Hashed Coordinate Sparse Tensor Storage with MATLAB

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Jama MeiLi Charles

August 2023

For my cat, Gonzo, who has navigated many homes before coming home to me.

Acknowledgements

I would like to thank my advisor, Dr. Berry, for his guidance and support throughout my entire journey as a graduate student, as well as my committee, Dr. Michael Jantz, Dr. Joan Lind, and Dr. Audris Mockus. I'd like to provide special thanks to Dr. Robert Lowe, who has been a steadfast and encouraging presence from my time as an undergraduate until now, Dr. Maria Siopsis and Dr. Cara Phillips for their aid in my pursuit of professional opportunities, and Mrs. Cindy Arnold, who unsel shly took the time to guide and mentor me through co-teaching with her at Pellissippi State Community College.

Abstract

Tensors, or *n*-way arrays, are incredibly useful for storing indexable data in an arbitrary number of dimensions. Interest in tensor analysis using tensor decomposition has expanded to a variety of fields, including data mining, signal processing, computer vision, and machine learning. Tensors modelling interesting data may also be sparse, where the majority of its values are zero. These tensors can be extremely large and contain millions of entries that cannot be stored explicitly. To address this problem, various formats have arisen in the past decade to compress and compact such massive data. However, most of these existing structures are static and do not support tensor updates. This motivated the proposal of a new format in 2021, Hashed Coordinate Storage (HaCOO), a mode-agnostic format that stores sparse tensor indexes and values in a separate chaining hash table to rapidly insert and access arbitrary entries in constant time. To investigate the benefits of this novel format, we introduce a MATLAB class to create and manipulate sparse tensors in HaCOO format. This class was evaluated alongside MATLAB Tensor Toolbox using several real-world sparse tensor datasets to compare tensor update capability and MTTKRP, a key kernel in Canonical Polyadic Decomposition. Additionally, we discuss how HaCOO format can greatly accelerate building document tensors in a practical application of using sparse tensor decomposition in a text analysis model.

Table of Contents

Chapter 1

Introduction

Tensors, or *n*-way arrays, can express data in an arbitrary number of dimensions. Interest in tensor analysis has expanded to a variety of fields including data mining [12], text analysis [4], machine learning [22], chemometrics [9], signal processing [13], neuroscience [6], computer vision [28], and more. Similar to storing values in a two-way matrix, tensors can store data in any number of dimensions. Many important application domains produce and manipulate massive amounts of multidimensional data; for example, a database that stores customer data. Expressing this data as a tensor is an intuitive way of organizing any number of customers with an arbitrary number of dimensions. In tensor form, individual attributes of any customer can be referenced by a unique location, or index.

Tensor data can also be sparse, where the majority of its entries are zero. Consider the previous example using a tensor to represent customer data. Suppose a field is reserved to track whether a customer has purchased a specific product using a binary flag indicator. If we isolate our view to a single customer and the individual's purchased products, it is likely that the customer has not purchased the majority of items offered, so most values would be represented by zero. It can be reasonably inferred that this is typical customer behavior, so a tensor representing this data

would be sparse. However, it is not efficient to store this data in dense format, if we only consider nonzero values to be meaningful.

Therefore, the question of how to efficiently store and work with sparse tensor data has been the focal point of various research in the past decade. Multiple specialized sparse tensor formats have been developed, ranging from list, block, and tree data structures. While all of these formats offer varying ranges of benefits and drawbacks, most formats produce static data structures and are unable to support tensor updates, such as inserting a new value, without completely rebuilding from scratch. This issue motivated this study to implement a recently proposed sparse tensor storage format, known as Hashed Coordinate format, or HaCOO, which uses hashing to quickly insert and reference sparse tensor entries. Chapter 2 introduces basic tensor concepts and notation, as well as Canonical Polyadic tensor decomposition and its role in tensor analysis. Section 2.3 provides an overview of common sparse tensor formats, as well as HaCOO format. Chapter 3 discusses our approach for a robust MATLAB HaCOO class implementation. Chapter 4 presents evaluations for this class compared to MATLAB Tensor Toolbox, which uses Coordinate, or COO format, the de-facto standard sparse tensor storage format. Chapter 5 discusses HaCOO format in a practical application of text analysis using influence modeling. Chapter 6 presents conclusions as well as future goals.

Chapter 2

Background

This paper follows the conventional notation described in Kolda and Bader's Survey of Sparse Tensor Applications [11]. A reference for relevant symbols and notation is provided in Table 2.1.

2.1 Notation and Terminology

Although the term \tensor" is shared with elds such as physics and engineering, this discussion concerns the coordinate form of a tensor, which is a multimodal array. A tensor is typically represented using calligraphic letters (e.g. $\mathcal{X}$). The *order* of a tensor, or N, is determined by the number of ways, or *modes*. Vectors, or tensors of order one, are denoted by boldface letters, and matrices, or tensors of order two, are denoted by capital boldface letters. Tensors of order three and above are simply called higher order tensors. For the sake of simplicity, a three-way tensor, or a cube, seen in Figure 2.1, will be used for discussion, but concepts can be extended to an arbitrary number of dimensions. As with matrices, tensor elements can be accessed using subscripts. Given a three-way tensor $\mathcal{X}$ 2 $\mathrm{R}^{I\ J\ K}$, the $i; j; k_{th}$ entry can be written as $\mathcal{X}_{i;j;k}$.

Table 2.1: Tensor notation.

Notation	Description
$\mathcal{X}$	Tensor
N	Tensor order
M	Number of nonzeros in $\mathcal{X}$
$\mathcal{X}_{i;j;k}$	Element at $(i;j;k)$ of $\mathcal{X}$
$\mathcal{X}_{:;j;k}$	Column ber of $\mathcal{X}$
$\mathcal{X}_{i;:;:}$	Horizontal slice of $\mathcal{X}$
$\mathbf{A}$	Matrix
$\mathbf{a}$	Vector
$\mathbf{A}_{i;j}$	Element at index $(i;j)$ of matrix $\mathbf{A}$
$\mathbf{a}_i$	Element at index i of vector $\mathbf{a}$
	Hadamard product
	Kronecker product
	Khatri-Rao product

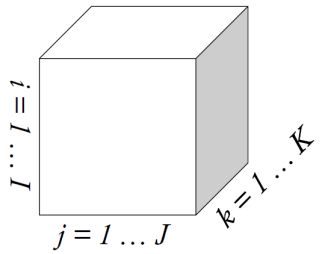

Figure 2.1: A third-order tensor.

Fibers and Slices

Subsets of tensor elements can be extracted by using a colon (:) to refer to all indexes in that mode. A tensor *ber* can be formed by xing all but one mode. Similarly, xing all but two modes results in a tensor *slice*. All variations of bers and slices of a three-way tensor are illustrated in Figure 2.2.

Matricization

A tensor can be *matricized*, or unfolded along any of its modes. A matricized tensor unfolded along the n^{th} mode is denoted by $\mathcal{X}_{(n)}$, which is composed of mode-N bers. For example, given the following frontal slices of tensor $\mathcal{X}$:

$$\mathcal{X}_1 = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \quad \mathcal{X}_2 = \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix};$$

its respective matricizations would be

$$\mathcal{X}_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}; \quad \mathcal{X}_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix};$$

$$\mathcal{X}_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}:$$

Several matrix and tensor products play an important role in Canonical Polyadic Decomposition, which is discussed in upcoming Section 2.2. The *Hadamard product* is an element-wise multiplication of matrices, denoted by $\mathbf{A} \quad \mathbf{B}$. Element $(i; j)$ of $\mathbf{A} \quad \mathbf{B}$ is determined by $\mathbf{A}(i; j) \quad \mathbf{B}(i; j)$. The *Kronecker product* [11] of matrices $\mathbf{A}\ 2\ \mathrm{R}^{I\ J}$ and $\mathbf{B}\ 2\ \mathrm{R}^{K\ L}$ is denoted by $\mathbf{A} \quad \mathbf{B}$. The resultant matrix $\mathbf{A} \quad \mathbf{B}$ is of size $(IJ) \quad (KL)$ and is de ned by:

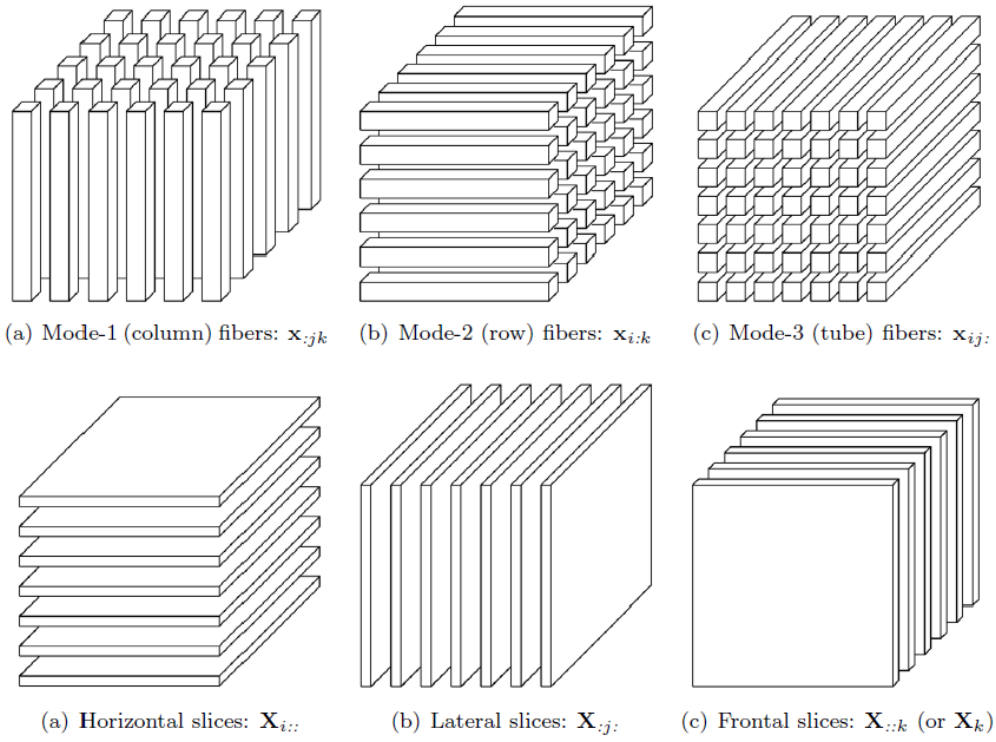

Figure 2.2: Third-order tensor bers and slices [11].

$$\mathbf{A} \quad \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix}$$

$$= [\mathbf{a}_1 \quad \mathbf{b}_1 \quad \mathbf{a}_1 \quad \mathbf{b}_2 \quad \mathbf{a}_1 \quad \mathbf{b}_3 \quad \cdots \quad \mathbf{a}_J \quad \mathbf{b}_{L\ 1} \quad \mathbf{a}_J \quad \mathbf{b}_L].$$

The *Khatri-Rao product* [24] is defined in terms of the Kronecker product. The Khatri-Rao product of matrices $\mathbf{A}^{I\ J}$ and $\mathbf{B}^{M\ J}$ is written as $\mathbf{A} \quad \mathbf{B}$:

$$\mathbf{A} \quad \mathbf{B} = [a_1 \quad b_1; a_2 \quad b_2; \ldots; a_n \quad b_n];$$

where the resulting matrix's size is $(IM) \quad J$.

The *outer product* [22] of two vectors is the product of the vector's elements, denoted by . We can calculate the outer product of two vectors $\mathbf{a}$ and $\mathbf{b}$ to produce matrix $\mathbf{X}$ given by:

$$\mathbf{X} = \mathbf{a} \quad \mathbf{b} = \mathbf{a}\mathbf{b}^T.$$

We can extend the outer product to a general tensor *outer product* [17]. Given three vectors a, b, and c, we can construct a third-order tensor $\mathcal{X}$ where each element is obtained by:

$$\mathcal{X}_{ijk} = a_i b_j c_k.$$

2.2 Canonical Polyadic Decomposition

Canonical Polyadic Decomposition, or CP, is a commonly used technique in tensor analysis. The goal of CP is to approximate a mode-N tensor $\mathcal{X}$ as the sum of R *rank-1* tensors. A tensor is *rank-1* if it can be written as the outer product of N

vectors, illustrated in Figure 2.3. If we are computing a rank R CP decomposition for a third-order tensor $\mathcal{X}^{I\ J\ K}$, we are mainly interested in solving for three factor matrices, which are the combination of the vectors from the rank-1 tensors: $\mathbf{A}\ 2\ \mathrm{R}^{I\ R}$, $\mathbf{B}\ 2\ \mathrm{R}^{J\ R}$, and $\mathbf{C}\ 2\ \mathrm{R}^{K\ R}$.

