PARAFAC-SPARK: PARALLEL TENSOR DECOMPOSITIONS ON SPARK

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Selim Eren Bekçe
August 2019

PARAFAC-SPARK: Parallel Tensor Decompositions on Spark
By Selim Eren Bekçe
August 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Muhammet Mustafa Özdal(Co-Advisor)

Halil Altay Güvenir

Fahreddin Şükrü Torun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

PARAFAC-SPARK: PARALLEL TENSOR DECOMPOSITIONS ON SPARK

Selim Eren Bekçe
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
Co-Advisor: Muhammet Mustafa Özdal
August 2019

Tensors are higher order matrices, widely used in many data science applications and scientific disciplines. The Canonical Polyadic Decomposition (also known as CPD/PARAFAC) is a widely adopted tensor factorization to discover and extract latent features of tensors usually applied via alternating squares (ALS) method. Developing efficient parallelization methods of PARAFAC on commodity clusters is important because as common tensor sizes reach billions of nonzeros, a naive implementation would require infeasibly huge intermediate memory sizes. Implementations of PARAFAC-ALS on shared and distributed-memory systems are available, but these systems require expensive cluster setups, are too low level, not compatible with modern tooling and not fault tolerant by design. Many companies and data science communities widely prefer Apache Spark, a modern distributed computing framework with in-memory caching, and Hadoop ecosystem of tools for their ease of use, compatibility, ability to run on commodity hardware and fault tolerance. We developed *PARAFAC-SPARK*, an efficient, parallel, open-source implementation of PARAFAC on Spark, written in Scala. It can decompose 3D tensors stored in common coordinate format in parallel with low memory footprint by partitioning them as grids and utilizing compressed sparse rows (CSR) format for efficient traversals. We followed and combined many of the algorithmic and methodological improvements of its predecessor implementations on Hadoop and distributed memory, and adapted them for Spark. During the kernel MTTKRP operation, by applying a multi-way dynamic partitioning scheme, we were also able to increase the number of reducers to be on par with the number of cores to achieve better utilization and reduced memory footprint. We ran *PARAFAC-SPARK* with some real world tensors and evaluated the effectiveness of each improvement as a series of variants compared

with each other, as well as with some synthetically generated tensors up to billions of rows to measure its scalability. Our fastest variant (*PS-CSRSX*) is up to 67% faster than our baseline Spark implementation (*PS-COO*) and up to 10 times faster than the state of art Hadoop implementations.

Keywords: spark, tensor, factorization, decomposition, parafac, cpd, als, alternating squares, parafac-als, cpd-als, hadoop, grid, partitioning, data science, big data.

ÖZET

PARAFAC-SPARK: SPARK İLE PARALEL TENSÖR AYRIŞIMLARI

Selim Eren Bekçe
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
İkinci Tez Danışmanı: Muhammet Mustafa Özdal
Ağustos 2019

Tensörler veri bilimi uygulamalarında ve bilimsel çalışmalarda sıkça kullanılmakta olan çok boyutlu matrislere verilen isimdir. Paralel Faktör Analizi (PARAFAC) adıyla bilinen ayrışım şekli, yaygın olarak kullanılan alternatif kökler (ALS) tensör ayrıştırma algoritması sayesinde veri üzerindeki örtük özellikler ve faktör matrisleri ortaya çıkarabilmektedir. Günümüzde gelişmiş teknolojiler ve büyük veri biliminin yaygınlaşmasıyla birlikte ortaya çıkan tensörler milyarlarca satır veri içerebilmektedir. Bu algoritmanın basit bir uygulaması çok büyük boyutlarda ara matris ve veri iletişimi gerektirdiğinden, paralel sistemlerde etkin bir biçimde uygulanabilmesi, büyük veri biliminin gelişimi için önem kazanmaktadır. PARAFAC-ALS'in paylaşımlı veya dağıtık bellekli sistemlerde uygulamaları mevcuttur fakat bu sistemler maliyetli sistem yatırımları ve düşük seviye kodlama gerektirir, çağdaş programlama araçlarıyla uyumsuzdur ve olağan sistem ve altyapı arızalarına dayanıklı da değillerdir. Apache Spark önbellek destekli çağdaş ve dağıtık bir programlama platformudur ve Apache Hadoop ekosistemi ile birlikte birçok şirket ve veri bilimcisi tarafından uyumlu, ekonomik ve hataya dayanıklı olmaları sebebiyle tercih edilmektedir. Spark üzerinde Scala diliyle geliştirdiğimiz paralel *PARAFAC-SPARK* uygulaması, üç boyutlu tensörleri düşük bellek tüketimiyle ayrıştırabilmektedir. İşlem sırasında tensörler daha hızlı ve dağıtık şekilde işlenebilmeleri için sıkıştırılmış seyrek satırlar (CSR) formatına dönüştürülür ve tensör küp şeklinde parçalara ayırılıp dağıtılarak işlenir. Bu çalışmada, önceki dağıtık bellek ve Hadoop uygulamalarındaki algoritmik ve yöntemsel geliştirmeler derlenip en uygun şekilde Spark için yeniden uyarlanmıştır. Ayrıca, ana Matrisleştirilmiş Tensör ile Khatri-Rao Çarpımı (MTTKRP) operasyonu sırasında çok boyutlu dinamik paylaştırma tekniği uygulanmış, bu sayede MTTKRP operasyonunun bellek tüketiminde dinamik

paylaştırma katsayısı oranında azalma ve operasyonun son indirgeme aşamasında da işlemci kapasite kullanım oranında artma sağlanmıştır. *PARAFAC-SPARK* uygulamasını 11 gerçek veri içeren tensör ve ölçeklenebilirliği test etmek adına sentetik olarak üretilmiş tensörler için çalıştırdık. En ileri varyasyonumuzun (*PS-CSRSX*), temel Spark varyasyonuna (*PS-COO*) göre %67'e kadar daha hızlı olduğunu ve en ileri Hadoop uygulamalarına göre 10 kata kadar daha hızlı olduğunu gördük.

Anahtar sözcükler: spark, tensör, ayrışım, faktörizasyon, parafac, cpd, alternatif kökler, parafac-als, cpd-als, hadoop, grid, bölümleme, veri bilimi, büyük veri.

Acknowledgement

To my professors Cevdet Aykanat and Mustafa Özdal, I am grateful for your valuable and constructive suggestions and criticism during this research work, thank you for everything.

To my grandfather, an honest, strong, diligent, generous, loving, passionate and modest man, he who always supported me during my studies and endeavors, called me "my professor", showed us how to live in peace and harmony by sharing his unbounded love and unfolding his life's stories in humbleness; you are with us now and we will always remember and love you, rest in peace.

To my beloved wife, mother, father and sister for loving, supporting and encouraging me through my studies; this accomplishment would not have been possible without you, I love you all.

Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 Tensors 4

2.1.1 Definition 4

2.1.2 Representations 5

2.1.3 Storage Formats 7

2.1.4 Matrix Operations 10

2.2 CPD and PARAFAC 11

2.2.1 Operations 13

2.2.2 MTTKRP 14

2.3 Apache Spark 16

2.3.1 RDD Operations 17

2.3.2 Partitioning 19

2.3.3 Persistence 20
2.3.4 Executor Settings 21
2.4 Prior Art 22

3 Methods **24**

3.1 RDD Representations 24
3.2 MTTKRP 26
3.3 Caching & Reordering of Computations 28
3.4 Dense Form Factor Matrices (DFFM) 29
3.5 Grid Partitioning 32
3.6 Compressed Sparse Row (CSR) Processing 37
3.7 Multi-Mode Partitioning 41
3.8 TrMM 43
3.9 DstMM 46
3.10 NormC 47

4 Evaluation **49**

4.1 Variants 50
4.2 Setup and Parameter Tuning 50
4.3 Experiments and Discussion 52

4.3.1 Real-World Tensor Runtime Experiment 52

4.3.2 Data Scalability Experiment 56

5 Conclusion **59**

List of Figures

2.1 A three dimensional tensor 5

2.2 MTTKRP Operation, Naive 14

3.1 MTTKRP Operation, Simplified 26

3.2 Grid partitioning 33

3.3 Multi-Mode Partitioning (Partition Stretching) 41

3.4 TrMM Operation 44

4.1 Bar charts that show the iteration and total runtime values of *gowalla*, *facebook-rm*, *movies-amazon*, *brightkite*, *food* and *NELL-2* datasets for Real-World Tensor Runtime Experiment including results of all variants. 53

4.2 Bar charts that show the iteration and total runtime values of *NELL-1*, *enron-3*, *flickr-3*, *yelp* and *delicious* datasets for Real-World Tensor Runtime Experiment including results of all variants. 54

4.3 Data scalability experiment results that show the correlation between NNZ vs Runtime, with tensors of $I \times I \times I$ size where $I = 10^5$. 56

4.4 Data scalability experiment results that show the correlation between Mode Length (I=J=K) vs Runtime, where $nnz = 10 \times I$. . 57

List of Tables

2.1 Transformation Operations 18

2.2 Action Operations 19

2.3 Storage Levels in Spark 21

3.1 CSR Forms 40

3.2 CSR Form Selection of Factor Matrices 40

4.1 *PARAFAC-SPARK* variants 50

4.2 List of the real-world input tensors that were used for evaluation of all variants with their mode lengths and non-zero counts. . . . 52

4.3 Full combined list of all iteration and total runtime results for Real-World Tensor Runtime Experiment including results of all variants. 55

Chapter 1

Introduction

Tensors are the generalization of matrices into higher dimensions. They are widely used in data science where each dimension models specific attributes, e.g., customers, products, movies, interests, ratings, dates, measurements, etc. Such tensors may have billions of non-zeros. To understand the underlying properties of the data well, various techniques like Tensor Decomposition are exercised. Canonical Polyadic Decomposition (CPD) is one form of Tensor Decomposition which can decompose tensors into lower rank tensors (factor matrices) that allows understanding critical relationships between its elements to be further utilized by data scientists.

PARAFAC-ALS is an algorithm for finding the CPD of an input tensor by utilizing Alternating Least Squares method. It finds core factor matrices which can represent the original tensor with least error. However, the key problem with PARAFAC-ALS is that a naive implementation becomes extremely inefficient due to the huge sizes of the intermediate matrices that need to be materialized during the kernel operation. Therefore the problem needs to be solved progressively in such a way that those intermediary matrices would not need to be materialized even in a distributed setting. This is an interesting property and the key challenging feature of the several remarkable distributed tensor factorization solutions in

distributed and/or shared memory systems such as [1, 2]. Although these systems are historically very popular due to the ability of low-level high-performance parallel programming interfaces, those systems often have high setup and maintenance costs, too low level, not fault tolerant by design and incompatible with popular frameworks and infrastructures used by real-world data scientists.

In this work, we implement a fast and effective tensor decomposition solution on Apache Spark, which is a high-level distributed computing framework, for its built-in fault tolerance, compatibility with Apache Hadoop and popularity among the business world and data science professionals for its integrative ecosystem. Since there are no existing tools for this task in Spark, we will first examine existing solutions on Hadoop's MapReduce and build a tangible baseline implementation of PARAFAC-ALS on Spark that solves the key kernel optimization problem via progressively calculating the intermediaries without fully materializing them, which also performs strictly better than Hadoop counterparts like [3, 4, 5]. This baseline will be our building point for the next phase of optimizations; Reordering of Computations, Caching, Grid Partitioning, Compressed Sparse Row (CSR) Format processing and Multi-Mode Partitioning, which are explained in detail in Chapter 3. Each of these optimizations has some parameters to control their behavior, therefore during our experiments, we will adjust these variables then run different runtime measurements. We will name our implementation *PARAFAC-SPARK*.

Chapter 2 gives detailed preliminary information on tensors, their notation, representation, storage methods; definitions and examples of matrix operations used in PARAFAC-ALS, Canonical Polyadic Decomposition (CPD) and its applications; PARAFAC-ALS algorithm itself and the motivation behind its parallelization. This chapter also introduces Apache Spark, explains its working principles and Resilient Distributed Datasets (RDDs), lists its core operations used in the main phases of *PARAFAC-SPARK* and its partitioning and persistence features. This is important to understand the improvements suggested in the subsequent chapter.

Chapter 3 explains how *PARAFAC-SPARK* was designed and implemented

at its core level. It first starts with how tensors and matrices used in PARAFAC-ALS will be represented as RDDs in Spark context, then moves on to describe the key implementation steps of MTTKRP such as Caching, Factor Matrices, Grid Partitioning, CSR Processing and Multi-Mode Partitioning. It then completes the *PARAFAC-SPARK* definition by describing Spark implementations of all sub-algorithms TrMM, DstMM and NormC with their runtime complexities to mark the end of the chapter.

Chapter 4 starts with explaining the runtime setup, evaluation criteria, methodology and experiments performed including the properties of real-world tensors that were used for evaluation. It then gives the experimental results and includes discussion on those results.

Chapter 5 concludes this work by summarizing achievements and discussion on future work.

Chapter 2

Preliminaries

2.1 Tensors

2.1.1 Definition

Tensors are multi-dimensional arrays. A vector is a one-dimensional tensor, and a matrix is a two dimensional one and the word *tensor* is generally used for higher dimensional arrays. Three-dimensional tensors can be visualized as a cube (Figure 2.1).

Tensors of three or more dimensions are increasingly found in many applications, such as an e-commerce website constructing a three-mode tensor with users, products, and dates of purchase. Such tensor can be utilized in a recommendation or analysis system to derive metrics for the company. Another example tensor is generated sensor data, in which the first dimension represents the units, the second one represents the sensor identifier, and the third is the time of occurrence. Such tensors can be huge; it is typical for an e-commerce website having hundreds of thousands of users and millions of products.

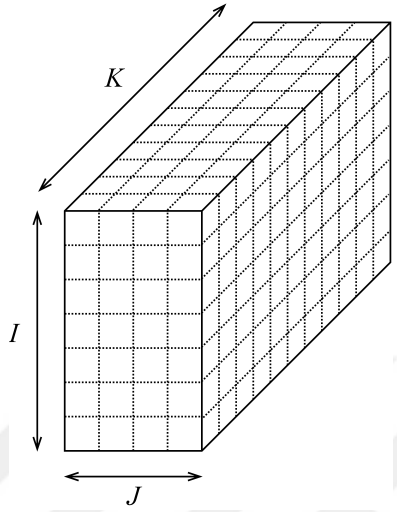

Figure 2.1: A three dimensional tensor

2.1.2 Representations

Vectors are represented with lowercase letters, e.g., a. Matrices are represented with uppercase letters, e.g., B. Tensors of three or more dimensions are represented with uppercase calligraphic fonts, e.g., $\mathcal{X}$.

