Ph.D. DISSERTATION

SPACE-EFFICIENT REPRESENTATION OF SEMI-STRUCTURED DOCUMENT FORMATS UTILIZING SUCCINCT DATA STRUCTURES

간결한 자료구조를 활용한 반구조화된 문서 형식들의 공간 효율적 표현법

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Junhee Lee

SPACE-EFFICIENT REPRESENTATION OF SEMI-STRUCTURED DOCUMENT FORMATS UTILIZING SUCCINCT DATA STRUCTURES

간결한 자료구조를 활용한 반구조화된 문서 형식들의 공간 효율적 표현법

지도교수 Srinivasa Rao Satti

이 논문을 공학박사학위논문으로 제출함

2020 년 10 월

서울대학교 대학원

전기 · 컴퓨터공학부

이 준 희

이준희의 박사학위논문을 인준함

2020 년 12 월

위 원 장	박 근 수	(인)
부위원장	Srinivasa Rao Satti	(인)
위 원	나 중 채	(인)
위 원	이 인 복	(인)
위 원	조 승 범	(인)

Abstract

Space-efficient Representation of Semi-structured Document Formats Utilizing Succinct Data Structures

Junhee Lee
Department of Electrical Engineering
and Computer Science
College of Engineering
The Graduate School
Seoul National University

Numerous big data are generated from a plethora of sources. Most of the data stored as files contain a non-fixed type of schema, so that the files are suitable to be maintained as semi-structured document formats. A number of those formats, such as XML (eXtensible Markup Language), JSON (JavaScript Object Notation), and YAML (YAML Ain't Markup Language) are suggested to sustain hierarchy in the original corpora of data. Several data models structuring the gathered data – including RDF (Resource Description Framework) – depend on the semi-structured document formats to be serialized and transferred for future processing.

Since the semi-structured document formats focus on readability and verbosity, redundant space is required to organize and maintain the document. Even though general-purpose compression schemes are widely used to compact the documents, applying those algorithms hinder future handling of the corpora, owing to loss of internal structures.

The area of succinct data structures is widely investigated and researched in theory, to provide answers to the queries while the encoded data occupy space close to the information-theoretic lower bound. Bit vectors and trees are the notable succinct data structures. Nevertheless, there were few attempts to apply the idea of succinct data structures to represent the semi-structured documents in space-efficient manner.

In this dissertation we propose a unified, space-efficient representation of various semi-structured document formats. The core functionality of this representation is its compactness and query-ability derived from enriched functions of succinct data structures. Incorporation of (a) bit indexed arrays, (b) succinct ordinal trees, and (c) compression techniques engineers the compact representation. We implement this representation in practice, and show by experiments that construction of this representation decreases the disk usage by up to 60% while occupying 90% less RAM. We also allow processing a document in partial manner, to allow processing of larger corpus of big data even in the constrained environment.

In parallel to establishing the aforementioned compact semi-structured document representation, we provide and reinforce some of the existing compression schemes in this dissertation. We first suggest an idea to encode an array of integers that is not necessarily sorted. This compaction scheme improves upon the existing universal code systems, by assistance of succinct bit vector structure. We show that our suggested algorithm reduces space usage by up to 44% while consuming 15% less time than the original code system, while the algorithm additionally supports random access of elements upon the encoded array.

We also reinforce the SBH bitmap index compression algorithm. The main strength of this scheme is the use of intermediate super-bucket during operations, giving better performance on querying through a combination of compressed bitmap indexes. Inspired from splits done during the intermediate process of the SBH algorithm, we give an improved compression mechanism sup-

porting parallelism that could be utilized in both CPUs and GPUs. We show by experiments that this CPU parallel processing optimization diminishes compression and decompression times by up to 38% in a 4-core machine without modifying the bitmap compressed form. For GPUs, the new algorithm gives 48% faster query processing time in the experiments, compared to the previous existing bitmap index compression schemes.

Keywords: Semi-structured Document Formats, Succinct Data Structures, Compression Algorithms, Space-efficient Algorithms, Integer Arrays, Bitmap Indexes, Big Data Processing.

Student Number: 2013-23134

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every minute, a plethora of instruments from a variety of areas gather and accumulate data. It is known that 2.5 quintillion bytes of the data are created every day [114]. Velocity of data accumulation exceeds that of storage increment. In other words, tremendous amount of information is produced on a daily basis, and the storage needed to save this is increasing significantly.

Given the limitations of the situation of available storage, it is becoming crucial for data to be compressed as much as possible. On the other hand, on the advent of *Big Data*, modern systems are still expected to execute operations and relevant analyses of data efficiently, even on large amounts of data. This requires functional compression approaches to be adopted, thereby allowing data to be compactly stored while permitting the efficient execution of operations.

In the real world, data are rarely a large sequence of random numbers, and real-world string data tend to exhibit foreseeable characteristics, structural regularities, and often subject to domain constraints. This predictability enables compression schemes to represent the same information in a space-efficient manner, while still permitting the original data to be retrieved in full. This property necessitates a new type of compression scheme that allows a set of queries be-

ing extracted from encoded data, which in turn differs from general-purpose compression schemes [117].

A common method for storage and exchange of the data in modern systems is through data exchange languages. These languages tend to be formats designed to describe data in ways that permit it to be read, parsed, and understood by the most diverse set of languages and platforms, decoupling data from particularities of its processing environment.

Over the course of the years, several data exchange formats were suggested, including XML (eXtensible Markup Language) [19], JSON (JavaScript Object Notation) [18], and YAML (YAML Ain't Markup Language) [14]. Many modern systems such as CouchDB [4] and MongoDB [32] have reported using JSON or BSON (Binary JSON) [1] as their native format for data storage and in-memory representation; while web service APIs – Twitter [104], Facebook, Google [57] – commonly adopt those formats as their data interchange language for transferring information between servers and clients.

Nevertheless, to the best of our knowledge, there are only few schemes in the literature specifically suggested for efficient compression or efficient in-memory representation of semi-structured documents. Some sets of libraries including JSONC [26], written in Javascript, focus on the compression of documents transferred between clients and web service APIs by employing traditional text compression methods. Those sets of encoding libraries do not integrate querying functionality, so that the entire document needs to be decompressed again for future analyses.

On the other hand, a separate research area known as succinct data structures has studied how the data could be compacted in terms of space, while supporting a set of operations. This research area is initiated by Jacobson [62], and as of current a plethora of data structures such as indexable dictionaries [93], trees [86, 15], permutations [85], and range minimum queries [51] are maintained using succinct data structures. We focus on this research area to

make semi-structured documents compact, while one can retrieve answers to the data analyses queries, agreeing upon an initial idea suggested by Rincy and Rajesh [96].

One notable work to mention is the one by Ottaviano and Grossi [92]: they proposed a scheme that supports random access to the JSON document stored on the disk, more efficiently, using a *semi-index*. The semi-index enables users to navigate the file by encoding the document structure succinctly. The semi-index is basically a bit vector in which bits are set for specific locations that separate the elements. This scheme is feasible since a different separator is employed for each possible type. At the cost of a space overhead for storing a succinct representation of the document tree structure, their semi-index allows for random access of specific values without having to load the JSON file entirely into the main memory. Furthermore, the semi-index includes pointers that indicate the position of the corresponding element in the JSON file stored on disk.

Unfortunately, their representation neither actually compresses the document, nor strives to represent the JSON content succinctly in memory, but rather offers a layer of indirection for accessing the underlying stored data. In this respect, the total amount of disk space required by their approach is strictly higher, though not by much, than the original document, as it additionally requires the storage of the semi-index.

These intuitions have led to the research aim of space-efficient representation of semi-structured document formats utilizing succinct data structures, which this dissertation.

1.1 Contribution

In this dissertation, we suggest a compact, memory-efficient representation of semi-structured document formats, and its practical implementation engineered by leveraging ideas of diverse succinct data structures. Our scheme saves RAM and disk usage in three aspects of those document formats. First, we model the

document structure as an ordinal tree, and encode it through succinct ordinal tree representations [62, 86, 15]. Second, redundancies in attributes are removed and the remaining unique strings are stored in a simple contiguous array, which can be compressed using space-efficient data structures for strings, including compressed suffix arrays [81, 109, 80]. Lastly, values of the documents are encoded compactly and stored in a heterogeneous structure named as *bit string indexed array*. Users can store this set of representations, either on RAM or on the disk. For the RAM and disk representations we allow users to query general information of a semi-structured document, without retaining the original document.

Along with this main contribution, we provide subsidiary implementations of space-efficient data structures and algorithms whose idea is incorporated onto the main representation.

- We propose an idea to compress an integer array which needs not be sorted [71]. This algorithm optimizes Elias Gamma coding system [45], a type of universal code system, by using a compact bit vector structure [90]. Experiments show that the proposed scheme reduces space usage by up to 44% while consuming 15% less time than the original code system, while this scheme additionally supports accessing an arbitrary element in the compressed array without whole decompression.

- We improve SBH, one of the byte-based bitmap index compression algorithms [70]. The core contribution of this algorithm is the existence of intermediate super-bucket during compression and decompression. This data structure assists querying through compressed bitmap indexes. Inspired from splits done during the intermediate process of the SBH algorithm, we give an improved compression mechanism supporting parallelism that could be utilized in both CPUs and GPUs. We show by experiments that this CPU parallel processing optimization diminishes

compression and decompression times by up to 38% in a 4-core machine without modifying the bitmap compressed form. For GPUs, the new algorithm gives 48% faster query processing time in the experiments, compared to the previous existing bitmap index compression schemes.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces basic concepts used throughout the dissertation. The following three chapters discuss intermediate space-efficient data structures and algorithms whose idea is utilized in the main representation. Chapter 3 goes over an approach to represent an array of increasing integers, in compact manner. In Chapter 4 we suggest and reinforce the bitmap index compression scheme. The overall contribution of this dissertation – space-efficient representation for semi-structured documents – is elaborated in Chapter 5. Finally, we re-summarize our contribution in Chapter 6.

Chapter 2

Background

In this chapter we introduce common concepts dealt in the following chapters.

2.1 Model of Computation

Throughout the dissertation we consider the standard **word RAM (Random Access Memory) model** [83] as the model of computation. In this model, a memory cell stores a word of size w. The algorithm can read and write any arbitrary cell in the memory, as well as perform arithmetic and boolean operations in a cell, in $O(1)$ time.

In theoretic point of view, the word size w is set to $\Theta(\log n)$ bits, where we use $\log n$ to denote $\log_2 n$ in the entire dissertation. While we focus on practical performance of the algorithms, we set w to be 64 bits, following the tendency of modern machines.

Other than the word RAM model, cell probe model [83] and external memory model [108] are also often considered in literature. Additionally, there exists more constrained models, such as ROM model and in-place model [27, 66].

2.2 Succinct Data Structures

In succinct data structures [63], we strive to solve algorithmic problems by designing data structures that use amounts of space close to the information-theoretic lower bound, while still supporting the operations efficiently. One can think of succinct data structures as an extension of data compression, in which space is close to the information-theoretic lower bound and queries are efficient. Succinct data structures for a wide range of fundamental problems have been designed in the past couple of decades.

One of the fundamental structures employed when devising a new succinct data structure solution is the *bit string*. A bit string is a string over the alphabet $\Sigma = \{0, 1\}$. A bit string by itself has limited use. Although its compactness provides us a framework upon which information can be concisely represented, few useful operations can be efficiently performed on raw bit strings. To enhance its usability, bit strings can be extended in terms of functionality with auxiliary structures for `rank` and `select` operations.

Given a string S of length n over the alphabet Σ, the `rank` and `select` operations are defined as follows:

- $rank_\alpha(S, i)$: the number of occurrences of α in the first i positions of S, for any $\alpha \in \Sigma$.
- $select_\alpha(S, i)$: the position of the i-th α in S, for any $\alpha \in \Sigma$.

In the case that S is a bit string and, thus, $\Sigma = \{0, 1\}$, we have the operations $rank_0$, $rank_1$, $select_0$ and $select_1$. For instance, if $S = 110101$, then $rank_0(S, 4) = 2$ and $select_1(S, 2) = 1$. Extensive research has been conducted on succinct implementations of rank and select structures over bit strings [33, 84, 62, 87]. One can support both operations in $O(1)$ time while using $o(n)$ additional bits of space.

On the same vein, a second auxiliary structure built around bit strings is

the *balanced parentheses*. This data structure conceptually interprets set bits (i.e., 1s) and unset bits (i.e., 0s) of a bit string as open and close parentheses, respectively. When this resulting sequence of opening and closing parentheses is balanced, it is considered as a balanced parentheses structure.

Just as it was the case with `rank` and `select`, the core operations in balanced parentheses over an n length bit string can be performed in constant time with additional $o(n)$ bits [87]. For convenience we can define `rank` and `select` operations over balanced parentheses bit strings as $rank_{open}(S, i) \equiv rank_1(S, i)$, $rank_{close}(S, i) \equiv rank_0(S, i)$, $select_{open}(S, i) \equiv select_1(S, i)$, and $select_{close}(S, i) \equiv select_0(S, i)$.

Chapter 3

Space-efficient Representation of Integer Arrays

In this chapter we propose a space-efficient representation of integer arrays, regardless of the sorted-ness of arrays. This work is based on a combined work of Lee and Satti [71].

3.1 Introduction

Tremendous amounts of numerical data are generated on a daily basis, from numerous sources. The era of big data has emerged, numerous data are created every day. Increment of those data is more rapid than that of the storage. In most of the programming languages, a numeric variable occupies fixed amount of memory space. For instance, in the C++ programming language, an integer variable defined as `long long` type uses 8 bytes of space, regardless of its digits. To alleviate this circumstance, a plethora of compression algorithms are being studied.

Intensive amount of studies are done to support fast access to specific elements in the compressed integer array, not requiring decompression but still

occupying lesser space than maintaining the ordinary array. It is shown that integer arrays can be compacted using universal lossless compression schemes that most text compression schemes rely on. Nevertheless, applying those algorithms eradicates inherent properties of integers and necessitates whole decompression of the arrays to process even an single element. Therefore, this mechanism is not appropriate for processing queries operated by accessing an arbitrary location, extracting the element and performing computations needed.

In this chapter we suggest an alternate, space-efficient representation of Elias Gamma coding [45] using succinct data structures [90]. We suggest an improved code system to represent an integer array by utilizing concepts of succinct data structures. This system is based on a scheme of succinct bit vector that allows compression of delimiter bit array while supporting access queries in constant time. Experimental results show that the encoded array in fact uses lower space, while not sacrificing time efficiency.

3.2 Preliminaries

In this section we briefly describe various universal code systems that the new scheme is based on, and a succinct data structure supporting operations needed for the new representation.

3.2.1 Universal Code System

Prefix code is a code system satisfying prefix property. Prefix property indicates that a code in a code system is not able to be a prefix of any other codes in the system. By using prefix codes, one can decode any continuous messages encoded in the code system without existence of the delimiter, which marks either beginning or ending location of any individual messages.

An universal code is a binary representation of prefix codes storing positive integers, considering probabilistic distribution. When a ratio between the actual code length and the ideal code length predicted by the distribution is bounded

by the entropy function, universal codes get the maximum performance. Well-known universal codes include Elias Gamma coding, Elias Delta coding [45] and Fibonacci coding [53].

Elias Gamma Code

Elias Gamma code for a positive integer x is computed using the following procedures:

---

1. Compute $N = \lfloor \log_2 x \rfloor$.
2. Attach N 0s to the code.
3. Attach x in binary form to the code.

By the above operations, it is clear that $2N + 1 = 2 \lfloor \log_2 x \rfloor + 1$ bits are needed to represent x.

Elias Delta Code

Elias Delta code for a positive integer x is computed using the following procedures:

---

1. Compute $N = \lfloor \log_2 x \rfloor$.
2. Encode $N + 1$ using Elias Gamma code.
3. Attach x in binary form, but without the MSB, to the code.

To represent x in Elias Delta code, $N + 2 \lfloor \log_2(N + 1) \rfloor + 1 = \lfloor \log_2 x \rfloor + 2 \lfloor \log_2(\log_2 x + 1) \rfloor + 1$ bits are needed.

Fibonacci Code

Fibonacci code for a positive integer x is computed using the following procedures:

---

1. Find the largest Fibonacci number $F(i)$ that is not larger than x. Note that $F(0) = 1$, $F(1) = 2$, $F(n) = F(n-1) + F(n-2)$ $(n \geq 2)$. Let remainder be $x - F(i)$.
2. Change the $i-2$th leftmost bit in the code to 1.
3. Repeat the former two steps, setting the remainder to x until remainder becomes 0.
4. Attach 1 to the rightmost location in the code.

Since no continuous 1s can exist in all codes in the aforementioned coding systems, we can distinguish each code in the array.

Other code systems used to compress integer arrays, although not universal, include Golomb code [56] and Rice code [95]. These two systems divide each integer into two parts using parameters, and consider those parts using different types of codes.

Moreover, there are works to store integer arrays compactly by maintaining difference between two incident elements, utilizing relationship between them [107, 98]. However, these works are only able to handle monotonically increasing arrays, extra sorting needs to be applied for those algorithms to work. This requires additional $O(n \log_2 n)$ time.

Lastly, DAC supports random access by considering only digits of an integer with values and storing those hierarchically [22].

3.2.2 Bit Vector

We have defined a *bit vector* as a set of bits (either 0 or 1) in Chapter 2. Given a bit vector S with n elements, the following operations are defined [35]:

- `access`(S, i): return the i-th element in S. $(0 \leq i \leq n-1)$
- `rank`(S, a, i): return number of a (either 0 or 1)s until the i-th location in S. $(0 \leq i \leq n-1)$
- `select`(S, a, p): return the location of pth a (either 0 or 1) in S.

A well-known research area using bit vectors is bitmap indexing which will be discussed in Chapter 4, where database is indexed using bit vectors. By using this indexing scheme, one can process `OR` or `AND` queries in fast speed. Nevertheless, extra space needed to maintain those indices is proportional to the multiplication of cardinality and number of elements, which is a significant overhead.

There exists approaches to process the aforementioned operations using lower time, while occupying lesser space. If the raw bit vectors are uncompressed, `rank` and `select` operations are done in $O(1)$ time [62]. Several succinct data structures are suggested to compress the bit vectors while supporting the operations in constant time. Notably, works by Raman et al. [93] and Okanohara and Sadakane [90] do not alter the time complexity while significantly compressing the bit vectors if they contain many more 0s than 1s.

3.3 Algorithm Description

We illustrate our new way to represent integer arrays in this section. First, we introduce the proposed code system. Following the introduction, we design an optimization on the system.

3.3.1 Main Principle

The code system we suggest is derived from the phenomenon that N bits of 0s are unneeded while producing Elias Gamma code. When an integer x is represented using $N+1$ bits, its MSB (Most Significant Bit) is always 1. Therefore, this code system considers code c of an integer x without its MSB, being N necessary to maintain c. To also represent 0 and 1, this code system encodes $x+2$.

Unfortunately, this breaks the prefix property mentioned in the previous section, so an additional maintenance of delimiter is needed. A delimiter d of an integer x needs N bits of space. Every bits of d is initialized to 0, except the $N-1$th bit which is set to 1. When a sequence of delimiters is constructed, by calling `select` operation, one can access location of any arbitrary integers in the integer array.

This may seem as a contradiction to the discussion, neglecting necessity of delimiters being a core strength of most code systems. Fortunately, existence of succinct bit vector representation proposed by Okanohara and Sadakane [90] enables compaction of a vector of delimiters. This is mainly because we assume delimiters contain much less 1s than 0s, since at most 62 0s can be matched to a single 1.

The representation in [90] uses $m\lceil\log_2(n/m)\rceil+2m$ bits to store a bit vector of length n with m set bits. Utilization of this representation onto the delimiter allows an integer array of n elements to use $T+n\log_2\log_2\hat{x_i}+2n$ bits in average, where $T=\sum_{i=1}^{n}\lfloor\log_2 x_i\rfloor$ and $\hat{x_i}$ is average value of integers inside the array.

Proof. Each entry in the delimiter vector contains $\lfloor\log_2 x\rfloor$ bits, with a single set bit. When the integer array contains n elements with an average value of element being $\hat{x_i}$, by [90] the succinct delimiter representation occupies $n\lceil\log_2(T/m)\rceil+2n+o(n)\approx 2n+n\log_2\log_2\hat{x_i}$ bits. Thus, the overall representation uses $T+2n+n\log_2\log_2\hat{x_i}$ bits. □

This is less than the space occupied by ordinary Elias Delta code which is $T + 2n \log_2 \log_2 \hat{x_i} + n$ bits, even considering both the code and the delimiter vector, since, $2 \log_2 \log_2 x + 1 - (\log_2 x \log_2 x + 2) = \log_2 \log_2 x - 1 \geq 0$ for $x \geq 1$.

Since the succinct representation of delimiter vector supports $\texttt{select}(x, 1, p)$ operations in theoretically constant time, random access to the space-efficient integer arrays is also supported in constant time. This is not possible in the ordinary Elias Gamma code and other coding systems.

