TensorDB and Tensor-Relational Model (TRM) for Efficient Tensor-Relational Operations

by

Mijung Kim

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

Approved July 2014 by the
Graduate Supervisory Committee:

Kasim Selçuk Candan, Chair
Hasan Davulcu
Hari Sundaram
Jieping Ye

ARIZONA STATE UNIVERSITY

August 2014

ABSTRACT

Multidimensional data have various representations. Thanks to their simplicity in modeling multidimensional data and the availability of various mathematical tools (such as tensor decompositions) that support multi-aspect analysis of such data, tensors are increasingly being used in many application domains including scientific data management, sensor data management, and social network data analysis. Relational model, on the other hand, enables semantic manipulation of data using relational operators, such as projection, selection, Cartesian-product, and set operators. For many multidimensional data applications, tensor operations as well as relational operations need to be supported throughout the data life cycle. In this thesis, we introduce a tensor-based relational data model (TRM), which enables both tensor-based data analysis and relational manipulations of multidimensional data, and define tensor-relational operations on this model. Then we introduce a tensor-relational data management system, so called, TensorDB. TensorDB is based on TRM, which brings together relational algebraic operations (for data manipulation and integration) and tensor algebraic operations (for data analysis). We develop optimization strategies for tensor-relational operations in both in-memory and in-database TensorDB. The goal of the TRM and TensorDB is to serve as a single environment that supports the entire life cycle of data; that is, data can be manipulated, integrated, processed, and analyzed.

ACKNOWLEDGMENTS

I would like to thank, first and foremost, my advisor, Dr. Kasim Selçuk Candan for his encouragement and support, and for guiding me with his knowledge and experience throughout my PhD study. He continuously encouraged me with his generosity and patience whenever I came across challenges and difficulties in carrying out my research and helped me through. Without his guidance and support, this thesis would not have been accomplished. I thank my committee members, Dr. Hasan Davulcu, Dr. Hari Sundaram, and Dr. Jieping Ye for sharing their insights for my thesis. I am thankful to all my colleagues at EMIT lab who I had great pleasure to work with. I must also acknowledge iTel Companies, Inc., BioDesign Institute at ASU, and HP Labs that I was fortunate enough to work for as an intern and supported me for my PhD study. Especially I cannot say more thanks to my mentor, Jun Li at HP Labs for his great deal of guidance and support. Last but not least, I would like to express my love and gratitude to my parents, Juho Kim and Youngsun Park, and my beloved son, Heejae Kang, for their endless encouragement and emotional support. I hope that they are proud of my achievements, as I am proud of them.

This work is supported by the NSF Grants #1043583, MiNC: NSDL Middleware for Network- and Context-aware Recommendations, #116394, RanKloud: Data Partitioning and Resource Allocation Strategies for Scalable Multimedia and Social Media Analysis, and #1016921, One Size Does Not Fit All: Empowering the User with User-Driven Integration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 Motivation

From data collection to decision making, the life cycle of data often involves various steps of operations, integration, manipulation, and analysis. Figure 1.1 depicts a life cycle of data from the extraction to the data analysis through multiple operations.

Today's complex data analytic problems involved in scientific, sensor data management, and social network data analysis require mathematically and algorithmically sophisticated data processing and analysis more and more. In the data life cycle, data are often integrated from different sources before it goes through other manipulation steps and the final step of the data processing step is almost always the data analysis.

To be able to provide end-to-end support for the full data life cycle, today's data management and decision making systems increasingly incorporate operations for data manipulation, integration with data analysis.

Today's data are unprecedentedly large-scale. Scalability is one of the important aspects to be satisfied in the data management frameworks. As conventional relational models support query optimization strategies for the cost reduction in relational query plans, the new data models require to design new query optimization strategies for the cost reduction in query plans involving not only data manipulation and integration operations but also data analysis operations.

Multidimensional data have various representations. Let $\mathrm{A}_1, \ldots, \mathrm{A}_n$ be a set of attributes in a relation and $\mathrm{D}_1, \ldots, \mathrm{D}_n$ be the attribute domains. The *relational model* [22] represents the data as sets of tuples, where each tuple is an instance in $\mathrm{D}_1 \times$

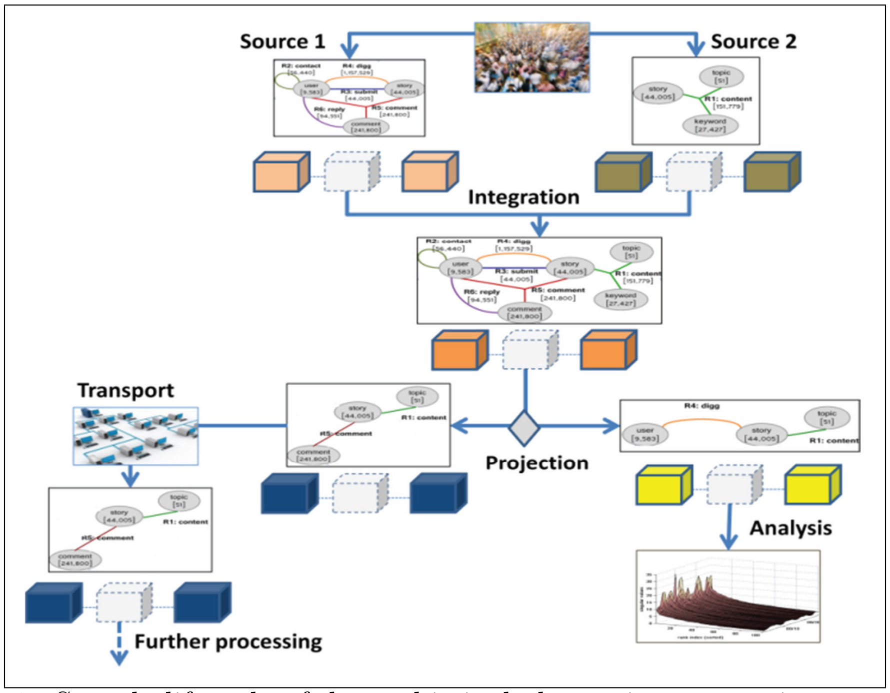


Figure 1.1: Sample lifecycle of data; this includes various operations, such as capture, integration, projection, decomposition, and data analysis

$\ldots \times \mathrm{D}_n$; the model also encodes the functional dependencies between the attributes. The *vector model* [64] maps each attribute to a dimension in an n-dimensional space and represents each tuple as a point in this space (a natural representation when attributes are totally ordered). The *tensor model*, on the other hand, maps each attribute to a mode in an n-dimensional array where each possible tuple is a cell, the existence (absence) of a particular tuple in a database instance can be denoted as 1 (0) in the cell; similarly, the model can also represent fuzzy or probabilistic tuples by filling the cells with values between 0 and 1. The clear notions of neighborhood in tensor is important in data with ordered attribute domains such as text, image, and video data and also data analysis operations, such as convolution, data clustering, and compression, which heavily rely on the neighborhood definition in the data.

A tensor is a higher-order generalization of a matrix. While matrices have only two dimensions, there are many scenarios that we need more; e.g., data analysis on time-evolving social network data (e.g., authors, keywords, timestamps) [70] and computer vision problems since images are naturally formed by the interaction of multiple factors that depend on scene geometry, viewpoint, and illumination conditions [76]. Two-way analysis methods may not capture underlying information of the data and two-way factor models are often not accurate nor unique. For example, for a unique solution, SVD (Singular Value Decomposition) of a matrix requires additional constraints such as orthogonality constraints, on the other hand, CP [19, 34] decomposition is unique with much weaker conditions [46]. Tensor-based data analysis has advantage over two-way data model in numerous research areas in terms of uniqueness, robustness to noise, and easy interpretation, etc.

As well as the tensor model provides a natural representation in modeling multidimensional data, the availability of mathematical tools, such as tensor decomposition, that support multi-aspect analysis of multidimensional data promotes the use of tensors in many application domains including scientific data management, sensor data management, and social network data analysis. Relational model, on the other hand, enables semantic manipulation of data using relational operators, such as projection, selection, Cartesian-product, and set operators such as union and intersection. Therefore, combining the tensor model and the relational model enables to support both data management and analysis operations, i.e., the relational algebraic operations support data manipulation and integration and the tensor algebraic operations support data analysis.

1.2 Challenges in Tensor-Relational Data Management

Lack of Tensor-Relational based Data Management and High Cost in Tensor Decomposition. However, there is little prior research done on efficient implementation of complex and semantically-rich data operations, such as joins, in conjunction with tensor analysis operations, such as tensor decompositions. Moreover, we need to deal with complex data processing plans where multiple relational algebraic and tensor algebraic operations are composed with each other.

While, in traditional relational algebra, the costliest operation is known to be the join, in a framework that provides both relational and tensor operations, tensor decomposition tends to be the computationally costliest operation. In dense tensor representation, the cost increases exponentially with the number of modes of the tensor. While decomposition cost increases more slowly (linearly with the number of nonzero entries in the tensor) for sparse tensors, the operation can often be prohibitive for today's large-scale data sets. Therefore, it is most critical to manipulate the data processing plans in a way that reduces the cost of the tensor decomposition step.

There are several strategies to address this high cost of tensor decompositions for efficient tensor-based data analysis.

Phan *et al.* [59] proposed a modified ALS PARAFAC algorithm called grid PARAFAC for large scale tensor data. The grid PARAFAC divides a large tensor into sub-tensors that can be factorized using any available PARAFAC algorithm in a parallel manner and iteratively combines into the final decomposition. The grid PARAFAC can be converted to grid NTF by enforcing nonnegativity. [78] parallelized NTF by dividing a given original 3-mode tensor into three semi-non negative matrix factorization problems. These matrices are distributed to independent processors to facilitate parallelization. [10] presented an algorithm for NTF that is specialized for Compute Uniform Device Architecture (CUDA) parallel computing framework.

GigaTensor [39] employed the MapReduce framework to address the intermediate memory blow-up problem in PARAFAC and run large-scale tensor decomposition. [58] proposed a highly parallelizable tensor decomposition algorithm, which produces sparse approximation of tensor decompositions.

When ALS-based tensor decomposition run in parallel, since one variable can be optimized given that the other variables are fixed, the communication overhead is not avoidable. Tucker decomposition [74] is even more challenging for parallelization. Unlike the CP decomposition where the factors in different modes can only interact factorwise, the core tensor of Tucker decomposition allows an interaction for a factor with any factor in the other modes, which makes it hard for Tucker decomposition to split and solve independently in parallel. Moreover none of these works provide optimization strategies for tensor decomposition in conjunction with semantically-rich data operations such as join operations.

In-Memory Limitation of Tensor-based Data Analysis. MATLAB is widely used as a mathematical software package for manipulating and analyzing multidimensional data which is represented in a multidimensional array. In MATLAB, many external tools including [12] support a tensor model and tensor algorithms such as CP [19, 34] and Tucker [74] for data analysis. While MATLAB-based in-memory linear algebra operations are widely used for implementing tensor decomposition algorithms, these implementations are limited with the amount of memory available to the MATLAB software. As the today's data sets get larger, these in-memory based schemes for tensor decomposition become increasingly ineffective. Moreover in-memory based tensor decomposition operations often result 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 difficult. In-database tensor decomposition operation on disk-resident tensor data can be a solution to eliminate the challenge posed by the memory-limitations.

Challenges in In-Database Models for Tensor Representation. Relational databases are not suitable for storage and manipulation of large arrays. Arrays are ordered and rely on the neighborhood definition which is important in ordered attribute domain (text, image, and video, etc). However relational database is based on set so the ordering information must be explicitly defined in the schema and stored in the database [18]. Many data analytic algorithms on array data in aforementioned domains are iterative tasks such as gradient descent, which is not suitable to be expressed in SQL due to the lack of convenient syntax for iteration in SQL [35]. Tables with a primary key in relational database can be viewed as one dimensional array however arrays can have any number of dimensions. Many relational databases support a built-in array type and a limited set of basic array operations. For extending basic array-based operations, they provide user-defined functions (UDF) and aggregates (UDA). However, one critical limitation of UDF/UDA-based approaches is that the output data should be resided in the available memory [35] and it is not always the case in tensor manipulation.

The array model [14, 18, 27, 75] is a natural representation to store multidimensional data and facilitate multidimensional data analysis. There are several approaches to represent array based data. The first approach is to represent the array in the form of a table: e.g., a 2D array $A[row, column]$ can be represented using a relational schema $(row, column, value)$ [75] or, if the model allows vector data types, as (row, row_vector) [23]. A second approach is to use blob type in a relational database as a storage layer for array data [14, 27]. Sparse matrices can also be represented using a graph-based abstraction [51]. For example, in [51], ALS (alternating least squares) is solved using a graph algorithm that represents a sparse matrix as a bipartite graph. The last approach is to consider a native array model and an array-based storage scheme, such as a chunk-store, as in [18].

Although array models fit in representation of tensor data, none of these approaches support tensor decomposition algorithms. Moreover, in-database tensor

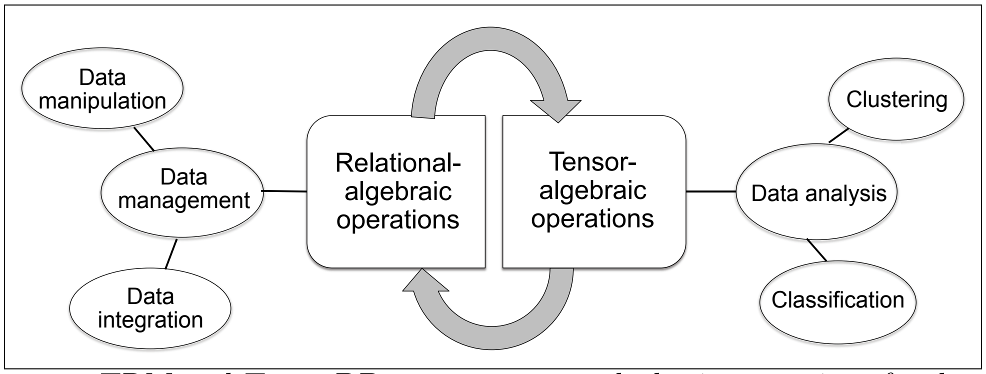


Figure 1.2: TRM and TensorDB support tensor-algebraic operations for data analysis as well as relational-algebraic operations for data manipulation and integration

decomposition algorithms tend to involve computationally expensive operations and require significant amounts of data movement, which also results in high I/O load and many operations involved in tensor decomposition are order sensitive and the way data is laid on disk may have a big impact on the total cost of tensor decomposition task. These necessitate optimization strategies for in-database tensor decompositions.

1.3 Contributions

1.3.1 TensorDB and Tensor-Relational Model (TRM)

The main goal of this thesis is to build a tensor-relational data management system, so called, TensorDB. TensorDB is based on the tensor-based relational model (TRM) in which we define tensor-relational operations on data represented as tensor [41]. TRM and TensorDB bring together relational algebraic operations (for data manipulation and integration) and tensor algebraic operations (for data analysis) to support the entire life cycle of data (Figure 1.2).

We also consider the in-database implementations of tensor-relational operations on disk-resident data to address the memory limitations and introduce the in-database TensorDB on chunk-based array data stores. In-database TensorDB extends an open source software platform of data management and analytic system for array data, SciDB [4]. As an extension of SciDB, TensorDB shares the basic system architecture and the query languages of SciDB and performs all-in-one from the query interface and query optimization to the query execution for tensor-relational operations.

1.3.2 Optimization Strategies in In-Memory and In-Database TensorDB

TensorDB deals with complex data processing plans where multiple relational algebraic and tensor algebraic operations are composed with each other. As an optimization strategy for the tensor-relational query plans, we consider the *decomposition push-down* technique to reduce the cost of tensor decomposition (which is the most expensive operation in the tensor-relational model) when running with data integration operations such as join and union and we propose `join-by-decomposition` (JBD) and `union-by-decomposition` (UBD).

To address the high-cost of the tensor decomposition by reducing the number of modes that is the main factor of the cost, we consider *vertical partitioning* strategy and propose the `decomposition-by-normalization` (DBN) scheme that leverages the vertical partitioning technique.

In these optimized schemes of decomposition push-down and vertical partitioning strategies, i.e, `join-by-decomposition`, `union-by-decomposition`, and `decomposition-by-normalization`, each individual decomposition of the subtensors can also be obtained in parallel, leading to highly parallelizable execution plans.

For the in-database TensorDB, to minimize the data movement in operations, which causes the high I/O, we consider data-ordering optimization and materialization for the matricization operation in the in-database static tensor decomposition and to optimize in-database matrix multiplications, the compressed matrix multiplication technique [57] is leveraged. We use the compressed matrix multiplication technique for the covariance matrix computation in the in-database incremental tensor decomposition.

1.3.3 Join-By-Decomposition (JBD): Optimized Query Plan with Joins and Decompositions

As we see in Figure 1.1, data are often integrated from different sources before it goes through other manipulation steps and the final step of the tensor relational query plan is almost always a tensor decomposition operation for data analysis. This is a challenging situation since the decomposition operation is preceded by a join operation increases the number of modes of the tensor to be decomposed and we develop a decomposition push-down strategy for the query plan of the tensor decomposition operation when combined with the join operation [41].

- Given a query plan that joins the two relational tensors and then performs tensor decompositions on the joined tensor, we propose an alternative query plan, so called `join-by-decomposition` (JBD), that would involve first decomposing the input tensors into their spectral components and then combining these into the decomposition of the joined tensor. Since the join operation tends to push the cost of tensor decomposition higher, we argue that a `join-by-decomposition` (JBD) scheme will be more efficient than the join on the input tensors first, then the decomposition on the joined tensor, so called, `join-then-decompose` (JTD).

- There are many different ways that one can decompose the input tensors and combine them to obtain the final decomposition. Each different scheme may have different processing costs and accuracies. We explore various measures to determine the best approximation with respect to the original joined tensor.

1.3.4 Union-by-Decomposition (UBD): Pushing-Down Tensor Decomposition Strategy to Promote Reuse of Materialized Decompositions

As a data integration operation like the join operation, the union operation tends to increase the size of the input tensor, so does the cost of tensor decomposition when it is performed with tensor decomposition operation. Thus the pushing-down tensor decomposition strategy can help optimize data processing workflows that involve data integration from multiple sources through unions and tensor decomposition tasks. Given a query plan that performs first the union operation on the data and then performs the tensor decomposition on the union of the data, which we refer to as `union-then-decompose` (UTD), an alternative query plan with decomposition push-down first performs the tensor decompositions on each smaller input data and then combines these decomposed tensors, which is referred to as `union-by-decomposition` (UBD) [44]. We argue that a `union-by-decomposition` (UBD) plan with decomposition push-down over the union operations reduces the overall data processing times and promotes reuse of materialized tensor decomposition results. Specifically the `union-by-decomposition` is advantageous over the conventional `union-then-decompose` (UTD) plan as followings:

- Since the union operation can combine relatively small and sparse tensors into a larger and denser tensor, the decomposition over the union data can be much more expensive than the decompositions over the input data sources. Moreover multiple tensor decompositions on input tensors can run in parallel, which will further reduce the cost.

- A `union-by-decomposition` (UBD) based plan provides opportunities for materializing decomposition of data tensors and re-using these materialized decompositions in more complex queries requiring integration of data.

1.3.5 Decomposition-by-Normalization (DBN): Leveraging Approximate Functional Dependencies for Efficient Tensor Decompositions

To tackle the high computational cost of tensor decomposition process, since the number of modes of the tensor data is one of the main factors contributing to the cost of the tensor operations, we focus on how we reduce the number of modes and the size of the input tensor. We argue that a higher-order tensor can be normalized (i.e., vertically partitioned) into multiple lower-order tensors which are decomposed independently, then combined into the decomposition of the original data tensor and the multiple compositions on lower-modal tensors is more efficient than one decompostion on a higher-modal tensor. We refer to this as the `decomposition-by-normalization` (DBN) scheme [42, 43].

- The `decomposition-by-normalization` scheme first normalizes the given relation into smaller tensors based on the functional dependencies of the relation and then performs the decomposition on these smaller tensors. The decomposition and recombination steps of the `decomposition-by-normalization` scheme fit naturally in settings with multiple cores.

- For the normalization process, we identify the approximate functional dependencies and partition the data into two partitions in such a way that will lead to least amount of errors during later stages.

1.3.6 In-Database Implementations of Tensor Decomposition Operations

To address the constraints imposed by the main memory limitations when handling large and high-order tensor data in the TensorDB, we consider in-database implementations of tensor decomposition operations on disk-resident data sets and propose in-database static and dynamic tensor decompositions and the optimization schemes based on chunk-based array data stores.

- For the static in-database tensor decomposition, we implement in-database alternating least squares operations on a chunk-based data storage system. We also focus on developing optimization strategies in the chunk-based data storage system such as optimization of the data ordering in operations and materialization to save the execution time for expensive operations such as matricization.

- For the incremental in-database tensor decomposition, we develop dynamic tensor analysis, so called DTA [70], which dynamically maintains and revises the tensor decomposition to avoid the cost of decomposing the data tensor from scratch with each update. We note that the covariance matrix computation of the matricized input tensor is the most computationally challenging operation in the in-database DTA thus we optimize it by leveraging recently introduced compressed matrix multiplication techniques [57].

1.3.7 Thesis Organization

The rest of this thesis is organized as follows.

- In Chapter 2, we review background and related works in the literature.
- In Chapter 3, we first introduce tensor-relational model (TRM) and we define tensor-relational operations on this model.
- In Chapter 4, we present TensorDB, which is based on TRM. We also focus on the optimization strategies for tensor-relational operations in both in-memory and in-database TensorDB.
- In Chapter 5, we present the `join-by-decomposition` scheme for the query plan of the tensor decomposition operation when combined with the join operation.
- In Chapter 6, we present the `union-by-decomposition` scheme that optimizes data processing workflows for data integration from multiple sources through unions and tensor decomposition tasks.
- In Chapter 7, we present the `decomposition-by-normalization` scheme for optimization of tensor decomposition operation.
- In Chapter 8, we present in-database static and dynamic tensor decomposition implementations, and optimization for core operations involved in in-database tensor decompositions.

In Chapter 9. we conclude the thesis and discuss the future work.

Chapter 2

BACKGROUND AND RELATED WORKS

We provide the relevant background and discuss the related works of the thesis.

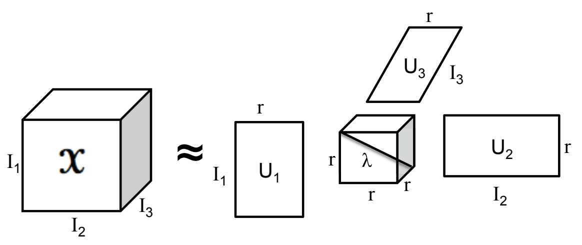
2.1 Tensor Representations

Tensors are generalizations of matrices: while a matrix is essentially a two dimensional array, a tensor is an array of arbitrary dimensions. Thus, a vector can be thought of as a tensor of 1^{st} order and an object-feature matrix is a tensor of 2^{nd} order, while a multi-sensor data stream (i.e., sensors, features of sensed data, and time) can be represented as a tensor of 3^{rd} order. As in the case of matrices, the dimensions of the tensor array are referred to as its *modes*. For example, an $M \times N \times K$ tensor of 3^{rd} order has three modes: M columns (mode 1), N rows (mode 2), and K *tubes* (mode 3). These 1D arrays are collectively referred to as the *fibers* of the given tensor. Similarly, the $M \times N \times K$ tensor can also be considered in terms of its M horizontal *slices*, N lateral slices, and K frontal slices: each slice is a 2D array (or equivalently a matrix, or a tensor of 2^{nd} order).

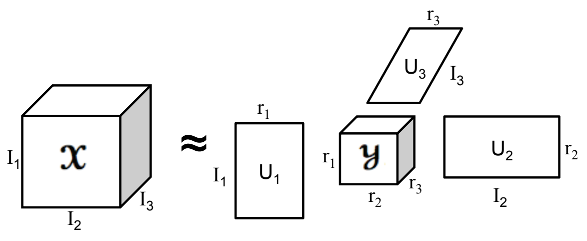
As matrices can be multiplied with other matrices or vectors, tensors can also be multiplied with other tensors, including matrices and vectors. For example, given an $M \times N \times K$ tensor, $\boldsymbol{\mathfrak{T}}$, and a $P \times N$ matrix, $\mathbf{A}$,

$$\boldsymbol{\mathfrak{T}}' = \boldsymbol{\mathfrak{T}} \times_2 \mathbf{A},$$

is an $M \times P \times K$ tensor where each lateral slice $\boldsymbol{\mathfrak{T}}[][j][]$ has been multiplied by $\mathbf{A}^T$. In the above example, the tensor-matrix multiplication symbol "$\times_2$" states that the matrix $\mathbf{A}^T$ will be multiplied with $\boldsymbol{\mathfrak{T}}$ over its lateral slices. Multiplication of a tensor



(a) CP decomposition



(b) Tucker decomposition

Figure 2.1: CP and Tucker decompositions

with a vector is defined similarly, but with a different notation: given M dimensional vector, $\vec{v}$,

$$\mathfrak{T}'' = \mathfrak{T}\bar{\times}_1\vec{v},$$

is a $N \times K$ tensor, such that $\vec{v}$ has been multiplied with each column, $\mathfrak{T}[][j][k]$. The symbol "$\bar{\times}_1$" states that vector $\vec{v}$ and columns of $\mathfrak{T}$ will get into dot products.

2.2 Tensor Decomposition

The order of a tensor is the number of modes (or ways). For example, a second-order tensor is simply a matrix. Matrix data is often analyzed for its latent semantics and indexed for search using a matrix decomposition operation known as the singular value decomposition (SVD). This operation identifies a transformation which takes data, described in terms of an m dimensional vector space, and maps them into a vector space defined by $k \leq m$ orthogonal basis vectors (also known as latent semantics) each with a score denoting its contributions in the given data set. The more general analysis operation which applies to tensors with more than two modes is known as the *tensor decomposition.*

Tensor decomposition has been used in a large number of domains, including signal processing, computer vision, and data mining. Tensor-based data representation and tensor analysis are also increasingly popular in emerging fields, such as social network analysis [13, 21, 28, 45, 47, 48, 49, 52, 71]. . The two most popular tensor decompositions are the CANDECOMP/PARAFAC [19, 34] decompositions (Figure 2.1(a)) and the Tucker [74] (Figure 2.1(b)).

CANDECOMP [19] and PARAFAC [34] decompositions (together known as the CP decomposition) take a different approach and decompose the input tensor into a sum of component rank-one tensors. The Tucker decomposition generalizes singular value matrix decomposition (SVD) to higher-dimensional matrices and decomposes a given tensor into a core tensor multiplied by a matrix along each mode.

More specifically, the rank-r CP Decomposition, $CP(\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N})$, of the tensor $\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N}$ is defined as $\mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(N)}$ such that

$$\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N} \approx \sum_{k=1}^{r} P_k^{(1)} \circ P_k^{(2)} \circ \cdots \circ P_k^{(N)}. \tag{2.1}$$

We also use the formulation where the column vectors of each factor are normalized to the unit length with the weights absorbed into a vector λ; i.e., $CP(\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N}) = \langle \lambda, \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(N)} \rangle$, such that

$$\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N} \approx \sum_{k=1}^{r} \lambda_k \circ P_k^{(1)} \circ P_k^{(2)} \circ \cdots \circ P_k^{(N)}, \tag{2.2}$$

where λ_i is the ith element of vector λ of size r and $U_i^{(n)}$ is the ith unit-length column vector of the matrix $\mathbf{P}^{(n)}$ of size $I_n \times r$, for $n = 1, \cdots, N$.

Note that the CP decomposition operation is an approximate operation and $\boldsymbol{\mathcal{P}}$ may not be exactly reconstructed from $\tilde{\boldsymbol{\mathcal{P}}}$. In other words, the following weighted sum, $\hat{\mathcal{P}}$, of the rank-one tensors may be different from $\mathcal{P}$:

$$\hat{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N} = \sum_{k=1}^{r} \lambda_k \circ U_k^{(1)} \circ U_k^{(2)} \circ \cdots \circ U_k^{(N)}. \tag{2.3}$$

Therefore, the norm of $\mathcal{P}$ denoted by $\|\mathcal{P}\|$ may also be different from $\|\hat{\mathcal{P}}\|$. Note that $\|\hat{\mathcal{P}}\|$ can be computed directly from the decomposition $\tilde{\mathcal{P}}$ without having to reconstruct the tensor $\hat{\mathcal{P}}$, since $\|\hat{\mathcal{P}}\| = \|\tilde{\mathcal{P}}\|$, which is computed in [11] as

$$\|\tilde{\mathcal{P}}\| = \lambda^T (\mathbf{U}^{(N)T}\mathbf{U}^{(N)} * \cdots * \mathbf{U}^{(1)T}\mathbf{U}^{(1)})\lambda. \tag{2.4}$$

Tucker decomposition [74] generalizes singular value matrix decomposition (SVD) to higher-dimensional tensors. The rank-$(r_1, r_2, ..., r_N)$ Tucker Decomposition of the tensor $\mathcal{P}_{I_1 \times I_2 \times \cdots \times I_N}$ can be defined as

$$Tucker(\mathcal{P}_{I_1 \times I_2 \times \cdots \times I_N}) = \tilde{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N} = \langle \mathcal{G}, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \rangle,$$

such that

$$\mathcal{P}_{I_1 \times I_2 \times \cdots \times I_N} \approx \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)}, \tag{2.5}$$

where $\mathcal{G}$ is a core tensor of size $r_1 \times r_2 \times \cdots \times r_N$ and $\mathbf{U}^{(n)}$ is the n^{th} factor matrix of size $I_n \times r_n$, for $n = 1, ..., N$. Again, the norm of Tucker decomposition of $\mathcal{P}$, $\|\hat{\mathcal{P}}\|$ is computed directly from the decomposition $\tilde{\mathcal{P}}$ (see [11] for the computation of $\|\tilde{\mathcal{P}}\|$).

For example, an $M \times N \times K$ tensor, $\mathcal{T}$, is decomposed for rank-(r, s, t) as follows:

$$\mathcal{T}_{M \times N \times K} \;\approx\; \mathcal{G}_{r \times s \times t} \times_1 \mathbf{U}_{M \times r} \times_2 \mathbf{V}_{N \times s} \times_3 \mathbf{X}_{K \times t}.$$

Tucker decomposition fails to guarantee a unique and perfect decomposition of the input tensor. Instead, most approaches involve searching for orthonormal $\mathbf{U}$, $\mathbf{V}$, $\mathbf{X}$ matrices and a $\mathcal{G}$ tensor that collectively minimize the decomposition error. For example, the *High Order SVD* approach to Tucker decomposition first identifies the left eigenvectors (with the highest eigenvalues) of the horizontal, lateral, and frontal slices to construct $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{X}$.

Many of the algorithms for decomposing tensors are based on an iterative process that approximates the best solution until a convergence condition is reached. The *alternating least squares* (ALS) method is relatively old and has been successfully applied to the problem of tensor decomposition [19, 34]. ALS estimates, at each iteration, one factor matrix, maintaining other matrices fixed; this process is repeated for each factor matrix associated to the dimensions of the input tensor. Non-iterative approaches to tensor decomposition include closed form solutions, such as generalized rank annihilation method (GRAM) [65] and direct trilinear decomposition (DTLD) [66], which fit the model by solving a generalized eigenvalue problem.

Dynamic Tensor Analysis (DTA). While the CP/Tucker decompositions are static tensor decompositions, there are dynamic tensor decompositions, such as Dynamic Tensor Analysis (DTA) [70]. When the data tensors are updated frequently, incremental tensor techniques, which dynamically maintain and revise the tensor decomposition are commonly used to avoid the cost of decomposing the data tensor from scratch with each update [70].

Nonnegative Tensor Decomposition (NTF). Tensor decompositions can be interpreted probabilistically, if additional constraints (nonnegativity and summation to 1) are imposed. In the case of the CP decomposition, for example, each nonzero element in the core can be thought of as a cluster and the values of entries of the factor matrices can be interpreted as the conditional probabilities of the entries given clusters. FacetCube [21] is a framework that extends the existing nonnegative tensor factorizations using probabilistic interpretation incorporated by users' prior knowledge. In [48], the nonnegative tensor factorization model is used as probability distributions to detect community structures in multi-dimensional social network data.

For a software tool to solve NTF, the N-way Toolbox for MATLAB [9] provides both CP and Tucker decompositions with the nonnegativity constraints.

Scalable Tensor Decomposition. Tensor decomposition is a costly process. In dense tensor representation, the cost increases exponentially with the number of modes of the tensor. While decomposition cost increases more slowly (linearly with the number of nonzero entries in the tensor) for sparse tensors, the operation can still be very expensive for large data sets.

[73] uses randomized sampling to approximate the tensor decomposition where the tensor does not fit in the available memory. A modified ALS algorithm proposed in [59] computes Hadamard products instead of Khatri-Rao products for efficient PARAFAC for large-scale tensors. [47] developed a greedy PARAFAC algorithm for large-scale, sparse tensors in MATLAB. [60] proposed a fast approach for CP that decomposes an unfolded tensor in lower order, instead of directly factorizing the high order tensor. [45] proposed a memory-efficient Tucker (MET) decomposition to address the intermediate blowup problem in Tucker decomposition. According to the ALS method for solving Tucker Decomposition, the bottleneck computation is the input tensor $\boldsymbol{\mathcal{X}}$ of size $I_1 \times I_2 \times \cdots \times I_N$ times factor matrices $\mathbf{A}^{(n)}$ of size $I_n \times r_n$ for $n = 1, ..., N$,

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{A}^{(1)} \cdots \times_{(n-1)} \mathbf{A}^{(n-1)} \times_{(n+1)} \mathbf{A}^{(n+1)} \cdots \times_N \mathbf{A}^{(N)}.$$

Since the intermediate result of a sparse tensor multiplied by factor matrices can be dense, intermediate results may be too big to fit in the available memory, even when the final result $\boldsymbol{\mathcal{Y}}$, whose size is $\max_n(I_n \prod_{m \neq n} r_m)$ may easily fit. MET addresses this problem by calculating $\boldsymbol{\mathcal{Y}}$ in an element-wise manner for reducing the size of intermediate memory. Instead of updating the whole $\boldsymbol{\mathcal{Y}}$, MET updates a subset of the modes (e.g., each slice $\mathbf{Y}_{:j_2:}$ or fiber $\mathbf{y}_{:j_2j_3}$) As a result, the size of intermediate result of MET is $\prod_{m \notin \varepsilon} I_m$, where ε is a subset of modes computed element-wise.

Parallel Tensor Decomposition. [73] proposed a randomized Tucker decomposition algorithm, MACH, which is parallelizable. Phan *et al.* [59] proposed a modified ALS PARAFAC algorithm called grid PARAFAC for large scale tensor data. The grid PARAFAC divides a large tensor into sub-tensors that can be factorized using any available PARAFAC algorithm in a parallel manner and iteratively combines into the final decomposition. The grid PARAFAC can be converted to grid NTF by enforcing nonnegativity.

[78] parallelized NTF by dividing a given original 3-mode tensor into three semi-non negative matrix factorization problems. These matrices are distributed to independent processors to facilitate parallelization. [10] presented an algorithm for NTF that is specialized for Compute Uniform Device Architecture (CUDA) parallel computing framework.

Note that since these block-based parallel algorithms are based on ALS where one variable can be optimized given that the other variables are fixed, the communication cost among the blocks is not avoidable. In the proposed parallelized optimization strategies in this thesis, on the other hand, each block is completely separable and run independently.

GigaTensor [39] employed the MapReduce framework to address the intermediate memory blow-up problem in PARAFAC and run large-scale tensor decomposition. [58] proposed a highly parallelizable tensor decomposition algorithm, which produces sparse approximation of tensor decompositions.

2.3 Challenges in Tensor-based Data Representation and Analysis

Thanks to their simplicity in modeling high-dimensional data and the availability of various mathematical tools (such as tensor decompositions) that support multi-aspect analysis of such data, tensors are increasingly being used in many application domains including scientific data management [6, 9, 17, 19, 34, 59, 74, 78], sensor data management [70, 73], and social network data analysis [13, 21, 28, 45, 47, 48, 49, 52, 71]. In data (such as text, image, and video) with ordered attribute domains, tensors are natural since the definition of a cell neighbor is clear. This becomes important for data analysis operations, such as convolution to support data filtering and summarization. Similarly, data clustering and compression which rely on the neighborhood definition are easier to express over tensors. Because of these properties, tensors have emerged as useful representations for analysis of multi-dimensional data. Especially for social network data analysis, tensor-based representations have proven to be useful for modeling the multiple aspects of the data to capture high-order structures for recommendation systems [52, 71], community discovery [13, 21, 45, 48, 49], and web link analysis [28, 47] in a tensor-based framework. For example, social network data consists of multiple types of objects (e.g., users, documents, tags) and their relationships (e.g., friend, follow, post) and tensors can be used to conveniently represent the relationships between different types of entities.

Spectral domain data analysis and data processing (such as spectral domain manipulation, analysis, and indexing [25, 38]) are commonly used techniques in domains, such as text, image, and video processing, where the data matrix shows significant degrees of redundancy. Spectral analysis of tensor data is often preceded by a tensor decomposition operation, which involves partitioning a large tensor into a smaller core tensor (i.e., spectral coefficients) and factor matrices (i.e., basis matrices) that represent different facets of the data for multi-aspect analysis: each factor matrix describes one specific aspect of the data, whereas the core tensor describes the strength (e.g., amount of correlation) of the relationships between these distinct data dimensions. For example, nonnegative CANDECOMP/PARAFAC (CP) tensor decomposition is often used for cluster analysis: the core tensor represents the weights (or the relative strengths) of the clusters and each entry, U_{ij} (normalized to between 0 and 1), of the factor matrix, $\mathbf{U}$, can be seen as the conditional probability of the given attribute value to belong to the corresponding cluster; i.e., $P(U_i|C_j)$ which is the conditional probability of the ith element of mode $\mathbf{U}$ given the jth cluster, C_j.

Many data-intensive applications, such as social network systems, where these tensor operations are used for the data analysis, handle large amounts of data. In addition to being large-scale, another characteristic of such data is multi-dimensional. Multi-aspect analysis for such data using tensor operations, such as tensor decomposition, has high computing costs.

Therefore, tensor decomposition operation is often prohibitive when the tensor data have a large number of modes:

- One obvious problem is the space needed to hold the input tensors. When the tensor is *dense* (i.e., has a large number of nonzero entries) or when a dense tensor representation is used for algorithmic reasons, the space required to hold the data increases exponentially with the number of modes. This renders the decomposition process expensive as all the data needed to perform the operation often do not fit in the main memory.

- The Tucker decomposition may in fact be infeasible for large data sets (even if the original tensor is sparse), since the tensors needed to represent intermediate results are often dense. As a result, the memory can overflow even when the memory is sufficient to store the input and the output tensors [45].

While tensor decomposition is relatively cheaper when the tensor is sparse (linear with the number of nonzero entries for CP decomposition [47]), the operation can still be prohibitively expensive when the data sets are large data. Unfortunately, recent attempts to parellelize tensor decomposition [10, 59, 78] also face difficulties, including large synchronization and data exchange overheads and (while there are some initial solutions for the CP decomposition) parallelizing Tucker decomposition is still a challenging task.

2.4 Functional Dependencies

A functional dependency (FD) is a constraint between two sets of attributes X and Y in a relation denoted by $X \rightarrow Y$, which specifies that the values of the X component of a tuple uniquely determine the values of the Y component.

The discovery of FDs in a data set is a challenging problem since the complexity increases exponentially in the number of attributes [54]. Many algorithms for FD and approximate FD discovery exist [36, 50, 54, 77]. TANE proposed in [36] used the definition of approximate FDs based on the minimum fraction of tuples that should be removed from the relation to hold the exact FDs.

The computation of FDs in TANE [36] and [50] is based on levelwise search [55]. Dep-Miner [50] finds the minimal FD cover of a hypergraph using a levelwise search. Similarly to Dep-Miner, FastFD [77] finds the minimal cover, however, differently from Dep-Miner, it uses a depth-first search that addresses the problem in a levelwise approach which increases the cost exponentially in the number of attributes. The main factor in the cost of FastFD is the input size. FastFD works well when the number of attributes is large. TANE takes linear time with respect to the size of the input whereas FastFD takes more than linear time of the input size.

CORDS [37] generalized FDs to determine statistical dependencies, which is referred to as soft FD. In a soft FD, a value of an attribute determines a value of another attribute with high probability. CORDS only discovers pairwise correlations reducing a great amount of complexity that nevertheless can remove most of correlation-induced selectivity error. In this thesis, we also leverage pairwise FDs to measure dependency *between* partitions (interFD) and *within* a partition (intraFD).

2.5 Array Databases

There are several in-database data models for modeling tensor data. Column-oriented organizations [69] are efficient when many or all rows are accessed, such as during an aggregate computation. Row-oriented organizations, on the other hand, are efficient when many or all of the columns on a single row are accessed or written on a single disk seek. Key-value organizations [3] are useful when working with less structured data, such as documents, which tend not to be relational. The array model [14, 18, 27, 75] is a natural representation to store multidimensional data and facilitate multidimensional data analysis. How arrays are organized and stored depends largely on whether they are dense or sparse. Approaches to represent array based data can be broadly categorized into four types. (a) The first approach is to represent the array in the form of a table: e.g., a 2D array $A[row, column]$ can be represented using a relational schema $(row, column, value)$ [75] or, if the model allows vector data types, as (row, row_vector) [23]. (b) A second approach is to use blob type in a relational database as a storage layer for array data [14, 27]. (c) Sparse matrices can also be represented using a graph-based abstraction [51]. For example, in [51], ALS (alternating least squares) is solved using a graph algorithm that represents a sparse matrix as a bipartite graph. (d) The last approach is to consider a native array model and an array-based storage scheme, such as a chunk-store, as in [18].

Chapter 3

TENSOR-RELATIONAL MODEL (TRM)

3.1 Introduction

Multi-dimensional data have various representations. Let $\mathrm{A}_1, \ldots, \mathrm{A}_n$ be a set of attributes in a relation and $\mathrm{D}_1, \ldots, \mathrm{D}_n$ be the attribute domains. The *relational model* [22] represents the data as sets of tuples, where each tuple is an instance in $\mathrm{D}_1 \times \ldots \times \mathrm{D}_n$; the model also encodes the functional dependencies between the attributes. The *vector model* [64] maps each attribute to a dimension in an n-dimensional space and represents each tuple as a point in this space (a natural representation when attributes are totally ordered). The *tensor model*, on the other hand, maps each attribute to a mode in an n-dimensional array where each possible tuple is a cell, the existence (absence) of a particular tuple in a database instance can be denoted as 1 (0) in the cell; similarly, the model can also represent fuzzy or probabilistic tuples by filling the cells with values between 0 and 1.

3.2 Tensor-based vs. Relational Data Manipulation

We can manipulate multidimensional data in several ways, including tensor algebra [46] and relational algebra [22]. Tensor algebra includes operators, such as addition, multiplication of a mode with a vector, multiplication of a mode with a matrix, inner product of tensors, and the norm of a tensor. Relational algebra, on the other hand, manipulates relational data using operators such as projection, selection, Cartesian-product, and set operators such as union and intersection. The difference between the two algebras is that tensor algebra focuses on manipulation

Relational operation	Tensor manipulation
Select	Slicing of a tensor (or taking a single or subset of elements across a given mode)
Project	Creating a sub-cube with a smaller set of modes
Cartesian-Product and Equi-Join	Composition of multiple tensors through outer-product
Union	Cell-wise OR (and row/slice insertion)
Intersection	Cell-wise AND (and row/slice elimination)

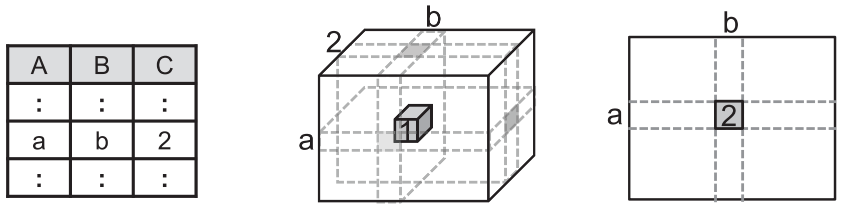
Table 3.1: Implementation of relational operations through tensor manipulation

of tensors, such as norms and inner products, whereas relational algebra focuses on set operations, such as union and intersection or operations on the attributes such as projection and joins. Therefore, *data management systems increasingly need to support both tensor-algebraic operations (for analysis) as well as relational-algebraic operations (for data manipulation and integration – Figure 1.1).*

3.3 Tensor-based Relational Data Model (TRM)

Common tensor operations (such as scalar addition/multiplication and tensor addition/multiplication) are well understood. While, logically, many relational algebraic operators can be implemented by manipulating tensors (Table 3.1), there is little prior research on efficient implementation of complex and semantically-rich data operations, such as *joins*, in conjunction with tensor analysis operations, such as *decompositions*. Therefore, in this thesis, we first introduce a tensor-based relational data model (TRM) and define tensor-relational algebraic operations on this model.

As we mentioned earlier, tensors have been used for representing and manipulating relational data. For example, [13] presented a multi-way clustering framework which operates on relational data represented in the form of multi-mode tensor. Most existing works assume that the available data has been pre-integrated into a single multi-mode tensor, which can then be manipulated using tensor operations. In practice, however, data rarely exists in a pre-integrated form and its lifecycle (from collection



(a) relation instance (b) occurrence tensor (c) value tensor

Figure 3.1: (a) A sample relation, (b) the occurrence tensor, and (c) the value tensor (assuming that the attribute set $\{\mathrm{A}, \mathrm{B}\}$ is a candidate key for the relation)

to analysis) involves various integration and other manipulation steps (Figure 1.1). In this section, we present a tensor-relational model (TRM) for data represented as tensors.

3.3.1 Types of Tensors Representing Relations

Let $\mathrm{A}_1, \ldots, \mathrm{A}_n$ be a set of attributes in the schema of a relation, R, and $\mathrm{D}_1, \ldots, \mathrm{D}_n$ be the attribute domains. Let the relation instance $\mathcal{R}$ be a finite multi-set of tuples, where each tuple $t \in \mathrm{D}_1 \times \ldots \times \mathrm{D}_n$.

Occurrence Tensor. We define an *occurrence tensor* $\boldsymbol{\mathcal{R}}_o$ corresponding to the relation instance $\mathcal{R}$ as an n-mode tensor, where each attribute $\mathrm{A}_1, \ldots, \mathrm{A}_n$ is represented by a mode. For the ith mode, which corresponds to A_i, let $\mathrm{D}'_i \subseteq \mathrm{D}_i$ be the (finite) subset of the elements such that

$$\forall v \in \mathrm{D}'_i \; \exists t \in \mathcal{R} \;\; s.t. \;\; t.\mathrm{A}_i = v$$

and let $idx(v)$ denote the rank of v among the values in D'_i relative to an (arbitrary) total order, $<_i$, defined over the elements of the domain, D_i. The cells of the *occurrence tensor* $\boldsymbol{\mathcal{R}}_o$ are such that

$$\boldsymbol{\mathcal{R}}_o[u_1, \ldots, u_n] = 1 \leftrightarrow \exists t \in \mathcal{R} \;\; s.t. \;\; \forall_{1 \leq j \leq n} \, idx(t.\mathrm{A}_j) = u_j$$

and 0 otherwise. Intuitively, each cell indicates whether the corresponding tuple exists in the multi-set corresponding to the relation or not (Figures 3.1(a) and (b)).

Counting Tensor. Note that the relation instance $\mathcal{R}$ is a finite *multi-set* of tuples; i.e., there can be two tuples, t_a and t_b, in $\mathcal{R}$ such that $\forall_{1 \leq j \leq n}\, t_a.\mathrm{A}_j = t_b.\mathrm{A}_j$. We define the corresponding n-mode *counting tensor*, $\boldsymbol{\mathcal{R}}_c$, such that

$$\boldsymbol{\mathcal{R}}_c[u_1, \ldots, u_n] = |\{t \in \mathcal{R} \mid \forall_{1 \leq j \leq n}\, idx(t.\mathrm{A}_j) = u_j\}|$$

Intuitively, each cell counts the number of corresponding tuples in the multi-set corresponding to the relation.

Value Tensor. Let again $\mathrm{A} = \{\mathrm{A}_1, \ldots, \mathrm{A}_n\}$ be the set of attributes in the schema of the relation, R. In the relational model, a *candidate key* of the relation $\mathcal{R}$ is defined as a subset, K, of the attributes that uniquely determines the tuple. More formally,

$$\forall t_a, t_b \in \mathcal{R}\, (\forall_{\mathrm{A}_i \in \mathrm{K}} t_a.\mathrm{A}_i = t_b.\mathrm{A}_i) \;\rightarrow\; (\forall_{\mathrm{A}_i \in \mathrm{A}} t_a.\mathrm{A}_i = t_b.\mathrm{A}_i)\,.$$

Given a relation $\mathcal{R}$ with an attribute set $\mathrm{A} = \{\mathrm{A}_1, \ldots, \mathrm{A}_n\}$ and a candidate key $\mathrm{K} = \{\mathrm{A}_{\mathrm{K}(1)}, \ldots, \mathrm{A}_{\mathrm{K}(m)}\} \subset \mathrm{A}$, let $X = \{\mathrm{A}_{\mathrm{X}(1)}, \ldots, \mathrm{A}_{\mathrm{X}(n-m)}\}$ denote the set of remaining attributes; i.e., $(\mathrm{X} \cup \mathrm{K} = \mathrm{A}) \;\wedge\; (\mathrm{X} \cap \mathrm{K} = \emptyset)$. Then, for this relation, we define the corresponding *value tensor* as an m-mode tensor, $\boldsymbol{\mathcal{R}}_v$, such that

$$\begin{aligned}\boldsymbol{\mathcal{R}}_v[u_1, \ldots, u_m] = \langle v_1, \ldots, v_{n-m}\rangle \;\; s.t. \\ \exists t \in \mathcal{R}\; (\forall_{\mathrm{A}_{\mathrm{K}(i)} \in \mathrm{K}}\, idx(t.\mathrm{A}_{\mathrm{K}(i)}) = u_i \wedge \forall_{\mathrm{A}_{\mathrm{X}(j)} \in \mathrm{X}}\, t.\mathrm{A}_{\mathrm{X}(j)} = v_j).\end{aligned}$$

If K = A, then the value tensor is not defined. Intuitively, in this case, each mode corresponds to an attribute in the candidate key of the relation and each cell represents the values of the attributes determined by the corresponding instance of the candidate key. Figure 3.1(c) presents an example.

Tensor Conversion. A counting or value tensor can be converted into an occurrence tensor by adding an additional mode, which represents the *count* in the counting tensor or the *value* in the value tensor, respectively. Let $\boldsymbol{\mathcal{P}}$ be a counting or value tensor; the mapping $occ(\boldsymbol{\mathcal{P}})$ gives the corresponding occurrence tensor. Similarly, given a candidate key K, an occurrence tensor, $\boldsymbol{\mathcal{P}}$, can be converted into a value tensor, $val(\boldsymbol{\mathcal{P}}, \mathrm{K})$.

3.3.2 Tensor Relational Algebra

Next, we introduce tensor relational algebra operations to manipulate relations represented as tensors. Let $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$ be two tensors, representing relation instances $\mathcal{P}$ and $\mathcal{Q}$, with attribute sets, $\mathrm{A}^P = \{\mathrm{A}_1^P, \ldots, \mathrm{A}_n^P\}$ and $\mathrm{A}^Q = \{\mathrm{A}_1^Q, \ldots, \mathrm{A}_m^Q\}$, respectively. In the rest of this section, we denote the index of each cell of $\boldsymbol{\mathcal{P}}$ as $(i_1, i_2, ..., i_n)$; similarly, the index of each cell of $\boldsymbol{\mathcal{Q}}$ is denoted as $(j_1, j_2, ..., j_m)$. The cell indexed as $(i_1, \ldots, i_n)$ of $\boldsymbol{\mathcal{P}}$ is denoted by $\boldsymbol{\mathcal{P}}[i_1, \ldots, i_n]$ and the cell indexed as $(j_1, \ldots, j_m)$ of $\boldsymbol{\mathcal{Q}}$ is denoted by $\boldsymbol{\mathcal{Q}}[j_1, \ldots, j_m]$.