Elements of tensors are denoted with indices inside square brackets. The ith element of vector a is denoted as $a[i]$. The element on indices (i, j) of matrix A is denoted as $A[i, j]$. The element on indices (i, j, k) of three-mode tensor $\mathcal{X}$ is denoted as $\mathcal{X}[i, j, k]$ and so on.

The number of dimensions of a tensor is called *order* or *mode* of the tensor. Index of each dimension start from 0 and increase until the length of that dimension, e.g., $i = 0, \ldots, I - 1$.

A *fiber* is a vector obtained from fixing all but one dimension of an input tensor, e.g., each row and column of a matrix is a fiber. Fixed indices are denoted with non-negative integers or variables of same sort, whereas a colon character (:) represents a free index, e.g., $\mathcal{X}[i, j, :]$ means the first two indices are fixed and the third is free. For example, $M[2, :]$ denotes the second row of matrix M.

A *slice* is a matrix obtained from fixing all but two dimensions of an input tensor, e.g., each horizontal plane of a three-order tensor is a slice. For example, $\mathcal{X}[:,:,3]$ denotes the matrix in the third plane of tensor $\mathcal{X}$.

Higher order tensors are typically *unfolded* to lower dimensions before applying some mathematical operations on them. An N-dimensional tensor can be *unfolded* (or *flattened*) into N tensors of $N-1$ dimensions. Unfolding a three-mode tensor into a matrix is also called *matricization*. For example, let $\mathcal{X}$ be a three-dimensional tensor with size $I \times J \times K$. $\mathcal{X}^{(1)}$, $\mathcal{X}^{(2)}$, $\mathcal{X}^{(3)}$ are three unfoldings of sizes $I \times JK$, $J \times IK$, $K \times IJ$, respectively. The unfolding operation is applied with following method; to obtain $\mathcal{X}^{(i)}$, traverse ith dimension of $\mathcal{X}$ by looping through next dimension in line each time the boundary of the current dimension is reached.

Example: Let $\mathcal{X}$ be a $2 \times 3 \times 3$ tensor. Each plane (slices of third dimension) is seperated with |, respectively.

$$\mathcal{X} = \left[\begin{array}{ccc|ccc|ccc} 0 & 1 & 7 & 3 & 0 & 8 & 9 & 0 & 1 \\ 0 & 0 & 2 & 6 & 0 & 4 & 2 & 5 & 0 \end{array}\right] \tag{2.1}$$

$\mathcal{X}^{(1)}$, $\mathcal{X}^{(2)}$, $\mathcal{X}^{(3)}$ denote unfoldings around rows, columns and planes of input tensor, respectively.

$$\mathcal{X}^{(1)} = \begin{bmatrix} 0 & 1 & 7 & 3 & 0 & 8 & 9 & 0 & 1 \\ 0 & 0 & 2 & 6 & 0 & 4 & 2 & 5 & 0 \end{bmatrix}$$

$$\mathcal{X}^{(2)} = \begin{bmatrix} 0 & 3 & 9 & 0 & 6 & 2 \\ 1 & 8 & 0 & 0 & 0 & 5 \\ 7 & 0 & 1 & 2 & 4 & 0 \end{bmatrix}$$

$$\mathcal{X}^{(3)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 7 & 2 \\ 3 & 6 & 8 & 0 & 0 & 4 \\ 9 & 2 & 0 & 5 & 1 & 0 \end{bmatrix}$$

Tensors are usually very *sparse*, e.g., containing many zero values (effectively empty cells). As an example, think of a tensor of *user* × *product* × *date* which records transactions for every *user*, *product* and *date*. The most dense tensor (no empty cells) would mean every user bought every product on every day. In real life, only a tiny subset of users will buy a tiny subset of products every day, therefore, this tensor would contain a lot of empty cells. The number of non-zero values in the tensor is denoted with $nnz(\mathcal{X})$. Sparsity is measured with the number of non-zeros divided by the number of cells.

2.1.3 Storage Formats

A tensor can be physically stored and processed in several formats.

2.1.3.1 Dense Format

Dense Format stores an M-mode tensor as an M-D array. An M-mode tensor with dimension sizes l_m where $m = 0 \dots M-1$ will require $\prod_{m=0}^{M-1} l_m$ array cells in dense form. With a 3-mode tensor, the storage space is proportional to IJK. This form takes up the full space of the tensor and only suitable for tensors with small dimensions with very dense content. However, real-world tensors usually have large dimension sizes and are very sparse.

2.1.3.2 Coordinate (COO) Format

Coordinate Format stores a tensor as a list of indices and non-zero values and is a widely used storage format. Each record consists of indices (e.g., $[i, j, k]$) and

value of a non-zero cell. By storing only the non-zero values with their indices, the storage size grows only with $nnz(\mathcal{X})$, compared to the product of dimension sizes in Dense Form. An M-mode tensor $\mathcal{X}$ can be stored with $(M+1) \times nnz(\mathcal{X})$ numbers in Coordinate Format.

Below is the COO representation of the tensor $\mathcal{X}$ given in 2.1. Note that the first three numbers are 0-based indices and the last one is cell value. While the indices are always integers, the cell value can be a real value.

$$
\begin{gathered}
0, 1, 0, 1.0 \\
0, 2, 0, 7.0 \\
1, 2, 0, 2.0 \\
0, 0, 1, 3.0 \\
0, 2, 1, 8.0 \\
1, 0, 1, 6.0 \\
1, 2, 1, 4.0 \\
0, 0, 2, 9.0 \\
0, 2, 2, 1.0 \\
1, 0, 2, 2.0 \\
1, 1, 2, 5.0
\end{gathered}
$$

One advantage of COO format is; since every line is treated as a row, the tensor can now be stored and read in parallel independently at arbitrary intervals. However, since each value is not stored contiguously in the memory, the random access nature brings performance overhead for some operations.

2.1.3.3 Compressed Sparse Row (CSR) Format

Compressed Sparse Row stores an M-mode tensor $\mathcal{X}$ as $M-1$ one-dimensional nnz size arrays and a compressed index array with the size of a preferred major dimension in $\mathcal{X}$. This form is similar to Coordinate Format, it starts with M one-dimensional arrays, but it compresses the index row by replacing it with pointers

for start and end indices.

Below is the CSR representation of the tensor $\mathcal{X}$ given in 2.1. In this example, the first dimension was chosen to be major so the index array A contains pointers for non-zero values. The first index contains value 0; it means the data in the first row of tensor starts at index 0 and ends on index 6 on arrays B, C and V, which contain indices of the second dimension, indices of the third dimension and non-zero values, respectively.

$$A = \begin{bmatrix} 0 & 6 \end{bmatrix}$$

$$B, C, V = \begin{bmatrix} 1 & 2 & 0 & 2 & 0 & 2 & 2 & 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 1 & 1 & 2 & 2 \\ 1 & 7 & 3 & 8 & 9 & 1 & 2 & 6 & 4 & 2 & 5 \end{bmatrix}$$

Storage-wise, since nnz is almost always much larger than the size of any dimension and it requires less amount of index space to store its major dimension, this format is more efficient than COO format. The amount of space saved is dependent on which dimension was chosen to be the major choosing a smaller dimension to be the major produces maximum storage efficiency.

Computationally, CSR is advantageous if the operation requires contiguous memory access which significantly improves processor cache performance. Many matrix operations can easily take advantage of this property; therefore, it is almost always more performant to use CSR for our needs. We will see the performance difference between COO and CSR in subsequent chapters.

The major disadvantage of applying CSR to the whole tensor is that the lengths of these one-dimensional arrays may easily exceed practical limits with many real-world tensors, which hints to split the data into multiple manageable CSR representations. We will discuss more on that in the next chapters.

2.1.4 Matrix Operations

Kronecker Product of two matrices A and B is denoted with $A \otimes B$. Let A be of size $(I \times J)$ and B of size $(K \times L)$, $A \otimes B$ will be of size $(IK \times JL)$.

$$A \otimes B = \begin{bmatrix} A[0,0]B & A[0,1]B & \cdots & A[0,J-1]B \\ A[1,0]B & A[1,1]B & \cdots & A[1,J-1]B \\ \vdots & \vdots & \ddots & \vdots \\ A[I-1,0]B & A[I-1,1]B & \cdots & A[I-1,J-1]B \end{bmatrix} \tag{2.2}$$

Khatri-Rao Product of two matrices A and B is denoted with $A \odot B$. Let A be of size $(I \times J)$ and B of size $(K \times J)$, $A \odot B$ will be of size $(IK \times J)$.

$$A \odot B = \begin{bmatrix} A[:,0] \otimes B[:,0] & A[:,1] \otimes B[:,1] & \cdots & A[:,J-1] \otimes B[:,J-1] \end{bmatrix} \tag{2.3}$$

Example:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 1 \\ 1 & 1 \\ 2 & 3 \end{bmatrix}$$

$$A \odot B = \begin{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} & \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 1 & 2 \\ 2 & 6 \\ 6 & 1 \\ 2 & 1 \\ 4 & 3 \\ 3 & 3 \\ 1 & 3 \\ 2 & 9 \end{bmatrix}$$

Hadamard Product of two matrices A and B is basically element-wise multiplication and denoted by $A * B$. Let A and B be of size $(I \times J)$, $A * B$ will be of size $(I \times J)$, too.

$$A * B = \begin{bmatrix} A[0,0]B[0,0] & \cdots & A[0,J-1]B[0,J-1] \\ \vdots & \ddots & \vdots \\ A[I-1,0]B[I-1,0] & \cdots & A[I-1,J-1]B[I-1,J-1] \end{bmatrix} \tag{2.4}$$

Moore-Penrose Pseudoinverse of a matrix A is denoted as $A^{\dagger}$. Let A be of size $(I \times J)$, $A^{\dagger}$ will then be of size $(J \times I)$.

Outer Product of N vectors produces an N-mode tensor $\mathcal{X}$. The lengths of N vectors define the dimension lengths of the output tensor.

$$\mathcal{X} = a_1 \circ a_2 \circ \ldots \circ a_N \tag{2.5}$$

$$\mathcal{X}[i_1, i_2 \ldots i_N] = a_1[i_1]a_2[i_2] \ldots a_N[i_N] \tag{2.6}$$

A N-mode tensor is called a rank-one tensor if it can be represented with outer product of N vectors.

2.2 CPD and PARAFAC

An N-mode tensor can be decomposed into a finite number of N rank-one tensors. Decomposing multi-dimensional tensors are useful to discover hidden relationships between variables. Parallel Factor Analysis (PARAFAC) is a standard method for decomposing tensors into their factor matrices as in Equation 2.7. It is also known as Canonical Polyadic Decomposition (CPD).

$$\mathcal{X} \approx \sum_{r=0}^{R-1} A[:, r] \circ B[:, r] \circ C[:, r] \tag{2.7}$$

$$\mathcal{X}[i, j, k] \approx \sum_{r=0}^{R-1} A[i, r] B[i, r] C[i, r] \tag{2.8}$$

PARAFAC-ALS is an algorithm for finding the CPD of a tensor by utilizing Alternating Least Squares (ALS) method. The core principle is to find factor matrices that represent the original tensor with least error by iterating multiple times and converging to the best values. It is also regarded as an application of Singular Value Decomposition (SVD) to higher order tensors.

Algorithm 1 PARAFAC-ALS

Require: $\mathcal{X}$ is a three-dimensional tensor with size $I \times J \times K$

1: Initialize A, B, C with random values
2: **for** $t = 0$ to $T - 1$ **do**
3: $A = \mathcal{X}^{(1)}(C \odot B)(C^T C * B^T B)^\dagger$
4: Normalize columns of A
5: $B = \mathcal{X}^{(2)}(C \odot A)(C^T C * A^T A)^\dagger$
6: Normalize columns of B
7: $C = \mathcal{X}^{(3)}(B \odot A)(B^T B * A^T A)^\dagger$
8: Normalize columns of C
9: **end for**

Algorithm 1 decomposes a three-mode tensor $\mathcal{X}$ with dimensions $(I \times J \times K)$ into R factors. Each dimension of $\mathcal{X}$ is decomposed into R vectors with lengths equal to that dimension. Those vectors are then merged into matrices, which are denoted as A, B and C with dimensions $(I \times R)$, $(J \times R)$ and $(K \times R)$, respectively. The formula can be extended to higher dimensions, but we are only going to study the three-dimensional tensor decompositions in this thesis.

The inherent issue for the parallelization of this algorithm is the amount of memory required to calculate the naive Khatri-Rao products $C \odot B$, $C \odot A$ and $B \odot A$, which will be of respective sizes $JK \times R$, $IK \times R$ and $IJ \times R$, where numbers from 10^5 to 10^7 are common for dimension lengths in real-world tensors. These big numbers cause the calculation of factor matrices via naive Khatri-Rao

product extremely inefficient therefore the problem needs to be solved progressively such that the whole matrix product would not need to be materialized. In the Chapter 3, solutions will be described to calculate the resulting factor matrix progressively without materializing it in Spark.

2.2.1 Operations

We will focus on updating factor matrix A (Lines 3 of the PARAFAC-ALS Algorithm 1), then use the same principle for B and C.

$$A = \mathcal{X}^{(1)}(C \odot B)(C^T C * B^T B)^{\dagger}, \tag{2.9}$$

where $A \in \mathbb{R}^{I \times R}$, $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$, $\mathcal{X}^{(1)} \in \mathbb{R}^{I \times JK}$, $(C \odot B) \in \mathbb{R}^{JK \times R}$ and $(C^T C * B^T B)^{\dagger} \in \mathbb{R}^{R \times R}$. The equation has two matrix products inside, since matrix product has associative property, we can approach it from both directions. The runtime complexity of the operation differs by which side the operation was approached:

$$A = [\mathcal{X}^{(1)}(C \odot B)](C^T C * B^T B)^{\dagger} \tag{2.10}$$

$$A = \mathcal{X}^{(1)}[(C \odot B)(C^T C * B^T B)^{\dagger}] \tag{2.11}$$

As stated out in [3], (2.10) requires smaller number of flops compared to (2.11) in practical cases, therefore we will split the computation as,

$$A' = \mathcal{X}^{(1)}(C \odot B) \tag{2.12}$$

$$A'' = (C^T C * B^T B)^{\dagger} \tag{2.13}$$

$$A = A' A'' \tag{2.14}$$

2.2.2 MTTKRP

Matricized Tensor Times Khatri-Rao Product (MTTKRP) is defined in (2.12) and is basically the product of two matrices; the first level unfolding of tensor $\mathcal{X}$ and the Khatri-Rao product of C and B, where $A' \in \mathbb{R}^{I \times R}$, $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$, $\mathcal{X}^{(1)} \in \mathbb{R}^{I \times JK}$ and $(C \odot B) \in \mathbb{R}^{JK \times R}$.

