Efficiently Compressing String Columnar Data Using Frequent Pattern Mining

by

Xiaojian Wang

B.Sc., Simon Fraser University, 2014
B.Eng., Wuhan University, 2011

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Xiaojian Wang 2016
SIMON FRASER UNIVERSITY
Summer 2016

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: **Xiaojian Wang**

Degree: **Master of Science (Computing Science)**

Title: ***Efficiently Compressing String Columnar Data Using Frequent Pattern Mining***

Examining Committee: **Chair:** Dr. Fred Popowich
Professor
School of Computing Science

Dr. Jian Pei
Senior Supervisor
Professor
School of Computing Science

Dr. Robert D. Cameron
Supervisor
Professor
School of Computing Science

Dr. Anoop Sarkar
Internal Examiner
Professor
School of Computing Science

Date Defended: June 20th, 2016

Abstract

In modern column-oriented databases, compression is important for improving I/O throughput and overall database performance. Many string columnar data cannot be compressed by special-purpose algorithms such as run-length encoding or dictionary compression, and the typical choice for them is the LZ77-based compression algorithms such as GZIP [16] or Snappy [13]. These algorithms treat data as a byte block and do not exploit the columnar nature of the data. In this thesis, we develop a compression algorithm using frequent string patterns directly mined from a sample of a string column. The patterns are used as the dictionary phrases for compression. We discuss some interesting properties of frequent patterns in the context of compression, and develop a pruning method to address the cache inefficiencies in indexing the patterns. Experiments show that our compression algorithm outperforms Snappy in compression ratio while retains compression and decompression speed.

Keywords: data compression; column stores; columnar data

To my family.

Acknowledgements

I would like to express my sincere gratitude to my senior supervisor, Dr. Jian Pei, for his encouragement and support throughout my Master's studies. From the teachings of Dr. Pei, I learned to appreciate the beauty in scientific works with simple ideas and thorough justifications. Under his guidance, I developed research skills which are valuable for a lifetime.

My gratitude also goes to my supervisor, Dr. Robert Cameron, for reviewing my work and enlightening me with his valuable insights on high performance computing. I am grateful to Dr. Fred Popowich and Dr. Anoop Sarkar for serving in the examining committee.

I thank my lab mates, Dr. Dong-Wan Choi, Dr. Chen Lin, Xiao Meng, Juhua Hu, Chuancong Gao, Xiangbo Mao, Yu Yang, Zhefeng Wang, Mingtao Lei, Zicun Cong, Zijin Zhao, for their kind help.

I thank my parents forle encouraging and supporting me on my path of academic studies.

Table of Contents

Approval **ii**

Abstract **iii**

Dedication **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Existing Compression Techniques 3

1.3 Major Ideas 4

1.4 Contributions 4

1.5 Thesis Organization 5

2 Related Work **6**

2.1 Compression in Column Stores 6

2.2 Lempel-Ziv Compression 7

2.3 Compression of String Collections 11

2.4 Compression and Frequent Pattern Mining 12

2.5 Trie 13

3 Problem Definition **14**

3.1 Columnar String Pattern Mining 14

3.2 A Semi-static Encoding Format 16

3.3 The Problem of Optimal Compression with Encoding Format $\mathcal{E}$ 19

4 Compressing Columnar Data with Frequent String Patterns **21**

4.1 PrefixSpan for Columnar String Pattern Mining 21
4.2 Solving the Minimum Factorization Problem with Dynamic Programming . 27
4.3 Greedy Matching for Factorization and Phrase Selection 30
4.4 Pruning Trie 35
4.5 Summary 38

5 Experiments **40**
5.1 Datasets 40
5.2 Comparisons to GZIP and Snappy 41
5.3 Comparisons to Local Lookahead and Re-Pair Under Different Parameters . 46

6 Conclusions **56**

Bibliography **58**

List of Tables

Table 1.1 Examples of string columnar data 3

Table 3.1 An example of string patterns 15
Table 3.2 A semi-static encoding format for a compressed column block 18

Table 4.1 An example column for columnar string pattern mining 22

Table 5.1 Examples of synthetic datasets 41
Table 5.2 Examples of real-world datasets 41
Table 5.3 Dataset statistics 41

List of Figures

Figure 1.1 Row-oriented database storage vs. column-oriented database storage 2

Figure 2.1 LZ77 compression on input *abacababcab* using unbounded window . 8
Figure 2.2 LZ78 compression on input *abacababcab* 10