For instance, an encoding of the array $\{20, 16, 21, 19\}$ using Elias Gamma code occupies 36 bits as follows:

0000 10110 0000 10010 0000 10111 0000 10011

The proposed code system encodes the array with two different structures as follows:

- C: 0110 0010 0111 0011
- D: 0001 0001 0001 0001 (shown in uncompressed form)

Since most elements in D are 0s, this uses lesser space than the ordinary 16 bits if compressed using the succinct representation. To access the 3rd element 21 in the encoded array, we need to perform two `select` operations. In detail, by running $\texttt{select}(D, 1, 2) = 8$ and $\texttt{select}(D, 1, 3) = 12$, we retrieve the encoded form $C[8, 11] = 0111$. To summarize, the proposed code system supports the following two operations:

1. $(C, D) = \texttt{encode}(A)$: encode an integer array A with n elements. Encoded result B contains a code vector C and a delimiter vector D.
2. $x = access(B, l)$: retrieve the $l (1 \leq l \leq n)$th element x in B.

3.3.2 Optimization in the Implementation

Additionally, if the number of elements in array is known a-priori, we are able to partition D or combination of C and D in P chunks. Although additional $128P$ bits are needed to mark the first location each chunk maintains, this is negligible compared to the length of the entire code or delimiter. If two partitions have equal digits, then the relevant Ds are identical. Then, we do not need to maintain both the Ds altogether, but we may store a single D (let it be D_1) and let another D point to D_1. This optimization saves extra space. Note that this does not alter C, so result of `access` is not affected.

For example, when $P = 2$, encoding of the array $\{20, 16, 21, 19\}$ is as follows:

- C: 0110 0010 0111 0011
- D_1: 0001 0001
- D_2: 0001 0001

Since D_1 and D_2 are identical, we only maintain D_1 and point D_2 to D_1.

3.4 Experimental Results

The suggested code system is implemented in the C++ programming language. We use SDSL (Succinct Data Structure Library) [55] that implements succinct data structures from various literature. For representing a code vector C, `bit_vector` in the SDSL library is used. Delimiter vector D is implemented using `sd_vector` following the idea of [90].

Experimental environment for testing performance is equipped with Intel i7-6700 CPU, 16GB DDR4 RAM, 256GB SSD, and Linux kernel 4.9 is operated. The entire program code is compiled using g++ version 6.3.0 with O3 optimization. We compared the performance of the proposed code system with Elias Gamma, Elias Delta and DAC algorithms dealt in Section 3.2.1. These are pre-implemented in the SDSL as `vlc_vector<elias_gamma>`,

`vlc_vector<elias_delta>`, and `dac_vector`, respectively.

We performed experiments based on two types of data: synthetic and real. Synthetic data are generated using `uniform_int_distribution` and `default_random_engine` in C++, consisting of 30,000,000 random unsigned integers ($0 \leq x \leq 2^{63} - 1$). Real data are extracted from the trade volume of S&P 500 index ($47 \leq x \leq 21,474,836$), with $n = 122,574$. Arrays generated from both synthetic and real data are not sorted in any specific order.

Code System	Code size (KB)	Encoding time (ms)	Decoding time (ms)
Elias Gamma	441	3.3	-
Elias Delta	258	1.9	-
DAC	284	16	8.6
Simple	249	1.6	7.9

Table 3.1 Application of code system in the synthetic data.

Code System	Code size (KB)	Encoding time (ms)	Decoding time (ms)
Elias Gamma	454	7	-
Elias Delta	341	6	-
DAC	322	16	26
Simple	325	6	19

Table 3.2 Application of code system in the real data.

Tables 3.1 and 3.2 indicate experimental results of encoding and decoding synthetic and real data, respectively. Our representation is marked as *simple* in the two tables. Code size denotes the encoded data stored in disk, and includes the delimiter for our scheme. Decoding time records extracting i-th arbitrary element in the entire encoded sequence. Since Elias Gamma coding and Elias Delta coding do not support this functionality, the relevant values are not included in the tables. Optimization described in Section 3.3.2 is not included in *simple*.

# Elem / Part	Code size (KB)	Encoding time (ms)	Decoding time (ms)
2	244	14	15
3	294	14	17
6	778	22	22
93	642	20	22
186	491	15	21
659	379	9	20
3,954	326	6	18

Table 3.3 Application of code system with different number of partitions.

We can see that the enhanced scheme occupies lesser space than the original Elias-based code systems. Furthermore, the scheme takes similar or lesser encoding time than the two algorithms. When comparing the two code systems supporting random access in the encoded form, the *simple* scheme uses about 1% more space than DAC, but takes about 27% lesser decoding time in the real data. Our algorithm gives better performance by all means in the synthetic data.

Applying Optimization

Utilizing optimization discussed in Section 3.3.2, we tested with different number of partitions while encoding and decoding the real data.

Table 3.3 represents the experimental result. If the number of elements per partition gets lower, then we can see that the code size decreases. This is because possibility of the identical delimiter character increases. In contrast, when the number of elements per partition becomes higher, efficient compression of the delimiter is achieved, so the code size also becomes lower. For the intermediate cases, overhead of maintaining the partition information occurs, so efficiency diminishes.

Chapter 4

Space-efficient Parallel Compressed Bitmap Index Processing

In this chapter we illustrate processing compression and decompression of the bitmap indexes in parallel, which could be utilized to process bit vectors efficiently in the querying phase of the proposed semi-structured document representation. The compression scheme is based on the SBH compression scheme [70], which is a byte-based compression algorithm.

This chapter is based on, and an extended work of Kim et al. [70].

4.1 Introduction

Many enterprise applications generate a massive volume of data through logging transactions or collecting sensed measurements, which brings up the need for effective indexing methods for efficient storage and retrieval of data. To support some of the retrieval queries efficiently, database management systems make use of indexes such as B-trees and bitmap indexes [72, 99]. The index schemes based on B-tree are efficient for a wide variety of data, since they support both

searches and updates with nearly the same complexity. However, for most data warehousing applications, search operations are more frequent than updates. For such applications, bitmap indexes may improve the overall performance. Besides, when the number of unique values of an attribute is small (e.g., gender), one can achieve much better performance than B-trees by using bitmap indexes.

A bitmap index is a collection of bit vectors created for each distinct value in the indexed column. The number of bit vectors in a bitmap index for a given indexed column, also referred to as *cardinality*, is equal to the number of distinct values that appear in the indexed column. For a column with cardinality m, the i-th bit vector in the designated bitmap index corresponds to the i-th distinct value (in any order) and the j-th bit in the i-th bit vector is set to 1 if and only if the value of the j-th element in the column is equal to i. Thus, if an indexed column of n elements has cardinality m, then its bitmap index consists of m bit vectors with length n, occupying mn bits in total.

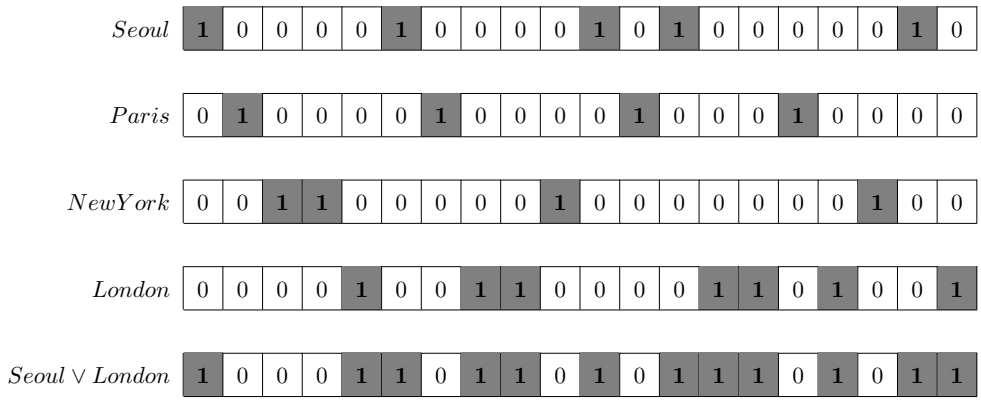


Figure 4.1 A set of bitmaps in a column `City`.

Figure 4.1 shows a set of bitmaps for an attribute `City`, along with a bit vector generated by an `OR` query. Cardinality of this attribute is four, since it contains four distinct values. Each bit vector represents whether the value of `City` is one of the four available values.

```
SELECT *
```

```
FROM T
WHERE City=Seoul OR City=London
```

The main operation performed in the bitmaps is a bitwise operation $B_{Seoul} \vee B_{London}$, where B_c is a bitmap corresponding to a city c. Standard SQL selection queries on bitmap indexes can be emulated by performing bitwise operations on the bit vectors. Since modern computers are optimized to perform bitwise operations, dealing with bitmap indexes is easy and efficient [28, 29, 91].

One of the main drawbacks of bitmap indexes is that their space usage is high when compared to the raw data or a B-tree index, especially when their cardinality is high. To overcome this issue, several bitmap index compression algorithms have been proposed. The simplest way to compress a bitmap index is to compress each individual bit vector using a standard text compression algorithm, such as LZ77 [117]. This drastically improves the space usage. Nevertheless, general text compression algorithms do not consider properties of bitmap indexes, so the whole decompression of the compressed data is required before queries are done. Therefore, performing logical operations on the compressed bit strings is usually much slower than uncompressed bit strings.

As a result, compression schemes that apply some form of *Run-Length Encoding* (RLE) are suggested to be used for compressing bitmaps. RLE is a lossless compression method that encodes a string by splitting it into sequences of *run*s, and then encoding the runs efficiently. Since a typical bitmap index used in database applications is assumed to have sparsely-distributed set bits (i.e., most bits are set to zero) forming long sequences of zeros, RLE-based schemes are expected to achieve decent compression ratio and so are commonly used to compress bitmaps.

Two of the most popular compression schemes, based on the run-length encoding method, are *Byte-aligned Bitmap Code* (BBC) [8, 9] and *Word-Aligned Hybrid* (WAH) bitmap compression [112]. Both the schemes divide the original

bit vectors to be compressed into blocks of a specific unit size. The main difference between these two schemes is that BBC uses a unit size of 8 bits, while WAH uses either 32 or 64 bits as unit size. In terms of performance, BBC typically consumes less space, while WAH supports faster query processing. Several derivatives are also suggested in literature, which we introduce in the following section.

Kim et al. [70] proposed an improved version of bitmap index compression scheme, mainly following BBC, called *Super Byte-aligned Hybrid* (SBH) bitmap compression. It uses a new feature that enhances time performance in processing logical operations so that its byte-aligned encoding can lead to better compressibility in comparison with a word-aligned scheme. Specifically, the superiority of SBH in compressibility becomes more pronounced when the cardinality of an indexed column grows larger than 50. The query processing time of SBH is about five times faster than that of WAH, while the size of compressed bitmap indexes was retained nearly close to that of BBC.

Since bitmap indexes are incorporated into a plethora of database management systems such as PostgreSQL and MonetDB, needs to process bitmap indexes in parallel to boost query performance are also more in demand. The main merit of controlling bitmap indexes in parallel is that the result of queries performed in a partial range of the bitmap is not dependent on the remaining parts while querying the relevant bitmaps, so the processing time is inversely proportional to the number of processes, in theory. This property affects both compression and querying (decompression) phases. While this observation is prevalent, little consideration is given for compressing bitmap indexes in parallel using CPU.

The use of Graphics Processing Unit (GPU) is well established to handle multiple calculations efficiently, thus a wide range of industries such as big data processing and machine learning utilize GPUs. In addition, some NoSQL database management systems such as OmniSci and SQream actively use GPUs

for data management. Recently, several works [89, 103] discuss GPU utilization over query processing in bitmap indexes. A well-known computing platform to employ GPUs is *Compute Unified Device Architecture* (CUDA) proposed by NVIDIA. This architecture enables software engineers to directly access GPU instructions, and provides a wide set of bitwise operations as well as shared memory. Two notable bitmap index compression algorithms – GPU-WAH [5] and GPU-PLWAH [6] – are proposed for GPUs. Both schemes run on the CUDA platform.

In this chapter, we suggest a parallelized version of SBH that works in both CPU and GPU, which significantly enhances the time performance compared to the plain version. We show that the processing time is almost proportional to the number of CPU cores the algorithms use. For the GPUs, we show that the optimized algorithm gives better performance in the query processing time. Both the algorithms produce the identical compressed form as well as the equivalent query result to the plain version.

The rest of this chapter is organized as follows. We first describe other bitmap compression schemes that were proposed in literature, including SBH, and how they were implemented in Section 4.2. In Section 4.3, we describe our parallelized optimization details of SBH. Section 4.4 discusses experimental results and comparison with the existing schemes.

4.2 Related Work

Several compression schemes based on run-length encoding have been proposed in the literature. The main merit of these schemes is that logical operations could be done without decompressing the whole bitmaps. One of the earliest such schemes that have been successful is the Byte-aligned Bitmap Code [9], or BBC for short. BBC is effective in compressing bit sequences, and in particular for representing bitmap indexes.

To improve runtime performance of BBC, a word-based bitmap compres-

sion algorithm, called the Word-Aligned Hybrid (WAH) scheme has been introduced [112], which takes advantage of the word-level bitwise operations. Subsequently, many other compressed bitmap indexing schemes have been proposed [25, 30, 40, 54, 61, 74, 101, 105, 111]. Some of the compression schemes are surveyed by Chen et al. [31] and mathematically analyzed by Guzun and Canahuate [60] and Wu et al. [115].

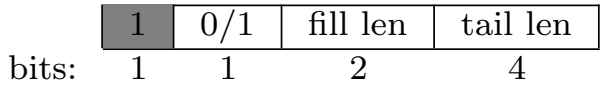
In fact, most of these schemes are variants of WAH, and perform better than WAH in terms of either time or space or both. They also often achieve better time performance than BBC, but not in space. As mentioned in the later experiments, SBH performs better than both BBC and WAH, in terms of both time and space. For this reason, we compare plain version of SBH only with these two schemes.

4.2.1 Byte-aligned Bitmap Code (BBC)

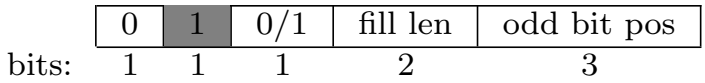
Byte-aligned Bitmap Code (BBC) [9] is a byte-based bitmap compression scheme that encodes an uncompressed sequence of bits by a sequence of bytes. More specifically, a *run* of bits (i.e., sequence of bits having the same value) followed by a small number of bytes are compressed into a *header byte*, optionally followed by one or more *counter bytes* followed by zero or more *literal bytes*, as explained below.

Two variants of BBC scheme have been proposed: a one-sided version that only compresses runs of zeros, and a two-sided version that compresses runs of both zeros and ones [113]. In this dissertation, we only describe the two-sided version since it can compress almost as well as the one-sided version [65]. In the two-sided version of the BBC scheme, there are four types of distinguishable runs. These four types are distinguishable by their header bytes, which are shown in Figure 4.2.

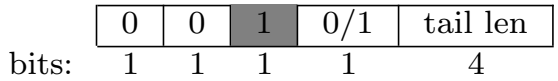
- Case 1: a run of length at most 3 bytes followed by a tail of length up to 15 bytes is encoded by a *header byte* followed by a sequence of *tail bytes*.



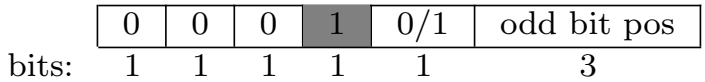
(a) Case 1. Fill<4 bytes, Tail<16 bytes.



(b) Case 2. Fill<4 bytes, odd bit position.



(c) Case 3. Fill≥4 bytes, Tail<16 bytes.



(d) Case 4. Fill≥4 bytes, odd bit position.

(e) Counters used for Case 3 and Case 4

Figure 4.2 Four cases of BBC header.

The *Most Significant Bit* (MSB) of the header byte is set to 1 to indicate this case. The bit following the MSB stores a *fill-bit*, which is either 0 or 1 depending on whether the run it encodes is a sequence of zeros or ones. This is followed by a two-bit representation storing the length of the run (in bytes), which in turn is followed by a four-bit representation indicating the length of the tail (again, in bytes). If k is a value stored in the last four bits of this header byte (where $0 \leq k \leq 15$), the k bytes following the header byte store the next k bytes following the run that is encoded by the header byte.

- Case 2: a run of at most 3 bytes, followed by a single *odd byte* is encoded by a single header byte. The odd byte has a property that all bits, except one, in the byte are set to the same bit as the run preceding it. The first two most-significant bits in the header byte are set to 01 to indicate this case. The next bit stores the fill bit, and the two bits following it store the length of fill. The remaining three bits store the *odd position* (i.e., position that has a different bit in the odd byte).

- Case 3: this case is similar to case 1, except that the length of fill is at least 4. Instead of putting the fill length in the header byte, it uses one or more *counter bytes* to encode the fill length. The MSB of a counter byte is set to 1 if there are more counter bytes, and is set to 0 otherwise. The remaining seven bits in the counter byte store the bit sequence corresponding to the fill length (spread over all the counter bytes).

- Case 4: this case is similar to Case 2, except that the length of the fill is at least 4. Again, counter bytes are used to store the fill length (as in Case 3).

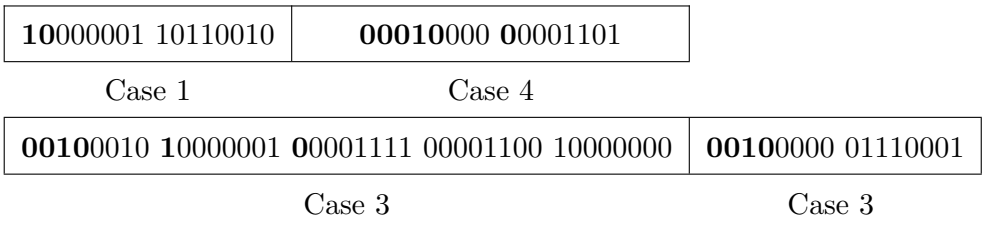
An example bit sequence and its result of compression is shown in Figure 4.3. The BBC scheme compresses nearly as well as *gzip* [65]. However, in the

10110010000...00000000001000...00000001100100000000000...000

(a) Uncompressed bit sequence of 1736 bits.

$$\underbrace{10110010000...}_{8}\underbrace{00000000001000...}_{14\times8}\underbrace{0000000110010000000000...}_{145\times8}\underbrace{000}_{57\times8}$$

(b) Consideration of literals and runs.



(c) BBC encoding.

Figure 4.3 A bit vector with BBC compression process.

worst case, the total time required to perform a logical operation on two BBC compressed bitmaps can still be longer than that on two uncompressed bitmaps.

4.2.2 Word-Aligned Hybrid (WAH)

Another compression scheme based on run-length encoding is *Word-Aligned Hybrid* (WAH) scheme [112]. This scheme is similar to BBC, except that unit of compression is a *word* (of length either 32 or 64 bits) instead of a byte. Throughout this chapter we consider size of a word to be 32 bits.

The scheme is much simpler compared to the two-sided BBC described in the earlier section. The input bitmap is divided into blocks of length 31 bits each. Each word in the compressed bitmap encodes one or more of these blocks. If a block contains both zeros and ones, then it is encoded as a *literal block* by setting the MSB to 0, and writing the 31 bits of the block as remaining bits in the word. For $k \leq 2^{30} - 1$, if there is a run of k blocks of zeros (or ones), then this run is encoded by setting the MSB to 1, the bit following the MSB to 0 (or 1), and the remaining bits to the value of k in binary. Runs that are at least

2^{30} blocks are encoded by splitting them into shorter runs that can be encoded in one word in a greedy manner.

An example bit sequence and its result of compression is shown in Figure 4.4.

101100100...0000...000000...00100..0000...000000...00001100010...0000...0000

(a) Uncompressed bit sequence of 1736 bits.

$$\underbrace{101100100...}_{31}\underbrace{0000...}_{2\times31}\underbrace{000000...}_{31}\underbrace{00100..}_{36\times31}\underbrace{0000...}_{31}\underbrace{000000...}_{31}\underbrace{00001100010...}_{14\times31}\underbrace{0000...0000}$$

(b) Grouping literals and runs.

0101100100...00		**10**0...00010	**0**0...010000
100...100100	**0**0...00110	**0**010...000	**10**0...1110

(c) WAH encoding.

Figure 4.4 A bit vector with WAH compression process.

4.2.3 WAH-derived Algorithms

We give a brief overview of the schemes whose idea is from the WAH compression scheme.

Enhanced Word-Aligned Hybrid (EWAH)

Enhanced Word-Aligned Hybrid (EWAH) [74] is a variant of WAH. This algorithm improves the WAH scheme but the most important difference from WAH is that EWAH never generates a compressed bitmap larger than (within 0.1% of) the uncompressed bitmap.

The EWAH compression scheme works as follows. The given bitmap is divided into 32-bit groups. These groups are classified as two types: *clean* word and *dirty* word. A clean word contains all equal bits, either all zeros or all ones. On the other hand, a dirty word cannot be compressed since it consists of heterogeneous bits (0 and 1). A marker word is a kind of header indicating

a type of next words. The marker word consists of three parts. The first part (one bit) is type of clean words (zero or one), half of a word (16 bits) store the number of clean words and the rest of bits (15 bits) store the number of dirty words. Thus, the maximum number of clean words and dirty words that can be compressed using a single marker word are $2^{16} - 1$ and $2^{15} - 1$, respectively.

Position List WAH (PLWAH)

Prior WAH-based schemes consider a word with only a single odd bit as a literal, which makes the word incompressible. *Position List WAH (PLWAH)* [40] tries to alleviate this, by storing location of an odd bit explicitly in a fill word using $s \log_2 w$ bits, where w is the size of a single word and s is the number of odd bits.