One way to achieve this is by using the Alternating Least Squares, or ALS, method. During each iteration of CP, one factor matrix is solved for, while the remaining matrices are xed. This process is repeated, taking turns solving for one factor matrix in an alternating fashion. For the three-way case, the ALS approach would x factor matrices $\mathbf{B}$ and $\mathbf{C}$ to solve for matrix $\mathbf{A}$:

$$\mathbf{A} = \min_{\mathbf{A}} k\ \mathcal{X}_{(1)} \quad \mathbf{A}(\mathbf{C} \quad \mathbf{B})^T\ k_F^2\ :$$

This problem is minimized by:

$$\mathbf{A} = \mathcal{X}_{(1)}[(\mathbf{C} \quad \mathbf{B})^T]^y;$$

but is usually preferred in the following form:

$$\mathbf{A} = \mathcal{X}_{(1)}(\mathbf{C} \quad \mathbf{B})(\mathbf{C}^T\mathbf{C} \quad \mathbf{B}^T\mathbf{B})^y: \tag{2.1}$$

This form only requires calculating the pseudo-inverse of an $R \quad R$ matrix, instead of $[(C \quad B)T]^y$, which results in a $(JK) \quad R$ matrix. This process is reapeated until an acceptable error threshold from the original tensor is reached or the maximum number of iterations is reached.

2.2.1 MTTKRP

In Equation 2.1, we need to note the calculation $\mathcal{X}_{(1)}(\mathbf{C} \quad \mathbf{B})$, which is called the Matricized Tensor Times Khatri-Rao Product, or MTTKRP. MTTKRP must be performed for each mode, every iteration of ALS, and is the typical bottleneck of CP.

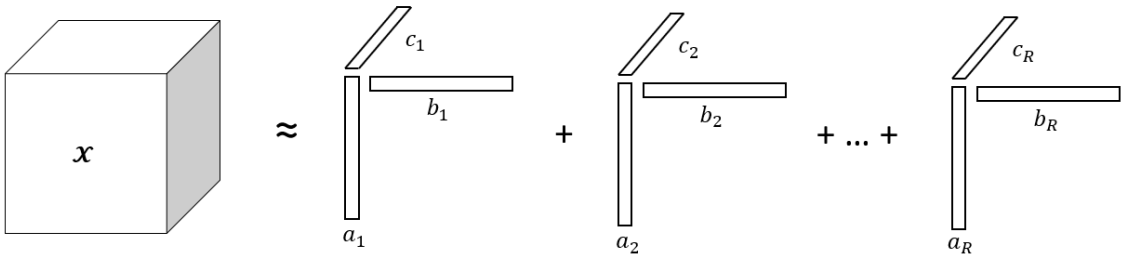

Figure 2.3: CP Decomposition of a third-order tensor [11].

Explicitly forming the product $\mathbf{C} \quad \mathbf{B}$ results in a dense matrix of size $(JK) \quad R$, which usually requires a massive amount of memory. Therefore, specialized MTTKRP algorithms are needed for dealing with sparse tensors. For instance, Algorithm 1 formulates MTTKRP as accumulated sparse tensor-vector products and only requires traversing each tensor element once, in any order.

2.3 Related Work

This section provides an overview of leading sparse tensor formats. Coordinate format, or COO, is regarded as the de-facto standard for sparse tensor storage. COO format was rst introduced in Kolda and Bader's 2009 Survey of Sparse Tensor Applications. COO stored tensor indexes tuples organized into either an ordered or unordered list, illustrated in Figure 2.4 [11]. COO's versatility and exibility has made it the favored format for applications like TensorFlow, MATLAB Tensor Toolbox, and sparse tensor repository FROSTT [25, 5, 23].

Hierarchical Coordinate (HiCOO) format organizes a sparse tensor into individual sparse tensor blocks. To convert a COO sparse tensor to HiCOO format, all elements are encoded and sorted using Z-Morton order, which is a space lling curve that maps multidimensional data to one-dimension [21]. The resulting mapping is called a Z-value, or *Morton code* [21]. An index's Morton code can be obtained by interleaving the individual bits of each index component, resulting in a unique integer representing its location on the curve. To better illustrate this process, the following example is provided. We begin with a COO sparse tensor in part (*a*) of Figure 2.5. Part (*b*) shows an equivalent tensor converted to a binary representation.

Next, the bits of individual index components must be interleaved, shown in part (*c*) in Figure 2.6. The result yields the values in columns *bi*, *bj*, and *bk*, which form a block index, and columns *ei*, *ej*, and *ek*, which indicate the location of that index

i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
2	2	2	6
3	0	1	7
3	3	2	8

Figure 2.4: A sparse tensor in COO format [14].

Algorithm 1: MTTKRP via Sparse Tensor-Vector products [24]

Input: $indI[M]$, $indJ[M]$, $indK[M]$, $vals[M]$ dense matrices $B^{J\ R}$, $C^{K\ R}$
Output: dense matrix $\mathbf{M}^{I\ R}$

```
1 for f=0 to F do
2 |   for z=0 to M do
3 |   |   t[z] = vals[z]  B(indJ[z]; f)  C(indK[z]; f) ;
4 |   end
5 |   for z=0 to M do
6 |   |   M(indI[z]; f) = M(indI[z]; f) + t[z] ;
7 |   end
8 end
```

within the block. If we read each index's Morton code as a binary representation of an integer, rows 5-8 have integer values of 34, 56, 37, and 62, respectively. To maintain ascending order, rows 6 and 7 must swap places. Indices are then partitioned using the block index bits by matching bit patterns. In part (*d*) of Figure 2.6, four distinct patterns of bits are partitioned into four blocks. The beginning index of each block is stored in a *bptr* array, indicated in part (*e*) of Figure 2.7. As a result, indices are compressed within their own block and require a smaller amount of bits. Additionally, there is no need to convert indexes back to COO format, since individual elements can be referenced using the block size B and block indexes. In Figure 2.7, the location of any index can be retrieved by using the formula $i = bi \quad B + ei; j = bj \quad B + ej;$ and $k = bk \quad B + ek$ [14].] Since blocks are limited to a pre-speci ed size, search space is greatly reduced when retrieving arbitrary values. Due to its partitioning scheme, the degree of compression o ered by this format is sensitive to how tensor elements are distributed, resulting in less compression for extremely sparse tensors [26].

Flagged Coordinate format (F-COO) is also derived from COO format and gears toward reducing COO's memory requirements as well as enabling uni ed tensor computations; that is, using a single storage format for tensor operations over multiple modes [15]. F-COO stores tensor mode indexes and values in separate arrays as a list, but also incorporates two additional arrays: a bit- ag (*bf*) array and start- ag (*sf*) array. These arrays serve the purpose of capturing changes in tensor computations. For example, part (*b*) in Figure 2.8 has replaced mode one with a *bf* array, where a change from 0 to 1 indicates a change in mode one. Within each partition, the *sf* array indicates if a new ber or slice begins within the partition. Note that the rst partition will always be 1, since it will always begin a new ber or slice. Instead of explicitly storing every mode like COO, F-COO achieves increased memory e ciency since it only stores the indices on the product mode and an entire mode is replaced by a smaller bit ag array. Other formats aim to further reduce COO's memory footprint, such as Compressed Sparse Fiber (CSF) format, introduced by Smith and

COO

i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
2	2	2	6
3	0	1	7
3	3	2	8

(a)

i	j	k	val
00	00	00	1
00	01	00	2
01	00	00	3
01	00	10	4
10	01	00	5
10	10	10	6
11	00	01	7
11	11	10	8

(b)

Figure 2.5: (*a*) COO tensor and (*b*) an equivalent table with all index components converted into their binary representation.

bi	bj	bk	ei	ej	ek	val
0	0	0	0	0	0	1
0	0	0	0	1	0	2
0	0	0	1	0	0	3
0	0	1	1	0	0	4
1	0	0	0	1	0	5
1	0	0	1	0	1	7
1	1	1	0	0	0	6
1	1	1	1	1	0	8

Swap rows

(c)

	bi	bj	bk
Block 1	0	0	0
	0	0	0
	0	0	0
Block 2	0	0	1
Block 3	1	0	0
	1	0	0
Block 4	1	1	1
	1	1	1

(d)

Figure 2.6: (*c*) Table of Morton codes obtained from the indexes in part (*a*) of Figure 2.5. (*d*) Partitioning indexes into blocks based on the number of unique bit patterns formed from *bi*, *bj*, and *bk*.

HiCOO

	bptr	bi	bj	bk	ei	ej	ek	val
B1	0	0	0	0	0	0	0	1
		0	0	0	0	1	0	2
		0	0	0	1	0	0	3
B2	3	0	0	1	1	0	0	4
B3	4	1	0	0	0	1	0	5
		1	0	0	1	0	1	7
B4	6	1	1	1	0	0	0	6
		1	1	1	1	1	0	8

(e)

Figure 2.7: Converted HiCOO tensor [14].

COO

i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
2	2	2	6
3	0	1	7
3	3	2	8

(a)

F-COO

	bf	j	k	val
sf[0]=1	1	0	0	1
	0	1	0	2
	1	0	0	3
	0	0	2	4
sf[1]=1	1	1	0	5
	0	2	2	6
	1	0	1	7
	0	3	2	8

(b)

Figure 2.8: (*a*) COO sparse tensor and (*b*) its equivalent representation in Flagged Coordinate (F-COO) format [14].

Karypis in 2015 [24]. CSF acts as an extension of Compressed Sparse Row (CSR) format, a common technique to compress two-dimensional matrices. To generalize this approach to an arbitrary number of dimensions, CSF organizes tensor slices into a list of bers into a tree-like structure, seen in part (*b*) in Figure 2.9. Each tensor mode corresponds to a level in the tree while the leaves represent non-zero values. Tracing a path from root to leaf builds the value's corresponding index. Duplicate indices are eliminated when a node is split into a sub-tree or a leaf, which results in all but one mode being compressed in the tree [26]. Since a tensor can be stored in along any mode, you can create multiple representations of the same sparse tensor. This property quali es CSF as a *mode-speci c* format, which can impact performance, depending on which mode a tensor operation must be calculated. Therefore, it may be necessary to store additional representations of the same tensor.

The current state-of-the-art is known as Adaptive Linearized Storage of Sparse Tensors (ALTO), proposed by Helal et. al. in 2021 [8]. ALTO stores nonzero elements along a line using a bit mask based on mode cardinalities. The bit mask is based on the number of bits required to represent each mode, encoding the modes from right to left for the smallest mode rst while interleaving the bits for the larger modes. The nal bit mask is used to generate the index's unique position in its ALTO tensor.

Each nonzero, represented in part (*a*) in Figure 2.10 as a dark rectangle, corresponds to its ALTO counterpart in part (*c*), indicated by a dotted line. Mapping COO entries to the ALTO tensor requires the use of the ALTO bit mask in part (*b*), which is generated from the cardinalities of the tensor modes. The tensor's modes in part (*a*) is 4 4 3, or modes i,j, and k, respectively. Each dimension requires two bits to be written in binary, so the total number of bits required to represent all modes is 6, or the length of the ALTO bit mask. Starting from the least signi cant bit on the right, the bit for the k mode is encoded, since it is the smallest mode, then the i and j modes, which are the same size. Therefore, the corresponding bit mask $b_{i;1}; b_{j;1}; b_{k;1}; b_{i;0}; b_{j;0}; b_{k;0}$ is used to both linearize COO entries and de-linearize an ALTO tensor's compressed indexing metadata.

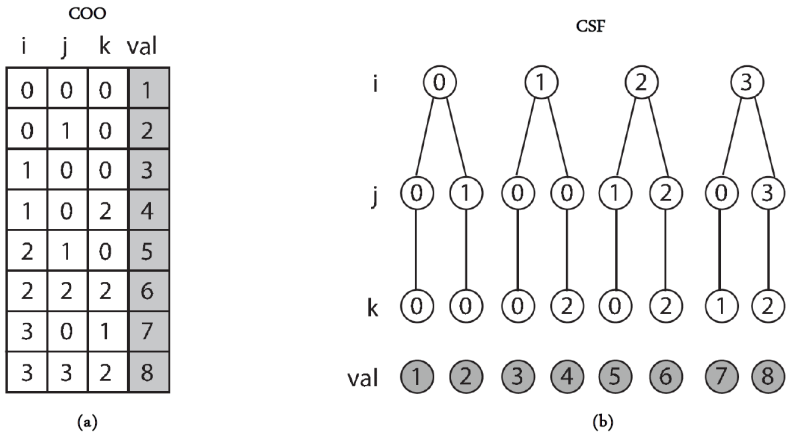

Figure 2.9: (*a*) COO format sparse tensor and (*b*) its equivalent representation in Compressed Sparse Fiber (CSF) format over mode *i* [14].