Selection (σ). In relational algebra, the *selection* operation is an operation which takes as input a single relation and a condition, φ, and returns all the tuples in the relation satisfying the given condition:

$$(t \in \sigma_\varphi(\mathcal{P})) \;\leftrightarrow\; (t \in \mathcal{P}) \wedge \varphi(t).$$

Given an occurrence or counting tensor $\boldsymbol{\mathcal{P}}$ and a selection condition, φ, we define the *condition tensor*, $\boldsymbol{\mathcal{C}}_\varphi$, as a tensor of the same dimensions as $\boldsymbol{\mathcal{P}}$, such that for $i_1, i_2, ..., i_n$, if $\varphi(i_1, i_2, ..., i_n)$, then $\boldsymbol{\mathcal{C}}_\varphi[i_1, i_2, ..., i_n] = 1$ and $\boldsymbol{\mathcal{C}}_\varphi[i_1, i_2, ..., i_n] = 0$, otherwise. Given the condition tensor, $\boldsymbol{\mathcal{C}}_\varphi$, the tensor selection operation for the given occurrence or counting tensor $\boldsymbol{\mathcal{P}}$ is defined as

$$\sigma_\varphi(\boldsymbol{\mathcal{P}}) \overset{\text{def}}{=} comp(\boldsymbol{\mathcal{P}} * \boldsymbol{\mathcal{C}}_\varphi),$$

where $*$ is the cell-wise product of $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{C}}_\varphi$ and *comp*() is the compaction operator which eliminates all-0 slices from the resulting tensor to ensure that the elements along all modes correspond to attribute values that occur in at least one tuple satisfying the selection condition.

If $\boldsymbol{\mathcal{P}}$ is a value tensor with a candidate key K, on the other hand, the selection operation is defined as

$$\sigma_\varphi(\boldsymbol{\mathcal{P}}) \stackrel{\text{def}}{=} val(comp(occ(\boldsymbol{\mathcal{P}}) * \boldsymbol{\mathcal{C}}_\varphi), \mathrm{K}).$$

Projection (π). In relational algebra, the *projection* operation takes as input a single relation $\mathcal{R}$ with an attribute set, $\mathrm{A}^P = \{\mathrm{A}_1^P, \ldots, \mathrm{A}_n^P\}$, and an attribute set $\mathrm{A} = \{\mathrm{A}_{a_1}, \ldots, \mathrm{A}_{a_k}\} \subseteq \mathrm{A}^P$ and maps the relation into a new relation $\pi_\mathrm{A}(\mathcal{R})$ with the attribute set A such that

$$\forall t \in \mathcal{R}\ \exists t' \in \pi_\mathrm{A}(\mathcal{R})\ \ s.t.\ \ \forall_{\mathrm{A}_{a_h} \in \mathrm{A}}\ t'.\mathrm{A}_{a_h} = t.\mathrm{A}_{a_h}.$$

The corresponding tensor projection operator eliminates all the modes that do not belong to the target attribute set A; i.e., given $\boldsymbol{\mathcal{P}}$ with an attribute set, A^P, and the projection attribute set $\mathrm{A} \subseteq \mathrm{A}^P$, the result is a new tensor $\pi_\mathrm{A}(\boldsymbol{\mathcal{P}})$ with the attribute set A. More specifically, if $\boldsymbol{\mathcal{P}}$ is an occurrence tensor, then

$$\pi_\mathrm{A}(\boldsymbol{\mathcal{P}})[i_{a_1}, ..., i_{a_k}] = 1 \ \leftrightarrow\ \exists \boldsymbol{\mathcal{P}}[\ldots, i_{a_1}, \ldots, i_{a_k}, \ldots] = 1.$$

On the other hand, if $\boldsymbol{\mathcal{P}}$ is a counting tensor, then

$$\pi_\mathrm{A}(\boldsymbol{\mathcal{P}})[i_{a_1}, ..., i_{a_k}] = \sum_{(\ldots, i_{a_1}, \ldots, i_{a_k}, \ldots)} \boldsymbol{\mathcal{P}}[\ldots, i_{a_1}, \ldots, i_{a_k}, \ldots].$$

As before, if $\boldsymbol{\mathcal{P}}$ is a value tensor, we can define the projection by first converting it into an occurrence tensor:

$$\pi_\mathrm{A}(\boldsymbol{\mathcal{P}}) = val(\pi_\mathrm{A}(occ(\boldsymbol{\mathcal{P}})), \mathrm{K}).$$

Cartesian product ($\times$). Given two relations $\mathcal{P}$ and $\mathcal{Q}$, with attribute sets, $\mathrm{A}^P = \{\mathrm{A}_1^P, \ldots, \mathrm{A}_n^P\}$ and $\mathrm{A}^Q = \{\mathrm{A}_1^Q, \ldots, \mathrm{A}_m^Q\}$, the *cartesian product* operator returns a new relation, $\mathcal{P} \times \mathcal{Q}$, with an attribute set $\{\mathrm{A}_1^P, \ldots, \mathrm{A}_n^P, \mathrm{A}_1^Q, \ldots, \mathrm{A}_m^Q\}$:

$$t \in \mathcal{P} \times \mathcal{Q} \ \leftrightarrow \exists_{t_1 \in \mathcal{P}} \exists_{t_2 \in \mathcal{Q}} \ t = \mathit{concatenate}(t_1, t_2).$$

If we consider two occurrence or counting tensors, $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$, as inputs, we can define the tensor relational algebraic cartesian product simply in terms of the outer product ($\otimes$) of the two input tensors:

$$\boldsymbol{\mathcal{P}} \times \boldsymbol{\mathcal{Q}} \overset{\text{def}}{=} \boldsymbol{\mathcal{P}} \otimes \boldsymbol{\mathcal{Q}}.$$

As before, for value tensors, we can define the cartesian product by first converting the tensors into occurrence tensors:

$$\boldsymbol{\mathcal{P}} \times \boldsymbol{\mathcal{Q}} = \mathit{val}((\mathit{occ}(\boldsymbol{\mathcal{P}}) \times \mathit{occ}(\boldsymbol{\mathcal{Q}})), K_P \cup K_Q).$$

Join ($\bowtie$). In relational algebra, given two relations $\mathcal{P}$ and $\mathcal{Q}$, and a condition φ, the *join* operation is defined as a cartesian product of the input relations followed by the selection operation. Therefore, given two relational tensors $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$, and a condition φ, we can define their join as

$$\boldsymbol{\mathcal{P}} \bowtie_{\varphi} \boldsymbol{\mathcal{Q}} \overset{\text{def}}{=} \sigma_{\varphi}(\boldsymbol{\mathcal{P}} \times \boldsymbol{\mathcal{Q}}). \tag{3.1}$$

Given two relations $\mathcal{P}$ and $\mathcal{Q}$, with attribute sets, $\mathrm{A}^P = \{\mathrm{A}_1^P, \ldots, \mathrm{A}_n^P\}$ and $\mathrm{A}^Q = \{\mathrm{A}_1^Q, \ldots, \mathrm{A}_m^Q\}$, and a set of attributes $\mathrm{A} \subseteq \mathrm{A}^P$ and $\mathrm{A} \subseteq \mathrm{A}^Q$, the *equi-join* operation, $\bowtie_{=,\mathrm{A}}$, is defined as the join operation, with the condition that matching attributes in the two relations will have the same values, followed by a projection operation that eliminates one instance of A from the resulting relation. While this equi-join operation can be implemented using the outer product based definition in Equation (3.1), the cost of the outer product operation for high-order tensors can be prohibitive.

Therefore we also consider a more efficient version of the equi-join operator using inner products: Let $\mathcal{P}$ of size $I_1 \times I_2 \times \cdots \times J \times \cdots \times I_{N_p}$ and $\mathcal{Q}$ of size $I'_1 \times I'_2 \times \cdots \times J \times \cdots \times I'_{N_q}$ be two occurrence tensors we want to equi-join on the join mode $\mathbf{J}$ whose size is J. Let us consider an augmented tensor $\mathcal{P}'$ of size $I_1 \times I_2 \times \cdots \times J \times J \times \cdots \times I_{N_p}$ from $\mathcal{P}$ by replicating the mode $\mathbf{J}$:

$$\forall_{k=1\ldots J} \forall_{k'=k}\ \mathcal{P}'[\ldots, k, k', \ldots] = \mathcal{P}[\ldots, k, \ldots].$$

This duplicated mode becomes the mode $\mathbf{J}$ of the joined tensor after the mode $\mathbf{J}$ of each tensor is removed by an inner product of the two tensors in the mode $\mathbf{J}$. Given this, the equi-join operation can be defined as

$$\forall_{k'=1\ldots J}\ \mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}[\ldots, k', \ldots] = \sum_{k=1}^{J} \mathcal{P}'[\ldots, k, k', \ldots]\mathcal{Q}[\ldots, k, \ldots],$$

which can be implemented as an inner product operation for sparse tensors involving the cost of sorting all the nonzero entries of $\mathcal{P}$ and $\mathcal{Q}$ thus $O((|\mathcal{P}| + |\mathcal{Q}|) \log(|\mathcal{P}| + |\mathcal{Q}|))$ where $|\mathcal{P}|$ and $|\mathcal{Q}|$ are the number of nonzero entries of $\mathcal{P}$ and $\mathcal{Q}$ respectively [11] [1] .

Union ($\cup$).

$$\mathcal{P} \cup \mathcal{Q} = \{t | t \in \mathcal{P} \text{ or } t \in \mathcal{Q}\}.$$

Intersection ($\cap$).

$$\mathcal{P} \cap \mathcal{Q} = \{t | t \in \mathcal{P} \text{ and } t \in \mathcal{Q}\}.$$

Set difference ($-$).

$$\mathcal{P} - \mathcal{Q} = \{t | t \in \mathcal{P} \text{ and } t \notin \mathcal{Q}\}.$$

[1] In the implementation, we use the sparse tensor inner product operator available in the MATLAB Tensor Toolbox [12]. Although details are not presented in this thesis, this provides a performance similar to the equi-join operations implemented using the MySQL DBMS.

CP Decomposition. The rank-r CP Decomposition of the tensor $\boldsymbol{\mathcal{P}}_{I_1\times I_2\times\cdots\times I_N}$ can be defined as

$$CP(\boldsymbol{\mathcal{P}}_{I_1\times I_2\times\cdots\times I_N}) = \tilde{\boldsymbol{\mathcal{P}}}_{I_1\times I_2\times\cdots\times I_N} = \langle\lambda, \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\rangle,$$

such that

$$\boldsymbol{\mathcal{P}}_{I_1\times I_2\times\cdots\times I_N} \approx \sum_{k=1}^{r} \lambda_k \circ U_k^{(1)} \circ U_k^{(2)} \circ \cdots \circ U_k^{(N)}, \tag{3.2}$$

where λ_i is the ith element of vector λ of size r and $U_i^{(n)}$ is the ith column vector of the matrix $\mathbf{U}^{(n)}$ of size $I_n \times r$, for $n = 1, \cdots, N$.

Note that the CP decomposition operation is an approximate operation and $\boldsymbol{\mathcal{P}}$ may not be exactly reconstructed from $\tilde{\boldsymbol{\mathcal{P}}}$. In other words, the following weighted sum, $\hat{\boldsymbol{\mathcal{P}}}$, of the rank-one tensors may be different from $\boldsymbol{\mathcal{P}}$:

$$\hat{\boldsymbol{\mathcal{P}}}_{I_1\times I_2\times\cdots\times I_N} = \sum_{k=1}^{r} \lambda_k \circ U_k^{(1)} \circ U_k^{(2)} \circ \cdots \circ U_k^{(N)}. \tag{3.3}$$

Therefore, the norm of $\boldsymbol{\mathcal{P}}$ denoted by $\|\boldsymbol{\mathcal{P}}\|$ may also be different from $\|\hat{\boldsymbol{\mathcal{P}}}\|$. Note that $\|\hat{\boldsymbol{\mathcal{P}}}\|$ can be computed directly from the decomposition $\tilde{\boldsymbol{\mathcal{P}}}$ without having to reconstruct the tensor $\hat{\boldsymbol{\mathcal{P}}}$:

$$\|\hat{\boldsymbol{\mathcal{P}}}\|^2 = \|\tilde{\boldsymbol{\mathcal{P}}}\|^2 = \lambda^T(\mathbf{U}^{(N)T}\mathbf{U}^{(N)} * \cdots * \mathbf{U}^{(1)T}\mathbf{U}^{(1)})\lambda. \tag{3.4}$$

Chapter 4

TENSORDB: TENSOR-RELATIONAL DATA MANAGEMENT SYSTEM

4.1 Introduction

Today's data management systems increasingly need to support both tensor-algebraic operations (for analysis) as well as relational-algebraic operations (for data manipulation and integration – Figure 1.1). Based on tensor-relational data model that we defined in the previous chapter, we build such a data management system that supports tensor-relational operations, so called, *TensorDB*. We propose two types of TensorDB, in-memory and in-database based TensorDB.

TensorDB supports tensor-relational query plans and query optimization strategies for tensor-relational operations. Since the costliest operation in TRM is tensor decomposition among both relational and tensor operations, we focus on developing optimization strategies for the tensor decomposition operations in the in-memory TensorDB.

In-database TensorDB is to address the in-memory limitation of MATLAB-based implementation of tensor-relational operations in the in-memory TensorDB. We focus on building the in-database implementation of these tensor-relational operations in an array database, SciDB [4], which is an open source software platform of data management and analytic system for array data. We also consider optimization strategies for efficient in-database tensor decomposition operations on disk-resident tensor data.

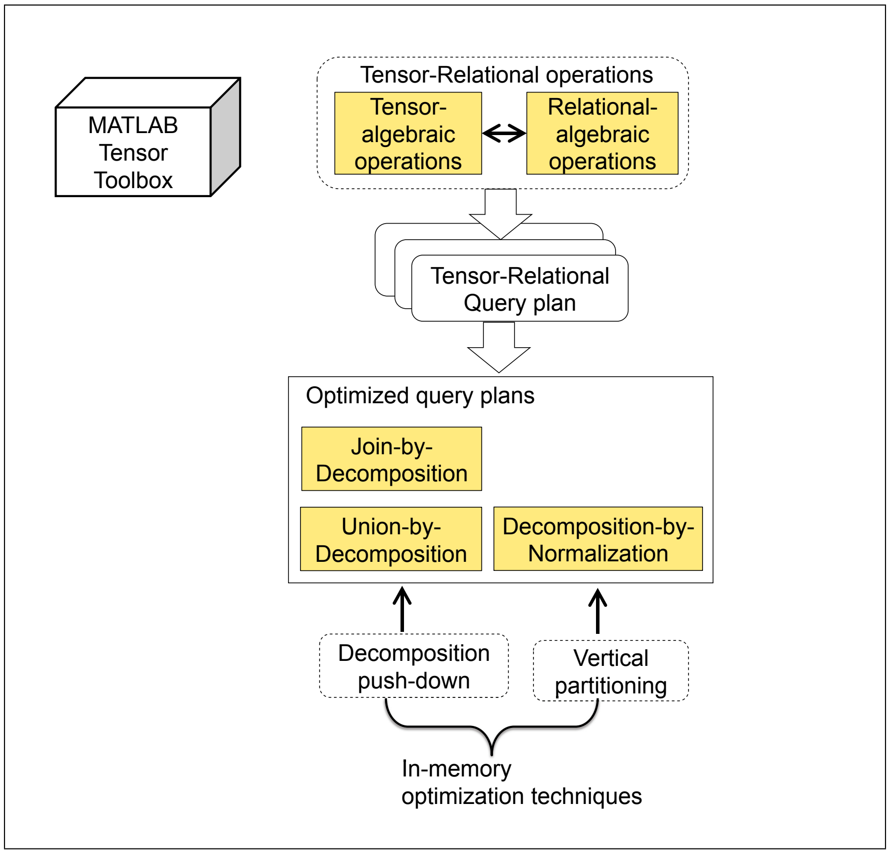
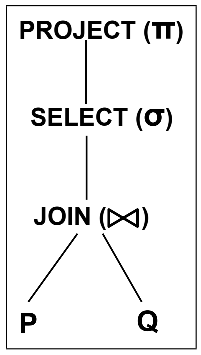
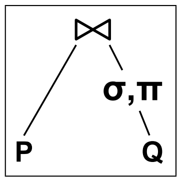

MATLAB Tensor Toolbox
Tensor-Relational operations
Tensor-algebraic operations
Relational-algebraic operations
Tensor-Relational Query plan
Optimized query plans
Join-by-Decomposition
Union-by-Decomposition
Decomposition-by-Normalization
Decomposition push-down
Vertical partitioning
In-memory optimization techniques

Figure 4.1: In-Memory TensorDB

4.2 In-Memory TensorDB

While, in traditional relational algebra, the costliest operation is known to be the join, in the TensorDB that provides both relational and tensor operations, tensor decomposition tends to be the computationally costliest operation. TensorDB involves complex query plans of both tensor-algebraic and relational-algebraic operations and when we run such a query plan where tensor decomposition operations are performed with data integration operations such as join and union, the data to be decomposed gets larger and the cost of tensor decomposition gets more expensive on this larger data. Therefore, the main optimization strategies we consider are to optimize query plans involving the tensor decompositions, especially, with the data integration operations.





(a) Logical query plan

(b) physical query plan

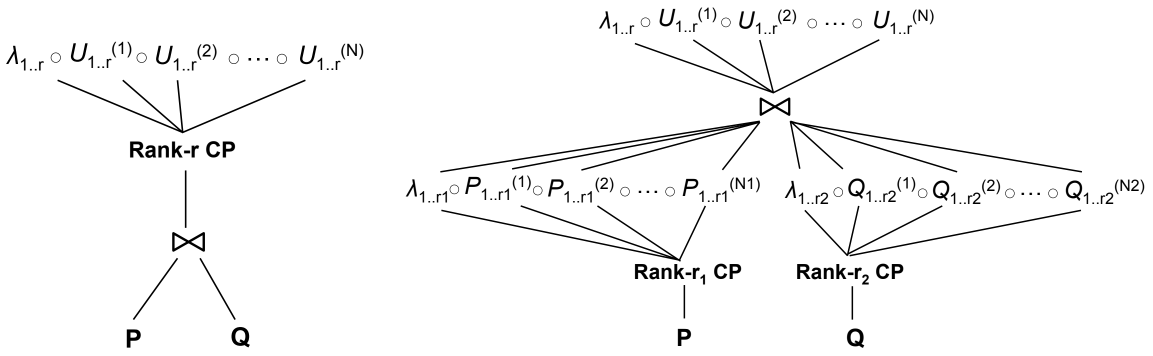
(selection, and projection are pushed-down)

Figure 4.2: Query optimization in relational algebra: (a) A logical query plan involving selection, projection, and join operations: (b) an equivalent physical plan where the selection and projection operations are *pushed-down* to minimize the amount of data fed into the join operator

In this section, we discuss these optimization strategies in the in-memory TensorDB. Figure 4.1 shows the in-memory optimization techniques and optimized query plans in the in-memory TensorDB.

4.2.1 Decomposition Push-Down Strategy for Optimizing TRM Workflows

In relational algebra, the costliest operation is the join operation. Consequently, given a complex query plan, the relational optimizers push-down data reduction operations, such as selections (which reduce the number of tuples) and projections (which reduce the number of data attributes) over join-operations to reduce the amount of data fed into the join operators (Figure 4.2). In TensorDB, however, tensor decomposition operation tends to be the computationally costliest operation: for dense tensors, the cost is exponential in the number of modes of the data. While the operation is relatively cheaper for sparse tensors, the cost and memory requirement still outweigh other more traditional relational operators. The cost of tensor decomposition relies on the number of non-zero entries and number of modes. Therefore, a key criterion for



λ1..r ○ U1..r(1) ○ U1..r(2) ○ ··· ○ U1..r(N)
Rank-r CP
⋈
P
Q
λ1..r ○ U1..r(1) ○ U1..r(2) ○ ··· ○ U1..r(N)
⋈
λ1..r1 ○ P1..r1(1) ○ P1..r1(2) ○ ··· ○ P1..r1(N1)
λ1..r2 ○ Q1..r2(1) ○ Q1..r2(2) ○ ··· ○ Q1..r2(N2)
Rank-r1 CP
Rank-r2 CP
P
Q

Figure 4.3: (a) A query plan with a join operation of two tensors, $\mathcal{P}$ and $\mathcal{Q}$, preceding a tensor decomposition operation and (b) an alternative query plan with two tensor decomposition operations followed by a join operation

optimizing query workplans in TensorDB is to reduce the number of data modes and non-zero data entries in the tensors that need to be decomposed. In TensorDB, we consider optimization strategies for complex queries involving tensor decomposition with tensor manipulation operations, particularly, join and union operations that integrate data from multiple sources since the data integration operations increase the cost of tensor decomposition.

Firstly, we consider query plans that involve join operations and tensor decompositions (Figures 4.3(a)) and propose a *decomposition push-down* strategy that reduces the number of modes of the data tensors being decomposed, which is referred to as *join-by-decomposition* (JBD). This *join-by-decomposition* (JBD) strategy *pushes-down* the tensor-decomposition operation so that the input tensors (which have smaller number of modes than the join tensor) are decomposed into their spectral components and then these decompositions are combined to obtain the final decomposition as shown in Figure 4.3(b).

Secondly, we focus on query plans that involve tensor decomposition and union operations (as in Figure 4.4(a)) and propose the second *decomposition push-down* strategy, so called, *union-by-decomposition* (UBD) strategy (as in Figure 4.4(b)) that help reduce the overall cost of the query plan. The query plan with decomposition

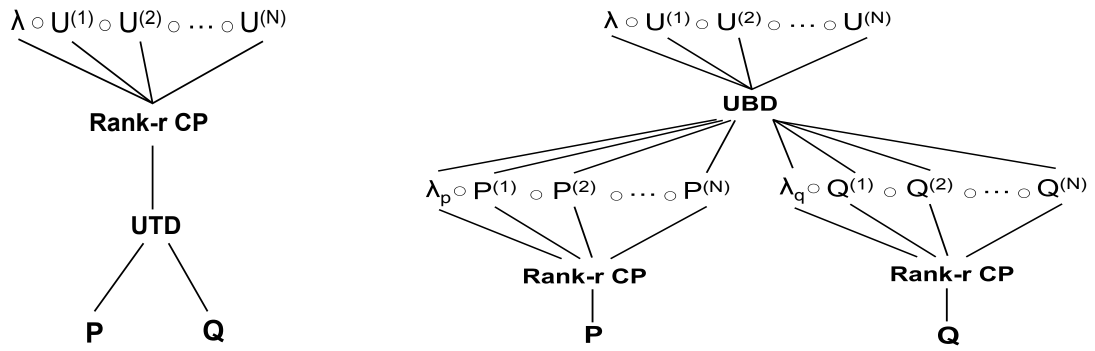

λ ○ U(1) ○ U(2) ○ … ○ U(N)
Rank-r CP
UTD
P
Q
λ ○ U(1) ○ U(2) ○ … ○ U(N)
UBD
λp ○ P(1) ○ P(2) ○ … ○ P(N)
λq ○ Q(1) ○ Q(2) ○ … ○ Q(N)
Rank-r CP
Rank-r CP
P
Q

Figure 4.4: (a) A query plan with an union of two tensors, $\mathcal{P}$ and $\mathcal{Q}$ preceding tensor decomposition and (b) an alternative query plan where the decomposition is pushed-down over union

push-down, which first performs the tensor decompositions on each input data source and then combines these decomposed tensors as the *union-by-decomposition* (UBD) plan.

4.2.2 Vertical Partitioning Strategy for Optimizing Tensor Decomposition Process

An optimization strategy to tackle the high computational cost of the tensor decomposition process is vertical partitioning. Since the number of modes of the tensor data is one of the main factors contributing to the cost of the tensor operations, we argue that if a tensor with large number of modes can be vertically partitioned into tensors with smaller number of modes and each sub-tensor is decomposed independently, then the resulting partial decompositions can be efficiently combined to obtain the decomposition of the original tensor. We propose *decomposition-by-normalization* (DBN) scheme as the vertical partitioning optimization strategy for the tensor decomposition operations.

4.3 In-Database TensorDB

We introduce the in-database TensorDB for efficient implementations of in-database tensor decompositions on chunk-based array data stores. In-database TensorDB extends an open source software platform of data management and analytic

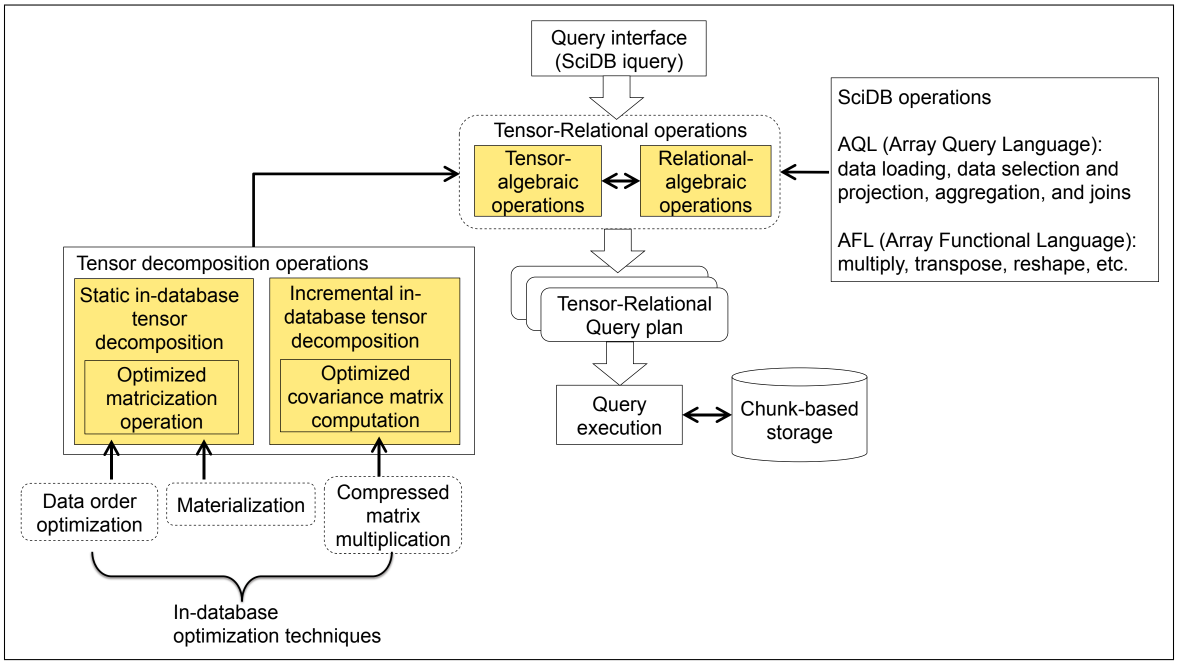

Query interface
(SciDB iquery)
Tensor-Relational operations
Tensor-algebraic operations
Relational-algebraic operations
SciDB operations
AQL (Array Query Language): data loading, data selection and projection, aggregation, and joins
AFL (Array Functional Language): multiply, transpose, reshape, etc.
Tensor decomposition operations
Static in-database tensor decomposition
Optimized matricization operation
Incremental in-database tensor decomposition
Optimized covariance matrix computation
Tensor-Relational Query plan
Query execution
Chunk-based storage
Data order optimization
Materialization
Compressed matrix multiplication
In-database optimization techniques

Figure 4.5: In-Database TensorDB

system for array data, SciDB. As an extension of SciDB, TensorDB shares the basic system architecture and the query languages of SciDB and performs all-in-one from the query interface and query optimization to the query execution for tensor-relational operations.

Leveraging the SciDB engine and SciDB languages, we develop tensor operations such as tensor decomposition and TensorDB supports tensor-relational query plans of tensor decomposition operations, along with relational operations such as selection and join operations.

While the in-database TensorDB can address the memory-limitations in in-memory TensorDB, in-database implementation for tensor operations on disk-resident data can bring in other challenges. We will discuss several optimization strategies in developing in-database tensor decomposition operations later in this section.

Figure 4.5 illustrates the query processing workflow of in-database TensorDB for tensor-relational query plans using tensor-algebraic and relational-algebraic opera-

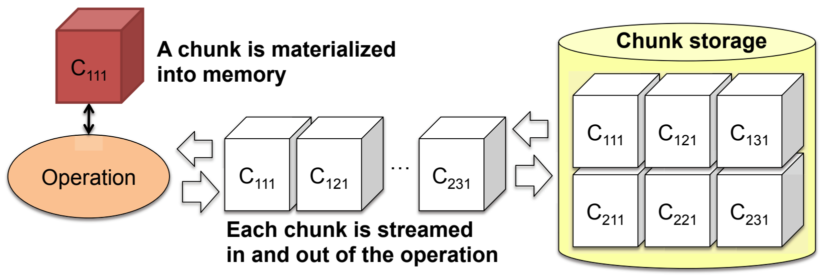


Figure 4.6: Pipelined chunk-based query processing in SciDB

tions and optimization strategies in in-database tensor decomposition operations. Before we introduce in-database TensorDB operations and optimization strategies, we first review the SciDB architecture and operations and how we leverage these to build the in-database TensorDB.

4.3.1 SciDB Preliminary

SciDB [4] is an open-source array-based DBMS. SciDB uses multidimensional arrays as its basic storage and processing unit. Arrays are partitioned into chunks and each chunk is processed in a parallel manner, whenever possible. Figure 4.6 illustrates the pipelined chunk-based query processing. Especially for data types, such as images, where nearby cells are correlated, SciDB stores/loads arrays in run-length encoding/decoding to leverage correlations in consecutive data elements. SciDB also provides various chunk-based array manipulation operations, including linear algebra operators.

These operations are provided by SciDB's two query language interfaces. The first one is AQL, the Array Query Language and the second is AFL, the Array Functional Language. SciDB's Array Query Language (AQL) is a high-level declarative language as the SQL for relational databases, providing operations such as data loading, data selection and projection, aggregation, and joins. SciDB's Array Functional Language (AFL) is a functional language for working with SciDB arrays. In order to issue commands of the AQL and AFL commands, SciDB provides a command-line query interface, `iquery` to run these operations. Below, we briefly review the relevant SciDB operators that we leverage to build the tensor-relational operations.

A new tensor is created using `CREATE ARRAY` command. When factor matrices, which are results of the tensor decomposition are initialized randomly, the `build` command is used.

• `CREATE ARRAY name <attributes> [dimensions];` This operator creates the template for an array with the specified name and schema (attributes and dimensions). For instance, `CREATE ARRAY A <val:double> [i=1:100,10,0, j=1:100,10,0]` creates an array template `A` that has one attribute named `val` of type `double` and two dimensions of length 100, chunk size 10, and chunk overlap [1] 0.

• `build(template_array|schema_def.,expression);` This operator produces an array with the shape of the given template, with values equal to the given expression. For example, `build(<val:double>[i=1:100,10,0,j=1:100,10,0], random())` produces an array that has one attribute named `val` of type `double` and two dimensions of length 100, chunk size 10, and chunk overlap 0, populated with random values. While `CREATE ARRAY` creates a template of an array, `build` populates the array with values defined the given expression.

To store the result from a `build` operator, we use `store` command.

• `store(operator(args),array);` This operator updates `array` with the result of the operation specified in `operator(args)`. The `store` operator creates a new version of the destination array (with all previous versions also maintained). The `store` operator utilizes run-length encoding to compress the array data.

SciDB's AQL Data Manipulation Language (DML) provides queries to access and operate on array data such as the AQL SELECT for selecting data from a SciDB

[1]Chunk overlap specifies the number of overlapping dimension-index values for adjacent chunks.

array.

• `SELECT expression [INTO target_array] FROM array_expression | src_array [WHERE expression];`

• `SELECT expression FROM src_array1, src_array2;` The inner join of src_array1 and src_array2.

There are operations reducing an array by taking some subsets of the data or concatenating arrays into an array.

• `slice(src_array,dimension1,values1[dimension2,value2,...]);` Return a subset of the source array on the value(s) of the given dimension(s).

• `subarray(array,low_coord1[,low_coord2,...], high_coord1[,high_coord2,...]);` Return a result array by selecting a contiguous area of cells of each dimension.

• `concat(left_array,right_array);` Concatenate two arrays.

Now, we review matrix operations.

• `multiply(left_array,right_array);` [2] This operator performs matrix multiplication of two input arrays, `left_array` and `right_array`, and returns a result array.

• `transpose(array);` This operator transposes the given `array`.

• `reshape(src_array,template_array|schema_def.);` This operator reshapes `src_array` with `template_array` or `schema_definition`. The `template_array` or `schema_definition` has the same number of cells as the `source_array`, but a different shape. For example, this can be used to transform a 3x4 array into a 6x2 array.

• `redimension(src_array,template_array|schema_def.);` This operator is used to re-arrange dimensions of `src_array` with `template_array` or `schema_definition`. Unlike `reshape`, it does not alter the dimension sizes, but it switches the dimension order.

[2]we use the `multiply` operator supported in SciDB 12.12.

4.3.2 Tensor Manipulation on Chunks

Currently, most array databases provide limited built-in array operations and leave the responsibility of implementing complex operations through user-defined functions (UDF) and aggregates (UDA) [23, 27, 30] to the users. One critical limitation of UDF/UDA-based approaches is that the data, such as coefficient vectors, should comfortably reside in the available memory [23, 30] and this is not always the case, in many tensor operations, such as tensor decomposition.

In building the in-database TensorDB, we describe how to extend a native array database, SciDB [18], with tensor manipulation operations; specifically we focus on in-database, chunk-based implementation of the operations needed to achieve tensor decomposition. Naturally, there are optimization and scalability issues in in-database implementation of tensor manipulation operations, including how we partition the data into chunks and how we move them in and out of the memory.

4.3.3 TensorDB Operators

TensorDB deals with complex query plans where relational operations run along with tensor operations. SciDB supports relational operations for data manipulation and integration such as selection, projection, join, etc. and linear algebra operations, such as transpose, multiply. However it lacks the tensor-algebraic operations such as tensor decomposition operations. Therefore in-database TensorDB focuses on building in-database implementation of tensor operations such as static and dynamic tensor decompositions.

For a static tensor decomposition, we consider an alternating least squares (ALS) based implementation of CP decomposition [19, 34]. While we leverage some of the operations in SciDB, most of the operations involved in implementing the CP decomposition in an array database are not available in common array databases. We provide chunk-based implementations of the various operations involved in the CP decomposition.

For a dynamic tensor decomposition, we adapt the Dynamic Tensor Analysis (DTA) algorithm [70] for in-database operation. Note that, unlike CP, DTA assumes a dense core matrix as in Tucker decomposition [74]; but, as shown in [8], results of Tucker decompositions can be used as a first step towards bootstrapping CP decomposition. DTA incrementally maintains covariance matrices for each mode and computes factor matrices by taking the leading eigen-vectors of the covariance matrices.

The in-database CP and DTA algorithm are implemented using the TensorDB operators along with SciDB operators. The TensorDB operators are chunk-based tensor operators (*matricization, Khatri-Rao product, Hadamard product, normalization*, and *copyArray* operators) needed for implementing in-database tensor decompositions. Each of these leverages the chunk ordering and access mechanism in Figure 4.6.

In addition to the above chunked operators, we also implement two non-chunked operators, *pseudoinverse* and *eigen-decomposition*. While these require their inputs to fit into the memory, since (during tensor decomposition) inputs are often relatively small matrices, this rarely constitutes a problem. More details of each of these TensorDB operators are described in Section 8.

4.3.4 Tensor-Relational Query Plans

TensorDB supports tensor-relational query plans needed for both data manipulation and integration, and data analysis. SciDB provides data manipulation operations such as `SELECT`, `subarray`, `slice`, etc. and data integration operations such as `JOIN`. For details of the SciDB operators, see the SciDB user guide [4]. TensorDB provides tensor decomposition operations for data analysis, e.g., `cp_als.py`, which is a python application for the CP decomposition [3] . The followings show a number of query examples for tensor-relational query plans.

Example 4.3.1 `slice` *and* `cp_als` *operations.*

- `iquery -nq "SELECT * into T_slice FROM slice(T, i, 1)"`

  - *Selects the 1st slice of mode* `i` *of a tensor* `T` *of size* $I_1 \times I_2 \times I_3 \times I_4$ *with chunk size* $J_1 \times J_2 \times J_3 \times J_4$ *and saves the result into* `T_slice`*.* `iquery` *is the SciDB query interface.*

- `cp_als.py T_slice` I_2, I_3, I_4 J_2, J_3, J_4 `rank`

  - *Runs* `cp_als` *on* `T_slice`*. (Usage: cp_als.py <tensor_name> <tensor_size> <chunk_size> <target_rank>).*

[3]The source code and user guide of TensorDB are available at `https://github.com/mkim48/TensorDB`

Example 4.3.2 `subarray` *and* `cp_als` *operations.*

- `iquery -nq "SELECT * into T_subarray FROM subarray(T, 1, 1, 1,` I'_1, I'_2, I'_3`)"`

  - *Takes a sub-tensor of size* $I'_1 \times I'_2 \times I'_3$ *from a tensor* `T` *of size* $I_1 \times I_2 \times I_3$ *with chunk size* $J_1 \times J_2 \times J_3$ *and saves the result into* `T_subarray`*.*

- `cp_als.py T_subarray` I'_1, I'_2, I'_3 J_1, J_2, J_3 `rank`

  - *Runs* `cp_als` *on* `T_subarray`*.*

Example 4.3.3 `join` *and* `cp_als` *operations.*

- `iquery -nq "SELECT ratings.val * genre.val into ratings_genre FROM ratings JOIN genre ON ratings.movie_id = genre.movie_id"`

  - *Joins the two tensors,* `ratings` *of size* $I_1 \times I_2 \times I_3$ *with chunk size* $J_1 \times J_2 \times J_3$ *and* `genre` *of size* $I_1 \times I_4$ *with chunk size* $J_1 \times J_4$ *on the 1st mode of each tensor,* `movie_id` *and saves the result into* `ratings_genre`*.*

- `cp_als.py ratings_genre` I_1, I_2, I_3, I_4 J_1, J_2, J_3, J_4 `rank`

  - *Runs* `cp_als` *on* `ratings_genre`*.*

Example 4.3.4 `dta` *operations.*

- `dta.py T1` I_1, I_2, I_3 J_1, J_2, J_3 r_1, r_2, r_3

  - *Takes a tensor* `T1` *of size* $I_1 \times I_2 \times I_3$ *with chunk size* $J_1 \times J_2 \times J_3$ *and decomposes the tensor with target ranks* $r_1 \times r_2 \times r_3$*.*

- `dta.py T2` I_1, I_2, I_3 J_1, J_2, J_3 r_1, r_2, r_3 `T1`

  - *Takes a tensor* `T2` *of size* $I_1 \times I_2 \times I_3$ *with chunk size* $J_1 \times J_2 \times J_3$ *and incrementally updates the tensor decomposition with target ranks* $r_1 \times r_2 \times r_3$ *of the old tensor* `T1` *with the new tensor* `T2`*.*

4.3.5 Optimization Strategies in In-Database TensorDB

Many operations involved in tensor decomposition are order sensitive and the way data is laid on disk may have a big impact on the total cost of tensor decomposition task. Specifically we consider the matricization operation. The matricization operation transforms a tensor into a matrix by arranging the fibers of a mode into the columns of the resulting matrix. The matricization is a costly operation due to the data movements that it may require and depending on how the data is laid out physically, different matricizations may involve different amount of data movements, which (when data is stored on secondary storage) may result in high I/O load. Thus we consider an optimization strategy that minimizes the data movement in matricization operations and introduce a chunk-optimized matricization operator.

To further reduce the cost of matricization, we can also leverage materialization of the matricization results. The materialization of the matricization can help reduce the running time of in-database CP, especially on input tensors with higher number of modes and dense representations.

In-database tensor decomposition algorithms tend to involve computationally expensive operations such as matrix multiplication. The in-database matrix multiplication can be costly, for example, the covariance matrix computation in the in-database DTA [70] that involves the matrix muliplication of matricized tensor and its transpose. Since the matricized tensor is as big as the input tensor, the covariance matrix computation can be very costly. We propose to address this by leveraging the compressed matrix multiplication technique [57] to optimize the covariance matrix computation. The idea of the compressed matrix multiplication is that the matrix product can be approximated with high probability if the matrix product is compressible, i.e., if the Frobenius norm of the matrix product is dominated by a sparse subset of entries of the product and this condition is often satisfied for the covariance matrix computation since covariance matrices tend to be skewed. We can also determine in advance whether regular or compressed matrix multiplication is advantageous, based on the sparsity of the initial covariance matrix. When considering chunk-based in-database implementations, various further optimizations need to be considered such as chunk density and chunk shaped based optimizations.

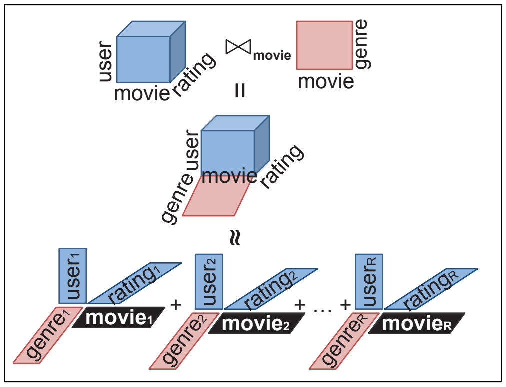
Chapter 5

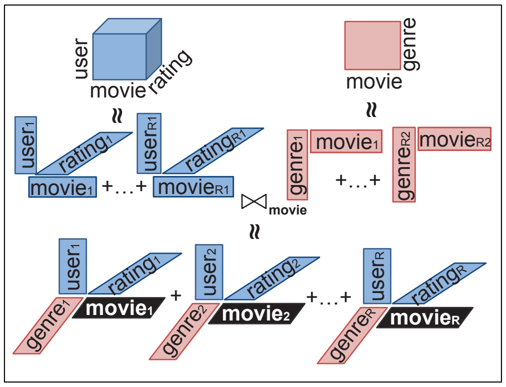
JOIN-BY-DECOMPOSITION (JBD) FOR EFFICIENT TENSOR DECOMPOSITION WITHIN THE JOIN OPERATION

In traditional relational algebraic systems the join operation and in tensor-algebraic framework the tensor decomposition operation tend to be the computationally costliest operations. In the data lifecycle, data are often integrated from different sources before it goes through other manipulation steps and the final step of the tensor relational query plan is almost always a tensor decomposition operation for data analysis. For an efficient tensor decomposition operation when combined with the join operation, we propose a highly efficient, effective, and parallelizable `join-by-decomposition` (JBD) approach and the corresponding optimization strategies for analysis of integrated multidimensional data. Experimental results show that the proposed `join-by-decomposition` performs faster than the conventional `join-then-decompose` scheme on large data sets and also confirm that the proposed `join-by-decomposition` scheme enables effective parallelization of smaller rank decompositions to achieve higher efficiencies, especially for large-scale problems.

5.1 Introduction

The lifecycle of data involves multiple operations to support the data manipulation/integration and analysis. Consider the following example involving analysis of data integrated from multiple data sources: the query in the example requires both *a join operation (costliest relational algebraic operation) for data integration and a decomposition (costliest tensor manipulation operation) for data analysis*:





(a) `join-then-decompose` (b) `join-by-decomposition`

Figure 5.1: (a) `join-then-decompose`: rank-r CP decomposition of the tensor obtained by joining `(user, movie, rating)` and `(movie, genre)` relations. (b) `join-by-decomposition`: rank-r_1 CP decomposition of `(user, movie, rating)` relation and rank-r_2 CP decomposition of `(movie, genre)` relation are combined on the `movie` mode into rank-r CP decomposition of the joined tensor.

Example 5.1.1 *Consider two relations described as tensors* [1] *: a 3-mode relational tensor of* `(user, movie, rating)` *and a 2-mode relational tensor of* `(movie, genre)`*. Let us assume that we have an application that requires us to first combine these two relations based on the* `movie` *attribute and then obtain the decomposition of the integrated tensor: Figure 5.1(a) illustrates how we would first combine these two relational tensors on the* `movie` *attribute into a 4-mode multi-relational tensor* `(user, rating, movie, genre)` *and then perform a tensor decomposition. In the rest of the chapter, we refer to this as the* `join-then-decompose` *processing.* ♢

Note that in this example the combined tensor is higher-dimensional than both input tensors (see Figure 5.1(a)), therefore its decomposition is likely to be significantly more expensive than the decompositions of the two original input tensors for dense data sets. Even for sparse data sets, for which there are decomposition algorithms that have time complexities linear in the number of nonzero entries [70], join operations

[1]Since we use a tensor model to describe relational data, the corresponding terms in the tensor and the relational model (e.g., a relation and a tensor, an attribute and a mode, etc) can be used interchangeably throughout the thesis.

with high-join rates may result in decomposition operations that are prohibitively costly. Since the costliest operation in the `join-then-decompose` processing is not the join operation (as in the traditional relational systems), but the decomposition operation and since the join operation tends to push the cost of tensor decomposition higher, *we argue that a* `join-by-decomposition` *scheme will be more efficient than the conventional* `join-then-decompose` *scheme.*

Example 5.1.2 *Consider the query in the previous example. An alternative processing scheme would involve first decomposing the input tensors into their spectral components and then combining these into the decomposition of the joined tensor. Figure 5.1(b) illustrates this* `join-by-decomposition` *scheme.* ◊

However, implementing the `join-by-decomposition` scheme presented in the above example requires overcoming a number of challenges:

- **Challenge I:** Tensor decomposition can be seen as searching for the eigen-basis of the given tensor and a mapping of the input data onto this eigen-basis. While this representation is very useful when a fixed basis for analysis is not available, it also poses challenges when integrating decompositions of multiple tensors: since each tensor has its *own* eigen-basis, combining different decompositions to obtain the decomposition of the joined tensor is not straightforward.

- **Challenge II:** Unlike the decomposition of the joined tensor, which captures the relationships between all four modes (`user`, `movie`, `rating`, and `genre`) simultaneously in the above example, the individual decompositions of the input tensors capture the relationships of the partial subsets of these four modes. As a result, it is important to be able to select a `join-by-decomposition` strategy that will closely approximate the conventional `join-then-decompose` strategy.

In this chapter, we focus on query-plans including nonnegative CP tensor decompositions of joined tensors and, to tackle the above challenges, we propose an approximate tensor decomposition scheme for joined tensors which involves combining two smaller rank decompositions of the input tensors (rather than decomposing the joined tensor itself). Since in nonnegative CP tensor decomposition,

- the elements of the core tensor can be seen as clusters of the data and
- each entry, U_{ij}, can be seen as the conditional probability $P(U_i|C_j)$ of the ith element of mode $\mathbf{U}$ to belong to the jth cluster, C_j,

we propose to obtain rank-r decomposition of the joined tensor by combing two rank decompositions whose ranks are the factors of r such that $r_1 \times r_2 = r$. Intuitively, each of the r clusters of the joined tensor is constructed by combining a pair of clusters from the two input tensors. While finding factorizations of a given value r is a computationally hard problem [24], in most applications of interest, the value of r is too small (at most 10s or 100s) for this to be an issue.

It is important to note that, given an r (e.g., 6) with multiple factorizations (e.g., 1×6, 2×3, 3×2, and 6×1) into multiplicand-multiplier pairs, different factorizations may result in different time gain/approximation error trade-offs. Therefore, as highlighted in Challenge II, we need to select, among all possible multiplicand-multiplier factorization of r, one pair that is likely to provide the best gain/error trade-off. Therefore we need to find a way to (efficiently) predict the degree of fit of the overall decomposition from the decompositions of each pair. In this chapter, we explore various measures to determine the best pair which is likely to have the least fit error and show that the norm of the combined decomposition leads to a good approximation of the fitness with respect to the original joined tensor.

One advantage of the proposed `join-by-decomposition` strategy, as opposed to the conventional `join-then-decompose` strategy, is that we need to operate with multiple smaller tensor decompositions, performed independently from each other. As a consequence, often these small decomposition operations can naturally fit in multiple cores of a given processor and be executed in a parallel manner. This leads to a highly efficient, effective, and parallelizable algorithm for `join-by-decomposition` strategy. Therefore, in this chapter, we also investigate parallel multi-core execution strategies.

5.2 Challenge: Query Plans with Joins and Decompositions

In Chapter 3, we presented the tensor-relational model and basic tensor relational algebraic operations. As in the case of relational algebra, a query (or data manipulation) plan can be visualized as a tree, where the leaves of the tree are the input tensors and each node of the tree is a tensor relational operation, selecting, projecting, or joining its inputs. As illustrated in Figure 1.1, however, the tensor-relational operations are part of a larger framework that involves other tensor operations, such as tensor decompositions. In fact, in most cases, the tensor relational operations precede a tensor decomposition operation to manipulate the data into a form ready for the context of the analysis. Therefore, the root (i.e., the final step) of the tensor relational query plan tree is almost always a tensor decomposition operation (Figure 4.3 (a)).

In traditional relational algebra, the costliest operation is known to be the join operation which, depending on the implementation, can take up to $O(|\mathcal{P}| \times |\mathcal{Q}|)$ where $|\mathcal{P}|$ and $|\mathcal{Q}|$ are the numbers of tuples of two relations $\mathcal{P}$ and $\mathcal{Q}$ respectively, in tensor-algebraic framework, tensor decomposition tends to be the computationally costliest operation of all, which is exponential in the number of modes for dense tensors. Tensor decomposition is an expensive operation also for sparse tensors. The situation is especially aggravated when the decomposition operation is preceded by a join operation which increases the number of modes of the tensor to be decomposed (for dense tensors) or increases the number of tuples, i.e. nonzero entries in the resulting tensor (for sparse tensors). In both of these cases, data integration through joins tends to increase the cost of the whole plan significantly and even renders the whole query infeasible if sufficient resources and time are not available.

The rest of the chapter is as follows:

- We provide an overview of the proposed `join-by-decomposition` scheme in Section 5.3.
- We then focus on the problem of selecting the best pair of factors of r and present various approaches to find the best pair that is likely to provide good time/accuracy tradeoffs (Section 5.3.4).
- In Section 5.3.6, we show that the proposed `join-by-decomposition` scheme gives rise to novel parallelization opportunities.
- In Section 5.4, we experimentally evaluate the proposed `join-by-decomposition` scheme in both stand-alone and parallel configurations for CP and Tucker decompositions. We focus on the accuracy and the running time of the alternative algorithms. Experimental results show that the norm of join-by-decomposition can approximate the fitness of `join-by-decomposition` with respect to the original joined tensor. This helps ensure the efficiency and effectiveness of the proposed `join-by-decomposition` approach.

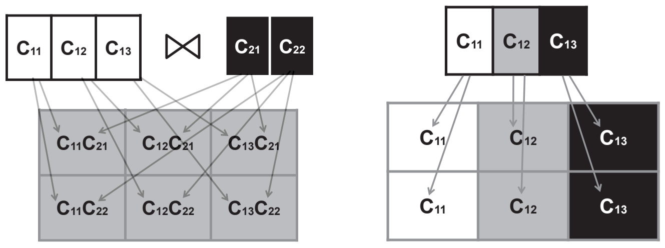
5.3 Optimization of Tensor Decompositions within Join Plans

As described above, tensor decompositions following joins in a query plan tend to be expensive when processed in a traditional, `join-then-decompose` manner (Figure 4.3(a)). In this section, we present an alternative `join-by-decomposition` approach to obtain decompositions of joined tensors. In particular, as visualized in Figure 4.3(b), instead of decomposing the higher-modality joined tensor as in the `join-then-decompose` scheme, we first decompose the lower-modality input tensors and then combine these decompositions to obtain the final decomposition.

A key challenge is that, in general, there may be many different ways that one can decompose the input tensors and combine them to obtain the final decomposition. These different `join-by-decomposition` schemes may have different processing costs and accuracies. Therefore, we present approaches to select an effective `join-by-decomposition` scheme among the alternatives.

5.3.1 Overview of the Join-by-Decomposition (JBD) Process

In this section, we present the proposed `join-by-decomposition` approach for obtaining decompositions of joined tensors for both CP decompositions and Tucker decompositions of joined tensors. We refer to the JBD algorithm for CP decomposition and Tucker decompositions as JBD-CP and JBD-Tucker, respectively. For simplicity and clarity, we limit the discussion to nonnegative CP decompositions, with probabilistic interpretations. While JBD-CP and JBD-Tucker are formulated for nonnegative decompositions, they can generally perform for tensor decompositions involving negative values.



(a) (b)

Figure 5.2: (a) When two groups of clusters on join factor matrices are combined, each cluster in one group is combined with all the clusters of the other group. If one group has three clusters and the other has two clusters, then there are potentially up to six combination clusters. (b) Clusters in other factor matrices than the join factor matrices are extended to as many clusters as the number of the combined clusters.

5.3.2 Join-by-Decompositio (JBD) for Nonnegative CP Decomposition (JBD-CP)

The JBD-CP scheme works as follows: as illustrated in Figure 5.1(b), to construct a rank-r decomposition of the joined tensor, we consider two integers, r_1 and r_2, such that $r_1 \times r_2 = r$ and we find rank-r_1 and rank-r_2 decompositions of the two input tensors. We then combine these two decompositions along the given factor matrix which corresponds to the join attribute in the equi-join operation (the process is trivially extended to the case where there are multiple equi-join attributes in the query). Intuitively, we treat each diagonal element in the core tensor as a cluster and the factor matrices as the conditional probabilities of the attribute values along the modes belonging to the given clusters. Therefore, we seek to obtain the r clusters of the joined tensor by finding $r_1(\leq r)$ and $r_2(\leq r)$ clusters of the input tensors (where $r_1 \times r_2 = r$) respectively and combining them based on the equi-join attribute (Figure 5.2). Figure 5.3 illustrates an example of rank-12 decomposition of a joined tensor by JBD-CP.