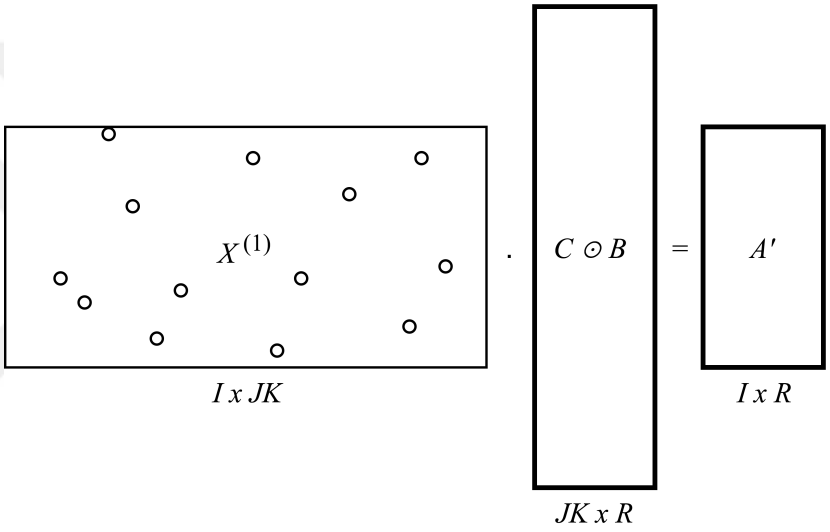

Figure 2.2: MTTKRP Operation, Naive

The inherent problem with this operation is the size of the intermediate matrix $(I \times JK)$ being too huge, also known as Intermediate Data Explosion problem. It becomes infeasible to calculate the Khatri-Rao product naively (Figure 2.2). This can be addressed by directly calculating A' in an indirect element-wise manner, without materializing $(C \odot B)$ [3].

$$A'[i,r] = \sum_{k=0}^{K} C[k,r] \sum_{j=0}^{J} B[j,r] \mathcal{X}[i,j,k] \tag{2.15}$$

Example:

$$\mathcal{X} = \left[\begin{array}{ccc|ccc|ccc} 0 & 1 & 7 & 3 & 0 & 8 & 9 & 0 & 1 \\ 0 & 0 & 2 & 6 & 0 & 4 & 2 & 5 & 0 \end{array}\right]$$

$$B = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 1 \\ 1 & 1 \\ 2 & 3 \end{bmatrix}$$

$$C \odot B = \begin{bmatrix} 3 & 2 \\ 6 & 1 \\ 3 & 3 \\ 1 & 2 \\ 2 & 1 \\ 1 & 3 \\ 2 & 6 \\ 4 & 3 \\ 2 & 9 \end{bmatrix}$$

$$\mathcal{X}^{(1)}(C \odot B) = \begin{bmatrix} 58 & 115 \\ 40 & 57 \end{bmatrix}$$

This calculates A' according to (2.12). Each element in $\mathcal{X}$ is multiplied with B and C then the sum in the last matrix multiplication becomes the result.

$$\begin{aligned} A'[0,0] &= \sum_{k=0}^{2} C[k,0] \sum_{j=0}^{2} B[j,0]\mathcal{X}[0,j,k] \\ &= 3(1 \times 0 + 2 \times 1 + 1 \times 7) \\ &+ 1(1 \times 3 + 2 \times 0 + 1 \times 8) \\ &+ 2(1 \times 9 + 2 \times 0 + 1 \times 1) \\ &= 3(9) + 1(11) + 2(10) = 58 \end{aligned}$$

This calculates the first cell of A' according to (2.15), only to find the same result since two formulas are equivalent. The difference is (2.15) is expressed in a series of operations without explicitly materializing the intermediate matrix.

Exploiting the sparsity of the input tensor via bounding to the non-zero count rather than the dimension lengths allows a scalable execution [6].

(3.1) shows an updated version that calculates an entire row of A' instead of a single cell.

2.3 Apache Spark

Apache Spark is a modern general-purpose parallel programming framework for large scale processing. It was initially developed at Berkeley's AMPLab and later the codebase was given to Apache Foundation and Apache Foundation continues to maintain and extend its functionalities.

It was developed to address some practical and operational limitations of MapReduce paradigm, which uses a linear flow of computations in which a map task reads data from disk and transforms it, then a reduce task aggregates or applies a collaborative function on it, finally storing on disk again. Spark's remarkable implicit data parallelism paradigm, also known as Resilient Distributed Dataset (RDD), provides a distributed memory-like approach to this problem by not requiring the materialization of data between intermediate operations by allowing the next job to use the data from the distributed memory through building directed acyclic graphs (DAG) that represent computational steps. Together with a rich software ecosystem, this paradigm allows a wide range of applications to be developed, including but not limited to, batch processing, stream processing, data warehousing and monitoring, interactive data analysis, iterative algorithms, parallel algorithms, scientific computations and machine learning.

Spark has built-in fault tolerance that allows automatic recalculations of distributed data partitions due to node failures. This feature is more prominent for clusters made up with commodity hardware on top of commodity networks, and the possibility of a node failure in the cluster increases rapidly as the number of machines increases.

Spark programs consist of a driver program; it invokes parallel operations on RDDs by passing lambda functions to the master node, which then in turn schedules and coordinates executions on worker nodes [7].

2.3.1 RDD Operations

Resilient Distributed Datasets can be imagined as big, distributed, unmaterialized and fault-tolerant datasets, partitioned across the entire cluster. An RDD represents an immutable logical data state on which further operations can be applied to produce new RDDs. Operations to be applied on RDDs are two of a kind; *transformations* and *actions*. Transformations are functions that produce new RDDs from existing RDDs, e.g., *Map* is a transformation that produces a new RDD formed by passing each element of input RDD. More types of transformations with their input and output types can be seen in Table 2.1. Actions are functions that mark the end step of an input RDD, by consuming it in the process then either computing some output value or doing other useful work, e.g., *Count* is an action that counts the number of elements in the input RDD; its output is an integer number. More types of actions with their input and output types can be seen in Table 2.2. All transformations are evaluated lazily, e.g., the computation only takes place when an action is applied at the end of given RDD's lifetime. The Spark scheduler builds a directed acyclic graph (DAG) to plan the transformations to be executed until the action step. These operations are written as lambda functions by the programmer, then the driver program serializes and distributes them to the worker nodes, which then run them on their portion of RDDs in parallel.

Name **Parameters** **Input Type** **Output Type**	**Description**
map $[U \rightarrow T]$ $RDD[U]$ $RDD[T]$	Changes type of RDD, the resulting RDD is generated by given function.
$filter$ $[U \rightarrow boolean]$ $RDD[U]$ $RDD[U]$	Filters elements of this RDD, resulting RDD only contains passed elements.
$join$ $RDD[(K,T)]$ $RDD[(K,U)]$ $RDD[(K,(U,T))]$	Joins this RDD with other by their key, the resulting RDD contains all pairs of elements of each key.
$reduceByKey$ $[(U,U) \rightarrow U]$ $RDD[(K,U)]$ $RDD[(K,U)]$	Reduces values with the same key using the provided function, the resulting RDD has same signature.
$partitionBy$ $partitioner$ $RDD[(K,U)]$ $RDD[(K,U)]$	Applies a new partitioner to the keys of input RDD.
$mapPartitions$ $[Iterator[U] \rightarrow Iterator[T]]$ $RDD[U]$ $RDD[T]$	Applies *map* operation to the whole partition rather than to each element; more efficient than *map*.
$mapPartitionsWithIndex$ $[(Int, Iterator[U]) \rightarrow Iterator[T]]$ $RDD[U]$ $RDD[T]$	Like *mapPartitions* but includes global partition index as a parameter as well.
$repartitionAndSortWithinPartitions$ $partitioner$ $RDD[U]$ $RDD[U]$	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys, with customizable ordering.

Table 2.1: Transformation Operations

Each RDD generated via transformations represents an on-the-fly immutable state of a dataset. Once used with next transformation or action, they need to be recalculated again from the start. To reuse RDDs later without the performance penalty of recalculating, they need to be marked as persistent by the programmer. It is possible to store this transient state of data in memory or disk, or to serialize it using a fast serialization library of choice. Caching RDDs in memory across the cluster is a unique feature of Spark, which allows faster execution times via not requiring the intermediate data to be serialized to disk. On the contrary, classic MapReduce needs writing to distributed storage at the end of each computation step.

Name **Parameters** **Output Type**	**Description**
count – *Long*	Returns the count of elements in this RDD.
collect – $Iterable[U]$	Collects all elements of this RDD from the cluster to the driver.
collect $[U \rightarrow Unit]$ –	Applies given function to all elements of input RDD (without collecting to the driver).
reduce $[U \rightarrow U]$ U	Reduces all elements of input RDD to one output element using the given function

Table 2.2: Action Operations

2.3.2 Partitioning

The partitioning of Pair RDDs (RDDs that consist of key and value tuples) is done with a *partitioner* function; a function takes an element and outputs a partition number for that element. Each worker holds consecutive partitions of data, e.g., if there are 20 partitions and 4 workers, each worker gets 5 consecutive partitions. Spark applies hash partitioning by default; the hash of the element becomes its partition, which omits data locality. Doing custom partitioning with

respect to data helps improving locality and reducing communication with other workers.

2.3.3 Persistence

Caching in Spark is very important for its performance benefits. Remember that RDD operations (transformations and actions) are applied according to the generated job DAG, based on the operations performed on driver program. Once an operation is applied on an RDD, it *consumes* the RDD data either to get another RDD (transformation) or a result (action). If the driver program requires the intermediate RDD again, Spark will reapply the operations from the start. If there are some re-used RDDs, this behavior may cause too many recalculations which translate into a poor performance.

Spark's persistence feature allows to mark RDDs as persistent, which will save the RDDs data to memory or disk, so when that RDD is reused in a later step, it will be served directly from the persisted location, skipping calculations up until the persisted point.

In Hadoop, after every map and reduce operation, the results are written back to HDFS by some Writer. Then, the next job (map and reduce pair) reads that data back from HDFS via some Reader. Since HDFS must be involved between every pair of jobs, it is not performant enough compared to in-memory caching, which Spark takes advantage of.

There are multiple storage levels in Spark, which control where and how the data will be stored, shown in Table 2.3. RDDs are not persisted, by default. It is logical because most RDDs are intermediate RDDs between two transformations and they are usually used only once. When desired, an RDD can be persisted by calling `persist(storageLevel)` on them. This call would mark the RDD as persistent and the framework will persist (or cache) the data of this RDD the first time it is computed and will use the persisted data when the data is requested subsequently. Programmer can *unpersist* an RDD by calling `unpersist()`.

Storage Level	Description
MEMORY_ONLY	Store data as objects in the JVM. If the memory is full, recompute the spilling parts on the fly (requires partial recomputation, just as not using any persistence at all).
MEMORY_AND_DISK	Store data as objects in the JVM. After the memory becomes full, spill the extra data to disk.
DISK_ONLY	Store data to disk as serialized objects.
(NONE)	Do not use persistence (compute or re-read if needed).

Table 2.3: Storage Levels in Spark

2.3.4 Executor Settings

There are a couple of runtime parameters that Spark accepts in order to optimize the executor and memory allocations, which need to be optimized for each cluster to be as efficient as possible. They control the number of executors, cores per executor and memory per executor, which affect parallelism and throughput of the application. [8] shows there are three main approaches to set these parameters; tiny executor setting essentially creates one executor per core, fat executor setting creates one executor per node and hybrid executor setting which focuses on fixing cores per executor (4 for our application) and generating the runtime parameters around this value. We confirmed that Tiny and Fat executor settings each perform subprime compared to Hybrid setting according to our experiments.

$$E = (N \times (C_n - 1)) \div C_e - 1 \tag{2.16}$$

$$M_e = 0.93 \times M_n \div ((E + 1) \div N) \tag{2.17}$$

Let N be the total number of nodes in the cluster, C_n be number of cores per node, M_n be the amount of memory per node, E be the number of executors and C_e be number of cores per executor and M_e be the amount of memory per

executor. N, C_n and M_n are known via the cluster property, C_e is chosen to be 4, and equations 2.16 and 2.17 calculate E and M_e respectively according to hybrid setting definition of [8], plus a memory adjustment of 0.93 to account for the other processes on every node.

2.4 Prior Art

Kolda and Bader [9] published algorithms to avoid intermediate materialization of the Khatri-Rao product, which led to new implementations on distributed memory and Hadoop systems.

DFacTo [10] is a distributed memory kernel written in C++ Eigen library. It was reported to be significantly faster than GigaTensor and MATLAB CPALS, but was surpassed later on.

Distributed Memory SPLATT (DMS) in [1], [2] and [6] is the most efficient public implementation of PARAFAC-ALS that runs on distributed memory systems with MPI or shared memory systems with OpenMP.

GigaTensor [3] is an outdated implementation of PARAFAC-ALS on Hadoop. It was released in 2012 and was one of the first papers that explore Tensor Decomposition problem on Hadoop. It solved the scalability problem by using a fully distributed system for the MTTKRP step, but this was causing too much communication overhead and not scaling so well.

HaTeN2 [4] is an efficient implementation on Hadoop, which carefully reorders operations and unifies Tucker and PARAFAC decompositions into a general framework. They synthetically generate tensors to compare scalability with respect to mode lengths and density.

Parallel PARAFAC [5] is also an Hadoop implementation of PARAFAC which experiment different slicing techniques for long and narrow tensors, instead of tensors with arbitrary shapes. They compare their results with HaTeN2; for the

non-zero experiment, while HaTeN2 turns out to be faster for low nnz numbers, Parallel PARAFAC claims to perform better for high nnz values.

Chapter 3

Methods

The PARAFAC algorithm optimized for Spark will be explained in this chapter. First, we will explain the RDD representations used throughout the implementation. We will then move on to explaining how we implemented specific operations in the PARAFAC algorithm on Spark, including decisions made and optimizations performed. After explaining the details of each operation, we will provide a big picture of all operations working together to complete the PARAFAC algorithm. Later comes the essential optimizations such as Caching, Grid Partitioning, CSR processing, and Multi-Mode Partitioning. The careful and selective application of these optimizations are the vital contributions of this thesis.

3.1 RDD Representations

Let $\mathcal{X}$ be a three-dimensional sparse tensor with sizes $I \times J \times K$. A real-world tensor is almost always stored in Coordinate Format, i.e., consisting of $nnz(\mathcal{X})$ rows for each tuple $(i, j, k, value)$. It is usually not preferred to store real-world tensors in either Dense Form or CSR Form for the reasons that they are usually very sparse and CSR requires at least 2 separate arrays (files) to be read, which is not practical for storage purposes. Our program will be reading the tensors in

COO format.