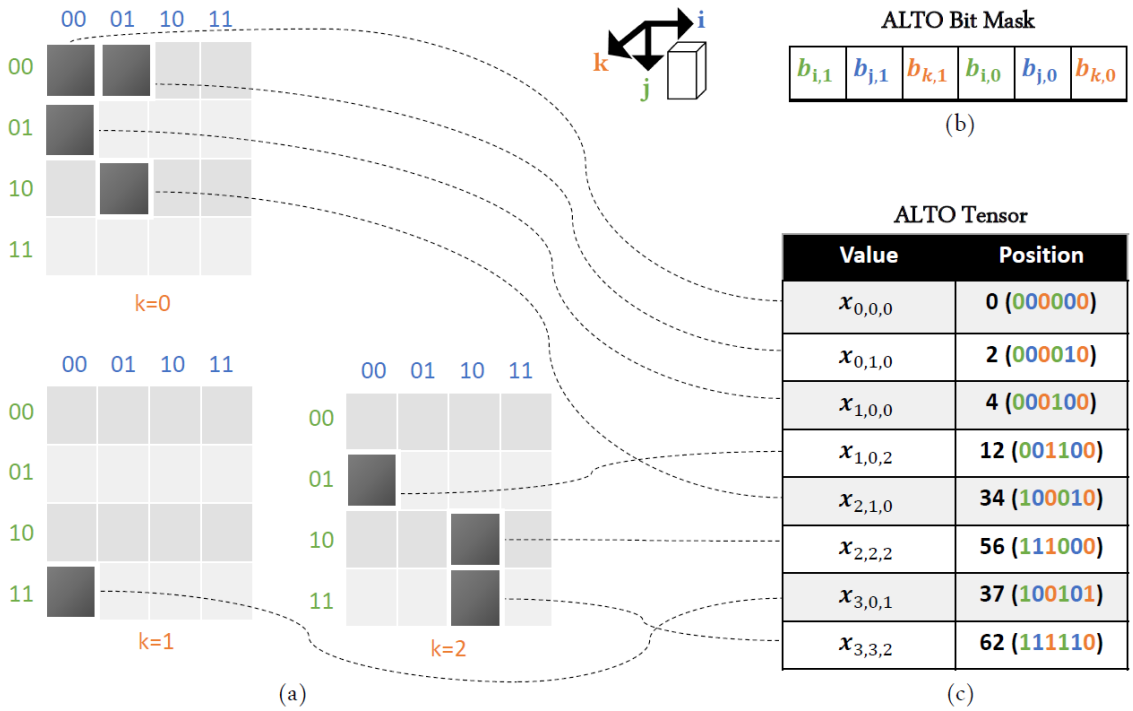

Figure 2.10: Converting a COO tensor to an ALTO tensor.

2.4 HaCOO Format

Hashed Coordinate format, or HaCOO, is a novel sparse tensor storage format introduced by Lowe et al. in 2021, and is this paper's primary focus [16]. Table 2.2 contains helpful terms and notation used when discussing this format. HaCOO uses a separate chaining hash table to store sparse tensor indices. Each entry in the table contains two elds, the index's Morton encoding and its corresponding value. Similar to HiCOO, HaCOO uses *z*-order mapping in conjunction with a hashing function to map COO indexes to a hash key. Pseudocode to obtain an index's Morton encoding using bit masking is shown in Algorithm 2 [16].

To further reduce the collision rate, another function is applied to uniformly distribute values across the hash table. This algorithm draws inspiration from Bob Jenkins' One-at-a-time hashing function, which was originally published in the Dr. Dobbs Journal [10]. Algorithm 3 determines three values, *sx*, *sy*, and *sz*, which are required by Algorithm 4 [16]. These were scaled from the original 32-bit Jenkins hash to account for an arbitrary number of bits. Algorithm 4 uses a series of bitwise operations to mix the bits of the Morton code. This results in index values being distributed uniformly along the hash table.

The nal step is to constrain the hash key to the size of the table by applying the modulus operation. If an index maps to an already lled slot, the new entry is appended to the end of that slot's list. This occurrence is called a *hash collision*. To allow for fast modulus operations via bit-masking, the size of the hash table always grows by a power of two. If the number of entries exceeds the table's load factor, or the maximum percent of buckets that can be occupied without increasing the number of buckets in the table. If the number of non-zero entries is known in advance, the hash table's size can be determined using:

$$T_{size} = 2^{d\log(M/load\_factore)};$$

Table 2.2: Table of HaCOO algorithm notation.

Symbol	Description
nbuckets	Number of buckets/slots in the hash table
bits	Number of bits
mask	Bit mask
morton(i)	Morton encoding of index *i*
j	Logical OR
&	Logical AND
	Logical XOR
	Bitwise left shift
	Bitwise right shift
dxe	Ceiling of x
$max_{x;y}$	Maximum value of x and y

Algorithm 2: Morton Encoding Function

Input: Tensor index as a list of non-negative integers *vals*
Output: Morton encoding of the index *result*

```
1  result = 0
2  n = length(vals)
3  /* set the right most bit to act as as a bit mask */ bit = 1
4  /* loop to interleave the individual bits of each integer into \result" */
5  while sum(vals) 6= 0 do
6      for i in range(n) do
7          /* use the masking bit to copy one bit from \value" into \result" */
8          if vals [i] & 0x1 not equal 0 then
9              result = result j bit
10         end
11         /* shift the integer to the right once and masking bit to the left to get
           the next bit from the value */
12         vals [i] = vals [i]     1
13         bit = bit     1
14     end
15 end
```

Algorithm 3: Hash Values

Input: number of buckets in hash table *nbuckets*

Output: *sx*, *sy*, *sz*, *mask*

1 $bits = \log_2(nbuckets)$

2 $sx = \text{ceil}(bits = 8)$ 1

3 $sy = 4$ sx- 1

4 $sz = \text{ceil}(bits = 2)$ 1

5 $mask = nbuckets$-1

Algorithm 4: Hash Algorithm

Input: list of non-zero integers *index*

Output: morton value *m*, hash key *k*

1 $m = \text{morton}(index)$

2 $hash = hash + hash$ sx

3 $hash = hash$ $hash$ sy

4 $hash = hash + hash$ sz

5 $k = hash \ \& \ mask$

Table 2.3: Hashing parameters calculated using Algorithm 3 given the COO tensor in Figure 2.11.

.

nbuckets = 16	*sx = 0*
bits = 4	*sy = 1*
mask = 15	*sz = 2*

Figure 2.11 illustrates the process of converting a COO tensor to HaCOO format. Table 2.3 provides hashing parameters and Table 2.4 displays the intermediate steps of obtaining each index's hash key.

2.5 Motivation

Despite improvements over the past decade, the memory reduction o ered by several formats can be heavily a ected by the spatial distribution of elements within a sparse tensor. Furthermore, formats such as HiCOO, F-COO, and ALTO, do not support updates or insertion of new elements without rebuilding the tensor from scratch. Each of these formats must rst sort all nonzero tensor elements before partitioning them into its respective data structure. Adding new elements could potentially a ect ALTO's bit mask, which relies on mode cardinalities, or alter HiCOO's partitioning of its sparse tensor blocks.

Furthermore, sorted COO requires some additional steps to insert a tensor index. In relation to the number of tensor nonzeros (M), a binary search of complexity $O(log(M))$ is required to nd the index insertion location, then $O(M)$ to perform an in-order insertion [26]. In comparison, HaCOO maintains an amortized $O(1)$ constant insertion and retrieval time [16]. This opens up new possibilities for \on the y" sparse tensor updates. Another issue is that several common sparse tensor formats are implemented in standalone libraries; CSF in The Surprisingly ParalleL spArse Tensor Toolkit (SPLATT), HiCOO in A Parallel Tensor Infrastructure! (ParT!), ALTO in ALTO, and so on. Although these libraries can be extremely bene cial, setting up multiple computing environments to handle several standalone libraries can be an added burden on the user. Tew's study of sparse tensor formats observes that creating a \master" library would be an incredibly massive task to undertake, as well as present the issue of maintaining a vast repository of code. Regardless, Tew asserts that consolidating these formats into one interface would be worthwhile to increase accessibility and provide quality of life improvements [26].

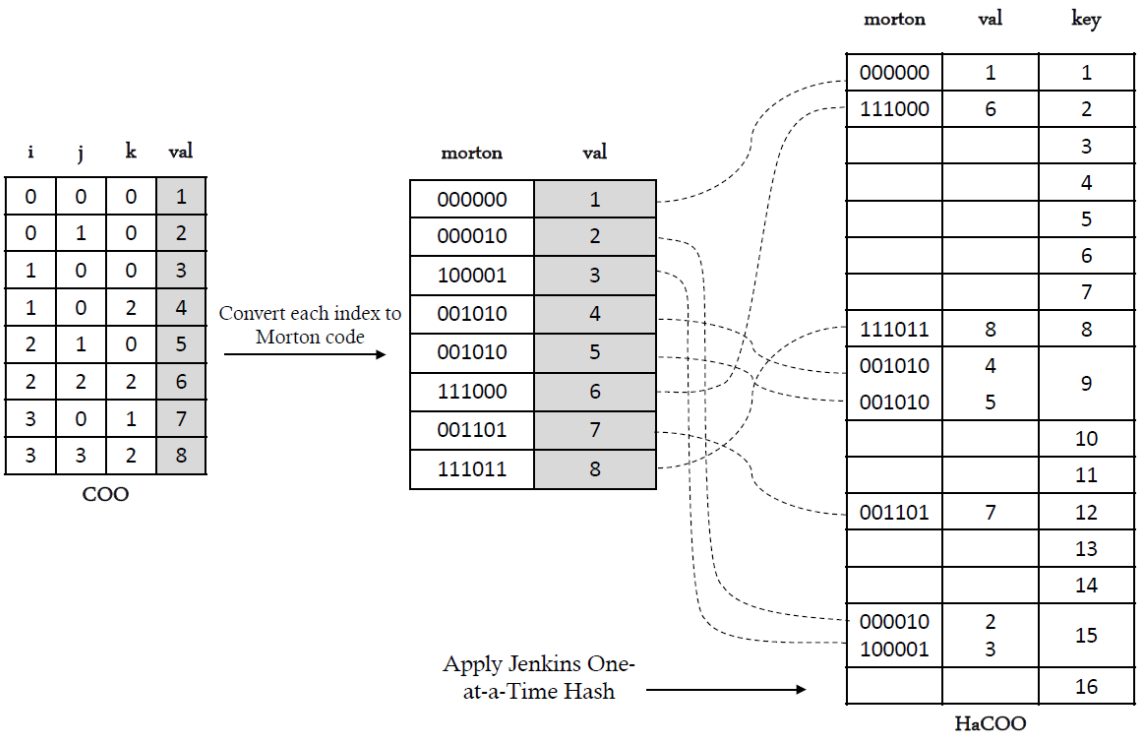

Figure 2.11: Converting a COO tensor to HaCOO format.

Table 2.4: Applying the Jenkins One-at-a-time hash function to COO indexes. Columns 2-6 indicate the result after executing lines 1-5 of Algorithm 4.

idx	morton	step 2	step 3	step 4	k
0, 0, 0	000000	0000000	0000000	000000000	0
0, 1, 0	000010	0000100	0000110	000011110	14
1, 0, 0	000001	0000010	0000011	000001111	15
1, 0, 2	100001	1000010	1100011	111101111	15
2, 1, 0	001010	0010100	0011110	010010110	6
2, 2, 2	111000	1110000	1001000	101101000	8
3, 0, 1	001101	0011010	0010111	001110011	3
3, 3, 2	111011	1110110	1001101	110000001	1

Presently, we do not have such a master library. These considerations motivated the decision to implement HaCOO in MATLAB. Additionally, a library to use and manipulate dense and sparse tensors already exists for the application, known as Tensor Toolbox. The library contains classes to store both dense and sparse tensors, manipulation methods, tensor arithmetic, as well as implementations for common tensor decomposition algorithms. By integrating the new HaCOO class into an already existing a suite of tools, users can bene t from a variety of sparse tensor format options.