Let us consider two 3-mode relational tensors, $\mathcal{P}$ and $\mathcal{Q}$, with $u \times l \times m$ and $u \times d \times s$ dimensions, respectively, and an equi-join operation on the first mode of

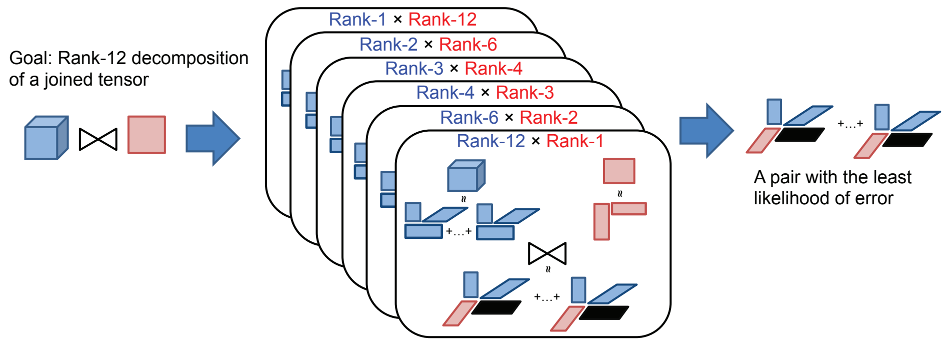


Figure 5.3: Rank-12 decomposition of a joined tensor by JBD-CP: there are 6 pairs of decompositions and the join of the pair with the least predicted likelihood of error is chosen as the final decomposition

these tensors (note that for simplicity, we assume that both modes have u slices along the join attribute, representing the common values for the two relations along the equi-join attribute. The rank-r_p and rank-r_q CP decompositions of $\mathcal{P}$ and $\mathcal{Q}$ are as follows:

$$\mathcal{P}_{u\times l\times m} \approx \sum_{a=1}^{r_p} \lambda_a \circ U_a \circ L_a \circ M_a, \quad \mathcal{Q}_{u\times d\times s} \approx \sum_{b=1}^{r_q} \lambda_b' \circ U_b' \circ D_b \circ S_b.$$

When decompositions are nonnegative and the tensors are properly normalized, the equation for $\mathcal{P}$ can be interpreted probabilistically as

$$\begin{aligned}\mathcal{P}_{u\times l\times m} \approx \sum_{a=1}^{r_p} P(C_a^p) \sum_{i=1}^{u} P(U_i|C_a^p) \\ \sum_{j=1}^{l} P(L_j|C_a^p) \sum_{k=1}^{m} P(M_k|C_a^p).\end{aligned} \tag{5.1}$$

Here C_*^p are the clusters of $\mathcal{P}$; analogously, the equation for $\mathcal{Q}$ can also be interpreted probabilistically.

Let us denote the equi-join tensor $\mathcal{P} \bowtie_{=,\mathbf{U}} \mathcal{Q}$ as $\mathcal{X}$. Similarly to the input tensors $\mathcal{P}$ and $\mathcal{Q}$, we can also probabilistically interpret the rank-r decomposition of $\mathcal{X}$:

$$\begin{aligned}\mathcal{X}_{u\times l\times m\times d\times s} \approx \sum_{c=1}^{r} P(C_c^x) \sum_{i=1}^{u} P(U_i|C_c^x) \\ \sum_{j=1}^{l} P(L_j|C_c^x) \sum_{k=1}^{m} P(M_k|C_c^x) \\ \sum_{f=1}^{d} P(D_f|C_c^x) \sum_{g=1}^{s} P(S_g|C_c^x),\end{aligned} \tag{5.2}$$

where C_*^x are the clusters of the joined tensor. Note that if the r_p and r_q clusters of the input tensors are independent from each other and $r_p \times r_q = r$, we can rewrite this in terms of the clusters and membership probabilities of the input tensors as

$$\begin{aligned}\hat{\mathcal{X}}_{u\times l\times m\times d\times s} = \sum_{a=1}^{r_p} \sum_{b=1}^{r_q} P(C_a^p) P(C_b^q) \\ \sum_{i=1}^{u} P(U_i|C_a^p) P(U_i|C_b^q) \\ \sum_{j=1}^{l} P(L_j|C_a^p) \sum_{k=1}^{m} P(M_k|C_a^p) \\ \sum_{f=1}^{d} P(D_f|C_b^q) \sum_{g=1}^{s} P(S_g|C_b^q).\end{aligned} \tag{5.3}$$

This gives us a way to reconstruct the decomposition of the join tensor directly from the decompositions of the input tensors, which are much cheaper to obtain. However, this reconstruction makes sense only if the clusters of the input tensors are independent from each other:

$$\begin{aligned}
P(C^x_{a,b}) &= P(C^p_a \wedge C^q_b) = P(C^p_a)P(C^q_b),\\
P(U_*|C^x_{a,b}) &= P(U_*|C^p_a \wedge C^q_b) = P(U_a|C^p_a)P(U_b|C^q_b),\\
P(L_*|C^x_{a,b}) &= P(L_*|C^p_a \wedge C^q_b) = P(L_*|C^p_a),\\
P(M_*|C^x_{a,b}) &= P(M_*|C^p_a \wedge C^q_b) = P(M_*|C^p_a),\\
P(D_*|C^x_{a,b}) &= P(D_*|C^p_a \wedge C^q_b) = P(D_*|C^q_b),\\
P(S_*|C^x_{a,b}) &= P(S_*|C^p_a \wedge C^q_b) = P(S_*|C^q_b).
\end{aligned} \tag{5.4}$$

Otherwise, there will be a nonzero difference between $\mathcal{X}$ and $\hat{\mathcal{X}}$. Next we describe how to minimize the approximation error, $\|\mathcal{X} - \hat{\mathcal{X}}\|$.

5.3.3 Join-by-Decomposition (JBD) for Nonnegative Tucker Decomposition (JBD-Tucker)

In this subsection, we extend JBD to nonnegative Tucker decompositions (JBD-Tucker). Similarly to the formulation of JBD-CP, we formulate JBD-Tucker as follows. Consider two 3-mode relational tensors, $\mathcal{P}$ and $\mathcal{Q}$, with $u \times a \times b$ and $u \times d \times e$ dimensions, respectively. The rank-(R_p, S, T) and rank-(R_q, V, W) nonnegative Tucker decompositions of $\mathcal{P}$ and $\mathcal{Q}$ are as follows:

$$\mathcal{P} \approx \mathcal{G}_{\boldsymbol{p}} \times_1 \mathbf{U} \times_2 \mathbf{A} \times_3 \mathbf{B} = \sum_{r_p=1}^{R_p} \sum_{s=1}^{S} \sum_{t=1}^{T} \mathcal{G}_{\boldsymbol{p}}[r_p, s, t]\, U_{r_p} \circ A_s \circ B_t.$$

$$\mathcal{Q} \approx \mathcal{G}_{\boldsymbol{q}} \times_1 \mathbf{U}' \times_2 \mathbf{D} \times_3 \mathbf{E} = \sum_{r_q=1}^{R_q} \sum_{v=1}^{V} \sum_{w=1}^{W} \mathcal{G}_{\boldsymbol{q}}[r_q, v, w]\, U'_{r_q} \circ D_v \circ E_w.$$

Here, each core tensor of $\mathcal{P}$ and $\mathcal{Q}$, $\mathcal{G}_{\boldsymbol{p}}$ and $\mathcal{G}_{\boldsymbol{q}}$ respectively, expresses the weight (or strength) of the interaction between the different components. Similarly to CP decomposition in Section 5.3.2, if decompositions are nonnegative and normalized, the Tucker decomposition for $\mathcal{P}$ can be interpreted probabilistically with respect to rank R_p as

$$\mathcal{P} \approx \sum_{r_p=1}^{R_p} P(C^p_{r_p}) \sum_{i=1}^{u} P(U_i|C^p_{r_p}) \sum_{j=1}^{a} A_j \sum_{k=1}^{b} B_k,$$

where C^p_* are the clusters of the values of the join attribute for $\mathcal{P}$ and $P(C^p_{r_p}) = \sum_{s=1}^{S}\sum_{t=1}^{T} P(C^p_{r_p} \wedge C^p_s \wedge C^p_t)$. Analogously, the Tucker decomposition for $\mathcal{Q}$ can also be interpreted probabilistically with respect to rank R_q as

$$\mathcal{Q} \approx \sum_{r_q=1}^{R_q} P(C^q_{r_q}) \sum_{i=1}^{u} P(U_i|C^q_{r_q}) \sum_{l=1}^{d} D_l \sum_{m=1}^{e} E_m,$$

where C^q_* are the clusters of the values of the join attribute for $\mathcal{Q}$ and $P(C^q_{r_q}) = \sum_{v=1}^{V}\sum_{w=1}^{W} P(C^q_{r_q} \wedge C^q_v \wedge C^q_w)$.

Let us denote the equi-join tensor $\mathcal{P} \bowtie_{=,\mathbf{U}} \mathcal{Q}$ as $\mathcal{X}$. Similarly to the input tensors $\mathcal{P}$ and $\mathcal{Q}$, we can also interpret the rank-(R, S, T, V, W) Tucker decomposition of $\mathcal{X}$ probabilistically with respect to R:

$$\mathcal{X} \approx \hat{\mathcal{X}} = \sum_{r=1}^{R} P(C^x_r) \sum_{i=1}^{u} P(U_i|C^x_r) \sum_{j=1}^{a} A_j \sum_{k=1}^{b} B_k \sum_{l=1}^{d} D_l \sum_{m=1}^{e} E_m,$$

where C^x_* are the clusters of the values of the join attribute for the joined tensor $\mathcal{X}$ and $P(C^x_r) = \sum_{s=1}^{S}\sum_{t=1}^{T}\sum_{v=1}^{V}\sum_{w=1}^{W} P(C^x_r \wedge C^x_s \wedge C^x_t \wedge C^x_v \wedge C^x_w)$.

Note that if the R_p and R_q clusters of the input tensors are independent from each other and $R_p \times R_q = R$, we can rewrite this in terms of the clusters and membership probabilities of the input tensors as

$$\mathcal{X} \approx \sum_{r_p=1}^{R_p}\sum_{r_q=1}^{R_q} P(C^p_{r_p})P(C^q_{r_q})P(U_i|C^p_{r_p})P(U_i|C^q_{r_q}) \sum_{j=1}^{a} A_j \sum_{k=1}^{b} B_k \sum_{l=1}^{d} D_l \sum_{m=1}^{e} E_m. \quad (5.5)$$

Once again, this gives us a way to reconstruct the nonnegative Tucker decomposition of the join tensor directly from the nonnegative Tucker decompositions of the input tensors, which are much cheaper to obtain. However, this reconstruction makes sense only if the clusters of the input tensors are independent from each other:

$$P(C^x_{r_p,r_q}) = P(C^p_{r_p} \wedge C^q_{r_q}) = P(C^p_{r_p})P(C^q_{r_q}),$$
$$P(U_*|C^x_{r_p,r_q}) = P(U_*|C^p_{r_p} \wedge C^q_{r_q}) = P(U_*|C^p_{r_q})P(U_*|C^q_{r_q}).$$

Otherwise, there will be a nonzero difference between $\mathcal{X}$ and $\hat{\mathcal{X}}$. As in JBD-CP, we employ norm-based pair selection (psm_{norm}) method for selecting the rank-(...,R_p,...) and rank-(...,R_q,...) Tucker decompositions of $\mathcal{P}$ and $\mathcal{Q}$. Again, $\|\hat{\mathcal{X}}\|$ can be computed directly from the decomposition $\tilde{\mathcal{X}}$, thus psm_{norm} is computed much more efficiently than $\|\mathcal{X} - \hat{\mathcal{X}}\|$. Also psm_{norm} approximates the fit error effectively in JBD-Tucker (psm_{norm} selected the best pair in terms of fit in all the cases of JBD-Tucker experiments in Table 5.9).

5.3.4 Minimization of the Approximation Error

Since the above formulation is based on the assumption that the conditional probabilities of the attribute values given the clusters of the input tensors $\mathcal{P}$ and $\mathcal{Q}$ are independent of each other, the natural approach to minimize the approximation error involves searching for input clusters (i.e., decompositions of the input tensors) that are the most independent relative to the join attribute.

Independency Desideratum: The decompositions of the input tensors should be the most independent relative to the join attribute.

However, once a pair of rank-r_p and rank-r_q decompositions of $\mathcal{P}$ and $\mathcal{Q}$ (where $r_p \times r_q = r$) are given, there is no room for carrying out such a search. Yet, if we consider the set $\{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}$ which is all possible factorizations of r, then we can select among all these pairs the one that leads to clusters that are the most independent relative to the join attribute. This presents two challenges:

- We need to enumerate pairs of ranks that multiply to r and then obtain the corresponding decompositions of the input tensors $\mathcal{P}$ and $\mathcal{Q}$. As we will see in the experiment section (Section 5.4), while this involves enumeration of multiple (low-modal) decompositions, the overall cost of the process is often much less than the cost of decomposing the (high-modal) joined tensor.

- Given a pair of rank-r_p and rank-r_q decompositions of $\mathcal{P}$ and $\mathcal{Q}$, this requires a measure to quantify the independence of the clusters relative to the join attribute. The problem is that the term $P(U|C_a^p \wedge C_b^q)$ in Equation (5.4) requires counting joins falling within a cluster given by the decomposition of the joined tensor; but this is not known.

- In addition, we have to consider whether rank-r_p and rank-r_q decompositions are, in fact, appropriate for the input tensors $\mathcal{P}$ and $\mathcal{Q}$. Selecting inappropriate decomposition ranks for $\mathcal{P}$ and $\mathcal{Q}$ may increase the overall error, since the final decomposition will also depend on the accuracy of the decompositions of $\mathcal{P}$ and $\mathcal{Q}$.

Next we discuss alternative approaches for selecting the rank-r_p and rank-r_q decompositions of $\mathcal{P}$ and $\mathcal{Q}$ in such a way that the resulting clusters are independent from each other.

KL-based Pair Selection (psm_{KL})

The first alternative is a Kullback-Leibler divergence (KL)-based pair selection measure (psm_{KL}). Intuitively, we could say that *two input clusterings are independent relative to the join attribute* if given a join value U_j that connects clusters C_a^p and C_b^q, another join value U_l is neither more likely or less likely to connect these two clusters. More specifically, given the join elements in C_a^p, we would expect to see the distribution of the corresponding C_*^q to be uniformly distributed. Similarly, given the join elements in C_b^q, the distribution of the corresponding C_*^p should be uniform. One way to quantify this would be to measure the KL-divergence of the conditional probabilities of the values in the join mode against the uniform distribution:

$$\begin{aligned} psm_{KL}(r_p, r_q) &= (\sum_b KL[P(C_*^p|C_b^q), uniform] + \\ & \quad \sum_a KL[P(C_*^q|C_a^p), uniform])^{-1}. \end{aligned}$$

A potential problem with this approach is that these conditional probabilities are not directly comparable for the different pairs since there are different numbers of clusters for the join mode of each pair and probability distributions tend to be more uniformly distributed as the sample size, which in this case is the number of clusters, increases. For example, (1, 12)-rank and (12, 1)-rank pairs are likely to be more uniformly distributed, in other words have smaller KL-divergence against the uniform distribution than (3, 4)-rank and (4, 3)-rank pairs. A second drawback of this measure is that it takes into account only the independence between the join modes without considering the other factor matrices.

Fit-based Pair Selection (psm_{in})

The second alternative, $psm_{in}(r_p, r_q)$ measures the degree of fit between the input tensors and their decompositions:

$$\begin{aligned} psm_{in}(r_p, r_q) &= (\quad \|\boldsymbol{\mathcal{P}} - \hat{\boldsymbol{\mathcal{P}}}_{r_p}\| + \\ & \quad \|\boldsymbol{\mathcal{Q}} - \hat{\boldsymbol{\mathcal{Q}}}_{r_p}\|)^{-1}, \end{aligned}$$

where $\hat{\boldsymbol{\mathcal{W}}}$ is the tensor obtained by recombining its (approximate) decomposition $\tilde{\boldsymbol{\mathcal{W}}}$ of the tensor $\boldsymbol{\mathcal{W}}$. This measure takes all factor matrices into account and is not affected by the number of clusters. However, it also has a potential weakness: it does not account for the errors that may be generated when combining the two tensor decompositions. This error is likely to be high especially when the two tensors are correlated.

Norm-based Pair Selection (psm_{norm})

For a joined tensor $\mathcal{X}$ of size $K_1 \times K_2 \times \cdots \times K_{N_k}$, it is clear that an approximate decomposition $\hat{\mathcal{X}}$ with the minimum fit error $\|\mathcal{X} - \hat{\mathcal{X}}\|$ is the best pair. However the direct computation of the fit error $\|\mathcal{X} - \hat{\mathcal{X}}\|$ would require large amounts of memory (often impossible for large-scale data) for obtaining $\hat{\mathcal{X}}$. Therefore the third alternative we consider is to use the difference between the norm of $\mathcal{X}$ and the norm of $\hat{\mathcal{X}}$ as an approximation for $\|\mathcal{X} - \hat{\mathcal{X}}\|$; in other words, we can use the following measure to select the appropriate pair.

$$psm_{norm}(r_p, r_q) = |\|\mathcal{X}\| - \|\hat{\mathcal{X}}_{r_p,r_q}\||^{-1}.$$

Note that we have seen in Equation (3.4) that $\|\hat{\mathcal{X}}\|$ can be computed directly from the decomposition $\tilde{\mathcal{X}}$ using the formulation for $\tilde{\mathcal{X}}$.

The intuition behind this pair selection measure is as follows: Since $\mathcal{W}, \hat{\mathcal{W}} \geq 0$, we can use the reverse triangle inequality: $\|\mathcal{W} - \hat{\mathcal{W}}\| \geq |\|\mathcal{W}\| - \|\hat{\mathcal{W}}\||$, i.e., while the term $|\|\mathcal{W}\| - \|\hat{\mathcal{W}}\||$ is only a lower bound on $\|\mathcal{W} - \hat{\mathcal{W}}\|$, it may still provide an indication of the size of the term and thus we may be able to minimize the term $\|\mathcal{W} - \hat{\mathcal{W}}\|$ by minimizing $|\|\mathcal{W}\| - \|\hat{\mathcal{W}}\||$.

Note that the computation of the norm based pair selection measure involves combining the tensor decompositions $\tilde{\mathcal{P}}_{r_p}$ and $\tilde{\mathcal{Q}}_{r_q}$ to obtain $\tilde{\mathcal{X}}_{r_p,r_q}$ for all pairs of r_p and r_q, where $r_p \times r_q = r$. The cost of the combination step (see Figure 5.2(a)) for each entry of the joined mode $\mathbf{J}$, is $O(J \sum_{i=1}^{n} r_{p,i} \cdot r_{q,i})$ where J is the size of the mode $\mathbf{J}$ and n is the number of (r_p, r_q) pairs. The computation of $\|\mathcal{X}\|$ requires $O(|\mathcal{X}|)$

Table 5.1: Notation used in this chapter

Notation	Description
$\mathcal{P}$	the 1st input tensor of size $I_1 \times I_2 \times \cdots \times J \times \cdots \times I_{N_p}$
$\mathcal{Q}$	the 2nd input tensor of size $I'_1 \times I'_2 \times \cdots \times J \times \cdots \times I'_{N_q}$
$\mathcal{X}$	the joined tensor of $\mathcal{P}$ and $\mathcal{Q}$ on the mode $\mathbf{J}$ of size J; i.e., $\mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}$ of size $K_1 \times K_2 \times \cdots \times J \times \cdots \times K_{N_x}$
r	the rank of $\mathcal{X}$
$r_{p,i}$ and $r_{q,i}$	the ith ranks of $\mathcal{P}$ and $\mathcal{Q}$, resp.; i.e., $(r_{p,i}, r_{q,i}) \in \{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}$
N_p	# of modes of $\mathcal{P}$
N_q	# of modes of $\mathcal{Q}$
N_x	# of modes of $\mathcal{X}$
$\alpha_{r,*}$	# of ALS iterations needed to rank-r decompose the tensor denoted by "*"
$\lvert\mathcal{P}\rvert$	# of nonzero entries of a tensor $\mathcal{P}$
$\lvert\mathcal{Q}\rvert$	# of nonzero entries of a tensor $\mathcal{Q}$
$\lvert\mathcal{X}\rvert$	# of nonzero entries of a tensor $\mathcal{X}$
n	# of (r_p, r_q); i.e., $\lvert\{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}\rvert$

Table 5.2: Cost for `join-then-decompose` and `join-by-decomposition`

Algorithm	Step		Cost
`join-then-decompose`	Decomp.	dense tensors	$O(\prod_{i=1}^{N_x} K_i)^{\dagger}$
		sparse tensors	$O(\alpha_{r,\mathcal{X}}\, r\, \lvert\mathcal{X}\rvert\, N_x)^{\dagger\dagger}$
	(Equi-)Join		$O((\lvert\mathcal{P}\rvert + \lvert\mathcal{Q}\rvert)\log(\lvert\mathcal{P}\rvert + \lvert\mathcal{Q}\rvert))$
`join-by-decomposition`	Decomp.	dense tensors	$O(n(\prod_{i=1}^{N_p} I_i + \prod_{i=1}^{N_q} I'_i))^{\dagger}$
		sparse tensors	$O(\sum_{i=1}^{n}(\alpha_{r_{p,i},\mathcal{P}}\, r_{p,i}\, \lvert\mathcal{P}\rvert\, N_p + \alpha_{r_{q,i},\mathcal{Q}}\, r_{q,i}\, \lvert\mathcal{Q}\rvert\, N_q))^{\dagger\dagger}$
	(Equi-)Join		$O(J \sum_{i=1}^{n} r_{p,i} \cdot r_{q,i})$
	Norm comp.		$O(\lvert\mathcal{X}\rvert + nr^2 \sum_{i=1}^{N_x} K_i)$

†The execution time cost for dense tensors is based on [70].

††The execution time cost for sparse tensors is based on the analysis of the code in [12].

time, where $\lvert\mathcal{X}\rvert$ is the number of nonzero entries of $\mathcal{X}$ [11] and the norm computation for each pair takes $O(nr^2 \sum_{i=1}^{N_k} K_i)$ from Equation (3.4).

5.3.5 *Time Complexities of* `join-then-decompose` *and* `join-by-decomposition`

Table 5.2 presents the time complexities for the `join-then-decompose` and `join-by-decomposition` operations. The symbols used in this table and the rest of this chapter are described in Table 7.1.

- The `join-then-decompose` process includes the join and decomposition costs. As discussed in Section 3.3.2, for the join operation in the `join-then-decompose`, we use an efficient equi-join operator based on the inner product for sparse tensors; the cost for this operator is described in Section 3.3.2.

- The `join-by-decomposition` involves the decomposition and join operations, and norm computation. The decomposition operation in the `join-by-decomposition` takes as many operations as the number of each rank r_p and r_q of the rank pairs (r_p, r_q). The join operation and norm computation of `join-by-decomposition` are performed for each pair (r_p, r_q) (see Section 5.3.4 for the details of the costs of the join operation and norm computation).

Dense Tensors

For dense tensors, decomposition is clearly the most dominant cost and tensor decompositions with smaller number of modes, as in `join-by-decomposition`, are much more efficient than the tensor decompositions with large number of modes, as in the `join-then-decompose`.

Sparse Tensors

The cost of the decomposition operation for sparse tensors depends on the rank, the number of nonzero entries, and the number of modes for each iteration of the ALS in CP algorithm [12].

In the case of sparse tensors, in order to predict whether `join-then-decompose` or `join-by-decomposition` will be more efficient, we need to consider the number of nonzero entries in the input tensors as well as the output tensor: if the join selectivity

$$js = |\mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}|/(|\mathcal{P}||\mathcal{Q}|)$$

of the join operation is high and we have more tuples (nonzero entries) in the joined tensor than the input tensors, then the `join-by-decomposition` will be more efficient than the `join-then-decompose`; otherwise, `join-then-decompose` may be competitive.

In particular, for the `join-by-decomposition` approach to outperform `join-then-decompose` in the costly decomposition step, (assuming the number of ALS iterations of the decompositions are similar) the following must hold:

$$r|\mathcal{X}|N_x > \sum_{i=1}^{n}(r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q),$$

or, equivalently,

$$|\mathcal{X}| > \sum_{i=1}^{n}(r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q)/(rN_x).$$

Since we have $|\mathcal{X}| = |\mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}|$, we can rewrite the above inequality as

$$js(|\mathcal{P}||\mathcal{Q}|) > \sum_{i=1}^{n}(r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q)/(rN_x),$$

and this gives us a lower bound, $js_\perp$, on the join selectivity:

$$js > js_\perp = \sum_{i=1}^{n}(r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q)/(|\mathcal{P}||\mathcal{Q}|rN_x).$$

This lower bound threshold provides a practical predictor to judge whether the `join-by-decomposition` will be more advantageous the `join-then-decompose` operation for sparse tensors. In Section 5.4.3, we evaluate this effectiveness of this predictor for sparse tensors.

5.3.6 Parallelization of the Join-by-Decomposition Operation

Let us reconsider Figure 5.3 which graphically illustrates a rank-12 `join-by-decomposition` process. As this figure shows, the `join-by-decomposition`

```
pp-JBD (input: two tensors $\mathcal{P}$, $\mathcal{Q}$, rank $r$, and the modes of the join factor matrices of $\mathcal{P}$ and $\mathcal{Q}$)
 1: parfor each pair $(r_p, r_q)$ such that $r_p \times r_q = r$ do {parfor denotes Parallel for-loops of MATLAB Parallel Computing Toolbox}
 2:     Run any available nonnegative CP algorithm to get $\tilde{\mathcal{P}}_{r_p}$ and $\tilde{\mathcal{Q}}_{r_q}$ such that $\tilde{\mathcal{P}}_{r_p}$ = rank-$r_p$ nonnegative CP of $\mathcal{P}$, $\tilde{\mathcal{Q}}_{r_q}$ = rank-$r_q$ nonnegative CP of $\mathcal{Q}$
 3:     Combine $\tilde{\mathcal{P}}_{r_p}$ and $\tilde{\mathcal{Q}}_{r_q}$ on their join factor matrices into $\tilde{\mathcal{X}}_{r_p,r_q}$
 4:     Compute and record the pair selection measure, $psm(r_p, r_q)$, for $r_p$ and $r_q$
 5: end parfor
 6: Return $\tilde{\mathcal{X}}_{r_p,r_q}$ corresponding to $(r_p, r_q)$ with the best $psm(r_p, r_q)$ value
```

Figure 5.4: Pseudo-code of `pair-wise parallel join-by-decomposition (pp-JBD)`

```
ip-JBD (input: two tensors $\mathcal{P}$, $\mathcal{Q}$, rank $r$, and the modes of the join factor matrices of $\mathcal{P}$ and $\mathcal{Q}$)
 1: parfor each factor $k$ of a pair $(r_p, r_q) \in \{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}$ do
 2:     if $k = r_p$ then
 3:         $\mathcal{T}_k = \mathcal{P}$
 4:     else {$k = r_q$}
 5:         $\mathcal{T}_k = \mathcal{Q}$
 6:     end if
 7:     Run any available nonnegative CP algorithm to get $\tilde{\mathcal{T}}_k$ such that $\tilde{\mathcal{T}}_k$ = rank-$k$ nonnegative CP of $\mathcal{T}_k$
 8: end parfor
 9: parfor each pair $(r_p,\ r_q)$ such that $r_p \times r_q = r$
10:     Combine $\tilde{\mathcal{P}}_{r_p}$ and $\tilde{\mathcal{Q}}_{r_q}$ on their join factor matrices into $\tilde{\mathcal{X}}_{r_p,r_q}$
11:     Compute and record the pair selection measure, $psm(r_p, r_q)$, for $r_p$ and $r_q$
12: end parfor
13: Return $\tilde{\mathcal{X}}_{r_p,r_q}$ corresponding to $(r_p, r_q)$ with the best $psm(r_p, r_q)$ value
```

Figure 5.5: Pseudo-code of `input parallel join-by-decomposition (ip-JBD)`

involves creation of many alternative join pairs, which are independently evaluated for accuracy and the one that is predicted to provide the best accuracy is used for obtaining the final result. This provides a natural way to parallelize the `join-by-decomposition` operation: we can associate each pair of rank decompositions (and the computation of the corresponding pair selection measure) to a different processor core. In addition, when more cores are available, standard parallel tensor decomposition [59] and parallel join processing techniques [26] can also be used to further parallelize each pair.

Figures 7.3 and 5.5 show two alternative ways in which the `join-by-decomposition` operation can be parallelized; we refer to these two strategies as the `pair-wise parallel join-by-decomposition (pp-JBD)` and the `input parallel join-by-decomposition (ip-JBD)`, respectively:

- In the `pair-wise parallel join-by-decomposition (pp-JBD)` strategy, each (r_p, r_q) rank pair is assigned to a separate core.
- In the `input parallel join-by-decomposition (ip-JBD)`, on the other hand, each individual decomposition is assigned to a separate core.

It is important to note that parallelization comes with certain amount of overhead. First of all, moving the data to the different cores can add to the running time of each individual decomposition. Furthermore, balancing the work may not always be possible since decompositions with a higher rank tend to take more time than decompositions with a lower rank. In Section 5.4, we compare the parallelized versions of the `join-by-decomposition` with a block-based `join-then-decompose` using the grid NTF [59]. Note that the grid NTF based `join-then-decompose` divides the joined tensor into sub-tensors whose modes are same as that of the joined tensor. It then follows an iterative update process to reconstruct the original factors from each individual sub-factor of sub-tensor decompositions. This means that, unlike parallel `join-by-decomposition`, the subtasks of the parallel `join-then-decompose` cannot run completely separately from each other since each factor matrix construction depends on other factors within the iterative process.

5.4 Experimental Evaluation

In this section, we present experimental results assessing the efficiency and effectiveness of the proposed `join-by-decomposition` scheme relative to the conventional implementation of the `join-then-decompose` approach in both stand-alone and parallelized versions.

5.4.1 Implementation Details

We ran our experiments on an 6 cores Intel(R) Xeon(R) CPU X5355 @ 2.66GHz with 16GB of RAM. We used MATLAB Version 7.11.0.584 (R2010b) 64-bit (glnxa64) for the general implementation and MATLAB Parallel Computing Toolbox for the parallel implementations of `join-by-decomposition` and `join-then-decompose`.

We used the MATLAB Tensor Toolbox [12] to manipulate a relational tensor as a sparse tensor using a sparse tensor model. For implementing the tensor decomposition operation, we experimented with several algorithms for both `join-by-decomposition` and `join-then-decompose` operations.

5.4.2 Evaluation Criteria

Each experiment is run at least 5 times and we report the average accuracy and execution time of these runs.

Accuracy

We use the following *fit* function to measure tensor decomposition accuracy:

$$\text{fit}(\mathcal{X}, \hat{\mathcal{X}}) = 1 - \frac{\|\mathcal{X}, \hat{\mathcal{X}}\|}{\|\mathcal{X}\|}, \tag{5.6}$$

where $\|\mathcal{X}\|$ is the norm of a tensor $\mathcal{X}$. The fit is a normalized measure of how accurate the tensor decomposition $\hat{\mathcal{X}}$ is with respect to the input tensor $\mathcal{X}$.

Our evaluation criteria also include *relative fit* which indicates how accurate the `join-by-decomposition` approach is compared to the `join-then-decompose` scheme in terms of *fit* to the joined tensor. The relative fit (fit_{rel}) is defined as

$$fit_{rel} = \frac{\text{fit}(\mathcal{X}, \hat{\mathcal{X}}_{jbd})}{\text{fit}(\mathcal{X}, \hat{\mathcal{X}}_{jtd})} \tag{5.7}$$

where $\mathcal{X}$ is the joined tensor, $\hat{\mathcal{X}}_{jtd}$ is the tensor obtained by re-composing the `join-then-decompose` tensor, and $\hat{\mathcal{X}}_{jbd}$ is the tensor obtained by re-composing the `join-by-decomposition` tensor. Note that the higher relative fit is, the better the proposed `join-by-decomposition` scheme is.

Execution Time

The execution times are measured by MATLAB's `tic` and `toc` commands to start and stop the clock at the beginning and the end of the process, respectively.

5.4.3 JBD-CP Experiments

Data Sets

We used two movie rating data sets obtained from [56]: (a) MovieLens 100K data set consists of 100,000 ratings from 1,000 users on 1,700 movies, (b) MovieLens 1M data set consists of 1 million ratings from 6,000 users on 4,000 movies. In addition to the rating information, these two data sets also include movie metadata, such as movie

Table 5.3: Input relations and joined relations

Data set	1st input relation ($\mathcal{P}$)	2nd input relation ($\mathcal{Q}$)	Join mode	Joined relation ($\mathcal{X}$)
MovieLens 100K	`(user,rating,movie)`	`(movie,genre)`	`(movie)`	`(user,rating,movie,genre)`
	`(movie,rating,user)`	`(user,occupation)`	`(user)`	`(movie,rating,user,occupation)`
MovieLens 1M	`(user,rating,movie)`	`(movie,genre)`	`(movie)`	`(user,rating,movie,genre)`
	`(movie,rating,user)`	`(user,occupation)`	`(user)`	`(movie,rating,user,occupation)`

Table 5.4: Statistics of the joined relations

Data set	Joined relation	#cases	Tensor sizes	#nonzero entries
MovieLens 100K	`(user,rating,movie,genre)`	153	$100\times5\times100\times19$ to $900\times5\times1600\times19$	1051 to 194361
	`(movie,rating,user,occupation)`	153	$100\times5\times100\times21$ to $1600\times5\times900\times21$	532 to 91992
MovieLens 1M	`(user,rating,movie,genre)`	50	$800\times5\times1000\times18$ to $6000\times5\times3800\times18$	72098 to 1825034
	`(movie,rating,user,occupation)`	50	$1000\times5\times800\times21$ to $3800\times5\times4000\times21$	34189 to 819185

genre, and user metadata, such as user location and occupation. From the data sets, we created two pairs of relational tensors (see Tables 5.3):

- The first data set includes a 3-mode `(user, movie, rating)` and a 2-mode `(movie, genre)`; these join into a 4-mode `(user, rating, movie, genre)` tensor.

- The second set includes a 3-mode `(movie, user, rating)` and a 2-mode `(user, occupation)` tensors; these join into a 4-mode `(movie, rating, user, occupation)` tensor.

For each of these, we created relational tensors corresponding to different table sizes by randomly selecting entries from the MovieLens 100K and 1M data sets (see Tables 5.4). Note that all tensors are encoded as occurrence tensors, where each entry is set 1 or 0 which indicates whether the corresponding tuple exists or not. Therefore, we also record the number of nonzero entries of each tensor. The averages of tensor sizes and numbers of nonzero entries of each relation are shown in Table 5.5.

In the set of experiments reported here, we consider rank-12 decompositions. The `join-by-decomposition` scheme uses 6 combinations (1×12, 2×6, 3×4, 4×3,

Table 5.5: Averages of tensor sizes and numbers of nonzero entries of the input relations

Data set	Average of tensor sizes and # nonzero entries					
MovieLens 100K	`(user, rating, movie)`		`(movie, genre)`		`(user, rating, movie, genre)`	
	$650 \times 5 \times 900$	40484.25	900×19	1501.375	$650 \times 5 \times 900 \times 19$	84840.5
	`(movie, rating, user)`		`(user, occupation)`		`(movie, rating, user, occupation)`	
	$900 \times 5 \times 650$	20963.375	650×21	457	$900 \times 5 \times 650 \times 21$	20963.375
MovieLens 1M	`(user, rating, movie)`		`(movie, genre)`		`(user, rating, movie, genre)`	
	$3392 \times 5 \times 2704$	397055.18	2704×18	4385.82	$3392 \times 5 \times 2704 \times 18$	836104.1
	`(movie, rating, user)`		`(user, occupation)`		`(movie, rating, user, occupation)`	
	$2616 \times 5 \times 2352$	264911.24	2352×21	2352	$2616 \times 5 \times 2352 \times 21$	264911.24

6×2, and 12×1) for rank-12 decomposition for each relation in Table 5.4.

Single-Core Implementations

Firstly, we used the N-way PARAFAC algorithm with nonnegativity constraint (we call this simply N-way PARAFAC in the rest of the section) which is available in the N-way Toolbox for MATLAB [9]. We refer to `join-by-decomposition` and `join-then-decompose` using the N-way PARAFAC as JBD-NWAY and JTD-NWAY respectively.

Since the N-way PARAFAC implementation of MATLAB uses a dense tensor (multi-dimensional array) representation, it is not suitable for large data sets. The main memory required by this algorithm for the MovieLens 1M data set (the largest tensor in our experiments is $6000 \times 5 \times 3800 \times 18$ – see Table 5.4 for details.) is beyond the capability of common hardware/software setting. Another PARAFAC implementation, the CP-ALS algorithm [12], on the other hand, can run with both sparse and dense tensors. In the sparse tensor model, the cost increases linearly as the number of nonzero entries of the tensor increases. The basic CP-ALS algorithm, however, does not support nonnegative constraints. Therefore, we implemented a variant of the single grid NTF [59] using CP-ALS as the base PARAFAC algorithm.

We refer to the `join-by-decomposition` and `join-then-decompose` based on CP-ALS as JBD-CP and JTD-CP respectively.

Multi-Core Implementations

We use two alternative approaches for the parallel `join-by-decomposition`, which are pp-JBD and ip-JBD. Since pp-JBD has a slightly better performance and also for simplicity in parallelization, we mainly use pp-JBD.

The parallelized (multi-core) versions of the `join-then-decompose` were implemented using the grid NTF technique [59], with three different partition strategies. We used N-way PARAFAC and CP-ALS as the base PARAFAC algorithm. For simplicity, we use the same grid size for the `movie` and `user` mode; the `genre`, `occupation`, and `rating` modes are not divided because their sizes are already small (19 or 18, 21, and 5 respectively). For the `movie` and `user` modes (each mode becomes the 1st mode or the 3rd mode), we use 2, 4, and 8 grid cells. Therefore we divided the given tensor into sub-tensors of size $2 \times 1 \times 2 \times 1$, $4 \times 1 \times 4 \times 1$, and $8 \times 1 \times 8 \times 1$. We refer to the grid NTF algorithm for the parallel `join-then-decompose` using N-way PARAFAC of grid size $2 \times 1 \times 2 \times 1$, $4 \times 1 \times 4 \times 1$, and $8 \times 1 \times 8 \times 1$ as JTD-NWAY-GRID2, JTD-NWAY-GRID4, and JTD-NWAY-GRID8 respectively. Similarly, we refer to as JTD-CP-GRID2, JTD-CP-GRID4, and JTD-CP-GRID8 for CP-ALS.

Each cell of the grid is run with the base PARAFAC algorithm separately in parallel and iteratively combined into the final decomposition using a modified ALS approach.

Table 7.8 lists the various algorithms we consider in our experiments.

Table 5.6: Algorithms. Note that the decomposition algorithms in parentheses are used as the base PARAFAC for the grid NTF

Algorithm	Description
JBD-NWAY	`join-by-decomposition` using N-way PARAFAC
JBD-CP	`join-by-decomposition` using single grid NTF (CP-ALS)
pp-JBD-NWAY	pair-wise parallel `join-by-decomposition` using N-way PARAFAC
pp-JBD-CP	pair-wise parallel `join-by-decomposition` using single grid NTF (CP-ALS)
ip-JBD-CP	input parallel `join-by-decomposition` using single grid NTF (CP-ALS)
JTD-NWAY	`join-then-decompose` using N-way PARAFAC
JTD-CP	`join-then-decompose` using single grid NTF (CP-ALS)
JTD-NWAY-GRID2	`join-then-decompose` using $2 \times 1 \times 2 \times 1$ grid NTF (N-way PARAFAC)
JTD-NWAY-GRID4	`join-then-decompose` using $4 \times 1 \times 4 \times 1$ grid NTF (N-way PARAFAC)
JTD-NWAY-GRID8	`join-then-decompose` using $8 \times 1 \times 8 \times 1$ grid NTF (N-way PARAFAC)
JTD-CP-GRID2	`join-then-decompose` using $2 \times 1 \times 2 \times 1$ grid NTF (CP-ALS)
JTD-CP-GRID4	`join-then-decompose` using $4 \times 1 \times 4 \times 1$ grid NTF (CP-ALS)
JTD-CP-GRID8	`join-then-decompose` using $8 \times 1 \times 8 \times 1$ grid NTF (CP-ALS)

Table 5.7: Correlations between pair selection measures and accuracy for different joined relations

Measure	Joined relation	Correlation
psm_{kl}	`(movie, rating, user, occupation)`	0.46
	`(user,rating,movie,genre)`	-0.56
psm_{in}	`(movie, rating, user, occupation)`	0.76
	`(user,rating,movie,genre)`	0.56
psm_{norm}	`(movie, rating, user, occupation)`	0.86
	`(user,rating,movie,genre)`	0.82

Evaluation of the Alternative Pair Selection Measures

As discussed in Section 5.3.4, given a target rank-r decomposition for the joined tensor, the proposed `join-by-decomposition` strategy first identifies alternative rank-r_p and rank-r_q decompositions of the input tensors (such that $r_p \times r_q = r$) and then selects the most promising pair of decompositions to compute the final result. We have listed three alternative pair selection measures (psm_{kl}, psm_{in}, and psm_{norm}) in Section 5.3.4. Before we provide a detailed comparison of `join-by-decomposition` and `join-then-decompose` strategies, we first evaluate these different pair selection measures in terms of accuracy. In order to quantify the benefits of the different *psm* functions, we measure the correlation of the alternative *psm* values with the corresponding accuracy. In these experiments, we use the MovieLens 100K data. The results reported in Table 5.7 can be summarized as follows:

- The psm_{kl} measure shows slight correlation for the `(movie,rating,user,occupation)` relation, but it does not show any meaningful correlation for the `(user,rating,movie,genre)` relation.

- The correlation between the psm_{in} measure and the accuracy is stronger than the correlation of psm_{kl} for both relations.

- Finally, the psm_{norm} measure has the strongest correlation with accuracy for both relations.

  In fact, the largest psm_{norm} corresponds the best fit for 92% for the `(user, rating, movie, genre)` relation and 97% for the `(movie, rating, user, occupation)` relation.

  Also, even for the cases where the largest psm_{norm} does not give the best fit, the difference between the best fit and the fit with the largest psm_{norm} is only 0.67% and 0.02% of the best fit in average for `(user, rating, movie, genre)` and `(movie, rating, user, occupation)`, respectively.

These results indicate that the norm based measure provides the best accuracy. Therefore in the rest of this section, we use psm_{norm} as the default pair selection measure.

An interesting observation is that all measures show higher degrees of correlation for the `(movie, rating, user, occupation)` relation than for the `(user, rating, movie, genre)` relation. We conjecture that this is because the attributes of the pair

Table 5.8: Join selectivity and threshold when obtaining for different joined relations

Data set	Joined relation	$js_{\perp}$	js
MovieLens 100K	`(user, rating, movie, genre)`	0.00119	0.00221
	`(movie, rating, user, occupation)`	0.00388	0.00253
MovieLens 1M	`(user, rating, movie, genre)`	0.00040	0.00054
	`(movie, rating, user, occupation)`	0.00075	0.00055

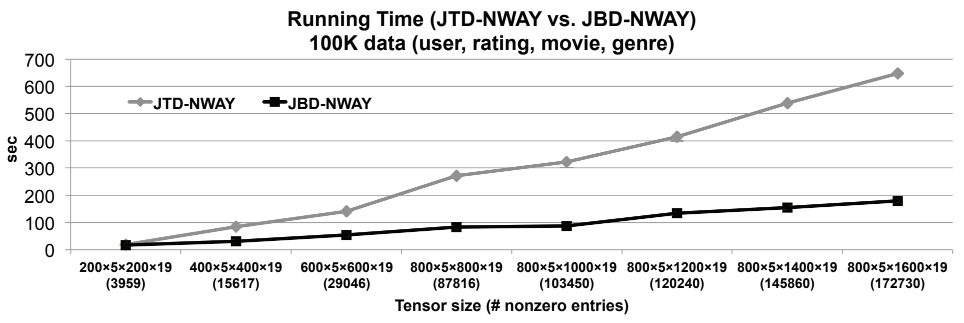
of relations `(movie, user, rating)` and `(user, occupation)` contributing to the former have less dependence with each other than the attributes of the pair of relations `(user, movie, rating)` and `(movie, genre)` contributing to the latter.

Evaluation of the Join Selectivity based Performance Predictor for Sparse Tensors

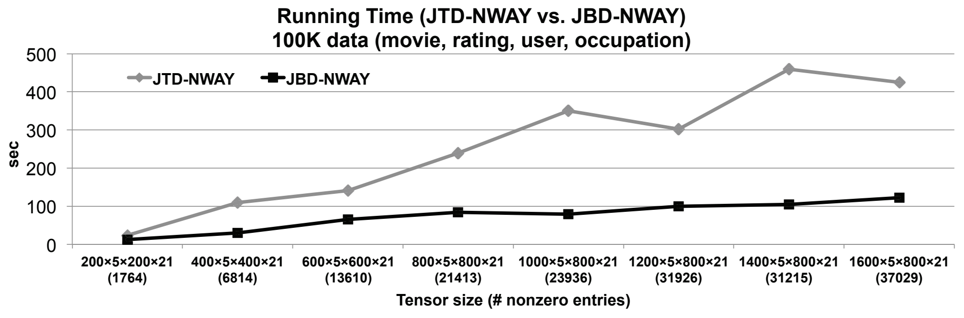
As discussed in Section 5.3.5, for sparse tensors, the join selectivity (js) can be used for predicting whether the `join-by-decomposition` can have time gain over the `join-then-decompose`. In Table 7.10, we report $js_{\perp}$ and js values for the two pairs of relations, `(user, rating, movie, genre)` and `(movie, rating, user, occupation)` of each data set. As shown in the table, the `(user, rating, movie, genre)` relation has a higher js than $js_{\perp}$ lower bound, whereas js of the `(movie, rating, user, occupation)` relation is lower than $js_{\perp}$. Therefore we expect that, for sparse tensors, `join-by-decomposition` will be more effective than `join-then-decompose` for the `(user, rating, movie, genre)` relation, but not for the `(movie, rating, user, occupation)` relation. We will also evaluate this prediction in the following sections.

Single-core Execution Time Results

MovieLens 100K Data Set. First of all, we present the execution time results for the smaller MovieLens 100K data set and compare the efficiency of the various imple-



(a) `(user, rating, movie, genre)`

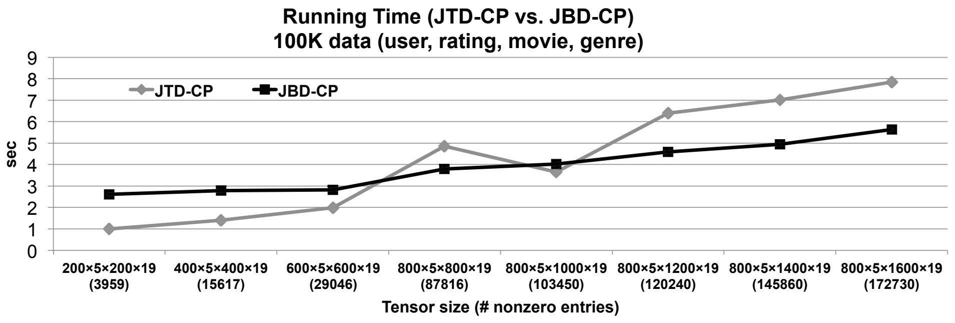


(b) `(movie, rating, user, occupation)`

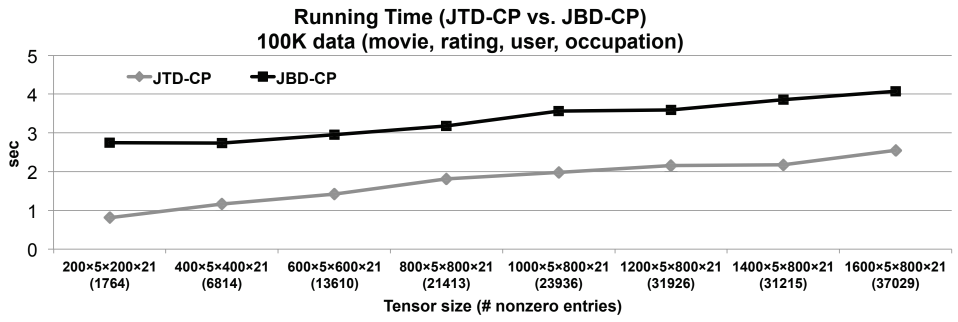
Figure 5.6: Running times of JTD-NWAY vs. JBD-NWAY for obtaining decompositions of the joined relations (a) `(user, rating, movie, genre)` and (b) `(movie, rating, user, occupation)` of MovieLens 100K data set

mentations of the `join-by-decomposition` and `join-then-decompose` algorithms. The total times reported in the plots includes all of the costs, including the time to compute the norms, which were negligible (< 0.01 sec) and, thus, are not shown separately. Since the data size is small, in this case, we are able to evaluate execution times with both dense (NWAY) and sparse (CP) tensor representations.

Dense Representation. Overall, JTD-NWAY has the slowest running time (see Figure 5.8). Since JBD-NWAY performs decompositions on tensors with much smaller number of modes, JBD-NWAY has a much lower decomposition cost than JTD-NWAY. As the tensor size increases, the time gain of JBD-NWAY over JTD-NWAY increases both for the `(user, rating, movie, genre)` and `(movie, rating, user, occupation)` relations (see Figure 5.6). Figure 5.8 shows how the



(a) `(user, rating, movie, genre)`



(b) `(movie, rating, user, occupation)`

Figure 5.7: Running times of JTD-CP vs. JBD-CP for obtaining decompositions of the joined relations (a) `(user, rating, movie, genre)` and (b) `(movie, rating, user, occupation)` of MovieLens 100K data set

execution time is distributed among the join and decomposition subtasks. As expected, for both schemes the cost is dominated by the cost of the decomposition step. In terms of join processing times, `join-by-decomposition` (JBD) is slightly faster than `join-then-decompose` (JTD). JBD does the join operation on the join factor matrices which always have smaller (or equal) number of modes than those of the input tensors which JTD does the join operation on.

Sparse Representation. Note that on this small data set, JTD-CP and JBD-CP are showing similar running times for decomposition: This is because CP-ALS, which assumes a sparse tensor model, is not as much affected by the number of tensor modes as the NWAY based implementation which relies on the dense tensor model. JBD-CP performs against JTD-CP differently, as predicted by the join selectivity

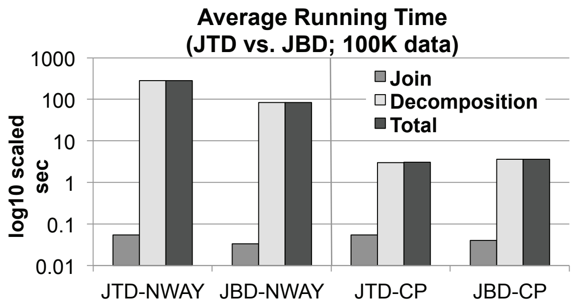
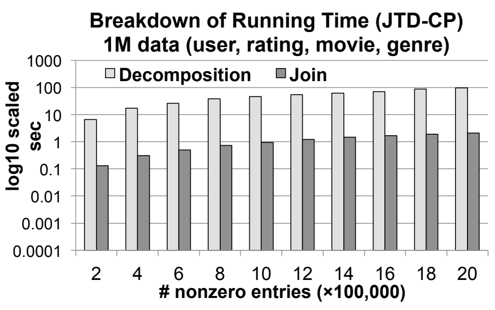


Figure 5.8: Breakdown of average running times (log10 scale) of each algorithm of `join-then-decompose` vs. `join-by-decomposition` for MovieLens 100K data set: both algorithms are dominated by the decomposition step

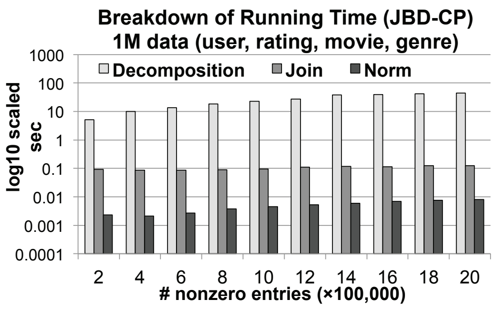
lower bound (see Table 7.10). For the `(user, rating, movie, genre)` relation (where $js > js_{\perp}$), JBD-CP outperforms JTD-CP as the number of nonzero entries increases (see Figure 5.7(a)); on the other hand, for the `(movie, rating, user, occupation)` relation (where $js < js_{\perp}$) as expected, JBD-CP is outperformed by JTD-CP (see Figure 5.7(b)). We next compare JTD-CP and JBD-CP for large data sets.

MovieLens 1M Data Set. Here we compare JTD and JBD decompositions based on sparse representations on two different 1M data sets.