Figure 3.1 An example of string factorization by greedy matching 17

Figure 4.1 A prefix tree containing all symbols in $\mathcal{C}$ and their positions 22
Figure 4.2 Grouping positions of a by length-2 substrings starting with a . . . 23
Figure 4.3 Towards proving Lemma 4.5 34
Figure 4.4 Uncompressed trie containing *abc*, *abd*, and *bcd* and all their substrings 36
Figure 4.5 An example of pruning trie to improve CPU cache performance in compression 37

Figure 5.1 Compression ratio of our methods compared to GZIP and Snappy . 43
Figure 5.2 Compression speed of our methods compared to GZIP and Snappy 43
Figure 5.3 Average number of candidate dictionary phrases from a sample . . 44
Figure 5.4 Decompression speed of our methods compared to GZIP and Snappy 44
Figure 5.5 Memory usage compared to GZIP and Snappy 45
Figure 5.6 Comparison of compression ratio for *fine_foods* with $minSupport = 5$ 48
Figure 5.7 Comparison of compression speed for *fine_foods* with $minSupport = 5$ 48
Figure 5.8 Comparison of decompression speed for *fine_foods* with $minSupport = 5$ 49
Figure 5.9 Comparison of number of trie nodes in compression, with $minSupport = 5$ 49
Figure 5.10 Comparison of percentages of CPU cycles spent on memory access (L3 cache misses) in compression, with $minSupport = 5$ 50
Figure 5.11 Comparison of compression ratio for *fine_foods* with sampling rate 0.5% 50
Figure 5.12 Comparison of compression speed for *fine_foods* with sampling rate 0.5% 51
Figure 5.13 Comparison of decompression speed for *fine_foods* with sampling rate 0.5% 51

Figure 5.14 Comparison of number of trie nodes in compression, with sampling rate 0.5% 52
Figure 5.15 Comparison of percentages of CPU cycles spent on memory access (L3 cache misses) in compression, with sampling rate 0.5% 52
Figure 5.16 Comparison of memory usage with $minSupport = 5$ 55

Chapter 1

Introduction

In this chapter, we discuss why compression is important in modern database systems, and why we design yet another compression algorithm for column stores. Then we summarize the major contributions and describe the structure of the thesis.

1.1 Background and Motivation

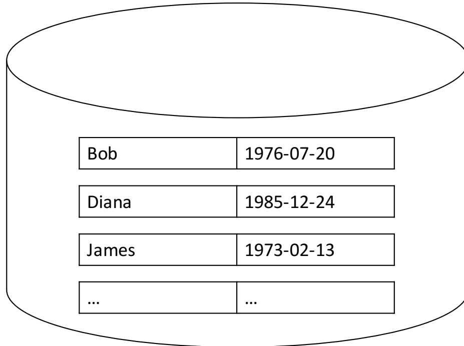
We are witnessing the advent of big-data era. Huge volume of data is being pushed into distributed data stores on a daily basis. Fast data retrieval and processing are becoming more and more important for analytical applications that support business intelligence. Many challenges arise in designing the system infrastructures for efficient storage, retrieval and processing of large volume of data. These challenges are characterized by a group of researchers from Facebook as: fast data loading, fast query processing, efficient storage space utilization, and strong adaptivity to dynamic work loads [21]. Data compression plays an important role in addressing these challenges. With compression, modern database systems enjoy better query processing performance because of faster disk and memory I/O, even after paying the cost of decompression [2, 52]. This is largely because disk and memory bandwidth become a bottleneck for database systems after the rapid advancement in CPU performance.

Traditionally, databases store values in a table row consecutively. Recently, column-oriented data stores are becoming popular, where an entire column (or a block of column) is stored consecutively [1]. Figure 1.1 illustrates the difference between the storage formats of row stores and column stores. Column stores perform better in analytical applications which aggregate over a small fraction of table columns, since they avoid the cost of reading and discarding columns irrelevant to the query. Another advantage of column stores is better compression. Fields in a single column are more likely to be similar to each other, and can be compressed more effectively when they are stored consecutively. Compression has been shown to significantly boost performance in column stores [2, 52]. If the compression scheme

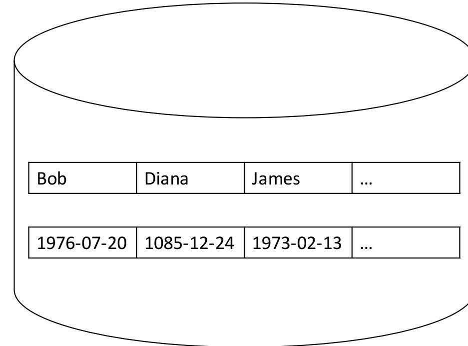
allows directly applying operators on compressed data, the performance improvement can be an order of magnitude [2].