To apply partitioning to RDDs based on their indices, we need to enclose the indices so the RDD typing becomes $((i, j, k), value)$. This means storing a number of $((i, j, k), value)$ typed elements distributed across the worker nodes.

Factor matrices of A, B and C have three representation forms; distributed, dense and hybrid. Their dimensions are $(I \times R)$, $(J \times R)$ and $(K \times R)$, respectively. R is a small number (≈ 10), on the other hand, I, J and K typically reach millions. Therefore the factor matrices are usually thin and long matrices.

Distributed Form packs each row of the matrix into one small array with length R and distributes these rows with their indices as an RDD. So the RDD typing of each of these factor matrices becomes $(index, row)$. In other words, each element of this RDD will map to one row from the factor matrix, including the index of the row and the values in the row. The index of the row is needed because the contents of RDDs are unordered. This scheme allows distributing both calculation and storage of factor matrices on the worker nodes. Without RDD representations, the matrices would need to be fit into a single machine's memory. Distributed Form is suited for matrices that need to be fully distributed because computational work demands so and provides ease of programming at best, but the bookkeeping adds significant additional overhead.

Dense Form, as one can guess, stores the matrix as a full 2D array. When the size of the matrix is small enough, it makes a significant difference vs. using *Distributed Form* as it is free of bookkeeping overheads. As dimension sizes become larger, using *Dense Form* increases communication volume unnecessarily, unless the entire matrix will be needed in every executor.

Hybrid Form splits the factor matrix into many small *Dense Form* matrices on dynamically assigned row intervals. This approach allows executors to only fetch required chunks, instead of requesting rows one-by-one compared to the *Distributed Form* and the entire matrix in *Dense Form*.

The advantage of this method is, it provides best of both worlds: the ability to

distribute and operate on huge matrices and the speed of local computation by utilizing cache coherency, assuming the desired computation can be split along the dimension without needing communication between pairs of out-of-order rows.

3.2 MTTKRP

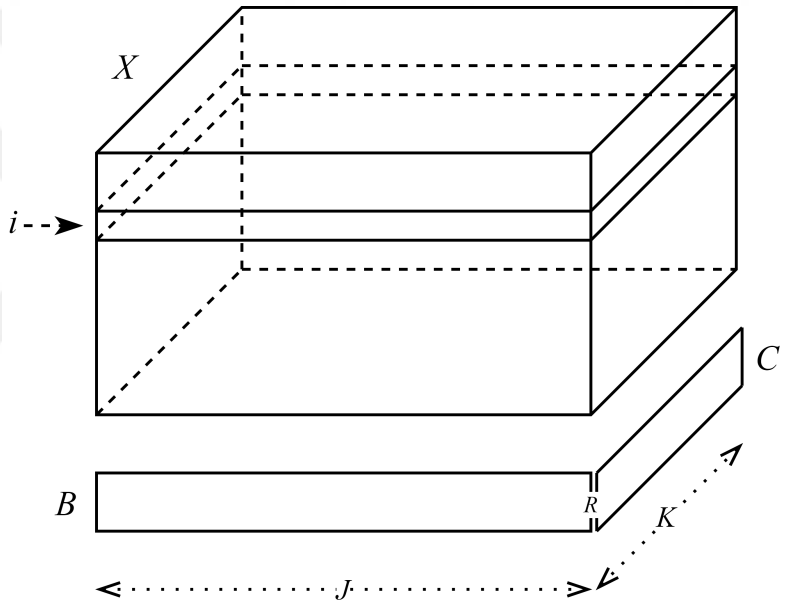

Figure 3.1: MTTKRP Operation, Simplified

$$A'[i,:] = \sum_{k=0}^{K} C[k,:] \sum_{j=0}^{J} B[j,:]\mathcal{X}[i,j,k] \tag{3.1}$$

(3.1) is an improved version of (2.15) where cell-wise computation is replaced by row-rise computation. Since $R \approx 10$, doing whole row at a time makes it more efficient due to improved data locality and reduced upkeep costs.

Each row of A' is calculated by multiplying and summing with B and C across the relevant slice in tensor $\mathcal{X}$. In Figure 3.1, we see tensor $\mathcal{X}$, factor matrices B and C, in the middle of calculating ith row of A', according to (3.1).

Algorithm 2 MTTKRP on Spark, Naive

Require: $\mathcal{X}$ is a three-dimensional tensor with size $I \times J \times K$. B and C are distributed form factor matrices. A' is the intermediate matrix for calculating factor matrix A.

1: join $(k, i, j, \mathcal{X}[i, j, k])$ and $C[k, :]$ on k
2: **for** each $\mathcal{X}[i, j, k], i, j, k$ in $\mathcal{X}$ **do**
3: map $(i, j, \mathcal{X}[i, j, k] \cdot C[k, :])$ as Q
4: **end for**
5: join $(j, i, Q[i, j])$ and $B[j, :]$ on j
6: **for** each $(i, j, Q[i, j])$ in Q **do**
7: map $(i, Q[i, j] \cdot B[j, :])$ as Q'
8: **end for**
9: $A'[i, :] \leftarrow$ reduce $Q'[i, :]$ by i then $\sum array\ Q'[i]$

Algorithm 2 shows the first Spark version of (3.1), which calculates A' by multiplying nonzeros in $\mathcal{X}$ with factor matrices B and C and apply reduction then sum to obtain updated values. Note that this approach uses RDD transformations *map*, *join* and *reduceByKey* to shuffle and gather values on indices therefore $\mathcal{X}$ is stored in COO format and B and C are in distributed form. This is a symbolic description of MTTKRP in Spark and will be rewritten to accommodate further improvements in subsequent sections. We aim to make this calculation distributed in Spark but also want to enforce high local computation power with as little communication as possible.

$$
\begin{aligned}
\text{Example (i=0, r=0)} \rightarrow & \begin{bmatrix} \mathcal{X}[0,0,0] & \mathcal{X}[0,1,0] \\ \mathcal{X}[0,0,1] & \mathcal{X}[0,1,1] \end{bmatrix} \rightarrow \text{join with } C \\
\rightarrow & \begin{bmatrix} \mathcal{X}[0,0,0]C[0,0] & \mathcal{X}[0,1,0]C[0,0] \\ \mathcal{X}[0,0,1]C[1,0] & \mathcal{X}[0,1,1]C[1,0] \end{bmatrix} \rightarrow \text{sum columns} \\
\rightarrow & \begin{bmatrix} XC[0] & XC[1] \end{bmatrix} \rightarrow \text{join with } B \\
\rightarrow & \begin{bmatrix} XC[0]B[0,0] & XC[1]B[1,0] \end{bmatrix} \rightarrow \text{sum rows} \\
\rightarrow & A'[0,0]
\end{aligned}
$$

3.3 Caching & Reordering of Computations

Algorithm 3 PARAFAC-SPARK, Naive

Require: $\mathcal{X}$ is RDD representation of input the tensor with size $I \times J \times K$.
A, B and C are RDD representations of factor matrices.
Functions are named after operation names in subsequent sections.

1: **for** main loop **do**
2: $A \leftarrow mttkrp(\mathcal{X}, C, B)$
3: $A \leftarrow dstmm(A, hadamardInverse(trmm(C), trmm(B)))$
4: $A \leftarrow normc(A)$
5:
6: $B \leftarrow mttkrp(\mathcal{X}, C, A)$
7: $B \leftarrow dstmm(B, hadamardInverse(trmm(C), trmm(A)))$
8: $B \leftarrow normc(B)$
9:
10: $C \leftarrow mttkrp(\mathcal{X}, B, A)$
11: $C \leftarrow dstmm(C, hadamardInverse(trmm(B), trmm(A)))$
12: $C \leftarrow normc(C)$
13: **end for**

Algorithm 3 defines the main loop of *PARAFAC-SPARK* with all operations discussed in subsequent sections. We will apply reordering to reduce redundant operations and caching to take advantage of powerful caching features in Spark.

Performance boost of caching (Section 2.3.3) is related to the reuse factor of each RDD after initially materializing it. Zero reuse means zero benefits minus the storage bookkeeping costs; thus we must make sure to apply it on reused RDDs. Input tensor $\mathcal{X}$ is the most reused tensor with three reuses per iteration. The construction of a partition of $\mathcal{X}$ consists of partially reading a file line by line. If we do not apply any storage level to $\mathcal{X}$, there will be no extra disk or memory usage and no bookkeeping costs, but parsing data cost is higher than reading already parsed data in serialized form. If we apply `DISK_ONLY` level, it will parse the text only once but actual read performance will still be bound to disk, and there will be temporary disk storage costs for the newly allocated files leading to high first iteration and bookkeeping costs. If we apply `MEMORY_ONLY`, the memory will be used for re-reading data with no disk read involved, but it has still some bookkeeping costs and it needs much more memory than the file size

and when there isn't enough memory the spill would lead to more bookkeeping and less performance.

To apply persistence properly in the main loop, we need to understand the behavior of each operation in terms of RDDs. Every operation type has different semantics; *mttkrp* applies transformation on three RDDs, *trmm* applies action on RDD and produces local matrix, *dstmm* applies transformation on RDD, *normc* applies one action then one transformation on RDD. Note that transformation and action are Spark terms discussed in Section 2.3.1.

We must avoid recalculation of factor matrices, and such recalculations occur during actions. The best practice is to *persist* before actions and bring actions closer to each other as much as possible. The latter is already applied with the reordering: *normc* and *trmm* operations have both actions and calls to those functions are very close to each other. We should *persist* after *dstmm* and before *normc* calls, one time for each factor matrix. Storage level of these matrices should be `MEMORY_AND_DISK` since we must avoid recalculation at all costs. Algorithm 4 shows applied reorderings and persistence calls.

3.4 Dense Form Factor Matrices (DFFM)

Using distributed form with factor matrices requires using Spark's *join* transformation to retrieve relevant rows of factor matrices during MTTKRP operation in (2). It results in a global shuffle of data on join key, which needs to be reset before joining with *map* transformation, adds additional complexity and bookkeeping on top of the distributed data. This is not efficient enough for our needs, and we want to eliminate these calls and utilize local computation as much as possible.

Using Dense Form for factor matrices and broadcasting them to all executors would allow us to replace expensive *join* operations with simple *map* operations

Algorithm 4 PARAFAC-SPARK, with reordering & caching

Require: $\mathcal{X}$ is RDD representation of input the tensor with size $I \times J \times K$.
A, B and C are RDD representations of factor matrices.
AtA, BtB, CtC are local transpose matrices of respective factor matrices.
Functions are named after operation names in subsequent sections.

1: $\mathcal{X} \leftarrow persist(\mathcal{X}, \texttt{MEMORY\_ONLY})$
2: $CtC \leftarrow trmm(C)$
3: $BtB \leftarrow trmm(B)$
4: **for** main loop **do**
5: $\quad A \leftarrow mttkrp(\mathcal{X}, C, B)$
6: $\quad A \leftarrow dstmm(A, hadamardInverse(CtC, BtB))$
7: $\quad A \leftarrow persist(A, \texttt{MEMORY\_AND\_DISK})$
8: $\quad A \leftarrow normc(A)$
9: $\quad AtA \leftarrow trmm(A)$
10:
11: $\quad B \leftarrow mttkrp(\mathcal{X}, C, A)$
12: $\quad B \leftarrow dstmm(B, hadamardInverse(CtC, AtA))$
13: $\quad B \leftarrow persist(B, \texttt{MEMORY\_AND\_DISK})$
14: $\quad B \leftarrow normc(B)$
15: $\quad BtB \leftarrow trmm(B)$
16:
17: $\quad C \leftarrow mttkrp(\mathcal{X}, B, A)$
18: $\quad C \leftarrow dstmm(C, hadamardInverse(BtB, AtA))$
19: $\quad C \leftarrow persist(C, \texttt{MEMORY\_AND\_DISK})$
20: $\quad C \leftarrow normc(C)$
21: $\quad CtC \leftarrow trmm(C)$
22: **end for**

that make use of the broadcast factor matrices. This change seems to add communication overhead to the system, but we have found that it results in great runtime improvements compared to using *join* transformations which seem to add far too much communication and bookkeeping overhead. We use Spark's *Torrent-Broadcast* implementation which can efficiently distribute large data objects in small chunks across the cluster via P2P transfers.

The size of the broadcast grows linearly with the size of factor matrices. Table 4.2 highlights the real world tensors that we will use for evaluation and the size of broadcast for a matrix of size $(J \times R)$ is $8JR$ bytes plus administration overhead. As an example, for $J = 10^5$ and $R = 10$, the size is 8 MiB.

Algorithm 5 MTTKRP on Spark, DFFM

Require: $\mathcal{X}$ is a three-dimensional tensor with size $I \times J \times K$.
Require: B and C are factor matrices in *Dense Form*.
Require: A' is the intermediate matrix for calculating factor matrix A.
1: Broadcast B and C
2: **for** each $(i, j, k, \mathcal{X}[i, j, k])$ in $\mathcal{X}$ **do**
3: emit tuple $(i, \mathcal{X}[i, j, k] \cdot C[k, :] \cdot B[j, :])$ as Q
4: **end for**
5: $A'[i, :] \leftarrow$ reduce $Q[i, :]$ by i then $\sum array\ Q[i]$

The change can be seen in Algorithm 5; instead of *join* operation, factor matrices B and C are broadcast at the start of the main loop. Therefore, it allows direct access to the factor matrices in the loop, which makes it possible to remove all *join* calls in each iteration by replacing all transformations with a single *map* function call. This change effectively applies Dense Form for factor matrices to cut down Spark's bookkeeping costs. This is useful to understand the impact of using different runtime forms of the same data but still not ideal as larger factor matrices would increase memory and communication costs of the broadcast. In the subsequent section, we will use Hybrid Form for trimming down memory and communication costs.

3.5 Grid Partitioning

Recall from Section 2.3.2 that Spark RDDs consist of *partitions*, assigned by a *partitioner* function, which allows the programmer to adjust the shape and locality of the data. Input tensor $\mathcal{X}$ and factor matrices A, B and C as well can be partitioned along their dimensions. When $\mathcal{X}$ is imagined as a giant 3D rectangular prism, each partition of $\mathcal{X}$ would be a rectangular prism, cut at approximately equal intervals for every dimension; which -together with all partitions- would make up all the cells of $\mathcal{X}$. This results in increased data locality as every partition can now be processed as a whole with less Spark overhead, rather than shuffled and processed randomly among the entire cluster.

