] Li, X., S. Huang, K. S. Candan and M. L. Sapino, "Focusing decomposition accuracy by personalizing tensor decomposition (ptd)", in "Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management", pp. 689–698 (ACM, 2014).

[41] Li, X., S. Huang, K. S. Candan and M. L. Sapino, "2pcp: Two-phase cp decomposition for billion-scale dense tensors", in "Data Engineering (ICDE), 2016 IEEE 32nd International Conference on", pp. 835–846 (IEEE, 2016).

[42] Liu, S., Y. Garg, K. S. Candan, M. L. Sapino and G. Chowell-Puente, "Notes2: Networks-of-traces for epidemic spread simulations.", in "AAAI Workshop: Computational Sustainability", (2015).

[43] Mnih, A. and R. R. Salakhutdinov, "Probabilistic matrix factorization", in "Advances in neural information processing systems", pp. 1257–1264 (2008).

[44] Morton, G. M., "A computer oriented geodetic data base and a new technique in file sequencing", (1966).

[45] Neal, R. M., "Probabilistic inference using markov chain monte carlo methods", (1993).

[46] on Disaster Reduction, S., "Grand challenges for disaster reduction", (2005).

[47] Oseledets, I. and E. Tyrtyshnikov, "Tt-cross approximation for multidimensional arrays", Linear Algebra and its Applications **432**, 1, 70–88 (2010).

[48] Oseledets, I. V., "Tensor-train decomposition", SIAM Journal on Scientific Computing **33**, 5, 2295–2317 (2011).

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

[50] Phan, A. H. and A. Cichocki, "Block decomposition for very large-scale nonnegative tensor factorization", in "Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2009 3rd IEEE International Workshop on", pp. 316–319 (IEEE, 2009).

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

[52] Priebe, C., J. Conroy, D. Marchette and Y. Park, "Enron data set, 2006", URL http://cis. jhu. edu/˜ parky/Enron/enron. html (2006).

[53] Rogers, M., L. Li and S. J. Russell, "Multilinear dynamical systems for tensor time series", in "Advances in Neural Information Processing Systems", pp. 2634–2642 (2013).

[54] Sadiq, S. and P. Papotti, "Big data quality-whose problem is it?", in "Data Engineering (ICDE), 2016 IEEE 32nd International Conference on", pp. 1446–1447 (IEEE, 2016).

[55] Silvestro Roberto, P., L. S. Maria, L. Sicong, C. Xilun, G. Yash, H. Shengyu, H. K. Jung, L. Xinsheng, N. Parth and K. Selcuk Candan, “Simdms: Data management and analysis to support decision making through large simulation ensembles”, in “20th International Conference on Extending Database Technology (EDBT’17)”, pp. 582–585 (OpenProceedings. org, 2017).

[56] Sun, J., S. Papadimitriou and S. Y. Philip, “Window-based tensor analysis on high-dimensional and multi-aspect streams.”, in “ICDM”, vol. 6, pp. 1076–1080 (2006).

[57] Tang, J., H. Gao and H. Liu, “mtrust: discerning multi-faceted trust in a connected world”, in “Proceedings of the fifth ACM international conference on Web search and data mining”, pp. 93–102 (ACM, 2012).

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

[59] Xiong, L., X. Chen, T.-K. Huang, J. Schneider and J. G. Carbonell, “Temporal collaborative filtering with bayesian probabilistic tensor factorization”, in “Proceedings of the 2010 SIAM International Conference on Data Mining”, pp. 211–222 (SIAM, 2010).

[60] Zafarani, R. and H. Liu, “Users joining multiple sites: Friendship and popularity variations across sites”, Information Fusion **28**, 83–89 (2016).

[61] Zafarani, R., L. Tang and H. Liu, “User identification across social media”, ACM Transactions on Knowledge Discovery from Data (TKDD) **10**, 2, 16 (2015).

[62] Ziegler, C.-N., S. M. McNee, J. A. Konstan and G. Lausen, “Improving recommendation lists through topic diversification”, in “Proceedings of the 14th international conference on World Wide Web”, pp. 22–32 (ACM, 2005).