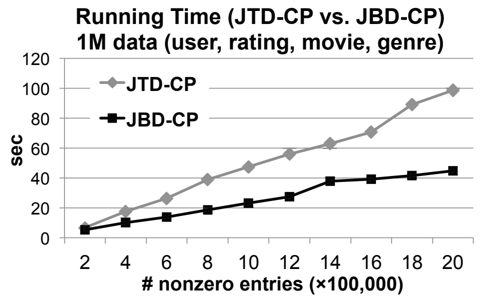
We first present the results for the `(user, rating, movie, genre)` relation. Figures 5.9(a) and (b) present the breakdowns of the average running times of JTD-CP and JBD-CP with respect to the number of nonzero entries of tensors in the MovieLens 1M data set without parallelization. Since (as shown in Table 7.10) for this data set we have $js > js_{\perp}$, we expect that the total running time of JTD-CP increases faster than that of JBD-CP as the tensor gets larger. This expectation is confirmed in the log10 scale plots of Figures 5.9(a) and (b). The decomposition step in both schemes dominates the running time even more than that in the result of the MovieLens 100K data set and the join processing time of `join-by-decomposition` (JBD) scales better with the larger data compared to that of `join-by-decomposition` (JTD). Note that



(a)



(b)

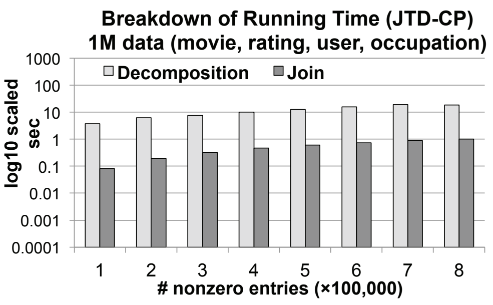


(c)

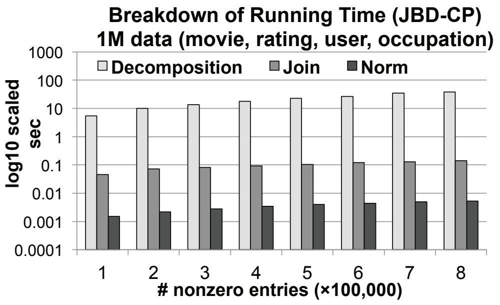
Figure 5.9: Breakdown of average running times (log10 scale) of (a) `join-then-decompose`, (b) `join-by-decomposition`, and (c) average total running times of `join-then-decompose` and `join-by-decomposition` without parallelization for obtaining the decomposition of the joined relation `(user, rating, movie, genre)` of MovieLens 1M data set

the norm computation overhead of JBD is negligible. Figure 5.9(c) reconfirms that the `join-by-decomposition` scheme scales much better than `join-then-decompose`.

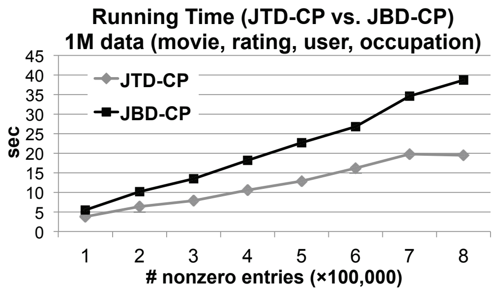
Figure 5.9 shows the execution time results for the `(movie, rating, user, occupation)` relation. Also in this case, similarly to the result of the `(user, rating, movie, genre)` relation in Figure 5.9, the decomposition cost is the most dominant component for both `join-by-decomposition` and `join-then-decompose` (see Figures 5.10(a) and (b)). On the other hand, for the `(movie, rating, user, occupation)` relation, we have $js < js_{\perp}$ (see Table 7.10) thus we expect that the decomposition cost of the `join-by-decomposition` will exceed that of the `join-then-decompose`. Indeed, as shown in Figure 5.10(c)), the running time of



(a)



(b)



(c)

Figure 5.10: Breakdown of average running times (log10 scale) of (a) `join-then-decompose`, (b) `join-by-decomposition`, and (c) average total running times of `join-then-decompose` and `join-by-decomposition` without parallelization for obtaining the decomposition of the joined relation `(movie, rating, user, occupation)` of MovieLens 1M data set

`join-by-decomposition` increases faster than that of the `join-then-decompose`. This confirms that the join selectivity based threshold can be used to decide when to use `join-by-decomposition` instead of `join-then-decompose`.

Multi-core Execution Time Results

MovieLens 100K Data Set. Figure 5.11 presents average multi-core running times for the MovieLens 100K data set, for both dense and sparse representations. The parallelized versions of the algorithms ran on 6 cores - i.e., in the case of rank-12 `join-by-decomposition` (JBD) each pair was assigned to a separate core. For the parallel strategies of `join-then-decompose` (JTD), we report the average of three

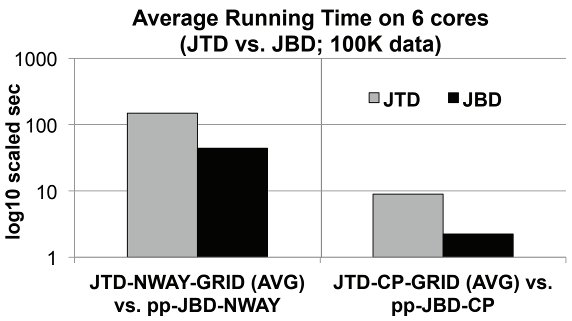


Figure 5.11: Average running times of JTD and JBD with parallelization on 6 cores for MovieLens 100K data set

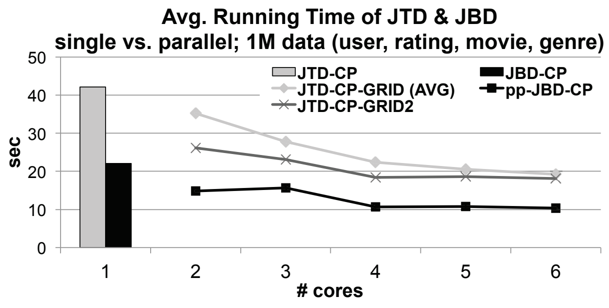
different partition schemes (see Table 7.8).

As can be seen in the figure, both JBD-NWAY (dense) and JBD-CP (sparse) benefit more from parallelization than JTD-NWAY (dense) and JTD-CP (sparse). It is also important to note that, while on a single core (as was reported in Figure 5.8) JTD-CP and JBD-CP take almost the same time, on 6 cores, JBD-CP significantly outperforms JTD-CP.

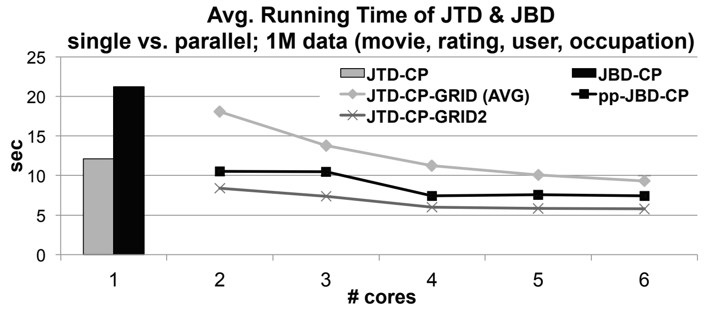
MovieLens 1M Data Set. In Figure 5.12, we compare the average running times of JTD-CP and JBD-CP with vs. without parallelization for the MovieLens 1M Data Set under sparse representation. For parallel JTD, we report the average running time for 3 different grid settings (JTD-CP-GRID (AVG)). We also report the best running time for all grid settings – which is the running time for the JTD-CP-GRID2 configuration [2] .

As Figure 5.12(a) shows, similarly to the result of the MovieLens 100K data set, JBD-CP performs better than JTD-CP-GRID (AVG) as well as JTD-CP-GRID2 for the `(user, rating, movie, genre)` relation (where $js > js_{\perp}$). A very interesting result (reported in Figure 5.12(b)) is that also on the `(movie, rating, user,`

[2]We conjecture that this is because, with sparse tensors (where the memory requirement is small), the gains from higher core utilization due to the smaller sizes sub-tensors may be lower than the increase in the communication overhead due to the larger number of sub-tensors.



(a) `(user, rating, movie, genre)`

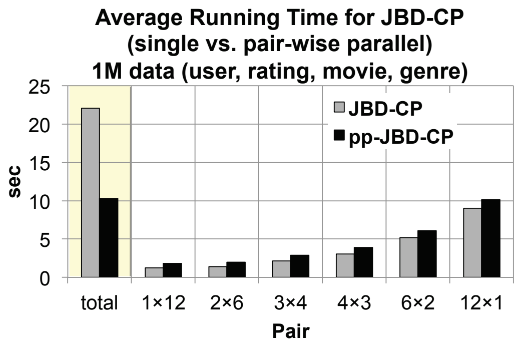


(b) `(movie, rating, user, occupation)`

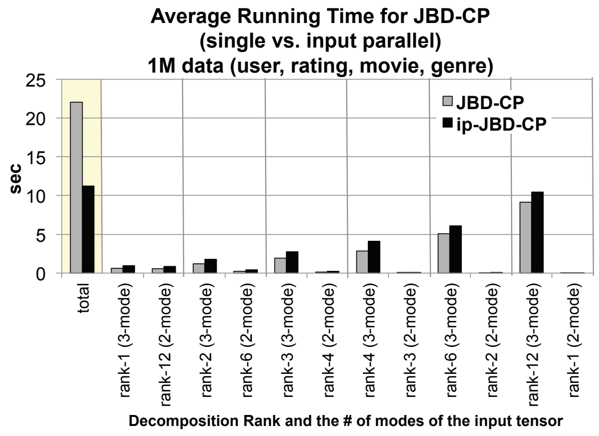
Figure 5.12: Average running times of JTD-CP and JBD-CP on single vs. multiple numbers of cores for obtaining the decompositions of the joined relations (a) `(user, rating, movie, genre)` and (b) `(movie, rating, user, occupation)` of MovieLens 1M data set

`occupation)` relation (where $js < js_\perp$ and, hence, JTD-CP outperforms JBD-CP on a single core), JBD-CP performs better than JTD-CP-GRID (AVG) when given more cores. Even the best of the all JTD-CP configurations, JTD-CP-GRID2, performs only slightly better than JBD-CP with more cores. This drop in performance for the JTD based approaches in multi-core architecture is due to the increased communication overhead, which is avoided by the JBD-based schemes.

It is also interesting to note that the performance of the algorithms saturates with around 4 cores. This is explained in Figures 5.13(a) and (b), which show the average running times for two different parallelization strategies presented in Section 5.3.6 as well as the distributions of the execution times for different sub-tasks:



(a) JBD (single core) vs. pp-JBD results



(b) JBD (single core) vs. ip-JBD results

Figure 5.13: Total and sub-task execution times for `join-by-decomposition` (a) when each pair is assigned to a separate core (pp-JBD) and (b) when each decomposition is assigned to a separate core (ip-JBD) for obtaining the decomposition of the joined relation `(user, rating, movie, genre)` of MovieLens 1M data set. The figure also shows the total and sub-task execution times when `join-by-decomposition` is running on a single core (JBD).

- As can be seen here, the two parallelization strategies are comparable in execution time: in fact, the per-decomposition parallelization strategy is slightly slower than per-pair parallelization (due to increased parallelization overheads), but the difference between the two schemes is negligible.

- For both pp-JBD and ip-JBD schemes, there is a parallelization overhead for the sub-tasks due to data movement among the cores.

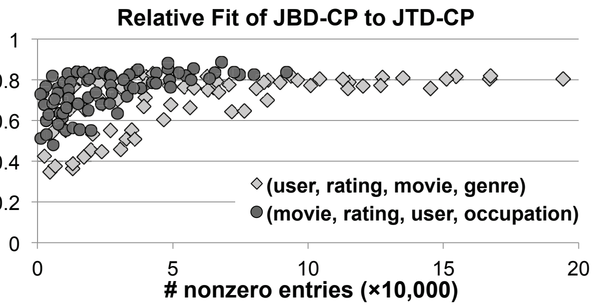


Figure 5.14: Relative fit of JBD-CP to JTD-CP for obtaining the decompositions of the joined relations (`user, rating, movie, genre`) and (`movie, rating, user, occupation`) of MovieLens 100K data set

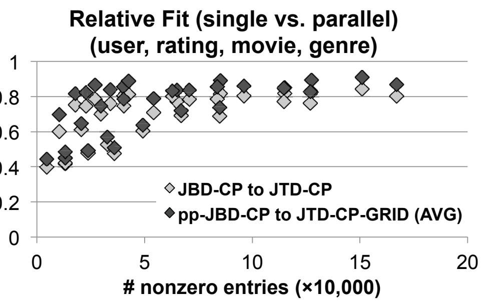
- The distributions of the costs of the different tasks are not uniform – for this data set, the pair (rank-12, rank-1) is the costliest task and dominates the overall execution time. As a result, as shown in Figures 5.13(a) and (b), the same execution time speed-up as 6 cores can be obtained using only 4 cores by assigning more than one of the cheaper tasks onto one single shared core.

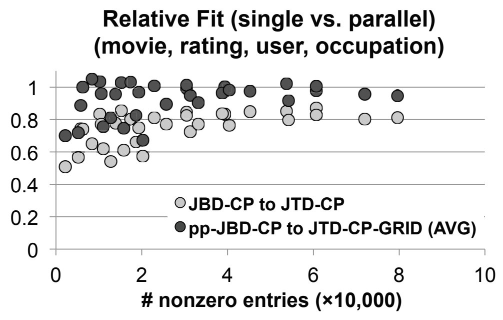
Note that when there are more cores available, it would be possible to further divide the work of the pair (rank-12, rank-1) to smaller chunks and assign to more cores to achieve further speed-ups. We will consider further parallelization of individual pairs (using a block-based decomposition, such as the grid NTF, or using JBD in a hierarchical manner) in our future work.

Accuracy Results

In this subsection, we compare the accuracy of `join-by-decomposition` to `join-then-decompose`. Note that we focus on accuracy results for JTD-CP and JBD-CP (results for JTD-NWAY and JBD-NWAY are similar). We use the MovieLens 100K data set for fit measurement for both algorithms since fit computation for the MovieLens 1M data set requires more main memory than is available.

We first present the relative fit (see Equation (8.8)) of JBD-CP to JTD-CP with





(a) (b)

Figure 5.15: Relative fit of JBD-CP to JTD-CP and pp-JBD-CP to JTD-CP-GRID (AVG) (the average fit of all different grid settings with parallelization) for obtaining the decompositions of the joined relations (a) `(user, rating, movie, genre)` and (b) `(movie, rating, user, occupation)` of MovieLens 100K data set

respect to the number of nonzero entries of the joined tensor (see Figure 5.14). The relative fit increases as the number of nonzero entries increases, getting higher than 0.8 for both `(user,rating,movie,genre)` and `(movie,rating,user,occupation)` relations. This result shows that `join-by-decomposition` works quite consistently, not being affected by the increase of the number of nonzero entries while the quality of `join-then-decompose` degenerates more severely for the larger tensors. Note that the relative fit is slightly higher for `(movie,rating,user,occupation)` relation which is likely to have higher independence among the clusters of its input relations than those of the `(user,rating,movie,genre)` relation: a movie rating of a user is more likely to be affected by the movie genre than the user occupation. This confirms the basic premise of the proposed `join-by-decomposition` scheme that the algorithm is likely to work better when the clusters from each decomposition are more independent from each other.

Interestingly, as Figure 5.15 shows, the relative fit increases in the case of parallelized execution, *even exceeding 1.0 in some cases.* This is because the accuracy of `join-then-decompose` can degenerate when the input tensor is divided into a grid of sub-tensors for parallelization, whereas `join-by-decomposition` does not suffer

Table 5.9: Input relations and joined relations for JBD-Tucker experiment

Data set	1st input relation ($\mathcal{P}$)	2nd input relation ($\mathcal{Q}$)	Join mode	Joined relation ($\mathcal{X}$)
MovieLens 1M	`(user,rating,movie)`	`(movie, genre)`	`(movie)`	`(user,rating,movie,genre)`
	`(movie,rating,user)`	`(user,occupation)`	`(user)`	`(movie,rating,user,occupation)`
Enron	`(time,sender,recipient)`	`(recipient, recipient's position)`	`(recipient)`	`(time,sender,recipient, recipient's position)`

Table 5.10: Statistics of the joined relations for JBD-Tucker experiment

Data set	Joined relation	Tensor sizes	#nonzero entries
MovieLens 1M	`(user, rating, movie, genre)`	$6000 \times 5 \times 3800 \times 18$	1825034
	`(movie, rating, user, occupation)`	$3800 \times 5 \times 4000 \times 21$	819185
Enron	`(time, sender, recipient, recipient's position)`	$5632 \times 184 \times 184 \times 8$	34257

from such degradations during parallelization.

5.4.4 JBD-Tucker Experiments

As discussed in Section 5.3, we extended JBD to Tucker decomposition (JBD-Tucker). We, therefore, evaluate JBD-Tucker in Section 5.4.4 using the `MovieLens 1M` data set obtained from [56] and the `Enron` data set [61] (Table 5.9):

- The `MovieLens 1M` data set consists of 1 million ratings from 6,000 users on 4,000 movies. In addition to the ratings information, this data set also includes various movie metadata, such as movie genre, and user metadata, such as user occupation. From this data set, we created two pairs of relational tensors:
  - The first data set includes a 3-mode `(user, movie, rating)` and a 2-mode `(movie, genre)`; these join into a 4-mode `(user, rating, movie, genre)` tensor.
  - The second set includes a 3-mode `(movie, user, rating)` and a 2-mode `(user, occupation)` tensors; these join into a 4-mode `(movie, rating, user, occupation)` tensor.

Table 5.11: Algorithms

Algorithm	Description
Tucker-ALS	Tucker-ALS algorithm
MET*	"*" modes element-wise Memory-Efficient Tucker
JTD-Tucker-ALS	JTD-Tucker using Tucker-ALS
JTD-MET*	JTD-Tucker using MET*
JBD-Tucker-ALS	JBD-Tucker using Tucker-ALS
JBD-MET*	JBD-Tucker using MET*

- The `Enron` data set consists of email exchanges among 184 email addresses during 5,632 days. This data set is used to create a 3-mode relation `(time, sender, recipient)` and a 2-mode relation `(recipient, recipient's position)`. These two relations join into a 4-mode relation, `(time, sender, recipient, recipient's position)`.

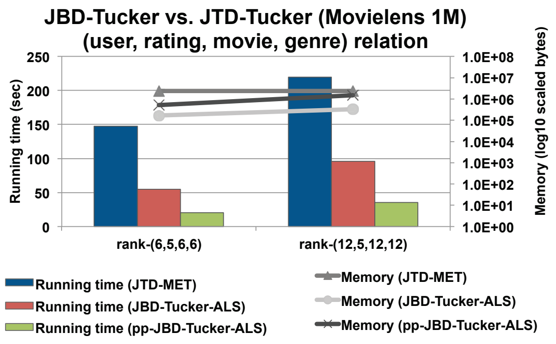
Data tensor dimensions and the number of nonzero entries are shown in Table 5.10.

Tucker Decomposition Algorithms (Single Core)

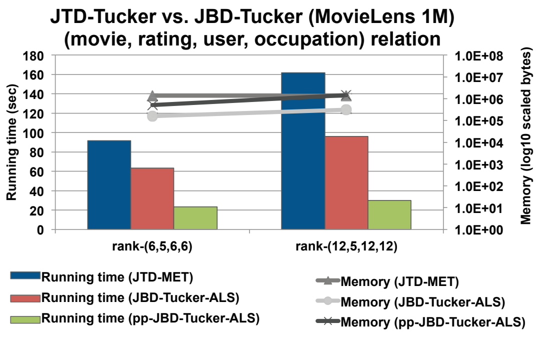
Conventional Tucker decomposition algorithms, such as [9], quickly become ineffective on dense data sets. Therefore, we focus on Tucker decompositions of sparse data sets. We consider Tucker-ALS algorithm which is available in the Tensor Toolbox for MATLAB [12]. We also use MET (Memory-Efficient Tucker) in [45]. For MET, there are multiple variations that we denote MET* according to how many modes are handled element-wise; the number of modes handled element-wise is denoted by "*". Tucker-ALS and MET* are also used as the base Tucker algorithm for JBD-Tucker; these are referred to as JBD-Tucker-ALS and JBD-MET*, respectively.

Tucker Decomposition Algorithms (Parallel, Multi-core)

Since neither of the conventional decomposition algorithms, Tucker-ALS and MET, supports parallelization, we only consider parallelization of JBD-Tucker. In partic-



(a)



(b)

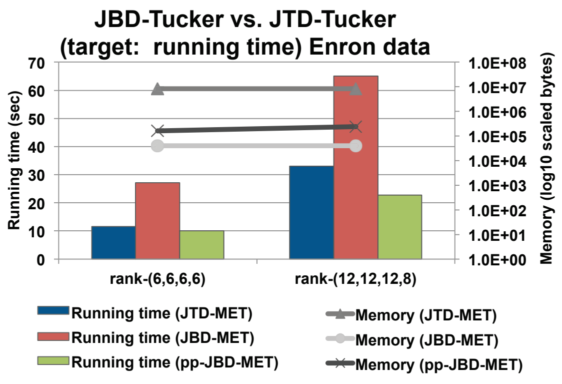
Figure 5.16: Running time and bottleneck memory consumption of JBD-Tucker vs. JTD-Tucker for obtaining decompositions of the joined relations (a) `(user, rating, movie, genre)` and (b) `(movie, rating, user, occupation)` of `MovieLens 1M` data set

ular, we consider two parallelizations: JBD-Tucker-ALS and JBD-MET*, which are referred to as pp-JBD-Tucker-ALS and pp-JBD-MET* respectively.

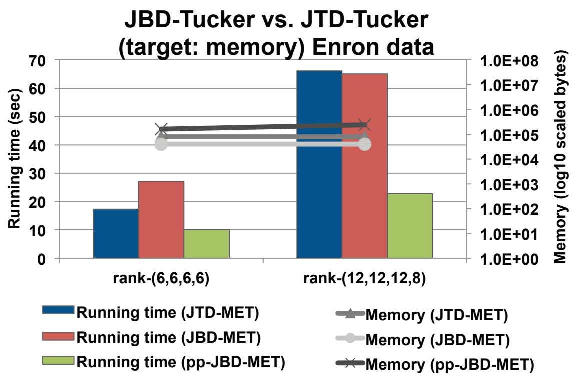
Results: JBD for Tucker Decomposition (JBD-Tucker)

As discussed earlier, we extended JBD to Tucker decompositions (JBD-Tucker). We first evaluate the efficiency and effectiveness of the JBD against the conventional join-then-decompose (JTD) approach. Since we extended JBD to Tucker decompositions, our experiments focus on Tucker decomposition (JBD-Tucker vs. JTD-Tucker) (see [**?**] for the experimental result for JBD-CP vs. JTD-CP). Especially for dense tensors and Tucker decomposition, memory usage can be the major bottleneck. Thus we report the maximum intermediate memory use provided by the MET* algorithm.

Figure 5.16 shows the running times and the memory consumption of JBD-Tucker (single core and parallel) and JTD-Tucker schemes for the two relations of `MovieLens 1M` data set. Since both JBD-Tucker and JTD-Tucker can be implemented using different Tucker implementations, each with different memory consumption, the figure displays only best performing alternative for JTD-Tucker (JTD-MET1), single-core



(a)



(b)

Figure 5.17: Running time and bottleneck memory consumption of JBD-Tucker vs. JTD-Tucker for `Enron` data set using configurations that provide (a) (target: running time) and (b) (target: memory)

JBD-Tucker (JBD-Tucker-ALS), and parallel JBD-Tucker (pp-JBD-Tucker-ALS). As we see in the figure, JBD-Tucker schemes outperform JTD-Tucker both in terms of execution time and memory. Moreover, the parallelized version of JBD-Tucker can further bring the running time cost significantly down (though at the expense of increased memory consumption) relative to the single-core JBD-Tucker. Note also that the advantage of the JBD-Tucker is especially pronounced for the `(user, rating, movie, genre)` relation, which is denser than the `(movie, rating, user, occupation)` relation (0.089% vs. 0.051%). This confirms our observation that JBD-Tucker is especially useful in cases where JTD-Tucker is likely to fail.

As we see in Figures 5.17 (a – "running time" targeted configurations) and (b – " memory" targeted configurations), on the other hand, the `Enron` data set is challenging for JBD-Tucker: this is because, as discussed in Section 7.5, the reduction in the number of modes when using JBD-Tucker is only one and the savings obtained through JBD-Tucker (in this case JBD-MET2) does not amortize the additional overheads to outperform JTD-Tucker in terms of execution time (JTD-MET2 in Figure 5.17(a) and JTD-MET3 in Figure 5.17(b)). However, when the parallelization

Table 5.12: Relative fit of JBD-Tucker to JTD-Tucker of the joined relations

Data set	Joined relation	rank-6	rank-12
MovieLens 1M	`(user, rating, movie, genre)`	0.8534	0.7685
	`(movie, rating, user, occupation)`	0.9569	0.9457
Enron	`(time, sender, recipient, recipient's position)`	0.8919	0.7026

opportunities provided by JBD-Tucker are leveraged, JBD-Tucker easily outperforms JTD-Tucker. Moreover, in terms of memory consumption, JBD-Tucker outperforms JTD-Tucker.

Finally, Table 5.12 shows the relative fit (see Equation **??**) of JBD-Tucker relative to JTD-Tucker: the relative fit is bigger than 0.7 for all cases and for the `(movie, rating, user, occupation)` relation, it reaches to ~ 0.95.

Chapter 6

UNION-BY-DECOMPOSITION (UBD): PUSHING-DOWN TENSOR DECOMPOSITION STRATEGY TO PROMOTE REUSE OF MATERIALIZED DECOMPOSITIONS

From data collection to decision making, the life cycle of data often involves many steps of integration, manipulation, and analysis. To be able to provide end-to-end support for the full data life cycle, todays data management and decision making systems increasingly combine operations for data manipulation, integration as well as data analysis. Tensor-relational model (TRM) is a framework proposed to support both relational algebraic operations (for data manipulation and integration) and tensor algebraic operations (for data analysis). In this chapter, we consider joint processing of relational algebraic and tensor analysis operations. In particular, we focus on data processing workflows that involve data integration from multiple sources (through unions) and tensor decomposition tasks. While, in traditional relational algebra, the costliest operation is known to be the join, in a framework that provides both relational and tensor operations, tensor decomposition tends to be the computationally costliest operation. Therefore, it is most critical to reduce the cost of the tensor decomposition task by manipulating the data processing workflow in a way that reduces the cost of the tensor decomposition step. Therefore, in this chapter, we consider data processing workflows involving tensor decomposition and union operations and we propose a novel scheme for pushing down the tensor decompositions over the union operations to reduce the overall data processing times and to promote reuse of materialized tensor decomposition results. Experimental results confirm the efficiency and effectiveness of the proposed scheme.

6.1 Introduction

As a higher-order generalization of matrices, tensors provide a suitable data representation for multidimensional data sets and tensor decomposition (which is a higher-order generalization of SVD/PCA for multi-aspect data analysis) helps capture the higher-order latent structure of such datasets. Consequently, the tensor data model is increasingly being used by many application domains including scientific data management [19, 34, 59, 74], sensor data management [73], and social network data analysis [45, 47, 58]. On the other hand, from data collection to decision making, the *life cycle* of data often involves many steps of integration, manipulation, and analysis. Therefore, to be able to provide end-to-end support for the full data life cycle, today's data management and decision making systems increasingly need to combine different types of operations for data manipulation, integration, and analysis.

The tensor-relational model (TRM) brings relational algebraic operations (for data manipulation and integration) and tensor algebraic operations (for data analysis) together and supports complex data processing plans where multiple relational algebraic and tensor algebraic operations are composed with each other.

In this chapter, *we focus on query plans that involve tensor decomposition and union operations (as in Figure 4.4(a)) and propose novel* decomposition push-down *strategies (as in Figure 4.4(b)) that help reduce the overall cost of the query plan.* We refer to the query plan that first performs the union operation on the data and then applies the tensor decomposition on the union of the data as *union-then-decompose* (UTD) plan. The query plan with decomposition push-down, which first performs the tensor decompositions on each input data source and then combines these decomposed tensors as the *union-by-decomposition* (UBD) plan.

6.1.1 Contributions

A *union-by-decomposition* (UBD) plan, with decomposition push-down, has various advantages over the conventional *union-then-decompose* (UTD) plan:

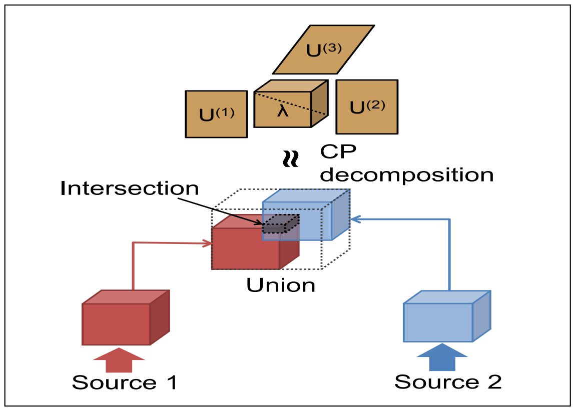
- Firstly, especially when the overlaps between the input data sources are small, the union operation can combine relatively small and sparse tensors into a larger and denser tensor. Consequently, the decomposition over the union data can be much more expensive than the decompositions over the input data sources. Moreover multiple tensor decompositions on input tensors can run in parallel, which will further reduce the cost.

- Secondly, a *union-by-decomposition* (UBD) based plan provides opportunities for materializing decomposition of data tensors and re-using these materialized decompositions in more complex queries requiring integration of data.

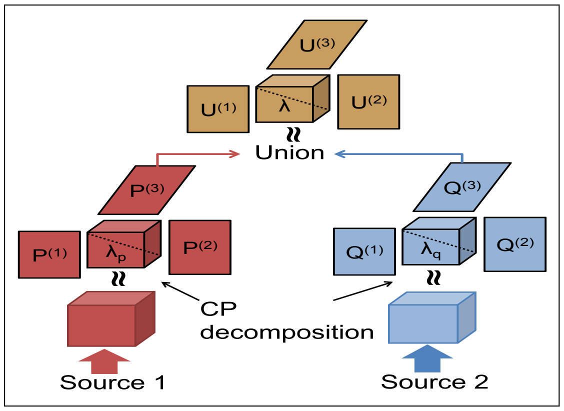
Despite these advantages, however, implementing the UBD strategy requires us to address a number of key challenges:

- **Challenge 1: How can we combine the factor matrices of tensor decompositions with their own eigen basis into the eigen basis of the union tensor?** If tensor decomposition is thought of as a group of clusters, combining different groups of clusters for different tensors into another group of clusters for the union of the tensors is not straightforward.

- **Challenge 2: For the common data elements at the intersection of multiple data sources, which factors (clusters when the clustering analogy is used) among the different tensor decompositions should we choose?** This is critical as the choice can impact the final accuracy of the UBD based plan.



(a) Union-then-Decompose (UTD)



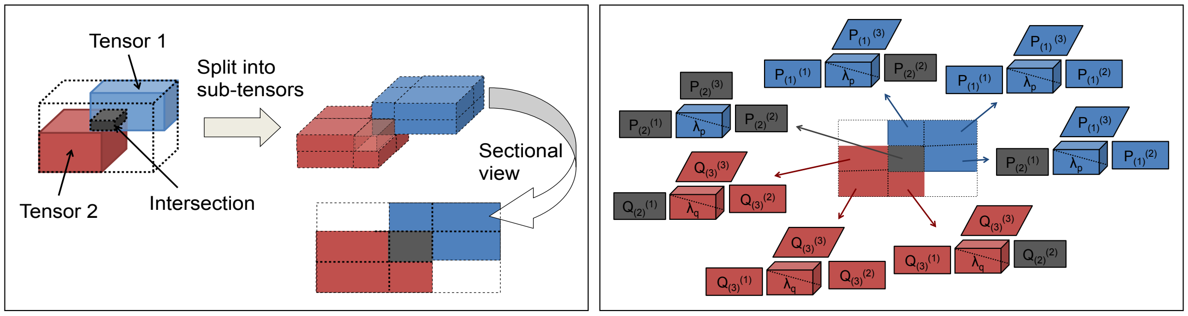
(b) Union-by-Decomposition (UBD)

Figure 6.1: (a) Tensor decomposition on the union of the two relations and (b) the union operation on the two tensor decompositions of the input relations

In this chapter, we present algorithms and techniques to address these questions. In Section 6.2, we extend TRM with the proposed *union-by-decomposition* operation: we discuss strategies for combining the tensor decompositions for the union of the tensors from different sources and consider alternative selection measures to choose a group of factors for data entries common to input data sources. We also consider query plans that include both join and union operations along with tensor decomposition. We, then, experimentally evaluate the proposed scheme in Section 6.4.

6.2 Union-by-Decomposition (UBD) and Decomposition Push-Down

In this section, we describe our proposed union-by-decomposition (UBD) approach that pushes down tensor decompositions over union operators: Unlike the more conventional union-then-decompose (UTD) scheme, which applies decomposition on the union of the two relations (Figure 6.1(a)), UBD first performs the tensor decomposition on the input tensors then these decompositions are combined into the final result (Figure 6.1(b)).



(a) (b)

Figure 6.2: Naive grid-based UBD: (a) Input tensors are partitioned into an intersecting sub-tensor and non-intersecting sub-tensors; (b) intermediary decompositions of grid-based UBD

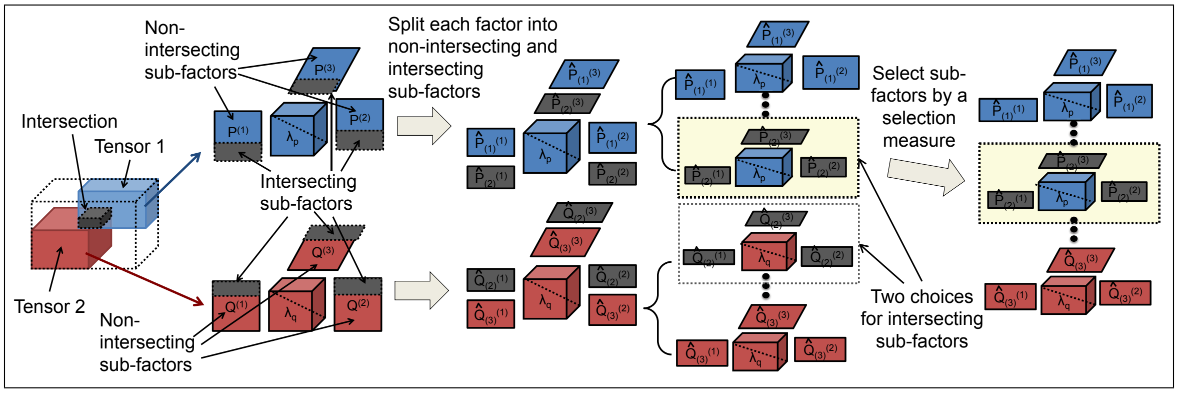
6.2.1 *Challenge 1: Implementing UBD through Partition-based ALS*

Naive Grid-based UBD.

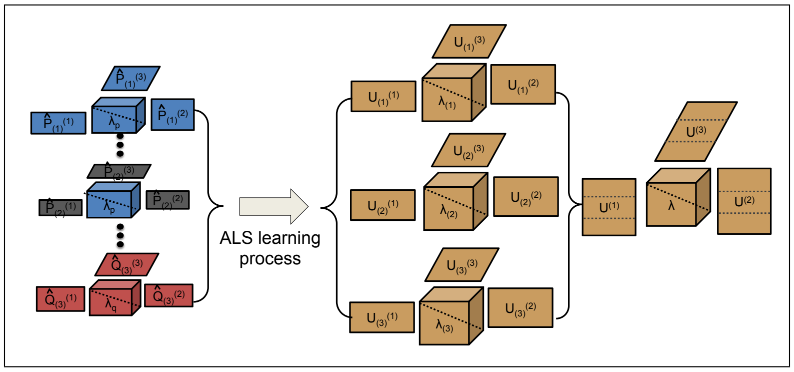
One way to implement the UBD operation is to divide the input tensors into *common (or intersection)* and ($2^N - 1$ many when the number of modes is N) *uncommon* sub-tensors as shown in Figure 6.2(a) and then considering each partition as a cell of a larger tensor partitioned into a grid as shown in Figure 6.2(b) and applying the grid-based tensor decomposition strategy proposed in [59] to combine these into a single decomposition.

Proposed Implementation of UBD.

An obvious shortcoming of the naive grid-based UBD discussed above is that it leads to a very large number of intermediary decompositions and this number increases quickly with the number of modes of the input tensors. To tackle this challenge, we propose to decompose input tensors directly (through decomposition push-down) and recombine the resulting factor matrices in a way that reflects the common and non-intersecting sub-factors of these decompositions as shown in Figure 6.3. The high-level pseudocode of this partition-based UBD scheme is shown in Algorithm 1.


Non-intersecting sub-factors
Split each factor into non-intersecting and intersecting sub-factors
Intersection
Tensor 1
Intersecting sub-factors
Tensor 2
Non-intersecting sub-factors
Select sub-factors by a selection measure
Two choices for intersecting sub-factors

(a)



(b)

Figure 6.3: UBD: (a) first the inputs tensors are decomposed and (b) these decompositions are recombined by considering the common and non-intersection parts of the factor matrices.

We next present the details of the proposed UBD process:

Let us assume that we are given two tensors $\mathcal{P}_{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathcal{Q}_{J_1 \times J_2 \times \cdots \times J_N}$ and let us assume we have already computed their CP decompositions

$$CP(\mathcal{P}) = \hat{\mathcal{P}} = \langle \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(N)} \rangle \quad \text{and} \quad CP(\mathcal{Q}) = \hat{\mathcal{Q}} = \langle \mathbf{Q}^{(1)}, \ldots, \mathbf{Q}^{(N)} \rangle. \tag{6.1}$$

Our goal is to estimate $CP(\mathcal{P} \cup \mathcal{Q}) = \langle \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \rangle$ efficiently using these decompositions. To achieve this, we solve the ALS problem

$$\min \|(\mathcal{P} \cup \mathcal{Q}) - \langle \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \rangle\| \tag{6.2}$$

by appropriately combining sub-factors of the input tensors. More specifically, each factor of $\mathcal{P}$ and $\mathcal{Q}$ are split into two: a non-intersecting ($\mathbf{P}^{(n)}_{(1)}$ and $\mathbf{Q}^{(n)}_{(3)}$) and intersect-

Algorithm 1: Union-By-Decomposition (UBD) (input: two tensors $\mathcal{P}_{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathcal{Q}_{J_1 \times J_2 \times \cdots \times J_N}$, optional input: CP decompositions of $\mathcal{P}$ and $\mathcal{Q}$, $\langle \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(N)} \rangle$ and $\langle \mathbf{Q}^{(1)}, \ldots, \mathbf{Q}^{(N)} \rangle$, respectively, output: factors $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$ for $\mathcal{P} \cup \mathcal{Q}$)

```
1: if no existing decompositions given then
2:         Run any available CP algorithm on $\mathcal{P}$ and $\mathcal{Q}$ *in parallel* to get factors $\mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(N)}$ and
           $\mathbf{Q}^{(1)}, \ldots, \mathbf{Q}^{(N)}$
3: end if
4: for each mode $n$ do
5:     create sub-factors $\hat{\mathbf{P}}^{(n)}_{(1)}$ and $\hat{\mathbf{P}}^{(n)}_{(2)}$,
       and $\hat{\mathbf{Q}}^{(n)}_{(2)}$ and $\hat{\mathbf{Q}}^{(n)}_{(3)}$ with non-intersecting and intersecting sub-factors of $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$, respectively
       (see Figure 6.3(a))
6: end for
7: select either $\hat{\mathbf{P}}^{(n)}_{(2)}$ and $\hat{\mathbf{Q}}^{(n)}_{(2)}$ for factors $\mathbf{T}^{(n)}$ for intersection $\mathcal{P} \cap \mathcal{Q}$ by a selection measure (see Section 6.2.2)
8: repeat the update process for sub-factors $\mathbf{U}^{(n)}_{(1)}$, $\mathbf{U}^{(n)}_{(2)}$, and $\mathbf{U}^{(n)}_{(3)}$ using Equation 6.7 until a stopping
   condition is satisfied, which are combined to $\mathbf{U}^{(n)}$ by Equation 6.5
```

ing ($\mathbf{P}^{(n)}_{(2)}$ and $\mathbf{Q}^{(n)}_{(2)}$) partitions. Given these, the CP decompositions of $[k_1, k_2, \ldots, k_N]$-th sub-tensor of $\mathcal{P}$ and $\mathcal{Q}$ are

$$CP(\mathcal{P}^{(\bar{\mathbf{k}})}) = \langle \mathbf{P}^{(1)}_{(k_1)}, \ldots, \mathbf{P}^{(N)}_{(k_N)} \rangle \quad \text{and} \quad CP(\mathcal{Q}^{(\bar{\mathbf{k}})}) = \langle \mathbf{Q}^{(1)}_{(k_1)}, \ldots, \mathbf{Q}^{(N)}_{(k_N)} \rangle, \tag{6.3}$$

respectively, where $\bar{\mathbf{k}} = [k_1, k_2, \ldots, k_N]$ for $k_n \in \{1, 2\}$ for $\mathcal{P}^{(\bar{\mathbf{k}})}$ and $k_n \in \{2, 3\}$ for $\mathcal{Q}^{(\bar{\mathbf{k}})}$. Given these, we can approximate the decompositions of each sub-tensor of $\mathcal{P}$ and $\mathcal{Q}$ with the CP decompositions of $\mathcal{P}$ and $\mathcal{Q}$, respectively (see Figure 6.3(a)):

$$CP(\mathcal{P}^{(\bar{\mathbf{k}})}) \approx \langle \hat{\mathbf{P}}^{(1)}_{(k_1)}, \ldots, \hat{\mathbf{P}}^{(N)}_{(k_N)} \rangle \quad \text{and} \quad CP(\mathcal{Q}^{(\bar{\mathbf{k}})}) \approx \langle \hat{\mathbf{Q}}^{(1)}_{(k_1)}, \ldots, \hat{\mathbf{Q}}^{(N)}_{(k_N)} \rangle. \tag{6.4}$$

Let us denote the CP decomposition of $[k_1, k_2, \ldots, k_N]$-th sub-tensor of $\mathcal{P} \cup \mathcal{Q}$ as

$$CP((\mathcal{P} \cup \mathcal{Q})^{(\bar{\mathbf{k}})}) = CP(\mathcal{Y}^{(\bar{\mathbf{k}})}) = \langle \mathbf{U}^{(1)}_{(k_1)}, \ldots, \mathbf{U}^{(N)}_{(k_N)} \rangle,$$

where $\bar{\mathbf{k}} = [k_1, k_2, \ldots, k_N]$ for $k_n \in \{1, 2, 3\}$. Note that each factor of $CP(\mathcal{P} \cup \mathcal{Q})$ can be split into three partitions

$$\mathbf{U}^{(n)} = [\mathbf{U}_{(1)}^{(n)T} \mathbf{U}_{(2)}^{(n)T} \mathbf{U}_{(3)}^{(n)T}]^T, \tag{6.5}$$

one corresponding to a non-intersecting sub-factor from one input matrix, the other corresponding to a common sub-factor, and the last corresponding to a non-intersecting sub-factor from the second input matrix. Given these, we can reformulate the minimization problem in Equation (6.2) for each sub-tensor $\boldsymbol{\mathcal{Y}}^{(\bar{\mathbf{k}})}$ of $\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}$ as minimizing D, where

$$D = \frac{1}{2} \sum_{k_1=1}^{3} \cdots \sum_{k_N=1}^{3} \|\boldsymbol{\mathcal{Y}}^{(\bar{\mathbf{k}})} - \langle \mathbf{U}_{(k_1)}^{(1)}, \ldots, \mathbf{U}_{(k_N)}^{(N)} \rangle\|,$$

or, considering the n-mode matricized tensor $\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})}$ of $\boldsymbol{\mathcal{Y}}^{(\bar{\mathbf{k}})}$, as minimizing

$$D = \frac{1}{2} \sum_{\bar{\mathbf{k}}} \|\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})} - \mathbf{U}_{(k_n)}^{(n)} \{\mathbf{U}_{(k_1)}^{(1)} \odot \mathbf{U}_{(k_2)}^{(2)} \odot \cdots \odot \mathbf{U}_{(k_{n-1})}^{(n-1)} \odot \mathbf{U}_{(k_{n+1})}^{(n+1)} \odot \cdots \odot \mathbf{U}_{(k_N)}^{(N)}\}\|,$$

where $\odot$ is the Khatri-Rao product.

This minimization problem can be solved using an ALS problem by identifying gradient components with respect to sub-factors as in [59]. More specifically, the gradient component with respect to sub-factor $\mathbf{U}_{(k_n)}^{(n)}$ is

$$\begin{aligned}
\Delta_{\mathbf{U}_{(k_n)}^{(n)}} D &= \sum_{\bar{\mathbf{k}}_n = k_n} \left(-\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})} \mathbf{U}_{(\bar{\mathbf{k}})}^{\odot - n} + \mathbf{U}_{(k_n)}^{(n)} \mathbf{U}_{(\bar{\mathbf{k}})}^{\odot - n^T} \mathbf{U}_{(\bar{\mathbf{k}})}^{\odot - n} \right) \\
&= \sum_{\bar{\mathbf{k}}_n = k_n} \left(-\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})} \mathbf{U}_{(\bar{\mathbf{k}})}^{\odot - n} + \mathbf{U}_{(k_n)}^{(n)} \{\mathbf{U}_{(\bar{\mathbf{k}})}^{T} \mathbf{U}_{(\bar{\mathbf{k}})}\}^{\circledast - n} \right),
\end{aligned} \tag{6.6}$$

where $\circledast$ is the Hadamard (element-wise) product. Given this, each sub-factor $\mathbf{U}_{(k_n)}^{(n)}$ can be updated using the update rule

$$\mathbf{U}_{(k_n)}^{(n)} \leftarrow \left(\sum_{\bar{\mathbf{k}}_n = k_n} \mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})} \mathbf{U}_{(\bar{\mathbf{k}})}^{\odot - n} \right) \left(\sum_{\bar{\mathbf{k}}_n = k_n} (\mathbf{U}_{(\bar{\mathbf{k}})}^{T} \mathbf{U}_{(\bar{\mathbf{k}})})^{\circledast - n} \right)^{-1}. \tag{6.7}$$

Note that, from Equation 6.4, for each sub-tensor $\mathcal{Y}^{(\bar{\mathbf{k}})} = \mathcal{P}^{(\bar{\mathbf{k}})}$, considering to the first input matrix we have

$$\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot -n} \approx \hat{\mathbf{P}}_{(k_n)}^{(n)}\hat{\mathbf{P}}_{(\bar{\mathbf{k}})}^{\odot -n T}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot(-n)}. \tag{6.8}$$

Similarly, for each sub-tensor $\mathcal{Y}^{(\bar{\mathbf{k}})} = \mathcal{Q}^{(\bar{\mathbf{k}})}$, considering to the second input matrix, we have

$$\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot -n} \approx \hat{\mathbf{Q}}_{(k_n)}^{(n)}\hat{\mathbf{Q}}_{(\bar{\mathbf{k}})}^{\odot -n T}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot(-n)}. \tag{6.9}$$

Finally, for each sub-tensor $\mathcal{Y}^{(\bar{\mathbf{k}})}$ such that $\mathcal{Y}^{(\bar{\mathbf{k}})} = \mathcal{P} \cap \mathcal{Q}$,

$$\mathbf{Y}_{(n)}^{(\bar{\mathbf{k}})}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot -n} \approx \mathbf{T}^{(n)}\mathbf{T}^{\odot -n T}\mathbf{U}_{(\bar{\mathbf{k}})}^{\odot(-n)}, \tag{6.10}$$

where $\mathbf{T}^{(n)}$ are the factors of $CP(\mathcal{P} \cap \mathcal{Q})$. Note that $\mathbf{T}^{(n)}$ can be estimated from either the CP decomposition of $\mathcal{P}^{(\bar{\mathbf{2}})}$

$$CP(\mathcal{P} \cap \mathcal{Q}) = CP(\mathcal{P}^{(\bar{\mathbf{2}})}) \approx \langle \hat{\mathbf{P}}_{(2)}^{(1)}, \ldots, \hat{\mathbf{P}}_{(2)}^{(N)} \rangle,$$

where $\bar{\mathbf{2}} = [k_1, k_2, \ldots, k_N]$ for all $k_n = 2$, or the CP decomposition of $\mathcal{Q}^{(\bar{\mathbf{2}})}$

$$CP(\mathcal{P} \cap \mathcal{Q}) = CP(\mathcal{Q}^{(\bar{\mathbf{2}})}) \approx \langle \hat{\mathbf{Q}}_{(2)}^{(1)}, \ldots, \hat{\mathbf{Q}}_{(2)}^{(N)} \rangle.$$

The choice is critical and can impact significantly on the accuracy of the overall process. Therefore, we next discuss how to select whether to use $\hat{\mathbf{P}}_{(2)}^{(n)}$ or $\hat{\mathbf{Q}}_{(2)}^{(n)}$ to estimate $\mathbf{T}^{(n)}$.

6.2.2 Challenge 2: Selection of Sub-Factors for the Overlapping Sub-Tensor

As described above, the factors $\mathbf{T}^{(n)}$ of the overlapping sub-tensor, $\mathcal{P} \cap \mathcal{Q}$ (used in the computation of $CP(\mathcal{P} \cup \mathcal{Q})$) can be selected from either $\hat{\mathbf{P}}_{(2)}^{(n)}$ or $\hat{\mathbf{Q}}_{(2)}^{(n)}$. As also explained before, the choice is critical as it may impact the accuracy of the final decomposition, $CP(\mathcal{P} \cup \mathcal{Q})$. Therefore, in this subsection, we explore alternative ways for choosing the sub-factors, $\mathbf{T}^{(n)}$, of $CP(\mathcal{P} \cap \mathcal{Q})$.

Intersection-based selection criteria.

When we are choosing between $\hat{\mathbf{P}}^{(n)}_{(2)}$ and $\hat{\mathbf{Q}}^{(n)}_{(2)}$ to use as $\mathbf{T}^{(n)}$, one criteria would be to consider how well $\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})} = \langle \hat{\mathbf{P}}^{(1)}_{(2)} \ldots \hat{\mathbf{P}}^{(N)}_{(2)} \rangle$ and $\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})} = \langle \hat{\mathbf{Q}}^{(1)}_{(2)} \ldots \hat{\mathbf{Q}}^{(N)}_{(2)} \rangle$ fit $\boldsymbol{\mathcal{P}} \cap \boldsymbol{\mathcal{Q}}$:

$$IC_1(\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}) = 1 - \frac{\|(\boldsymbol{\mathcal{P}} \cap \boldsymbol{\mathcal{Q}}) - \hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}\|}{\|\boldsymbol{\mathcal{P}} \cap \boldsymbol{\mathcal{Q}}\|} \text{ and } IC_1(\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}) = 1 - \frac{\|(\boldsymbol{\mathcal{P}} \cap \boldsymbol{\mathcal{Q}}) - \hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}\|}{\|\boldsymbol{\mathcal{P}} \cap \boldsymbol{\mathcal{Q}}\|}.$$

One obvious difficulty with this fit-based intersection criterion, IC_1, is that the fit computations can be very costly. Alternatively, if we consider the two tensor decompositions, $\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}$ and $\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}$ as two groups of clusters, then we need to choose the group of clusters on which the membership of the shared elements (the overlapping part) is more tight and we can use the norms of the sub-factors to quantify how strongly elements belongs to the corresponding clusters. Intuitively, norms of the sub-factors corresponding to the overlapping region

$$IC_2(\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}) = \|\langle \hat{\mathbf{P}}^{(1)}_{(2)}, \ldots, \hat{\mathbf{P}}^{(N)}_{(2)} \rangle\|, \quad IC_2(\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}) = \|\langle \hat{\mathbf{Q}}^{(1)}_{(2)}, \ldots, \hat{\mathbf{Q}}^{(N)}_{(2)} \rangle\|,$$

explain the contribution of each element to these clusters and the one with the larger intersection criterion measure, IC_2, can be used to $\mathbf{T}^{(n)}$.

Note that the norm of the sub-factors of the overlapping region excludes any knowledge about how the groups fit with the rest of the tensors. Alternatively, we can account for the strengths of the groups in the whole tensor by also considering the core tensor

$$IC_3(\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}) = \|\langle \lambda_p, \hat{\mathbf{P}}^{(1)}_{(2)}, \ldots, \hat{\mathbf{P}}^{(N)}_{(2)} \rangle\|, \quad IC_3(\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}) = \|\langle \lambda_q, \hat{\mathbf{Q}}^{(1)}_{(2)}, \ldots, \hat{\mathbf{Q}}^{(N)}_{(2)} \rangle\|,$$

and select the tensor which leads to the larger intersection criterion, IC_3, measure. Here, λ_p and λ_q are core vectors of $\hat{\boldsymbol{\mathcal{P}}}^{(\bar{\mathbf{2}})}$ and $\hat{\boldsymbol{\mathcal{Q}}}^{(\bar{\mathbf{2}})}$, respectively.