Along with $\mathcal{X}$, the factor matrices A, B and C are also partitioned with the same cut points that correspond to the partition cut points of $\mathcal{X}$ for every dimension and relevant factor matrix (Figure 3.2). Partitioning a *Dense Form* factor matrix requires storing the contents of the whole matrix in multiple separate 2D arrays, one for every part of the whole matrix, which in turn effectively form a *Hybrid Form* factor matrix.

MTTKRP step is then tweaked to only request and access the matching partition of every factor matrix, by using a lazy broadcast handle for each part of *Hybrid Form* factor matrix, which allows a selective broadcast logic to be implemented to only retrieve the data precisely needed for current step, as opposed to having access to the whole factor matrix unnecessarily. This reduces active executor memory and the communication volume for the distribution of factor matrices during every iteration. Nevertheless, this change also requires some implementation overhead to align indices of local partition to those of global tensor and vice versa at the beginning of MTTKRP core step, which is trivial for larger tensors.

The total number of partitions and number of partitions per dimension are denoted with $NP_{total}, NP_I, NP_J, NP_K$, respectively, where $NP_{total} = NP_I \times NP_J \times NP_K$. Dimension sizes of each partition $\mathcal{P}$ in $\mathcal{X}$ are denoted with $I_\mathcal{P}$, $J_\mathcal{P}$ and $K_\mathcal{P}$,

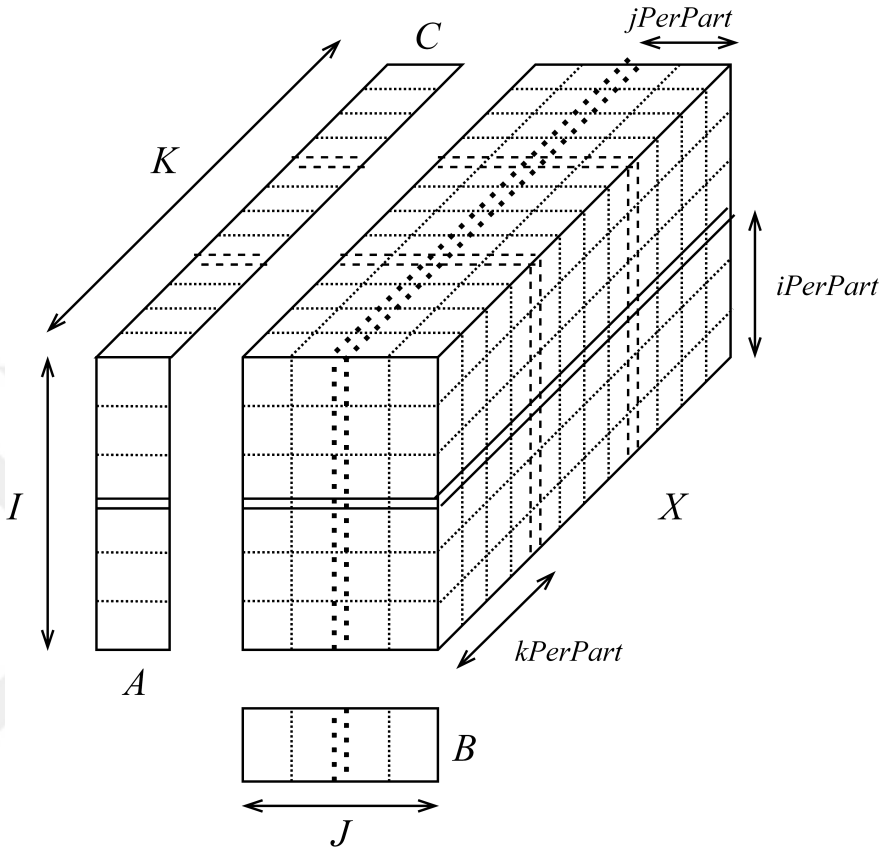

Figure 3.2: Grid partitioning

respectively. Global dimension start indices of $\mathcal{P}$ with respect to $\mathcal{X}$ are denoted with $SI_{\mathcal{P}}$, $SJ_{\mathcal{P}}$ and $SK_{\mathcal{P}}$, respectively. Position (indices) of partition $\mathcal{P}$ alongside $\mathcal{X}$ for each dimension are denoted with $PI_{\mathcal{P}}$, $PJ_{\mathcal{P}}$ and $PK_{\mathcal{P}}$, respectively, where $PI_{\mathcal{P}} \in [0..NP_I)$, $PJ_{\mathcal{P}} \in [0..NP_J)$ and $PK_{\mathcal{P}} \in [0..NP_K)$. Number of nonzero values of $\mathcal{P}$ is denoted as $nnz_{\mathcal{P}}$.

During recalculation of each factor matrix in MTTKRP, each partition works toward a relevant part of A', B' or C', together with the other partitions in the same row, during which the intermediate memory usage for partition $\mathcal{P}$ becomes $O(I_{\mathcal{P}} \times R)$, $O(J_{\mathcal{P}} \times R)$ or $O(K_{\mathcal{P}} \times R)$, with respect to the factor matrix getting build.

The total amount of intermediate memory used across all partitions is also an important metric to measure, though it does not accurately reflect the actual

memory used in the cluster as the computation for each partition can be handled independently, without any requirement of allocating that much memory at a given instant. It can be expressed as $O(NP_{total} \times R \times I \div NP_I)$, $O(NP_{total} \times R \times J \div NP_J)$ or $O(NP_{total} \times R \times K \div NP_K)$; then simplified to $O(NP_J \times NP_K \times I \times R)$, $O(NP_I \times NP_K \times J \times R)$ or $O(NP_I \times NP_J \times K \times R)$, with respect to the factor matrix getting built. This definition will be useful in the upcoming sections to show improvements.

We then replace MTTKRP kernel with a single lambda function to be applied locally on each executor for its assigned partitions as a whole, via *mapPartitionsWithIndex* transformation, rather than general-purpose transformations like *reduceByKey* and *join*. This change requires additional programming for mapping of global indices to local partition indices and vice versa and handling of Hybrid Form factor matrices. With this change, a fine-grained control over the core kernel on every executor can be utilized, instead of relying on the general purpose join and aggregation framework of Spark. This lambda function is applied on every grid partition of the input tensor and outputs a partial factor matrix and its partition index. They would then be folded with the partial factor matrices with the same partition index to produce the entire factor matrix to be used in *DstMM* and *NormC* operations.

The updated MTTKRP Algorithm 6 operates on the entire partition then outputs the intermediate A' matrix. It works with factor matrices in *Hybrid Form* by using lazy global broadcast handles to retrieve only the relevant part of factor matrices selectively. This is made possible by partitioning factor matrices and $\mathcal{X}$ at the same cutoff points for every dimension. indices now need to be mapped from global to local according to partitioning specifications, since all factor matrices relevant to each partition are distributed as small 2D matrices but the coordinate format data maintains global indices. We shall name this baseline implementation as *PS-COO*.

There is a fine balance for finding the best granularity along every mode. A high number of partitions could cause diminishing returns due to increased bookkeeping and context switching costs. On the other hand, less number of

Algorithm 6 MTTKRP on Spark (first dimension shown only), HFFM, PS-COO

Require: $\mathcal{X}$ is a three-dimensional grid partitioned tensor with size $I \times J \times K$, in COO form.

Require: B_{HFFM} and C_{HFFM} are lazy broadcast handles of factor matrices in *Hybrid Form*.

Require: A' is the intermediate matrix for calculating factor matrix A.

1: **for** each partition $\mathcal{P}$ in $\mathcal{X}$, apply following transformation **do**
2: $SI_{\mathcal{P}}$, $SJ_{\mathcal{P}}$, $SK_{\mathcal{P}} \leftarrow$ global start indices of $\mathcal{P}$
3: $I_{\mathcal{P}}$, $J_{\mathcal{P}}$, $K_{\mathcal{P}} \leftarrow$ dimension lengths of $\mathcal{P}$
4: $PI_{\mathcal{P}}$, $PJ_{\mathcal{P}}$, $PK_{\mathcal{P}} \leftarrow$ partition indices of $\mathcal{P}$
5: $C_{\mathcal{P}} \leftarrow$ retrieve $C_{HFFM}[PK_{\mathcal{P}}]$
6: $B_{\mathcal{P}} \leftarrow$ retrieve $B_{HFFM}[PJ_{\mathcal{P}}]$
7: $MAIN \leftarrow$ allocate $array[I_{\mathcal{P}}][R]$
8: **for** each tuple $(i, j, k, \mathcal{X}[i, j, k])$ in $\mathcal{P}$ **do**
9: $i' \leftarrow i - SI_{\mathcal{P}}$
10: $j' \leftarrow j - SJ_{\mathcal{P}}$
11: $k' \leftarrow k - SK_{\mathcal{P}}$
12: increment $MAIN[i'][:]$ by $C_{\mathcal{P}}[k'][:] \cdot B_{\mathcal{P}}[j'][:] \cdot \mathcal{X}[i, j, k]$
13: **end for**
14: **for** $idx = 0$ to $I_{\mathcal{P}}$ **do**
15: emit tuple $(idx + SI_{\mathcal{P}}, MAIN[idx])$ as Q
16: **end for**
17: **end for**
18: $A'[i, :] \leftarrow$ reduce $Q[i, :]$ by i then $\sum array\ Q[i]$

Algorithm 7 Finding optimal partition counts [1]

Require: $NP_{total} \leftarrow$ total partition count target.
Require: $DIM \leftarrow$ array of dimension lengths (i.e., $[I, J, K]$)
1: $PF \leftarrow$ prime factors of NP_{total} in descending order
2: $NP \leftarrow$ allocate $array[length(DIM)]$
3: $NP[i] \leftarrow 1$ for every i
4: $target \leftarrow (\sum_{i=0}^{length(DIM)} DIM[i]) \div NP_{total}$
5: **for** $pf \leftarrow 0$ until $length(PF)$ **do**
6: $\quad furthest \leftarrow 0$
7: $\quad distances \leftarrow$ allocate $array[length(DIM)]$
8: $\quad$ **for** $i \leftarrow 0$ until $length(DIM)$ **do**
9: $\quad\quad distances[i] \leftarrow (DIM[i] \div NP[i]) - target$
10: $\quad\quad$ **if** $distances[i] > distances[furthest]$ **then**
11: $\quad\quad\quad furthest \leftarrow i$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad NP[furthest] \leftarrow NP[furthest] \times PF[pf]$
15: **end for**
16: **return** NP

partitions means reduced parallelism but bigger pieces of data to work on. The granularity of partitions are controlled by the partition count parameters NP_I, NP_J and NP_K, one for every mode, respectively. They are input parameters to the program, however, it is more desirable if those could be assigned automatically in an optimal way regarding the input tensor properties. While determining the partition counts, allocating higher number of partitions along longer modes increases cluster utilization during the fold operation. To automate the selection of partition counts, we adapt the algorithm given in [1] (Algorithm 7). It generates NP, number of partitions per each mode, by first finding prime factors of NP_{total} then assigning them to each mode starting from the largest prime factor, while selecting largest mode and rotating them via dividing their lengths by the assigned prime factors until there is no more prime factor left. In the end, $NP_{total} = NP_I \cdot NP_J \cdot NP_K$ holds true and every mode gets assigned a fair number of partitions depending on their length. It was observed that this automatic assignment scheme always results in a better or similar level of performance than manual assignment of partition counts.

3.6 Compressed Sparse Row (CSR) Processing

Compressed Sparse Row (CSR) format for the input tensor explained in Section 2.1.3.3 is the tensor processing format in MTTKRP kernel of Distributed Memory SPLATT [6] due to its cache locality and reduced memory overhead. We aimed to utilize CSR format in our Spark MTTKRP kernel for faster execution, by utilizing grid partitioning method from *PS-COO* and creating separate CSR structures for each partition $\mathcal{P}$ in $\mathcal{X}$ then modifying MTTKRP to work on these separate CSR structures locally within each partition. Since input tensors are supplied in COO format, they need to be partitioned first, then a CSR structure for each partition is created, according to the same partitioning specifications. For each partition $\mathcal{P}$ of a 3D input tensor $\mathcal{X}$, this means a single big tuple that holds four one dimensional arrays for the three dimensions and one for the non-zeros in CSR. This implementation will continue to utilize *Hybrid Form* for factor matrices and partition alignment features of *PS-COO* thus taking advantage of selective distribution of factor matrices with efficient communication. We shall call this implementation as *PS-CSR*.

The choice of major dimension in CSR format determines the processing direction of the input tensor, and a single CSR form would not suffice to run MTTKRP step without a huge intermediate memory. According to [1], only two CSR forms are enough to process the tensor to efficiently calculate factor matrices, instead of one form for every mode. Therefore, we prepare two CSR forms of the input tensor by implementing an additional preprocessing stage (Algorithm 8). The two smallest modes will be marked as major dimension of the CSR index array to save maximum space. For example, if the smallest modes are first and third, we prepare two cached CSR forms: one form with first mode as major and one form with third mode as major. The ordering of non-zeros is also important in the processing order. Algorithm 8 shows the preprocessing of a CSR form with first mode as major. The resulting CSR forms are cached in memory or disk using Spark's caching feature, so this process is only applied once per decomposition. Tables 3.1 and 3.2 show the two CSR forms, their sort modes and the which mode is used while calculating the factor matrices.