PLWAH compresses a bitmap by four steps: (i) divide the original bitmap into $w - 1$ bit groups; (ii) identify groups that are either 0-fills or 1-fills which can be merged; (iii) for non-mergeable groups, append a 0 to MSB to indicate that they are literals. Otherwise, put the length of a merge to $w - 2 - s \log_2 w$ *Least Significant Bit (LSB)*s, put 1 to MSB and put either 0 (for 0-fill) or 1 (for 1-fill) to the bit next to the MSB to represent the type of a merge; (iv) identify a nearly-identical literal following the merge, calculate the location of the odd bit and put it in a fill word.

Partitioned WAH (PWAH)

Partitioned WAH (PWAH) [105] divides a single word into P partitions, while a header of P bits exists to specify the type of a certain partition, whether it is a fill or a literal. Compared to WAH and EWAH, PWAH has the efficiency of storing bitmaps that have a sequence of literal and 0-fill. Depending on the word size, PWAH has fixed value of P. In case of $w = 32$, there are two different values of P: 2, 4, whose partition occupies $\frac{32-P}{P}$ bits.

PWAH uses extended fill to save the space needed to represent contiguous

zeros that exceeds the maximum value of a run to indicate. Rather than adding up the number of fill length consecutively, extended fills concatenate numbers in a bitwise manner.

COMPressed Adaptive indeX (COMPAX)

Different from three WAH-based compression schemes, *COMPressed Adaptive indeX (COMPAX)* [54] only considers 0-fills compressible. Nevertheless, COMPAX merges a sequential combination of 0-fills and literals to a single chunk (run), which is known as piggybacking approach. This applies when only a single byte in a word is not a 0-fill.

There are two more types of a single chunk other than a literal and a 0-fill. One is LFL, which stands for Literal-Fill-Literal, and another is FLF representing Fill-Literal-Fill. For each literal inside this merged form, only a single byte in a single word differs from 0-fills. Compression of bitmap is done on-the-fly with a look-back of at most two words for merging components of FLF and LFL.

Variable-Aligned Length WAH (VAL-WAH)

The *Variable-Aligned Length WAH (VAL-WAH)* [61] scheme proposes a framework that optimizes both compression and query time by allowing bitmaps to be compressed using variable-length encoding while maintaining alignment. The size of a word w is a power of 2 and it is fixed by a certain machine-dependent factor. However, setting an unit of compression different from $w-1$ is possible. Based on this, dividing a word with segments and putting compressed results there is suggested.

Assuming a word with size w bits, this scheme stores compressed results in blocks of size s, which is known as the segment factor. When $s = w - 1$, this algorithm can be considered same as WAH. Otherwise, more than two blocks appear in a single word. Given w and an alignment factor b, possible values of

s are determined by:

$$s = 2^i \times (b - 1)$$

$$\text{where } 0 \leq i \leq \log_2 w - \log_2 b$$

An input bitmap is first inspected by segment length selector. Upon the possible values of s and another user-tuning factor λ, the selector calculates the compressed size using a specific value of s and returns the one with the lowest probable size. After this process is finished, actual bitmap compression is done. A header is generated for each word, with the w/b most significant bits indicating the content of each segment.

Run-Length Huffman (RLH)

Run-Length Huffman (RLH) [101] is a combination of run-length encoding and Huffman encoding. Firstly, this scheme calculates distances between 1s in the bitmap and gets frequency of the distance. For instance, a bit sequence 0000001 can be represented as 6 and the frequency of 6 is one. After this process finishes, the result is compressed by Huffman encoding.

In the first step, distances between successive 1s are calculated. Frequencies of the gathered values are computed and are used to build the Huffman tree. The codes generated from this Huffman tree are used to compute the encoding.

4.2.4 GPU-based WAH Algorithms

We now describe two of the WAH-derivated bitmap index compression algorithms that focus on GPUs.

GPU Version of WAH (GPU-WAH)

GPU-WAH [5] is a modified version of the WAH scheme mentioned in the earlier section. The entire algorithm consists of three procedures – extension, compression, and decompression. First of all, for efficient computation, the input

bitmap is extended to a multiple of 32 bits, while considering each unit as a 31-bit word. This could be thought as a sequence of 32-bit words, with the first 31-bits being the original content and a zero afterwards.

Let n be the number of units in the input bitmap. The compression procedure first allocates an array F of size n in GPU. If the i-th unit of the extended bitmap is a literal, or type of the fill differs from the $i+1$-th unit, then the i-th element of F is set to 1, or 0 otherwise. The last element of F is fixed to 1. Exclusive scan on F generates another array SF. Next, by using the value $m = F[n-1] + SF[n-1]$, an array T with size of the (a-priori) output bitmap is allocated. T maintains the number of words up to the designated location $SF[i]$. Finally, the output compressed bitmap C is generated in parallel, by referring to T and the original bitmap.

Decompression procedure is done in reverse manner. The compressed bitmap is also considered as a sequence of 32-bit words. An array S of size being the number of units (m) is allocated. i-th element of S records either 1 (literal) or the number of fills. Once S is calculated, exclusive scan is done to compute another array SS. Number of words in the original bitmap n is then fixed to $SS[m-1] + S[m-1]$. Utilizing this information, F and SF are recovered, and then the original bitmap is allocated and decompressed from the two arrays.

GPU-WAH mainly assumes that all of the relevant queries are done in GPU, to prevent overhead of transfers between host (RAM) and device (GPU).

GPU Version of PLWAH (GPU-PLWAH)

GPU-PLWAH [6] is a parallel implementation of the PLWAH scheme described earlier. This algorithm performs all of the three procedures – compression, decompression, and querying – in GPU.

Extension of the input bitmap is done identical to that of the GPU-WAH algorithm. When the number of subbitmaps n is calculated, dedicated GPU memory is allocated. When traversing the bitmap, the algorithm marks whether

a word is a fill using the remaining bit. Once this procedure is completed, the algorithm determines whether the unit is identical to the upcoming unit and stores relevant information in an array F, so that they are able to be compressed. By these information, size of the compressed bitmap index m is known a-priori, by $F[m-1]+SF[m-1]$ where SF is the result of exclusive scan. Two subsidiary arrays $T1$, $T2$ with size m is allocated. $SF[i]$-th element of $T1$ stores $i+1$, while i-th element of $T2$ is set to $\lceil (T1[i]-T1[i-1])/(2^{25}-1) \rceil$. After all the computations are done, the algorithm allocates and generates the compressed bitmap using values of $T1$, $T2$, and the original bitmap.

Decompression algorithm does the reverse process. The overall procedure is identical to that of GPU-WAH, though an odd bit is directly negated by the stored position in the compressed bitmap.

This scheme also allows parallel bitwise operations between two PLWAH-encoded bitmaps. Nevertheless, to perform such operations, this algorithm needs to convert the bitmaps into WAH-encoded bitmaps. The result of the bitwise operations is encoded again in PLWAH compression.

4.2.5 Super Byte-aligned Hybrid (SBH)

Super Byte-aligned Hybrid (SBH) [70] is a byte-based compression scheme similar to BBC. It uses a concept denoted as *super-bucket* to speed up logical operations, at the expense of a slight increase in the space when compared with BBC.

Unlike BBC and WAH, SBH first divides the bit sequence into blocks, called super-buckets, of length l_b, for some integer parameter l_b. It then applies a byte-based compression scheme to retrieve the compressed bitmap. In the following, we describe compression and decompression algorithms for the SBH scheme in more detail.

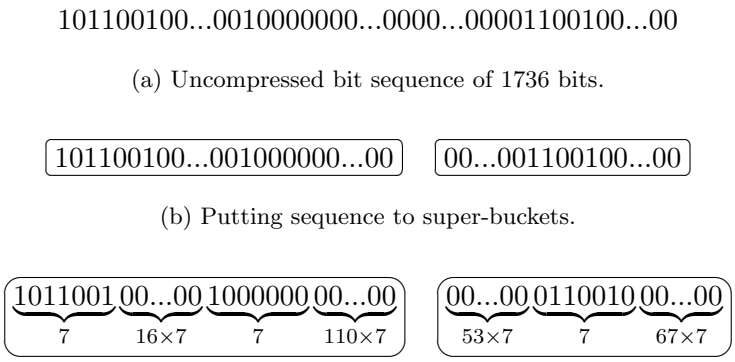


Figure 4.5 A bit vector with SBH compression process.

Compression Process

SBH first divides the input bitmap into super-buckets of length l_b, and further divides each super-bucket into buckets of length 7 bits each. When all the seven bits in a bucket are zeros (or ones), we call that a *0-fill* (or *1-fill*) bucket. Otherwise, it is considered as a *literal* bucket. The algorithm chooses l_b to be a multiple of 7, so that each super-bucket holds an integer number of buckets.

A literal bucket is stored as-is in a byte, with an added 0 in the MSB position. Any other bytes in the compressed bitmap that do not encode a literal bucket have their MSBs set to 1 to distinguish them from literal buckets. In particular, if there exists a run of zeros (or ones) that spans k buckets in the uncompressed bitmap, for $1 \leq k \leq 2^{12} - 1$, it is encoded by writing the value of k in binary using either one or two bytes (as explained later).

If the value k is at most 63 ($= 2^6 - 1$), it is simply wrote down as binary representation of k using six bits, and either **10** for 0-fill or **11** for 1-fill in the most significant positions are added. On the other hand, if k is larger than 63,

then binary representation of k is written using 12 bits. The procedure is: (i) split k into two equal halves; (ii) add **10** (or **11**) at the beginning of both halves; (iii) write the resulting bytes in reverse order (i.e., second byte followed by the first byte).

For instance, suppose there is a run of zeros that spans 91 buckets (i.e., 91 $\times$ 7 consecutive 0s). The algorithm encodes this run by writing the two bytes **10**011011 and **10**000001, which is obtained by writing the binary representation of 91 with leading zeros to make its length equal to 12 (namely, 000001 011011), splitting it into two equal halves, adding **10** to both (to indicate that they represent a run of zeros), and finally, reversing the two resulting bytes. The reversal is done so that we can decode the compressed sequence without any look-ahead. In the example, the first byte is decoded as a run of 27$\times$7 zeros, and the next byte (because it follows a byte with the same header part) is decoded as a run of 64$\times$7 zeros, which are then added to decode a run of 91$\times$7 zeros.

A detailed example of a bit sequence and its corresponding SBH-compressed bitmap are shown in Figure 4.5, and the algorithm describing the compression for SBH is described in Algorithm 1. Algorithms dealing with 0-fills and 1-fills are explained in Algorithms 2 and 3, respectively. For Figure 4.5, the length of super-bucket is arbitrarily set to $7 \times (2^7 - 1)$.

Decompression Process

The basic flow of decompression is the opposite of the compression process.The algorithm first reads bytes from a compressed bitmap (from left to right) and decodes them one by one. In addition, this algorithm uses super-bucket that reduces the query processing time. Initially, bits inside super-bucket are set to all zeros.

The MSB of each byte in the bitmap tells us whether it encodes a literal or a fill. If the MSB is zero, the scheme performs an OR operation between the byte and the current position inside super-bucket. On the other hand, if the MSB

Algorithm 1: Compression algorithm of SBH.

Input: Uncompressed bitmaps, size of super-bucket (l_b)

Output: Compressed bitmaps.

```
for x = 1 to cardinality do
    for y = 1 to length of bitmaps/l_b do
        for z = 1 to number of groups within super-bucket do
            // consider a group is whether a literal or a fill (zero or
             one);
            if group = zero-fill then
                MakeZeroFill (the number of zero-fill);
            else if group = one-fill then
                MakeOneFill (the number of one-fill);
            else if group = literal then
                append one bit = 0 and literal to compressed bitmaps;
    Process the next bitmap.
```

Algorithm 2: MakeZeroFill.

Input: The number of zero-fills ($count_0$).

Output: The compressed zero-fills format.

```
while count_0 > 0 do
    append two bits 10 and first 6 bits of count_0 to compressed bitmaps;
    count_0 = count_0 ≫ 6;
count_0 = 0;
```

is one, then that byte stores a fill. The bit next to the MSB represents the fill bit, either 0 or 1. The last six bits represent the counter value from which the scheme gets the length of the run. When decoding a fill, the scheme remembers the first two bits of that and check whether they match the first two bits of the following byte. If they match, then the scheme multiplies the value in second byte by 64 (2^6) and adds it to the counter represented by the previous byte.

Algorithm 3: MakeOneFill.

Input: The number of one-fills ($count_1$).

Output: The compressed one-fills format

while $count_1 > 0$ **do**

append two bits 11 and first 6 bits of $count_1$ to compressed bitmaps;

$count_1 = count_1 \gg 6$;

$count_1 = 0$;

01011001	**10**010000	**0**1000000	**10**101110	**10**000001	**10**110101	**0**0110010	**10**000010	**10**000001

(a) Compressed form.

Super-bucket (896 bits)				Super-bucket (840 bits)		
Literal	16 × 7 0s	Literal	110 × 7 0s	53 × 7 0s	Literal	66 × 7 0s
1011001	0000000...0000000	1000000	0000000...0000000	0000000...0000000	0110010	0000000...0000000

(b) Expansion till the size of super-bucket.

Super-bucket (896 bits)	Super-bucket (840 bits)
1011001000...0001000...000	000...00011001000...000

(c) Merged form, result of decompression.

Figure 4.6 A bit vector with SBH decompression process.

If the byte represents a 0-fill, the corresponding byte is not needed to be decompressed. That is, because bits inside this byte are all zeros, the original bitmap should only contain zeros. Thus, the next x number of bits can be skipped, where x is the value of counter. For example, assume the decompression position in the bucket is 8 and the number of the counter on the 0-fill is 6. The decompression position on the bucket is set to $8 + 42$ (6×7). Between the bit position 9 and 49, the algorithm does not perform any operations because bits inside the relevant position in super-bucket are not affected by the result of decompression.

Decompressing process of the result sequence in Figure 4.5 is shown in Figure 4.6.

Algorithm 4: Compression algorithm of SBH, with CPU parallelism.

Input: Uncompressed bitmaps (size of super-bucket (l_b)), number of processes (n_c).

Output: Compressed bitmaps.

```
for t = 1 to n_c in parallel do
    for x = 1 to ⌈cardinality/n_c⌉ do
        for y = 1 to length of bitmaps/l_b do
            for z = 1 to number of groups within super-bucket do
                if group = zero-fill then
                    MakeZeroFill (the number of zero-fill);
                else if group = one-fill then
                    MakeOneFill (the number of one-fill);
                else if group = literal then
                    append one bit = 0 and literal to compressed
                     bitmaps;
        Process the n_c-th next bitmap.
```

4.3 Parallelizing SBH

We consider the parallelized version of the aforementioned compression and decompression algorithms. SBH scheme supports both CPU and GPU for parallel processing.

4.3.1 CPU Parallelism

The overall compression algorithm, described in Algorithm 4, is identical to that of the plain version. Instead, not only processing a single bitmap per iteration, the process handles n_c number of bitmaps, where n_c is the number of cores in the CPU. Each compression is handled independent of the other $n_c - 1$ concurrent executions.

The super-bucket in the decompression phase is maintained in the shared

memory region, so that the concurrent transactions may perform queries in parallel without maintaining individual super-buckets. We atomically modify the super-bucket. Skips in the concurrent phase do not affect the content of the shared super-bucket, but the range of a skip may exceed the length of the super-bucket. We maintain a flag per process to indicate whether the process completed decompressing the bitmap index until the length of the super-bucket. Once all the processes set their flags, then the relevant row IDs are extracted as the query result.

4.3.2 GPU Parallelism

For GPUs, the compression algorithm is modified to process a single bitmap in parallel manner, instead of compressing a set of bitmap indexes. This is mainly because GPU has limited amount of memory to store all the bitmap indexes that could be processed in parallel. Although the statement is valid for processing large scale of indexed data, to support compressing multiple short bitmap indexes concurrently we allocate a bitmap index per single CUDA block.

In order to get concurrent compression work in a single bitmap, we logically divide a bitmap index into length of a super-bucket while compressing it. Each thread in a block handles the divided chunk, thus all the threads independently process compression. This way is identical to the compression algorithm described in Figure 4.5, so the parallelized algorithm guarantees that the division does not alter the compression result.

Algorithm 5 explains the modified compression algorithm. Two-level parallelism following the CUDA structure is applied in the algorithm. We assume in the algorithm that the relevant bitmap indexes need to be uploaded to GPU memory, and the result of compression is downloaded to RAM after encoding of a single bitmap index completes.

Since we have controlled compressed form of bitmap indexes to be fit into a single super-bucket, it is easier for to handle decompression even in GPU.

Algorithm 5: Compression algorithm of SBH, with GPU parallelism.

Input: Uncompressed bitmaps (size of super-bucket (l_b)), number of blocks (n_b), number of threads per block (n_t).

Output: Compressed bitmaps.

```
for b = 1 to n_b in parallel do
    for x = 1 to ⌈cardinality/n_b⌉ do
        for y = 1 to n_t in parallel do
            for s = 1 to ⌈number of groups /n_t⌉ do
                for z = 1 to number of groups within super-bucket do
                    if group = zero-fill then
                        MakeZeroFill (the number of zero-fill);
                    else if group = one-fill then
                        MakeOneFill (the number of one-fill);
                    else if group = literal then
                        append one bit = 0 and literal to compressed
                        bitmaps;
                Process the n_t-th next group.
            Process the n_b-th next bitmap.
```

Nevertheless, since the SBH-compressed form occupies smaller size than the ordinary bitmap, the decompression phase processes one bitmap index per thread. This assumption allows the super-bucket residing in a shared memory which is shared among threads in a block.

4.4 Experimental Results

In this section, we compare practical performance of our scheme with the other bitmap compression schemes. As mentioned earlier, we use two of the well-known bitmap compression schemes, for CPU and GPU, to compare with SBH. In addition, we also show how parallelization of the algorithm affects the overall performance.

Experiments were conducted on a PC with Intel i7-6700K 4.2GHz, 64GB RAM and 4TB hard disk. The machine ran Linux kernel 4.4 64-bit. For the GPU version experiments, a single NVIDIA Titan Xp GPU is attached to the machine. All tested compression techniques are implemented in C++. Two main factors we want to improve are the size of the compressed bitmap index and the time required to perform the boolean operations. In addition, we also consider optimization of compression time in terms of CPU parallelization.

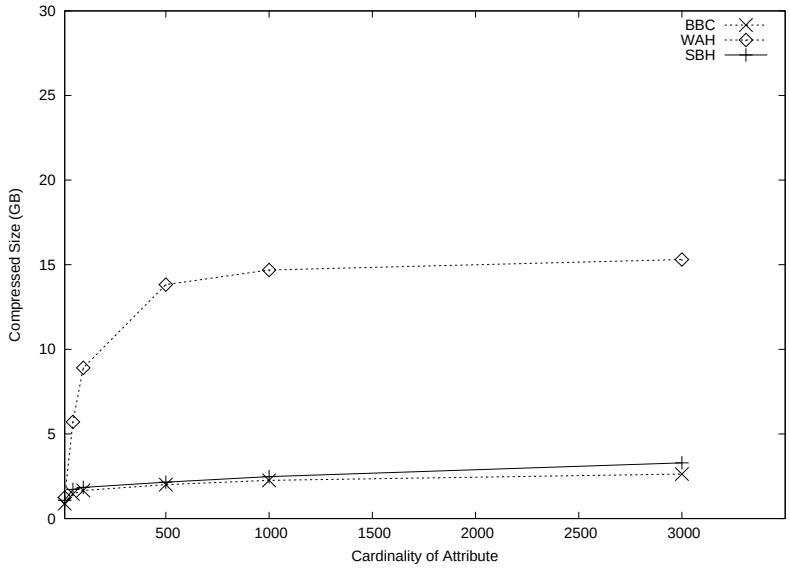
The size of the compressed bitmap indexes is measured using the Linux command `wc` to compute the overall size of bitmaps. Query time is measured for range queries which execute bitwise-OR operations. Even though other bitwise operations are also commonly used, we assume that time for these other Boolean operations is similar to that of OR operation. `std::chrono` in C++ for CPU and `cudaEventElapsedTime` in CUDA for GPU are used to measure the running time.

In our experiments, an OR operation can be described as a sequence of bitwise OR operations in the relevant bitmap indexes.

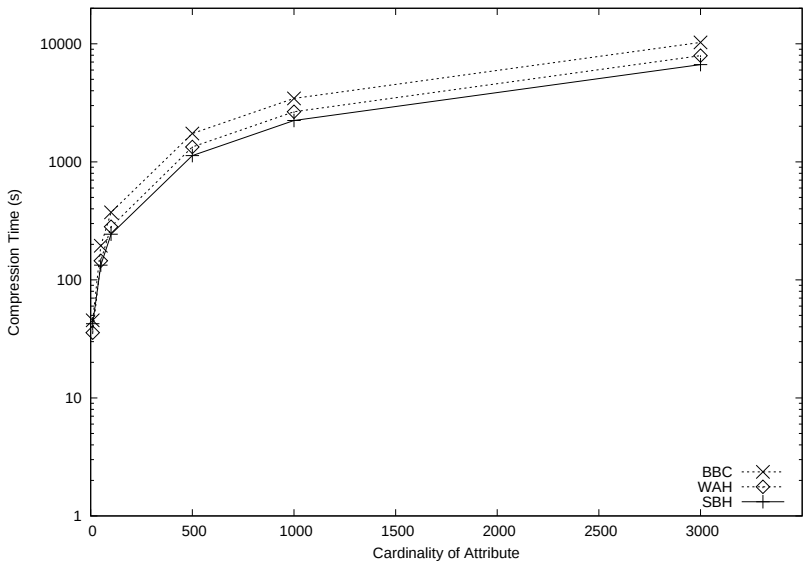
We use synthetic data for the experiments. The synthetic data are generated using `srand` and `rand` functions which are in the `stdlib.h` header provided by the GNU C Library.