Note that for IC_2 and IC_3, the columns of $\hat{\mathbf{P}}^{(n)}_{(2)}$ and $\hat{\mathbf{Q}}^{(n)}_{(2)}$ are normalized to length one with the weights absorbed into the vector λ_p and λ_q, respectively.

Union-based selection criteria.

The aforementioned intersection-based selection criteria have a potential weakness: as we see later in Section 8.3, the selection measures based on intersection fit and norm work well when the two input tensors are balanced in size. If the two tensors are unbalanced in size (i.e. one of the tensors is much larger than the other) the non-overlapping region of the larger tensor is likely to have a large impact on the final accuracy and the intersection-based selection criteria which primarily focus on the overlapping region of the tensors may fail to capture this. To address this limit of intersection-based selection criteria, we also consider *union-based* selection criteria that take into account both non-overlapping and overlapping parts of the tensors.

Firstly, we consider the fit of the union of the decomposed tensors to the union of the two original tensors

$$UC_1(\langle \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \rangle) = 1 - \frac{\|(\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}) - \langle \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \rangle\|}{\|\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}\|},$$

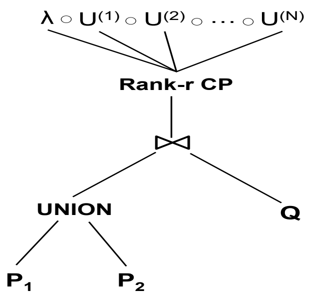
and we choose between the two alternatives by setting the initial $\mathbf{U}^{(n)}$ to $[\hat{\mathbf{P}}_{(1)}^{(n)T} \hat{\mathbf{P}}_{(2)}^{(n)T} \hat{\mathbf{Q}}_{(3)}^{(n)T}]^T$ and to $[\hat{\mathbf{P}}_{(1)}^{(n)T} \hat{\mathbf{Q}}_{(2)}^{(n)T} \hat{\mathbf{Q}}_{(3)}^{(n)T}]^T$ and observing which one leads to a better fit. UC_1 is the *initial* fit of the union of the decomposed tensors to the union of the two original tensors in the beginning of the update process of $\mathbf{U}_{(k_n)}^{(n)}$ for $k_n = 1, 2, 3$ (see Equation 6.7). Intuitively, this initial fit can be thought of as a rough indicator of whether the final fit of the union of the decomposed tensors solved by the learning process will be close to the decomposition on the union of two tensors or not.

As a second criterion, we consider the density of the input tensors, $\boldsymbol{\mathcal{P}}_{I_1 \times I_2 \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Q}}_{J_1 \times J_2 \times \cdots \times J_N}$,

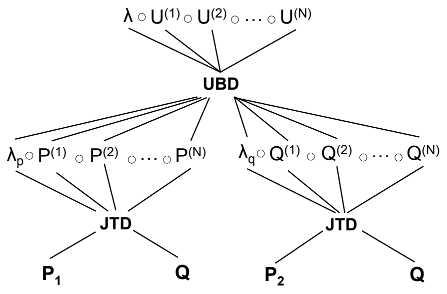
$$UC_2(\boldsymbol{\mathcal{P}}) = \frac{|\boldsymbol{\mathcal{P}}|}{\prod_{i=1}^{N} I_i}, \qquad UC_2(\boldsymbol{\mathcal{Q}}) = \frac{|\boldsymbol{\mathcal{Q}}|}{\prod_{i=1}^{N} J_i},$$

where $|\boldsymbol{\mathcal{X}}|$ is the number of non-zeros of $\boldsymbol{\mathcal{X}}$. Given this, we set the initial $\mathbf{U}^{(n)}$,

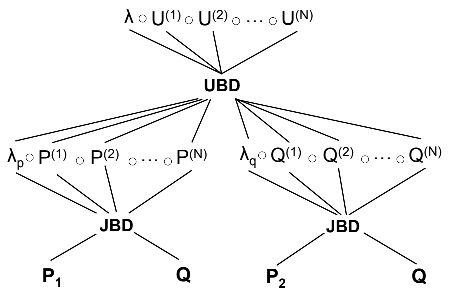
$$\mathbf{U}^{(n)} = [\hat{\mathbf{P}}_{(1)}^{(n)T} \hat{\mathbf{P}}_{(2)}^{(n)T} \hat{\mathbf{Q}}_{(3)}^{(n)T}]^T, \text{ if } \boldsymbol{\mathcal{P}} \text{ has a larger density, or}$$



(a) Union, join, then decompose


λ ∘ U(1) ∘ U(2) ∘ … ∘ U(N)
UBD
λp ∘ P(1) ∘ P(2) ∘ … ∘ P(N)
λq ∘ Q(1) ∘ Q(2) ∘ … ∘ Q(N)
JTD
JTD
P1
Q
P2
Q

(b) Join-then-decompose (JTD) and union-by-decomp. (UBD)


λ ∘ U(1) ∘ U(2) ∘ … ∘ U(N)
UBD
λp ∘ P(1) ∘ P(2) ∘ … ∘ P(N)
λq ∘ Q(1) ∘ Q(2) ∘ … ∘ Q(N)
JBD
JBD
P1
Q
P2
Q

(c) Join-by-decomposion (JBD) and union-by-decomp. (UBD)

Figure 6.4: Three alternative query plans for implementing a complex query plan with union, join, and decompose operations

$$\mathbf{U}^{(n)} = [\hat{\mathbf{P}}_{(1)}^{(n)T} \hat{\mathbf{Q}}_{(2)}^{(n)T} \hat{\mathbf{Q}}_{(3)}^{(n)T}]^T, \text{ if } \mathfrak{Q} \text{ has a larger density.}$$

Intuitively, the overlapping part will be more tightly connected with the non-overlapping part in the input tensor with the larger density – simply because there are less chances that an entry will be seen only in the overlapping part. Thus, given the choice between using the decompositions (for the overlapping part) of the input tensor with the larger density and of the tensor with the smaller density, the former is likely to lead to lesser errors.

6.3 Parallelization, Materialization, and Further Optimizations

The proposed union-by-decomposition (UBD) scheme leads to various optimization opportunities. First of all, assuming the availability of multiple computation

Table 6.1: Tensor data sets

Data set	Attributes	Size	Density (%)
3-mode MovieLens 1M	`(user, movie, rating)`	$6000 \times 3400 \times 5$	0.8451
3-mode book rating	`(user, book, rating)`	$105283 \times 340556 \times 11$	0.0003
4-mode Epinions	`(user, product, category, rating)`	$22111 \times 296000 \times 26 \times 5$	0.000007
4-mode MovieLens 1M	`(user, movie, genre, rating)`	$6000 \times 3400 \times 18 \times 5$	0.0994

units, the individual data sources can be decomposed in parallel. Moreover, each individual decomposition of the sub-tensors can also be obtained in parallel, leading to highly parallelizable execution plans. Secondly, as we see in Section 8.3, in situations where the same data source is integrated (unioned) with different data sources over time, we can decompose this data source once and materialize the decomposition for later reuse within a UBD process, thereby avoiding significant amount of runtime work.

In addition, the proposed union-by-decomposition (UBD) operator is compatible with other novel (decomposition push-down based) operators, including the *join-by-decomposition* (JBD) operator, discussed in Chapter 5, and can be used as part of a general optimization framework. Figure 6.4 provides an example: in Figure 6.4(b) first the join is pushed down over union and then the decomposition is pushed down over union, whereas in Figure 6.4(c) the decomposition is pushed down also over the join operator leading to (as we see in Section 8.3) a highly efficient query plan.

6.4 Experimental Evaluation

In this section, we present experimental results assessing the efficiency and effectiveness of the proposed union-by-decomposition (UBD) scheme and the selection criteria.

6.4.1 Experimental Setup

For these experiments, we used real data tensors (Table 6.1): (a) MovieLens 1M data set [56] with a 3-mode tensor `(user, movie, rating)` and (b) a 4-mode tensor `(user movie, genre, rating)`, (c) a book rating data set [79] with a 3-mode tensor `(user, book, rating)`, and (d) Epinions data set [72] with a 4-mode tensor `(user, product, category, rating)`. From each data tensor, we created pairs of sub-tensors (chosen randomly) with different degrees of intersection (10%, 20%, 40%, 60%). The target rank that we consider for the CP decomposition is 10. The default selection measure is the density-based selection measure, UC_2.

For evaluation, we consider both *execution time* and *degree of fit* defined as

$$\text{fit}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{P}} \hat{\cup} \boldsymbol{\mathcal{Q}}) = 1 - \frac{\|\boldsymbol{\mathcal{X}} - (\boldsymbol{\mathcal{P}} \hat{\cup} \boldsymbol{\mathcal{Q}})\|}{\|\boldsymbol{\mathcal{X}}\|}, \tag{6.11}$$

where $\boldsymbol{\mathcal{X}}$ is the union of $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$ and $\boldsymbol{\mathcal{P}} \hat{\cup} \boldsymbol{\mathcal{Q}}$ is the tensor obtained by re-composing the decomposition of $\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}$ in the considered scheme. Comparing the fit with respect to $\boldsymbol{\mathcal{X}}$ enables us not only to measure how well $\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}$ approximates the entries in $\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}$, but also whether $\boldsymbol{\mathcal{P}} \hat{\cup} \boldsymbol{\mathcal{Q}}$ includes any spurious entries that are not originally in $\boldsymbol{\mathcal{P}} \cup \boldsymbol{\mathcal{Q}}$.

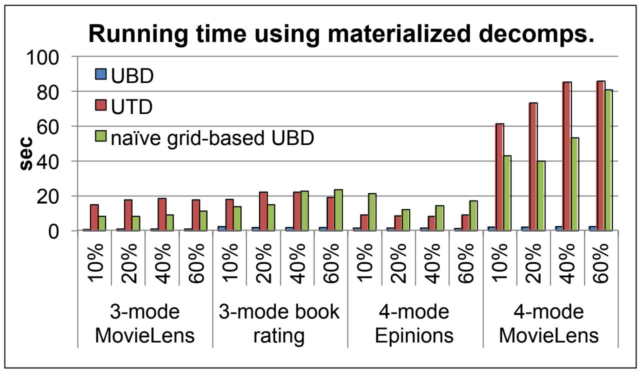
We ran all the experiments on a machine with Intel Core i5-2400 CPU @ 3.10GHz $\times 4$ with 7.7 GB RAM. We used MATLAB Version 7.13.0.564 (R2011b) 64-bit for the general implementation and MATLAB Parallel Computing Toolbox for the parallel implementations. We used the MATLAB Tensor Toolbox [12] to represent relational tensors as sparse tensors.

6.4.2 Results #1: UBD vs. UTD (with and without materialization)

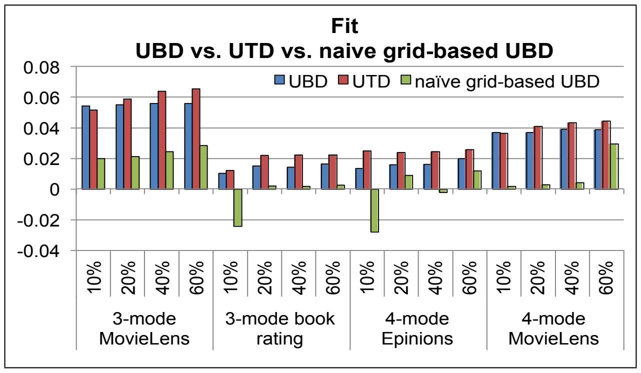
We first compare the <u>proposed UBD</u> against the more <u>conventional UTD</u> scheme. As a second competitor, we also consider the <u>naive grid-based UBD</u> discussed in Section 6.2.

Firstly, as we see in Figure 6.5(a), when there are opportunities for reusing existing materialized decompositions of the input tensors, as expected, UBD is much faster than the UTD as well as the naive grid-based UBD.

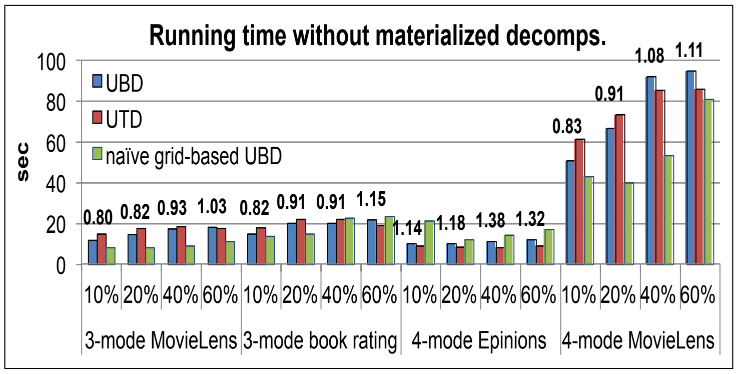
Secondly, in Figure 6.5(c), we consider the case where there are no opportunities



(a) Execution times (with materialization re-use)



(b) Accuracies



(c) Execution times (without materialization re-use)

Running time ratio (UBD/UTD) is shown on each bar

(the smaller the ratio, the better is the performance of UBD)

Figure 6.5: UBD vs. UTD vs. naive grid-based UBD on pairs of tensors with different intersection sizes (10%, 20%, 40%, 60%)

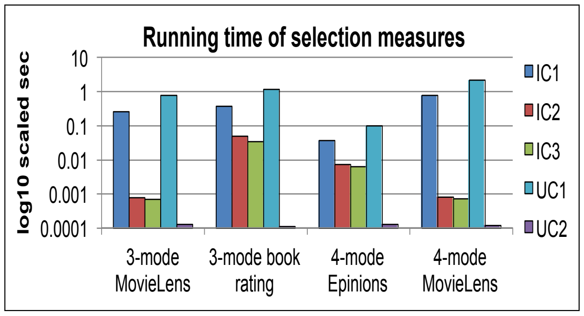
for reusing existing decompositions. As we see in this figure, as expected, when the input tensors have to be decomposed as part of the UBD process, whether UBD outperfoms UTD depends on the characteristics of the input tensors: in particular, as expected, UBD is faster than UTD when (a) *the degree of intersection is low* ($\leq 20\%$) and (b) *the input tensors are not extremely sparse*: if these conditions are not satisfied, the size of the union result is close to the sizes of input tensors and, if the result is also sparse, there is no gain in pushing down the decompositions.

Note that, when materialized decompositions of the input tensors do not exist, grid-based UBD can out-pace the proposed UBD and UTD in many configurations. However, as we see in Figure 6.5(b), this comes at the cost of a significant drop in accuracy: the proposed UBD scheme achieves fits close to the fit of UTD, whereas the accuracy of the grid-based UBD is much lower. Note also that the accuracy of UBD is especially good in data sets that are not *extremely sparse*.

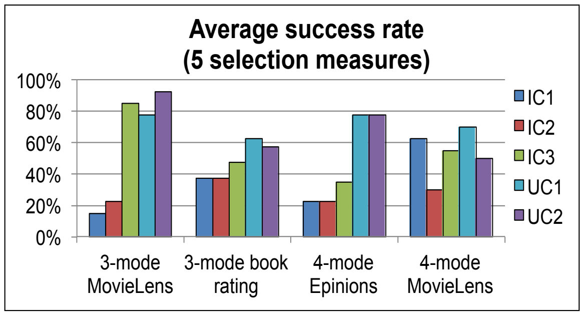
6.4.3 Results #2: Evaluation of the Alternative Selection Measures

In Section 6.2.2, we considered various approaches (IC_1, IC_2, IC_3, UC_1, and UC_2) for choosing the sub-factors for the overlapping parts of the input tensors. Figure 6.6(a) shows that fit-based measures (intersection fit, IC_1 and union fit, UC_1) are more expensive than norm-based measures (IC_2, IC_3). The *density*-based approach (UC_2) has an almost 0 execution cost. Note that, when we compare the computation times of these selection measures to the execution times of the UBD operators (Figure 6.5), we see that even the most expensive selection strategy is, in practice, affordable. Therefore, the major criterion for selecting among these measures should be accuracy.

For measuring the accuracy of different selection measures, we considered the percentage of the cases where each selection measure returned the best alternative. As shown in Figure 6.6(b), the union-based fit (UC_1) measure works best overall. The



(a) Computation times



(b) Average success rate

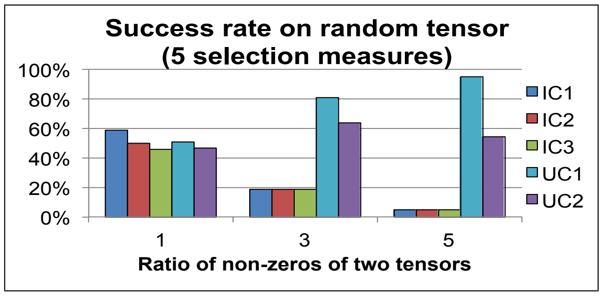
Figure 6.6: Efficiency and accuracy of the different selection measures in average of different intersection sizes (10%, 20%, 40%, and 60%)

Table 6.2: Average fit of the different selection measures (The highest fits for each data set are highlighted in bold)

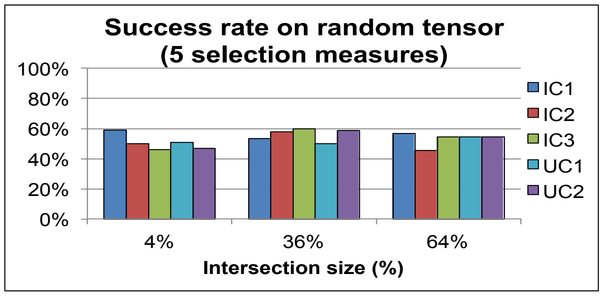
Data set	IC_1	IC_2	IC_3	UC_1	UC_2
3-mode MovieLens 1M	0.0538	0.0539	0.0551	0.0551	**0.0553**
3-mode book rating	0.0127	0.0127	0.0134	**0.0141**	0.0138
4-mode Epinions	0.0133	0.0133	0.0144	**0.0164**	**0.0164**
4-mode MovieLens 1M	**0.0380**	0.0376	0.0378	0.0377	**0.0380**

density measure (UC_2) also works well. The figure also shows that the intersection-based measures (IC_1, IC_2, IC_3) are not good indicators, even behave negatively in some cases: among them the IC_3 works the best since it also accounts for the non-overlapping regions through the cluster strength indicated by the core. Table 6.2 further studies the average degree of fits returned by the different strategies. The table confirms that the average fits obtained by the union-based selection measures are overall better than the intersection-based selection measures. While the numbers vary, the degrees of fit based on the union-based selection measures are up to 20% better than IC_1 and IC_2.

To further study the impacts of various parameters on the selection accuracy, we also created random tensors with different configurations, varying the balance (ratio of densities) of the input tensors and intersection sizes. For each experiment, we created 10 different random tensors of size $5000 \times 5000 \times 10$ and measured the



(a) Impact of balance



(b) Impact of the overlap size

Figure 6.7: Success rate in predicting the best fit of UBD using the 5 selection measures compared among different (a) ratios of non-zeros of two tensors and (b) intersection sizes

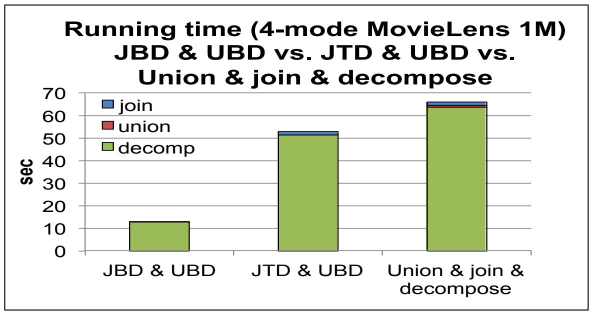
percentage cases in which each measure selected the better fitting tensor. As the default configuration, we set the ratio of non-zeros to 1 (most balanced), intersection size to 4%, and the density of the union tensor to 0.01%.

In Figure 6.7(a), we first study the impact of balance. Here, the configuration with $ratio = 1$ corresponds to the most balanced configuration. As we expected, when the tensors are balanced, all measures work similarly (with a slight edge to the intersection-based measures); however, as the imbalance among tensors increases, intersection-based measures get worse, while the union based measures, especially UC_1, improve.

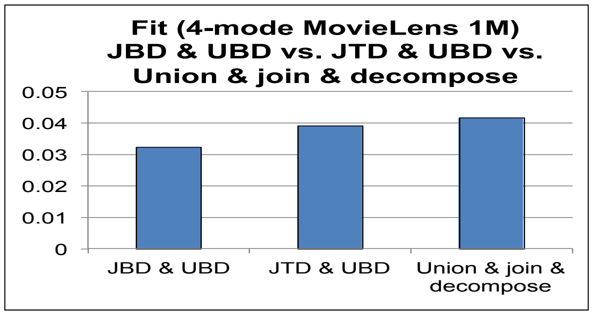
Unlike balance, the size of the intersection has no significant impact on the selection accuracy (Figure 6.7(b)), indicating that all measures are robust in this respect.

6.4.4 Results #3: Impact of Composition of UBD with other Operators

As we discussed in Section 6.3, the proposed union-by-decomposition (UBD) operator is compatible with other operators and can be used as part of a general optimization framework. In Figure 6.8 for a sample data, we study the alternative query plans considered in Figure 6.4. As expected, the figure shows that pushing decompositions down the join and union operations (i.e., using UBD, proposed in this chapter, and/or JBD, proposed in Chapter 5 provides a much faster execution times than the



(a)



(b)

Figure 6.8: (a) Running times and (b) fits of three alternative query plans "JBD and UBD" vs. "JTD and UBD" vs. "union, join, and decompose" (see Figure 6.4) on 4-mode MovieLens 1M

union operation and join operation followed by a final CP decomposition step. As shown in Figure 6.8(a), among these three alternative query plans, the query plan using JBD and UBD is the fastest (faster than 5× of the union, join, and decompose strategy) but comes with $\sim$ 20% drop in accuracy (Figure 6.8(b)). On the other hand, using UBD proposed in this chapter along with the conventional join-then-decompose (JTD) strategy instead of JBD reduces the execution time relative to "union, join, and decompose" by $\sim$ 20% (Figure 6.8(a)), with a negligible impact on accuracy (Figure 6.8(b)).

Chapter 7

DECOMPOSITION-BY-NORMALIZATION (DBN): LEVERAGING APPROXIMATE FUNCTIONAL DEPENDENCIES FOR EFFICIENT CP AND TUCKER DECOMPOSITIONS

For many multi-dimensional data applications, tensor operations as well as relational operations both need to be supported throughout the data lifecycle. Tensor based representations (including two widely used tensor decompositions, CP and Tucker decompositions) are proven to be effective in multi-aspect data analysis and tensor decomposition is an important tool for capturing high-order structures in multi-dimensional data. Although tensor decomposition is shown to be effective for multi-dimensional data analysis, the cost of tensor decomposition is often very high. Since the number of modes of the tensor data is one of the main factors contributing to the costs of the tensor operations, we focus on reducing the modality of the input tensors to tackle the computational cost of the tensor decomposition process. We propose a novel `decomposition-by-normalization` (DBN) scheme that first normalizes the given relation into smaller tensors based on the functional dependencies of the relation, decomposes these smaller tensors, and then recombines the sub-results to obtain the overall decomposition. The decomposition and recombination steps of the `decomposition-by-normalization` scheme fit naturally in settings with multiple cores. This leads to a highly efficient, effective, and parallelized `decomposition-by-normalization` algorithm for both dense and sparse tensors for CP and Tucker decompositions. Experimental results confirm the efficiency and effectiveness of the proposed `decomposition-by-normalization` scheme compared to the conventional nonnegative CP decomposition and Tucker decomposition approaches.

7.1 Introduction

Tensor based representations, including two widely used decompositions, CP [19, 34] and Tucker [74] decompositions, are proven to be effective in multi-aspect data analysis. Consequently, tensor decomposition is an important tool for capturing high-order structures in multi-dimensional data [10, 45, 46, 47, 59, 73, 78].

Unfortunately, tensor decomposition operation can be prohibitively costly when the tensor data have a large number of modes:

- One obvious problem is the space needed to hold the input tensors. When the tensor is *dense* (i.e., has a large number of nonzero entries) or when a dense tensor representation is used for algorithmic reasons, the space required to hold the data increases exponentially with the number of modes.
- The Tucker decomposition may be infeasible for large data sets (even if the original tensor is sparse) since the tensors needed to represent intermediate results are often dense.

Recent attempts to overcome these problems using parellel tensor decomposition [10, 59, 78] techniques also face difficulties, including synchronization and data exchange overheads.

7.1.1 Contributions

Our goal is to tackle the high computational cost of the tensor decomposition process. Since, as described above, the number of modes of the tensor data is one of the main factors contributing to the cost of the tensor operations, we argue that if

- a tensor with large number of modes can be normalized (i.e., vertically partitioned) into tensors with smaller number of modes and

- each sub-tensor is decomposed independently,

then the resulting partial decompositions can be efficiently combined to obtain the decomposition of the original tensor. We refer to this as the `decomposition-by-normalization` (DBN) scheme.

Example 7.1.1 *Consider the 5-attribute relation,* $\mathcal{R}$`(workclass, education, ID, occupation, income)` *in Figure 7.1(a) and assume that we want to decompose the corresponding tensor for multi-dimensional analysis.*

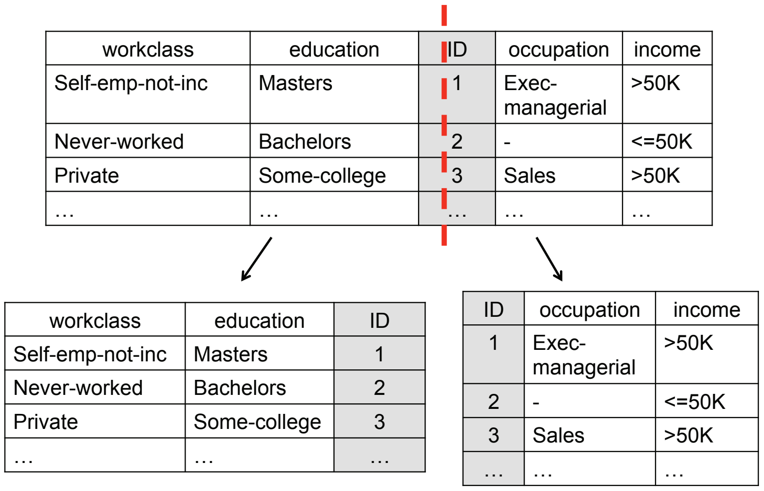
Figure 7.1(a) illustrates an example normalization which divides this 5-attribute relation into two smaller relations with 3 attributes, $\mathcal{R}_1$`(workclass, education, ID)` *and* $\mathcal{R}_2$`(ID, occupation, income)`*, respectively.*

Figures 7.1(b) and (c), then, illustrate the proposed DBN scheme for CP and Tucker decompositions, respectively: In both cases, once the two partitions are decomposed, we combine the resulting core tensors and factor matrices to obtain the decomposition of the original tensor corresponding to the relation $\mathcal{R}$*.* ○

Benefits of DBN for CP Decompositions: In the CP decomposition example above (Figure 7.1(b)),

- if the input relation $\mathcal{R}$ is *dense*, we argue that decompositions of partitions $\mathcal{R}_1$ and $\mathcal{R}_2$ will be much faster than that of the original relation $\mathcal{R}$ and the gain will more than compensate for the normalization and recombination costs of DBN.

- If the input relation $\mathcal{R}$ is *sparse*, on the other hand, the decomposition cost is not only determined by the number of modes, but also the number of nonzero entries in the tensor. Consequently, unless the partitioning provides smaller numbers of tuples in both partitions, we cannot theoretically expect DBN to provide large gains. However, as we experimentally verify in Section 7.6, DBN

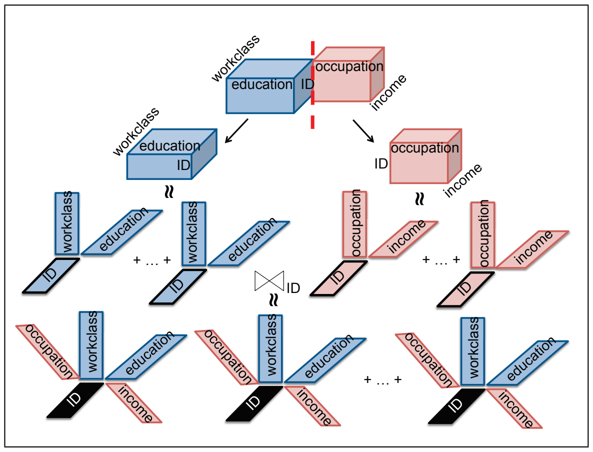


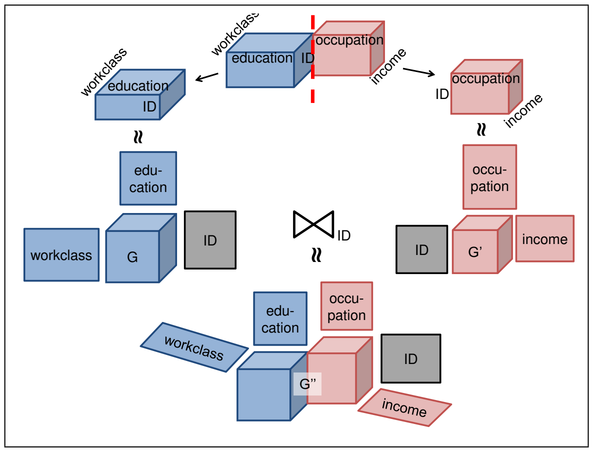
workclass	education	ID	occupation	income
Self-emp-not-inc	Masters	1	Exec-managerial	>50K
Never-worked	Bachelors	2	-	<=50K
Private	Some-college	3	Sales	>50K
…	…	…	…	…

workclass	education	ID
Self-emp-not-inc	Masters	1
Never-worked	Bachelors	2
Private	Some-college	3
…	…	…

ID	occupation	income
1	Exec-managerial	>50K
2	-	<=50K
3	Sales	>50K
…	…	…

(a) Normalization (vertical data partitioning)





(b) DBN process for CP decomposition (c) DBN process for Tucker decomposition

Figure 7.1: (a) Normalization of a relation $\mathcal{R}$(`workclass, education, ID, occupation, income`) into two relations $\mathcal{R}_1$(`workclass, education, ID`) and $\mathcal{R}_2$(`ID, occupation, income`) based on the key (`ID`); `decomposition-by-normalization` (DBN): normalization of $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$, (b) rank-r_1 CP decomposition of $\mathcal{R}_1$ and rank-r_2 CP decomposition of $\mathcal{R}_2$ that are combined on the `ID` mode into rank-$(r_1 \times r_2)$ CP decomposition of $\mathcal{R}$, and (c) rank-$(..., r_1, ...)$ Tucker decomposition of $\mathcal{R}_1$ and rank-$(..., r_2, ...)$ Tucker decomposition of $\mathcal{R}_2$ that are combined on the `ID` mode into rank-$(..., r_1 \times r_2, ...)$ Tucker decomposition of $\mathcal{R}$

scheme fits naturally in multi-core implementations, thus in practice provides significant advantages even for sparse input tensors.

Benefits of DBN for Tucker Decompositions: Since the scale of the intermediate blowup problem [45] depends largely on the modality of the input tensor, we argue that dividing the tensor into sub-tensors with smaller number of modes will help eliminate this notorious bottleneck. Moreover, similarly to the case in CP decompositions, each individual sub-tensor decomposition can run on an available core without having to communicate with other sub-tensor decompositions running on different cores, leading to effective parallelizations of Tucker decompositions.

Challenges and Contributions: Note that in general, a given tensor can be partitioned into two in multiple ways. The key challenges we address in this chapter are (a) *how best to partition* a given tensor into smaller tensors and (b) *how to recombine* the sub-result to obtain the decomposition of the original tensor. In particular, achieving the projected advantages of the DBN strategy requires us to address the following key challenges:

- *Challenge 1.* First of all, we need to ensure that the join attribute is selected in such a way that the normalization (i.e., the vertical partitioning) process does not lead to spurious tuples. Secondly, the join attribute needs to partition the data in such a way that the later steps in which decompositions of the individual partitions are combined into an overall decomposition do not introduce errors. One way to prevent the normalization process from introducing spurious data is to select an attribute which *functionally determines* the attributes that will be moved to the second partition. *This requires an efficient method to determine functional dependencies in the data.*

- *Challenge 2.* A second difficulty is that many data sets may not have perfect functional dependencies to leverage for normalization. *In that case, we need to be able to identify and rely on approximate functional dependencies in the data.*

- *Challenge 3.* Once the approximate functional dependencies are identified, *we need a mechanism to partition the data into two partitions in such a way that will lead to least amount of errors during later stages.* We argue that partitioning the attributes in a way that minimizes *inter-partition* functional dependencies and maximizes *intra-partition* dependencies will lead to least amount of errors in the recombination step.

- *Challenge 4.* Moreover, after data is vertically partitioned and individual partitions are decomposed, the individual decompositions need to be *recombined to obtain the decomposition* of the original relation. This process needs to be done in a way that is efficient and parallelizable.

The chapter is organized as follows: We provide an overview of the proposed DBN scheme in Section 7.2. We then focus on selecting the best partitions for the normalization step of DBN (Section 7.3). In Section 7.4, we present rank-pruning strategies to further reduce the cost of DBN. We experimentally evaluate DBN in Section 7.6 in both stand-alone and parallel configurations. We focus on the accuracy and the running time of the alternative algorithms. Experimental results provide evidence that in addition to being significantly faster than conventional decompositions, DBN can approximate well the accuracy of the conventional tensor decomposition techniques.

7.2 Decomposition-By-Normalization (DBN)

Our goal is to tackle the high computational cost of decomposition process through what we refer to as the `decomposition-by-normalization` (DBN). In this section, we first introduce the relevant notations, provide background on key concepts, and then present an overview of the DBN process.

7.2.1 Key Concepts

Without loss of generality, we assume that relations are represented in the form of *occurrence tensors* (Section 3.3.1).

Functional Dependencies

A functional dependency (FD) between two sets of attributes, $\mathbb{X}$ and $\mathbb{Y}$, is defined as follows [29].

Definition 7.2.1 (Functional Dependency) *A functional dependency (FD), denoted by $\mathbb{X} \rightarrow \mathbb{Y}$, holds for relation instance $\mathcal{R}$, if and only if for any two tuples t_1 and t_2 in $\mathcal{R}$ that have $t_1[\mathbb{X}] = t_2[\mathbb{X}]$, $t_1[\mathbb{Y}] = t_2[\mathbb{Y}]$ also holds.*

We refer to a functional dependency as a pairwise functional dependency *if the sets $\mathbb{X}$ and $\mathbb{Y}$ are both singleton.* ○

Intuitively, a functional dependency is a constraint between two sets of attributes $\mathbb{X}$ and $\mathbb{Y}$ in a relation denoted by $\mathbb{X} \rightarrow \mathbb{Y}$, which specifies that the values of the $\mathbb{X}$ component of a tuple uniquely determine the values of the $\mathbb{Y}$ component. Note that if $\mathbb{A} = \{A_1, \ldots, A_n\}$ is a set of attributes in the schema of a relation, R, and $\mathbb{X}, \mathbb{Y} \subseteq \mathbb{A}$ are two subsets of attributes such that $\mathbb{X} \rightarrow \mathbb{Y}$, then the relation instance $\mathcal{R}$ can be vertically partitioned into two relation instances $\mathcal{R}_1$, with attributes $\mathbb{A} \setminus \mathbb{Y}$, and $\mathcal{R}_2$, with attributes $\mathbb{X} \cup \mathbb{Y}$, such that $\mathcal{R} = \mathcal{R}_1 \bowtie \mathcal{R}_2$; in other words the set of attributes $\mathbb{X}$ serves as a foreign key and joining vertical partitions $\mathcal{R}_1$ and $\mathcal{R}_2$ on $\mathbb{X}$ gives back the relation instance $\mathcal{R}$ without any missing or spurious tuples.

Note that, discovery of FDs in a given data set is a challenging problem since the complexity increases exponentially in the number of attributes [54]. Moreover, in many data sets, attributes may not have perfect FDs due to exceptions and outliers in the data. In such cases, we may only be able to locate approximate FDs [36] instead of exact FDs:

Definition 7.2.2 (Approximate Functional Dependency) *An approximate functional dependency (aFD), denoted by $\mathbb{X} \xrightarrow{\sigma} \mathbb{Y}$ holds for relation instance $\mathcal{R}$, if and only if*

- *there is a subset $\mathcal{R}' \subseteq \mathcal{R}$, such that $|\mathcal{R}'| = \sigma \times |\mathcal{R}|$ and, for any two tuples t_1 and t_2 in $\mathcal{R}'$ that have $t_1[\mathbb{X}] = t_2[\mathbb{X}]$, $t_1[\mathbb{Y}] = t_2[\mathbb{Y}]$ also holds; and*
- *there is no subset $\mathcal{R}'' \subseteq \mathcal{R}$, such that $|\mathcal{R}''| > \sigma \times |\mathcal{R}|$ where the condition holds.*

We refer to the value of σ as the support of the aFD, $\mathbb{X} \xrightarrow{\sigma} \mathbb{Y}$. ○

Many algorithms for FD and approximate FD discovery exist, including TANE [36], Dep-Miner [50], FastFD [77], and CORDS [37].

7.2.2 Overview of the Decomposition-by-Normalization (DBN) Process

The overall structure of the `decomposition-by-normalization` (DBN) process, visualized in Figure 7.1, is similar for both CP and Tucker decompositions. In this subsection, we present and discuss the pseudo code of DBN. In the following sections, we will study the key steps of the process in greater detail.

The pseudo code of DBN algorithm is presented in Figure 7.2. In its first step (Line 1), DBN evaluates the pairwise (approximate) FDs among the attributes of the input relation. For this purpose, we employ and extend TANE [36], an efficient algorithm for discovering FDs. Our modification of the TANE algorithm returns a set of (approximate) FDs between attribute pairs and, for each candidate dependency, $A_i \rightarrow A_j$, it provides a corresponding support value, $\sigma_{i,j}$.

```
DBN algorithm (input: a relation $\mathcal{R}$, a decomposition algorithm (CP or Tucker))
1: Identify paFD (pairwise approximate FDs between the pairs of attributes of $\mathcal{R}$)
2: Select the attribute $A_k$ with the highest $\sum_{k \neq j} \sigma_{k,j}$ such that $\sigma_{k,j} \geq \tau_{support}$ as the vertical partitioning (and join) attribute $X$ (Desiderata 1 and 2)
3: if $\mathcal{R}$ is a sparse tensor then
4:     if $X$ (approximately) determines all attributes of $\mathcal{R}$ then
5:         findInterFDPartition(paFD,false) (see Figure 7.3)
6:     else
7:         Move $X$ and all attributes determined by $X$ to $\mathcal{R}_1$; move $X$ and remaining attributes to $\mathcal{R}_2$ (Desideratum 4 and 5).
8:     end if
9: else {i.e., $\mathcal{R}$ is a dense tensor}
10:    if $X$ (approximately) determines all attributes of $\mathcal{R}$ then
11:        findInterFDPartition(paFD,true)
12:    else
13:        Move $X$ and attributes determined by $X$ to $\mathcal{R}_1$; move $X$ and remaining attributes to $\mathcal{R}_2$ – these moves are constrained such that the number of attributes of $\mathcal{R}_1$ and $\mathcal{R}_2$ are similar (Desideratum 3 and 5)
14:    end if
15: end if
16: Partition $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$.
17: If the selected $X$ does not perfectly determine the attributes of $\mathcal{R}_1$ then remove sufficient number of outlier tuples from $\mathcal{R}$ to enforce the FDs between $X$ and the attributes of $\mathcal{R}_1$
18: Create occurrence tensors of $\mathcal{R}_1$ and $\mathcal{R}_2$
19: Run JBD-CP (Figure ??) or JBD-Tucker (Figure ??) algorithm according to the input decomposition algorithm with the tensors corresponding to $\mathcal{R}_1$ and $\mathcal{R}_2$
```

Figure 7.2: Pseudo-code of DBN

The next steps of the algorithm involve selecting the attribute, X, that will serve as the foreign key (Line 2) and partitioning the input relation $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$ around X (Lines 3 through 16). If the selected join attribute X does not perfectly determine the attributes of $\mathcal{R}_1$, then to prevent introduction of spurious tuples, we need to remove (outlier) tuples from $\mathcal{R}$ to restore the discovered FDs between the attribute, X, and the attributes that are selected to be moved to partition $\mathcal{R}_1$ (Line 17). Note that a major part of the DBN algorithm involves deciding how to partition

```
findInterFDPartition ( input: paFD, balanced)
1: Create a complete pairwise approximate FD graph with weighted nodes, $G$, where each node is an attribute with
   the weight, which is the size of the corresponding attribute and edge weights are the support values of paFD.
2: if balanced == false then
3:     Run minimum average cut on $G$ to find a maximally independent partitioning (Desideratum 5)
4: else {i.e., balanced == true}
5:     Run balanced cut on $G$ to find a balanced cut first and in case that there are alternative balanced cuts,
       maximally independent partitioning (Desideratum 3 and 5)
6: end if
```

Figure 7.3: Pseudo-code of interFD-based partition algorithm; this is detailed in Section 7.3.2

the input data into two in the most effective manner. In Section 7.3, we will discuss the partitioning process in detail.

Finally, once $\mathcal{R}_1$ and $\mathcal{R}_2$ are obtained, we create the occurrence tensors for the two partitions (Line 18) and execute the JBD-CP and JBD-Tucker modules which we proposed in Chapter 5.

7.3 Vertical Data Partitioning

As discussed in Section 7.2.2, a significant challenge that DBN has to address is to partition the input data into two in such a way that they can be recombined effectively through the JBD process introduced in the previous section. In this section, we discuss vertical partitioning strategies for CP and Tucker decompositions. Below we first list the key desiderata that govern how the DBN algorithm makes the partitioning decision.

- **Desideratum 1:** As we discussed above, when we need to use approximate FDs when partitioning the input data, this may result in the removal of outlier tuples to preserve the semantics of the FDs. Therefore, to prevent over-thinning of the relation $\mathcal{R}$, the considered approximate FDs need to have few outliers and high support; i.e., $\sigma_{i,j} \geq \tau_{support}$, for a sufficiently large support lower-bound, $\tau_{support}$.

Secondly, when we vertically partition the relation $\mathcal{R}$ with attributes $\mathbb{A} = \{A_1, \ldots, A_n\}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$, one of the attributes (X) of $\mathcal{R}_2$ should serve as a foreign key into $\mathcal{R}_1$ to ensure that joining of the vertical partitions $\mathcal{R}_1$ and $\mathcal{R}_2$ (on X) gives back $\mathcal{R}$ without missing any tuples or introducing any spurious ones.

- **Desideratum 2:** If $\mathbb{A}_1$ is the set of attributes of vertical partition $\mathcal{R}_1$ and $\mathbb{A}_2$ is the set of attributes of vertical partition $\mathcal{R}_2$, then there must be an attribute $X \in \mathbb{A}_2$, such that for each attribute $Y \in \mathbb{A}_1$, $X \xrightarrow{\sigma} Y$, for $\sigma \geq \tau_{support}$.

Since the overall size (in terms of modes and their dimensionalities) of the input tensor is a major cost factor for dense (for CP and Tucker decompositions) or Tucker decomposing sparse tensors, we prefer that the partitions are balanced in terms of their dimensionalities.

- **Desideratum 3:** For dense (CP and Tucker decompositions) and sparse (Tucker decomposition), vertical partitioning should be such that the sizes of $\mathcal{R}_1$ and $\mathcal{R}_2$ are similar.

When CP decomposing sparse tensors, the major contributor to the decomposition cost is the number of nonzero entries in the tensor.

- **Desideratum 4:** For CP decomposition of sparse tensors, vertical partitioning should be such that the total number of tuples of $\mathcal{R}_1$ and $\mathcal{R}_2$ are minimized.

Any information encoded by the FDs crossing the two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ is potentially lost when $\mathcal{R}_1$ and $\mathcal{R}_2$ are individually decomposed. This leads to our final desideratum:

- **Desideratum 5:** The vertical partitioning should be such that the support for the inter-partition FDs (except for the FDs involving the join attribute X) are minimized.

7.3.1 Overview of the Partitioning Strategies

We use different strategies to satisfy the above desiderata depending on whether we work on sparse or dense tensors and whether we seek CP or tucker decompositions:

- **Case 1: CP Decomposition on Sparse Tensors.** This case has two subcases:
  - *Case 1.1: Exact Functional Dependencies:* When the join attribute X determines all attributes of $\mathcal{R}$, we apply the interFD-based vertical partitioning strategy detailed in Section 7.3.2.
  - *Case 1.2: Approximate Functional Dependencies:* When the join attribute X approximately determines a subset of the attributes of $\mathcal{R}$, we create a partition $\mathcal{R}_1$ with all the attributes determined with a support higher than the threshold ($\tau_{support}$) by the join attribute. This helps us satisfy Desiderata 1 and 2. The second partition, $\mathcal{R}_2$, consists of the join attribute X and all the remaining attributes. Note that, since we can include any attribute in $\mathcal{R}_1$ as long as it is determined by X, there may be still multiple ways to partition the data. Therefore, we apply the interFD-based partitioning strategy discussed in Section 7.3.2 to choose the two partitions. Note also that, the size of $\mathcal{R}_2$ is, by construction, equal to the number of tuples in $\mathcal{R}$ independent of which attributes are included in it. On the other hand, the size of $\mathcal{R}_1$ can be reduced down to the number of unique values of X by eliminating duplicate tuples (to satisfy Desideratum 4).
- **Case 2: CP Decomposition on Dense Tensors or Tucker Decomposition.** When we are operating on dense tensors or when we seek Tucker decompositions of sparse or dense tensors, we consider Desideratum 3, which prefers

balanced partitions as discussed in Section 7.3.2. When there are alternative balanced partitioning cases, we apply the interFD-based vertical partitioning strategy to break ties, which is also discussed in Section 7.3.2.

7.3.2 InterFD Criterion and Vertical Partitioning Algorithms

Minimizing the Likelihood of Decomposition Errors

As discussed in Section 5.3, given a partitioning, $\mathcal{R} = \mathcal{R}_1 \bowtie_A \mathcal{R}_2$, the accuracy of the decomposition is likely to be high if the non-join attributes of the two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ are independent from each other. Building on this observation (which we also validate in Section 7.6), DBN tries to partition the input relational tensor $\mathcal{R}$ in such a way that the resulting partitions, $\mathcal{R}_1$ and $\mathcal{R}_2$, are as independent from each other as possible. We refer to this as the InterFD criterion.

Remember that the support of an approximate FD is defined as the percentage of tuples in the data set for which the FD holds. Thus, in order to quantify the dependence of pairwise attributes, we rely on the supports of pairwise FDs. Since we have two possible FDs ($X \rightarrow Y$ and $Y \rightarrow X$) for each pair of attributes, we use the average of the two as the overall support of the pair of attributes X and Y. Given these pairwise supports, we approximate the overall dependency between two partitions $\mathcal{R}_1$ and $\mathcal{R}_2$ using the average support of the pairwise FDs (excluding the pairwise FDs involving the join attribute) crossing the two partitions.

Let the pairwise FD graph, $G_{pfd}(V, E)$, be a complete, weighted, and undirected graph, where:

- each vertex $v \in V$ represents an attribute (mode),
- the size of the domain (dimensionality) of the mode corresponding to vertex, v, is represented as a weight of the vertex, w_v, and

- the weight, w_e, of the edge e between nodes v_i and v_j is the average support of the approximate FDs $v_i \rightarrow v_j$ and $v_j \rightarrow v_i$.

We argue that the interFD-based vertical data partitioning problem can be formulated in terms of locating a cut on G_{pfd} with the minimum average weight. To solve the problem efficiently, we extend the minimum *total* weighted cut algorithm presented in [68] to identify the minimum *average* weight. The overall process is similar to that presented in [68] and has the same time complexity of complexity, $O(|V||E| + |V|^2 log|V|)$:

Given an undirected graph $G_{pfd}(V, E)$, the algorithm copies V into V', where each edge $e \in E$ is annotated with a counter n_e initially set to 1. The algorithm then first picks a vertex v with the cut with the minimum average weight. We compute the average edge weight of a cut between a set of vertices S and $V \backslash S$, denoted by $\bar{w}_S$, such that

$$\bar{w}_S = \sum w_e / \sum n_e, \text{ for } e \in \{(v_1, v_2) \in E | v_1 \in S, v_2 \in V \backslash S\}. \tag{7.1}$$

Then, the algorithm selects a neighbor v' of v such that $\{v,v'\}$ has a cut from $V' \backslash \{v, v'\}$ with the smallest average weight. The algorithm shrinks V' by merging v and v' into a new vertex, v''. Any pair of edges $e = a \rightarrow v$ and $e' = a \rightarrow v'$ originating from the same vertex a is replaced by a new edge $e'' = a \rightarrow v''$, where $w''_e = w_e + w'_e$ and $n''_e = n_e + n'_e$. Any other edge to v or v' is simply re-routed to v''. The process is stopped when $|V'| = 1$. The minimum of the minimum average cuts at each step of the algorithm is returned as the minimum average cut. The following example shows how the minimum average cut algorithm runs on a graph step by step.

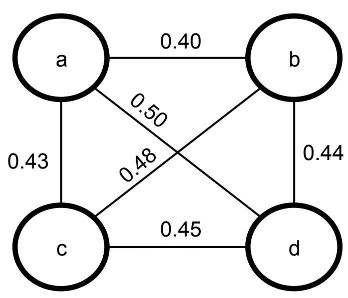
Example 7.3.1 *Consider the graph $G_{pfd}(V, E)$ in Figure 7.4(a). Initially, as shown in Figure 7.4(a), the weight of each edge is assigned with the average support of pairwise approx. FDs.*

- *Step 1: Among the vertices* a*,* b*,* c*, and* d *(*$\bar{w}_{\{a\}} = (0.40{+}0.50{+}0.43)/3 = 0.4433$*,* $\bar{w}_{\{b\}} = (0.40 + 0.48 + 0.44)/3 = 0.44$*,* $\bar{w}_{\{c\}} = (0.43 + 0.48 + 0.45)/3 = 0.4533$*, and* $\bar{w}_{\{d\}} = (0.44 + 0.50 + 0.45)/3 = 0.4633$*), the minimum average cut is the cut between* $\{b\}$ *and* $\{a,c,d\}$ *with the average weight (*$\bar{w}_{\{b\}} = 0.44$*) (see Figure 7.4(b)).*

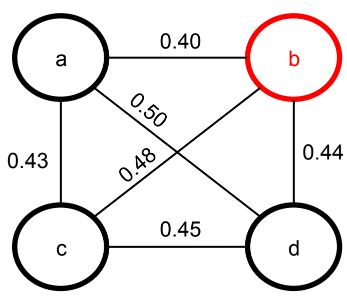
- *Step 2: Vertex* b *is merged with vertex* c *into* $\{b,c\}$ *with the smallest average weight among vertices* a *(*$\bar{w}_{\{a,b\}} = (0.43 + 0.48 + 0.50 + 0.44)/4 = 0.4625$*),* c *(*$\bar{w}_{\{b,c\}} = (0.43 + 0.40 + 0.45 + 0.44)/4 = 0.43$*), and* d *(*$\bar{w}_{\{b,d\}} = (0.45 + 0.48 + 0.50 + 0.40)/4 = 0.4575$*). Edges* $a \rightarrow b$ *and* $a \rightarrow c$ *are replaced by the edge* $a \rightarrow \{b,c\}$ *with weight* $w_{a\rightarrow\{b,c\}} = 0.40 + 0.43 = 0.83$ *and counter* $n_{a\rightarrow\{b,c\}} = 2$*. Edges* $d \rightarrow b$ *and* $d \rightarrow c$ *are replaced by the edge* $d \rightarrow \{b,c\}$ *with weight* $w_{d\rightarrow\{b,c\}} = 0.44 + 0.45 = 0.89$ *and counter* $n_{d\rightarrow\{b,c\}} = 2$*. The minimum average cut is the cut between* $\{b,c\}$ *and* $\{a,d\}$ *with the average weight (*$\bar{w}_{\{b,c\}} = 0.43$*) (see Figure 7.4(c)).*

- *Step 3:* $\{b,c\}$ *is merged with vertex* d *into* $\{b,c,d\}$ *with the smallest average weight among vertices* a *(*$\bar{w}_{\{a,b,c\}} = (0.89 + 0.50)/3 = 0.4633$*) and* d *(*$\bar{w}_{\{b,c,d\}} = (0.83 + 0.50)/3 = 0.4433$*). Edges* $a \rightarrow \{b,c\}$ *and* $a \rightarrow d$ *are replaced by the edge* $a \rightarrow \{b,c,d\}$ *with weight* $w_{a\rightarrow\{b,c,d\}} = 0.83 + 0.50 = 1.33$ *and counter* $n_{a\rightarrow\{b,c,d\}} = 3$*. The cut between* $\{a\}$ *and* $\{b,c,d\}$ *is the last cut with weight* $\bar{w}_{\{b,c,d\}} = 0.4433$ *(see Figure 7.4(d)).*

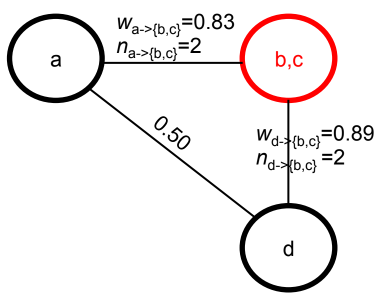
- *Step 4: The process ends since* $|V'| = 1$ *(see Figure 7.4(e)). The minimum of the minimum average cuts at each step is* $\{b,c\}$ *and* $\{a,d\}$ *in Step 2.*



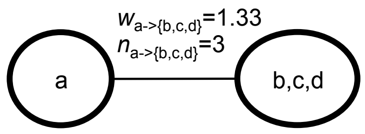
(a) $G_{pfd}(V, E)$

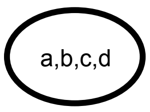


(b) Step 1


a
b,c
d
$w_{a\text{->}\{b,c\}}$=0.83
$n_{a\text{->}\{b,c\}}$=2
0.50
$w_{d\text{->}\{b,c\}}$=0.89
$n_{d\text{->}\{b,c\}}$=2

(c) Step 2





(d) Step 3 (e) Step 4

Figure 7.4: An example of the minimum average cut algorithm for $G_{pfd}(V, E)$ (see Example 7.3.1)

Balanced Partitioning

When targeting Desideratum 3, we seek a balanced partitioning of the attributes. Unfortunately, the general problem of obtaining balanced partitions is an NP-complete problem even for simple sets of values [33]. While there are various approximation and heuristic algorithms including [40], applying these directly would only optimize balance, but ignore other criteria. We therefore choose the average cut based partitioning scheme discussed above in a way that also considers balance of attributes. In particular, we associate a *balance score* to each vertex, v:

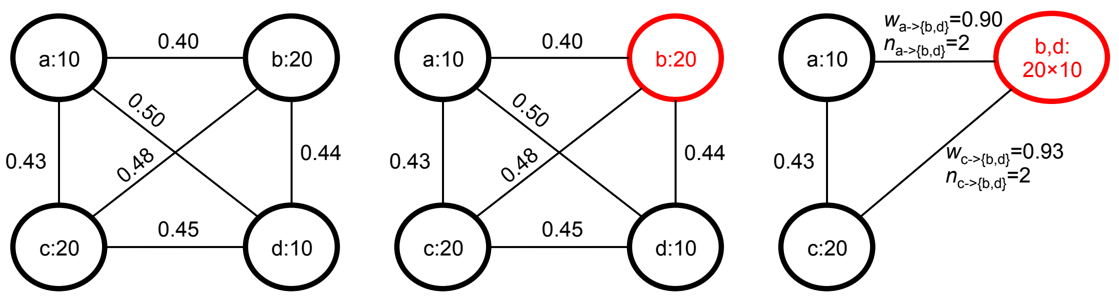
$$balance\_score(v) = \frac{max\{size(V_v), size(V \backslash V_v)\}}{min\{size(V_v), size(V \backslash V_v)\}}, \tag{7.2}$$

where V_v is the set of original vertices merged into v (if v is a original vertex, then V_v is $\{v\}$) and $size(V_v)$ is $\prod_{v_{org} \in V_v} w_{v_{org}}$ ($w_{v_{org}}$ is the weight of v_{org}), and minimize the balance score as the vertices are merged in a similar manner that interFD-based vertex partitioning minimizes the average edge weight. Note that the balance score will be 1.0 for the most balanced cut and the higher the score is, the less the partitions are balanced. Instead of using average weights as in interFD based partitioning, we now select the next cut based on the resulting balance score.