Name	Birthdate
Bob	1976-07-20
Diana	1985-12-24
James	1973-02-13
...	...

(a) A sample database



(b) Row-oriented database storage



(c) Column-oriented database storage

Figure 1.1: Row-oriented database storage vs. column-oriented database storage

Many compression techniques have been investigated and successfully used in column stores to improve data processing performance. Most notably among them are dictionary compression, run length encoding (RLE), bit vector encoding, frame of reference (FOR), and compression techniques in the LZ77 family [2, 21, 19, 52]. However, the first four techniques can only compress data with a certain type or distribution. LZ77 algorithms, on the other hand, accepts any block of byte sequence, but they do so without the awareness that data comes from a columnar format. In this thesis, we aim to answer this question: can we design a better compression algorithm, knowing that the input data is columnar? Such an algorithm can still be considered general purpose, since we can slice any block of data into columnar format. When these pieces are indeed similar to each other, the algorithm further aims to achieve lower compression ratio while being as fast as industry solutions such as Snappy [13] from Google. The major idea is to use frequent pattern mining on columnar data, and use the frequent patterns as a dictionary for compression. Our experiments show that such a simple idea can lead to faster and/or better compression in a variety of columnar inputs compared to Snappy.

1.2 Existing Compression Techniques

We further explain why there is room for a new compression algorithm, in the context of column stores.

Some existing algorithms demand the data to have a certain distribution or format. RLE, for example, compresses data only when there are consecutive fields in a column with the same value. As another example, dictionary compression can only compress data that has limited domain values (By dictionary compression we refer to the algorithm that assigns a single integer code for a column field). If there is a *color* column with only 3 possible values *blue*, *red*, *green*, then we only need 2 bits to represent each field, and effectively compress the column. However, a great variety of data columns don't fit into these assumptions. Let us look at the examples shown in Table 1.1. These string columns cannot be directly compressed by RLE or dictionary compression, although the data fields are structurally similar and exposes compressible common patterns. We are left with no other choice but the general purpose, byte oriented algorithms, which are usually LZ77-based.

Address
7397 9th Street New Bern, NC 28560
318 Holly Drive Ashburn, VA 20147
4919 Summer Street Midlothian, VA 23112
...
017 Hilll Street South Baby, GA 40737

(a) A column of addresses

User Agent
Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10_5_1; rv:1.9.2.20) Gecko/2013-12-30 21:42:29 Firefox/4.0
Opera/8.53.(Windows NT 6.1; sl-SI) Presto/2.9.176 Version/11.00
...
Mozilla/5.0 (Windows 98; en-US; rv:1.9.2.20) Gecko/2013-06-21 00:58:37 Firefox/3.6.7

(b) A column of user agent strings

Table 1.1: Examples of string columnar data

The idea of LZ77 is to replace repeating patterns with reference pointers to previous occurrences in the data [50]. A notable algorithm in this family, DEFLATE used in GZIP [16], has good compression ratio, but is quite slow. In the context of distributed data stores, fast decompression and low CPU overhead usually outweighs compression ratio for better overall performance, which is why faster LZ77-based algorithms such as Snappy and LZ4 are often used. they are exhaustively engineered to decompress at a very fast speed, while sacrificing the compression ratio to some extend. They perform well when RLE or dictionary compression are not applicable, but they are not designed to accept columnar data directly, but rather a sequence of bytes. Columnar data needs to be stitched together into a byte sequence to feed into these algorithms. Can we, however, compress columnar data directly?

1.3 Major Ideas

These observations lead us to the idea of applying frequent pattern mining to columnar data. The output of pattern mining is conveniently useful for compression: a dictionary of repeating patterns. The dictionary is mined from only a sample of the data, since it is not practical, and usually not necessary to mine the entire dataset. With the given dictionary, the running time of compression is dominated by scanning input data to replace patterns with integer codes. To speed up this process we use a variant of trie (prefix tree) for pattern search, to minimize per symbol operations and improve CPU cache performance. Experiments show that this algorithm is superior to Snappy for a variety of columnar inputs (but not all possible inputs, otherwise we would have discovered a better byte-oriented compression algorithm since we can convert any byte sequence into a column). The typical results show that this algorithm has similar compression and decompression speed to Snappy, and achieves better compression ratio for a variety of columnar data, an improvement up to 40%. We have made the code publicly available [1].