4.4.1 Plain Version

We first evaluate the performance of SBH scheme in several ways by comparing it with BBC and WAH, as in [70]. First, we compare the schemes in terms of compression time by varying the cardinality of the attribute. he performance of the schemes is analyzed with respect to the cardinality and the number of rows. Finally, we check the relationship between data size and query processing time, where a query consists of bitwise-OR operations.

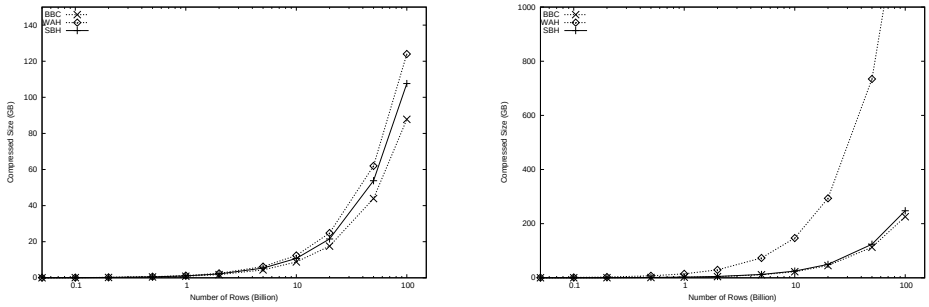


(a) Compression size.



(b) Compression time.

Figure 4.7 Tendency of compression with cardinality on synthetic data (one billion rows).



(a) 10 cardinality.

(b) 1,000 cardinality.

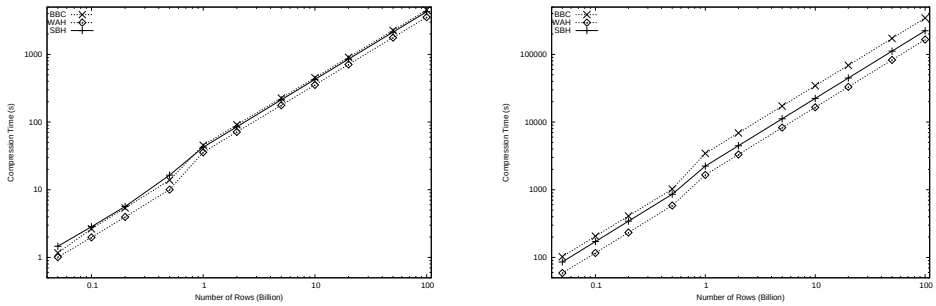
Figure 4.8 Tendency of compression size on synthetic data, along with the number of rows.

Compression Size and Time

Figure 4.7 shows the size of bitmap indexes and compression time in relation to the cardinality of attribute, for one billion rows. Note that except WAH, the compression ratio of all techniques gets better while having similar tendency. In Figure 4.7a, we notice that the index size of WAH increases rapidly until around 100 cardinality, but the graph grows smoothly after that. In terms of compression time shown in Figure 4.7b, SBH is the fastest.

Figure 4.8 shows the tendency of compressed index size on randomly-generated synthetic data. We measured the compression time with only these two cardinalities, since bitmap indexes are hardly compressed for small (10) cardinality, while for large (1,000) cardinality, bitmap indexes are compressed well. In both cases, the graph is linearly increased with respect to the number of rows. As we mentioned before, in 10 cardinality, compressing bitmap indexes is poorly done.

Figures 4.9 shows the tendency of compression time for cardinalities 10 and 1,000. It shows that the compression time of all techniques increases linearly. Intuitively, it is clear that the larger the number of rows gets, the longer time it takes to compress the bitmap indexes.



(a) 10 cardinality.

(b) 1,000 cardinality.

Figure 4.9 Tendency of compression time on synthetic data, along with the number of rows.

In terms of size, SBH and BBC schemes take significantly less space than WAH, since byte-based schemes have better compressibility than word-based schemes, in general. The relative performance does not change even though the size of data increases. To summarize, the compression time is proportional to the number of rows, while the size of compressed bitmap indexes using SBH is relatively small.

Query Processing Time

Figure 4.10 shows the query processing time with cardinality of attribute at one billion rows. After 50 cardinality, the query processing time of SBH is the fastest, regardless of cardinality and number of bitwise operations. The query processing time of SBH does not lag behind.

Figure 4.11 shows the experimental results measuring query processing time executed in 10 and 1,000 cardinality, where a query consists of eight `OR` operations. In high cardinality range denoted in Figure 4.11b, SBH is 50 times faster than BBC, and 5 times faster than WAH. Nevertheless, in low cardinality range shown in Figure 4.11a, query processing time of WAH is faster than that of SBH. This is because SBH is seldom compressible when cardinality is low, and WAH

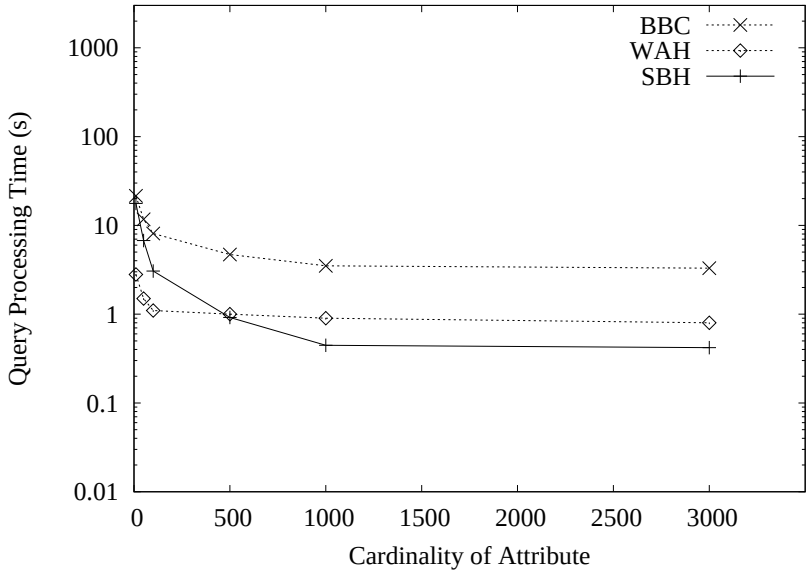
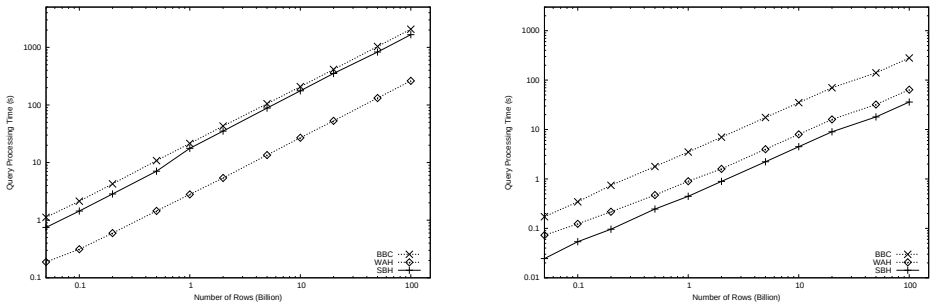


Figure 4.10 Tendency of query processing time with cardinality on synthetic data (one billion rows).

mostly considers only literals so that the query processing time gets lower. Thus, SBH may provide poor query processing time of the scheme in the low cardinality range, which is not as good as our expectation. Nonetheless, when the bitmap indexes are in range of high cardinality where SBH compression could be efficiently applied, we come to the conclusion that query processing time of SBH is also efficient.

To summarize, when the cardinality of bitmap is less than 50, all the techniques have similar sizes for the compressed bitmap indexes and also similar query processing times, because bitmap indexes are not compressed well in this interval. Thus, better performance of the SBH compression scheme comes when the cardinality of bitmap is above 50.



(a) 10 cardinality.

(b) 1,000 cardinality.

Figure 4.11 Tendency of query processing time on synthetic data, along with the number of rows.

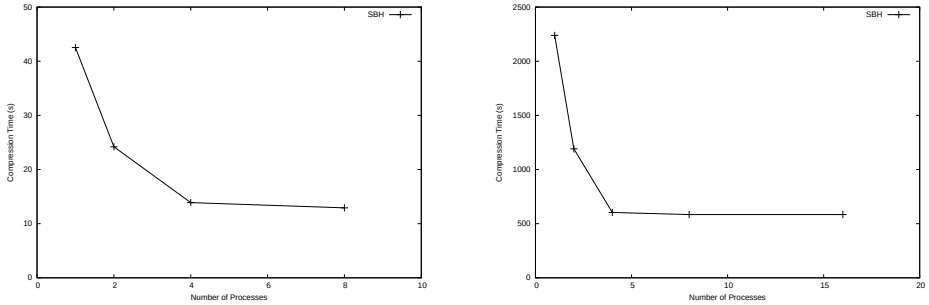
4.4.2 Parallelized Version

We implemented the SBH scheme in parallel fashion, both for CPU and GPU. In this subsection we compare the performance of the optimized algorithm, with the plain version (CPU) or the two WAH-based competitive algorithms (GPU).

CPU Parallelism

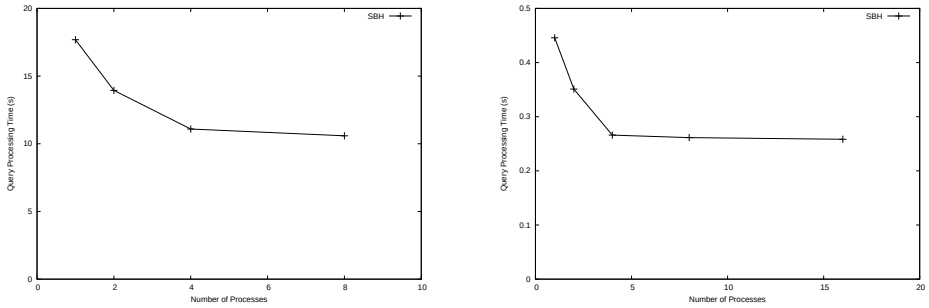
Figures 4.12 and 4.13 exhibit the compression time and the query processing time, respectively. We performed 32 `OR` operations per query, so we measured both the compressing and query processing performance until $n_c = 8$ when cardinality is 10.

We can see from Figure 4.12 that the compression time follows inversely proportional relationship to the number of concurrent processes. As aforementioned, this is mainly because the main compression process is independent of other concurrent executions. One of the reasons that the tendency is not completely inversely proportional is that disk accesses are needed to store the compressed result, that are dependent among the concurrent processes. Unfortunately, the CPU used in the experimental environment contains 4 cores, so the performance gain of parallelism gets diminished when $n_c > 4$.



(a) 10 cardinality. (b) 1,000 cardinality.

Figure 4.12 Compression time of synthetic data (one billion rows) with respect to the number of CPU processes.



(a) 10 cardinality. (b) 1,000 cardinality.

Figure 4.13 Query processing time of synthetic data (one billion rows) with respect to the number of CPU processes.

Query processing time shown in Figure 4.13 also satisfies the correlation explained above. For all cardinality ranges, the parallelism reduces query processing time compared to the plain version, until n_c becomes closer to the number of cores. The super-bucket decompression in the plain version already provides skipping functionality, meaning that the entire bitmap index needs not be decompressed while processing relevant queries. This property is also valid in the parallel version, but keeping the flags hinders the consecutive processes in the decompression phase.

Algorithm	Compression Time (s)	Query Processing Time (s)
GPU-SBH	27.03	0.889
GPU-WAH	1.324	3.228
GPU-PLWAH	1.459	5.793

Table 4.1 Comparison of synthetic data with cardinality 10 (one billion rows).

Algorithm	Compression Time (s)	Query Processing Time (s)
GPU-SBH	224.5	0.682
GPU-WAH	83.72	1.295
GPU-PLWAH	88.24	1.528

Table 4.2 Comparison of synthetic data with cardinality 1,000 (one billion rows).

GPU Parallelism

We compare the performance between the two aforementioned algorithms for GPU – GPU-WAH and GPU-PLWAH. Tables 4.1 and 4.2 show compression and decompression time occurred to encode the synthetic bitmap indexes. For all the algorithms, default parameters for parallelization are used. The two compared algorithms first compute the information in the bitmap index while modifying content of the index, whereas our GPU version does not alter the original bitmap index and outputs compressed result to RAM. Therefore, GPU-SBH does not have a-priori information available for fast compression process, giving relatively poor compression performance.

Fortunately, instead of the entire decompression of the queried bitmap indexes needed for query processing in GPU-WAH and GPU-PLWAH, GPU-SBH has skipping functionality which is handled by shared memory in a CUDA block. This gives at most 4 times faster query processing time than GPU-WAH, or 7 times faster time than GPU-PLWAH. Note that GPU-PLWAH involves conversion of the bitmap indexes into WAH before queries.

It is worth notable that the query processing time for high cardinalities takes

longer than that of the CPU, because all the algorithms involve data transfers between CPU and GPU. In contrast, the GPU compression time takes lesser than CPU for at most 80%, performance boost on the parallelized compression is remarkable.

4.4.3 Summary

- In all three techniques, the time to compress bitmap indexes increases linearly with the cardinality of the indexed column.
- The time to compress bitmap indexes is also proportional to the number of rows in all three techniques. For cardinalities less than 50, the compression ratio of all techniques is not high, i.e., all three schemes do not achieve good compression. For this reason, query processing time in all techniques is high.
- SBH scheme gets better performance for cardinalities larger than 50, and its compressed index size is lower than that of the other schemes, and is comparable to that of BBC.
- Parallelism for the SBH scheme gives the performance boost, close to either the number of concurrent processes for CPU.
- GPU-parallelized version of the SBH scheme gives better query processing time than the compared schemes.

Chapter 5

Space-efficient Representation of Semi-structured Document Formats

This chapter elaborates the core contribution of this dissertation: representing diverse semi-structured documents in space-efficient manner. Concepts discussed in the previous chapters are integrated in this representation. This chapter includes an extended work of Anjos et al. [7].

5.1 Preliminaries

5.1.1 Semi-structured Document Formats

Data exchange (also called data interchange) is the process of taking data structured under a source schema and transforming it into data structured under a target schema, in a systematic way such that the target data are an accurate representation of the source data [43]. A data exchange language, or format, is a language that is domain-independent and can be used for diverse types of data. Its semantic expression, capabilities, and qualities are largely determined in comparison with the capabilities of natural languages. A common charac-

teristic of such formats is that they can be parsed and correctly interpreted independently of programming language, running environment, and platform.

A realization of the data exchange language is known as semi-structured data [23]. Semi-structured datum, in general, is a form of structured data that does not follow tabular data model which is mainly used in the relational databases, but contains various forms of marker character to distinguish semantic elements. These markers control hierarchical information residual in the original data. A core property of semi-structured data is that the entities with equivalent class may include different sets of attributes.

Semi-structured document is a textual document that includes semi-structured data. Several types of semi-structured document formats exist, which we introduce some of those below.

XML

XML (eXtensible Markup Language) is a text-based data exchange language derived from SGML (Standard Generalized Markup Language), being a simpler alternative that is both human-readable and machine-readable [19]. While its original goal is to meet the challenges of large-scale electronic publishing, XML also plays a major role in the data exchange and storage of a wide variety of systems and web services. This position is attained primarily due to its higher level of application-independence compared to other data interchange formats of its time.

An XML document is a string of unicode characters. These are then divided into markups and contents. Markups are strings that begin with a less-than sign (<) and end with a greater-than sign (>). A component in the document is called an element, which has a start tag in the front and an end tag in the end. Inside the start tag, name-value pairs called attributes can exist.

Figure 5.1 depicts an example XML document.

One of the advantages of the XML format is that it is possible to check the

```
<Books>
    <Book ISBN="055321419">
        <title>Sherlock Holmes: Complete Novels</title>
        <author>Sir Arthur Conan Doyle</author>
    </Book>
    <Book ISBN="0743273567">
        <title>The Great Gatsby</title>
        <author>F. Scott Fitzgerald</author>
    </Book>
    <Book ISBN="0684826976">
        <title>Undaunted Courage</title>
        <author>Stephen E. Ambrose</author>
    </Book>
    <Book ISBN="0743203178">
        <title>Nothing Like It In the World</title>
        <author>Stephen E. Ambrose</author>
    </Book>
</Books>
```

Figure 5.1 Example XML document [79].

validity of a document with respect to a given schema. More than one XML documents can be associated with a single DTD (Document Type Definition). DTD is a schema language that contains a set of markup declarations defining elements and attributes, which in turn can be tested against XML documents for validity. Based on the schema, the structure of an XML document could be modeled into a tree of components, namely elements, attributes, and textual data. Due to this characteristic, XML is also known as a semi-structured data interchange format.

Furthermore, there are two major query languages for working with XML endorsed by the W3C. XPath is a query language for selecting nodes from an XML document using location paths that resemble tree navigation [34]. XQuery

is a more functional language that is designed to query and transform collections of data in XML documents [44].

Although advantages exist, the XML data exchange format is utterly verbose, cluttering the resulting file with nonessential structural information and metadata, which hinders the efficiency of the language in terms of usability, readability, and compactness. In an attempt to mitigate this size overhead, several algorithms implementing XML compression have been suggested in the literature, including XMill [77], TREECHOP [73], XQueC [10] and XBZipIndex [48]. These compressors are commonly classified with respect to whether or not the resulting compressed file support queries as it is; and whether structure and data content are stored alongside or separately. An XML compression method is called non-queryable if it necessitates the entire XML document to be decompressed before querying can take place. Homomorphic compressors encode both the structure and content of a document in a single container, while permutation-based compressors try to improve the compression ratio by differentiating the structural and content sections of a document.

For parsing XML documents, some libraries such as pugixml [68] provide DOM (Document Object Model)-like interface to manipulate the original document, though for pugixml it only supports documents that could be fit in main memory and it does not reduce the size of the processed representation. The SiXDOM library suggested by Delpratt et al. [41] utilizes succinct data structures while constructing the DOM structure in RAM, which supports some navigational queries. However, this library does not consider storing components other than the tree structure.

JSON

JSON (JavaScript Object Notation) is an open standard document format that uses human-readable structured text to represent data objects [18]. Designed as an alternative to XML, JSON is originally based on a subset of the JavaScript

programming language. Even though its name includes a specific language, JSON is a programming language-independent format widely used nowadays to exchange data on the web and to represent structured information.

The JSON interchange format is designed around two types of entities – objects and arrays. An *object* is an unordered list of name-value pairs, i.e., an associative array. Names are strings, and values are one of the possible JSON value types. Objects are wrapped around curly brackets ("{" and "}"), and successive name-value pairs are separated with commas (","). Though the specification states that pairs inside an object are, in fact, unordered, JSON parsers commonly assume some implicit ordering. There could be pairs with identical names inside an object, and in our representation we explicitly allow it. An *array* is an ordered list of values. It is enclosed in square brackets ("[" and "]") and subsequent values are separated by comma. Values in an array do not have associated names.

Core types of JSON values, beside the two s discussed above, include *number* (integer and real), *string*, *boolean* and *null*. Figure 5.2 shows an example of a small but complete JSON document.

Similar to XPath for XML, there are some standards for JSON querying. Google devised a functional query language Jaql [16] which is based on a flexible data model inspired by JSON, supporting manipulation of arrays and user-provided functions. A statement consists of a source, a sink, and pairs of an operator and a parameter. Some of the operators this language supports are `FILTER`, `GROUP` and `JOIN`. JSONiq [52] is another functional query language which can also process unstructured documents. This language has two different behaviors: its independent syntax and XQuery-like grammar. Since JSONiq is highly influenced by XQuery, its data model and list of supported queries resemble those in XQuery.

As opposed to XML, there are no well-known compression schemes to both encode and query JSON documents. Additionally, existing JSON libraries, such

```
{
    "id": 35420,
    "name": "Toaster",
    "tags": ["Kitchen", "Appliances"],
    "price": 32.99,
    "on_sale": true,
    "stock": {
        "warehouse": 300,
        "store": 20
    }
}
```

Figure 5.2 Example JSON document.

as JSONC [26], naively apply well-known generic text compression methods (e.g, gzip [117]) in a JSON document, so that the entire document needs to be decompressed for querying. Another approach commonly used when transferring JSON files is stripping unnecessary whitespaces in a process called JSON minification. This, however, does not constitute an actual compression technique. BSON [1] derived from MongoDB aims high processing speed by encoding documents in binary format and including additional information for traversals. Nevertheless, although this may reduce size of a JSON document, no actual compression is performed during the conversion. Thus, this representation also does not compose a compression scheme.

The most popular arguments in favor of XML are around the benefits of its interoperability and openness. However, none of these are inherent to XML itself. JSON offers the same qualities while improving in a number of aspects, especially with respect to conciseness, human-readability and ease of parsing and processing by a machine. JSON represents data as collections of arrays and records, which is what data actually are. XML represents data based on elements, attributes, content text, entities, and other metadata. Furthermore,

XML is document-oriented, while JSON is data-oriented. Data-oriented formats can be more easily mapped to object-oriented systems.

YAML

YAML (YAML Ain't Markup Language) is a human-readable readable data serialization language [14]. This language is also designed as an alternative of XML, targeting communication applications but only with a minimal syntax. Other design goals include portability between programming languages, one-pass processing and easy implementation.