2.6 FROSTT Tensors

This study uses a number of tensors from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT), a publicly available collection of various data to facilitate reproducible sparse tensor research. A summary of tensor characteristics are provided in Table 2.5. The data sets are summarized as follows:

The *Uber* tensor contains data on Uber pickups in New York City from April 2014 through August 2014. Modes represent *dates-hours-latitudes-longitudes*.

The *NELL-2* tensor is a smaller version of *NELL-1* with most sparse indices pruned. *NELL-1* is pulled from the Never Ending Language Learner knowledge base, part of a machine learning project from Carnegie Mellon University [3]. Non-zeros represent *entity-relation-entity* tuples.

The *Enron* tensor is email data that was publicly released during an investigation by the Federal Energy Regulatory Commission. The modes represent *sender-receiver-word-date*.

The *Chicago* tensor contains data on crime reports in the city of Chicago, where modes represent *day-hour-community-crimetype*, where a community is one of the communities of Chicago.

The *NIPS* tensor contains publications from the NeurIPS Conference on Neural Information Processing Systems from 1987 to 2003. Modes represent *paper-author-word-year*.

The *LBNL* tensor contains ten days of anonymized internal network tra c from Lawrence Berkeley National Laboratory (LBNL)/International Computer Science Institute (ICSI). The modes are *sender IP-sender port-destination IP-destination port-time*, and the values are the total packet length sent in a timestep (one second).

Table 2.5: Characteristics of FROSTT sparse tensors [23].

tensor	M	dimensions					storage
uber	3.3M	183	24	1.1K	1.7K		52.9 MB
nell-2	76.9M	12.1K	9.2K	28.8K			1.51 GB
enron	54.2M	6K	5.7K	1.2K			1.2 GB
chicago	5.3M	6.2K	24	77	32		80 MB
nips	3.1M	2.5K	2.9K	14K	17		58.9 MB
lbnl	1.7M	1.6K	4.2K	1.6K	4.2K	868.1K	55.1 MB

Chapter 3

Approach

3.1 Modified HaCOO Hashing

HaCOO's hashing algorithm initially converts each sparse tensor index to its corresponding Morton code by interleaving the index's bits, then applies the Jenkins' One-at-a-Time hash to obtain its hash key. Unlike the original algorithm, we found that omitting the Morton encoding step, concatenating all index components into one integer, then applying the One-at-a-Time hash was sufficient for maintaining a low collision rate and decreasing the maximum probe depth. Several FROSTT tensors were used for evaluating both hash functions. Dataset characteristics are provided in Section 2.6. Relevant evaluations are presented in Tables 3.1 and 3.2 to compare the original algorithm and the modified algorithm. Columns represent the tensor, collision rate, mean probe depth within a chain, maximum probe depth within a chain, and the mode and median number of entries per bucket. These results motivated the choice to implement the modified hash function in the final MATLAB HaCOO class. Instead of storing each index's Morton code, indexes were stored explicitly.

Table 3.1: Hashing statistics using the original hash algorithm.

Tensor	Collision Rate	Mean Probe Depth	Median Probe Depth	Mode Probe Depth	Max Probe Depth
uber	16.43%	1.20	1	1	7
nell-2	26.45%	1.36	1	1	11
enron	37.53%	1.60	1	1	37
chicago	57.27%	25.95	2	1	28
nips	77.31%	4.41	4	4	29
lbnl	89.39%	9.43	1	1	2994

Table 3.2: Hashing statistics using the modified hash algorithm.

Tensor	Collision Rate	Mean Probe Depth	Median Probe Depth	Mode Probe Depth	Max Probe Depth
uber	17.17%	1.21	1	1	7
nell-2	23.87%	1.31	1	1	9
enron	17.74%	1.22	1	1	8
chicago	26.23%	1.36	1	1	8
nips	16.40%	1.20	1	1	7
lbnl	17.13%	1.21	1	1	7

3.2 Implementation

In an effort to exploit HaCOO format within a familiar computational platform, HaCOO format was implemented as a MATLAB class called *htensor*. The primary goals for this library were to create and manipulate HaCOO format sparse tensors in MATLAB and perform CP decomposition using sparse tensors in HaCOO format.

A HaCOO sparse tensor in the *htensor* MATLAB class contains a hash table, represented as a cell array, with individual cells containing a matrix of index-value tuples that have hashed into that bucket. Tensor elements that hash into an already occupied bucket are concatenated vertically with the existing matrix. There are three variations of class constructors. The single argument constructor can accept a COO format *.txt* file or a nonzero scalar value to create a blank table with a specified number of buckets. The two argument constructor accepts two arrays containing tensor indexes with corresponding values and will return a populated HaCOO tensor. The three argument constructor accepts the same parameters as the two argument constructor, but with an additional third array of indexes which have already been concatenated. If no parameters are specified, the default constructor creates an empty hash table of an arbitrarily chosen size of 512. If the load factor threshold is met, the table with automatically double in size and rehash all existing entries. Additional functions required for CP-ALS decomposition were implemented by directly replacing sections of code that referenced Tensor Toolbox's *sptensor* class with *htensor* custom functions.

The HaCOO *htensor* class includes the following functions:

set - Insert a nonzero entry in the hash table.

get - Retrieve a tensor index.

search - Search for an index entry in hash table.

extract_val - Retrieve the value of tensor index.

hash - Hash the index and return entry's hash key.

rehash - Rehash existing entries to a new tensor of with a hash table double the existing size.

all_subsVals - Return all indexes and values in the tensor as separate arrays.

display_htns - Display all sparse tensor indexes and values.

I/O functions to write, load, and save a HaCOO tensor to a *:mat* le.

nnzLoc - Return the indexes of nonempty hash table buckets.

htns_mttkrp - Perform MTTKRP along a speci ed mode.

htns_cp_als - Perform CP decomposition using the Alternating Least Squares method.

Currently, HaCOO's borrows its MTTKRP implementation from version 3.3 of Tensor Toolbox. Instead of retrieving indexes and values from a COO *sptensor*, all indexes and values are extracted at once from a HaCOO *htensor*. The Tensor Toolbox implementation was chosen to fully exploit MATLAB's vectorized operations, since methods that accumulate values using *for* loops resulted in far worse performance. The remainder of the implementation was kept the same. Full details of this formulation of MTTKRP can be found at [5]. All code for the HaCOO library is accessible from its GitHub Repository [7]. Select class functions are provided in the Appendix.

3.3 Storage

HaCOO's hash table dominates the required storage of the entire HaCOO tensor, as the additional elds only contain scalars. Assume a numeric value of desired precision (single or double) takes $_{num}$ bytes, a cell array containing one numerical

value takes $_{bucket}$ bytes, and an empty cell array bucket consumes 0 bytes. Note that MATLAB stores numbers as oating-point values with double precision if the type is not speci ed, requiring 8 bytes, while the single type requires only 4 bytes [19]. A load factor must be speci ed, which is the percentage of hash buckets that can be occupied. Storing the hash table of a HaCOO sparse tensor $\mathcal{X}^{I_1 \cdots I_N}$ of order N with M nonzeros would require

$$S_{HaCOO} = load\_factor \quad M \quad _{bucket} \quad N$$

bytes, while COO format would only require

$$S_{COO} = N \quad M \quad _{num}$$

.

While COO requires less storage, HaCOO's constant time retrieval and insertion makes up for the tradeo of increased memory.

Chapter 4

Evaluations

This chapter presents evaluation results for HaCOO and COO format using several tensors from The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [23]. Tensor characteristics and properties are detailed in Section 2.6. All results are reported as an average over 10 trials.

4.1 Building FROSTT Tensors using HaCOO

Before evaluating the time required for both COO and HaCOO format's ability to insert new tensor entries, a few adjustments needed to be made. FROSTT's tensors are stored in COO format, presorted, and verified to contain no duplicate indices. To provide a comparison of solely the insertion step between the two formats, all rows of the original COO tensor were shuffled, then inserted one entry at a time, effectively inserting a random element from that tensor. Due to time constraints, re-building entire tensors in COO format was not possible, since time estimates indicated that rebuilding one COO tensor could take several days. As a result, inserting increasingly larger subsets of elements were tested. Results are presented as average wall-clock and CPU time in seconds required to build COO and HaCOO tensors using MATLAB's native functions for measuring code performance [18], accumulating only the time for element insertion. Wall-clock time was measured with MATLAB's *tic/toc* and *timeit*

functions, which return the elapsed time it took for the code to run. The *cputime* function measures the total CPU time summed across all threads. MATLAB's method to measure CPU time can exceed wall-clock time in certain scenarios, such an example being MATLAB using all processing cores equally. Results were obtained from an Apple MacBook Pro with a 2.4 GHz Dual-Core Intel Core i5 processor with 128 GB and 4 GB of 1600MHz DDR3L onboard memory [1].

Figures 4.1 and 4.2 present cumulative average wall-clock and CPU time for inserting 100,000 random elements respectively for all sample FROSTT tensors. HaCOO format began to consistently outperform COO once the number of elements inserted reached 25,000. Inserting 100,000 random elements from the Uber tensor using HaCOO format yielded around 91-93% reduction in both cumulative wall-clock and CPU time compared to using COO format. Since increasingly larger intervals of n were tested, Figure 4.3 provides a clearer insight into how the number of inserted elements a ected both formats, reporting the percent decrease in seconds required to insert n random elements from the Uber tensor using HaCOO format versus COO format. Note how a logarithmic trend line provided the best t for the data. As n grows, the time reduction from using HaCOO format appears to plateau as it asymptotically reaches 100% reduction. This is expected behavior, since reaching that amount of reduction is unfeasible, as that would mean it took no time at all to construct the tensor in MATLAB.

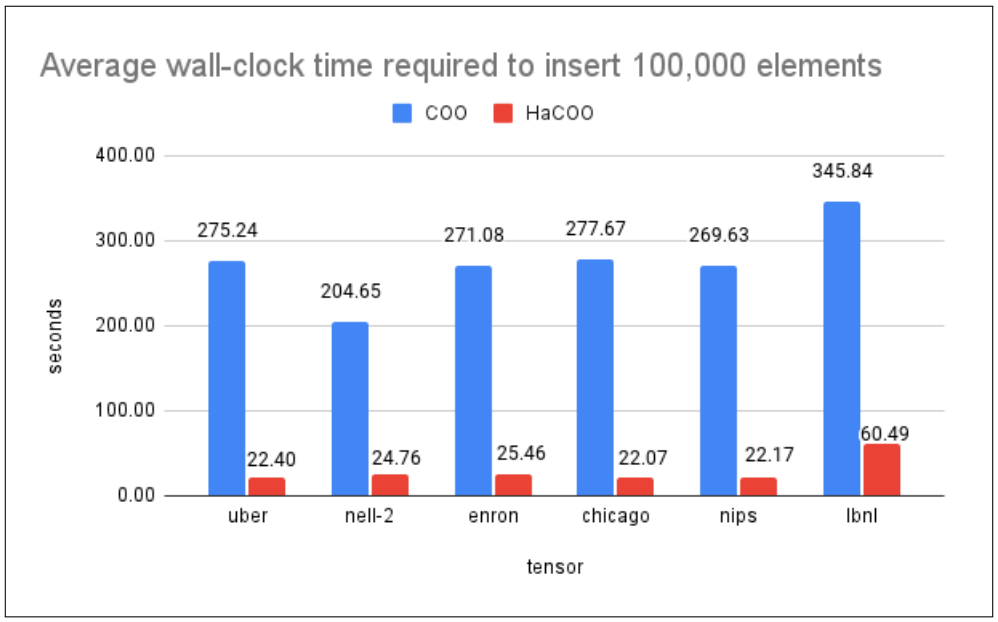

Figure 4.1: Average wall-clock time in seconds required to insert 100,000 random elements using COO format compared to HaCOO format.