However, it is not our goal to design a stable production-ready compression algorithm. It is our goal to show that by understanding the data better, we can compress the data better. On the foundation of this work, it is worthwhile for software engineers to deviate from optimizing LZ77-based algorithms, and switch to a pattern mining based approach to compress columnar data. They can further fine tune the search trie (or other search algorithms), prune the patterns for cache efficiency and invest other engineering effort to develop a production-ready compression software.

1.4 Contributions

We make the following contributions:

1. We develop a compression algorithm that achieves better compression ratio on a variety of columnar data, while maintaining similar compression and decompression speed compared to existing industry solutions.

2. We discuss algorithmic and data-structure choices for efficiently parsing and matching data with a set of frequent patterns.

3. We demonstrate that a pattern mining based approach can effectively compete with traditional LZ77-based algorithms to compress columnar data. We show that it is a promising direction for data compression in column stores.

[1] https://github.com/superxiao/FrequentPatternCompressor

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we review the related work on data compression methods, in the context of compression in column stores. We also review existing work on compressing collections of strings, and the trie data-structure used in some compression methods. In Chapter 3, we formally define pattern mining based compression on columnar data as an optimization problem. In Chapter 4, we present and discuss our sampling-based method and related data-structures for pattern match. Chapter 5 presents the experiment results to show the effectiveness of this method, and Chapter 6 concludes the thesis.

Chapter 2

Related Work

This chapter introduces related work. First, we review the compression methods currently used in column stores. Second, we introduce Lempel-Ziv algorithms, which replace phrases in the input with references to dictionary phrases in a dynamically constructed dictionary. Third, we review compression methods for string collections. Last, we review variations of the trie data-structure. A trie is used to index the dictionary in our method.

2.1 Compression in Column Stores

The foundations of the work on column stores were laid by several influential academic implementations, including MonetDB [26], VectorWise [10], and C-Store [41]. Major database vendors such as Microsoft, IBM and SAP also started to supply their implementations inside commercial database systems [30, 7, 17]. Facebook introduced RCFile [21], a hybrid of column-oriented and row-oriented storage format, into Hadoop systems. Compression is an indispensable part of all these column stores, because column-oriented storage provides opportunities for better compression, which improves I/O performance throughout the memory hierarchy [1]. A variety of compression techniques traditionally used in row stores have been ported into column stores. Compression ratio is improved, and different columns can be compressed by different methods that are most suitable [2, 52, 1]. In addition, it is possible to apply operators directly on compressed data in column stores, tremendously speeding up scan and aggregation operations [2, 3, 23, 1]. Column-store compression also enables other optimization opportunities such as super-scalar compression [52], and lazy decompression, where a column is decompressed only when absolutely necessary [21]. Commonly used compression schemes for column stores have been described in previous works in the context of column stores [2, 52, 1], and we briefly summarize them here.

Null suppression [2] is a compression scheme for databases where consecutive zeros or blanks are replaced by a short description of how to reconstruct them. For example, an integer less than 127 can be represented using one byte instead of four. To signal that this

is the last byte for the value, the most significant bit is set to 0. This scheme is usually called *variable byte encoding* [49], which is one form of null suppression. *Run length encoding (RLE)* [2] replaces consecutive appearances of any column value by a short description. This scheme is effective when the data contains runs of the same values, which are commonly seen in sorted columns, or columns with a small domain. Aggregation operators such as COUNT, SUM or AVG can be easily applied to RLE encoded data without decompression. *Bit vector encoding* [2] uses a bit vector for each domain value to indicate where in the column this value appears. For example, for a column [*red, green, red, yellow, yellow*], three bit vectors are created for the three possible colors: *red* 10100, *green* 01000, and *yellow* 00011. This scheme is useful with small domain (therefore small number of vectors). *Dictionary compression* [2] replaces each column field by a fixed-length integer code. An example was given in Chapter 1. Some variations of this scheme preserves order, which means a smaller value is assigned a smaller integer code [5, 9], to facilitate direct searching and sorting operations on compressed data. *Frame of reference (FOR)* [1] is a compression scheme where each field of a column is described as a difference between the original value and a common reference. For a column of integers in a certain range, the common reference can be the minimum integer in this column. The differences can be represented more compactly using bit-packing. A related scheme is *delta encoding* [1], where a field is replace by the difference to the previous field in the column. Delta encoding is particularly effective on sorted data. A recent work develops a vectorized variation of FOR [33], which uses SIMD instructions whenever possible, and is significantly faster than traditional implementations. This scheme is used in our approach to efficiently compress and decompress integer codes.