YAML achieves cleanness by minimizing the amount of structural characters, and allowing the data to show themselves in a natural way. This format utilizes colons (":") for separating key-value pairs, and dashes ("-") for representing lists. YAML focuses on encoding scalar data (strings and numbers), sequences (lists and arrays), and mappings. Nesting is implemented as whitespace indentations inside a YAML document. Other custom data formats are also able to be encoded by a combination of those primitives.

Two major differences between YAML and other semi-structured document formats are structures and data typing. Structures assist storing multiple documents inside a single file, and data typing allows users to explicitly define a designated type for an element in the document.

Recent YAML specification officially defines the format as a superset of JSON. Some of the functionalities YAML additionally supports include comments, extensible data types, relational anchors, strings without quotation marks, and mapping types preserving key order. Therefore, a JSON document could be considered as a valid YAML document. In contrast, the specification claims that YAML has no direct correlation between XML. For instance, YAML – as well as JSON – does not support attributes inside tags which are native in XML.

Figure 5.3 shows an example YAML document.

```
---
Time: 2001-11-23 15:01:42 -5
User: ed
Warning:
  This is an error message
  for the log file
---
Date: 2001-11-23 15:03:17 -5
User: ed
Fatal:
  Unknown variable "bar"
Stack:
  - file: TopClass.py
    line: 23
    code: |
      x = MoreObject("345\n")
  - file: MoreClass.py
    line: 58
    code: |-
      foo = bar
```

Figure 5.3 Example YAML document [14].

As of current, no specific query language dedicated to the YAML document format exists. Instead, the specification considers node graphs and event trees as an internal information model to process a YAML document. In addition, there exists no YAML-specific compressors to efficiently compact documents.

5.1.2 Resource Description Framework

One of the popular data models which is represented in the semi-structured document formats is RDF (Resource Description Framework) [3]. RDF is considered as a general method to describe and model web resources. The main

RDF resource is structured as a *Subject-Predicate-Object* triple. Subjects denote the resource, while predicates describe traits or aspects of the resource and show relationship between subjects and objects. A collection of RDF could be emulated into labeled, directed multigraph.

The subject of an RDF entity is either a URI (Uniform Resource Identifier) or a blank node, indicating resources. Resources denoted by blank nodes are called anonymous resources and are not directly identifiable. The predicate is a URI which also indicates a resource, representing a relationship. The object is a URI, blank node or a unicode string literal.

RDF itself is not a serialization format: it is rather an abstraction of entities and their relationship. Therefore, it is up to the specific semi-structured document format which determines rules of representation. Well-known serialization formats include RDF/XML [13], RDF/JSON [39], Turtle [12], and JSON-LD [100]. Out of those formats, RDF/XML is the initial representation of the resource description framework serialization, so it is often denoted as RDF itself. Nevertheless, note that this RDF serialization should be distinguished from the abstract RDF model. Throughout this dissertation we denote RDF as a serialization viewpoint.

Figures 5.4 and 5.5 are excerpts of an example RDF/XML and RDF/JSON serialization, respectively.

Various query languages to process RDF documents exist. SPARQL [2] is the predominant query language for RDF which resembles SQL syntax. This language is able to manipulate data stored in RDF format. Four different query variations exist in SPARQL:

- `SELECT`: extracting raw values from a SPARQL endpoint, in a table format.
- `CONSTRUCT`: extracting information from the SPARQL endpoint and transforming the results into valid RDF form.

```
<rdf:Description
rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="http://purl.org/net/dajobe/">
        </rdf:Description>
      </ex:homePage>
    </rdf:Description>
  </ex:editor>
</rdf:Description>
```

Figure 5.4 Example RDF/XML document [13].

- `ASK`: providing a boolean result for a query on a SPARQL endpoint.
- `DESCRIBE`: extracting an RDF graph from the SPARQL endpoint.

These query variations allow a set of analytic query operations – `JOIN`, `SORT` and `AGGREGATE` – if the schema is intrinsic from the original data.

Other RDF query languages include RDQL [97] and RQL [69]. When serialized in semi-structured document formats, connected query languages could be to query the structure of the RDF serialization. For example, XQuery is able to handle RDF/XML-serialized documents.

Several approaches to efficiently compress RDF serialization are introduced in literature. Cure et al. [38, 37] devised WaterFowl, a compact, self-indexed RDF store. This serialization is based on the utilization of succinct data structures which we introduce shWortly, following the main idea of this dissertation. Brisaboa et al. [21] consider a different type of self-index that reduces space usage while supporting basic SPARQL queries.

```
{
  "_:anna" : {
      "http://xmlns.com/foaf/0.1/name" :
      [ { "value" : "Anna",
        "type" : "literal" } ],
      "http://xmlns.com/foaf/0.1/homepage" :
      [ { "value" : "http://example.org/anna",
        "type" : "uri" } ]
  }
}
```

Figure 5.5 Example RDF/JSON document [39].

5.1.3 Succinct Ordinal Tree Representations

Succinct data structures are fundamentally based on representing elements of a given set in a compact form, in such a way that operations on its domain can still be executed efficiently [63]. In general, succinct data structures aim for representing instances of a set using space as close as possible to the information-theoretic lower bound, while still supporting operations efficiently.

We outline two space-efficient ordinal tree representations – Balanced Parentheses (BP) and Depth-First Unary Degree Sequence (DFUDS). Both achieve the optimal space of $2n$ bits for representing ordinal trees (since there are $C_n = \frac{1}{n+1}\binom{2n}{n}$ ordinal trees on n nodes, we need at least $2n - O(\log n)$ bits to encode an arbitrary ordinal tree on n nodes), and are able to perform a number of tree operations efficiently with the aid of `rank` and `select`, and balanced parentheses auxiliary structures in total $2n + o(n)$ bits of space. A summary of some of the most significant tree operations is given in Table 5.1 [11].

The BP tree representation is first proposed by Jacobson [62] and later improved by Munro and Raman [87]. In this method, a balanced parentheses bit sequence is constructed from a depth-first traversal of the tree, by writing

Tree Operation	Description
$pre\_rank(x)$	preorder rank of node x
$pre\_select(p)$	the node with preorder p
$isleaf(x)$	whether node x is a leaf
$ancestor(x, y)$	whether node x is an ancestor of y
$depth(x)$	depth of node x
$parent(x)$	parent of node x
$first\_child(x)$	first child of node x
$next\_sibling(x)$	next sibling of node x
$subtree\_size(x)$	number of nodes in the subtree of node x
$degree(x)$	number of children of node x
$child(x, i)$	i-th child of node x
$child\_rank(x)$	number of siblings to the left of node x

Table 5.1 Operations on ordinal trees [11].

an opening parenthesis when first arriving at a node, and a closing parenthesis after visiting all of its children, namely all nodes in its subtree. In this way every node has exactly two parentheses associated with it: an open parenthesis "`(`" and a close parenthesis "`)`". Thus, this encoding represents a tree with a bit string composed of $2n$ balanced parentheses. This representation uses space that is within lower-order terms of the information-theoretic lower bound (of $2n - O(\log n)$ bits) for encoding trees.

To support operations in this tree representation we then need to make use of the auxiliary structures equipped with `rank`, `select`, and balanced parentheses. Notice that in this encoding nodes of a subtree are stored contiguously in the designated bit string. Therefore, the subtree size can be computed by simply taking half the distance between the opening and closing parentheses that correspond to a node.

From the core operations provided by the `rank`, `select`, and balanced paren-

theses structures we can derive several tree operations efficiently. In fact, it is known that all of the core tree navigational operations presented in Table 5.1 can be performed in $O(1)$ time utilizing this encoding.

The DFUDS tree representation [15, 64] is an alternate approach to LOUDS (Level-Order Unary Degree Sequence) [62] and BP. To combine the virtues of these two representations, DFUDS writes a unary degree sequence of each node in a depth-first traversal of the tree. That is, whenever we arrive at a node during a depth-first traversal, we append d open parentheses and one closing parenthesis, where d is the number of children of the node being visited. A node is represented by the position of its first open parenthesis.

With the addition of one artificial open parenthesis prepended at the beginning of the bit string, the resulting encoding is also a balanced parentheses bit sequence. As a result, each node has exactly two bits corresponding to it. A 1 bit (open parenthesis) is written in the bit string when visiting its parent, and one 0 bit (close parenthesis) is written in the bit string when visiting the node itself. This generates a $2n$ length bit string, which is again within lower-order terms of the information-theoretic lower bound for representing a tree on n nodes.

Tree operations can then be supported with an additional $o(n)$ bits of space through auxiliary structures. As it was the case with the BP representation, nodes of a subtree are stored contiguously in the bit string generated through the DFUDS representation. Thus, the subtree size can be computed by simply taking half the distance between the opening and closing parentheses that correspond to a node.

From the core operations on parantheses, we can derive several tree operations efficiently. In fact, all the tree operations presented in Table 5.1 can be performed in $O(1)$ time, using this set of succinct ordinal tree representations along with auxiliary data structures. Arroyuelo et al. [11] provide emulation of navigational queries to preliminary operations supported in auxiliary

Tree Operation	BP	DFUDS
$pre\_rank(x)$	$rank_{open}(x)$	$rank_{close}(x-1)+1$
$pre\_select(p)$	$select_{open}(p)$	$select_{close}(p-1)+1$
$isleaf(x)$	$S[x+1]='('$	$S[x]='('$
$ancestor(x,y)$	$x \leq y \leq findc(x)$	$x \leq y \leq findc(enclose(x))$
$depth(x)$	$excess(x)$	–
$parent(x)$	$enclose(x)$	$prev_{close}(findo(x-1))+1$
$first\_child(x)$	$x+1$	$child(x,1)$
$next\_sibling(x)$	$findc(x)+1$	$findc(findo(x-1)-1)+1$
$subtree\_size(x)$	$(findc(x)-x+1)/2$	$(findc(enclose(x))-x)/2+1$
$degree(x)$	–	$next_{close}(x)-x$
$child(x,i)$	–	$findc(next_{close}(x)-i)+1$
$child\_rank(x)$	–	$next_{close}(y)-y; y=findo(x-1)$

Table 5.2 Operation details of tree operations in BP and DFUDS representations [11]. A dash is used to indicate operations that require additional auxiliary structures.

data structures. Table 5.2 summarizes those operations. $findc$ and $findo$ operations find the position of matching close and open parenthesis of a parenthesis, respectively. *excess* operation finds the difference between the number of open and closing parenthesis before a position. *enclose* operation in an open parenthesis returns the position of the open parenthesis corresponding to the closest matching parenthesis pair enclosing the input open parenthesis. For the DFUDS representation, $prev_{close}(x) \equiv select_{close}(rank_{close}(x))$ and $next_{close}(x) \equiv select_{close}(rank_{close}(x)+1)$.

Figure 5.6 shows an example ordinal tree, along with its BP and DFUDS represented tree structure.

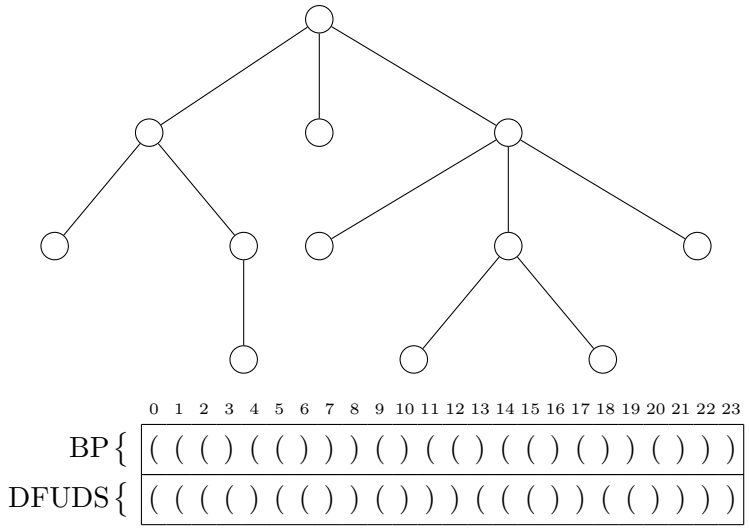


Figure 5.6 An ordinal tree with the succinct representations.

5.1.4 String Compression Schemes

General-purpose lossless compressors such as LZ77 [117], LZ78 [118], and LZW [110] perform dictionary-based encoding to support compression. These algorithms substitute contiguous length of text into the location of entry inside the dictionary. Although dictionary-based compression algorithms may provide high compression ratio, if one needs to randomly access and manipulate the encoded content, the whole sequence of it should be decompressed again, which can be a significant computation overhead. To alleviate this handicap, string compression schemes have been suggested.

In the early ages, indexable data structures such as suffix trees [109] and suffix arrays [81] are widely used to deal with string compression. Suffix trees are compressed tries containing all the suffixes of the given text as their keys and positions in the text as their values. The given text is terminated with a special character \$, which is considered the lexicographically smallest. Construction of a suffix tree takes $O(n)$ time, where n is the length of the given text. Each constructed suffix tree occupies $O(n)$ space. When a suffix tree is constructed,

string search and finding the longest substring queries can be done in either $\Theta(m)$ or $\Theta(n)$ time, where m is the length of a substring.

Suffix arrays are sorted arrays of all suffixes of a given string. When the length of a string is n, construction time and space usage are both $O(n)$, similar to suffix trees. Note that suffix arrays can be constructed by performing a depth-first traversal of the relevant suffix tree. Locating every occurrence of a substring pattern in the string using suffix arrays takes $O(m \log n)$ time, where m is the length of the pattern. Constructing compressed suffix arrays [59] also takes $O(n)$ time, and when compressed space usage becomes $O(nH_k(T)) + o(n)$. The operation to query a pattern in the compressed array takes $O(m)$ or $O(m+\log n)$ time.

Wavelet trees [58] and wavelet matrices [36] are also used as text indexing schemes. Wavelet tree is a succinct data structure for strings, and it supports `access` as well as `rank` and `select` operations for an alphabet in $O(\log \sigma)$ time, where σ is size of the alphabet. A string S occupies $nH_0(S) + o(|S| \log \sigma)$ bits when encoded using compressed wavelet trees. Wavelet matrix is an alternative representation of the balanced wavelet tree, also taking $nH_0(S) + o(|S| \log \sigma)$ bits when compressed.

Improving the conventional string compression algorithms, a plethora of data structures known as compressed string dictionaries have been suggested in literature. A string dictionary is a data structure acting as a bi-directional mapping between strings and integer identifiers. This data structure supports at least the following two operations:

- *string-to-ID*: given a string, locate its ID in the dictionary.
- *ID-to-string*: given an integer ID, extract its corresponding string in the dictionary.

Ferragina and Manzini proposed a text indexing scheme known as the FM-index [49]. This index relies on BWT (Burrows-Wheeler Transform) [24]. BWT

is a reversible transformation for strings for the preparation of efficient compression. While the transformation itself does not reduce the size of the string, it is able to convert the string to runs of repeated characters, feasible to be compressed using run-length encoding schemes. FM-index supports counting and locating operations in $O(p)$ and $O(p + occ \log^{\epsilon} u)$ time, respectively, where p is length of a pattern, u is length of a text, occ is number of the pattern occurrence and $0 < \epsilon < 1$ is an arbitrary parameter.

In addition, a compressed representation of tries known as XBW [47] is used to compress strings. This as well as the later version of the FM-index [50] utilize wavelet trees to reduce space for large alphabets. XBW structure consists of two parts when l internal nodes exist in the trie. An array is first constructed, containing l sorted suffixes, First part is a sequence storing the labels of the trie edges leading to the children of each internal node, considering the order of the array. Second part is a bit sequence marking the last child of each of the l internal nodes in the first part.

Martinez-Preto et al. [82] introduced and experimented a number of compressed string dictionary schemes. Their data structures are mostly based on front coding schemes. Kanda et al. [67] suggested a type of compressed string dictionary based on double-array trie. They used XOR-based compression scheme to further encode string dictionaries. Finally, Brisaboa et al. [20] combined hierarchical front coding scheme and LCP algorithm used in suffix arrays.

5.2 Representation

This section further explores details of the representation components and related data structures. The goal of this section is to suggest a compact, space-efficient representation for diverse semi-structured document formats, exploiting bit strings and succinct ordinal tree data structures.

Before moving on, although the description in this section is mainly based on compacting JSON documents, we claim that this representation could be easily

generalized into expressing XML and YAML document formats described in Section 5.1.1. This is because both semi-structured formats follow the identical components as those of JSON – content and DOM tree.

5.2.1 Bit String Indexed Array

In contiguous homogeneous arrays all entries are of a single type, and hence have the same fixed size in bytes. In such arrays, indexing is easily computed from the array's starting position and the length of each array element. In heterogeneous arrays, however, auxiliary structures are required for efficient indexing, as the size of entries is variable. A common technique used in modern programming languages to provide the illusion of heterogeneous arrays is to use a homogeneous array of pointers to elements. Each element pointed to may be of a different type, but the array is still homogeneous. This approach has the downside of requiring additional pointers, which, in modern 64-bit computers, correspond to 8 extra bytes per array entry.

We propose an indexing scheme based on bit strings along with the `select` auxiliary structure, which we denote as *bit string indexed array*. Consider a contiguous heterogeneous array A with n elements and a total size of m bytes. We generate a bit string S of length m bits such that the i-th bit is set if and only if the i-th byte of A corresponds to the beginning of one of its elements. Notice the bit string S as generated above has exactly n set bits, and occupies a total $m/8$ bytes. Now the problem of indexing the i-th entry of a bit string indexed array is reduced to a call to $select_1(S, i)$, that is, the position of the i-th 1 in S.

Figure 5.7 depicts how the bit string indexed array is structured on a sample heterogeneous array. In this example *boolean*s occupy 1 byte, *number*s take either 1, 2, 4 or 8 bytes based on their capacities.

The overall space overhead incurred by this scheme is m bits for the bit string S and extra $o(m)$ bits for the auxiliary `select` structure. More precisely

	0	1	2	3	4	5	6	7	8	9	10
S	1	0	1	0	0	0	1	1	0	0	0
A	2189		3.141592				T	322000			

Figure 5.7 A bit string index built on top of the example heterogeneous array $A = \{2189, 3.141592, \text{true}, 322000\}$.

the bit string index requires $m + o(m)$ extra bits in addition to the input array.

If more compression is needed, one can encode the index using *sdarray* [90], which is also used in Chapter 3. Since we can assume the bit vector is sparse (i.e., number of set bits are extremely smaller than unset bits), *sdarray* structure efficiently encodes the index in $n(2 + \log \frac{m}{n})$ bits, while supporting `select` queries in $O(1)$ time.

5.2.2 Main Structure

One of the main points in which this representation improves memory usage compared to other libraries lies in the fact that we devise a compact variable-length encoding for elements. In order to store a series of variable length encodings, we design a memory-efficient heterogeneous array discussed in the previous section. This array is in turn used to compose the underlying data structures used in this space-efficient representation.

In order to represent a given semi-structured document in a memory-efficient manner, we deconstruct the document tree structure portion from its content data. The two subdivisions are in turn separately encoded. This, in turn, allows us to leverage the characteristics and patterns particular to each specific data type, to achieve better memory usage.

We model the document structure using ordinal trees and implement encodings through the DFUDS succinct tree representation discussed in Section 5.1.3. Improvements in memory usage here derive from the fact that traditional libraries represent the tree structure through pointer-based implementations, in-

curring overhead of about 8 bytes per pointer in the semi-structured document in 64-bit systems. Succinct trees allow us to reduce this overhead to $2 + o(1)$ bits in our scheme.

A preliminary version of this representation [7] used the BP representation to represent the document tree, however this representation lacks efficient support for `child` and `degree` operations needed for querying the document, as denoted in Table 5.2. Therefore, we use the DFUDS ordinal tree representation to store the document tree in the library. If the BP representation is needed for operations such as `depth`, one can convert the existing DFUDS representation into BP representation, utilizing the technique of Farzan et al. [46]. This technique shows a way to extract either BP or DFUDS substring of an ordinal tree in $O(1)$ time. Nonetheless, we leave this conversion as a future work in this dissertation.

Given that the structure is dissociated from the document content, the raw data that remain are a series of names and values. These two components are also represented separately. According to the format specifications, names are exclusively strings which are repeatable among objects. Thus, names container may constitute lots of redundant entries. It is common for semi-structured documents with millions of nodes to have no more than a few dozen unique names. In our scheme, we strip redundancies in names and store the unique strings in a contiguous memory array. Values that have a name associated should encode with itself the index of its corresponding name.

Finally, a value can be any of the JSON types, and may or may not have a name associated with it. We encode a specific value according to its type as described in Table 5.3. All encodings start with a byte identifying its type, and whether the value has a name associated with it. If a JSON value has an attribute associated, its encoding includes an extra 8-byte index that identifies the corresponding entry in the attributes array. String values require special treatment as their lengths are highly variable. We store all strings in an array,

Type	Encoding	Size (bytes)
null	{type}	1
object	{type}	1
array	{type}	1
boolean	{type}	1
string	{type, index}	9+
number	{type, value}	$2, 3, 5, 9$

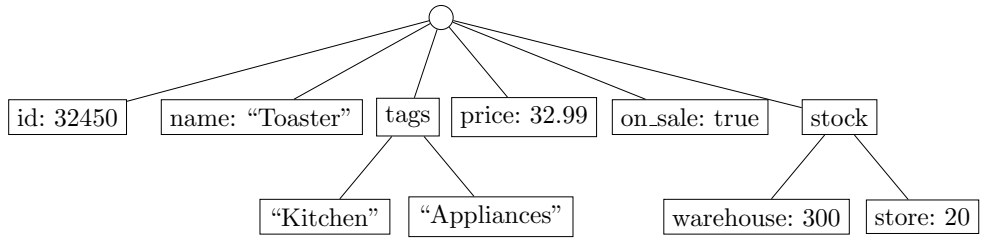
Table 5.3 Encodings of JSON types and respective sizes.

and the main representation only stores the index of its corresponding entry in an array `stringValues`. Number type encodings occupy 9 bytes for 64-bit numbers and 5 bytes for 32-bit numbers. Smaller values may use 16-bit numbers instead, for total encoding sizes of 3 bytes. Decimal numbers also follow this notation, depending on their precision.