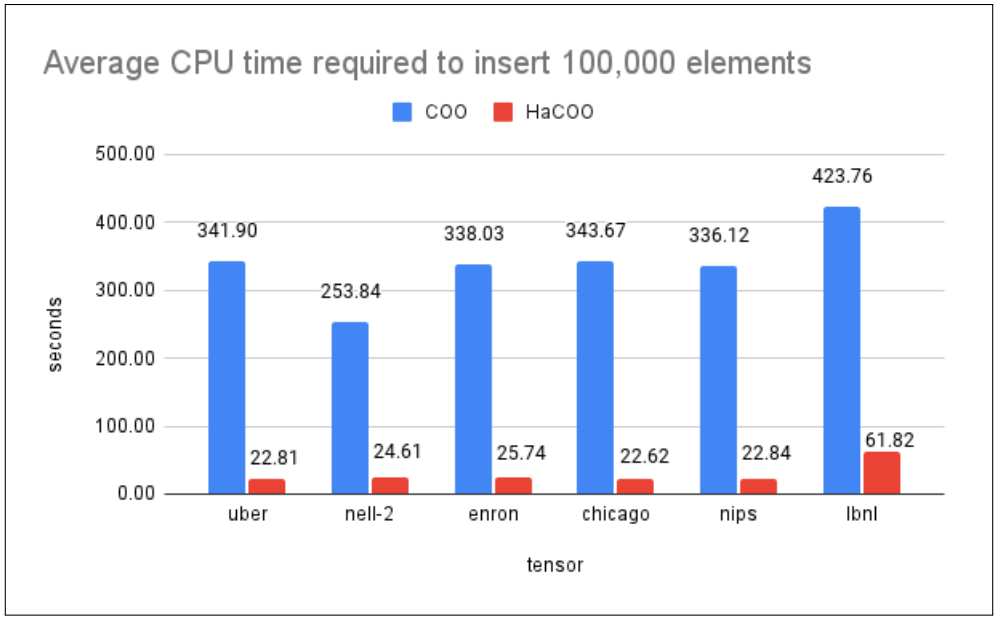

Figure 4.2: Average CPU time in seconds required to insert 100,000 random elements using COO format compared to HaCOO format.

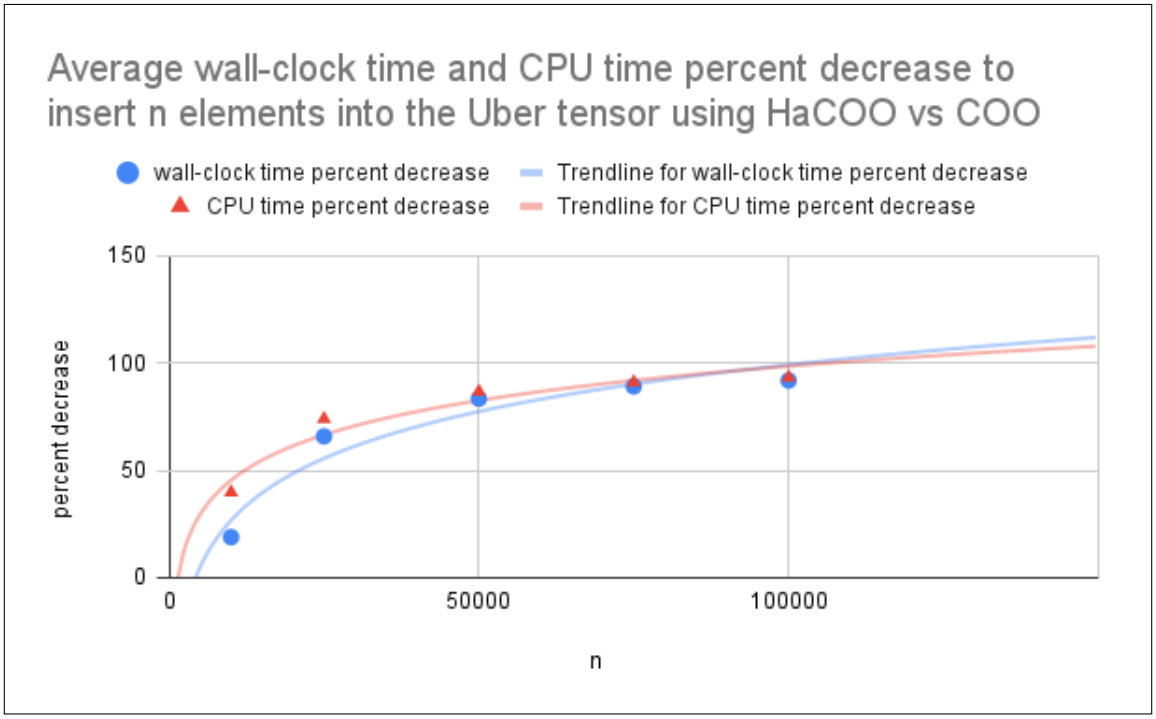

Figure 4.3: Percent decrease in seconds required to insert n random elements from the Uber tensor using HaCOO format compared to using COO format.

4.2 MTTKRP

Figures 4.4 and 4.5 report average wall-clock and CPU time required to calculate MTTRKP over all modes. The black data labels above the bars represent the average time over all modes. The *enron* and *nell-2* tensors were omitted from evaluations due to MATLAB's limited memory usage. While the MTTRKP calculation is identical for both formats, there is a slight di erence due to the time required to retrieve all tensor elements from HaCOO's hash table, which was implemented by indexing the table's nonempty cells, which act as buckets. On the other hand, this is a step that COO format can completely bypass. On average, HaCOO's current MTTRKP method incurs a 26.78% increase in time. Out of the chosen datasets, the largest overhead observed was calculating MTTKRP over mode 4 of the *chicago* tensor, which had an 77.88% increase in time to complete. Despite this percent increase, the maximum di erence in elapsed time over any mode was slightly over 4 seconds.

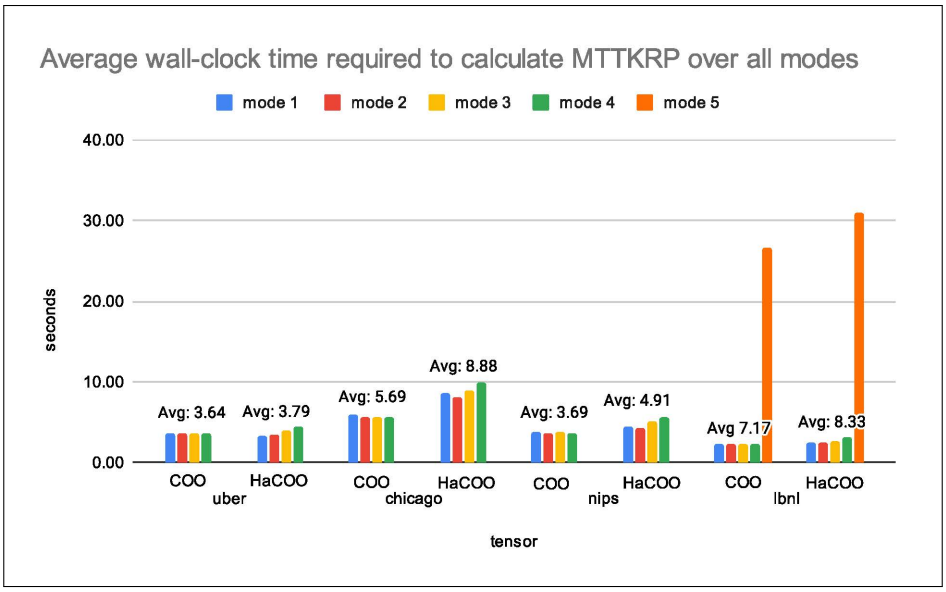

Figure 4.4: Average wall-clock time in seconds required to calculate MTTKRP over all tensor modes using COO format compared to HaCOO format. The black data labels above the bars indicate average time over all modes.

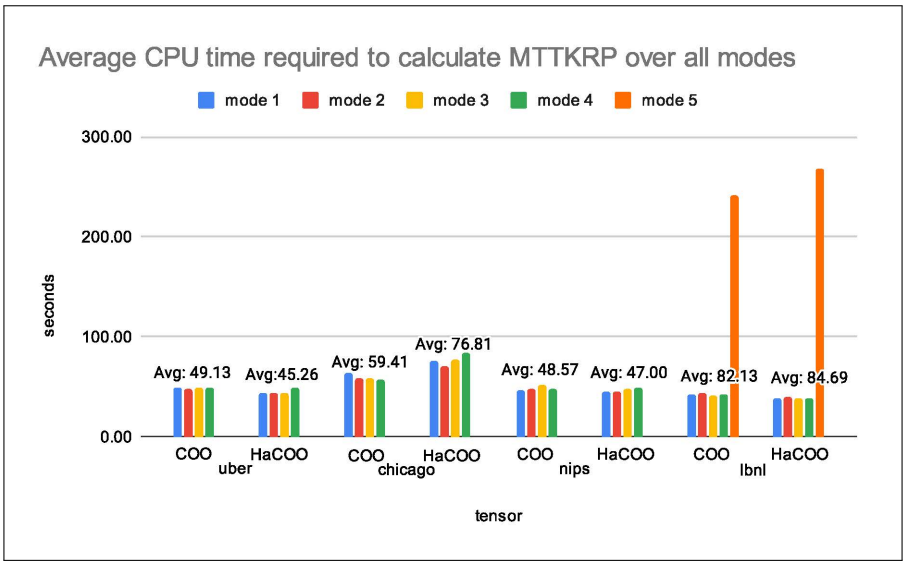

Figure 4.5: Average wall-clock time in seconds required to calculate MTTKRP over all tensor modes using COO format compared to HaCOO format. The black data labels above the bars indicate average time over all modes.

Chapter 5

Modeling Textual Influence

Sparse tensor decomposition can also play a central role in text analysis. This chapter focuses on the use of sparse tensors in Lowe's Textual In uence model, originally published in 2018. The model uses non-negative sparse tensor decomposition on document tensors to discreetly quantify the amount of in uence a collection, or *corpus*, of written documents exerts on a target document.

A document tensor is built by counting the frequency of the document's n-grams. An n-gram is a contiguous sequence of words and is constructed using a sliding window, shown in Figure 5.1 [17]. Each word in the n-gram corresponds to a unique index in a vocabulary list V, which contains all unique words in the corpus and is consecutively indexed. Thus, the frequency of the phrase $V_i V_j V_k$ in a document is represented as entry $\mathcal{D}_{i;j;k}$ in its respective document tensor. Pseudocode for building a document tensor is shown in Algorithm 5. This process is repeated for each document in the corpus. It follows that the size of the vocabulary V, drives the dimensions of a document tensor, since each document tensor must account for any combination of n words. However, since only speci c combinations of n-words hold any semantic meaning, the nal document tensor is very sparse. For instance,

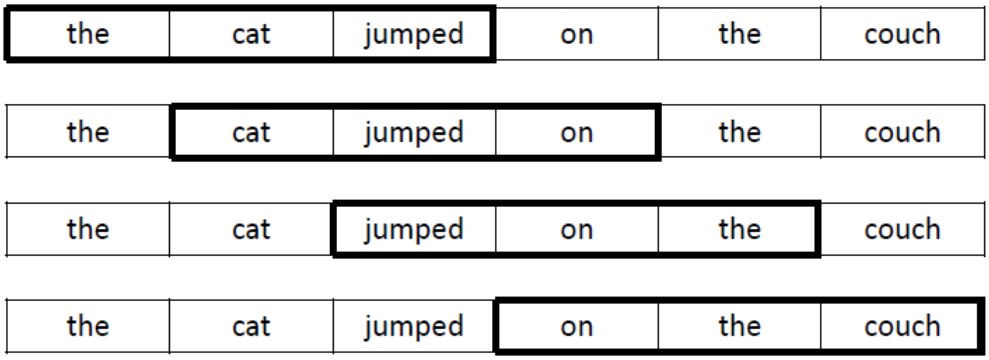

Figure 5.1: Counting *n*-grams using a sliding window, where *n* is set to 3.

Algorithm 5: Building a document tensor [17].

Input: text document *d*, length of *n*-gram *n*, vocabulary list *V*
Output: tensor $\mathcal{D}$

```
D = Tensor with dimension |V| |V| ... |V| (n times)
len = number of words in d
for i = 1 to len do
    /* Compute tensor element index */
    for j = 1 to n do
        index[j] = index of word d[i] in V
    end
    /* Update frequency of this n-gram */
    D [index] = D [index] + 1
end
return D
```

a document tensor for the story in Figure 5.2 could potentially have a nonzero entry for the n-gram *on then cat*". However, this combination of words would likely not occur together, since it holds no semantic meaning. If the phrase did not occur in the document, its corresponding tensor entry would have a value of zero. Since the number of potential phrases that do not occur far outweigh the number of phrases that do occur, document tensors tend to be sparse.