Algorithm 8 Preprocessing a 3D tensor to generate a CSR Form for every partition

Require: $\mathcal{X}$ is a three-dimensional grid partitioned tensor with size $I \times J \times K$.

for each partition $\mathcal{P}$ in $\mathcal{X}$ **do**
  $SI_{\mathcal{P}}$, $SJ_{\mathcal{P}}$, $SK_{\mathcal{P}} \leftarrow$ global start indices of $\mathcal{P}$
  $I_{\mathcal{P}}$, $J_{\mathcal{P}}$, $K_{\mathcal{P}} \leftarrow$ dimension lengths of $\mathcal{P}$
  $PI_{\mathcal{P}}$, $PJ_{\mathcal{P}}$, $PK_{\mathcal{P}} \leftarrow$ partition indices of $\mathcal{P}$
  $nnz_{\mathcal{P}} \leftarrow$ number of nonzeros in $\mathcal{P}$
  $CI_{\mathcal{P}} \leftarrow$ allocate $array[I_{\mathcal{P}}]$
  $CJ_{\mathcal{P}} \leftarrow$ allocate $array[nnz_{\mathcal{P}}]$
  $CK_{\mathcal{P}} \leftarrow$ allocate $array[nnz_{\mathcal{P}}]$
  $CV_{\mathcal{P}} \leftarrow$ allocate $array[nnz_{\mathcal{P}}]$
  $CI[0] \leftarrow 0$
  $lastI \leftarrow 0$
  $index \leftarrow 0$
  Sort in-place tuples $(i, j, k, \mathcal{X}[i, j, k])$ in $\mathcal{P}$ by i then j ascending
  for each tuple $(i, j, k, \mathcal{X}[i, j, k])$ in $\mathcal{P}$ **do**
    $i' \leftarrow i - SI_{\mathcal{P}}$
    $CJ_{\mathcal{P}}[index] \leftarrow j - SJ_{\mathcal{P}}$
    $CK_{\mathcal{P}}[index] \leftarrow k - SK_{\mathcal{P}}$
    $CV_{\mathcal{P}}[index] \leftarrow \mathcal{X}[i, j, k]$
    if $lastI \neq i'$ **then**
      while $lastI < i'$ **do**
        if $lastI > 0$ **then**
          $CI_{\mathcal{P}}[lastI] \leftarrow CI_{\mathcal{P}}[lastI - 1]$
        end if
        $lastI \leftarrow lastI + 1$
      end while
      $CI_{\mathcal{P}}[lastI] \leftarrow index$
    end if
    $index \leftarrow index + 1$
  end for
  return and cache $(CI_{\mathcal{P}}, CJ_{\mathcal{P}}, CK_{\mathcal{P}}, CV_{\mathcal{P}})$
end for

Algorithm 9 MTTKRP on Spark (first dimension shown only), PS-CSR

Require: $\mathcal{X}$ is a three-dimensional grid partitioned tensor with size $I \times J \times K$, in CSR form, major dimension is first. B_{HFFM} and C_{HFFM} are global handles for partitioned factor matrices B and C.

Require: CSR structure (a tuple of 4 arrays) of every partition of $\mathcal{X}$ is sorted internally by i then j.

1: **for** each partition $\mathcal{P}$ in $\mathcal{X}$ **do**
2: $(CI_{\mathcal{P}}, CJ_{\mathcal{P}}, CK_{\mathcal{P}}, CV_{\mathcal{P}}) \leftarrow$ CSR structure of $\mathcal{P}$
3: $SI_{\mathcal{P}}$, $SJ_{\mathcal{P}}$, $SK_{\mathcal{P}} \leftarrow$ global start indices of $\mathcal{P}$
4: $I_{\mathcal{P}}$, $J_{\mathcal{P}}$, $K_{\mathcal{P}} \leftarrow$ dimension lengths of $\mathcal{P}$
5: $PI_{\mathcal{P}}$, $PJ_{\mathcal{P}}$, $PK_{\mathcal{P}} \leftarrow$ partition indices of $\mathcal{P}$
6: $nnz_{\mathcal{P}} \leftarrow$ number of nonzeros in $\mathcal{P}$
7: $B_{\mathcal{P}} \leftarrow$ retrieve $B_{HFFM}[PJ_{\mathcal{P}}]$
8: $C_{\mathcal{P}} \leftarrow$ retrieve $C_{HFFM}[PK_{\mathcal{P}}]$
9: $MAIN \leftarrow$ allocate $array[I_{\mathcal{P}}][R]$
10: $CARRY \leftarrow$ allocate $array[R]$
11: **for** $i = 0$ to $I_{\mathcal{P}}$ **do**
12: $start \leftarrow CI_{\mathcal{P}}[i]$
13: $end \leftarrow$ if $(i == I_{\mathcal{P}} - 1)$ then $nnz_{\mathcal{P}}$ else $CI_{\mathcal{P}}[i+1]$
14: **while** $start < end$ **do**
15: $j \leftarrow CJ_{\mathcal{P}}[start]$
16: $k \leftarrow CK_{\mathcal{P}}[start]$
17: **if** j has changed **then**
18: increment $MAIN[i][:]$ by $CARRY[:] \cdot B_{\mathcal{P}}[j_{previous}][:]$
19: clear $CARRY$
20: **end if**
21: increment $CARRY[:]$ by $CV_{\mathcal{P}}[start] \cdot C_{\mathcal{P}}[k][:]$
22: $start \leftarrow start + 1$
23: **end while**
24: increment $MAIN[i][:]$ by $CARRY[:] \cdot B_{\mathcal{P}}[j_{previous}][:]$
25: clear $CARRY$
26: **end for**
27: **for** each index in $MAIN$ **do**
28: emit tuple $(index + SI_{\mathcal{P}}, MAIN[index])$
29: **end for**
30: **end for**

CSR Form	Sort By
CSR_1_2	(i, j)
CSR_3_2	(k, j)

Table 3.1: CSR Forms

Target Factor Matrix	Used CSR form
A	CSR_1_2
B	if $(K > I)$ CSR_1_2 else CSR_3_2
C	CSR_3_2

Table 3.2: CSR Form Selection of Factor Matrices

Algorithm 9 shows the core kernel of *PS-CSR*, whose main structure is similar to Algorithm 6 of *PS-COO*. The difference is the kernel operates on the 4 one dimensional arrays of CSR form, resulting in cache locality compared to processing tuples in COO form, with improved runtime, which seems to be correlated with the size of input tensor.

3.7 Multi-Mode Partitioning

During the calculation of A', B' and C' for an input tensor $\mathcal{X}$ with dimension sizes $I \times J \times K$ and partition counts NP_I, NP_J and NP_K, where each partition in the grid outputs a portion of the resulting factor matrix. Then, the portions with the same offset are folded via addition operation. The size of this intermediate matrix portions are I/NP_I, J/NP_J and K/NP_K respectively. During the reduction of each factor matrix, number of reduce jobs are equal to the partition counts of the respective factor matrix, which becomes the limiting factor to the cluster utilization as not all executors can be utilized since $E \gg max(NP_I, NP_J, NP_K)$, where E is the number of executors.

To alleviate this problem, we use different partitioning schemes for factorization of each factor matrix, which increases the number of reducers to match NP and increase utilization among the cluster. We multiply the number of partitions for the major dimension by a new *mmp* parameter and divide the number of partitions for the other dimension by the same parameter, for each of the two major dimensions. This change does not change the total number of partitions but changes the shapes of them to favor the first two major dimensions during MTTKRP. We shall call this implementation as *PS-CSRSX*, where $X = mmp$.

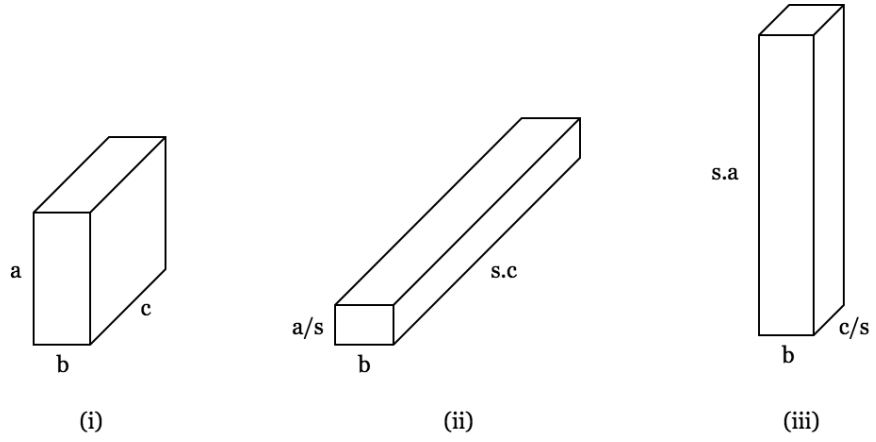

Figure 3.3: Multi-Mode Partitioning (Partition Stretching)

In Figure 3.3, (i) shows a single base partition in the grid, where a, b, c are the lengths of this partition, (ii) shows the resulting partition after applying

MMP to *first* mode by s and (iii) shows the resulting partition after applying MMP to *third* mode by s. For example, let $mmp = 4$, and let partition counts before applying MMP $NP_I = 16, NP_J = 2, NP_K = 32$ for all CSR modes. After applying MMP, partition counts become $NP_I = 64, NP_J = 2, NP_K = 8$ for A and $NP_I = 4, NP_J = 2, NP_K = 128$ for C, respectively.

Multi-Mode Partitioning also decreases both the intermediate memory usage of each partition and the total amount of intermediate memory used across all partitions as well, by mmp parameter during MTTKRP step for first and third mode (i.e., according to the scheme in Figure 3.3). This reduction is achieved by the fact that; while the total partition count NP_{total} remains the same, the amount of intermediate memory used for first and third modes are reduced by mmp, while the memory usage of second mode (the mode for which there is no major CSR form) remains unaffected because the calculation works off a CSR form which is not major relative to the current mode.

Let superscripts mmp_i and mmp_k denote MMP version of the corresponding variable during calculation of the first and third mode factor matrices, respectively; e.g, let $\mathcal{P}^{mmp_i}$ denote the resulting partition after applying MMP by mmp to $\mathcal{P}$ in first mode. The memory usages after applying MMP are found as follows; for first mode partition $\mathcal{P}^{mmp_i}$, it becomes $O(I_{\mathcal{P}}^{mmp_i} \times R)$, where $I_{\mathcal{P}}^{mmp_i} = I_{\mathcal{P}} \div mmp$; for third mode partition $\mathcal{P}^{mmp_k}$, it becomes $O(K_{\mathcal{P}}^{mmp_k} \times R)$, where $K_{\mathcal{P}}^{mmp_k} = K_{\mathcal{P}} \div mmp$, an effective reduction by mmp in both modes. Memory usage for second mode remains the same as the increase and decrease in number of partitions along first and third order cancel each other out. Furthermore, the total amount of intermediate memory used across all partitions after applying MMP are found as follows; for first mode it becomes $O(NP_{total} \times R \times I \div NP_I^{mmp_i})$ where $NP_I^{mmp_i} = NP_I \times mmp$, simplified to $O(NP_J \times NP_K \times I \times R \div mmp)$; for third mode it becomes $O(NP_{total} \times R \times K \div NP_K^{mmp_k})$ where $NP_K^{mmp_k} = NP_K \times mmp$, simplified to $O(NP_J \times NP_I \times K \times R \div mmp)$, an effective reduction by mmp at the total memory usage level in both modes as well.

Multi-Mode Partitioning narrows the surface of the major form partitions to increase parallelism by utilizing more cores with the reduction, without changing

the total data transfer during the entire cluster. Note that the amount of nonzeros for each partition to process remains roughly the same, but the amount of in-memory storage each partition needs to keep (size of *MAIN* array in Algorithm 9) is also reduced by *mmp* factor, as each partition now only needs to keep track and output smaller data, which also increases the reduction performance. We will name this variant as *PS-CSRSX*, where X is a number for *mmp* factor.

PS-CSRSX requires the two major factor matrices (A and C in the example) to be split into smaller partitions by *mmp* factor, as the inner factorizations need to use the same data structure. This increases the number of factor matrix partitions a grid partition must access during the MTTKRP operation by *mmp* factor, but since each partition contains less data by the same factor the amount of data accessed stays the same.

3.8 TrMM

Equation (2.13) defines Moore-Penrose pseudoinverse of Hadamard product of two TrMM operations. Hadamard product and the pseudoinverse is trivial since $C^TC \in \mathbb{R}^{R^2}$ and $R \approx 10$. However, the Transpose Matrix-Matrix Multiplication needs to be executed in parallel.

Transpose Matrix-Matrix Multiplication is the multiplication of the transpose of the input matrix and itself. The optimum runtime complexity of this operation would be the size of the input matrix since the whole matrix would need to be processed at least once. The input matrix is an RDD distributed among workers; therefore we aim for both an effective and efficient parallel method.

In Figure 3.4, matrix C is distributed row-wise among workers as RDD (shown as dotted partitions); therefore we need to keep in mind that only the current row(s) in the current partition will be available while processing. It is also imperative that only C is available to process, if possible C^T should not be recalculated or used explicitly (shown in lighter color). The output matrix A'' will be a small

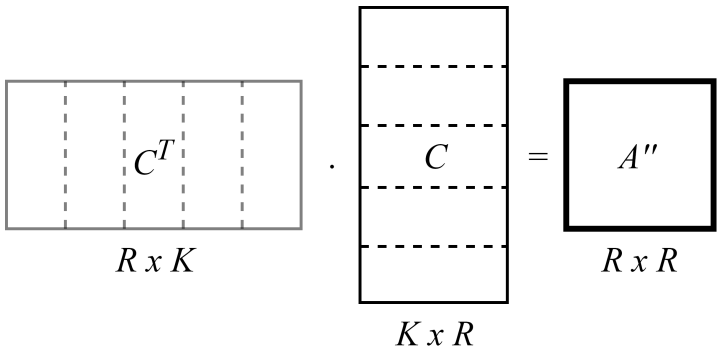

Figure 3.4: TrMM Operation

($R \times R \approx 100$) dense matrix, preferably collected to a local array on the driver program.

The operation C^TC in nature does matrix multiplication of each own pair of columns, resulting in a positive definite symmetric matrix. This property leads to a lesser number of elements to calculate; the other side of the diagonal can be synthetically replicated by one side and the diagonal itself and the diagonal elements are always self products of each column on the respective index.

Algorithm 10 shows the serial version of using a single array to do the entire counting operation, executed with dense form factor matrices, where Algorithm 11 shows the Spark version suitable for distributed and hybrid form factor matrices. It uses a single carry array per partition, instead of a global one, then reduces those small arrays by summing them up in the network.

We expand every row of the input matrix to a vector of pair products individually so that we would need $R(R+1) \div 2$ size intermediate vector for every row of the input matrix. We then apply summing reduction on every column of intermediate vectors to obtain a single vector with $R(R+1) \div 2$ elements. Finally, we convert it into a symmetric matrix to obtain the transpose product. Example:

Algorithm 10 Serial TrMM for DFFM

Require: C is a factor matrix in Dense Form with size $K \times R$. C^T is the non-materialized transpose matrix of C.