Given an undirected graph $G_{pfd}(V, E)$, the algorithm makes a copy V' or V and first picks the vertex v with the *minimum balance score*, among all vertices in V'. If there are multiple alternatives, then the algorithm selects the one which has the cut with the minimum average weight among the alternatives. Then, the algorithm selects a neighbor v' of v such that $\{v, v'\}$ has the smallest balance score; again, if there are alternatives, then the algorithm selects the neighbor such that $\{v,v'\}$ has a cut from $V'\backslash\{v, v'\}$ with the smallest average weight. The algorithm then shrinks V' by merging v and v' into a new vertex, v'', with the vertex weight, $w_{v''} = w_v \times w_{v'}$ and the balance score of v'' is computed using Equation 7.2. For any pair of edges $e = a \rightarrow v$ and $e' = a \rightarrow v'$ originating from the same vertex a, we create a new edge $e'' = a \rightarrow v''$, where the edge weight $w''_e = w_e + w'_e$ and counter $n''_e = n_e + n'_e$. Any other edge to v or v' is simply re-routed to v''. The process is stopped when the balance score is 1.0 or $|V'| = 1$. The most balanced cut among the most balanced cuts of each step is returned. The following is an example of the balanced cut algorithm.

Example 7.3.2 *For the balanced cut algorithm, we consider the graph $G_{pfd}(V, E)$ with weighted edges and weighted vertices (see Figure 7.5(a)).*

- *Step 1: The cut between $\{b\}$ and $\{a,\ c,\ d\}$ with the minimum average cut is chosen out of the two alternative cuts, (1) $\{b\}$ and $\{a,\ c,\ d\}$ ($\bar{w}_{\{b\}} = 0.44$) and (2) $\{c\}$ and $\{a,\ b,\ d\}$ ($\bar{w}_{\{c\}} = 0.4533$), with the equal balance score, (10 × 20 × 10) / 20 (see Figure 7.5(b)).*



a:10
b:20
c:20
d:10
0.40
0.50
0.43
0.48
0.44
0.45
b,d: 20×10
$w_{a\text{->}\{b,d\}}$=0.90
$n_{a\text{->}\{b,d\}}$=2
$w_{c\text{->}\{b,d\}}$=0.93
$n_{c\text{->}\{b,d\}}$=2

(a) $G_{pfd}(V, E)$ (b) Step 1 (c) Step 2

Figure 7.5: An example of the balanced cut algorithm for $G_{pfd}(V, E)$ with weighted edges and weighted vertices (see Example 7.3.2.)

- *Step 2: The vertex d is chosen with the minimum average weight out of two neighbors of the vertex b, (1) vertex a (*$\bar{w}_{\{a,b\}} = (0.43 + 0.48 + 0.50 + 0.44)/4 = 0.4625$*) and (2) vertex d (*$\bar{w}_{\{b,d\}} = (0.45 + 0.48 + 0.50 + 0.40)/4 = 0.4575$*) with the equal smallest balance score 1.0 and merged with the vertex b into* $\{b, d\}$ *with the vertex weight (20×10). Edges* $a \rightarrow b$ *and* $a \rightarrow d$ *are replaced by the edge* $a \rightarrow \{b, d\}$ *with weight* $w_{a\rightarrow\{b,d\}} = 0.40 + 0.50 = 0.90$ *and counter* $n_{a\rightarrow\{b,d\}} = 2$*. Edges* $c \rightarrow b$ *and* $c \rightarrow d$ *are replaced by the edge* $c \rightarrow \{b, d\}$ *with weight* $w_{c\rightarrow\{b,d\}} = 0.48 + 0.45 = 0.93$ *and counter* $n_{c\rightarrow\{b,d\}} = 2$*. The cut between* $\{a, c\}$ *and* $\{b, d\}$ *is the most balanced cut (balance score: 1.0) with the minimum average weight (*$\bar{w}_{\{b,d\}} = 0.4575$*); the process ends since the balance score is 1.0 (see Figure 7.5(c)).*

7.4 Further Optimizations: Rank Pruning based on Intra-Partition Dependencies

As discussed in the previous section, given a partitioning of $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$, to obtain a rank-r decomposition of $\mathcal{R}$ using JBD, we need to consider rank-r_1 and rank-r_2 decompositions of $\mathcal{R}_1$ and $\mathcal{R}_2$, such that $r = r_1 \times r_2$ and pick the (r_1, r_2) pair which is likely to minimize recombination errors. In this section, we argue that we can rely on the supports of the dependencies that make up the partitions $\mathcal{R}_1$ and $\mathcal{R}_2$ to prune

(r_1, r_2) pairs which are not likely to give good fits. In particular, we observe that the higher the overall dependency between the attributes that make up a partition, the more likely the data in the partition can be described with a smaller number of clusters. Since the number of clusters of a data set is related to the rank of the decomposition, this leads to the observation that the higher the overall dependency between the attributes in a partition, the smaller should be the decomposition rank of that partition.

Thus, given $\mathcal{R}_1$ and $\mathcal{R}_2$, we need to consider only those rank pairs (r_1, r_2), where if the average intra-partition FD support for $\mathcal{R}_1$ is larger than the support for $\mathcal{R}_2$, then $r_1 < r_2$ and vice versa. We refer to this as the *intraFD* criterion for rank pruning. Similarly to interFD, given the supports of FDs, we define intraFD as the average support of the pairwise FDs (excluding the pairwise FDs involving the join attribute) within each partition. In Section 7.6, we evaluate the effect of the interFD-based partitioning and intraFD-based rank pruning strategy of DBN for both dense and sparse tensor decomposition in terms of the efficiency and the accuracy.

7.5 Cost Analysis

In this section, we provide cost analyses for decomposition-by-normalization strategies for CP and Tucker decompositions (DBN-CP and DBN-Tucker, respectively).

Unlike the conventional tensor decomposition process, DBN involves a data partitioning (normalization) step followed by a series of partial decompositions, joins, and candidate selection steps (see Section 7.2.2): For a target r decomposition, DBN performs as many partial decompositions as the number, n_{pair}, of rank pairs (r_p, r_q) where $r = r_p \times r_q$. Join and norm-based candidate selection steps of the DBN are also performed once for each pair (r_p, r_q). Since these costs are negligible compared to the

Table 7.1: Notations used in this chapter

Notation	Description
$\mathcal{X}$	the input tensor of size $K_1 \times K_2 \times \cdots \times J \times \cdots \times K_{N_x}$ let $\mathcal{X}$ be partitoned into $\mathcal{P}$ and $\mathcal{Q}$ on the join mode $\mathbf{J}$ of size J; i.e., $\mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}$
J	the size of the join mode $\mathbf{J}$
$\mathcal{P}$	the 1^{st} partition tensor of size $I_1 \times I_2 \times \cdots \times J \times \cdots \times I_{N_p}$
$\mathcal{Q}$	the 2^{nd} partition tensor of size $I'_1 \times I'_2 \times \cdots \times J \times \cdots \times I'_{N_q}$
r	the rank of $\mathcal{X}$
$r_{p,i}$ and $r_{q,i}$	the i^{th} ranks of $\mathcal{P}$ and $\mathcal{Q}$, resp.; i.e., $(r_{p,i}, r_{q,i}) \in \{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}$
N_p	# of modes of $\mathcal{P}$
N_q	# of modes of $\mathcal{Q}$
N_x	# of modes of $\mathcal{X}$
$\alpha_{r,*}$	# of ALS iterations needed for the rank-r CP decomposition of the tensor denoted by "*"
$\lvert\mathcal{P}\rvert$	# of nonzero entries of a tensor $\mathcal{P}$
$\lvert\mathcal{Q}\rvert$	# of nonzero entries of a tensor $\mathcal{Q}$
$\lvert\mathcal{X}\rvert$	# of nonzero entries of a tensor $\mathcal{X}$
n_{pair}	# of (r_p, r_q); i.e., $\lvert\{(r_{p,i}, r_{q,i}) \mid r_{p,i} \times r_{q,i} = r\}\rvert$

Table 7.2: Execution time cost for CP decomposition

Algorithm		Cost
CP	dense tensors	$O(\prod_{i=1}^{N_x} K_i)^{\dagger}$
	sparse tensors	$O(\alpha_{r,\mathcal{X}}\ r\ \lvert\mathcal{X}\rvert\ N_x)^{\dagger\dagger}$
DBN-CP	dense tensors	$O(n_{pair}(\prod_{i=1}^{N_p} I_i + \prod_{i=1}^{N_q} I'_i))^{\dagger}$
	sparse tensors	$O(\sum_{i=1}^{n_{pair}}(\alpha_{r_{p,i},\mathcal{P}}\ r_{p,i}\ \lvert\mathcal{P}\rvert\ N_p + \alpha_{r_{q,i},\mathcal{Q}}\ r_{q,i}\ \lvert\mathcal{Q}\rvert\ N_q))^{\dagger\dagger}$

†The execution time cost for dense tensors is based on [70].

††The execution time cost for sparse tensors is based on the analysis of the code in [12].

tensor decomposition cost, in this section, we focus on tensor compositions cost.

7.5.1 *Cost of DBN-CP*

Table 7.2 presents an overview of the execution times for conventional CP (simply called CP in the rest of this section) and DBN-CP. Symbols used in this section are introduced in Table 7.1.

CP Decomposition of Dense Tensors

As we see in Table 7.2, For dense tensors, the DBN strategy increases the number of decomposition operations from one to n_{pair} (the number of rank-pairs), but each decomposition involves smaller numbers of modes. Since, in the case of dense tensors, the cost of the CP decomposition is exponential in the numbers of modes and since DBN-CP reduces the number of modes that need to be considered, as we experimentally observe in Section 7.6, DBN-CP is more efficient than the conventional CP decomposition.

CP decomposition of Sparse Tensors

As we also see in Table 7.2, the execution time cost of the conventional CP operation for sparse tensors depends on the rank, the number of nonzero entries, the number of modes, as well as the number of alternating least squares (ALS) iterations [12].

When all things equal, the main contributor to the cost of the CP decomposition on sparse tensors is the number of nonzero entries. Therefore, in order to predict whether CP or DBN-CP will be more efficient, we need to consider the number of nonzero entries in the input tensors: In particular, if the ratio

$$\phi = |\boldsymbol{\mathcal{X}}|/(|\boldsymbol{\mathcal{P}}||\boldsymbol{\mathcal{Q}}|)$$

is high and we have more tuples (nonzero entries) in the input tensor than the partition tensors, then DBN-CP is likely to be more efficient than the CP; otherwise, CP may be competitive. In other words, DBN-CP is likely to outperform CP if the following holds:

$$r|\boldsymbol{\mathcal{X}}|N_x > \sum_{i=1}^{n_{pair}} (r_{p,i}|\boldsymbol{\mathcal{P}}|N_p + r_{q,i}|\boldsymbol{\mathcal{Q}}|N_q),$$

or, equivalently,

$$|\boldsymbol{\mathcal{X}}| > \sum_{i=1}^{n_{pair}} (r_{p,i}|\boldsymbol{\mathcal{P}}|N_p + r_{q,i}|\boldsymbol{\mathcal{Q}}|N_q)/(rN_x).$$

Table 7.3: Notations for Tucker decomposition used in this chapter

Notation	Description
$r_{x,1}, ..., r_{x,N_x}$	decomposition ranks for $\mathcal{X}$
$r_{p,1}, ..., r_{p,N_p}$	decomposition ranks for $\mathcal{P}$
$r_{q,1}, ..., r_{q,N_q}$	decomposition ranks for $\mathcal{Q}$
$r_{p,i,l}$, $r_{q,j,l}$, and $r_{x,k}$	the l^{th} rank pair ($r_{p,i,l}$, $r_{q,j,l}$) of the join modes (i^{th} and j^{th} modes) of $\mathcal{P}$ and $\mathcal{Q}$ and the rank ($r_{x,k}$) of the join mode (k^{th} mode) of $\mathcal{X}$; i.e., $(r_{p,i,l}, r_{q,j,l}) \in \{(r_{p,i,l}, r_{q,j,l}) \mid r_{p,i,l} \times r_{q,j,l} = r_{x,k}\}$
n_{pair}	# of $(r_{p,i,l}, r_{q,j,l})$; i.e., $\lvert\{(r_{p,i,l}, r_{q,j,l}) \mid r_{p,i,l} \times r_{q,j,l} = r_{x,k}\}\rvert$
$\beta_{(r_1,...r_N),*}$	# of ALS iterations needed for the rank-$(r_1, ..., r_N)$ Tucker decomposition of the tensor denoted by "*"
ε_*	a subset of modes that are computed element-wise in MET for the tensor denoted by "*"
$C_{m,*}$	the eigen decomposition cost for the mth mode of the tensor denoted by "*" †

†For eigen decomposition, we assume that MATLAB's `eigs` function based on ARPACK uses an iterative power method to identify eigenvalues. Therefore, the overall eigen decomposition cost is a function of this iteration count.

Since we have $|\mathcal{X}| = |\mathcal{P} \bowtie_{=,\mathbf{J}} \mathcal{Q}| = \phi|\mathcal{P}||\mathcal{Q}|$, we can rewrite the above inequality as

$$\phi(|\mathcal{P}||\mathcal{Q}|) > \sum_{i=1}^{n_{pair}} (r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q)/(rN_x).$$

This gives us a lower bound, $\phi_\perp$, on the join selectivity:

$$\phi > \phi_\perp = \sum_{i=1}^{n_{pair}} (r_{p,i}|\mathcal{P}|N_p + r_{q,i}|\mathcal{Q}|N_q)/(|\mathcal{P}||\mathcal{Q}|rN_x).$$

This lower bound threshold provides a practical predictor to judge whether DBN-CP will be more advantageous, for sparse tensors, than CP.

7.5.2 Cost of DBN-Tucker

Table 7.3 lists additional notations needed for the analysis of the Tucker decomposition costs.

The main cost of Tucker decomposition for dense tensors is the number of modes, which is similar to CP decomposition for dense tensors. Therefore, the costs analysis

Table 7.4: Execution time cost for Tucker decomposition on sparse tensors

Algorithm		Cost
MET	$TTM_x(m)^\dagger$	$O(\sum_{m' \neq m}(\lvert\mathcal{X}\rvert K_{m'} r_{x,m'}) \prod_{m' \neq \varepsilon_x} r_{x,m'})$
	$SVD_x(m)^\dagger$	$O(K_m^2 \times \prod_{m' \neq m} r_{x,m'} + C_{m,\mathcal{X}})$
	Total	$O(\beta_{(r_{x,1},\ldots,r_{x,N_x}),\mathcal{X}} \sum_{m=1}^{N_x}(TTM_x(m) + SVD_x(m))$
DBN-Tucker (using MET)	$TTM_p(m)^\dagger$	$O(\sum_{m' \neq m}(\lvert\mathcal{P}\rvert I_{m'} r_{p,m'}) \prod_{m' \neq \varepsilon_p} r_{p,m'})$
	$SVD_p(m)^\dagger$	$O(I_m^2 \times \prod_{m' \neq m} r_{p,m'} + C_{m,\mathcal{P}})$
	$TTM_q(m)^\dagger$	$O(\sum_{m' \neq m}(\lvert\mathcal{Q}\rvert I'_{m'} r_{q,m'}) \prod_{m' \neq \varepsilon_q} r_{q,m'})$
	$SVD_q(m)^\dagger$	$O({I'}_m^2 \times \prod_{m' \neq m} r_{q,m'} + C_{m,\mathcal{Q}})$
	Total	$O(\sum_{l=1}^{n_{pair}}(\beta_{(r_{p,1},\ldots,r_{p,i,l},\ldots,r_{p,N_p}),\mathcal{P}} \sum_{m=1}^{N_p}(TTM_p(m) + SVD_p(m)) +$ $\beta_{(r_{q,1},\ldots,r_{q,j,l},\ldots,r_{q,N_q}),\mathcal{Q}} \sum_{m=1}^{N_q}(TTM_q(m) + SVD_q(m))$

†MET algorithm we consider consists of two major steps applied to each mode m: (a) TTM computation and (b) SVD computation; see [45] for details.

Table 7.5: Bottleneck memory cost for Tucker decomposition

Algorithm	Cost
MET	$O(\max_{\varepsilon_x}(\prod_{m \notin \varepsilon_x} K_m))^\dagger$
DBN-Tucker (using MET)	$O(\max(\max_{\varepsilon_p}(\prod_{m \notin \varepsilon_p} I_m), \max_{\varepsilon_q}(\prod_{m \notin \varepsilon_q} I'_m)))$

†The costs are based on [45].

for DBN-Tucker on dense tensors also follows the cost analysis of DBN-CP on dense tensors presented in Table 7.2.

The cost analysis for sparse tensors, on the other hand, is more complex. We focus on Tucker decomposition for sparse tensors.

In Table 7.4, we present the cost analysis of DBN-Tucker on sparse tensors assuming that it is build on MET [11]. As before, DBN-Tucker, involves as many partial Tucker decompositions as the number, n_{pair}, of rank pairs (r_p, r_q) where $r = r_p \times r_q$, but each decomposition involves smaller number of modes. Since, as we see in Table 7.4, the cost of Tucker tensor decomposition is exponential in the number of modes, we expect that DBN-Tucker will be more efficient than conventional Tucker decompositions. Experiment results reported in Section 7.6 verify this.

Note that as reported in Table 7.5, one major benefit for the proposed DBN based Tucker decomposition scheme is that the size of the intermediate results is smaller than that for conventional Tucker decomposition: this is because DBN decomposes smaller sub-tensors. Since a major challenge in Tucker decompositions is the memory needed to store the intermediary results, for large data sets, and especially when the available memory is limited, DBN-Tucker is likely to be more advantageous. We experimentally verify this in Section 7.6.

7.6 Experimental Evaluations

In this section, we present the result of the experiments we have carried out to assess the efficiency and effectiveness of the decomposition-by-normalization (DBN) strategy. We consider both CP and Tucker decompositions and use both sparse and dense tensors. We ran our experiments on a 6-core Intel(R) Xeon(R) CPU X5355 @ 2.66GHz machine with 24GB of RAM.

7.6.1 Setup - Data Sets

For evaluating DBN under different scenarios, we used various data sets from the UCI Machine Learning Repository [31]. In particular, we considered the two cases introduced in Section 7.3:

Case 1. We first evaluate DBN in situations where the join attribute X determines all attributes of the relation $\mathcal{R}$. For these experiments, we considered 15 different data sets (D1-D15) with different sizes and different attribute sets (Table 7.6). All tensors were encoded as occurrence tensors. In the cases where a suitable join attribute did not exist in the data, we selected an attribute with FD support $\geq \tau_{support} = 75\%$ against all other attributes. We then removed all non-supporting tuples to make sure that the data set $\mathcal{R}$ satisfies the properties of Case 1. Note that, each partitioned data set contains as many tuples (nonzero entries) as the input relation $\mathcal{R}$.

Table 7.6: Relational tensor data sets with the same number of nonzero entries for partitions for DBN

Data set		Size	# nonzero
D1	Adult	118×90×20263×5×2	20263
D2		7×20263×5×6×16	20263
D3		72×20263×90×2×2	20263
D4		20263×14×2×6×94	20263
D5		20263×5×2×90×72	20263
D6	Breast Cancer Wisconsin [53]	645×10×11×2×10	630
D7		10×645×9×10×10	630
D8		10×10×11×10×645	630
D9		2×10×10×10×645	630
D10		10×10×645×9×10	630
D11	IPUMS Census Database [63]	3890×4×13×3×3	4863
D12		545×3×17×3×2	698
D13		11×3×4×5×3	27
D14	Mushroom	10×3×5×2×7	24
D15	Dermatology	62×5×5×5×3	58

Table 7.7: Relational tensor data sets with different numbers of nonzero entries for partitions for DBN

Data set		Mode	Size	# nonzero of $\mathcal{R}_1$	# nonzero of $\mathcal{R}_2$
D16	Adult (subset)†	5	118×90×1000×5×2	1000	1102
D17	Adult	4	118×90×20263×94	20263	25331
		5	118×90×20263×94×72	20263	27351
		6	118×90×20263×94×72×42	20263	27424
D18	IPUMS Census Database	4	2241×1096×191×209	1096	2359
		5	3888×2241×1096×191×209	1096	5881
		6	3890×2241×51×1096×192×209	1096	6436

†For D16, we used a subset of randomly selected 1,000 entries from this data set for experiments with dense tensor model: the whole data set is too large for conventional decomposition operators under the dense tensor model.

Case 2. Secondly, we evaluate DBN in situations where the join attribute X determines only a subset of the attributes of the relation $\mathcal{R}$. In this case, we considered three different data sets (D16-D18). All tensors were encoded as occurrence tensors. The tensor size and numbers of nonzero entries of each relation are shown in Table 7.7. Note that the partition $\mathcal{R}_1$ containing X and the attributes determined by X has potentially smaller number of nonzero entries than $\mathcal{R}$; the number of nonzero entries of the other partition $\mathcal{R}_2$ is same as that of $\mathcal{R}$. As we discussed in Section 7.3, for dense tensors and Tucker decompositions, we targeted partitions where sizes of $\mathcal{R}_1$ and $\mathcal{R}_2$ are similar.

7.6.2 Setup - Target Ranks

Both for CP and Tucker decomposition experiments, we considered three target ranks: 6, 12, and 24. These lead to 4 rank pairs ($\langle 1,6\rangle$, $\langle 2,3\rangle$, $\langle 3,2\rangle$, $\langle 6,1\rangle$) to be considered for the target rank 6, 6 pairs ($\langle 1,12\rangle$, $\langle 2,6\rangle$, $\langle 3,4\rangle$, $\langle 4,3\rangle$, $\langle 6,2\rangle$, $\langle 12,1\rangle$) for the target rank 12, and 8 pairs ($\langle 1,24\rangle$, $\langle 2,12\rangle$, $\langle 3,8\rangle$, $\langle 4,6\rangle$, $\langle 6,4\rangle$, $\langle 8,3\rangle$, $\langle 12,2\rangle$, $\langle 24,1\rangle$) for the target rank 24.

7.6.3 Setup - Alternative Tensor Decomposition Algorithms

We experimented with various alternative algorithms for CP and Tucker decompositions. Table 7.8 lists the various algorithms we use in our experiments. We used MATLAB Version 7.11.0.584 (R2010b) 64-bit (glnxa64) and MATLAB Parallel Computing Toolbox.

CP Decomposition (Single Core)

The first decomposition algorithm we considered is the N-way PARAFAC algorithm with nonnegativity constraint which is available in the N-way Toolbox for MAT-

Table 7.8: Algorithms

Algorithm		Description
CP	DBN-NWAY	DBN-CP using N-way PARAFAC
	DBN-CP-ALS	DBN-CP using single grid NTF (CP-ALS)†
	NNCP-NWAY	NNCP using N-way PARAFAC
	NNCP-CP-ALS	NNCP using single grid NTF (CP-ALS)†
	NNCP-NWAY-GRID*	NNCP using grid NTF with "*" grid cells (N-way PARAFAC)†
	NNCP-CP-GRID*	NNCP-CP-ALS with "*" grid cells
	DBN*-CP-ALS	intraFD-based DBN-CP-ALS with "*" pairs
	DBN*-NWAY	intraFD-based DBN-NWAY with "*" pairs
	pp-DBN*-CP-ALS	pairwise parallel DBN*-CP-ALS
	pp-DBN*-NWAY	pairwise parallel DBN*-NWAY
Tucker	MET*	"*" modes element-wise Memory-Efficient Tucker
	DBN-MET*	DBN-Tucker using MET*
	pp-DBN-MET*	pairwise parallel DBN-MET*

†The algorithms in parentheses are the base PARAFAC for grid NTF

LAB [9]. We refer to DBN-CP and conventional non-negative CP (NNCP) implemented using this N-way PARAFAC implementation as DBN-NWAY and NNCP-NWAY, respectively.

Since MATLAB's N-way PARAFAC implementation uses a dense tensor (multi-dimensional array) representation, it is too costly to be practical for sparse tensors. Therefore, we implemented a variant of the single grid NTF [59] using CP-ALS as the base PARAFAC algorithm. We refer to DBN-CP and NNCP based on CP-ALS as DBN-CP-ALS and NNCP-CP-ALS respectively.

CP Decomposition (Parallel, Multi-core)

For the parallel version of the NNCP, we implemented the grid NTF algorithm [59] with different number of grid cells (2, 4, 6, and 8 grid cells along the join mode) using N-way PARAFAC and CP-ALS as the base PARAFAC algorithms. Each grid is run with the base PARAFAC algorithm separately in parallel. We refer to the grid

NTF algorithm for parallel NNCP implemented using N-way PARAFAC as NNCP-NWAY-GRID* (* denotes the number of partitions). Similarly, we refer to CP-ALS based implementations of parallel NNCP as NNCP-CP-GRID*.

The parallel version of DBN-CP are implemented using pairwise parallel DBN-NWAY and DBN-CP-ALS strategies where each pair is assigned to a separate processing unit; these are referred to as pp-DBN-NWAY and pp-DBN-CP-ALS respectively.

Tucker Decomposition (Single Core)

Conventional Tucker decomposition algorithms, such as [9], are ineffective on large dense data sets. Therefore, we focus on Tucker decompositions of sparse data sets. We consider MET (Memory-Efficient Tucker) in [45]. For MET, we considered different variants, denoted as MET* according to the number of modes handled element-wise; MET* is also used as the base Tucker algorithm for DBN-Tucker; this is referred to as DBN-MET*.

Tucker Decomposition (Parallel, Multi-core)

Since MET does not support parallelization, we only consider parallelization of DBN-MET*, which is referred to as pp-DBN-MET*.

7.6.4 Setup - Rank Pruning

For the experiments where we assess the impact of the intraFD-based rank pruning strategy described in Section 7.4. We considered 2, 3 and 4 pairs as limits; these are referred to as DBN2, DBN3, and DBN4, respectively (e.g., DBN-CP-ALS with 2 pairs selected is referred to as DBN2-CP-ALS).

Table 7.9: Different attribute sets, join attributes (X), supports of X (the lowest of all the supports of $X \to *$), and execution times for FDs discovery for D1-D18 where A_n is the n^{th} attribute of each data set

Data set	Attributes		Join attr. (X)	Support of X	exec. time for FDs
D1	$\{A_{11}, A_{12}, A_3, A_9, A_{10}\}$		A_3	97%	0.024s
D2	$\{A_2, A_3, A_9, A_8, A_4\}$		A_3	80%	0.022s
D3	$\{A_1, A_3, A_{12}, A_{15}, A_{10}\}$		A_3	80%	0.025s
D4	$\{A_3, A_7, A_{15}, A_8, A_{13}\}$		A_3	75%	0.023s
D5	$\{A_3, A_9, A_{15}, A_{12}, A_1\}$		A_3	80%	0.023s
D6	$\{A_1, A_4, A_7, A_{11}, A_6\}$		A_1	96%	0.004s
D7	$\{A_4, A_1, A_{10}, A_8, A_9\}$		A_1	96%	0.003s
D8	$\{A_6, A_5, A_7, A_8, A_1\}$		A_1	96%	0.002s
D9	$\{A_{11}, A_9, A_6, A_3, A_1\}$		A_1	98%	0.003s
D10	$\{A_5, A_4, A_1, A_{10}, A_8\}$		A_1	96%	0.003s
D11	$\{A_8, A_{17}, A_{19}, A_3, A_2\}$		A_8	99%	0.007s
D12	$\{A_{53}, A_2, A_{21}, A_3, A_4\}$		A_{53}	98%	0.006s
D13	$\{A_{13}, A_{48}, A_{17}, A_{14}, A_2\}$		A_{13}	98%	0.005s
D14	$\{A_4, A_9, A_{18}, A_{17}, A_2\}$		A_2	88%	0.004s
D15	$\{A_{34}, A_{24}, A_{33}, A_{25}, A_{11}\}$		A_{34}	80%	0.002s
D16	$\{A_{11}, A_{12}, A_3, A_9, A_{10}\}$		A_3	98%	0.024s
D17	4-mode	$\{A_{11}, A_{12}, A_3, A_{13}\}$	A_3	96%	0.024s
	5-mode	$\{A_{11}, A_{12}, A_3, A_{13}, A_1\}$			
	6-mode	$\{A_{11}, A_{12}, A_3, A_{13}, A_1, A_{14}\}$			
D18	4-mode	$\{A_{49}, A_{50}, A_{51}, A_{54}\}$	A_{50}	95%	0.007s
	5-mode	$\{A_8, A_{49}, A_{50}, A_{51}, A_{54}\}$			
	6-mode	$\{A_8, A_{49}, A_{50}, A_{51}, A_{54}, A_{58}\}$			

7.6.5 Setup - Functional Dependency Discovery

As discussed in Section 7.2.2, we extended TANE [36] to find approximate FDs. The supports of the approximate FDs for each attribute set of different relational data sets are shown in Table 7.9. The table also shows the execution times needed to discover the FDs for each data set. As the table shows, the modified TANE algorithm is very efficient for the considered numbers of attributes, i.e., 4 to 6 attributes in these experiments [1] . Since the execution times for finding approximate FDs are negligible compared to the tensor decomposition time, in the rest of the section, we focus only on the decomposition times.

7.6.6 Setup - Evaluation Criteria

Each experiment is run at least 5 times and we report the average accuracy and execution time of these runs.

Accuracy

We use the following *fit* function to measure tensor decomposition accuracy:

$$\mathrm{fit}(\mathcal{X}, \hat{\mathcal{X}}) = 1 - \frac{\|\mathcal{X}, \hat{\mathcal{X}}\|}{\|\mathcal{X}\|}. \tag{7.3}$$

Here, $\|\mathcal{X}\|$ is the Frobenius norm of a tensor $\mathcal{X}$. The fit is a normalized measure of how accurate a tensor decomposition of $\mathcal{X}$, $\hat{\mathcal{X}}$ w.r.t. a tensor $\mathcal{X}$.

Execution Time

The execution times are measured by MATLABs `tic` and `toc` commands to start and stop the clock at the beginning and the end of the decomposition process, respectively.

Memory

Especially for dense tensors and Tucker decomposition, memory usage can be a major bottleneck. For Tucker decompositions, we report the maximum intermediate memory use provided by the MET* algorithm. For single core DBN, we report the maximum of the memory used by each rank-pair (evaluated one after the other). For parallel DBN, we report the sum of the memory used by each rank-pair (evaluated in parallel).

[1]Note that the cost increases linearly in the size of the input relation [36]

Table 7.10: Join selectivity (ϕ) and thresholds with and without rank pruning ($\phi'_\perp$ and $\phi_\perp$). The bold fonts are the cases where the join selectivity is greater than the threshold.

Data set	mode	rank	$\phi_\perp$	$\phi'_\perp$ with rank pruning	ϕ
D1-D5	5-mode	rank-6	0.00012	0.00006	0.00005
		rank-12	0.00014	0.00007	
D6-D10	5-mode	rank-6	0.00381	0.00190	0.00159
		rank-12	0.00444	0.00222	
D11	5-mode	rank-6	0.00049	0.00025	0.00021
		rank-12	0.00058	0.00029	
D12	5-mode	rank-6	0.00344	0.00172	0.00143
		rank-12	0.00401	0.00201	
D13	5-mode	rank-6	0.08889	0.04444	0.03704
		rank-12	0.10370	0.05185	
D14	5-mode	rank-6	0.10000	0.05000	0.04167
		rank-12	0.11667	0.05833	
D15	5-mode	rank-6	0.04138	0.02069	0.01724
		rank-12	0.04828	0.02414	
D16	5-mode	rank-6	0.00229	0.00117	0.001
		rank-12	0.00267	0.00137	
D17	4-mode	rank-6	0.00011	0.00007	0.00005
		rank-12	0.00013	0.00008	
	5-mode	rank-6	0.00010	0.00006	0.00005
		rank-12	0.00012	0.00007	
	6-mode	rank-6	0.00010	**0.00004**	**0.00005**
		rank-12	0.00012	0.00007	
D18	4-mode	rank-6	0.00179	0.00113	0.00091
		rank-12	0.00209	0.00136	
	5-mode	rank-6	0.00130	**0.00087**	**0.00091**
		rank-12	0.00152	0.00105	
	6-mode	rank-6	0.00137	**0.00042**	**0.00091**
		rank-12	0.00191	0.00122	

7.6.7 Execution Time Results for CP Decompositions

We first present experimental results assessing the efficiency of the proposed DBN scheme relative to the conventional implementation of the CP based tensor decomposition in both stand-alone and parallelized versions. Note that, as discussed in Section 7.5, for sparse tensors, CP decomposition cost depends largely on the number of nonzero entries and this necessitates a way to leverage the join selectivity (of the partitioning attribute) to predict whether DBN will outperform conventional CP decomposition schemes. In Table 7.10, we report the join selectivity ϕ and selectivity cut-off, $\phi_{\perp}$, values (with and without rank pruning) for each of the data sets we considered in our experiments: we predict that DBN will most easily outperform the conventional CP decomposition schemes (even for sparse data) in the cases where $\phi > \phi_{\perp}$. As we see in this table, however, in many cases, ϕ is lower than $\phi_{\perp}$. Therefore, we expect these situations to be challenging for DBN against conventional schemes. We evaluate this prediction in the following subsections and also show that DBN provides advantages even in these cases when parallel execution plans are considered.

As described in Section 7.3, for CP decompositions, we need to consider two distinct situations. In the first of these (D1-D15), the join attribute X determines all attributes of the relation $\mathcal{R}$; i.e., $\text{nnz}(\mathcal{R}_1) = \text{nnz}(\mathcal{R}_2) = \text{nnz}(\mathcal{R})$, where $\text{nnz}(\mathcal{X})$ denotes the number of nonzero entries of $\mathcal{X}$. In the second case, the join attribute X determines only a subset of the attributes of the relation $\mathcal{R}$; i.e., $\text{nnz}(\mathcal{R}_1) \leq \text{nnz}(\mathcal{R})$ and $\text{nnz}(\mathcal{R}_2) = \text{nnz}(\mathcal{R})$.

Case 1: X Determines all Attributes of $\mathcal{R}$

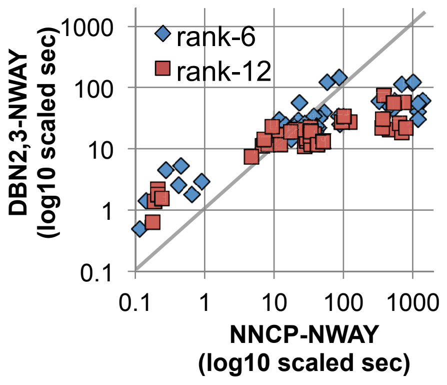
Dense Tensors: In Figure 7.6, we first compare the execution times for (NWAY based) DBN with NNCP for dense tensors. The figure includes results both for single-core and multi-core setups. As we see in these results, in both setups, DBN outperforms NNCP when the problems get more difficult to solve and tensor decomposition algorithms require more time. As the problem difficulty increases DBN provides $\sim$ 1 order (for single core) to $\sim$ 2 orders (for multi core) time gains over NNCP.

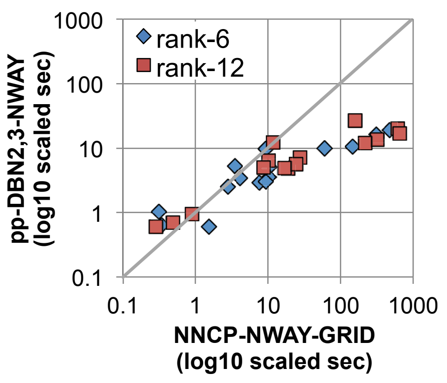
Sparse Tensors: In Figure 7.7, we compare the execution times for (CP-ALS based) DBN with NNCP for sparse tensors. The figure includes results both for single-core and multi-core setups. Remember that in this case, we predict that when ϕ of the input relations are lower than $\phi_{\perp}$, we expect DBN to have difficulties. This is confirmed in Figure 7.7(a), where we see that in single core scenarios, DBN-CP based schemes are not as competitive as NNCP as predicted based on the ϕ and $\phi_{\perp}$ values in Table 7.10.

It is important to note, however, that DBN still provides significant advantages even when $\phi < \phi_{\perp}$ when parallel execution opportunities are leveraged. As we see in Figure 7.7(b), on the same data, when using multiple cores, DBN based scheme outperforms NNCP in most cases.

Case 2: X does not Determine all Attributes of $\mathcal{R}$

Figure 7.8(a) shows the results for the corresponding subset of the `Adult` data set (D16) for which the conventional NWAY based decomposition schemes is feasible. As expected, DBN-CP based schemes outperform conventional CP decomposition schemes for different target ranks (rank-6 and rank-12) in both single-core and multi-





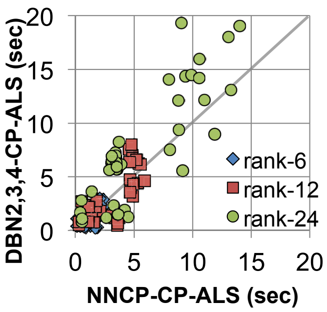
(a) Single core (b) Multiple cores

Figure 7.6: Average running times of DBN2,3-NWAY (DBN2 and DBN3 for rank-6 and rank-12, respectively) vs. NNCP-NWAY on (a) a single core and (b) 4 and 6 cores for rank-6 and rank-12, respectively (NNCP-NWAY-GRID is avg of GRID2 and GRID4 and avg of GRID2 and GRID6 for rank-6 and rank-12, respectively). On both a single core and multi cores, majority of data points are located under the diagonal, which indicates that DBN-NWAY outperforms NNCP-NWAY, especially when running times are bigger. Note that rank-24 results have been excluded from these charts because the conventional NWAY based NNCP is not feasible for this target rank with the hardware setup used for the experiments.

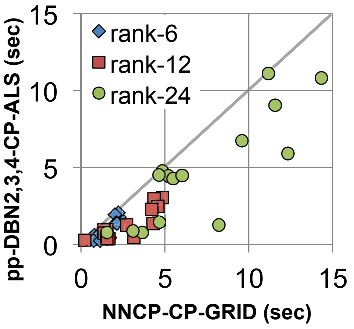
core settings.

In Figures 7.8(b) and (c), we compare DBN-CP against the CP-ALS based algorithms for different target ranks (rank-6, rank-12, and rank-24) on `Adult` (D17) and `IPUMS` (D18) data sets, respectively and in Figure 7.9 (a) and (b), we compare DBN-CP against the CP-ALS based algorithms for different number of modes (4-mode, 5-mode, and 6-mode) for rank-12 decomposition on `Adult` (D17) and `IPUMS` (D18) data sets, respectively.

As we see here, in almost all cases (especially when the data modality is high), DBN-CP based schemes outperform CP-ALS based schemes and pp-DBN-CP-ALS is the fastest in all cases. Note that these high modality cases are also the cases where the join selectivity ϕ of the relations are higher than the lower bound $\phi'_{\perp}$ with rank pruning (see Table 7.10) and the results confirm that DBN-CP is more advantageous in these cases as discussed in Section 7.5.1. It is interesting to note



(a) Single core



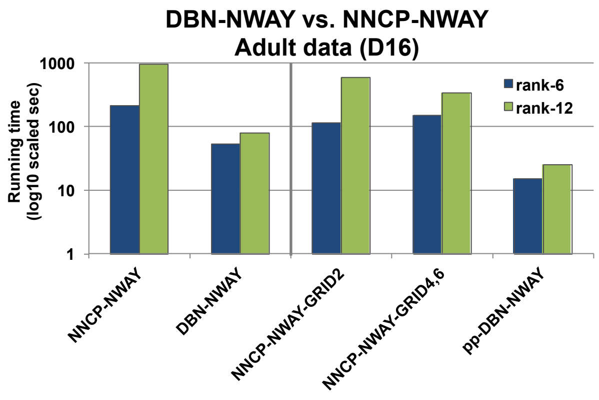
(b) Multiple cores

Figure 7.7: Average running times of DBN2,3,4-CP-ALS (DBN2, DBN3, and DBN4 for rank-6, rank-12, and rank-24, respectively) vs. NNCP-CP-ALS on (a) a single core and (b) 4, 6, and 8 cores for rank-6, rank-12, and rank-24 respectively (NNCP-CP-GRID is avg of GRID2 and GRID4, avg of GRID2 and GRID6, and avg of GRID2 and GRID8 for rank-6, rank-12, and rank-24, respectively). On a single core, more than half points are upper the diagonal; i.e., DBN-CP is beaten by NNCP. However, when DBN-CP and NNCP are parallelized, DBN-CP outperforms NNCP in most cases.

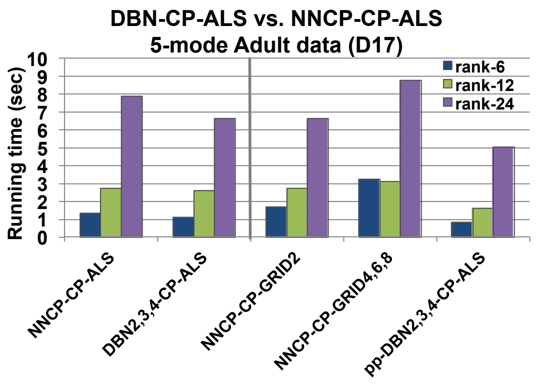
Table 7.11: Different partitioning cases for `Adult` (D17) and `IPUM` (D18) data sets. The partitions in bold are the most balanced among all three.

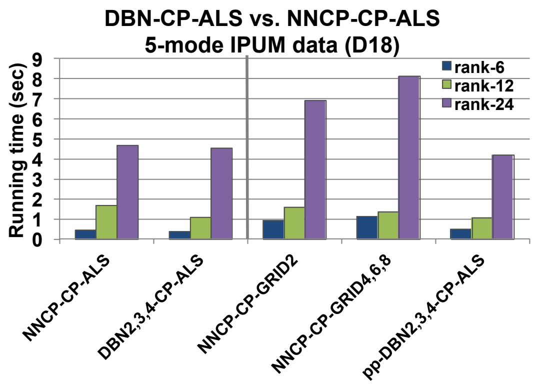
Data set	Partition size	
	$\mathcal{R}_1$	$\mathcal{R}_2$
D17	118×20264	91×20264×95×73
	91×20264	118×20264×95×73
	118×91×20264	**20264×95×73**
D18	3888×1096	2241×1096×191×209
	2241×1096	3888×1096×191×209
	3888×2241×1096	**1096×191×209**

that while GRID-based parallel version of CP-ALS may in practice negatively impact performance (since the underlying ALS-based combining approach involves significant communication overheads), parallelized DBN-CP is effective in reducing execution times.



(a)



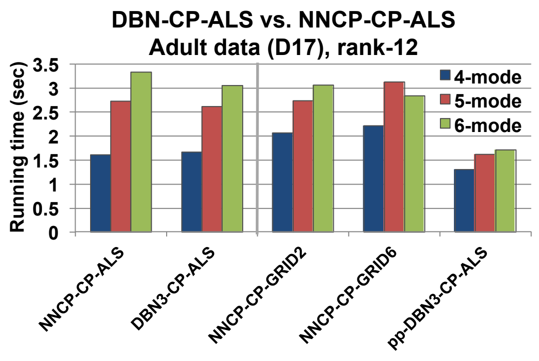


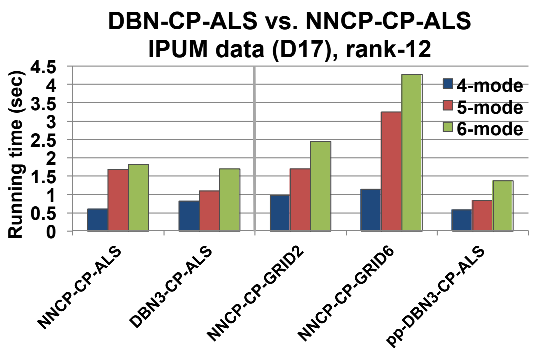
(b) (c)

Figure 7.8: Running times of (a) DBN-CP vs. NWAY based algorithms on 5-mode `Adult`(subset) data set (D16) and DBN-CP vs. CP-ALS based algorithms on (b) 5-mode `Adult` data set (D17) and (c) 5-mode `IPUM` data set (D18) with different target ranks in both single core and multi-core

7.6.8 Execution Time Results for Tucker Decompositions

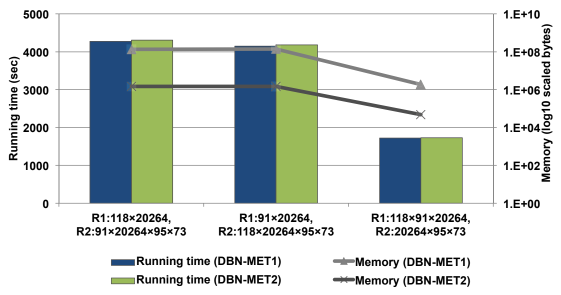
For the Tucker decomposition experiments, we focus on the data sets, D17 and D18, where the sizes of the modes are large. For comparison against the DBN strategy, we consider the MET (Memory-Efficient Tucker) [45] implementation of Tucker. Since there are multiple mode-selection strategies for MET, unless otherwise specified, we present the results for the strategy that leads to best running time and memory consumption for MET. We also consider different implementations of MET, denoted as MET1 and MET2.

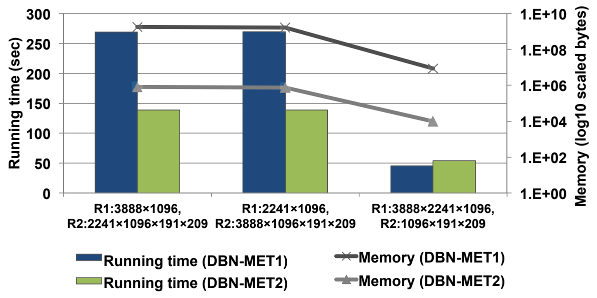




(a) `Adult` data set (D17) (b) `IPUMS` data set (D18)

Figure 7.9: Running times of DBN-CP vs. CP-ALS based algorithms for (a) `Adult` data set (D17) and (b) `IPUMS` data set (D18) for 4-mode, 5-mode, and 6-mode input tensors for rank-12 decomposition in both single-core and multi-core



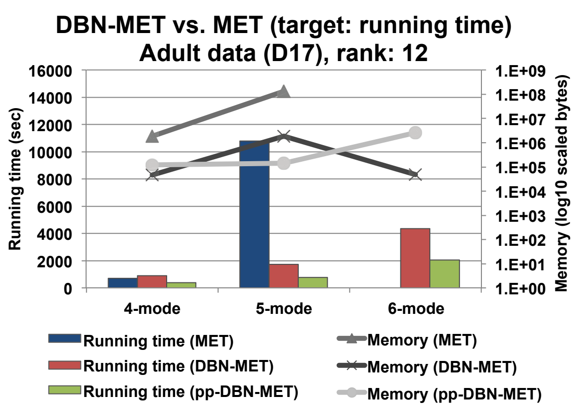


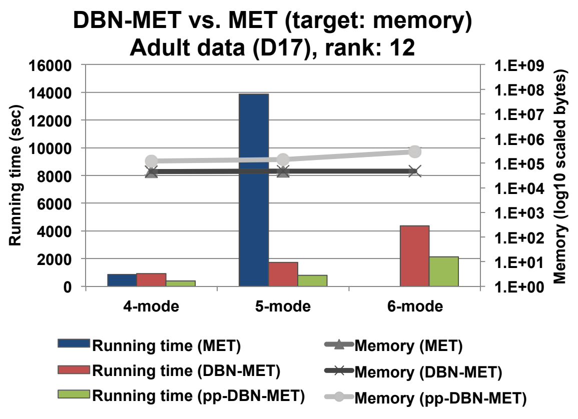
(a) `Adult` data set (D17) (b) `IPUM` data set (D18)

Figure 7.10: Running time and bottleneck memory for different partitioning cases where the two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ have different sizes (see Table 7.11), which are run by DBN-MET1 and DBN-MET2 for 5-mode (a) `Adult` data set (D17) (b) `IPUM` data set (D18) for rank-(12,12,12,12,12) decomposition

Impact of Partition Balance

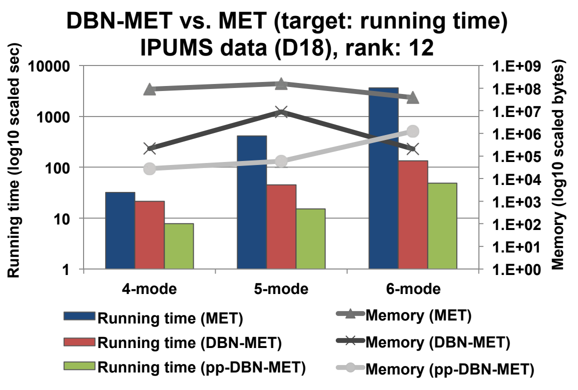
Before we compare the DBN strategy against conventional Tucker decompositions, we investigate the impact of partition balance on the performance of DBN. As we discussed in Section 7.3, for Tucker decompositions, we expect that partition strategies that lead to balanced sub-relations will lead to better DBN performance. In Figure 7.10, we present execution time and memory consumption results for three different partitioning strategies for each of the D17 and D18 data sets in Table 7.11. We note that($\mathcal{R}_1$: $\mathbf{118{\times}91{\times}20264}$, $\mathcal{R}_2$: $\mathbf{20264{\times}95{\times}73}$) and ($\mathcal{R}_1$: $\mathbf{3888{\times}2241{\times}1096}$,

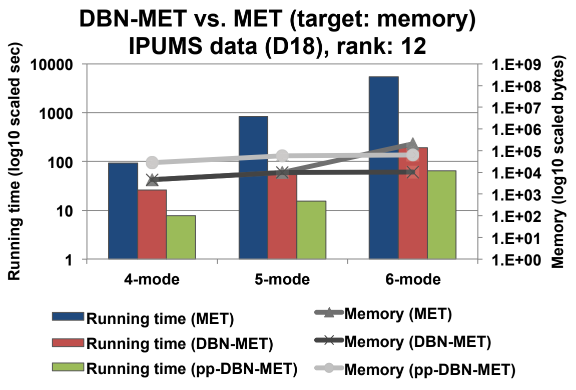




(a) Target: running time (b) Target: memory

Figure 7.11: The running time and bottleneck memory consumption for `Adult` data set (D17) of DBN-MET vs. MET: in (a) the optimization target is the running time, whereas in (b) the optimization target is the memory. Here, we use rank 12 for each mode of the relation. When the tensor has 6 modes none of the conventional MET algorithms fit the available memory and thus they are not included in the plots.