After building and decomposing all document tensors, the factors yielded by non-negative CP decomposition are reassembled, then L_1 distances between all factors are calculated and stored as a matrix. Finally, each factor in the target document is assigned the closest factor, given it meets a certain threshold, indicating the amount of in uence that factor exerted on that document. Since this part of the model is not a ected by tensor storage format, we will only focus on building and decomposing document tensors. A full discussion can be found in [17].

5.1 A Small Example

To illustrate the process of building a document tensor, this section covers a trivial example. A sample document is provided in Figure 5.2 and a corresponding indexed vocabulary of unique words is shown in Table 5.1. By referencing this vocabulary, we can build indexes for all n-grams in the document, shown in Table 5.2. Table 5.3 represents the document from Figure 5.2 in COO format. Since none of the n-grams occurred more than once, all values representing an n-gram's frequency are assigned a value of one.

5.2 Building Document Tensors in MATLAB

To evaluate the HaCOO MATLAB class, a script was required to build the document tensors. A number of optional parameters can be speci ed, including n-gram size, vocabulary size (the default set to 10^4), and an option to save the vocabulary to a

The cat jumped on the couch. He yawned and stretched. Then he fell asleep.

Figure 5.2: Sample short story document.

Table 5.1: Extract vocabulary V consisting of all unique words in the corpus.

1	the	7	yawned
2	cat	8	and
3	jumped	9	stretched
4	on	10	then
5	couch	11	fell
6	he	12	asleep

Table 5.2: List of n-grams with corresponding indices.

1, 2, 3	the cat jumped	6, 7, 8	he yawned and
2, 3, 4	cat jumped on	7, 8, 9	yawned and stretched
3, 4, 1	jumped on the	8, 9, 10	and stretched then
4, 1, 5	on the couch	9, 10, 6	stretched then he
1, 5, 6	the couch he	10, 6, 11	then he fell
5, 6, 7	couch he yawned	6, 11, 12	he fell asleep

Table 5.3: Document tensor in unsorted COO format.

i	**j**	**k**	**value**
1	2	3	1
2	3	4	1
3	4	1	1
4	1	5	1
1	5	6	1
5	6	7	1
6	7	8	1
7	8	9	1
8	9	10	1
10	6	11	1
6	11	12	1

le. The script creates document tensors from all *.txt* files in the current directory. For each document, the script disregards all numbers and punctuation and converts all words to lowercase. Punctuation is filtered out using MATLAB's Text Analytics Toolbox [20]. All unique words are stored in a MATLAB Map, a container that maps unique keys to values [27]. This Map is sorted in descending order by frequency, and consecutively indexed. In cases where a constrained vocabulary is desired, this list can be truncated to a specific length. Individual tensor entries are built using a sliding window to locate an n-gram, referencing the Map for each word's index, and are combined to create the full tensor index. This entry is then inserted into a sparse tensor. The window then advances by one word until we have exhausted all possible n-grams, where we must stop n words from the end of the document. To avoid rehashing in HaCOO format, the initial number of buckets was specified to be 1,048,576, or 2^{20}. The full script can be found in the Appendix.

5.3 Datasets

Two collections of documents were tested. The smaller corpus was mentioned in [17], which we will call the *Conference* corpus. The collection consists of five reports on the topic of handwritten digit recognition with an additional two papers discussing unrelated topics, totaling 45,152 words with a 5,236-word vocabulary. The second corpus consists of seven works by William Shakespeare, which we will refer to as the *Shakespeare* corpus, totaling 181,760 words with a 15,203-word vocabulary. Works were obtained from the Project Gutenberg website [2]. Tables 5.4 and 5.5 provide a complete list of documents for both corpuses.

Table 5.4: Documents in the Conference corpus.

Num	Document Information
1	Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In Proc. DMKD 2003, pages 211. ACM Press, 2003.
2	Andreas Schlapbach and Horst Bunke. Using hmm based recognizers for writer identifcation and verifcation. In Proc. FHR 2004, pages 167172. IEEE, 2004.
3	Yusuke Manabe and Basabi Chakraborty. Identity detection from on-line handwriting time series. In Proc. SMCia 2008, pages 365370. IEEE, 2008.
4	Sami Gazzah and Najoua Ben Amara. Arabic handwriting texture analysis for writer identifcation using the dwt-lifting scheme. In Proc. ICDAR 2007, pages 11331137. IEEE, 2007.
5	Kolda, Tamara Gibson. Multilinear operators for higher-order decompositions. 2006
6	Blei, David M and Ng, Andrew Y and Jordan, Michael I. Latent dirichlet allocation. 2007
7	Serfas, Doug. Dynamic Biometric Recognition of Handwritten Digits Using Symbolic Aggregate Approximation. Proceedings of the ACM Southeast Conference 2017

Table 5.5: Documents in the Shakespeare corpus.

Num	Document Information
1	\Hamlet, Prince of Denmark by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1524. Accessed 10 July 2023.
2	\Julius Caesar by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1522. Accessed 10 July 2023. |
3	\Macbeth by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1533. Accessed 10 July 2023. |
4	\A Midsummer Night's Dream by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1514. Accessed 10 July 2023. |
5	\Othello, the Moor of Venice by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1531. Accessed 10 July 2023. |
6	\The Tragedy of Romeo and Juliet by William Shakespeare." Project Gutenberg, Nov. 1997, www.gutenberg.org/ebooks/1112. Accessed 10 July 2023. |
7	\Twelfth Night; Or, What You Will by William Shakespeare." Project Gutenberg, Nov. 1998, www.gutenberg.org/ebooks/1526. Accessed 10 July 2023. |

Figures 5.3 and 5.4 present wall-clock time and CPU time in seconds to build all document tensors in the Conference corpus in HaCOO and COO format, then use their respective CP-ALS method to decompose all document tensors. Reported times are an average over 10 trials. Note that during the tensor building stage, reported wall-clock and CPU time only considers the time required to insert elements into the document tensors. The *n*-gram size was set to three and 50 components were specified while invoking the CP-ALS method. Document tensors were built using both constrained and unconstrained vocabularies to observe how vocabulary growth affected element insertion time. The constrained case limited the corpus vocabulary to the 600 most frequent words, and the unconstrained vocabulary accounted for all unique words in the corpus. Results indicated around 44-49% reduction in both wall-clock and CPU time from using HaCOO format as opposed to COO format for both the constrained and unconstrained cases.

Figures 5.5 and 5.6 present average wall-clock and CPU required to build and decompose all document tensors into 50 components for the Shakespeare corpus using the CP-ALS method. Since this corpus had a far larger word and vocabulary count, a more substantial difference was observed between the two formats. The constrained case yielded in around a 14% decrease in wall-clock time, while the unconstrained case yielded around a 72% decrease. With regard to CPU time, around 32% and 78% decrease was observed for the constrained and unconstrained case, respectively.

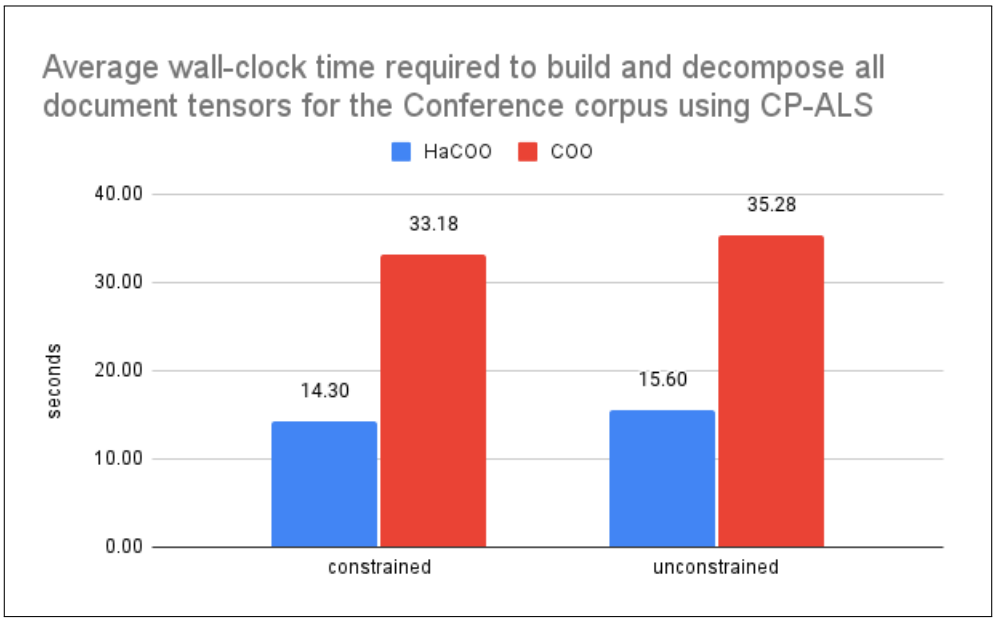

Figure 5.3: Average wall-clock time required to build and decompose all document tensors using CP-ALS for the Conference Corpus using COO format compared to HaCOO format.

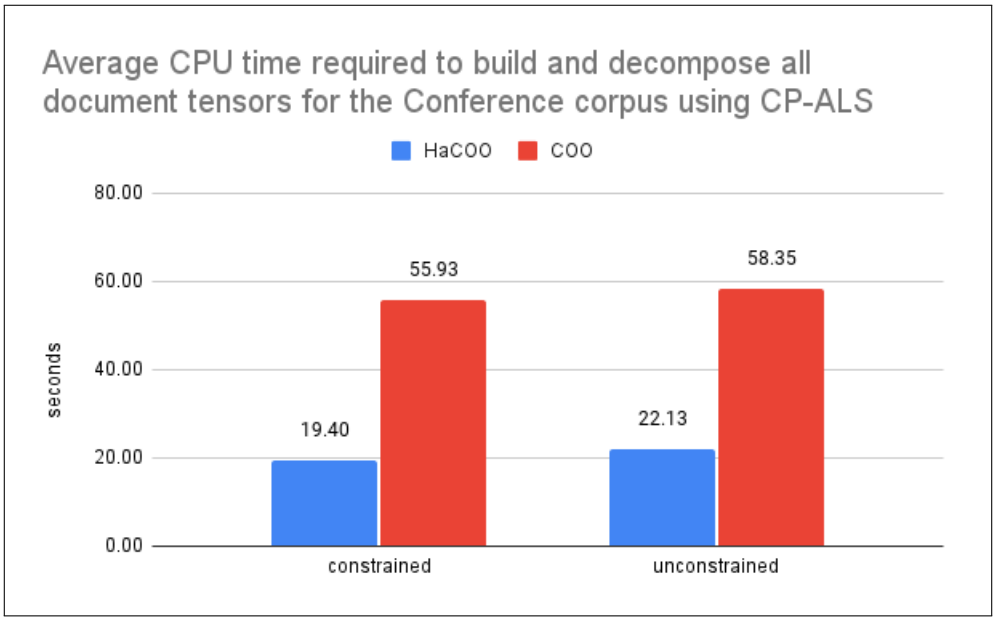

Figure 5.4: Average CPU time required to build and decompose all document tensors using CP-ALS for the Conference Corpus using COO format compared to HaCOO format.

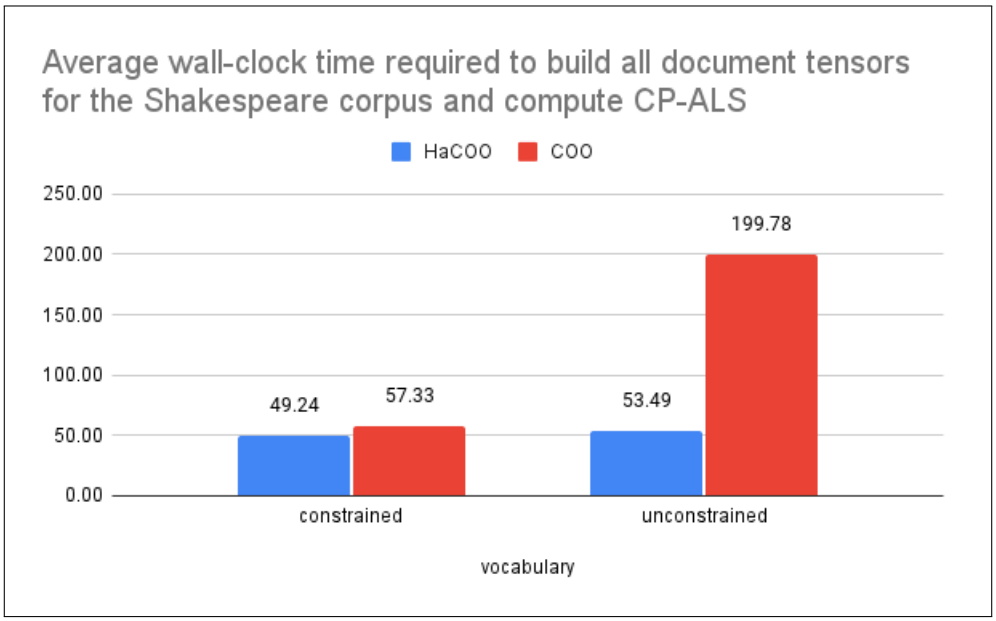

Figure 5.5: Average wall-clock time required to build and decompose all document tensors using CP-ALS for the Shakespeare Corpus using COO format compared to HaCOO format.