$V \leftarrow$ allocate $array[R \times (R+1) \div 2]$
for $k = 0$ to K **do**
  $idx \leftarrow 0$
  for $r = 0$ to R **do**
    for $r' = r$ to R **do**
      increment $V[idx]$ by $C[k, r] \cdot C[k, r']$
      increment idx
    end for
  end for
end for
$C^TC \leftarrow$ convert V to a symmetric matrix with size $R \times R$
return C^TC

Algorithm 11 TrMM on Spark for HFFM

Require: C is the factor matrix with size $K \times R$. C_p is the p^{th} partition of C with size $K_p \times R$. C^T is the non-materialized transpose matrix of C. NP_K is the number of partitions on third dimension.

for $p = 0$ to NP_K, locally **do**
  $V_p \leftarrow$ allocate $array[R \times (R+1) \div 2]$
  for $k = 0$ to K_p **do**
    $idx \leftarrow 0$
    for $r = 0$ to R **do**
      for $r' = r$ to R **do**
        increment $V_p[idx]$ by $C_p[k, r]C_p[k, r']$
        increment idx
      end for
    end for
  end for
  emit V_p
end for
$Y \leftarrow$ reduce V then $\sum$ array $V[k]$
$C^TC \leftarrow$ convert Y to a symmetric matrix with size $R \times R$
return C^TC

$$
\begin{aligned}
C = &\begin{bmatrix} 3 & 1 \\ 1 & 1 \\ 2 & 3 \end{bmatrix} \rightarrow \text{map: expand rows by multiplying each pair} \\
&\rightarrow \begin{bmatrix} 9 & 3 & 1 \\ 1 & 1 & 1 \\ 4 & 6 & 9 \end{bmatrix} \rightarrow \text{reduce: sum columns} \\
&\rightarrow \begin{bmatrix} 14 & 10 & 11 \end{bmatrix} \rightarrow \text{convert to symmetric matrix} \\
&\rightarrow \begin{bmatrix} 14 & 10 \\ 10 & 11 \end{bmatrix} = C^T C
\end{aligned}
$$

The time and space complexity Algorithm 11 are both $\mathcal{O}(KR^2)$.

3.9 DstMM

Distributed Matrix-Matrix Multiplication is defined in (2.14), where $A' \in \mathbb{R}^{I \times R}$ and $A'' \in \mathbb{R}^{R \times R}$. Note that the first element A' is a distributed but A'' is a local matrix. Since A'' is very small (≈ 100), broadcasting it to all workers in dense form using Spark's broadcasting method and doing a fast local matrix multiplication on each worker would be the best action.

The output of DstMM will also be an RDD, with the same RDD type as the input matrix. The key is to use a map function in Spark that transforms the input matrix by multiplying with the local matrix. Since both the input and the output matrices are of the same size, the partitioning does not change. The operation does not need any communication other than the broadcast at the beginning.

Algorithm 12 DstMM on Spark, HFFM

Require: A' is the intermediary output matrix of MTTKRP with size $I \times R$. A'_p is the p^{th} partition of A'. A'' is a local matrix of size $R \times R$. NP_I is the number of partitions on first dimension.

1: Broadcast A'' to all workers
2: **for** $p = 0$ to NP_I, locally **do**
3: **for** each row V in A'_p **do**
4: $O \leftarrow$ allocate $array[R]$
5: **for** $j = 0$ to R **do**
6: $O'[j] \leftarrow 0$
7: **for** $i = 0$ to R **do**
8: $O'[j] \leftarrow O'[j] + V[i]A''[i][j]$
9: **end for**
10: **end for**
11: emit O' as $O[p]$
12: **end for**
13: **end for**
14: **return** O

3.10 NormC

NormC stands for Normalizing Columns, and it is the last operation during calculating one iteration of a factor matrix (Algorithm 13). This function first reads the factor matrices and accumulates the sum of squares for every row, applying an action on the factor matrix. Then, it applies a transformation and returns a new RDD by applying normalization to each element. It is the parallel version of `normc` function in MATLAB.

Algorithm 13 NormC on Spark, HFFM

Require: C is a factor matrix of size $K \times R$. C_p is the p^{th} partition of C with size $K_p \times R$. NP_K is the number of partitions on third dimension.

1: **for** $p = 0$ to NP_K, locally **do**
2: $accLocal \leftarrow$ allocate local $array[R]$
3: **for** each row in C_p **do**
4: **for** $r = 0$ to R **do**
5: $accLocal[r] \leftarrow accLocal[r] + row[r]^2$
6: **end for**
7: **end for**
8: emit $accLocal$
9: **end for**
10: $accGlobal \leftarrow$ reduce $accLocal$ by $\sum$
11: **for** $r = 0$ to R **do**
12: $accGlobal[r] \leftarrow \sqrt{accGlobal[r] \div K}$
13: **end for**
14: Broadcast $accGlobal$
15: **for** each partition index p, locally **do**
16: **for** each row in C_p **do**
17: **for** $r = 0$ to R **do**
18: $row[r] \leftarrow row[r] \div accGlobal[r]$
19: **end for**
20: **end for**
21: **end for**

Chapter 4

Evaluation

In this chapter, we present the experiment setup, runtime setup, evaluation method, runtime results and discuss on them.

We first show the variants in Section 4.1, then explain our runtime setup and parameter tuning methods in Section 4.2, then move on to experiments details and results in Section 4.3, which includes discussion as well.

We first evaluate how does our solution perform with some real-world tensors in Section 4.3.1, and how do the different variants (Table 4.1) compare against each other in terms of runtime. This will allow us to evaluate and compare the improvement of each variant to validate our work and its applicability. We inspect the results for each chart individually and discuss on observed outcomes.

In Section 4.3.2, we evaluate the scalability of our variants with respect to different non-zero counts and mode length by measuring how well they each perform with logarithmically increasing controlled tensor sizes, for which we generate a number of random input tensors for each case. This makes us to understand how well each variant performs in a controlled environment with regard to input tensor properties and the correlation of size vs runtime.

4.1 Variants

We shall evaluate three variants throughout all of our experiments to study the effects and tradeoffs of each development step to runtime. The baseline variant *PS-COO* is a COO format implementation including cache optimization, grid partitioning and *HFFM* (Section 3.5), *PS-CSR* adds distributed CSR processing (Section 3.6), then *PS-CSRSX* adds Multi-Mode Partitioning to reduce intermediate memory usage among partitions and increase cluster utilization (Section 3.7).

Variant	**Description**
PS-COO	Baseline. COO, Grid, HFFM
PS-CSR	CSR Processing
PS-CSRS4	CSR MMP with $mmp = 4$
PS-CSRS8	CSR MMP with $mmp = 8$

Table 4.1: *PARAFAC-SPARK* variants

4.2 Setup and Parameter Tuning

Our experiments were performed on a 21 node (20 workers and 1 driver) Hadoop YARN cluster running on Google Cloud Dataproc with 8 cores and 40 GB RAM per node.

While running the experiments, we recorded the iteration timing of each epoch and picked the best among them to be used as the primary metric for our evaluation. We also recorded total runtime as the supporting metric, which includes pre-processing of intermediary forms and additional I/O as well. We fixed T and R to 10.

A shell script was also developed to run the experiments in an automated fashion on the YARN cluster, one run at a time in the allocated private Spark cluster, which also records the iteration and total runtime of each run. To overcome the timing sensitivity issues that may arise in the private cloud, each run

was performed as 3 identical runs in succession, then the minimum runtime was recorded out of those identical runs to alleviate some perturbances.

Command line parameters for executor properties such as number of executors, cores per executor and memory per executor were calculated via the optimized hybrid setting provided in Section 2.3.4.

The total number of partitions parameter NP_{total} controls total number of partitions of the grid explained in Section 3.5. The minimum value of NP_{total} should be the total number of cores, which is $N \times C_n$. This setting gives a decent performance, but since Spark preallocates tasks to executors in batch, it is advised to increase number of partitions to optimize batch computation and increase cluster utilization futhermore. We executed our experiments with at least two different settings and regard the one that gives the best performence.

All input tensors were uploaded to Google Cloud Storage and read as RDDs by using GCS connector on Spark. The GCS connector allows the reader partitions to scale to an arbitrary executor number without changing any storage properties, where having the files on HDFS would require rewriting partitions to scale reader processes properly.

4.3 Experiments and Discussion

4.3.1 Real-World Tensor Runtime Experiment

We use the real-world input tensors listed in Table 4.2 which were retrieved from the collective tensor archive of our department, where each tensor comes from a particular business case and they all have different mode lengths and non-zero counts. We ran all variants shown on Table 4.1 on all the tensors in Table 4.2 with the setup described in Section 4.2.

Dataset	**I**	**J**	**K**	**nnz**
gowalla	107092	597	1280969	6264232
facebook-rm	42390	39986	1506	738078
movies-amazon	87818	4395	226522	15029290
brightkite	51406	942	772966	2680688
food	67082	11761	82343	5570220
NELL-2	2068070	425	338133	2546098
NELL-1	4454328	434	673976	93194625
enron-3	6066	5699	244268	31312375
flickr-3	319685	28153044	1607878	112894389
yelp	686556	85539	773273	185521461
delicious-3	532923	17262470	2474233	140123362

Table 4.2: List of the real-world input tensors that were used for evaluation of all variants with their mode lengths and non-zero counts.

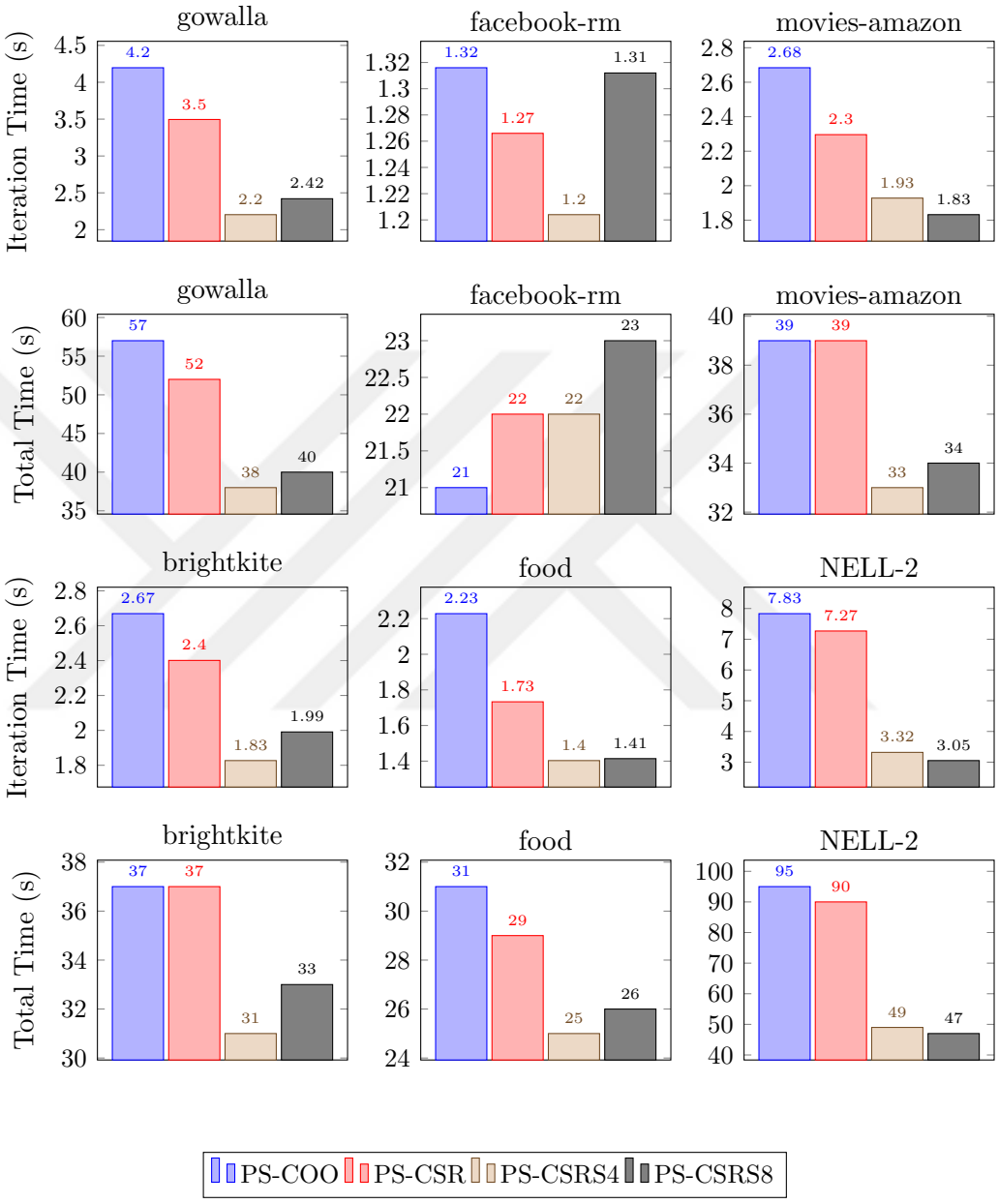

Figure 4.1: Bar charts that show the iteration and total runtime values of *gowalla*, *facebook-rm*, *movies-amazon*, *brightkite*, *food* and *NELL-2* datasets for Real-World Tensor Runtime Experiment including results of all variants.

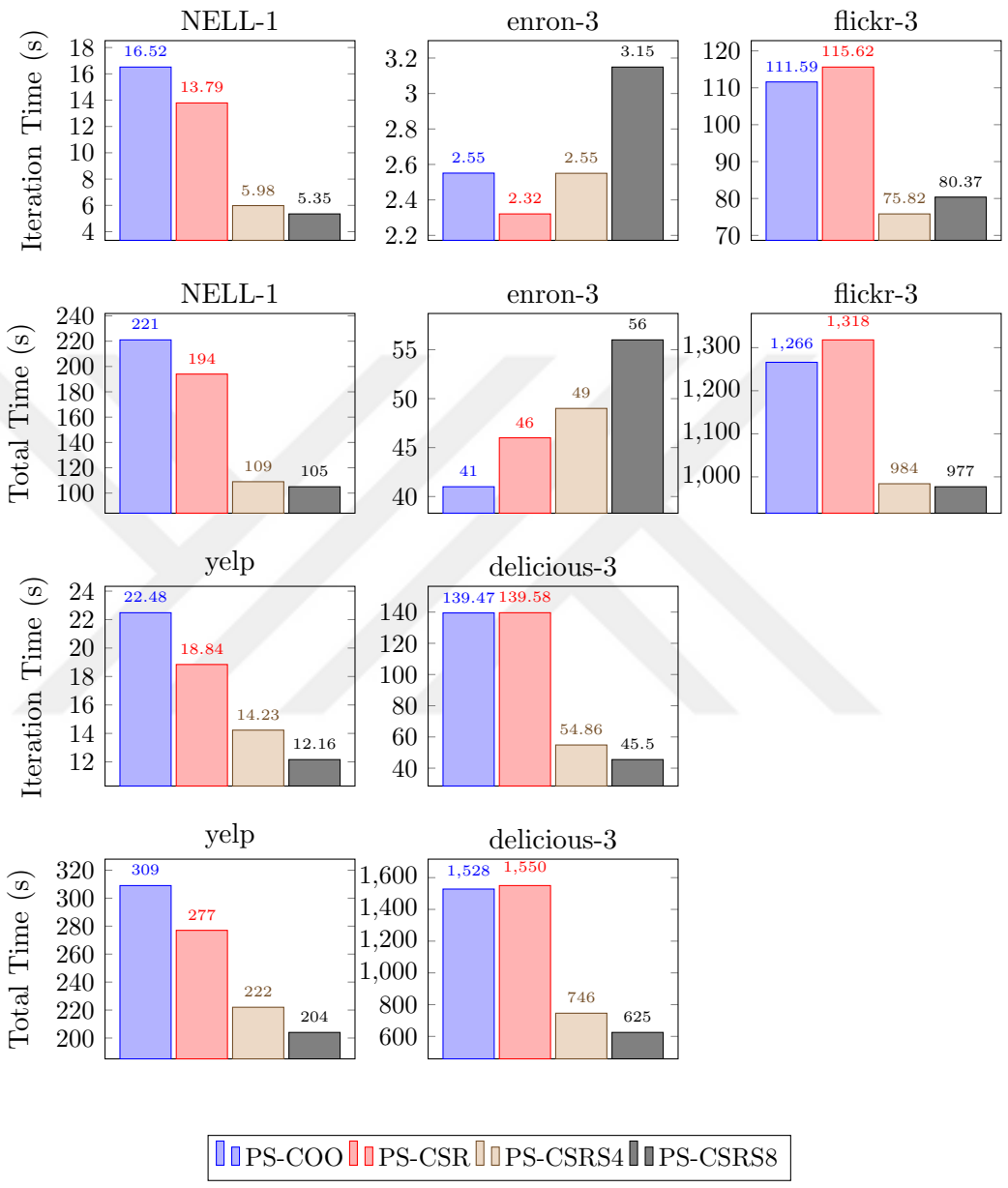

Figure 4.2: Bar charts that show the iteration and total runtime values of *NELL-1*, *enron-3*, *flickr-3*, *yelp* and *delicious* datasets for Real-World Tensor Runtime Experiment including results of all variants.