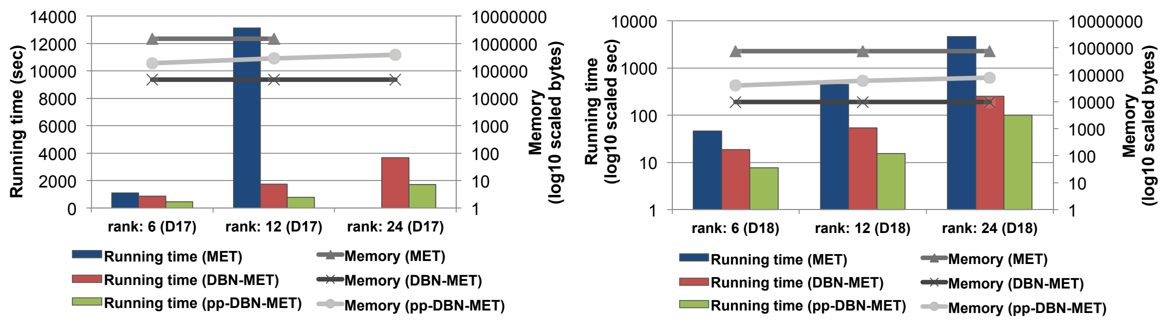


(a) Target: running time (b) Target: memory

Figure 7.12: The running time and bottleneck memory consumption for `IPUM` data set (D18) of DBN-MET vs. MET: in (a) the optimization target is the running time, whereas in (b) the optimization target is the memory. Here, we use rank 12 for each mode of the relation.

$\mathcal{R}_2$: $\mathbf{1096 \times 191 \times 209}$) partitioning alternatives are the most balanced among all three for the D17 and D18 data sets, respectively.

The results confirm that, as expected, the most balanced partitioning case (in terms of both size and number of modes) shows the best performance in terms of both running time and memory consumption.



(a) 5-mode `Adult` data set (D17) (b) 5-mode `IPUM` data set (D18)

Figure 7.13: The running time and bottleneck memory consumption of DBN-MET vs. MET for rank-(6,6,6,6,6), rank-(12,12,12,12,12), vs. rank-(24,24,24,24,24) (here we simply denote rank: 6, rank: 12, and rank: 24 respectively) for 5-mode `Adult` data set (D17) and `IPUM` data set (D18). For the D17 data set, for target rank 24, none of the conventional MET algorithms fit the available memory and thus they are not included in the plot.

DBN-MET vs. Conventional MET – Impact of the Number of Modes

We next compare the DBN strategy against conventional MET with respect to the impact of the number of modes of the tensor on the decomposition performance. Since there are multiple MET strategies with different run-times and memory consumptions, we present two sets of results, the first targeting better MET run-time and the second better MET memory consumption.

Figure 7.11 presents results for the `Adult` data set (D17). As we see here, as the number of modes increases, the running times of all decomposition algorithms increase. Experiment results confirm that the increase in the execution time is much slower for the DBN based decompositions and, as expected, the parallelized version of DBN (pp-DBN) is the fastest among all alternatives. The results with the `IPUMS` data set (D18), reported in Figure 7.12 re-confirm these results. Note that, the time results for this `IPUM` data set are presented in *log-scale* due to the significant differences in execution times between DBN-based and conventional decomposition strategies.

DBN-MET vs. Conventional MET – Impact of the Rank

In Figure 7.13, we compare the performance of DBN against conventional MET for different target ranks. Again, since there are different MET implementations, we present results for the strategy that provides the best running time for MET.

As we see in the figure, as the target rank increases, the running time of the conventional MET algorithm increases very quickly. In contrast, the running times of DBN-based strategies increase much more slowly. Again, the parallelized version of DBN (pp-DBN) is the fastest among all alternatives. Note that, as expected, the memory consumption of pp-DBN is higher than DBN-MET; however, it is still at least an order lesser than MET.

7.6.9 Accuracy Results

So far, we have shown that DBN-based strategies are significantly more efficient than their conventional counterparts. In this subsection, we experimentally assess the accuracy of DBN-based strategies.

Impacts of the IntraFD-based Rank Pruning and InterFD-based Partitioning on Accuracy

Before we compare DBN-based strategies against conventional decompositions, we first study the impacts of the interFD-based partitioning (Section 7.3) and intraFD-based rank pruning (Section 7.4) strategies on accuracy. These results are presented in Figure 7.14(a) and (b), respectively, where we compare the fit values obtained when using the proposed strategies against the maximum potential fit values one can obtain using a DBN-based strategy. In these plots, the closer to the 45 degree line the results are, the more effective are the FD based rank pruning and data partitioning strategies.

Table 7.12: Correlation between fits of DBN vs. NNCP and MET in original and outlier-eliminated (85% cumulative value preserved) CP and Tucker decomposition for all partitioning cases of D1-D15

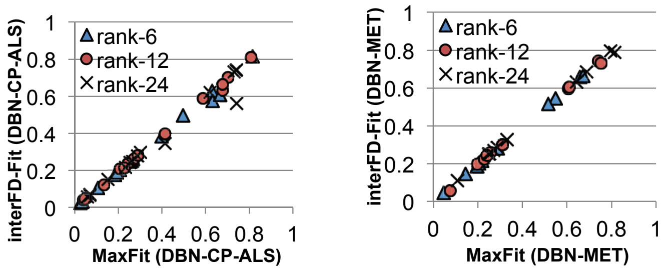
		Correlation	
		Original	85%
DBN-CP-ALS vs. NNCP-CP-ALS	rank-6	0.94	0.97
	rank-24	0.93	0.96
DBN-MET vs. MET	join mode rank: 6	0.99	0.99
	join mode rank: 24	0.99	0.99

In Figure 7.14(a), we study the impact of the interFD-based partition selection approach on the decomposition accuracy. The results show that, for both CP and Tucker decompositions, the interFD-based partitioning strategy results in accuracies that are very close to the maximum possible accuracy, using an optimal partitioning strategy for all rank configuration.

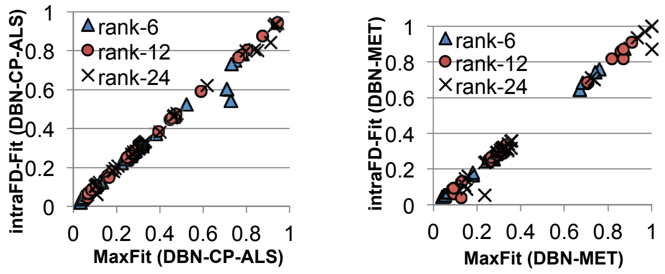
In Figure 7.14(b), we investigate the impact of intraFD-based rank pruning (with only the best 50% of the rank pairs considered by the JBD module among the potential rank pairs). Since the intraFD strategy ignores pairwise FDs involving the join attribute, we consider only the situations where each sub-tensor has more than 2 attributes (the join attribute and a determined attribute). As we see in Figure 7.14(b), the intraFD-based rank pruning strategy is very effective: except in a very few cases, the intraFD-based rank pruning does not eliminate the rank pair that will lead to the maximum possible fit with a DBN strategy.

DBN vs. Conventional Decompositions

We next evaluate the accuracy of DBN based decompositions against conventional decomposition algorithms, NNCP-CP-ALS for CP decomposition (since results for DBN-NWAY and NNCP-NWAY are similar we only present NNCP-CP-ALS) and MET for Tucker decomposition. We report accuracy results for data sets D1-D15 since



(a) interFD-Fit vs. MaxFit

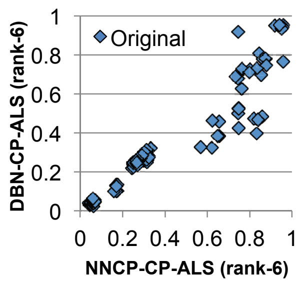


(b) intraFD-Fit vs. MaxFit

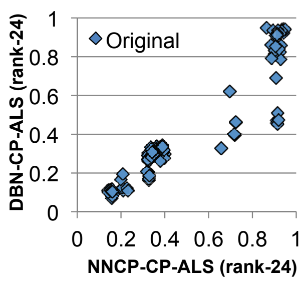
Figure 7.14: (a) InterFD-based fit vs. maximum fit and (b) intraFD-based fit vs. maximum fit of DBN-CP-ALS and DBN-MET for D1-D15 (we omit D13 and D15 for Tucker decomposition as the sizes of the join mode in the data sets are smaller than rank 24)

reconstructing the decomposed tensor needed for computing the fit value on larger data sets is not feasible with the available resources. Figures 7.15 and 7.16 present the accuracy results of DBN vs. NNCP and MET for CP and Tucker decompositions for different data sets, respectively. We also present the correlations between accuracies of DBN and conventional decomposition strategies in Table 7.12.

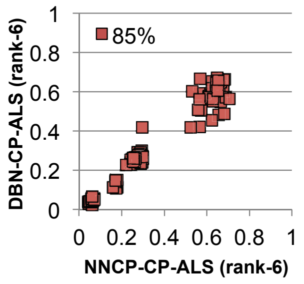
Figures 7.15(a), (b) and Table 7.12 shows that the accuracy of DBN is highly correlated with the accuracy of NNCP. There are, however, cases in which DBN has lower accuracies than NNCP. In order to understand whether this is a fundamental limitation of the DBN strategy or whether it is due to simple outliers in the data, we next consider whether the problem also occurs in the cases where the decomposition results are sparcified by ignoring the outliers: for this purpose we leverage a commonly used decomposition sparcification strategy [7]: treating each core element



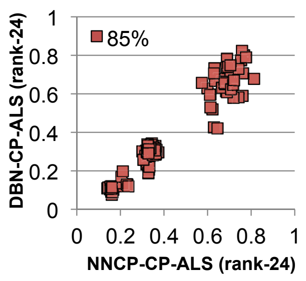
(a) original (rank: 6)



(b) original (rank: 24)



(c) 85% (rank: 6)

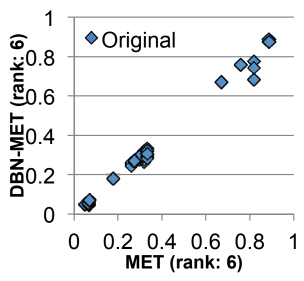


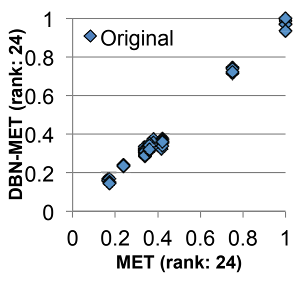
(d) 85% (rank: 24)

Figure 7.15: Accuracies of DBN vs. NNCP for original and outlier-eliminated (85% cumulative value preserved) CP decompositions for all partitioning cases of D1-D15

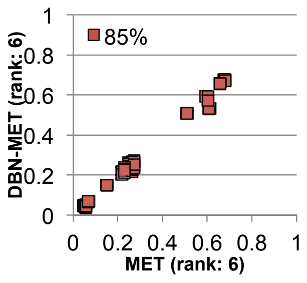
as a cluster and each factor entry as a cluster membership probability, we eliminate those elements that have very small likelihood of being a member of a given cluster. In particular, we remove sufficient outlier elements to eliminate the lowest 15% of the membership probabilities. Figures 7.15(c), (d) and Table 7.12 show that once the outliers are removed from consideration, DBN-based strategies perform as good as the conventional CP decomposition strategies. In fact, once the outliers are ignored in the decomposition, in a significant portion of the cases, the DBN strategy results in higher accuracies than the conventional DBN (indicated by an increase in the number of results above the 45 degree line).

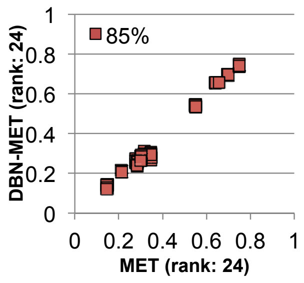
For Tucker decomposition, the correlations between the fits of DBN and MET are very high (almost 1.0) for both original and outlier-eliminated tensors (see Figure 7.16 and Table 7.12). These results confirm that the proposed DBN scheme is especially





(a) original (rank: 6) (b) original (rank: 24)





(c) 85% (rank: 6) (d) 85% (rank: 24)

Figure 7.16: Accuracies of DBN vs. MET in original and outlier-eliminated (85% cumulative value preserved) Tucker decomposition for all partitioning cases of D1-D15 (we omit D13 and D15 as the sizes of the join mode in the data sets are smaller than rank 24)

effective for Tucker decompositions.

Chapter 8

IN-DATABASE TENSOR DECOMPOSITION OPERATIONS AND OPTIMIZATION STRATEGIES IN IN-DATABASE TENSORDB

Tensor decomposition techniques, such as CP decomposition, are commonly used for discovering underlying structures (e.g. clusters) of multidimensional data sets. However, as the relevant data sets get large, existing in-memory schemes for tensor decomposition become increasingly ineffective and, instead, memory-independent solutions, such as in-database analytics, are necessitated. In this chapter, we present techniques for efficient implementations of in-database tensor decompositions on chunk-based array data stores. The proposed static and incremental in-database tensor decomposition operators and their optimizations address the constraints imposed by the main memory limitations when handling large and high-order tensor data. Firstly, we discuss how to implement alternating least squares operations efficiently on a chunk-based data storage system. Secondly, we consider scenarios with frequent data updates and show that compressed matrix multiplication techniques can be effective in reducing the incremental tensor decomposition maintenance costs. To the best of our knowledge, this thesis presents the first attempt to develop efficient and optimized in-database tensor decomposition operations. We evaluate the proposed algorithms using tensor data sets that do not fit into the available memory and results show that the proposed techniques significantly improve the scalability of this core data analysis technique.

8.1 Introduction

As the today's data sets get large, the in-memory based schemes for tensor decomposition become increasingly ineffective. A key difficulty in tensor decomposition is that the operation 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* [39, 45] and renders purely in-memory implementations of tensor-decomposition difficult. While today MATLAB-based in-memory linear algebra operations are widely used for implementing tensor decomposition algorithms [45, 47], these implementations are limited with the amount of memory available to the MATLAB software. Moreover, exporting data from a large database to import into MATLAB is often costly and elimination of this overhead can provide performance gains of several orders of magnitude [23].

8.1.1 In-Database Tensor Decompositions: Opportunities and Challenges

Because of the above limitations of in-memory solutions, *we consider in-database implementations of tensor decomposition operations on disk-resident data sets.* In particular, we argue that the ability to implement tensor decomposition operations on disk-resident tensor data can eliminate the challenge posed by the memory-limitations. However, we also recognize that in-database tensor analytics brings its own challenges

- *Challenge 1:* Tensor decomposition algorithms tend to involve computationally expensive operations (such as matrix multiplication) and require significant amounts of data movement, which (when data is stored on secondary storage) may result in high I/O load.

- *Challenge 2:* Many operations involved in tensor decomposition are order sensitive and the way data is laid on disk may have a big impact on the total cost of tensor decomposition task.

In this chapter, we attempt to address these challenges. In particular, we consider an array model [1] representation of the tensor data, leverage a chunk-based framework to store and retrieve data, extend array operations to tensor operations, and introduce optimization schemes for efficient in-database tensor decompositions.

8.1.2 Frequent Data Updates

In order to avoid the cost of decomposing the data tensor from scratch with each update when the data is frequently updated, we consider incremental tensor decompositions such as Dynamic Tensor Analysis (DTA) [70]. Despite the cost savings they provide, however, these incremental tensor decomposition techniques still suffer from high memory overheads. In this chapter, we show that the cost of this operation can be significantly reduced by leveraging recently introduced compressed matrix multiplication techniques, such as [57], instead of using traditional matrix multiplication implementations. In particular, we show that the proposed chunk-based operations, complemented with compressive matrix multiplication, can be highly effective in reducing the incremental tensor decomposition maintenance costs.

8.1.3 Contributions

To the best of our knowledge, this thesis is the first to study in-database tensor decomposition on a chunk-store. The proposed *static* and *incremental* in-database tensor decomposition techniques and the optimizations in the in-database TensorDB and address the memory limitations when handling large and high-order tensor data.

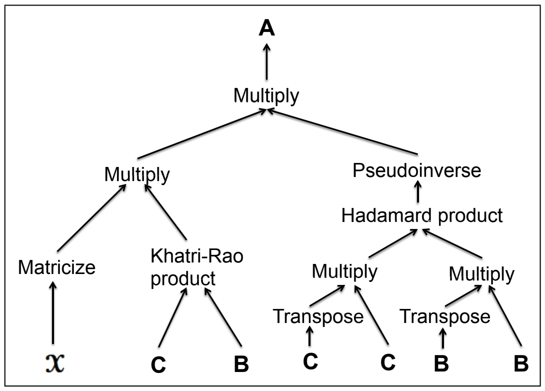
[1]We extend SciDB [18], an open source array-based DBMS.

Table 8.1: Notations used in this chapter

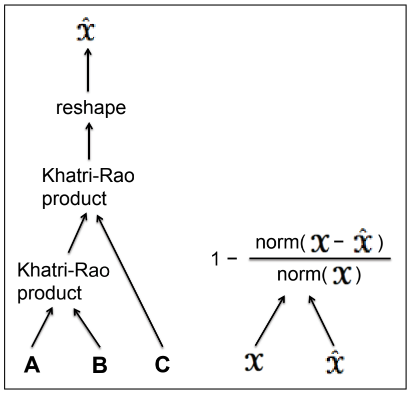
Notation	Description
$\mathcal{X}$	a tensor
r	rank (number of target component)
$\mathbf{A}$	a matrix
$\mathbf{a}_i$	ith column vector of a matrix $\mathbf{A}$
a_{ij}	(i, j)-th element of a matrix $\mathbf{A}$
$I_1 \times I_2 \times \cdots \times I_N$	N-mode tensor size
$K_1 \times K_2 \times \cdots \times K_N$	N-mode chunk size
$i_1, i_2, ..., i_N$	an element index of a N-mode tensor
$c_1, c_2, ..., c_N$	a chunk index of a N-mode tensor
$\mathbf{X}_{(m)}$	mode-m matricization
$X_{ij_{(m)}}$	(i, j)-th chunk of mode-m matricization $\mathbf{X}_{(m)}$
A_{ij}	(i, j)-th chunk of a matrix $\mathbf{A}$
$\mathbf{U}^{(d)}$	factor matrix of mode-d
$\mathbf{M}^{\dagger}$	Moore-Penrose pseudoinverse of a matrix $\mathbf{M}$
$\circ$	outer product
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product
$*$	Hadamard product
$\|\mathcal{X}\|$	Frobenius norm of a tensor $\mathcal{X}$

The chapter is organized as follows:

- We first provide an overview of the proposed in-database tensor operations (Section 4.3.2). We discuss the implementation and optimizations for various (chunk-based and otherwise) core operations involved in in-database tensor decomposition.
- We next focus on data updates and, in Section 8.2, we discuss efficient implementation of in-database dynamic tensor analysis operations through compressive matrix multiplication.
- We experimentally evaluate the static and dynamic in-database tensor decomposition operators and their optimizations in Section 8.3.



(a) $\mathbf{A} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\dagger}$



(b) fit computation

Figure 8.1: Execution plans for the different steps involved in the in-database CP decomposition of a tensor $\mathcal{X}$

8.1.4 Overview of Chunk-based CP

We consider an alternating least squares (ALS) based implementation of CP decomposition [19, 34]. Let us consider a 3-mode tensor $\mathcal{X}$ (Figure 2.1(a)). CP decomposition involves finding three factor matrices, such that

$$\min_{\hat{\mathcal{X}}} \|\mathcal{X} - \hat{\mathcal{X}}\| \quad \text{with} \quad \hat{\mathcal{X}} = \sum_{k=1}^{r} \mathbf{a}_k \circ \mathbf{b}_k \circ \mathbf{c}_k, \tag{8.1}$$

where $\mathbf{a}_k$, $\mathbf{b}_k$, and $\mathbf{c}_k$ are the kth column vectors of the factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively. This optimization problem can be solved through an alternating least squares process (Figure 8.2), where at each step all but one of the factor matrices are fixed and the remaining factor matrix is updated using least square estimation:

• *Initialize factor matrices:* Firstly, we create a factor matrix $\mathbf{U}^{(d)}$ for each mode d and initialize these with random data. Tensors and matrices are represented as multidimensional arrays in in-database CP and created by a SciDB operation, `create array`. Random data matrices are initialized by `build` operation. In our in-database CP implementation, all factor matrices are updated in an iterative manner by a new `copyArray` operator, described in Section 8.1.5. This operator performs in place array updates, thus as shown in the experiment section, significantly reduces I/O costs.

```
Alternating Least Squares for in-database CP algorithm (input: a tensor $\mathcal{X} \in I_1 \times I_2 \times \cdots \times I_N$ with the chunk size
$K_1 \times K_2 \times \cdots \times K_N$, a target rank $r$)
1: for each mode $d = 1, \ldots, N$ do
2:     create array $\mathbf{U}^{(d)}$ <val:double>[i=1:$r$,$r$,0,j=1:$I_d$,$K_d$,0]
3:     copyArray(build(<val:double>[i=1:$r$,$r$,0,j=1:$I_d$,$K_d$,0],random()),$\mathbf{U}^{(d)}$) {initialize a factor matrix $\mathbf{U}^{(d)}$
       randomly}
4: end for
5: repeat
6:     for each mode $d = 1, \ldots, N$ do
7:         copyArray(multiply(multiply(matricize($\mathcal{X}$,d),khatrirao(...,khatrirao(khatrirao($\mathbf{U}^{(1)}$,$\mathbf{U}^{(2)}$),...,),
           $\mathbf{U}^{(d-1)}$),$\mathbf{U}^{(d+1)}$),...,$\mathbf{U}^{(N)}$)), pseudoinverse(hadamard(...,hadamard(multiply(transpose($\mathbf{U}^{(1)}$),
           $\mathbf{U}^{(1)}$),multiply(transpose($\mathbf{U}^{(2)}$),$\mathbf{U}^{(2)}$)),..., multiply(transpose($\mathbf{U}^{(d-1)}$),$\mathbf{U}^{(d-1)}$)),
           multiply(transpose($\mathbf{U}^{(d+1)}$),$\mathbf{U}^{(d+1)}$)),..., multiply(transpose($\mathbf{U}^{(N)}$),$\mathbf{U}^{(N)}$)))),$\mathbf{U}^{(d)}$)
           {$\mathbf{U}^{(d)} = \mathcal{X}_{(d)}(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(d+1)} \odot \mathbf{U}^{(d-1)} \odot \cdots \odot \mathbf{U}^{(1)})(\mathbf{U}^{(1)T}\mathbf{U}^{(1)} * \cdots * \mathbf{U}^{(d-1)T}\mathbf{U}^{(d-1)} * \mathbf{U}^{(d+1)T}\mathbf{U}^{(d+1)} *$
           $\cdots * \mathbf{U}^{(N)T}\mathbf{U}^{(N)})^{\dagger}$}
8:         normalize($\mathbf{U}^{(d)}$) {Normalize columns of $\mathbf{U}^{(d)}$ (storing norms in vector $\lambda$)}
9:     end for
10: until fit converges or maximum iterations are exhausted
11: return $\lambda$, factor matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}$
```

Figure 8.2: Alternating Least Squares for in-database CP algorithm

- *(Iteratively) solve for factor matrices:* Next, one mode at a time, we iteratively solve for each factor matrix $\mathbf{U}^{(d)}$ (of mode d) given $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(d-1)}, \mathbf{U}^{(d+1)}, \ldots, \mathbf{U}^{(N)}$. For example, for a 3-mode tensor $\mathcal{X}$ and factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, solving for a factor matrix $\mathbf{A}$ can be formulated as

$$\min_{\mathbf{A}} \|\mathbf{X}_{(1)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T\|, \tag{8.2}$$

where $\mathbf{X}_{(1)}$ is mode-1 matricization of $\mathcal{X}$ and $\odot$ denotes a Khatri-Rao product. The optimal solution for Equation 8.2 can be formulated as

$$\mathbf{A} = \mathbf{X}_{(1)}[(\mathbf{C} \odot \mathbf{B})^T]^{\dagger} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\dagger}. \tag{8.3}$$

Here $\mathbf{M}^{\dagger}$ is the Moore-Penrose pseudo inverse of $\mathbf{M}$. Figure 8.1(a) shows the query plan for Equation 8.3.

- *Evaluate fit (after each factor matrix computation):* After obtaining a factor matrix, the columns of the factor matrix are normalized and each norm is stored in the core λ. Solving each factor matrix continues repeatedly until a measure of *fit* (defined as

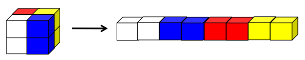
$$\mathrm{fit}(\mathcal{X}, \hat{\mathcal{X}}) = 1 - \frac{\|\mathcal{X} - \hat{\mathcal{X}}\|}{\|\mathcal{X}\|}, \tag{8.4}$$

where $\hat{\mathcal{X}}$ is the approximate reconstruction of the $\mathcal{X}$ tensor from the current decomposition and $\|\mathcal{Y}\|$ is the Frobenius norm of a tensor $\mathcal{Y}$) converges or a target maximum number of iterations are exhausted. The reconstruction of $\hat{\mathcal{X}}$ is done through a series of `Khatri-Rao` products and a `reshape` operation as shown in Figure 8.1(b).

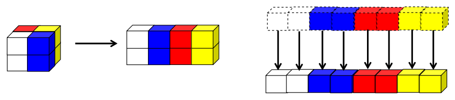
Figure 8.2 shows the outline of the steps involved in implementing the CP decomposition in an array database. Note that while some of the operations involved in the process (such as *multiply*) are already implemented in SciDB [18] and other array databases, most of the operations needed to implement Equation 8.3 are not available in common array databases. Furthermore, as we discuss next, even those existing array operations may require new implementations, more suitable for implementing tensor decomposition operations. In the next subsections, we will discuss chunk-based implementations of the various operations involved in the process and the proposed optimizations.

8.1.5 Chunk-based Tensor Operators

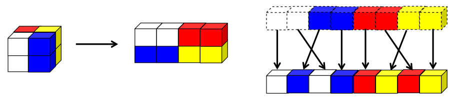
In this subsection, we introduce the novel chunk-based tensor operators (*matricization, Khatri-Rao product, Hadamard product, normalization*, and *copyArray* operators) needed for implementing in-database tensor decompositions. Each of these leverages the chunk ordering and access mechanism in Figure 4.6. In what follows, we refer to an array with two modes as `matrix` and to an array with more than two modes as `tensor`.



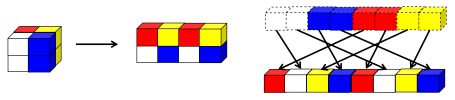
(a) Physical layout



(b) Mode-1 matricization



(c) Mode-2 matricization



(d) Mode-3 matricization

Figure 8.3: (a) Physical layout of a 3-mode input tensor in the physical memory; (b-d) different matricizations involve different amount of data movement

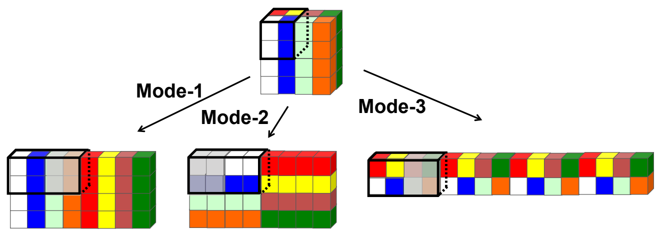
```
matricize(tensor, m)
```

Matricization transforms a tensor into a matrix along the given mode, m. More specifically, an element $(i_1, i_2, ..., i_m, ..., i_N)$ of tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times, \cdots, \times I_N}$ is mapped to (i_m, j) of mode-m matricization, $\mathbf{X}_{(m)}$, such that (assuming row-major representation of the result)

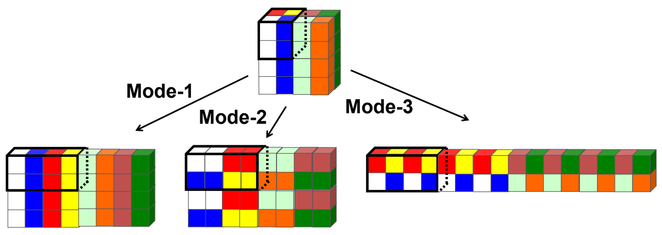
$$j = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{n=k+1 \\ n \neq m}}^{N} I_n \right) i_k.$$

Note that a tensor can be matricized using different column orderings [2] and, as shown in Figure 8.3, depending on how the data is physically laid out, different matricizations may involve different amounts of data movements. Therefore, our goal is to reduce this data movement.

[2]But the same order should be used in all related calculations [46]. In our work, the data ordering is aligned with the ordering of the result of Khatri-Rao product (see Equation 8.3).



(a)



(b)

Figure 8.4: (a) Matricization on each mode of a chunk of a 3-mode tensor: gray-shaded cells of the result matrices cannot be retrieved from the same chunk; (b) proposed chunk-based matricization

Impact of Data Ordering on Chunk-based Matricizations. One straightforward way to implement matricization would be to use SciDB's `reshape` and `redimension` operators as illustrated in the following example.

Example 8.1.1 *Consider a 3-mode tensor of size 100×100×100. We can implement mode-1 matricization of this tensor, assuming chunk sizes of 100×100, using the* `reshape` *operator, as follows:*

```
reshape(tensor, <val:double>
[i=1:100,100,0,j=1:10000,100,0]).
```

Mode-2 and mode-3 matricizations of the tensor, on the other hand, can be implemented by first re-arranging the dimensions using the `redimension` *operator,*

```
redimension(tensor,<val:double>
[j=1:100,100,0,i=1:100,100,0,k=1:100,100,0]),
```

followed by the `reshape` *operator as above.*

The problem with these straightforward implementations is that (as we also see in the experimental evaluations section), in the presence of chunk-based storage, these SciDB operators result in significant amounts of data traffic. Figure 8.4(a) visualizes matricizations of a 3-mode tensor using the `reshape` and `redimension` operations. As shown in the figure, when accessing data one chunk at a time, matricizations using `reshape` and `redimension` require repeated chunk-swapping in and out of memory to construct the output chunks, resulting in significant I/O overheads. Even in cases where multiple chunks can be stored in the buffer, the movement of data across chunks is costly. We address this problem by introducing a new chunk-optimized matricization operator.

Chunk-Optimized Matricization. We implement chunk-based mode-m matricization $\mathbf{X}_{(m)}$ of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times, \cdots, \times I_N}$ with chunks of size $K_1 \times K_2 \times, \cdots, \times K_N$ as

$$\mathbf{X}_{(m)} = \left(\begin{array}{c|c|c|c} X_{11_{(m)}} & X_{12_{(m)}} & \cdots & X_{1J_{(m)}} \\ \hline X_{21_{(m)}} & X_{22_{(m)}} & \cdots & X_{2J_{(m)}} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline X_{I_m 1_{(m)}} & X_{I_m 2_{(m)}} & \cdots & X_{I_m J_{(m)}} \end{array} \right),$$

where $X_{ij_{(m)}}$ is the (i,j)-th chunk of $\mathbf{X}_{(m)}$ and $J = I_1 \times \cdots \times I_{m-1} \times I_{m+1} \times \cdots \times I_N$.

An element of $(i_1, i_2, \ldots, i_m, \ldots, i_N)$ in a chunk of $(c_1, c_2, \ldots, c_m, \ldots, c_N)$ of $\mathcal{X}$ is mapped to an element of (i_m, j) in a chunk of (c_m, d) of $\mathbf{X}_{(m)}$, such that

$$j = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{l=k+1 \\ l \neq m}}^{N} K_l \right) i_k \text{ and } d = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{l=k+1 \\ l \neq m}}^{N} \lceil I_l / K_l \rceil \right) c_k. \tag{8.5}$$

As Figure 8.4(b) illustrates, the proposed chunk-based matricization process does not require repeated chunk-swaps to fill in the result chunks. Furthermore, since the data movement is constrained within individual chunks, the global order in which chunks are considered does not impact performance.

Materialization of the Results of the Matricization Operation. Tensor matricization is a costly operation, requiring at least one full read-and-write of the tensor data. Moreover, in CP decomposition, matricization of all modes of tensor $\mathcal{X}$ are needed in each iteration (see Equation 8.3). Therefore, one way to minimize the overall matricization overhead, is to *materialize the matricization results*: more specifically, once a matricization is computed, the result can be materialized on disk and this materialized matricization can be is used in all subsequent iterations. While materialization of the matricization results introduces additional I/O costs and storage requirements, especially in cases where the number of modes and number of iterations are large, materialization can bring significant savings.

`Khatri-Rao(left_matrix, right_matrix)`

Given a left matrix, $\mathbf{A} \in \mathbb{R}^{I \times K}$, and a right matrix, $\mathbf{B} \in \mathbb{R}^{J \times K}$, their Khatri-Rao product is denoted by $\mathbf{A} \odot \mathbf{B}$. The result is a matrix of size $(IJ) \times K$, defined as

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1 \; \mathbf{a}_2 \otimes \mathbf{b}_2 \cdots \mathbf{a}_n \otimes \mathbf{b}_n \cdots \mathbf{a}_K \otimes \mathbf{b}_K],$$

where $\mathbf{a}_n$ and $\mathbf{b}_n$ are columns of $\mathbf{A}$ and $\mathbf{B}$, respectively and $\otimes$ is the Kronecker product. Note that the Kronecker product, $\mathbf{U} \otimes \mathbf{V}$, of matrices $\mathbf{U} \in \mathbb{R}^{x \times y}$ and $\mathbf{V} \in \mathbb{R}^{w \times z}$ results in matrix of size $(xw) \times (yz)$, where

$$\mathbf{U} \otimes \mathbf{V} = \begin{pmatrix} u_{11}\mathbf{V} & u_{12}\mathbf{V} & \cdots & u_{1y}\mathbf{V} \\ u_{21}\mathbf{V} & u_{22}\mathbf{V} & \cdots & u_{2y}\mathbf{V} \\ \vdots & \vdots & \ddots & \vdots \\ u_{x1}\mathbf{V} & u_{x2}\mathbf{V} & \cdots & u_{xy}\mathbf{V} \end{pmatrix}.$$

Khatri-Rao products of factor matrices generate tall and generally dense matrices, which often do not fit into main memory. This is a well-known bottleneck in CP decompositions. [3] The proposed chunk-based Khatri-Rao product addresses this problem by dividing the resulting matrix into small enough chunks. We define the chunk-based Khatri-Rao product for matrices $\mathbf{A}$ (with chunks $A_{11}, ..., A_{IJ}$) and $\mathbf{B}$ (with chunks $B_{11}, ..., B_{IJ}$), as follows:

$$\mathbf{A} \odot \mathbf{B} = \left(\begin{array}{c|c|c|c} A_{11} \odot B_{11} & A_{12} \odot B_{12} & \cdots & A_{1J} \odot B_{1J} \\ \hline A_{21} \odot B_{21} & A_{22} \odot B_{22} & \cdots & A_{2J} \odot B_{2J} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline A_{I1} \odot B_{I1} & A_{I2} \odot B_{I2} & \cdots & A_{IJ} \odot B_{IJ} \end{array}\right).$$

Once again, since the data movement is constrained within individual chunks, the order in which chunks are considered does not impact performance.

`Hadamard(left_matrix, right_matrix)`

The Hadamard product is the elementwise matrix product; more specifically, given matrices $\mathbf{A}$ and $\mathbf{B}$, both of size $I \times J$, their Hadamard product, denoted as $\mathbf{A} * \mathbf{B}$, results in the following size $I \times J$ matrix:

$$\mathbf{A} * \mathbf{B} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{pmatrix}.$$

Given this, we define the chunk-based Hadamard product for matrices $\mathbf{A}$ with

[3] Matricization times Khatri-Rao product together can be formulated in alternative ways for sparse tensors [12, 39]. In this work, we consider the more general formulation also applicable to dense data.

chunks $A_{11}, ..., A_{IJ}$ and $\mathbf{B}$ with chunks $B_{11}, ..., B_{IJ}$ as follows:

$$\mathbf{A} * \mathbf{B} = \left(\begin{array}{c|c|c|c} A_{11} * B_{11} & A_{12} * B_{12} & \cdots & A_{1J} * B_{1J} \\ \hline A_{21} * B_{21} & A_{22} * B_{22} & \cdots & A_{2J} * B_{2J} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline A_{I1} * B_{I1} & A_{I2} * B_{I2} & \cdots & A_{IJ} * B_{IJ} \end{array} \right).$$

Again, the data movement is constrained within individual chunks and, thus, the order in which chunks are considered does not impact performance.

`copyArray(operator(args), array)`

This operator copies the result of `operator(args)` to a temporary array, `array`, and is used for updating the intermediate results (e.g., in the in-database CP decomposition, for updating the factor matrices, which get updated in each iteration). In contrast, SciDB's analogous operation, `store`, does not update an existing array but creates a new version of the array (also maintaining the previous versions). Also, unlike `store`, `copyArray` does not use run-length encoding/decoding, since frequently updated and relatively small factor matrices, do not benefit from run-length encoding/decoding.

Other Chunk-based Operators

In addition to the above, our in-database CP implementation also requires chunk-based implementations of other operators, such as operators for normalizing the columns of the input matrix, or computing the Frobenius norm of the difference between the given tensor and the decomposed tensor used for fit computation (see Equation 8.4).

8.1.6 Non-Chunked Tensor Operations

In addition to the above chunked operators, we also implement two non-chunked operators, *pseudoinverse* and *eigen-decomposition*. While these require their inputs to fit into the memory, since (during tensor decomposition) inputs are often relatively small matrices, this rarely constitutes a problem.

`pseudoinverse(matrix)`

This operator returns the pseudo-inverse of the input `matrix`. We implement this operator using a C++ linear algebra library from [67], where SVD is used to solve pseudo-inverse problem. Since during CP decomposition, the input to the pseudo-inverse operation is a matrix of size `rank` $\times$ `rank`, where `rank` is a relatively small number of target components, this matrix easily fits the main memory and does not require a chunk-based implementation.

`eigen(matrix, r)`

This operator returns `r` leading eigen-vectors of the input `matrix`. Similar to the `pseudoinverse` operator, eigen-decomposition is an in-memory operation and we use the eigen-decomposition function provided in [67] for implementation.

We use this eigen-decomposition operation to implement incremental tensor decomposition. In particular, we take the leading eigen-vectors of the $I_d \times I_d$ covariance matrix to generate factor matrices, where I_d is the size of the mode d of the tensor. Note that this matrix is often much smaller than the whole tensor and, thus, we assume that the covariance matrix fits into the main-memory. In cases where this does not hold, it is possible to leverage block decomposition techniques, such as incremental SVD [15] to implement this on chunks.

```
In-database DTA (input: a tensor $\mathcal{X} \in I_1 \times I_2 \times \cdots \times I_N$ with the chunk size $K_1 \times K_2 \times \cdots \times K_N$, a target rank
$r_1, r_2..., r_N$, old covariance matrices $\mathbf{C}^{(1)}_{old}, \mathbf{C}^{(2)}_{old}, ..., \mathbf{C}^{(N)}_{old}$)
 1: for each mode $d = 1, \ldots, N$ do
 2:     create array $\mathbf{C}^{(d)}$ <val:double>[i=1:$I_d$,$I_d$,0,j=1:$I_d$,$I_d$,0] {create a covariance matrix $\mathbf{C}^{(d)}$}
 3:     store(multiply(matricize($\mathcal{X}$, $d$), transpose(matricize($\mathcal{X}$, $d$)) {$\mathbf{C}^{(d)} = \mathbf{X}^{(d)}\mathbf{X}^{(d)T}$}
 4:     select a.val+b.val into $\mathbf{C}^{(d)}$ from $\mathbf{C}^{(d)}_{old}$ a, $\mathbf{C}^{(d)}$ b; {$\mathbf{C}^{(d)} = \mathbf{C}^{(d)}_{old} + \mathbf{C}^{(d)}$}
 5: end for
 6: for each mode $d = 1, \ldots, N$ do
 7:     create array $\mathbf{U}^{(d)}$ <val:double>[i=1:$I_d$,$K_d$,0,j=1:$r_d$,$r_d$,0] {create a factor matrix of the mode $d$, $\mathbf{U}^{(d)}$}
 8:     copyArray(eigen($\mathbf{C}^{(d)}$, $r_d$),$\mathbf{U}^{(d)}$) {set the $r_d$ leading eigen vectors to factor matrices $\mathbf{U}^{(d)}$}
 9: end for
10: Let $\mathbf{T}_1, \mathbf{T}_2 \ldots \mathbf{T}_N$ be temporary arrays to store intermediate results
11: store(reshape($\mathcal{X}$, <val:double>[i=$I_2 \times \cdots \times I_N$,$K_2 \times \cdots \times K_N$,0,j=1:$I_1, K_1$,0]),$\mathbf{T}_1$)
12: for each mode $d = 1, \ldots, N-1$ do
13:     store(reshape(multiply($\mathbf{T}_d$, transpose($\mathbf{U}^{(d)}$)), <val:double>[i=1:$I_1 \times \cdots \times I_{d-1} \times I_{d+1} \times \cdots \times I_N$,$K_1 \times$
        $\cdots \times K_{d-1} \times K_{d+1} \times \cdots \times K_N$,0,j=1:$I_d, K_d$,0]),$\mathbf{T}_{d+1}$)
14: end for{compute the core tensor $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}^{(1)T} \times_2 \cdots \times_N \mathbf{U}^{(N)T}$ and $\mathbf{T}_N$ is the core tensor}
15: $\mathcal{Y} = \mathbf{T}_N$
16: return core $\mathcal{Y}$, factor matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)} \ldots \mathbf{U}^{(N)}$
```

Figure 8.5: In-database DTA

8.2 Frequent Data Updates

As described in the introduction, when the data are frequently updated, techniques which incrementally maintain tensor decompositions tend to be more efficient than repeatedly decomposing the whole data tensor with each update.

8.2.1 Chunk-based Dynamic Tensor Analysis

In our work, we adapt the Dynamic Tensor Analysis (DTA) algorithm [70] for in-database operation. Note that, unlike CP, DTA assumes a dense core matrix as in Tucker decomposition [74]; but, as shown in [8], results of Tucker decompositions can be used as a first step towards bootstrapping CP decomposition.

DTA incrementally maintains covariance matrices for each mode and computes factor matrices by taking the leading eigen-vectors of the covariance matrices. More specifically,

- first, the algorithm computes the covariance matrix $\mathbf{C}^{(d)}$ for each mode d;
- then, each $\mathbf{C}^{(d)}$ is updated by adding the old covariance matrix $\mathbf{C}^{(d)}_{old}$;
- next, r_d leading eigen vectors of each covariance matrix are copied into corresponding factor matrix;
- finally, the core tensor is obtained by multiplying the input tensor with factor matrices along each mode.

Figure 8.5 provides the pseudo-code for in-database, chunk-based DTA, implemented using the operators described in the previous section. Note that in-database DTA benefits from chunk-based operators in reducing the I/O overhead when dealing with disk-resident, large-scale data. However, as we also experimentally establish in Section 8.3, a significant portion of the execution cost of the above algorithm is due to the step in which the covariance matrix, $\mathbf{C}^{(d)}$, for each mode, d, is computed. Therefore, a key challenge is to reduce the cost of this step. We discuss this next.

8.2.2 Chunk-based Covariance Matrix Estimation

As shown in Figure 8.5, the covariance matrix of a given tensor along a given mode, d, is computed by first matricizing the tensor along mode d and then multiplying the matricized tensor with its transpose. Both the matricization operation and the matrix multiplication can be implemented and optimized using chunk-based techniques (as discussed in the previous section) to reduce I/O costs. However, given two matrix chunks U_{ij} and V_{kl}, which are (i, j)-th chunk of a matrix $\mathbf{U}$ and (k, l)-th chunk of a matrix $\mathbf{V}$ respectively, (the first one from the matricized tensor and the second from its transpose) brought into the memory, computation of $U_{ij}V_{kl}$ is still a costly process. We propose to address this by performing, when appropriate, (approximate) compressed matrix multiplication, instead of using conventional multiplication operators.

Compressed Covariance Matrix Estimation

In general, for U_{ij} of size $n \times m$ and V_{kl} of size $m \times n$, the matrix product, $U_{ij}V_{kl}$, can be obtained as follows:

$$U_{ij}V_{kl} = \sum_{r=1}^{n} \mathbf{u}_r \mathbf{v}_r, \tag{8.6}$$

where $\mathbf{u}_1, ..., \mathbf{u}_n$ are the row vectors of U_{ij} and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are the column vectors of V_{kl}. Compressed matrix multiplication, on the other hand, is a recent technique which leverages compressive sensing to obtain an approximation of the matrix multiplication result, without performing n outer products explicitly [57]. While, the details of this algorithm is outside of the scope of this thesis, it is sufficient to note that the algorithm computes a linear count sketch [20] of the entries of each outer product of Equation 8.6. The algorithm has two key parameters, **b** and **d**: **b** regulates the detail of the count sketches obtained for each column vector of U_{ij} and row vector of V_{kl}; **d**, on the other hand, regulates the number of count sketches obtained to improve accuracy.

[57] showed that it is possible to approximate a matrix product with high probability *if the matrix product is compressible*, i.e., *if the Frobenius norm of the matrix product is dominated by a sparse subset of entries of the product.* We argue that this condition is often satisfied when computing covariance matrices, as for most data of interest, input matrices (i.e., matricized data tensors) have skewed distributions and, thus, the resulting covariance matrices tend to be sparse. Consequently, in most practical cases, approximate compressed matrix multiplication can be applied to obtain accurate estimates of covariance matrices. Most importantly, we can decide ahead of the time whether to use regular or compressed matrix multiplication, based on the

sparsity of the initial covariance matrix. On the other hand, as we see next, when considering chunk-based in-database implementations, various further optimizations need to be considered.

Chunk Density based Optimization

According to [57], given a parameter pair, **b** and **d** (which together govern the number of collected sketches and two square matrices), the cost of multiplying two matrices of sizes $n_1 \times n_2$ and $n_2 \times n_3$, is

$$O(\mathbf{d}n_2(n_1 + n_3 + \mathbf{b}\log\mathbf{b}) + n_1n_3).$$

Since, when estimating covariance matrices, we multiply chunks of size $n \times m$ with chunks of size $m \times n$ (of the transpose matrix), the cost of the operation for each chunk pair can be computed as

$$O(\mathbf{d}m(2n + \mathbf{b}\log\mathbf{b}) + n^2),$$

or equivalently as

$$O(\mathbf{d}mn(2 + \frac{\mathbf{b}\log\mathbf{b}}{n} + \frac{n}{\mathbf{d}m})). \tag{8.7}$$

Since, when chunks are dense, multiplying these two chunks using a straightforward matrix multiplication algorithm would cost $O(mn^2)$, as long as the inequality,

$$\mathbf{d}(2 + \frac{\mathbf{b}\log\mathbf{b}}{n} + \frac{n}{\mathbf{d}m}) < n,$$

is true, compressed matrix multiplication is likely to outperform exact matrix multiplication.

When the matrices that are multiplied are *sparse*, however, there are faster matrix multiplication algorithms [1]. Thus, as we experimentally show in Section 8.3, when the input chunks are sparse, compressed matrix multiplication may not provide significant time gains. Therefore, we utilize compressed matrix multiplication only for pairs of chunks that are both dense; we revert back to the default matrix multiplication algorithm if one of the chunks is sparse.

Optimization of the Chunk Shape

Chunks used in tensor decomposition are constrained by the amount of buffer available for storing them once they are read from the secondary storage into the main memory; on the other hand, it is possible to use chunk of different shapes, as long as the chunk size fits the allocated memory (i.e., $n \times m \sim \beta$ for some target buffer size, β). However, we see that the cost function (Equation 8.7) and the associated inequality, together provide additional constraints on n and m. These constraints can help determine the optimal shape of the chunk under a given buffer constraint.

In particular, the cost function implies that, for a given parameter pair, $\mathbf{b}$ and $\mathbf{d}$, the running time gets faster when n gets smaller than m, given $n \times m$ fixed. On the other hand, since when creating count sketches, the column (or row) vectors (of size n) need to be scanned sequentially, matrices with $n > m$ (when $n \times m$ is fixed) are likely to be scanned faster. Therefore, in practice, as we see in Section 8.3.3, the best execution times are observed when $m \sim n$.

8.3 Evaluation

In this section, we evaluate the proposed static and dynamic in-database, chunk-based tensor decomposition operators. We ran experiments on Ubuntu 12.04 64-bit, 7.7 GB RAM, Intel Core i5-2400 CPU @ 3.10GHz $\times$ 4, and 112.6 GB disk. We implemented the proposed tensor manipulation operators by extending SciDB 12.12 [4]. In memory baselines are implemented using MATLAB tensor toolbox [12].

Table 8.2: Tensor data sets for in-database CP decomposition

Data set	**Attributes**	**Size**	**Chunk size**	**Density (%)**
Enron email data (Enron)	`(recipient's position, sender,recipient,time)`	8×184×184×5632	8×184×184×200	0.0022
MovieLens 1M (Movie)	`(rating,genre,movie,user)`	5×18×3400×6000	5×18×200×200	0.1
Extended Yale Face Database B (Face)	`(image_no,x-coord,y-coord)`	5000×480×640	250×240×320	100

8.3.1 Experimental Setup

Tensor Representations. We use tensors of different densities and different tensor representations: sparse tensor representation (shortly referred to as STR), where only non-zero entries are kept, and dense tensor representation (DTR). We consider tensors with different densities, and in each figure we highlight the tensor density along with the tensor representation utilized; e.g., STR:0.001% for sparse representation of a tensor of 0.001% density.

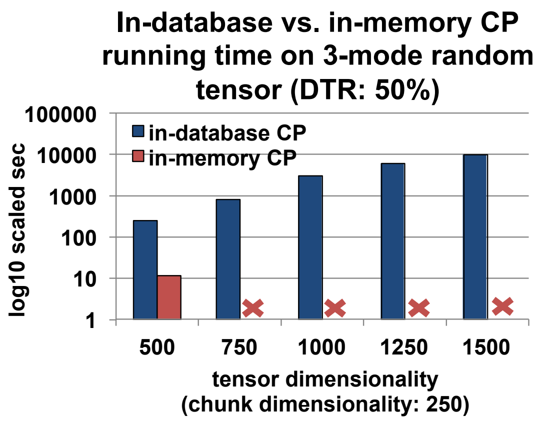
Evaluation Criteria. In addition to the execution times, our evaluation criteria also include *fit* (Equation 8.4) and *relative fit* (fit_{rel}), which indicates how accurate the proposed scheme is compared to the baseline in terms of *fit*:

$$fit_{rel} = \text{fit}(\mathcal{X}, \hat{\mathcal{X}}_{opt}) \ / \ \text{fit}(\mathcal{X}, \hat{\mathcal{X}}_{base}), \tag{8.8}$$

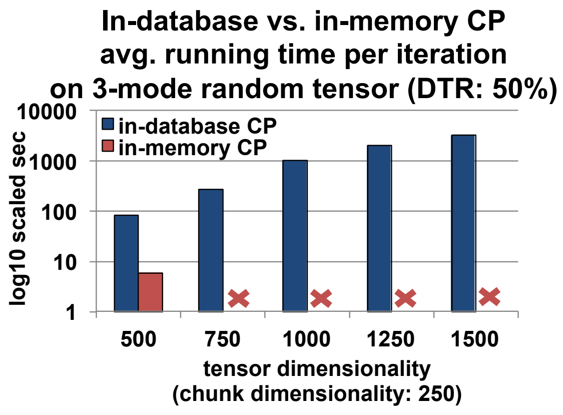
where $\mathcal{X}$ is the input tensor, $\hat{\mathcal{X}}_{base}$ is the tensor obtained by re-composing the decomposed tensor in the baseline scheme, and $\hat{\mathcal{X}}_{opt}$ is the tensor obtained by re-composing the decomposed tensor in the proposed scheme.

8.3.2 In-Database CP

Data Sets. In these experiments, in addition to random data sets, we also used real data sets with different characteristics: Enron email data set (Enron) [61], MovieLens 1M data sets (Movie) [56], and a face data set (Face) [5] with 5,000 images.