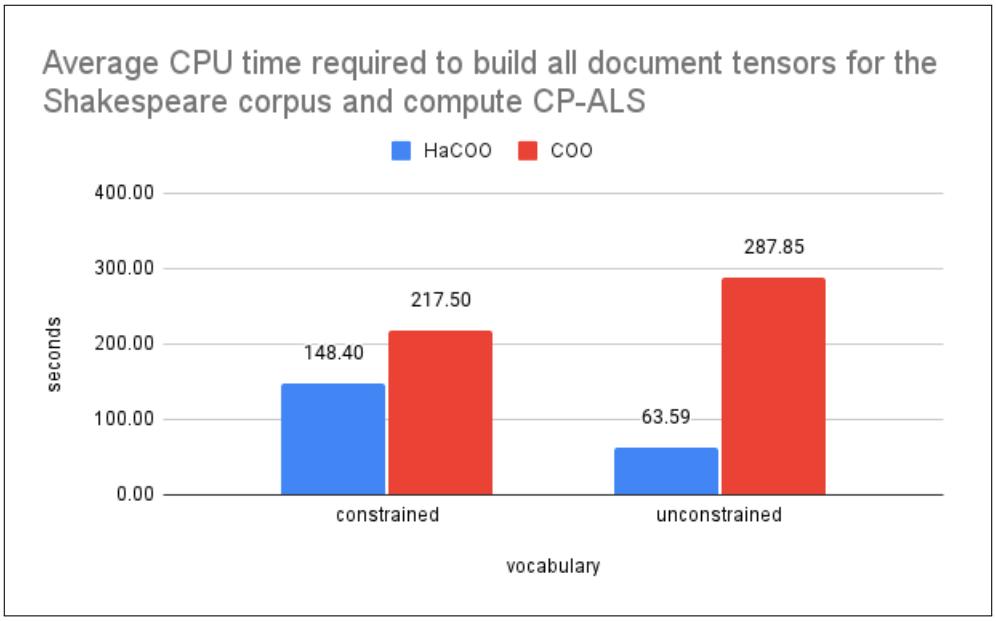

Figure 5.6: Average CPU time required to build and decompose all document tensors using CP-ALS for the Shakespeare Corpus using COO format compared HaCOO format.

Chapter 6

Conclusions

This work has presented evidence that supports HaCOO as a viable format for sparse tensor storage. HaCOO's support of constant time tensor updates and retrieval is a notable shift from previous formats. HaCOO format trades additional storage requirements for ensuring that time to search, insert, or retrieve tensor indexes does not grow due to larger tensor dimensions, which is a feature lacking from most common formats, such as standard COO, as well as the current state-of-the-art, ALTO. Additionally, HaCOO has the ability to modify an already existing sparse tensor data structure on the fly, in constant time.

Compared to Tensor Toolbox, HaCOO's MATLAB implementation outperformed standard COO format in terms of tensor updates once the number of elements reached a specific threshold. Performance with regard to tensor decomposition using the CP-ALS method was comparable, due to the MTTRKP operation incurring a small amount of overhead from extracting tensor elements from HaCOO's hash table. Furthermore, realizing HaCOO format in MATLAB provides easy accessibility to sparse tensor analysis without any additional requisite computing equipment. A modified HaCOO hash function was also proposed, consistently achieving lower collision rates as well as reducing the maximum chain length. The modified hash

function reduced the largest collision rate from 89.39% to 25.95% while maintaining an average probe depth of 1.

6.1 Future Research

Since HaCOO is still a fairly recent format, there are a number of tasks that can further expand the HaCOO MATLAB class and take advantage of HaCOO format's utility. The MATLAB HaCOO *htensor* class is missing some common tensor arithmetic functions such as tensor addition, subtraction, and so on. Additional code clean-up is necessary to make usage conventions consistent with Tensor Toolbox. It would also be bene cial to identify speci c properties of sparse tensors that are best suited for HaCOO format. Preliminary evaluations suggest that tensors with a higher number of dense bers increase the probability of hash collisions. An exact pro ling would be helpful in determining what sparse tensor properties result in increased or decreased collision rates. Additional modi cations to the hash algorithm can aid in further decreasing the rate of hash collisions.

Bibliography

[1] Macbook pro (retina, 13-inch, late 2013) - technical speci cations. https://support.apple.com/kb/sp691?locale=en_US, 2013. Accessed on July 20, 2023. 30

[2] Project gutenberg. https://www.gutenberg.org/, 2021. Accessed on July 20, 2023. 39

[3] Read the web research project at carnegie mellon university. http://rtw.ml.cmu.edu/rtw/overview, 2023. Accessed on July 20, 2023. 22

[4] Brett W Bader, Michael W Berry, and Murray Browne. Discussion tracking in enron email using parafac. In *Survey of Text Mining II*, pages 147{163. Springer, 2008. 1

[5] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 2.6. Available misc, February 2015. Accessed on July 20, 2023. 10, 27

[6] C.F. Beckmann and S.M. Smith. Tensorial extensions of independent component analysis for multisubject fmri analysis. *NeuroImage*, 25(1):294{311, Mar 2005. 1

[7] gymmyp1. Github - gymmyp1/hacoo-matlab. https://github.com/gymmyp1/hacoo-matlab, 2023. Accessed on July 20, 2023. 27

[8] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. Alto. *Proceedings of the ACM International Conference on Supercomputing*, Jun 2021. 15

[9] Rene Henrion. N-way principal component analysis theory, algorithms and applications. *Chemometrics and Intelligent Laboratory Systems*, 25(1):1{23, Sep 1994. 1

[10] Bob Jenkins. Hash functions. *Dr. Dobb's*, 1997. Accessed on July 20, 2023. 17

[11] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455{500, 2009. 3, 5, 6, 9, 10

[12] T.G. Kolda, B.W. Bader, and J.P. Kenny. Higher-order web link analysis using multilinear algebra. *Fifth IEEE International Conference on Data Mining (ICDM'05)*, 2023. 1

[13] L. Lathauwer and B Moor. From matrix to tensor : Multilinear algebra and signal processing, 2015. 1

[14] J. Li, J. Sun, and R. Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238{252, 2018. 11, 12, 14, 16

[15] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. A uni ed optimization approach for sparse tensor operations on gpus. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 47{57. IEEE, 2017. 12

[16] Robert Lowe, MeiLi Charles, and Amritpreet Singh. Hashed coordinate storage of sparse tensors. In *SC21 International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2021. 17, 20

[17] Robert E Lowe and Michael W Berry. Using non-negative tensor decomposition for unsupervised textual in uence modeling. In *Supervised and Unsupervised Learning for Data Science*, pages 59{82. Springer, 2020. 7, 35, 36, 37, 39

[18] MathWorks. Measure the performance of your code. https://www.mathworks.com/help/matlab/matlab_prog/measure-performance-of-your-program.html, 2022. Accessed on July 20, 2023. 29

[19] MathWorks. Strategies for e cient use of memory. https://www.mathworks.com/help/matlab/matlab_prog/strategies-for-efficient-use-of-memory.html, 2023. Accessed on July 20, 2023. 28

[20] Mathworks. Text analytics toolbox. https://www.mathworks.com/products/text-analytics.html, 2023. Accessed on July 20, 2023. 39

[21] Guy M Morton. A computer oriented geodetic data base and a new technique in le sequencing. 1966. 10

[22] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. *Introduction to Tensor Decompositions and their Applications in Machine Learning*. 1, 7

[23] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017. 10, 23, 29

[24] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1{7, 2015. 7, 11, 15

[25] TensorFlow. Working with sparse tensors, 2022. Accessed on July 20, 2023. 10

[26] Parker Allen Tew. An investigation of sparse tensor formats for tensor libraries. Master's thesis, Massachusetts Institute of Technology, 2016. 12, 15, 20

[27] Inc. The MathWorks. Map object that maps unique keys to values. https://www.mathworks.com/help/matlab/map-containers.html, 2022. Accessed on July 20, 2023. 39

[28] M. Alex O. Vasilescu and Demetri Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. *Computer Vision | ECCV 2002*, page 447{460, 2002. 1

Appendix

MATLAB HaCOO Class Declaration

```
% HACOO class for sparse tensor storage.

classdef htensor
    properties
        table   %hash table
        nbuckets  %number of slots in hash table
        modes   %modes list
        nmodes %number of modes
        bits
        sx
        sy
        sz
        mask
        max_chain_depth
        %number of elements in the hash table
        hash_curr_size
        %percent of the table that can be filled before
           rehashing
        load_factor
```

```
19      end
20      methods
21
22  %{
23  HACOO Create a sparse tensor using HaCOO storage.
24  Parameters:
25          1 argument construtors:
26              file - Load a .mat file with a HaCOO tensor
27              that has been created using write_htns()
                   function.
28              nbuckets - create a HaCOO tensor with a
                   specified number of buckets
29   OR
30          2 argument constructors:
31          subs - array of nonzero tensor subscripts
32          vals - array of nonzero values
33   OR
34          subs - array of nonzero tensor subscripts
35          vals - array of nonzero values
36          concatIdx - array of concatenated indexes
37  %}
38
39  function t = htensor(varargin) %<-- Class constructor
40
41      t.hash_curr_size = 0;
42      t.load_factor = 0.6;
43
44      switch nargin
```

```
45
46          case 1
47              % if number of buckets is specified
48              if isscalar(varargin{1})
49                  t.modes = [];
50                  t.nmodes = 0;
51                  t = hash_init(t,varargin{1});
52
53              % else load from .mat file
54              elseif isstring(varargin{1})
55                  loaded = matfile(varargin{1});
56                  t = loaded.t;
57              end
58          case 2 % subs and vals specified as arg1 and arg2.
59
60              idx = varargin{1};
61              vals = varargin{2};
62
63              % concatenate indexes
64              T = arrayfun(@string,idx);
65              X = strcat(T(:,1),'',T(:,2));
66
67              for i=3:size(T,2)
68                  X= strcat(X(:,:),'',T(:,i));
69              end
70
71              concatIdx = arrayfun(@str2double,X);
72
```

```
73              t.modes = max(idx);
74              t.nmodes = length(t.modes);
75
76              nnz = size(idx,1);
77              reqSize = power(2,ceil(log2(nnz/t.load_factor)
                    ));
78              NBUCKETS = max(reqSize,512);
79
80              % Initialize all hash table related things
81              t = hash_init(t,NBUCKETS);
82              t = t.init_table(idx,vals,concatIdx);
83
84          case 3
85              idx = varargin{1};
86              vals = varargin{2};
87              concatIdx = varargin{3};
88
89              t.modes = max(idx);
90              t.nmodes = length(t.modes);
91
92              nnz = size(idx,1);
93              reqSize = power(2,ceil(log2(nnz/t.load_factor)
                    ));
94              NBUCKETS = max(reqSize,512);
95
96              % Initialize all hash table related things
97              t = hash_init(t,NBUCKETS);
98              t = t.init_table(idx,vals,concatIdx);
```

```
99
100         otherwise
101             t.modes = [];   %<-- EMPTY class constructor
102             t.nmodes = 0;
103             NBUCKETS = 512;
104             t = hash_init(t,NBUCKETS);
105     end
106 end
```

Hash Function

```
1  %{
2  Hash the index and return the key.
3
4  Parameters:
5      t - The sparse tensor
6      m - Concatenated index
7  Returns:
8      k - hash key
9  %}
10 function k = hash(t, m)
11     hash = m;
12     %bit shift to the left
13     hash = hash + (bitshift(hash,t.sx));
14      %bit shift to the right
15     hash = bitxor(hash, bitshift(hash,-t.sy));
16      %bit shift to the left
17     hash = hash + (bitshift(hash,t.sz));
```

```
18      k = mod(hash,t.nbuckets);
19  end
```