Dataset	Iteration Time (s)				Total Time (s)			
	COO	**CSR**	**CSRS4**	**CSRS8**	**COO**	**CSR**	**CSRS4**	**CSRS8**
gowalla	4.197	3.495	2.204	2.42	57.0	52.0	38.0	40.0
facebook-rm	1.316	1.266	1.204	1.312	21.0	22.0	22.0	23.0
movies-amazon	2.684	2.296	1.928	1.832	39.0	39.0	33.0	34.0
brightkite	2.669	2.401	1.827	1.991	37.0	37.0	31.0	33.0
food	2.228	1.733	1.403	1.414	31.0	29.0	25.0	26.0
NELL-2	7.833	7.269	3.319	3.052	95.0	90.0	49.0	47.0
NELL-1	16.516	13.789	5.983	5.351	221.0	194.0	109.0	105.0
enron-3	2.551	2.321	2.55	3.149	41.0	46.0	49.0	56.0
flickr-3	111.59	115.617	75.818	80.369	1266.0	1318.0	984.0	977.0
yelp	22.481	18.844	14.231	12.16	309.0	277.0	222.0	204.0
delicious-3	139.474	139.582	54.857	45.503	1528.0	1550.0	746.0	625.0

Table 4.3: Full combined list of all iteration and total runtime results for Real-World Tensor Runtime Experiment including results of all variants.

As seen in Table 4.3 and Figures 4.1 and 4.2, runtime results show that *PS-CSR* is faster than *PS-COO* by up to 15%. This shows the processing advantage of CSR form due to its use of cache locality and smaller memory footprint. CSR's memory and cache advantages are most significant with denser tensors, which can be observed via inspecting runtimes of *flickr-3* vs *yelp* datasets where the latter is heavily denser then the former and *PS-CSR* performs visibly better than *PS-COO* in *yelp* dataset. A similar deduction can be made for other dataset pairs as well.

Meanwhile, *PS-CSRS4* is faster than *PS-CSR* by up to 61 % for large tensors. This shows our MMP improvement was useful on increasing cluster utilization and reducing memory footprint of the reduction at the end of MTTKRP stage. The speed boost of *PS-CSRSX* seems to work out best for bigger but sparser tensors; the increase for *NELL-2* and *movies-amazon* are 55% and 28% respectively, where *NELL-2* is sparser than *movies-amazon*.

In regard to favoring sparsity vs. density, *PS-CSRSX* and *PS-CSR* seem to complement each other well, where *PS-CSRSX* has a clear runtime advantage over any other variant.

For tensors that have two very small modes such as *enron-3*, *PS-CSRSX* seems to perform worse than other variants due to already tiny modes are forced to split.

Runtime difference between *PS-CSRS4* and *PS-CSRS8* marginally stands out on tensors whose dimensions are large enough to take advantage of increased number of splits, such as *NELL-1*, *yelp* and *delicious*, as the larger reduction surface is more useful in those cases. For other tensors, the difference seems insignificant.

4.3.2 Data Scalability Experiment

To evaluate and compare our variants from a data size scalability perspective, we run them against two sets of synthetically generated tensors whose mode length and/or non-zero count properties are increased logarithmically in a controlled manner. This helps us to understand how well they perform against each other as well as how does the runtime correlate with mode length and non-zero counts, for which we generate different sets of tensors up to billion scale, run every variant against them and chart the runtime logarithmically.

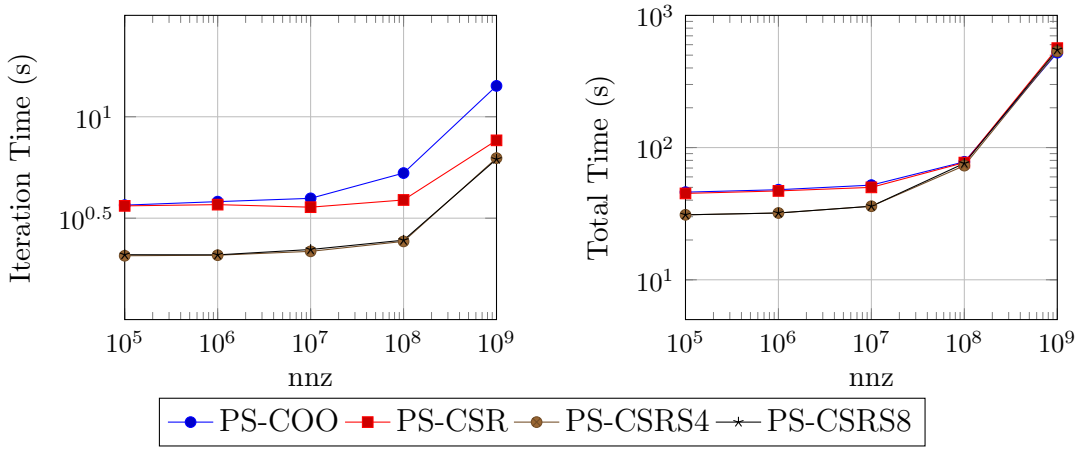

Figure 4.3: Data scalability experiment results that show the correlation between NNZ vs Runtime, with tensors of $I \times I \times I$ size where $I = 10^5$.

Non-zero count: To test the pure effect of increasing non-zero counts while mode lengths stay the same, we generate synthetic tensors with size $I \times I \times I$ where $I = 10^5$ and $nnz = 10^6, 10^7, 10^8, 10^9$, which effectively creates tensors with

1 million non-zeros up to 10 billion non-zeros. We set $NP_{total} = 512$ since input tensors are symmetric and we want to have equal partition sizes among all modes.

As seen in Figure 4.3, *PS-CSRSX* outperforms *PS-COO* and *PS-CSR* in iteration time by 45% to 56%, and *PS-CSR* outperforms *PS-COO* by up to 46%. However, as number of non-zeros increase up to billion scale, CSR preprocessing time also increases (observed via checkpoint timings) and this affects total runtime, resulting in similar performance levels for all variants. If the CSR forms of input tensors would be provided directly to CSR variants, then the total runtime advantage would be on par with iteration time advantage. Nevertheless, these results show *PARAFAC-SPARK* adapts well with increasing non-zero counts (denser tensors) without breaking much sweat.

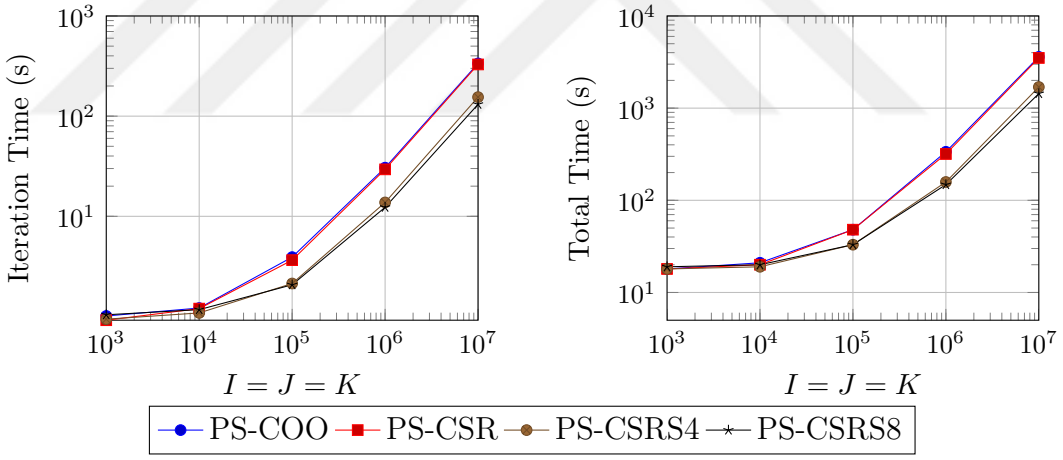

Figure 4.4: Data scalability experiment results that show the correlation between Mode Length (I=J=K) vs Runtime, where $nnz = 10 \times I$.

Mode length: To test the effect of different mode lengths to runtime, we generate synthetic tensors with size $I \times I \times I$, where $I = 10^3, 10^4, 10^5, 10^6, 10^7$ and $nnz = 10 \times I$, which effectively creates tensors which have mode lengths up to 10 million. We again set $NP_{total} = 512$ for balanced partition counts.

As seen in Figure 4.4, for largest tensor, *PS-CSRS8* is faster than *PS-COO* and *PS-CSR* by 55% to 61%. Furthermore, *PS-CSRS8* is faster than *PS-CSRS4* by up

to 15%, which shows the advantage of Multi-Mode Partitioning on large tensors. Increasing mode lengths have a more visible effect than increasing non-zeros, as mode lengths directly affect the communication and computation volume due to the fact that the intermediate memory used during MTTKRP being directly proportional with $O(I + J + K)$, whereas MMP aims to reduce this intermediate memory, which is why this variant performed its best on the largest tensor with ten million mode lengths each. Nevertheless, these results show that *PARAFAC-SPARK* scales well and almost linearly for tensors with bigger mode lengths.

By looking at the published runtime results Parallel PARAFAC [5] provides on a 624-core Hadoop cluster with synthetic tensors whose main dimensions are smaller than the synthetic tensors used in our non-zero count experiment, our fastest variant *PS-CSRSX* seems to be approximately 10 times faster than their best implementation. Furthermore, the data scalability results HaTen2 [4] published by utilizing a 40-node cluster regarding tensor size and density also suggests that *PS-CSRSX* is faster by up to 5 times in iteration time.

Chapter 5

Conclusion

In this work, we explored the tensor factorization problem and implemented *PARAFAC-SPARK*, an efficient general-purpose PARAFAC-ALS implementation on Apache Spark. Some of the optimizations on this algorithm are inspired from the inner workings of previous works such as the reordering of computations from GigaTensor [3], 3D grid partitioning and one-dimensional decompositions of factor matrices from Distributed Memory SPLATT [2], using CSR forms and mode-2 tensor slices from [6]. We aimed to build a practical tool for data scientists and companies for their data science operations which often operate Spark on Hadoop clusters with many other data science tools on commodity hardware rather than utilizing expensive and low-level distributed memory systems. It is useful to note that Spark is not a strictly shared or distributed memory programming framework like OpenMP or MPI, but rather a general purpose data processing framework with automatic failover support and the ability to integrate well with other widely accepted tools. Therefore, there are some missing features when compared to a distributed memory system, e.g. direct message passing between nodes, which need to be rethought via clever use of transformation and action operations the framework offers. We analyzed the theoretical background of PARAFAC and state of the art implementations in distributed and shared memory systems, as well as efficient implementations on Hadoop. We then adapted and implemented some of those concepts that would suit Spark's programming

model carefully from scratch and compiled a baseline implementation, by utilizing a grid partitioning model. Then, we incorporated distributed Compressed Sparse Row (CSR) processing inside Spark partitions which resulted in up to 15% runtime boost compared to baseline variant. Furthermore, we improved it with Multi-Mode Partitioning (MMP) which applies different partitioning properties to each mode, which effectively increases the number of reducers thus increasing cluster utilization and reduce the intermediate memory usage across all executors. It resulted in up to a remarkable 67% runtime improvement over baseline variant. We benchmarked these variants on some real-world input tensors to measure how well our optimizations performed against each other, such that we learned the usefulnesses and tradeoffs of our optimizations across real-world tensors of different variety. Furthermore, we conducted some data scalability experiments to measure the correlation of tensor size and runtime to understand how well our variants perform with mode lengths and non-zero counts reaching up to billion scale. We observed that the runtime grows almost linear with respect to mode length and non-zero counts. We also compared published runtime results of other state-of-art Hadoop implementations on paper and *PARAFAC-SPARK* is up to 10 times faster than any other state-of-art Hadoop implementation, even on less powerful hardware setup. We believe *PARAFAC-SPARK* would be an efficient and valuable tool for tensor factorization on modern computing systems that leverage Apache Spark.

Bibliography

[1] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," *30th IEEE International Parallel & Distributed Processing Symposium*, 2016.

[2] S. Smith and G. Karypis, "Dms: Distributed sparse tensor factorization with alternating least squares," in *UMN-CS TR 15-007*, 2015.

[3] U. Kang, 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, (New York, NY, USA), pp. 316–324, ACM, 2012.

[4] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*, pp. 1047–1058, April 2015.

[5] K. S. Aggour and B. Yener, "Adapting to data sparsity for efficient parallel parafac tensor decomposition in hadoop," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 294–301, Dec 2016.

[6] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *29th IEEE International Parallel & Distributed Processing Symposium*, 2015.

[7] Wikipedia, "Apache spark — wikipedia, the free encyclopedia," 2017. [Online; accessed 23-September-2017].

[8] S. Poddutur, "Distribution of executors, cores and memory for a spark application running in yarn," 2018. [Online; accessed 23-June-2019].

[9] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, pp. 455–500, Aug. 2009.

[10] J. H. Choi and S. V. N. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, (Cambridge, MA, USA), pp. 1296–1304, MIT Press, 2014.