(a)



(b)

Figure 8.6: (a) Running times of in-memory and in-database CP decomposition and (b) average running times per iteration on 3-mode random tensors (DTR: 50%). Red 'x' marks show that in-memory CP runs out of memory

Table 8.3: Fit of in-database CP compared against the fit achieved by in-memory CP

	in-database CP	in-memory CP [12]†
Enron	0.0774	0.0671
Movie	0.0447	0.0447
Face	0.5511	Not Enough Memory

†In-memory CP on Face data set is not feasible

Table 8.2 shows the detail of each data set. In these experiments, we considered target decomposition rank of 10.

In-Database CP vs. In-Memory CP

Scalability. We first compare the running times of in-memory and in-database CP on 3-mode dense random tensors (DTR: 50%). Here we use the same chunk dimensionality (250) for all tensors. Figure 8.6 shows that, as expected, when the data fits into the memory, in-memory decomposition is faster than in-database operation; however, the proposed in-database decomposition operator is able to operate even in situations where the in-memory decomposition is not feasible.

Accuracy. We next evaluate the accuracy of the in-database CP decomposition on various real data sets, listed in Table 8.3 – two sparse tensors (in sparse tensor representation, STR) and a dense tensor (in dense tensor representation, DTR). The sparse tensors we considered are small enough to fit into the main memory to enable us compare the accuracy of the proposed in-database CP decomposition operator to MATLAB based in-memory decomposition, specially designed for sparse tensors [12]. As we see in this table, on the datasets where the in-memory decomposition was feasible, in-database CP decomposition achieves *equal to or better fit than* the in-memory CP decomposition.

Detailed Analysis of In-Database CP

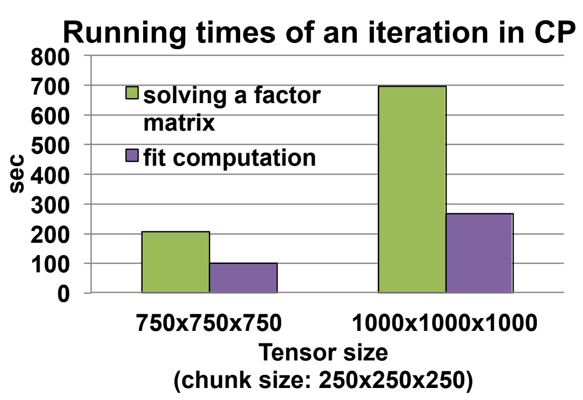
Next we study the cost of in-database CP in detail on random tensors where the computation does not fit into the memory.

Cost Breakdown of a Single Iteration. Figure 8.7(a) provides a high-level breakdown of a single iteration. As we see here, the cost of fit computation step in in-database CP is not negligible and thus the operations involved in this step also need to be carefully optimized. Figure 8.7(b) confirms that the `copyArray` operator introduced to efficiently update factor matrices (Section 8.1.5) provides significant savings relative to SciDB's `store` operator.

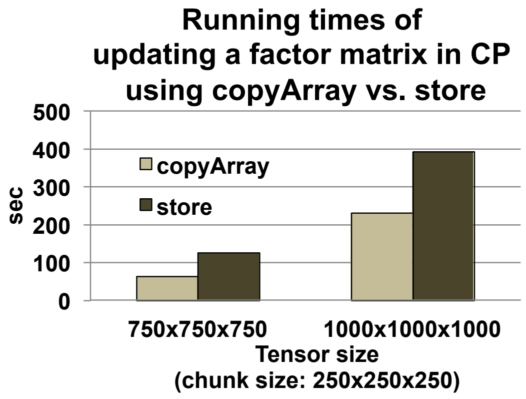
Cost Breakdown of Solving a Factor Matrix. Figure 8.7(c) focuses on the time needed to solve factor matrices and show that, on dense data sets [4] , matricization (which requires data re-ordering as discussed in Section 8.1.5) is the single costliest operation. Figure 8.7(d) further analyzes the running times of the remaining operations [5] (i.e., all except matricization) involved in solving a factor matrix: as

[4]Matricization is much cheaper on sparse data sets; results are omitted due to space limitations.

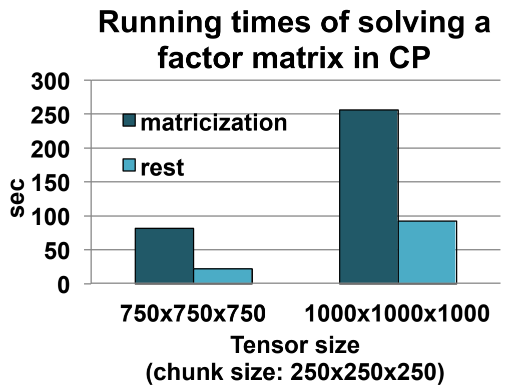
[5]While these are negligible for dense input data, for sparse data sets where matricization is fast, these operations, which always operate on dense factor matrices, even when the input tensor is sparse, will constitute the dominant cost.



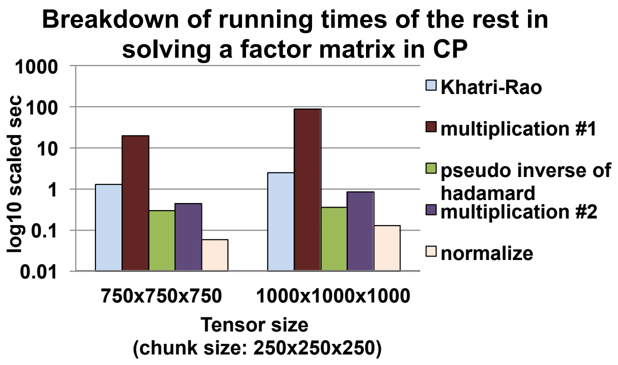
(a)



(b)



(c)

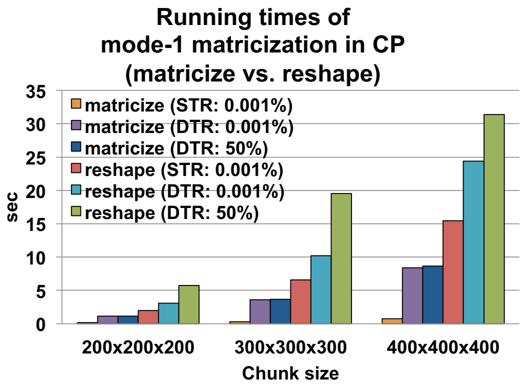


(d)

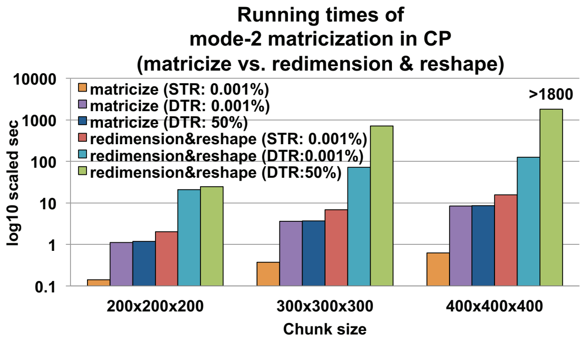
Figure 8.7: (a) Execution time breakdown of a single in-database CP iteration (input tensor – DTR:50%); (b) impact of the novel `copyArray` operator vs. `store` (c) for dense input tensors time needed for solving a factor matrix is dominated by matricization step; and (d) breakdown of the rest operations (except matricization) in solving a factor matrix (these operations are performed on DTR whether the input tensor is dense or sparse since factor matrices are generally dense

expected, multiplication of the result of matricization with the result of Khatri-Rao product consumes the largest amount of time among these steps.

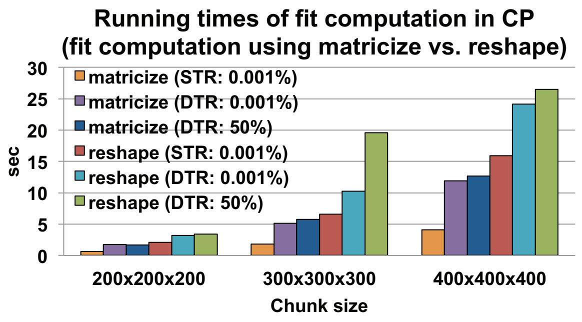
Impact of the `matricize` Operator. As we also discussed in Section 8.1.5, it is possible to implement matricization using SciDB's `redimension` and `reshape` operations, instead of the proposed special `matricize` operator. Figure 8.8 shows that the proposed `matricize` operator is significantly more efficient than SciDB's existing operators for all chunk densities and STR/DTR representations. Note that the



(a)



(b)



(c)

Figure 8.8: Running times of (a) mode-1 matricization using `matricize` vs. `reshape` operations, (b) mode-2 matricization that requires data re-ordering using `matricize` vs. `redimension` and `reshape` operations, and (c) fit computation using `matricize` vs. `reshape`

execution times reported in Figure 8.8(b) show the importance of using chunk-based `matricize` on modes that would otherwise necessitate the use of `redimension` operator: in this case, the savings in execution time through the use of chunk-based `matricize` are *multiple orders of magnitude.*

In addition to being useful when solving a factor matrix, as we have seen in Section 8.1.4, matricization is also useful while computing the degree of fit. Figure 8.8(c) shows that, also in this case, using the proposed `matricize` operator helps reduce the execution times.

Impact of Materialization of Matricization. To further reduce the cost of ma-

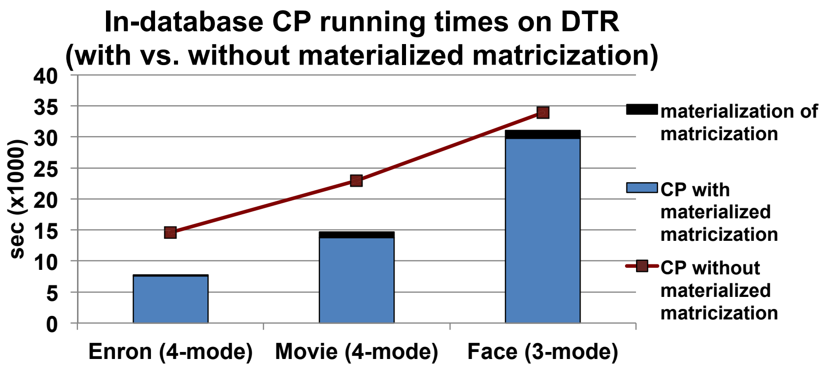


Figure 8.9: Running times of in-database CP with vs. without materialized matricization on real data (DTR)

Table 8.4: Tensor data sets for in-database DTA

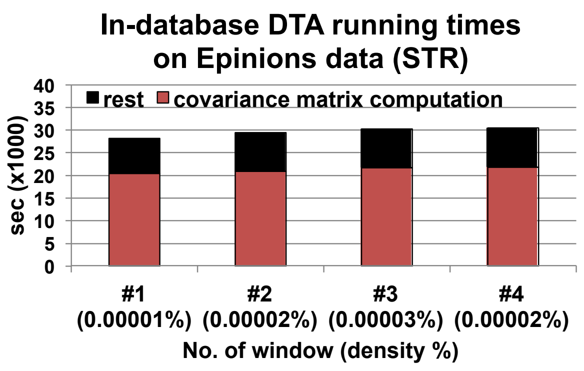
Data set	**No. of inputs**	**Attributes**	**Size**	**Chunk size**	**Avg density (%)**
Epinions	4 windows	`(user,product,category, rating,helpfulness)`	5000×5000×26×5×5	500×500×13×5×5	0.00002
MovieLens 10M	5 windows	`(user, movie, rating)`	5000×5000×10	500×500×10	0.28
Aerial view	6 image frames	`(x-coord, y-coord)`	2000×20004	2000×2000	100

tricization, we can also leverage materialization of the matricization results. As seen in Figure 8.9, the materialization of the matricization can help reduce the running time of in-database CP, especially on input tensors with higher number of modes and dense representations.

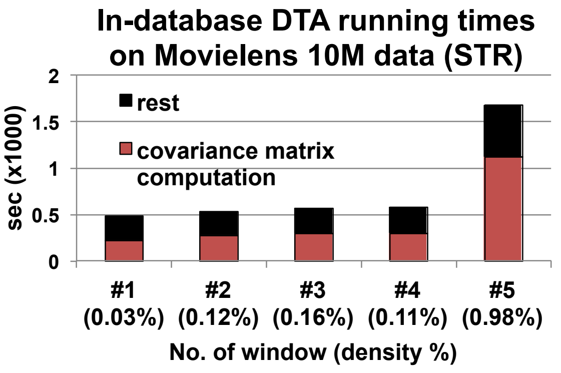
Note that the cost of materializing matricization gets amortized as the number of iterations increases. Note also that materialization of matricization requires additional storage on the hard disk (equal to the tensor size for each matricized mode). While this is often not an issue, when the storage is a concern, one can selectively materialize the matricization on a subset of modes.

8.3.3 In-Database Dynamic Tensor Analysis

We next present the experiment results for in-database dynamic tensor analysis (DTA) and in-database DTA with compressed matrix multiplication (C-DTA). For



(a)



(b)

Figure 8.10: Running times of in-database DTA of sparse data (STR) on (a) 4 windows of Epinions data and (b) 5 windows of MovieLens 10M data (note that in the last window, the tensor is much denser than the previous windows)

implementing the FFT process involved in the compressed matrix multiplication algorithm, we used a C subroutine library [32]. In these experiments, we set the ranks to 5 for each mode.

Data Sets. For these experiments, in addition to the randomly generated data, we used Epinions [72], MovieLens 10M [56], and Aerial views II [2] (Aerial view) data sets. For the Epinions data, we ran the in-database DTA for 4 windows on the input tensor of product ratings (user, product, category, rating, helpfulness). This tensor is of size $5000 \times 5000 \times 26 \times 5 \times 5$ (we considered 5000 frequent users and products). For the MovieLens 10M data, we used 5 windows of the movie rating data (movie, user, rating) on the input tensor of size $5000 \times 5000 \times 10$ (we considered 5000 frequent users and movies). For Epinions and MovieLens 10M data, each entry of the input tensors denotes whether the rating exists (1) or not (0) on the corresponding attributes in the window. For the Aerial view data, in-database DTA is performed on 6 gray-scale image frames (x-coord, y-coord) where each entry represents a gray-scale color (0-255). Table 8.4 shows the data sets.

Table 8.5: Average fit of in-database DTA compared against in-memory DTA on various data sets

	in-database DTA	in-memory DTA [70]†
Epinions	0.016	Not Enough Memory
MovieLens 10M	0.043	0.043
Aerial view	0.77	0.77

†In-memory DTA on Epinions data set is not feasible in available memory, while in-database DTA successfully completes the task.

In-Memory DTA vs. In-Database DTA

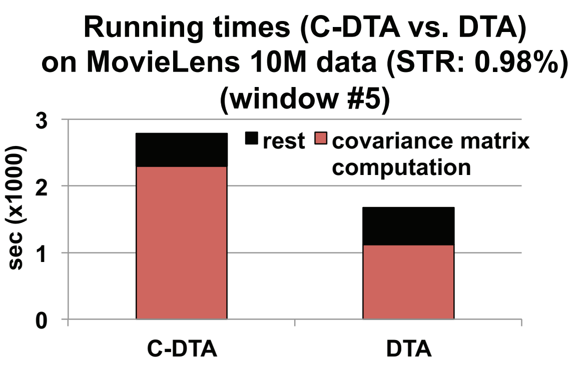
Firstly, Table 8.5 shows the accuracy (fit) results for in-database and in-memory DTA for various real data sets. As we see, the accuracy of the in-database DTA is the same as the accuracy of in-memory DTA. Moreover, in-database DTA is able to operate in cases (such as the Epinions data set) that are too large to run in the available memory.

Impacts of the Number of Modes and Data Density on DTA. We next evaluate the impact of the number of modes and data density on DTA, using Epinions data and MovieLens 10M data (Figure 8.10). As we see here, on sparse data, the number of tensor modes is a significant factor and 5-mode Epinions data requires much larger decomposition time than 3-mode MovieLens 10M data. As shown in Figure 8.10(b), running times get larger for denser data and the largest contributor to the execution time of DTA is the covariance matrix computation.

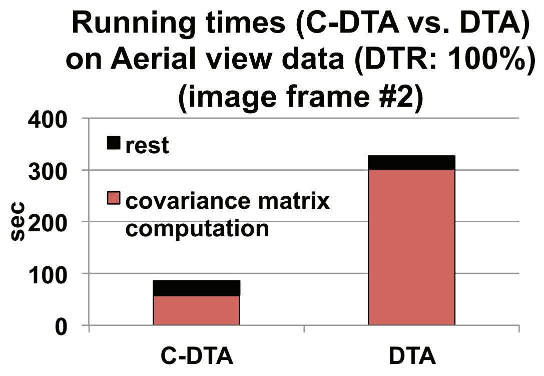
In-Database DTA vs. In-Database C-DTA

The default values of **b** is set to $n/2$ for $n \times n$ chunk of the covariance matrix and **d** is set to 30 (as explained later in this section).

Sparse vs. Dense Tensors. As we see in Figure 8.11, as expected, C-DTA is not advantageous for data with sparse representation (STR). On the other hand, for data with dense representation (DTR), C-DTA provides significant time gains.

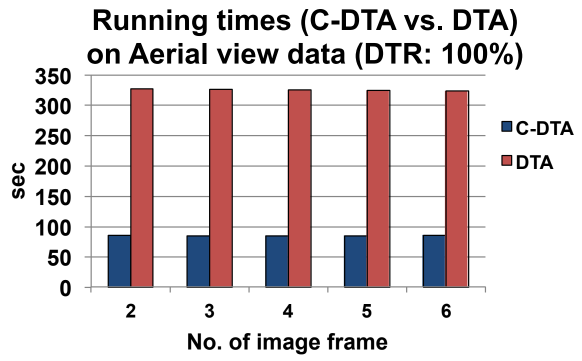


(a)

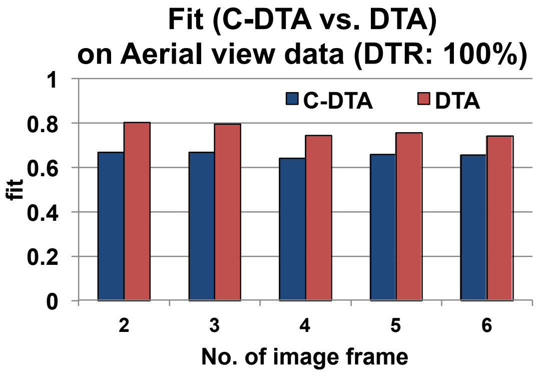


(b)

Figure 8.11: Running times of C-DTA vs. DTA on (a) a tensor with sparse tensor representation (STR) vs. (b) a tensor with dense tensor representation (DTR)



(a)

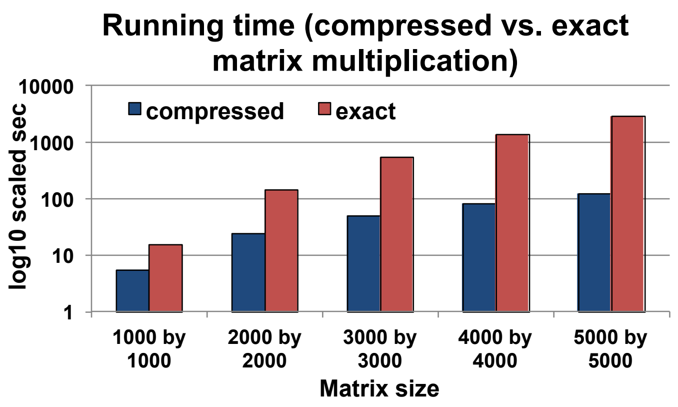


(b)

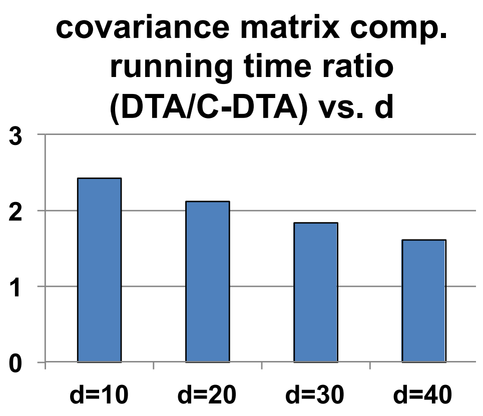
Figure 8.12: (a) Running times and (b) fits of incremental C-DTA vs. DTA on 5 image frames of Aerial view data

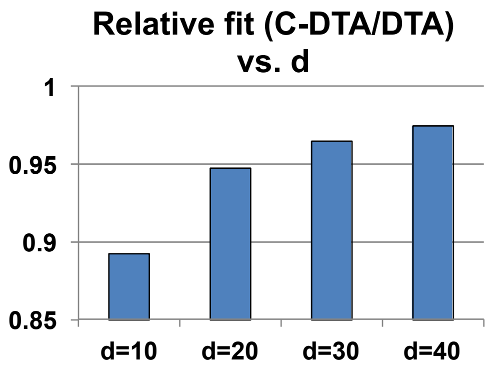
These confirm the observations reported in Section 8.2.2. Note that Figure 8.11 re-confirms that the running times of covariance matrix computation is the most dominant component in DTA and C-DTA.

Accuracy of C-DTA. Figure 8.12 presents the running times and fits of C-DTA vs. DTA on 5 consecutive image frames of the Aerial view data set. As we have already seen in Figure 8.11(b), on this data set, C-DTA consistently outperforms DTA ($\sim 3\times$) and, despite the significant drops in execution time, the fits of C-DTA are close to those of DTA ($\sim 80\%$ relative fit). Interestingly, while the fit of DTA



(a)





(b) (c)

Figure 8.13: (a) Running times of compressed vs. exact matrix multiplication on random matrices with varying sizes, (b) running time ratios of DTA vs. C-DTA (the higher the running time ratio, the more efficient is C-DTA) and (c) relative fit of C-DTA vs. DTA (the higher the relative fit, the more accurate is C-DTA) – in (b) and (c) we vary the parameter **d** (# of count sketches) in compressed matrix multiplication. In all cases, we use random tensors (DTR:50%)

drops as more update windows are considered, the degree of fit of C-DTA remains mostly consistent.

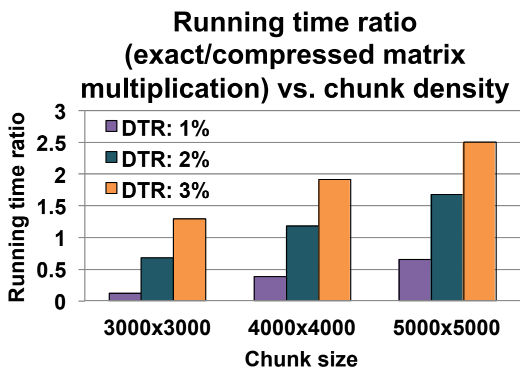
Analysis of Covariance Matrix Maintenance through Compressed Matrix Multiplication

As we have confirmed above, C-DTA is useful mainly for dense tensors. Thus, here, we focus on dense matrices.

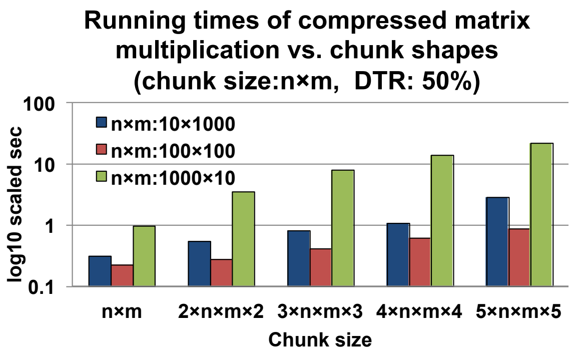
Scalability of Compressed Matrix Multiplication. We first compare the running time of compressed matrix multiplication with running time of exact matrix multiplication. As we have seen in Section 8.2.2, as the row length of the matrix increases, the time complexity of compressed matrix multiplication increases linearly, whereas the running time of the exact matrix multiplication increases quadratically. This is confirmed in the results presented in Figure 8.13(a).

Time/Accuracy Trade-Offs for Covariance Computation. Next, we evaluate the time/accuracy trade-offs in computing the covariance matrix with and without compressed matrix multiplication in in-database DTA (C-DTA vs. DTA). In particular, we consider different values of the parameter, $\mathbf{d}$, which controls the number of count sketches of the matrix product. In Figures 8.13(b) and (c), the tensor size is 5000x100x10 and we considered the covariance matrix on the first mode (of size 5000x5000). As we see in these figures, as we obtain more count sketches, the accuracy of C-DTA improves, but the execution time gains drop. Based on these results, we choose $\mathbf{d} = 30$ as the default value for our experiments.

Impact of Chunk Density in Compressed Matrix Multiplication. The cost analysis of the compressed matrix multiplication in Section 8.2.2 as well as C-DTA vs. DTA experiments reported in Figure 8.11 implied that compressed matrix multiplication is not effective for sparse data. We next evaluate the running times of compressed matrix multiplication on chunks of different densities. Figure 8.14(a) reconfirms that, as expected, compressed matrix multiplication is not advantageous for



(a)



(b)

Figure 8.14: (a) Impact of chunk density on the running time ratio of exact vs. compressed matrix multiplication; (b) impact of chunk shape on the running time of compressed matrix multiplication

sparse data, but the execution time gains become significant (e.g., 2.5×) as the chunk size and density increase.

Impact of Chunk Shape in Compressed Matrix Multiplication. As discussed in Section 8.2.2, shapes of the chunks can impact the performance of the compressed matrix multiplication. In Figure 8.14(b), we evaluate execution times for different chunk shapes (of the same size). The results show that, as the cost analysis in Section 8.2.2 implies, the running times are highest when n is largest. Moreover, running times are smallest when n and m are close to each other.

Chapter 9

CONCLUSION AND FUTURE WORK

In this chapter, we conclude the thesis and present our future work.

9.1 Conclusion of the thesis

Lifecycle of most data includes a diverse set of operations, from capture, integration, projection, to data decomposition and analysis. Tensor is a natural representation for multi-dimensional data due to its simplicity and tensor-based operations, particularly tensor decompositions have been used to capture higher-order structure of data as higher-order extensions of the matrix singular value decomposition: so they are widely used in multi-aspect analysis. For many multidimensional data applications, tensor operations as well as relational operations need to be supported throughout the data lifecycle. We introduced Tensor Relational Model (TRM) and defined tensor-relational operations on this model in Chapter 3. In Chapter 4, we introduced TensorDB, a tensor-relational data management system, based on TRM, which brings together relational algebraic operations (for data manipulation and integration) and tensor algebraic operations (for data analysis). Although tensor-based representations have proven to be useful for multi-dimensional analysis, the high cost of the operations, due to its high-modality and exponentially increasing complexity in the dimension of the data, makes the applications still challenging. We considered optimization strategies to deal with these challenges in TensorDB. We also focused on building the in-database implementation of static and dynamic tensor decompositions for the in-database TensorDB to address in-memory limitations in in-memory TensorDB.

- In Chapter 5, we proposed a highly efficient, effective, and parallelized `join-by-decomposition` (JBD) strategy for approximately evaluating decompositions within join operations. We also proposed pair selection schemes for the `join-by-decomposition` strategy to approximate the fitness of the combined decomposition. Experimental results confirmed that the efficiency and effectiveness of the proposed `join-by-decomposition` scheme compared to the `join-then-decompose`.

- In Chapter 6, we focused on data processing workflows involving both tensor decomposition and data integration (union) operations and proposed a novel scheme for pushing down the tensor decompositions over the union operations to reduce the overall data processing times and to promote reuse of materialized tensor decomposition results. Experimental results confirmed the efficiency and effectiveness of the proposed decomposition push-down strategy and the corresponding `union-by-decomposition` (UBD) operator.

- In Chapter 7, we proposed a highly efficient, effective, and parallelized `decomposition-by-normalization` (DBN) strategy for approximately evaluating decompositions by normalizing a large relation into the smaller tensors based on the FDs of the relation and then performing the decompositions of these smaller tensors for both CP and Tucker decompositions which are the two most widely used tensor decomposition methods. We also proposed interFD-based partitioning and intraFD-based rank pruning strategies for DBN based on pairwise FDs across the normalized partitions and within each normalized partition, respectively. Experimental results confirmed the efficiency and effectiveness of the proposed DBN scheme, and its interFD and intraFD based optimization strategies, compared to the conventional tensor decomposition.

- In Chapter 8, we focused on in-database implementation of static and dynamic tensor decompositions by leveraging an array database. To tackle the high cost of the operations, due to its high-modality and exponentially increasing complexity in the number of dimensions of the data on disk resident data sets, we discussed implementation of tensor decompositions on a chunk-based array store, and proposed in-database (static and dynamic) tensor decomposition operations to address memory blowup problems when dealing with large, higher-order tensor data, such as in social network and scientific applications.

9.2 Future Work

In this section, we discuss our future research directions.

9.2.1 New Optimization Strategies for Tensor-Relational Model

We proposed optimization strategies for tensor-relational query plans involving data integration operations such as join and union and tensor decomposition operations. Our future work extends these to new optimization strategies for data manipulation operations such as selection along with tensor decomposition operations.

Consider that we keep analyzing a massive amount of data and we want to analyze the data in multiple different contexts. If we already have a tensor compressed domain updated over time, since tensor decomposition is an expensive operation, we want to select only the data of interest in the compressed domain without performing multiple tensor decomposition operations on each subset of the data. For example, Figure 9.1 shows two alternative query plans. As shown in the figure, in the first query plan (a) and the second query plan (b), each query plan on a tensor $\mathcal{X}$ performs a selection operation, which is followed by a tensor decomposition. In the third query plan (c), two selection operations are performed on a tensor decomposition of $\mathcal{X}$, which is more

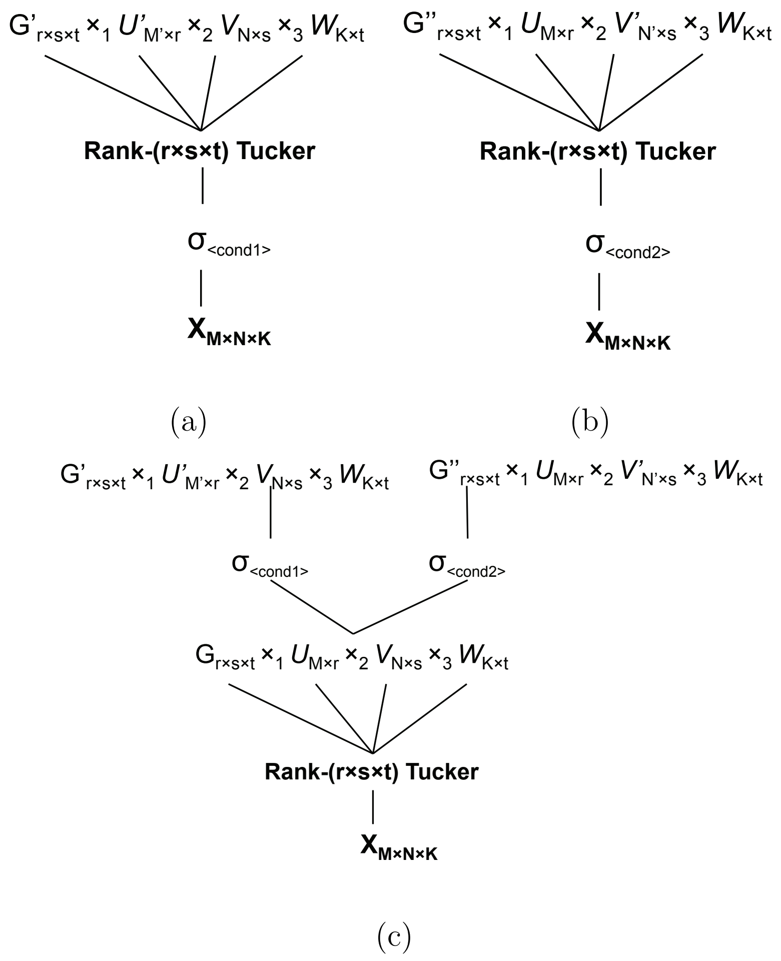

G'r×s×t ×1 U'M'×r ×2 VN×s ×3 WK×t
G"r×s×t ×1 UM×r ×2 V'N'×s ×3 WK×t
Rank-(r×s×t) Tucker
Rank-(r×s×t) Tucker
σ<cond1>
σ<cond2>
XM×N×K
XM×N×K
(a)
(b)
G'r×s×t ×1 U'M'×r ×2 VN×s ×3 WK×t
G"r×s×t ×1 UM×r ×2 V'N'×s ×3 WK×t
σ<cond1>
σ<cond2>
Gr×s×t ×1 UM×r ×2 VN×s ×3 WK×t
Rank-(r×s×t) Tucker
XM×N×K
(c)

Figure 9.1: In (a) and (b), a Tucker decomposition follows a selection operation on a tensor $\mathcal{X}$. Alternatively, two different selection operations are performed on a Tucker decomposition of $\mathcal{X}$ in (c).

efficient than the first and second query plans together.

The problem of performing selection operations on an existing model can also be generalized to a dynamic tensor update problem, which includes removing and inserting sub-tensors on the model. Online updating model in matrix factorization [62] and SVD [16] have been studied to update the model dynamically as the features are updated without rebuilding the model. Incremental tensor decomposition [70] is such an example for tensor decompositions.

These models can be extended to be a general tensor updating model for ten-

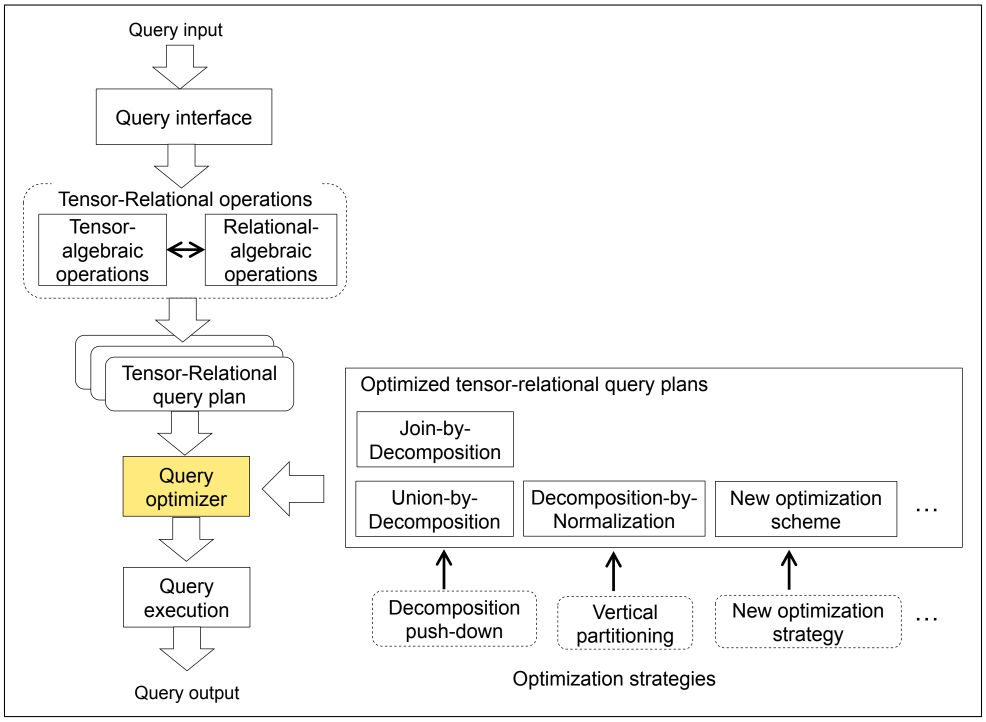

Query input
Query interface
Tensor-Relational operations
Tensor-algebraic operations
Relational-algebraic operations
Tensor-Relational query plan
Query optimizer
Query execution
Query output
Optimized tensor-relational query plans
Join-by-Decomposition
Union-by-Decomposition
Decomposition-by-Normalization
New optimization scheme
...
Decomposition push-down
Vertical partitioning
New optimization strategy
...
Optimization strategies

Figure 9.2: Extended TensorDB with a query optimizer and new optimization strategies

sor decompositions with relational operations such as selection, projection, union, intersection, etc. and within this model, optimization strategies for tensor-relational operations can be developed.

9.2.2 Extension of TensorDB

We proposed query optimization strategies in the tensor-relational model such as decomposition push-down and vertical partitioning and the optimization schemes such as `join-by-decomposition`, `union-by-decomposition`, and `decomposition-by-normalization`. So far we implemented these optimization schemes in in-memory TensorDB. Our future work includes supporting these optimization schemes and a query optimizer that applies the query optimization strategies for the tensor-relational query processing in in-database TensorDB. Figure 9.2 shows the extended TensorDB with the query optimizer and new optimization strategies.

In addition, future work will include exploring opportunities of parallelizing tensor-relational operations of TensorDB in high performance cluster or multicore environments.

REFERENCES

[1] "Sparse basic linear algebra subprograms (blas) library", http://math.nist.gov/spblas/.

[2] "Visual geometry group", http://www.robots.ox.ac.uk/~vgg/data1.html.

[3] "Hash tables and associative arrays", in "Algorithms and Data Structures", pp. 81–98 (Springer Berlin Heidelberg, 2008), URL http://dx.doi.org/10.1007/978-3-540-77978-0_4.

[4] 1, http://www.scidb.org.

[5] 3, "The extended yale face database b", http://vision.ucsd.edu/~leekc/ExtYaleDatabase/ExtYaleB.html.

[6] Acar, E., D. M. Dunlavy, T. G. Kolda and M. Mørup, "Scalable tensor factorizations for incomplete data", Chemometrics and Intelligent Laboratory Systems **106**, 1, 41–56 (2011).

[7] Allen, G. I., "Sparse higher-order principal components analysis", in "Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS)", (2012).

[8] Andersson, C. A. and R. Bro, "Improving the speed of multi-way algorithms:: Part i. tucker3", Chemometrics and intelligent laboratory systems **42**, 1, 93–103 (1998).

[9] Andersson, C. A. and R. Bro, "The n-way toolbox for matlab", Chemometrics and Intelligent Laboratory Systems **52**, 1, 1–4, http://www.models.life.ku.dk/source/nwaytoolbox/ (2000).

[10] Antikainen, J., J. Havel, J. R. Josth, A. Herout, P. Zemcik and M. Hauta-Kasari, "Nonnegative tensor factorization accelerated using gpgpu", IEEE Transactions on Parallel and Distributed Systems **22**, 7, 1135–1141 (2011).

[11] Bader, B. W. and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors", Tech. Rep. SAND2006-7592, Sandia National Laboratories (2006).

[12] Bader, B. W. and T. G. Kolda, "Matlab tensor toolbox version 2.2, jan. 2007", Http://csmr.ca.sandia.gov/ tgkolda/TensorToolbox/ (2007).

[13] Banerjee, A., S. Basu and S. Merugu, "Multi-way clustering on relation graphs", in "Proceedings of the 7th SIAM International Conference on Data Mining", (2006).

[14] Baumann, P., A. Dehmel, P. Furtado, R. Ritsch and N. Widmann, "The multidimensional database system rasdaman", in "Proceedings of the 1998 ACM SIGMOD international conference on Management of data", SIGMOD '98, pp. 575–577 (ACM, New York, NY, USA, 1998), URL `http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/276304.276386`.

[15] Brand, M., "Incremental singular value decomposition of uncertain data with missing values", in "Computer Vision–ECCV 2002", pp. 707–720 (Springer, 2002).

[16] Brand, M., "Fast online svd revisions for lightweight recommender systems", in "In SIAM International Conference on Data Mining", (2003).

[17] Bro, R. and S. De Jong, "A fast non-negativity-constrained least squares algorithm", Journal of Chemometrics **11**, 5, 393–401, URL `http://dx.doi.org/10.1002/(SICI)1099-128X(199709/10)11:5<393::AID-CEM483>3.0.CO;2-L` (1997).

[18] 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", SIGMOD '10, pp. 963–968 (ACM, New York, NY, USA, 2010), URL `http://doi.acm.org/10.1145/1807167.1807271`.

[19] Carroll, J. and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition", Psychometrika **35**, 283–319, URL `http://dx.doi.org/10.1007/BF02310791`, 10.1007/BF02310791 (1970).

[20] Charikar, M., K. Chen and M. Farach-Colton, "Finding frequent items in data streams", in "Automata, Languages and Programming", pp. 693–703 (Springer, 2002).

[21] Chi, Y. and S. Zhu, "Facetcube: a framework of incorporating prior knowledge into non-negative tensor factorization", in "Proceedings of the 19th ACM international conference on Information and knowledge management", pp. 569–578 (2010), URL `http://doi.acm.org/10.1145/1871437.1871512`.

[22] Codd, E. F., "A relational model of data for large shared data banks", Commun. ACM **26**, 1, 64–69, URL `http://doi.acm.org/10.1145/357980.358007` (1983).

[23] Cohen, J., B. Dolan, M. Dunlap, J. M. Hellerstein and C. Welton, "Mad skills: new analysis practices for big data", Proc. VLDB Endow. **2**, 2, 1481–1492, URL `http://dl.acm.org/citation.cfm?id=1687553.1687576` (2009).

[24] Crandall, R. and C. Pomerance, *Prime Numbers: A Computational Perspective* (Springer, 2001).

[25] Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer and R. Harshman, "Indexing by latent semantic analysis", Journal of the American Society for Information Science **41**, 6, 391–407 (1990).

[26] DeWitt, D. J. and R. H. Gerber, "Multiprocessor hash-based join algorithms", in "Proceedings of the 11th international conference on Very Large Data Bases", vol. 11, pp. 151–164 (1985), URL `http://dl.acm.org/citation.cfm?id=1286760.1286774`.

[27] Dobos, L., A. Szalay, J. Blakeley, T. Budavári, I. Csabai, D. Tomic, M. Milovanovic, M. Tintor and A. Jovanovic, "Array requirements for scientific applications and an implementation for microsoft sql server", in "Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases", AD '11, pp. 13–19 (ACM, New York, NY, USA, 2011), URL `http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/1966895.1966897`.

[28] Dunlavy, D. M., T. G. Kolda and E. Acar, "Temporal link prediction using matrix and tensor factorizations", ACM Trans. Knowl. Discov. Data **5**, 2, 10:1–10:27, URL `http://doi.acm.org/10.1145/1921632.1921636` (2011).

[29] Elmasri, R. and S. B. Navathe, *Fundamentals of database systems (2nd ed.)* (Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994).

[30] Feng, X., A. Kumar, B. Recht and C. Ré, "Towards a unified architecture for in-rdbms analytics", in "Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data", SIGMOD '12, pp. 325–336 (ACM, New York, NY, USA, 2012), URL `http://doi.acm.org/10.1145/2213836.2213874`.

[31] Frank, A. and A. Asuncion, "Uci machine learning repository", Irvine, CA: U. of California, School of ICS (2010).

[32] Frigo, M. and S. G. Johnson, "The design and implementation of FFTW3", Proceedings of the IEEE **93**, 2, 216–231, special issue on "Program Generation, Optimization, and Platform Adaptation" (2005).

[33] Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman & Co., New York, NY, USA, 1979).

[34] Harshman, R. A., "Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis", UCLA Working Papers in Phonetics **16**, 1, 84 (1970).

[35] Hellerstein, J. M., C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li and A. Kumar, "The madlib analytics library: or mad skills, the sql", Proc. VLDB Endow. **5**, 12, 1700–1711, URL `http://dl.acm.org/citation.cfm?id=2367502.2367510` (2012).

[36] Huhtala, Y., J. Kärkkäinen, P. Porkka and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies", Comput. J. **42**, 2, 100–111 (1999).

[37] Ilyas, I. F., V. Markl, P. J. Haas, P. Brown and A. Aboulnaga, "Cords: Automatic discovery of correlations and soft functional dependencies", in "SIGMOD Conference", pp. 647–658 (2004).

[38] Jolliffe, I., *Principal Component Analysis* (Springer-Verlag, 1986).

[39] 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”, KDD ’12, pp. 316–324 (ACM, New York, NY, USA, 2012), URL `http://doi.acm.org/10.1145/2339530.2339583`.

[40] Karmarker, N. and R. M. Karp, “The differencing method of set partitioning”, Tech. rep., Berkeley, CA, USA (1983).

[41] Kim, M. and K. S. Candan, “Approximate tensor decomposition within a tensor-relational algebraic framework”, in “Proceedings of the 20th ACM international conference on Information and knowledge management”, pp. 1737–1742 (2011), URL `http://doi.acm.org/10.1145/2063576.2063827`.

[42] Kim, M. and K. S. Candan, “Decomposition-by-normalization (dbn): Leveraging approximate functional dependencies for efficient tensor decomposition”, in “Proceedings of the 21th ACM international conference on Information and knowledge management”, pp. 355–364 (2012).

[43] Kim, M. and K. S. Candan, “Decomposition-by-normalization (dbn): Leveraging approximate functional dependencies for efficient cp and tucker decompositions”, Manuscript submitted for publication (2013).

[44] Kim, M. and K. S. Candan, “Pushing-down tensor decompositions over unions to promote reuse of materialized decompositions”, in “ECML/PKDD”, (2014).

[45] Kolda, T. and J. Sun, “Scalable tensor decompositions for multi-aspect data mining”, in “Proceedings of the 8th IEEE International Conference on Data Mining”, pp. 363 –372 (2008).

[46] Kolda, T. G. and B. W. Bader, “Tensor decompositions and applications”, SIAM Rev. **51**, 3, 455–500, URL `http://dx.doi.org/10.1137/07070111X` (2009).

[47] Kolda, T. G., B. W. Bader and J. P. Kenny, “Higher-order web link analysis using multilinear algebra”, in “Proceedings of the 5th IEEE International Conference on Data Mining”, pp. 242–249 (2005), URL `http://dx.doi.org/10.1109/ICDM.2005.77`.

[48] Lin, Y.-R., J. Sun, H. Sundaram, A. Kelliher, P. Castro and R. Konuru, “Community discovery via metagraph factorization”, ACM Trans. Knowl. Discov. Data **5**, 3, 17:1–17:44, URL `http://doi.acm.org/10.1145/1993077.1993081` (2011).

[49] Liu, X., S. Ji, W. Glanzel and B. D. Moor, “Multi-view partitioning via tensor methods”, IEEE Transactions on Knowledge and Data Engineering **99**, PrePrints (2012).

[50] Lopes, S., J.-M. Petit and L. Lakhal, "Efficient discovery of functional dependencies and armstrong relations", in "Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology", EDBT '00, pp. 350–364 (Springer-Verlag, London, UK, UK, 2000), URL `http://dl.acm.org/citation.cfm?id=645339.650138`.

[51] Low, Y., D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud", Proc. VLDB Endow. **5**, 8, 716–727, URL `http://dl.acm.org/citation.cfm?id=2212351.2212354` (2012).

[52] Mahoney, M. W., M. Maggioni and P. Drineas, "Tensor-cur decompositions for tensor-based data", in "Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining", pp. 327–336 (2006), URL `http://doi.acm.org/10.1145/1150402.1150440`.

[53] Mangasarian, O. L. and W. H. Wolberg, "Cancer diagnosis via linear programming", **23**, 5, 1–18 (1990).

[54] Mannila, H. and K.-J. Räihä, "On the complexity of inferring functional dependencies", Discrete Appl. Math. **40**, 2, 237–243, URL `http://dx.doi.org/10.1016/0166-218X(92)90031-5` (1992).

[55] Mannila, H. and H. Toivonen, "Levelwise search and borders of theories in knowledgediscovery", Data Min. Knowl. Discov. **1**, 3, 241–258, URL `http://dx.doi.org/10.1023/A:1009796218281` (1997).

[56] MovieLens, "Movielens dataset from grouplens research group", Http://www.grouplens.org.

[57] Pagh, R., "Compressed matrix multiplication", in "Proceedings of the 3rd Innovations in Theoretical Computer Science Conference", pp. 442–451 (ACM, 2012).

[58] Papalexakis, E. E., C. Faloutsos and N. D. Sidiropoulos, "Parcube: Sparse parallelizable tensor decompositions.", in "ECML/PKDD (1)", edited by P. A. Flach, T. D. Bie and N. Cristianini, vol. 7523 of *Lecture Notes in Computer Science*, pp. 521–536 (Springer, 2012), URL `http://dblp.uni-trier.de/db/conf/pkdd/pkdd2012-1.html#PapalexakisFS12`.

[59] Phan, A. H. and A. Cichocki, "Parafac algorithms for large-scale problems", Neurocomputing **74**, 11, 1970 – 1984, URL `http://www.sciencedirect.com/science/article/pii/S0925231211000415` (2011).

[60] Phan, A.-H., P. Tichavsky and A. Cichocki, "Candecomp/parafac decomposition of high-order tensors through tensor reshaping", Signal Processing, IEEE Transactions on **61**, 19, 4847–4860 (2013).

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

[62] Rendle, S. and L. Schmidt-Thieme, "Online-updating regularized kernel matrix factorization models for large-scale recommender systems", in "Proceedings of the 2008 ACM conference on Recommender systems", RecSys '08, pp. 251–258 (ACM, New York, NY, USA, 2008), URL http://doi.acm.org/10.1145/1454008.1454047.

[63] Ruggles, S. and M. Sobek, "Integrated public use microdata series: Version 2.0 minneapolis: Historical census projects", URL http://www.ipums.umn.edu/ (1997).

[64] Salton, G., A. Wong and C. S. Yang, "A vector space model for automatic indexing", Commun. ACM **18**, 11, 613–620, URL http://doi.acm.org/10.1145/361219.361220 (1975).

[65] Sanchez, E. and B. R. Kowalski, "Generalized rank annihilation factor analysis", Analytical Chemistry **58**, 2, 496–499, URL http://pubs.acs.org/doi/abs/10.1021/ac00293a054 (1986).

[66] Sanchez, E. and B. R. Kowalski, "Tensorial resolution: A direct trilinear decomposition", Journal of Chemometrics **4**, 1, 29–45, URL http://dx.doi.org/10.1002/cem.1180040105 (1990).

[67] Sanderson, C. *et al.*, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments", Tech. rep., Technical report, NICTA (2010).

[68] Stoer, M. and F. Wagner, "A simple min-cut algorithm", J. ACM **44**, 4, 585–591, URL http://doi.acm.org/10.1145/263867.263872 (1997).

[69] Stonebraker, M., D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik, "C-store: a column-oriented dbms", in "Proceedings of the 31st international conference on Very large data bases", VLDB '05, pp. 553–564 (VLDB Endowment, 2005), URL http://dl.acm.org.ezproxy1.lib.asu.edu/citation.cfm?id=1083592.1083658.

[70] Sun, J., D. Tao, S. Papadimitriou, P. S. Yu and C. Faloutsos, "Incremental tensor analysis: Theory and applications", ACM Trans. Knowl. Discov. Data **2**, 3, 11:1–11:37, URL http://doi.acm.org/10.1145/1409620.1409621 (2008).

[71] Symeonidis, P., A. Nanopoulos and Y. Manolopoulos, "A unified framework for providing recommendations in social tagging systems based on ternary semantic analysis", IEEE Trans. on Knowl. and Data Eng. **22**, 2, 179–192, URL http://dx.doi.org/10.1109/TKDE.2009.85 (2010).

[72] Tang, J., H. Gao, H. Liu and A. Das Sarma, "eTrust: Understanding trust evolution in an online world", (2012).

[73] Tsourakakis, C. E., "Mach: Fast randomized tensor decompositions.", in "Proceedings of the 10th SIAM International Conference on Data Mining", pp. 689–700 (2010), URL `http://dblp.uni-trier.de/db/conf/sdm/sdm2010.html#Tsourakakis10`.

[74] Tucker, L., "Some mathematical notes on three-mode factor analysis", Psychometrika **31**, 3, 279–311, URL `http://dx.doi.org/10.1007/BF02289464` (1966).

[75] van Ballegooij, A., R. Cornacchia, A. P. de Vries and M. Kersten, "Distribution rules for array database queries", in "Proceedings of the 16th international conference on Database and Expert Systems Applications", DEXA'05, pp. 55–64 (Springer-Verlag, Berlin, Heidelberg, 2005), URL `http://dx.doi.org/10.1007/11546924_6`.

[76] Vasilescu, M. A. O. and D. Terzopoulos, "Multilinear analysis of image ensembles: Tensorfaces", in "Proceedings of the 7th European Conference on Computer Vision-Part I", ECCV '02, pp. 447–460 (Springer-Verlag, London, UK, UK, 2002), URL `http://dl.acm.org/citation.cfm?id=645315.649173`.

[77] Wyss, C., C. Giannella and E. L. Robertson, "Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract", in "Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery", DaWaK '01, pp. 101–110 (Springer-Verlag, London, UK, UK, 2001), URL `http://dl.acm.org/citation.cfm?id=646110.679455`.

[78] Zhang, Q., M. W. Berry, B. T. Lamb and T. Samuel, "A parallel nonnegative tensor factorization algorithm for mining global climate data", in "Proceedings of the 9th International Conference on Computational Science", pp. 405–415 (2009), URL `http://dx.doi.org/10.1007/978-3-642-01973-9_45`.

[79] 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).