Search Function

```
1   %{
2   Search for an index entry in hash table.
3   Parameters:
4       idx - The nonzero index to search for
5   Optional:
6       concatIdx - If you already have the concatenated
            version of the index, hash using that.
7   Returns:
8       If m is found, it returns the (k, i) tuple where k is
            the bucket and i is its location in the chain (the
            row it's located in)
9       If m is not found, return (k, -1).
10  %}
11  function [k,i] = search(t, idx,varargin)
12      % Set parameters from input or by using defaults
13      params = inputParser;
14      params.addParameter('concatIdx',-1,@isscalar);
15      params.parse(varargin{:});
16
17      % Copy from params object
18      concatIdx = params.Results.concatIdx;
19
```

```
20      % Check if idx is a concatenated index or inividual
           index components
21      if concatIdx ~= -1
22          % Cass the concatenated index to hash function
23          k = t.hash(concatIdx);
24      else
25          % Concatenate the index
26          s = num2str(idx);
27          s = strrep(s,' ','');
28          s = str2double(s);
29          k = t.hash(s);
30      end
31
32      %ensure there are no keys equal to 0
33      if k <= 0
34          k = 1;
35      end
36
37      % Check if there are no entries in that bucket
38      if isempty(t.table{k})
39          i = -1;
40          return
41      else
42          % Attempt to find item in that bubcket's chain
43          for i = 1:size(t.table{k},1)
44              if t.table{k}(i,1:end-1) == idx
45                  return
46              end
```

```
        end
    end
    i = -1;
end
```

Set Function

```
%{
Function to insert a nonzero entry in the hash table.
 Parameters:
       t - The hacoo sparse tensor
       idx - The nonzero index array
       v - The nonzero value
 Optionally -
       update - If index already exists, update its
          existing value by v
       concatIdx - If you have already concatenated the
          index, then you can pass it to save the time
          required to convert it.
 Returns:
       t - the updated tensor
%}
function t = set(t,idx,v,varargin)
    % Set parameters from input or by using defaults
    params = inputParser;
    params.addParameter('update',0,@isscalar);
    params.addParameter('concatIdx',-1,@isscalar);
    params.parse(varargin{:});
```

```
19
20      % Copy from params object
21      update = params.Results.update;
22      concatIdx = params.Results.concatIdx;
23
24      % Build the modes if we need to
25      if t.nmodes == 0
26          t.modes = zeros(length(idx));
27          t.nmodes = length(idx);
28      end
29
30      % Update any mode maxes as needed
31      for m = 1:t.nmodes
32          if t.modes(m) < idx(m)
33              t.modes(m) = idx(m);
34          end
35      end
36
37      if concatIdx ~= -1
38          % If a concatenated index got passed, search using
                that
39          [k, i] = t.search(idx,'concatIdx',concatIdx);
40      else
41          % try to find the index
42          [k, i] = t.search(idx);
43      end
44
45      % Insert accordingly
```

```
    if i == -1
        if v ~= 0
            if isempty(t.table{k})
                t.table{k} = [idx v];
            else
                % If not empty, append to the end of
                   existing entries
                t.table{k} = vertcat(t.table{k},[idx v]);
            end
            t.hash_curr_size = t.hash_curr_size + 1;
            depth = size(t.table{k},1);
            if depth > t.max_chain_depth
                t.max_chain_depth = depth;
            end
        end
    elseif update
        t.table{k}(i,end) = t.table{k}(i,end) + v;
    else
        fprintf("Cannot set entry.\n");
        return
    end

    % Check if we need to rehash
    if((t.hash_curr_size/t.nbuckets) > t.load_factor)
        t = t.rehash();
    end
end
```

Retrieve all tensor entries

```
%{
Retrieve all indexes and vals from a HaCOO sparse tensor.
Parameters:
    t - HaCOO htensor
Returns:
    subs - array of all indexes in HaCOO tensor t
    vals - array of all values in HaCOO tensor t
%}

function [subs,vals] = all_subsVals(t)
    nnz = t.table(nnzLoc(t));
    A = vertcat(nnz{1:end,:});
    subs = A(:,1:end-1);
    vals = A(:,end);
end
```

Textual Influence Modeling

Script to build vocabulary list

```
%{
 File: build_vocab.m
 Purpose: Create an indexed vocabulary to build document
    tensors. Requires MATLAB Text Analytics Toolbox.

Parameters:
    vocabFile - save vocabulary file
    freqFile - save frequency file
    constraint - limit vocabulary to the n most frequent
       words

Returns:
    N - number of documents
    words - array of unique words in the corpus
    wordToIndex - indexed vocabulary for the corpus
    newFileNames - file names for saving document tensors
       to files
%}

```

```
function [N,words,wordToIndex,newFileNames] = build_vocab(
   varargin)
%% Set up params
params = inputParser;
params.addParameter('vocabFile','vocabulary',@isstring);
params.addParameter('freqFile','@',@isstring);
params.addParameter('constraint',10e4,@isscalar);
params.parse(varargin{:});

%% Copy from params object
vocabFile = params.Results.vocabFile;
freqFile = params.Results.freqFile;
constraint = params.Results.constraint;
%%

files = dir('*.TXT');
N = numel(files);
newFileNames = cell(N,1);
words = cell(N, 1);

for i = 1:length(files)
    fid1 = files(i).name;
    newFileNames{i} = replace(fid1,'.txt','_coo.txt');
    fidI = fopen(files(i).name,'r');
    temp = textscan(fidI, '%s');
    %requires MATLAB Text Analytics Toolbox
    temp{1} = erasePunctuation(temp{1});
    docWords = {};
```

```
    for j=1:length(temp{1})
        lowerCase = lower(temp{1});
        if all(isstrprop(lowerCase{j},'alpha'))
            docWords{end+1} = lowerCase{j};
        end
    end
    words{i} = transpose(docWords);
end

% Build raw vocabulary dictionary
vocab = containers.Map;
for doc=1:N  %for every doc
    for i=1:length(words{doc})  %for every word in a doc
        word = words{doc}{i};
        if ~isKey(vocab,word) %if word is not in voacb,
           add it
            vocab(word) = 1;
        else
            vocab(word) = vocab(word) + 1;
        end
    end
end

%Sort by frequency in descending order
keys = vocab.keys;
mvals = cell2mat(vocab.values);
[vocabVals, sortIdx] = sort(mvals,'descend');
vocabKeys = keys(sortIdx);
```

```
71
72 if constraint > 0 && constraint < length(keys)
73     % count the other
74     other = 0;
75     for word=constraint+1:size(vocab,1)
76         other = other + vocabVals(word);
77     end
78
79     % trim the list
80     vocabKeys = vocabKeys(1:constraint);
81     vocabVals = vocabVals(1:constraint);
82     vocabKeys{end+1} = '<other>';
83     vocabVals(end+1) = other;
84 end
85
86 % index the vocabulary
87 vocabIndex = 1:length(vocabKeys);
88
89 %Save the vocabulary
90 fileID = fopen(vocabFile,'w');
91 for i=1:length(vocabKeys)
92     fprintf(fileID,'%s %d\n',vocabKeys{i},vocabIndex(i));
93 end
94 fclose(fileID);
95
96 % optionally save the frequency file
97 if freqFile ~= '@'
98     fileID = fopen(freqFile,'w');
```

```
    for i=1:length(vocabKeys)
        fprintf(fileID,'%s %d\n',vocabKeys{i},vocabVals(i)
           );
    end
    fclose(fileID);
end

% construct the word lookup
wordToIndex = containers.Map(vocabKeys,vocabIndex);
end
```

Build document tensors

```
%{
 File: doctns.m
 Purpose: Build list of document tensors.

Parameters:
    N - number of documents
    words - array of unique words in entire corpus
    wordToIndex - indexed vocabulary for entire corpus
    newFileNames - file names for saving document tensors
       as .mat files
    tns_format - which tensor format to use (sptensor or
       htensor)
    ngram - set the number of consecutive words when
       building tensor
    mat_save - write document tensors as HaCOO .mat files
```

```
13      coo_save - write document tensors as COO .txt files
14
15  Returns:
16      tnsList - cell array of built HaCOO or COO tensors for
            each document in current directory
17  %}
18
19  function tnsList = doctns(N,words,wordToIndex,newFileNames
        ,varargin)
20  params = inputParser;
21  params.addParameter('format','default',@isstring);
22  params.addParameter('ngram',3,@isscalar);
23  params.addParameter('hacoo_save',0,@isscalar);
24  params.addParameter('coo_save',0,@isscalar);
25  params.parse(varargin{:});
26
27  %% Copy from params object
28  format = params.Results.format;
29  ngram = params.Results.ngram;
30  hacoo_save = params.Results.hacoo_save;
31  coo_save = params.Results.coo_save;
32
33  %Check if tensor format is valid
34  if strcmp(format,"sptensor") || strcmp(format,"coo")
35      fmtNum = 1;
36  elseif strcmp(format,"htensor") || strcmp(format,"hacoo")
37      fmtNum = 2;
38  else
```

```
39      fprintf("Tensor format invalid.\n");
40      return
41  end
42
43  tnsList = cell(N,1); %blank cell array to store document
        tensors
44
45  % construct the document tensors
46  for doc=1:N %for every doc
47      %If using HaCOO format
48      if fmtNum == 2
49          tns = htensor();
50      elseif fmtNum == 1 %else use COO format
51          tns = sptensor(ones(1,ngram));
52      end
53
54      curr_doc = words{doc}; %word list for current doc
55      i = 1;
56      limit = length(curr_doc) - ngram;
57      idxList = zeros(limit,ngram);
58      % count the ngrams
59      while i < limit+2
60          gram = curr_doc(i:i+ngram-1);
61          idx = zeros(1,ngram);
62
63          % build the index
64          for w=1:length(gram)
65              word = gram{w};
```

```
            if ~isKey(wordToIndex,word)
                word = '<other>';
            end
            idx(w) = wordToIndex(word);
        end

        %store the index
        idxList(i,:) = idx;
        % next word
        i = i+1;
    end

    %concatenate the indexes for HaCOO
    T = arrayfun(@string,idxList);

    %apply to each row
    X = strcat(T(:,1),'',T(:,2)); %To start the new array

    for i=3:size(T,2)
        X= strcat(X(:,:),'',T(:,i));
    end

    concatIdx = arrayfun(@str2double,X);

    %insert elements
    for i=1:size(idxList,1)
        %If using HaCOO format
        if fmtNum == 2
```

```
94              tns = tns.set(idxList(i,:),1,'update',1,'
                    concatIdx',concatIdx(i));
95
96          %If using COO format
97          elseif fmtNum == 1
98              %Check if this index is larger than the
                    sptensor size
99              updateModes = idxList(i,:) > size(tns);
100
101             if any(updateModes) %if any index modes are
                    larger, just insert
102                 tns(idxList(i,:)) = 1;
103             else
104                 %update the entry's val
105                 tns(idxList(i,:)) = tns(idxList(i,:)) + 1;
106             end
107         end
108     end
109
110     if fmtNum ==2
111         %update the locations of occupied buckets
112         tns = tns.init_nnzLoc();
113     end
114     %store the tensor & advance to the next document
115     tnsList{doc} = tns;
116
117     %----------
118     if hacoo_save
```

```
119         % write the file
120         fileID = newFileNames{doc};
121         write_htns(tns,fileID,'-v7.3');
122     end
123 
124     if coo_save
125         fileID = newFileNames{doc};
126         write_coo(tns,fileID);
127     end
128     %--------
129 end
130 
131 
132 end %<-- end function
```

Vita

MeiLi Charles was born in Jiangxi province, China and adopted by an American family in Maryville, Tennessee. She was fascinated by technology and computers from a young age, resulting in completing a Bachelor's degree at Maryville College in Computer Science. During their undergraduate career, she completed internships at the Blount County Courthouse IT department and ORNL as part of the Nuclear Engineering Science Laboratory Synthesis (NESLS) program.

She continued to pursue a PhD in Computer Science at the University of Tennessee in 2019, acting as a graduate teaching assistant for Fall and Spring courses and the University of Tennessee's Governor's School, a summer program exposing high school students to introductory-level college STEM skills. At the encouraging prodding by a former instructor and research cooperative, she began working as an adjunct instructor at Pellissippi State Community College spring 2023 and will begin as full-time instructor this fall.