These commonly used compression schemes all make certain assumptions of the data in order to provide compression benefit. Our approach can be used on columns not fitting these assumptions, particularly string columns with no fixed domain. Currently byte-oriented LZ77 algorithms are commonly used in this scenario. We describe LZ77 in the next section.

2.2 Lempel-Ziv Compression

Lempel and Ziv proposed two early influential works on data compression in the 70's, LZ77 [50] and LZ78 [51]. LZ77 is the basis of DEFLATE [15], which is used in the popular GZIP compression program [16]. It also forms the basis for most fast general-purpose compression tools currently used in distributed data storage systems, such as Snappy and LZ4. LZ78 is less popular but has practical applications as well. Our method is closely related to LZ78 as they both replace phrases by integer codes representing dictionary entries. We give an overview of the two Lempel-Ziv algorithms below.

LZ77 replaces phrases in the input by descriptors pointing to the patterns' previous occurrences. Such a descriptor is called an *LZ factor*, and usually consists of the length of the phrase to be replaced, and an offset value indicating the distance to the previous

occurrence. In the original LZ77 proposal, a factor contains another element which is the literal symbol immediately following the replaced phrase. This is useful for skipping symbols when no match is found. A later LZ77 variation, *LZSS*, eliminates the literal symbol in a factor, and uses bit flags to differentiate factors and literals [42]. This strategy is followed by most practical LZ77 implementations. The search of a matching phrase is usually constrained in a sliding window immediately preceding the input position. In GZIP, the sliding window is usually 32KB or less. The example in Figure 2.1 demonstrates how LZ77 processes the input string *abacababcab* into factors, using an unbounded window.

Step	Input	Output
1	$\|abacababcab$	$(0, 0, a)$
2	$a\|bacababcab$	$(0, 0, a)(0, 0, b)$
3	$\underline{a}b\|\underline{a}cababcab$	$(0, 0, a)(0, 0, b)(2, 1, c)$
4	$\underline{aba}c\|\underline{aba}bcab$	$(0, 0, a)(0, 0, b)(2, 1, c)(4, 3, b)$
5	$aba\underline{cab}ab\|\underline{cab}$	$(0, 0, a)(0, 0, b)(2, 1, c)(4, 3, b)(5, 3, \$)$

Figure 2.1: LZ77 compression on input *abacababcab* using unbounded window. The vertical bar marks the current input position in each step. Underlines mark the phrases to be replaced, and their previous occurrences. The output is a list of LZ factors, each consisting of the offset, the length, and the next symbol following the phrase to be replaced.

Step 1. The window before the current input position is initially empty. A triple $(0, 0, a)$ is sent to the output stream, where the offset 0 and length 0 indicate that no matching phrase is found in the window. The first literal symbol a is also sent to the output as a part of the triple, to advance the input position to the next symbol.

Step 2. The window is now a. The current input symbol b is not found in the window, so $(0, 0, b)$ is sent to the output. The input position is advanced by 1 symbol.

Step 3. The window is now ab. Since the current input symbol a is found in the window, a triple $(2, 1, c)$ is sent to the output. The triple indicates that 2 symbols to the left there is a match of length 1, which is a. The symbol after the match in the current input buffer is c, which is also sent to the output. The input position is advanced by 2 symbols.

Step 4. The window is now $abac$. A match aba of length 3 is found with an offset of 4, so the triple $(4, 3, b)$ is sent to the output, where b is the symbol after the match in the current input buffer. The input position is advanced by 4 symbols.

Step 5. The window is now $abacabab$. A match cab is found, and there is no next symbol in the input buffer after the match, so the triple $(5, 3, \$)$ is sent to the output, where \$ indicates the end of the input stream.

Practical LZ77 implementations use additional strategies to achieve compression. First, factors can be further compressed. GZIP uses Huffman coding to compress factors, while Snappy uses variable byte encoding. Second, short matches are usually skipped. GZIP uses a minimum match length of 3 bytes, and Snappy uses 4. Third, different algorithms are used for finding a match. GZIP uses a chained hash-table to index all 3-byte sequences in the sliding window, and search through a bucket for a long match. It also uses *local lookahead*, examining the cases of looking for a match starting from the next input byte, or skipping it in favor of a longer match. Snappy avoids conflict resolution altogether by using the most recent phrase prefixed by the next 4 input bytes. Snappy avoids local lookahead as well. These differences make Snappy a much faster algorithm, at a cost of compression ratio.