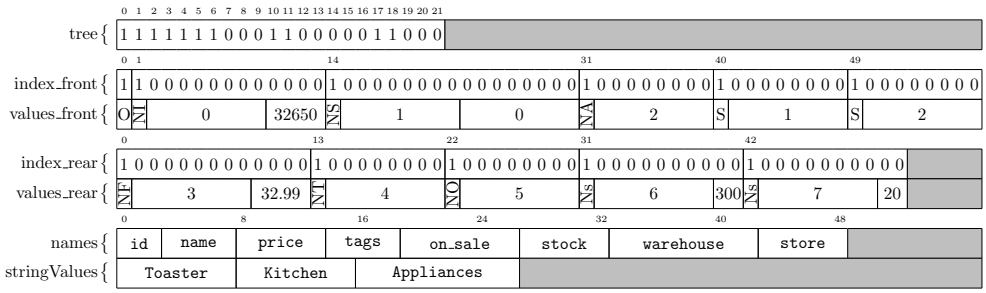
Initial version of the library [7] maintained strings in the `stringValue` array as-is. This allows easy extraction of relevant value in a pair. Nevertheless, since maintaining the original array does not actually involve compacting storage, we give an option to apply additional compression to this array. When space compaction needs to be considered in high priority, we provide apply additional compression schemes discussed in Section 5.1.4.

The list of value encodings is then stored in a bit string indexed array as outlined in Section 5.2.1 without further memory overhead. Each value is indexed by order of discovery in the depth-first traversal step performed when creating the succinct tree representation. That is, the i-th node in the succinct tree corresponds to the i-th element in the values array. This characteristic provides us with a lightweight and straightforward correspondence between tree nodes and associated data, based merely on array indexes.

Figure 5.8 depicts example document tree of the JSON file given in Figure 5.2 (top) and contains an illustration of the data structure generated by



(a) Document tree structure corresponding to the sample JSON file.



(b) In-memory representation of the sample JSON file.

Figure 5.8 Overall encoded representation of the sample JSON from file Figure 5.2.

this representation to process that document (bottom). `tree` and `index` are bit strings, while `values`, `names`, and `stringValues` are regular arrays. Entries in the `values` array start with a byte representing type of an element, where O, NA, NC, Ns, NI, NF, NS, and NT stand for *Object*, *Named Array*, *Named Char*, *Named Short*, *Named Integer*, *Named Float*, *Named String*, and *Named True*, respectively. Notice how named types include an additional 8-byte index to its associated entry in the attributes array. Note that the `stringValues` array is not converted to the compressed suffix array in these figures.

For manipulating other semi-structured document formats, we first provide a way to convert those documents into corresponding JSON documents. XML documents may contain attributes inside tags which JSON format does not support. In this case, we allocate additional type *Named Attribute*, storing the relevant key-value pair. For XML and YAML comment nodes, the representation adds a new *Comment* type. Those types are considered identical to *Named*

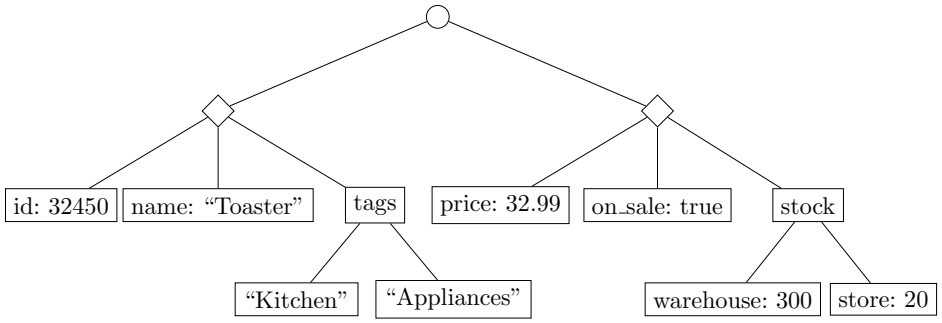


Figure 5.9 Document tree structure corresponding to the sample JSON document shown in Figure 5.2 divided into 2 chunks.

String.

Note that we use a byte to indicate a type in the bit indexed array, thus at most 2^7 different types can exist. Therefore, no extra space is needed to record the additional types.

5.2.3 Single Document as a Collection of Chunks

Inspired from *structure*s defined in YAML, this space-efficient representation also supports partial processing of a semi-structured document. Given a semi-structured document D with n elements (which is given a-priori), this representation can first be processed into c partial chunks of ordinal DOM tree and bit indexed array. Each chunk maintains about n/c elements, if the elements are evenly distributed with the same set of names. This collection of chunks is virtually connected by each other, in a separate ordinal tree.

Figure 5.9 visualizes result of chunk division applied in the sample JSON document ($c = 2$). Although depicted as a single tree, the representation consists of two levels. The major tree contains three nodes, a root node (circle) and two children nodes (square) pointing to a chunk. The compact representation is constructed per chunk: its DOM tree has a root node (square) and five descendant nodes (rectangle). When users need to query the constructed representation, the library first moves to the relevant chunk using the major

tree. Afterwards, it extracts and retrieves the relevant information from the bit indexed array and DOM tree residing in the navigated chunk.

This way of processing is useful when repetitive name-value pairs appear in the document. This ensures even distribution of chunk trees in the major tree, meaning that level of the tree gets not too deep. Additionally, if an input document is too big to be processed entirely in RAM, this chunk division assists processing it, by storing the intermediate chunks in disk.

5.2.4 Supporting Queries

From the query languages dealt in Section 5.1, it could be understood that it is indispensable for the document processing framework to support efficient traversals of the DOM tree as well as retrieval of the relevant name-value pairs. By utilizing the succinct tree structure as well as bit indexed arrays, our space-efficient representation suits those two core query objectives.

We support the following query operations in the library.

- `existsName(n)`: Given a string n, determine whether at least one element with name n exists in the document.
- `existsElement(n,v)`: Given a name n and value v, determine whether at least one element with name n and value v exists in the document.
- `countElements(n,v)`: Given a name n and value v, return number of elements with name n and value v.
- `listObjectNames(o)`: Given an JSON object o, return its list of names.
- `getObjectValue(o,n)`: Given an JSON object o, return value of an entity with name n. If multiple entities with identical names exist, return a list of values.
- `countArrayElements(a)`: Given an JSON array a, return its size.

- `getArrayValue(a,i)`: Given an JSON array a, return value of its i-th element.

For example, on the document in Figure 5.8, the answers to some of the queries are shown below:

- `existsName(‘‘warehouse’’)`: *true*
- `existsElement(‘‘warehouse’’,200)`: *false*
- `countElements(‘‘on_sale’’,true)`: 1
- `listObjectNames(0)`: id, name, tags, price, on_sale, stock
- `getObjectValue(8,‘‘store’’)`: 20
- `countArrayElements(3)`: 2
- `getArrayValue(3, 2)`: “Appliances”

In this dissertation we group the first three queries as *existence queries*, and the remaining four queries as *navigational queries*.

For `existsName` query, full scan of the `names` array is required. Nevertheless, since in the most real datasets the number of different names is lower than the number of distinct values, so the overhead of scan is not significant.

Performance of the `existsElement` and `countElements` queries is enhanced by maintaining bitmap indexes of the relevant names. In most cases when the number of possible elements is high, the bitmap index contains continuous 0s and sparse 1s, so it is feasible to apply bitmap index compression considered in Chapter 4. If the dedicated bitmap index is not constructed, then we naively need to scan the whole representation, which takes linear time to the number of elements in the tree.

The following formulae provide the emulation of navigational queries using the preliminary tree and bit indexed array operations.

- `listObjectNames(o)`:
- `getObjectValue(o,n)`:
- `countArrayElements(a)`:
- `getArrayValue(a,i)`:

The navigational queries first perform navigation of the succinct tree, using `child` or `parent` operations. Once the relevant node is located, the queries extract the desired information by either inquiring the bit indexed arrays or calling additional `degree` tree operation. Locating the exact place for answering queries in the bit indexed array takes theoretically constant time, by running `select` operation in the `index` array. Additionally, the DFUDS representation supports all of the aforementioned unit operations in theoretically constant time.

Therefore, the overall query time for the `listObjectNames` and `getObjectValue` operations is $O(c)$, where c is the number of children in the given object.

To summarize, a combination of those libraries enables efficient query processing of the semi-structured documents. Note that if users choose to compress the string values in a compressed suffix array, then extracting dedicated characters for answering queries takes $O(v)$ time instead of constant, where v is the length of the value.

5.3 Experimental Results

The library has been implemented in the C++ programming language and compiled with g++ 10.1.0. The environment in which the tests were executed features an Intel Core i7-6700K 4.20GHz CPU, 64GB DDR4 RAM, and 512GB NVMe drive. The machine runs Linux kernel version 5.8. RAM usage readings are done with `valgrind`, and elapsed time values for construction and querying are measured with the C++ STL library `chrono`.

The library borrows core concepts of the popular JSON processing library RapidJSON [102] while parsing a JSON document. For the two other semi-structured document formats, we attempt to convert the document into symmetric JSON-formatted document. If this is not possible, then we use our own parser library which is also engineered based on the RapidJSON library.

We make use of the SDSL [55] library to aid our implementation with bit strings and auxiliary structures, employing the `rank` structure as proposed by Vigna [106], `select` structure by Clark [33], balanced parentheses structure by Navarro and Sadakane [88] and compressed suffix arrays structure. Balanced parentheses structure is mainly used to query the succinct tree stored in bit strings. For boosting performance of the `exists` query, the plain implementation of bitmap index compression discussed in Chapter 4 is used.

We evaluate our scheme against three popular JSON libraries – JsonCpp [75], JSON for Modern C++ [78], and RapidJSON by measuring RAM usage and elapsed time during construction. Since those libraries equipped with processing JSON formats contain all the functionalities we support, thus we directly consider the JSON file format as the candidate of the main experiments.

With respect to compression, we compare our scheme against the original file size, blank-eliminated JSON file using JSONC [26], and gzip-applied [117] result. For evaluating querying performance, we also consider semi-indexing suggested by Ottaviano and Grossi [92].

In the following sections we denote *SSD* as our space-efficient representation of semi-structured documents.

5.3.1 Datasets

The experiments were performed on a collection of datasets of both synthetic and real world corpora. We generate synthetic datasets of single possible types in JSON (`array`, `bool`, `double`, `int`, `null`, `object` and `string`) with number of nodes from 2,000,000 to 100,000,000. With these datasets we intend to illustrate

Corpus	Nodes	XML Size (MB)	JSON Size (MB)
`Twitter`	3,249,499	168	90
`SNLI`	6,757,124	634	465
`Citylots`	13,805,883	575	181
`DBLP`	64,714,826	1,975	1,741
`150JS-evaluation`	420,358,521		4,910
`150JS-training`	878,277,103		10,247

Table 5.4 Overview of the real world datasets used in our experiments.

the performance behavior of the libraries on different value types. Since additional types introduced in XML and YAML can be emulated into either `string` or `object`, we only consider those types in the synthetic version of experiments.

More details of the real world corpora are described in Table 5.4. The original document is based on JSON format, and converted to XML format for equal comparison (without attribute-tag relationship). YAML size is of little difference to that of JSON, so it is not included in the table.

- `Twitter`: A list of 20,000 tweets and metadata collected in 2015.
- `SNLI` [17]: The Stanford Natural Language Inference corpus is a collection of human-written English sentences coupled with semantic metadata.
- `Citylots` [116]: This dataset is a JSON converted document of the CityLots spatial data layer, a representation of the City and County of San Francisco's Subdivision parcels.
- `DBLP` [42]: This dataset offers bibliography entries recorded in DBLP [76].
- `150JS` [94]: For 150,000 JavaScript files, their corresponding parsed AST (Abstract Syntax Tree)s are collected as two JSON documents: training (100,000) and evaluation (50,000).

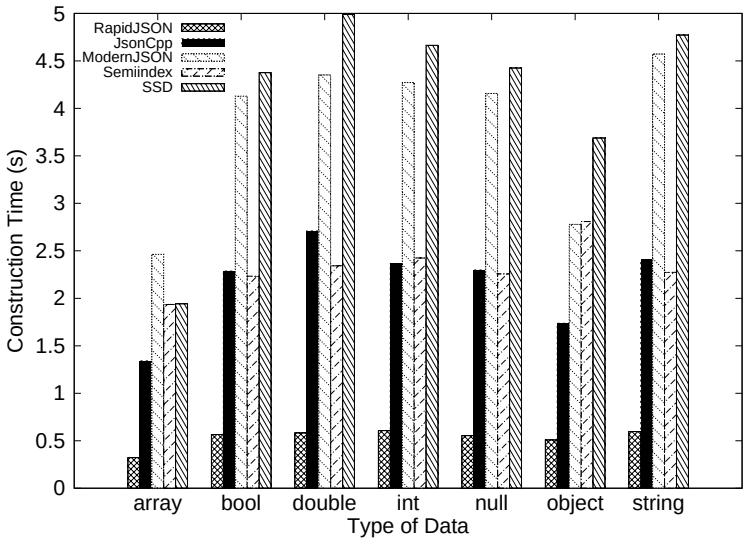
5.3.2 Construction Time

Figures 5.10 and 5.11 represent construction time of the libraries mentioned above. For all libraries, construction time includes reading a document file from disk and constructing auxiliary data structures based on that file in RAM.

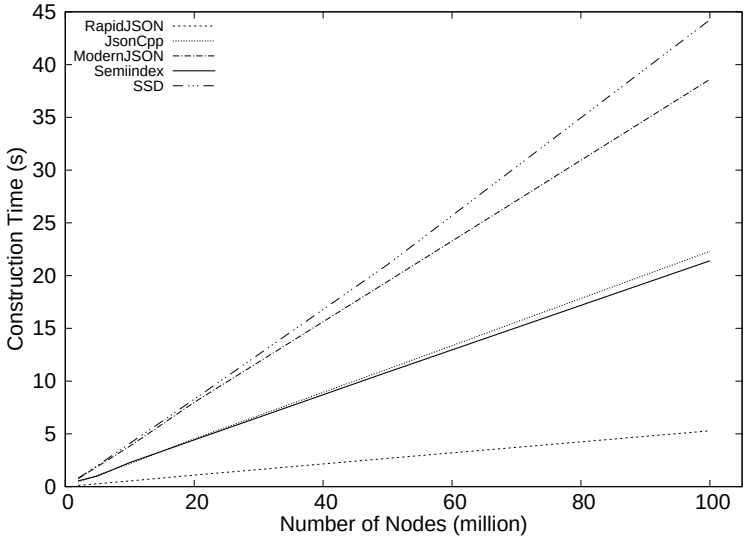
By examining the tendency of construction time on synthetic documents described in Figure 5.10b, it is clear that the time is proportional to the number of nodes. As our parsing scheme is derived from that of RapidJSON, we can observe that data structure construction takes most of the construction time. Establishing tree structure is concurrently done while traversing the document. Therefore, it is evident that preparing a succinct data structure for balanced parentheses needs to be further optimized for construction.

Although construction time is an important factor to process a semi-structured document in space-efficient manner, once the library supporting serialization (other than the original JSON document) parses the document it needs not reconstruct the whole representation. As mentioned later in Section 5.3.4, our library supports serialization and deserialization of the representation which significantly takes lesser time than construction shown above. Therefore, we consider that serializing the encoded representation to disk will mitigate demerits of slower construction time.

The experimental environment could not handle `150JS` corpora using third-party parsers because of insufficient RAM. Also, all the external libraries except RapidJSON could not run on the `DBLP` corpus, whereas our library is able to handle those documents as well. Since this is one of the merits of processing big data, we claim that our library has a strong point, suitable to handle larger documents, even when the amount of RAM available is small. It is worth mentioning that semi-index could also handle those documents since this library mainly focuses on constructing the tree structure while retaining the original document on disk, not actually constructing the parsed representation.



(a) $n = 10,000,000$.



(b) Varying n.

Figure 5.10 Construction time of the representation compared to different libraries, for synthetic data.

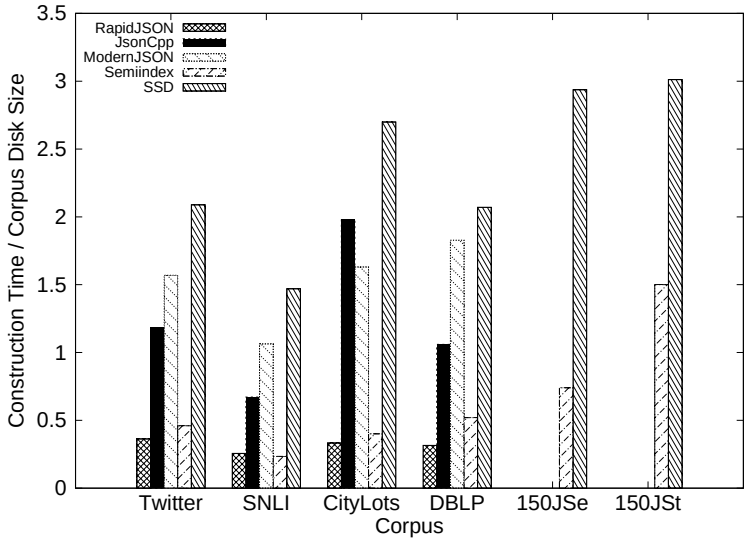


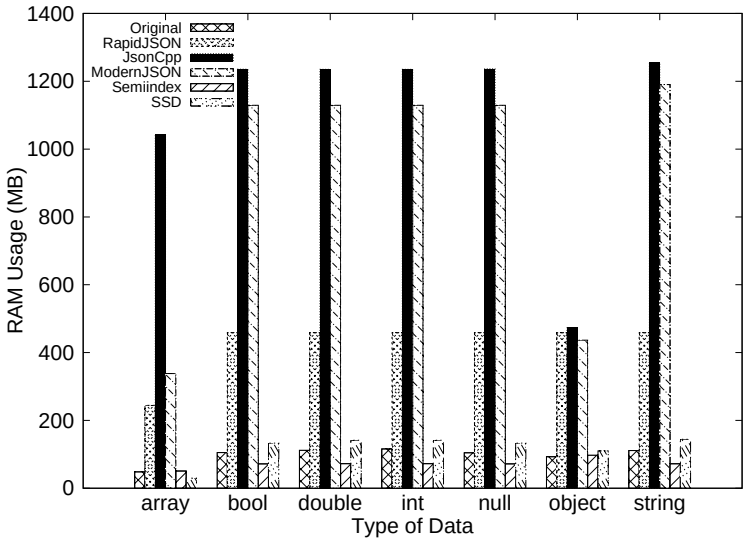
Figure 5.11 Relative construction time (with respect to corpus disk size) of the representation compared to different libraries, for real world corpora.

5.3.3 RAM Usage during Construction

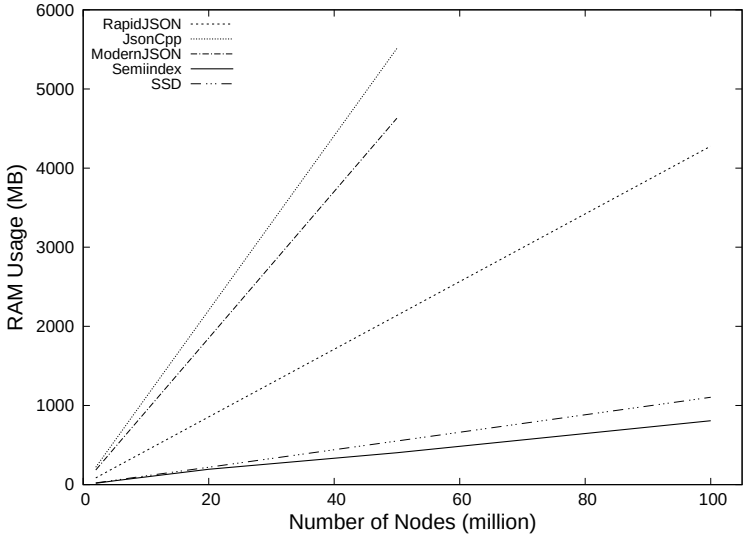
Figures 5.12 and 5.13 show the main memory usage of the library compared to JsonCpp, JSON for Modern C++ and RapidJSON. For visual comparison purposes, Figure 5.12 also includes the original file size on disk, while Figure 5.13 shows the relative ratio of the RAM usage compared to the original disk size.

It is evident from the experiments that RapidJSON performs best among the third-party libraries evaluated, and JsonCpp is the worst. Our library, mostly represents the input datasets in strictly less RAM than RapidJSON by up to 91% on synthetic data and 66% on real-world corpora, while outperforming JsonCpp by up to 98% and 84% on synthetic and real-world corpora, respectively. For corpora with pairs containing large string values, our library representation uses more space than RapidJSON, when compression is not applied.

Our scheme offers a significant improvement in memory efficiency by encod-



(a) $n = 10,000,000$.



(b) Varying n.

Figure 5.12 Memory usage of the representation compared to different libraries, for synthetic data.

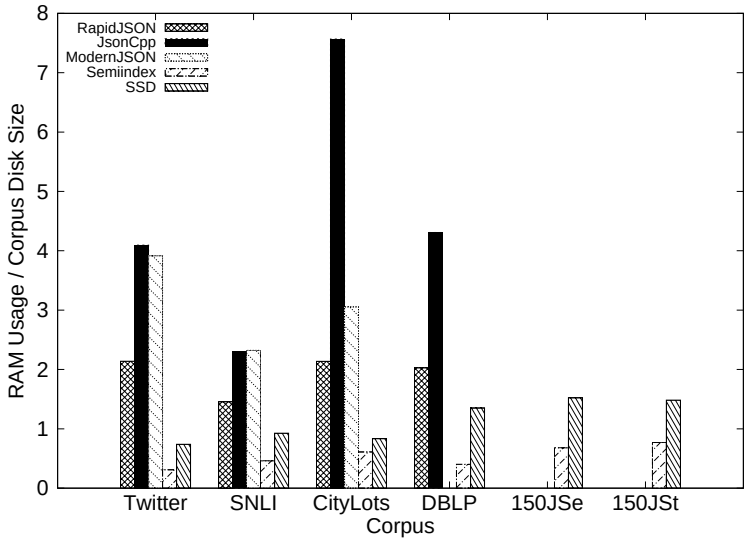


Figure 5.13 Relative memory usage of the representation (with respect to corpus disk size) compared to different libraries, for real world corpora.

ing values according to its type and data in a compact manner, using total memory proportional to the amount of information contained in a semi-structured file. On the other hand, common JSON libraries use fixed-length representations for all JSON values, leading to memory usage proportional to the total number of nodes. RapidJSON, for example, allocates 48 bytes for most values, regardless of type. Array entries are the exception, taking 24 bytes of memory. This explains why RapidJSON uses the same amount of memory for most synthetic datasets, except for array. JSON for Modern C++ and JsonCpp show similar behavior. Similar to construction time, RAM usage between the two succinct tree representations does not differ, reflecting the identical theoretic bound.

Compared to the previous version [7], the new representation uses about 30% more RAM in some of the synthetic corpora. This is we allocate 8 bytes instead of 4 for recording the IDs of the names and string values, to support

representing semi-structured documents with more than 2^{32} different possible strings. Nevertheless, by maintaining the string values efficiently in memory, representations of most of the corpora with strings use less RAM.

As mentioned in the previous section, all other libraries except ours could not process larger corpora.

5.3.4 Disk Usage and Serialization Time

In Figures 5.14 and 5.15 we illustrate the disk usage of our scheme compared to the original file size and to *gzip*. Our scheme is able to compress a document by up to 61% in synthetic files, and up to 28% in real-world corpora. From the figure, we can observe that disk usage is also proportional to the number of elements in the document. Our library effectively reduces file size especially for `array` and `object`.

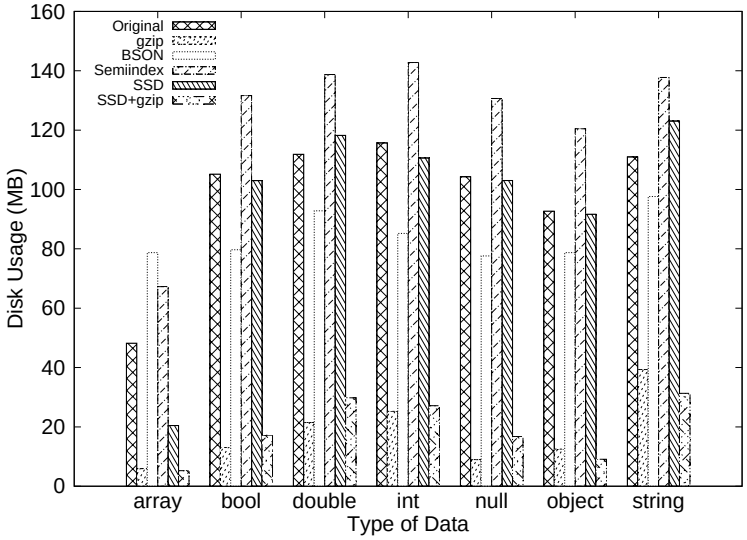
Although our compression is not as good as *gzip* with sizes about 2 to 9 times larger, it is easier to reload the compressed file generated by the scheme back to memory than to decompress and parse the *gzip*ped file. Note that once deserialized in RAM, we do not need to maintain the content stored on disk for future use. We also provide *gzip*ped result of the serialization, which further reduces the disk usage without penalizing the performance.

It is shown in the figure that BSON decreases disk usage by up to 33%. Unfortunately, the BSON library provided by MongoDB was not able to convert most of the real-world datasets, since it only supports UTF-8 characters.

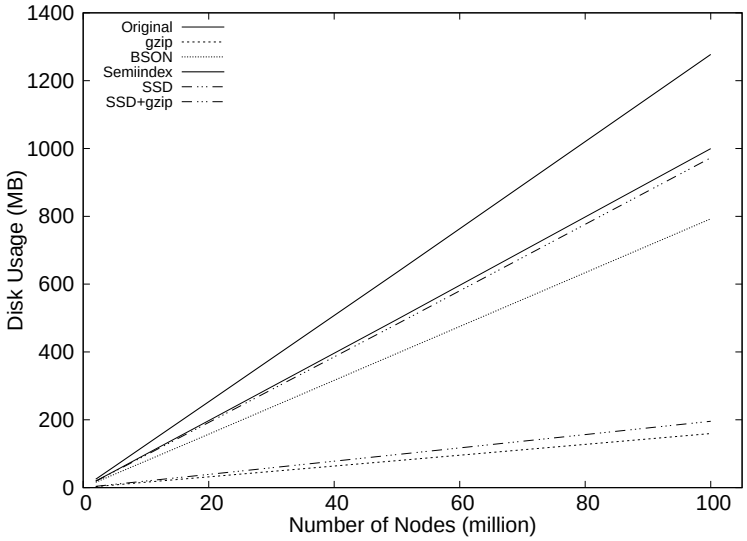
Table 5.5 summarizes serialization time of processed result. We can see that time needed to serialize corpora is proportional to their size.

5.3.5 Chunk Division

Even though maintaining large size documents is one of the merits of our representation, to improve the RAM usage further, we also tested by splitting a large semi-structured document into a collection of smaller chunks described



(a) $n = 10,000,000$.



(b) Varying n.

Figure 5.14 Disk usage of the representation compared to the original file size and to *gzip*, for synthetic data.

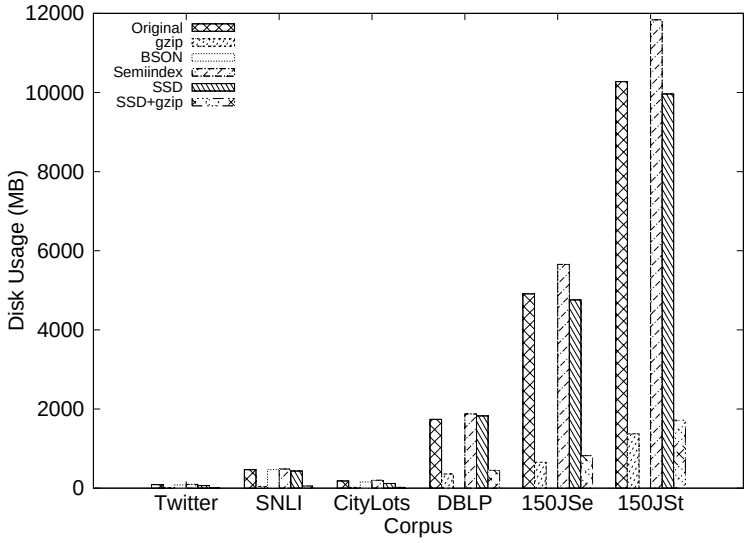


Figure 5.15 Disk usage of the representation compared to the original file size and to *gzip*, for real world corpora.

Corpus	Time (s)
`array`	0.148
`bool`	0.204
`double`	0.255
`int`	0.238
`null`	0.214
`object`	0.211
`string`	0.338

Corpus	Time (s)
`Twitter`	0.168
`SNLI`	1.345
`CityLots`	0.443
`DBLP`	3.168
`150JS-evaluation`	11.17
`150JS-training`	52.76

Table 5.5 Serialization time of the representation.

in Section 5.2.3. More specifically, we performed several experiments based on chunk division, where a single JSON document acting as a corpus is divided into multiple smaller chunks.

In all our datasets, the tree structure is fairly shallow, with small depth, and

hence this simple modification is enough to split the large document into several smaller chunks. This enables efficient RAM usage and large-scale document processing because we do not need to maintain and store all the intermediate representations to make queries work. Nevertheless, this may increase the disk size, given that representations of chunks do not share pre-constructed `names` and `stringValues` arrays. By adding a virtual root node to the entire tree structure, each part of a document is considered as a child tree with its own suite of arrays.

We tested the effect of chunk division using both synthetic and real-world corpora. For synthetic dataset ($n = 10,000,000$), we divided the corpora into 10 chunks. For real-world corpora, each corpus is divided into 25, 50 and 100 chunks, depending on its disk size.

Corpus	# Chunks	Time (s)	Ratio
`double`	10	5.38	1.08
`string`	10	5.29	1.11
`Twitter`	25	2.18	1.15
`SNLI`	25	7.78	1.13
`DBLP`	25	43.4	1.25
`DBLP`	50	43.8	1.26
`DBLP`	100	45.5	1.31
`150JS-evaluation`	100	397	1.44

Table 5.6 Construction time with chunk division enabled.

Since no intermediate procedure other than the serialization is needed, the construction time is almost identical to that of the original version, as in Table 5.6. From Table 5.7 it is clear that chunk division allows only a portion of RAM is needed to process the whole document. This intermediate representation is flushed to disk, so only a small amount of RAM is required even for a big JSON document. Note that these ratios are not inversely proportional to

Corpus	# Chunks	RAM Usage (MB)	Ratio
double	10	16	0.11
string	10	29	0.18
Twitter	25	9.1	0.06
SNLI	25	29	0.06
DBLP	25	108	0.05
DBLP	50	62	0.03
DBLP	100	35	0.02
150JS-evaluation	100	224	0.02

Table 5.7 Memory usage with chunk division enabled.

Corpus	# Chunks	Disk Usage (MB)	Ratio
double	10	125	1.02
string	10	144	1.23
Twitter	25	74.2	1.14
SNLI	25	496	1.09
DBLP	25	1,869	1.03
DBLP	50	1,877	1.03
DBLP	100	1,892	1.04
150JS-evaluation	100	10,352	1.04

Table 5.8 Disk usage of with chunk division enabled.

the number of chunks, since duplicate values among two individual chunks are not considered identical in the representation.

We have noticed negligible serialization and query time difference from the original representation since only one extra tree operation needs to be done. For queries, we assume that the entire data structure is already loaded into the RAM so that no extra de-serialization is needed during query processing. But as one can imagine, if the representation is not in the RAM, then the chunk

division approach will support the queries significantly faster as it only needs to load a small portion of the data structure into the RAM to answer the query.

5.3.6 String Compression

We also integrated some of the string compression and compressed string dictionary schemes to our library illustrated in Section 5.1.4, so that string values could be compressed efficiently while naive query support is guaranteed. We illustrate the details when the string compression is enabled, by comparing the experimental result to the original representation.

All of those data structures experimented support extracting a dedicated string in compressed form, but they act in different ways. A string could be directly extracted when the compressed dictionary schemes are used, while the remaining data structures support byte-based extract operations. For the latter structures we record the starting position of each string in the bit indexed array, rather than the ID. Note that this does not alter the theoretic time bound of the query operations.

Figure 5.16 denotes construction time when various string compression schemes are applied to some of the corpora. For the `string` corpus, n is equal to $10,000,000$. Following the tendency of the theoretic time bounds suggested, when a document contains a large portion of strings, constructing the relevant compressed data structures takes most of the construction time. Since compressed suffix trees and arrays are merged forms of different succinct data structures, the construction involves more time than the pure wavelet tree construction.

One alternative way to compress the `stringValues` array is to apply general-purpose compression schemes, however, the core penalty of this method is that the compressed form does not support random access without explicit decompression, which is a significant overhead while querying. Fortunately, the double-array based data structures provide decent compression time.

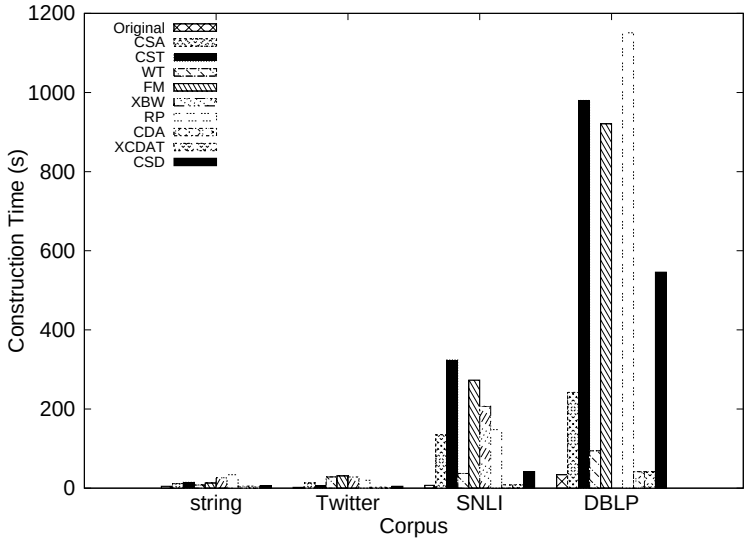


Figure 5.16 Construction time with string compression enabled.

Additional string compression drastically decreases the overall disk size – even competitive to *gzip*ped compression – if the original corpus contains a high portion of strings, summarized in Figure 5.17.

We claim that most semi-structured documents contain a large number of strings so that applying string compression to those guarantees less disk usage. If no string compression is applied, extracting an arbitrary string value from the `stringValues` array does not rely on the length of the value. Nevertheless, as dealt in the previous section, extracting a string from the compressed representation takes linear time proportional to the desired length of the string.

5.3.7 Query Time

In Section 5.2.4 several types of queries are discussed, and our library implements those as either array traversals or tree operation emulations.

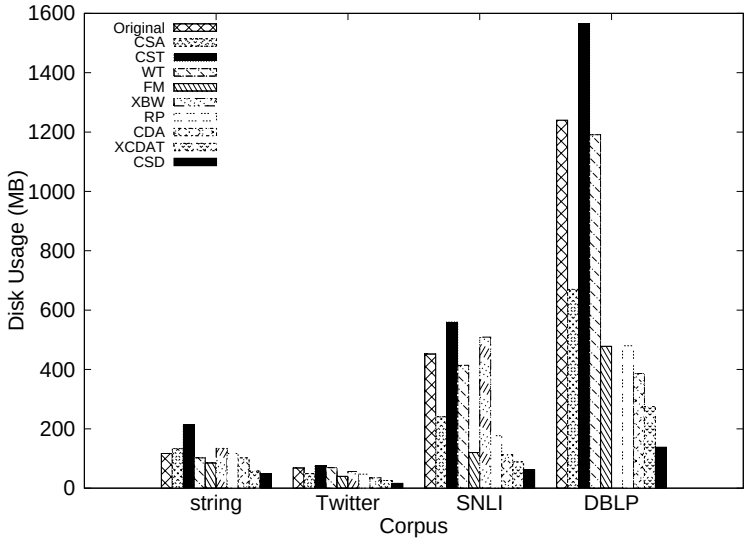


Figure 5.17 Disk usage with string compression enabled.

Existence Queries

Query Type	Time (ms)
`existsName`	0.012
`existsElement`	2,907
`countElements`	503,867

Table 5.9 Query time of existence queries in the `SNLI` corpus, without bitmap indexes.

Table 5.9 shows time values (in milliseconds) took by existence queries performed in the `SNLI` corpus when bitmap indexes are not constructed for the dedicated name, while Table 5.10 indicates the values when relevant SBH-compresseed bitmap indexes exist. Both existing and non-existing elements are queried, and the average values are recorded in both the tables. Since only the `names` array is needed for answering name existence (`existsName`), and the

Query Type	Time (ms)
`existsName`	0.012
`existsElement`	14
`countElements`	38

Table 5.10 Query time of existence queries in the `SNLI` corpus, with compressed bitmap indexes.

number of names in the corpus is small, the query only requires small amount of time. Bitmap indexes do not affect the query time for `existsName`.

Nevertheless, for the remaining two queries, the library without bitmap indexes needs to navigate all the representation from scratch in the worst case to answer those. Therefore, the query time is proportional to the number of elements in the entire document, which is infeasible for this library to process these types of queries. Fortunately, when compressed bitmap indexes exist, the two aforementioned queries can be substituted to loading and decompressing the relevant bitmap index, respectively. If compression is not applied in the entire bitmap indexes, then in the worst case $0.8 \times \textit{cardinality}$ MB of extra storage is needed. When the desired element is sparse in the representation, then size of the compressed bitmap index is decreased to $3 \times \textit{cardinality} = 20$MB for storing information of a single name entry in the entire `SNLI` corpus, which is significantly less than the original document size. As in Table 5.10, we claim that integrating bitmap index compression into the representation has strength while processing the existence queries.

Navigational Queries

Table 5.11 exhibits the navigational query time experimental results. Queries are invoked in various document locations, and their average time is calculated. Queries are invoked in various document locations, and their average time is calculated. For some corpora where arrays do not exist, only the object queries

Corpus	`listObjNames`	`getObjValue`	`cntArrElems`	`getArrValue`
`string`	209	17.2	-	-
`Twitter`	89.3	16.8	9.8	19.8
`SNLI`	106	17.2	10.3	20.7
`CityLots`	135	16.7	9.8	20.6
`DBLP`	131	16.8	10.2	19.7
`150JS-evaluation`	92.6	17.1	9.9	20.9
`150JS-training`	94.4	17.3	10.1	20.8

Table 5.11 Query time of navigational queries. Units are in microseconds.

are run in the experiments.

The time is mostly the same regardless of the location each query handles, because tree-navigational queries take constant time. Additionally, pointing the exact location in the bit indexed array takes constant time as well, by the constant-time implementation `rank` and `select` operations. For the `listObjNames` queries, the actual experimental time is highly affected by the degree of each element accessed.

We have also emulated the relevant queries as the native operations supported by the other libraries. Figure 5.18 shows comparison of query time in the `Twitter` corpus among the four libraries. Since the succinct tree representation are slower in supporting the tree navigational operations compared to the pointer-based representations, our representation mostly gives the worse performance in query time. Nonetheless, the new representation allows querying through a large document which other frameworks fail to process, with almost identical processing time regardless of the size of the document.

Semi-index supports retrieval of values in an arbitrary location when a name is given. This is done by traversing the whole tree with the assistance of the constructed index. Although our library does not explicitly support the whole traversal as of now, it is remarkable that emulation of traversal would guarantee

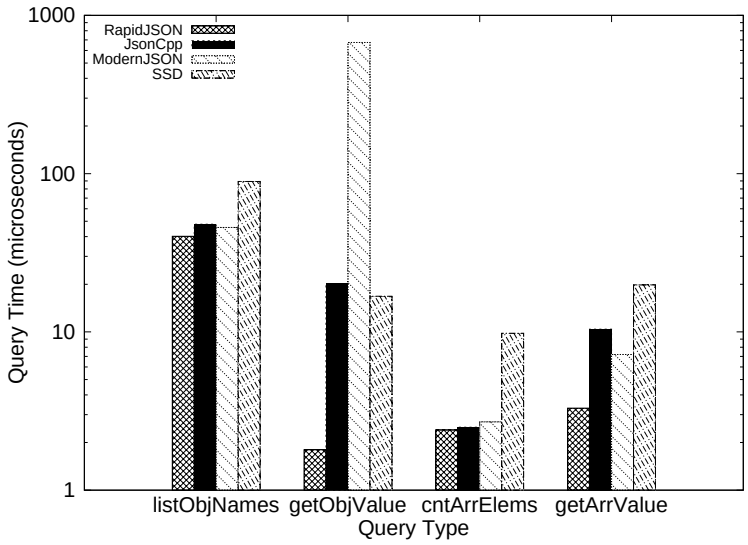


Figure 5.18 Query time of navigational queries compared to different libraries, for the `Twitter` corpus.

similar query time to that of semi-index.

Chapter 6

Conclusion

In this dissertation, we propose an unified space-efficient representation of various semi-structured document formats that mainly exploits compact and succinct data structures. While organizing this representation, we also give subsidiary space-efficient implementations of several compact data structures.

In Chapter 3, we suggest an improved code system for integer arrays. This system is based on the prefix code property, and heavily relies upon the succinct bit vector data structure. Experiments show that this representation is feasible for querying elements even in compressed manner, so that this structure could act as a framework to process big data.

In Chapter 4, we provide a tuned version of the SBH bitmap index compression scheme for efficient processing. While SBH guarantees almost optimal compressed size and decent query processing time by applying simpler byte-based compression algorithm and maintaining a common bucket for decompression, we prove that this algorithm is feasible to be converted in parallel fashion. We show by experiments that this tuning works for both CPUs and GPUs.

Finally, in Chapter 5, we construct a queryable compact representation handling a number of semi-structured document formats by combining two

space-efficient data structures – bit indexed array and succinct ordinal trees. Empirical analyses provide that this representation is both space-efficient and time-efficient, and is also suitable to the system containing less RAM. This representation is suitable for computing environment with constrained memory, by applying partial processing of the document.

We expect that the representation constructed in this dissertation can be utilized in a plethora of data processing frameworks. We leave this as an open research task as well as future work.

Bibliography

[1] BSON (Binary JSON): Specification. http://bsonspec.org/spec.html.

[2] SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview, 2013.

[3] RDF 1.1 Primer. https://www.w3.org/TR/rdf11-primer, 2014.

[4] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[5] W. Andrzejewski and R. Wrembel. GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid. In *International Conference on Database and Expert Systems Applications*, pages 315–329, 2010.

[6] W. Andrzejewski and R. Wrembel. GPU-PLWAH: GPU-based Implementation of the PLWAH Algorithm for Compressing Bitmaps. *Control and Cybernetics*, 40(3):627–650, 2011.

[7] E. Anjos, J. Lee, and S. R. Satti. SJSON: A Succinct Representation for JavaScript Object Notation Documents. In *International Conference on Digital Information Management*, pages 173–178, 2016.

[8] G. Antoshenkov. Byte-aligned Data Compression, 1994. US Patent 5,363,098.

[9] G. Antoshenkov. Byte-aligned Bitmap Compression. In *Data Compression Conference*, page 476, 1995.

[10] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing Queries to Compressed XML Data. In *International Conference on Very Large Data Bases*, pages 1065–1068, 2003.

[11] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct Trees in Practice. In *Meeting on Algorithm Engineering & Expermiments*, pages 84–97, 2010.

[12] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. RDF 1.1 Turtle. *World Wide Web Consortium*, 2014.

[13] D. Beckett and B. McBride. RDF/XML Syntax Specification. *World Wide Web Consortium*, 10(2.3), 2004.

[14] O. Ben-Kiki, C. Evans, and I. Dot. YAML 1.2 specification. https://yaml.org/spec/1.2/spec.html, 2009.

[15] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing Trees of Higher Degree. *Algorithmica*, 43(4):275–292, 2005.

[16] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *International Conference on Very Large Data Bases*, 2011.

[17] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning. A Large Annotated Corpus for Learning Natural Language Inference. *arXiv preprint arXiv:1508.05326*, 2015.

[18] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, 2014.

[19] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. *World Wide Web Consortium*, 1998.

[20] N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and G. Navarro. Improved Compressed String Dictionaries. In *ACM International Conference on Information and Knowledge Management*, pages 29–38, 2019.

[21] N. R. Brisaboa, A. Cerdeira-Pena, A. Farina, and G. Navarro. A Compact RDF Store Using Suffix Arrays. In *International Symposium on String Processing and Information Retrieval*, pages 103–115. Springer, 2015.

[22] N. R. Brisaboa, S. Ladra, and G. Navarro. Directly Addressable Variable-length Codes. In *International Symposium on String Processing and Information Retrieval*, pages 122–130. Springer, 2009.

[23] P. Buneman. Semistructured Data. In *ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[24] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical report, Digital Systems Research Center, 1994.

[25] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update Conscious Bitmap Indices. In *International Conference on Scientific and Statistical Database Management*, page 15, 2007.

[26] T. C. Casas. JSONC-JSON Compressor and Decompressor. https://github.com/tcorral/jsonc, 2015.

[27] S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. A Framework for In-place Graph Algorithms. In *European Symposium on Algorithms*, 2018.

[28] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better Bitmap Performance with Roaring Bitmaps. *Software: Practice and Experience*, 2015.

[29] C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *ACM SIGMOD Record*, volume 27, pages 355–366, 1998.

[30] J. Chang, Z. Chen, W. Zheng, Y. Wen, J. Cao, and W.-L. Huang. PLWAH+: A Bitmap Index Compressing Scheme Based on PLWAH. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 257–258, 2014.

[31] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakmaoui, and G. Peng. A Survey of Bitmap Index Compression Algorithms for Big Data. *Tsinghua Science and Technology*, 20(1):100–115, 2015.

[32] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[33] D. Clark. Compact Pat Trees. *PhD thesis, University of Waterloo*, 1997.

[34] J. Clark, S. DeRose, et al. XML Path Language (XPath) Version 1.0. *World Wide Web Consortium*, 1999.

[35] F. Claude and G. Navarro. Practical Rank/select Queries over Arbitrary Sequences. In *International Symposium on String Processing and Information Retrieval*, pages 176–187. Springer, 2008.

[36] F. Claude, G. Navarro, and A. Ordónez. The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets. *Information Systems*, 47:15–32, 2015.

[37] O. Curé and G. Blin. An Update Strategy for the Waterfowl RDF Data Store. 2014.

[38] O. Curé, G. Blin, D. Revuz, and D. C. Faye. Waterfowl: A Compact, Self-indexed and Inference-enabled Immutable RDF Store. In *European Semantic Web Conference*, pages 302–316, 2014.

[39] I. Davis, T. Steiner, and A. Hors. RDF 1.1 JSON Alternate Serialization (RDF/JSON). *World Wide Web Consortium*, 2013.

[40] F. Deliège and T. B. Pedersen. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *International Conference on Extending Database Technology*, pages 228–239, 2010.

[41] O. Delpratt, S. Joannou, N. Rahman, and R. Raman. The SiXML Project: SiXDOM 1.2. 2013.

[42] E. Demaine and M. Hajiaghayi. BigDND: Big Dynamic Network Data. http://projects.csail.mit.edu/dnd/DBLP/, 2014.

[43] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[44] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. *World Wide Web Consortium*, 2007.

[45] P. Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[46] A. Farzan, R. Raman, and S. S. Rao. Universal Succinct Representations of Trees. In *International Colloquium on Automata, Languages and Programming*, pages 451–462, 2009.

[47] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring Labeled Trees for Optimal Succinctness, and Beyond. In *Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, 2005.

[48] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and Searching XML Data via Two Zips. In *International Conference on World Wide Web*, pages 751–760, 2006.

[49] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

[50] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-text Indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.

[51] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, 2007.

[52] D. Florescu and G. Fourny. JSONiq: The History of a Query Language. *IEEE Internet Computing*, 17(5):86–90, 2013.

[53] A. S. Fraenkel and S. T. Klein. *Robust Universal Complete Codes as Alternatives to Huffman Codes*. 1985.

[54] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. In *VLDB Endowment*, volume 3, pages 1382–1393, 2010.

[55] S. Gog, T. Beller, A. Moffat, and M. Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *International Symposium on Experimental Algorithms*, pages 326–337, 2014.

[56] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, 1966.

[57] Google. Using JSON in the Google Data Protocol. https://developers.google.com/gdata/docs/json, 2014.

[58] R. Grossi, A. Gupta, and J. S. Vitter. High-order Entropy-compressed Text Indexes. In *Annual ACM-SIAM Symposium on Discrete algorithms*, pages 841–850, 2003.

[59] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[60] G. Guzun and G. Canahuate. Performance Evaluation of Word-aligned Compression Methods for Bitmap Indices. *Knowledge and Information Systems*, pages 1–28, 2015.

[61] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A Tunable Compression Framework for Bitmap Indices. In *International Conference on Data Engineering*, pages 484–495, 2014.

[62] G. Jacobson. Space-efficient Static Trees and Graphs. In *Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[63] G. J. Jacobson. Succinct Static Data Structures. *PhD thesis, Carnegie Mellon University*, 1988.

[64] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct Representation of Ordered Trees with Applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012.

[65] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *International Conference on Very Large Data Bases*, pages 278–289, 1999.

[66] F. Kammer and A. Sajenko. Linear-time In-place DFS and BFS on the Word RAM. In *International Conference on Algorithms and Complexity*, pages 286–298, 2019.

[67] S. Kanda, K. Morita, and M. Fuketa. Compressed Double-array Tries for String Dictionaries Supporting Fast Lookup. *Knowledge and Information Systems*, 51(3):1023–1042, 2017.

[68] A. Kapoulkine. pugixml. https://pugixml.org, 2006.

[69] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. RQL: A Functional Query Language for rDF. In *The Functional Approach to Data Management*, pages 435–465. Springer, 2004.

[70] S. Kim, J. Lee, S. R. Satti, and B. Moon. SBH: Super Byte-aligned Hybrid Bitmap Compression. *Information Systems*, 62:155–168, 2016.

[71] J. Lee and S. R. Satti. A Simple Integer Sequence Code System Supporting Random Access. *KIISE Transactions on Computing Practices*, 23(10):594–598, 2017.

[72] K. Lee and B. Moon. Bitmap Indexes for Relational XML Twig Query Processing. In *ACM Conference on Information and Knowledge Management*, pages 465–474, 2009.

[73] G. Leighton, T. Müldner, and J. Diamond. TREECHOP: a Tree-based Query-able Compressor for XML. In *Canadian Workshop on Information Theory*, pages 115–118, 2005.

[74] D. Lemire, O. Kaser, and K. Aouiche. Sorting Improves Word-aligned Bitmap Indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[75] B. Lepilleur. JsonCpp. https://github.com/open-source-parsers/jsoncpp, 2016.

[76] M. Ley. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *International Symposium on String Processing and Information Retrieval*, pages 1–10, 2002.

[77] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *ACM SIGMOD Record*, volume 29, pages 153–164, 2000.

[78] N. Lohmann. JSON for Modern C++, 2016.

[79] F. Maddix. Books. http://www.cems.uwe.ac.uk/ fj-maddix/Books.xml.

[80] V. Mäkinen and G. Navarro. Succinct Suffix Arrays based on Run-length Encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56, 2005.

[81] U. Manber and G. Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[82] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical Compressed String Dictionaries. *Information Systems*, 56:73 – 108, 2016.

[83] P. B. Miltersen. Cell Probe Complexity-a Survey. In *Conference on the Foundations of Software Technology and Theoretical Computer Science, Advances in Data Structures Workshop*, page 2, 1999.

[84] J. I. Munro. Tables. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

[85] J. I. Munro, R. Raman, V. Raman, and S. Rao. Succinct Representations of Permutations and Functions. *Theoretical Computer Science*, 438:74–88, 2012.

[86] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[87] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[88] G. Navarro and K. Sadakane. Fully Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3):16, 2014.

[89] M. Nelson, Z. Sorenson, J. M. Myre, J. Sawin, and D. Chiu. Gpu Acceleration of Range Queries over Large Data Sets. In *IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 11–20, 2019.

[90] D. Okanohara and K. Sadakane. Practical Entropy-compressed Rank/select Dictionary. In *Meeting on Algorithm Engineering & Expermiments*, pages 60–70. Society for Industrial and Applied Mathematics, 2007.

[91] P. E. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 24(3):8–11, 1995.

[92] G. Ottaviano and R. Grossi. Semi-indexing Semi-structured Data in Tiny Space. In *ACM International Conference on Information and Knowledge Management*, pages 1485–1494, 2011.

[93] R. Raman, V. Raman, and S. R. Satti. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007.

[94] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning Programs from Noisy Data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774, 2016.

[95] R. Rice and J. Plaunt. Adaptive Variable-length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communication Technology*, 19(6):889–897, 1971.

[96] T. Rincy and R. Rajesh. Space Efficient Structures for JSON Documents. *International Journal of Computer Engineering and Technology*, 5(12):148–153, 2014.

[97] A. Seaborne. RDQL-a Query Language for RDF. *http://www. w3. org/-Submission/RDQL/*, 2004.

[98] J. Seo, M. Han, K. Park, M. E. Esmaili, R. Entezari-Maleki, A. Movaghar, Y. Wang, H. An, Z. Liu, L. Li, et al. Efficient Accessing and Searching in a Sequence of Numbers. In *Korea-Japan Joint Workshop on Algorithms and Computation*, pages 95–103, 2014.

[99] R. Sinha, S. Mitra, and M. Winslett. Bitmap Indexes for Large Scientific Data Sets: a Case Study. In *International Parallel and Distributed Processing Symposium*, pages 10 pp.–, April 2006.

[100] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström. JSON-LD 1.0. *World Wide Web Consortium*, 16:41, 2014.

[101] M. Stabno and R. Wrembel. RLH: Bitmap Compression Technique based on Run-length and Huffman Encoding. *Information System*, 34(4-5):400–414, 2009.

[102] Tencent. Rapidjson. https://github.com/miloyip/rapidjson, 2015.

[103] B. Tran, B. Schaffner, J. Sawin, J. M. Myre, and D. Chiu. Increasing the Efficiency of GPU Bitmap Index Query Processing. In *International Conference on Database Systems for Advanced Applications*, pages 339–355. Springer, 2020.

[104] Twitter. Twitter Developers Documentation on REST APIs. https://dev.twitter.com/rest/public, 2016.

[105] S. J. van Schaik and O. de Moor. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *ACM SIGMOD International Conference on Management of Data*, pages 913–924, 2011.

[106] S. Vigna. Broadword Implementation of Rank/select Queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168, 2008.

[107] S. Vigna. Quasi-succinct Indices. In *ACM international Conference on Web Search and Data Mining*, pages 83–92, 2013.

[108] J. S. Vitter. External Memory Algorithms. In *European Symposium on Algorithms*, pages 1–25, 1998.

[109] P. Weiner. Linear Pattern Matching Algorithms. In *Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[110] T. A. Welch. A Technique for High-performance Data Compression. *Computer*, 17(6):8–19, 1984.

[111] Y. Wen, Z. Chen, G. Ma, J. Cao, W. Zheng, G. Peng, S. Li, and W.-L. Huang. SECOMPAX: A Bitmap Index Compression Algorithm. In *International Conference on Computer Communication and Networks*, pages 1–7, 2014.

[112] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[113] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on Design and Implementation of Compressed Bit Vectors. Technical report, LBNL/PUB-3161, 2001.

[114] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data Mining with Big Data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, 2014.

[115] Y. Wu, Z. Chen, Y. Wen, J. Cao, W. Zheng, and G. Ma. A General Analytical Model for Spatial and Temporal Performance of Bitmap Index Compression Algorithms in Big Data. In *International Conference on Computer Communication and Networks*, pages 1–10, 2015.

[116] M. Zeiss. City Lots San Francisco. https://github.com/zeMirco/sf-city-lots-json, 2012.

[117] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[118] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-rate Coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

요약

셀 수 없는 빅 데이터가 다양한 원본로부터 생성되고 있다. 이들 데이터의 대부분은 고정되지 않은 종류의 스키마를 포함한 파일 형태로 저장되는데, 이로 인하여 반구조화된 문서 형식을 이용하여 파일을 유지하는 것이 적합하다. XML, JSON 및 YAML과 같은 종류의 반구조화된 문서 형식이 데이터에 내재하는 구조를 유지하기 위하여 제안되었다. 수집된 데이터를 구조화하는 RDF와 같은 여러 데이터 모델들은 사후 처리를 위한 저장 및 전송을 위하여 반구조화된 문서 형식에 의존한다.

반구조화된 문서 형식은 가독성과 다변성에 집중하기 때문에, 문서를 구조화하고 유지하기 위하여 추가적인 공간을 필요로 한다. 문서를 압축시키기 위하여 일반적인 압축 기법들이 널리 사용되고 있으나, 이들 기법들을 적용하게 되면 문서의 내부 구조의 손실로 인하여 데이터의 사후 처리가 어렵게 된다.

데이터를 정보이론적 하한에 가까운 공간만을 사용하여 저장을 가능하게 하면서 질의에 대한 응답을 제공하는 간결한 자료구조는 이론적으로 널리 연구되고 있는 분야이다. 비트열과 트리가 널리 알려진 간결한 자료구조들이다. 그러나 반구조화된 문서들을 저장하는 데 간결한 자료구조의 아이디어를 적용한 연구는 거의 진행되지 않았다.

본 학위논문을 통해 우리는 다양한 종류의 반구조화된 문서 형식을 통일되게 표현하는 공간 효율적 표현법을 제시한다. 이 기법의 주요한 기능은 간결한 자료구조가 강점으로 가지는 특성에 기반한 간결성과 질의 가능성이다. 비트열로 인덱싱된 배열, 간결한 순서 있는 트리 및 다양한 압축 기법을 통합하여 해당 표현법을 고안하였다. 이 기법은 실재적으로 구현되었고, 실험을 통하여 이 기법을 적용한 반구조화된 문서들은 최대 60% 적은 디스크 공간과 90% 적은 메모리 공간을 통해 표현될 수 있다는 것을 보인다. 더불어 본 학위논문에서 반구조화된 문서들은 분할적으로 표현이 가능함을 보이고, 이를 통하여 제한된 환경에서도 빅 데이터를 표현한 문서들을 처리할 수 있다는 것을 보인다.

앞서 언급한 공간 효율적 반구조화된 문서 표현법을 구축함과 동시에, 본 학위논문에서 이미 존재하는 압축 기법 중 일부를 추가적으로 개선한다. 첫째로, 본 학위논문에서는 정렬 여부에 관계없는 정수 배열을 부호화하는 아이디어를 제시한다. 이 기법은 이미 존재하는 범용 코드 시스템을 개선한 형태로, 간결한 비트열 자료구조를 이용한다. 제안된 알고리즘은 기존 범용 코드 시스템에 비해 최대 44% 적은 공간을 사용할 뿐만 아니라 15% 적은 부호화 시간을 필요로 하며, 기존 시스템에서 제공하지 않는 부호화된 배열에서의 임의 접근을 지원한다.

또한 본 학위논문에서는 비트맵 인덱스 압축에 사용되는 SBH 알고리즘을 개선시킨다. 해당 기법의 주된 강점은 부호화와 복호화 진행 시 중간 매개인 슈퍼버켓을 사용함으로써 여러 압축된 비트맵 인덱스에 대한 질의 성능을 개선시키는 것이다. 위 압축 알고리즘의 중간 과정에서 진행되는 분할에서 영감을 얻어, 본 학위논문에서 CPU 및 GPU에 적용 가능한 개선된 병렬화 압축 매커니즘을 제시한다. 실험을 통해 CPU 병렬 최적화가 이루어진 알고리즘은 압축된 형태의 변형 없이 4코어 컴퓨터에서 최대 38%의 압축 및 해제 시간을 감소시킨다는 것을 보인다. GPU 병렬 최적화는 기존에 존재하는 GPU 비트맵 압축 기법에 비해 48% 빠른 질의 처리 시간을 필요로 함을 확인한다.

주요어: 반구조화된 문서 형식, 간결한 자료구조, 압축 알고리즘, 공간 효율적 알고리즘, 정수 배열, 비트맵 인덱스, 빅 데이터 처리.
학번: 2013-23134

Acknowledgements

This dissertation could not have been published if there was no help by various people.

First of all, I would like to send my all the best gratuity to my academic advisor, professor Srinivasa Rao Satti. He has encouraged me during my the life in the lab, both as an academic advisor and a gentle companion. He did not hesitate to not only educate me to become a decent researcher, but also listen to my trouble and give directions to solve it. His guidance to the diverse research projects related to data structures and algorithms, as well as external activities definitely boosted my entire career. I do not hold back to recommend him to others for initiating future research.

I also sincerely appreciate professor Kunsoo Park, professor Inbok Lee, professor Joongchae Na, and professor Seungbum Jo for supervising my work. They had given tremendous amount of comments during the publication of this dissertation, and based on their insights the idea of this dissertation as well as my knowledge of the research area were enriched. I show special gratitude to professor Park and Jo for their long-term instructions that positively affected my life in the lab.

Some of my research contribution is reinforced by fruitful advice from professor Bongki Moon, professor Bernhard Egger and professor Meng He, whom I would like to express my gratitude as well. Professor Moon allowed me to take

part in the research tasks that produced subsidiary works in literature. Professor Egger was willing to lend the research equipments for my experiments. Professor He gave several future research directions during my short stay in Canada.

Throughout my life in campus all of my family members – Sangku Lee, Hyunjoo Kim, and Jungheun Lee – supported me by all means. When I fell in chaos in the middle of my school life, their generous care alleviated negative issues so that I could set focus on academics. They were always on my side and acted as troubleshooters. I cordially appreciate their devotion. I also thank all the family relatives, including Sunjoo Lee and Seungha Kim.

If there were no friends I could not finish this entire research. First, I was fortunate to meet some mates from Canada, as a token of the national researcher exchange grant. Among them, I would like to thank Serikzhan Kazi who gave extreme assistance for both life and studies. I also send my gratituity to Changsub Chang, Hyun Lee, Sejin Oh, Pyojin Kim, Taeyoul Kim, Jongcheon Lim, and Younghyun Ryu who either stayed in campus or came to campus to assist me.

I thank all the lab alumni and members – Sangchul Kim, Heejeong Kim, Neha Dwivedi, Edman Anjos, Jeongsoo Shin, Yeonil Yoo, Wonil Jeong, Einass Tahiri, Seungeun Lee, Seyoung Kim, Wooyoung Park, Seungwoo Kim and Mohammadsadegh Najafi – for me to enjoy life in the lab. I also thank Sankardeep Chakraborty for his kind support during his visit in the lab. In addition, I also thank the members in the neighbor labs – Seounggook Sohn, Hanmin Lee, Taehoon Kim, Bogyeong Kim, Chanho Lee, Jisan Song, Heeran Lee, Younghyun Cho, Changyeon Jo, Chanseok Kang, Daeyong Shin, and Youngsu Cho.

It is my pity that I could not write every person's name that gave me a bit of help. As a wrap-up, I hope every single person I have during my academics stay healthy and achieve everything they pursue. Wish everyone all the best.