Compression can be generally considered from two different aspects, *modelling* and *coding* [38]. Modelling is the computation of redundancy characteristics of the input, producing a dictionary or other forms of information model that captures the repetitiveness of the input. Coding is using information from the model to replace symbols from the input. In the example of LZ77, the sliding window can be considered as a dictionary that is constantly updated during compression and decompression, and is part of the compression model. Since a LZ77 model is not separately stored in compressed data, but rather constructed and updated *on-line* during compression and decompression, it is said to be an *adaptive model*.

LZ78 builds up a dictionary which maps phrases to integer codes (dictionary indexes) as the input is parsed and compressed. At each step, it tries to find the longest dictionary phrase that matches the input. It then replaces the input phrase by its dictionary index. Meanwhile, it inserts a new dictionary phrase, which is the phrase being replaced, appended by the following symbol in the input. Dictionary grows during compression, as well as the bit length of integer indexes. Some implementations avoid degrading compression ratio by resetting the dictionary when it grows to a certain size. An LZ78 factor consists of the dictionary index of a phrase, and the next symbol following the phrase. *LZW* [47] removes this symbol from factors. LZ78 is also an adaptive modeling algorithm since the dictionary is not explicitly stored, and can be dynamically constructed in decompression, similar to how it is constructed in compression. During compression, a trie (prefix tree) is often used for the dictionary for fast match finding. Figure 2.2 shows an example of LZ78 compression with the input string *abacababcab*.

Step 1. The dictionary is initially empty. A pair $(0, a)$ is sent to the output where 0 indicates that a match is not found in the dictionary. The literal symbol a in the pair is the current input symbol. A new dictionary phrase a is discovered , and inserted into the dictionary with a code (index) 1. The input position is advanced by 1 symbol.

Step	Input	New Dictionary Entry	Output
1	$\vert abacababcab$	$a \rightarrow 1$	$(0, a)$
2	$a \vert bacababcab$	$b \rightarrow 2$	$(0, a)(0, b)$
3	$ab \vert \underline{a}cababcab$	$ac \rightarrow 3$	$(0, a)(0, b)(1, c)$
4	$abac \vert \underline{a}babcab$	$ab \rightarrow 4$	$(0, a)(0, b)(1, c)(1, b)$
5	$abacab \vert \underline{ab}cab$	$abc \rightarrow 5$	$(0, a)(0, b)(1, c)(1, b)(4, c)$
6	$abacababc \vert \underline{ab}$		$(0, a)(0, b)(1, c)(1, b)(4, c)(4, \$)$

Figure 2.2: LZ78 compression on input *abacababcab*. Vertical bar marks the current input position in each step. Underline marks the phrase to be replaced by a dictionary index. The output is a list of LZ factors, each consisting of an dictionary index, and the next symbol following the phrase.

Step 2. The current input symbol is b, and a match is not found in the dictionary. A pair $(0, b)$ is sent to the output, and b is inserted into the dictionary with a code 2. The input position is advanced by 1 symbol.

Step 3. The longest match found in the dictionary is a, a single symbol. The pair $(1, c)$ is sent to the output where 1 is the code of a, and c is the symbol following the match in the input buffer. A new phrase ac is discovered, and inserted into the dictionary with a code 3. The input position is advanced by 2 symbols.

Step 4. Similar to step 3, the pair $(1, b)$ is sent to the output and ab is inserted into the dictionary. The input position is advanced by 2 symbols.

Step 5. The longest match found in the dictionary is ab, so the pair $(4, c)$ is sent to the output where 4 is the code of ab. A new phrase abc is inserted into the dictionary. The input position is advanced by 3 symbols.

Step 6. The longest match found in the dictionary is ab, so the pair $(4, \$)$ is sent to the output, where $ indicates the end of the input stream.

Our approach is similar to LZ78 in building an explicit dictionary mapping phrases to integer codes in compression. However, we use a two-pass process, where a dictionary is first generated from the data, and then used for compression without dynamic update. To allow decompression, the dictionary must be stored explicitly in the compressed data. Such a two-pass process is referred to as *semi-static modelling* [8], while LZ77 and LZ78 are dynamic. A third kind is *static modelling* [8], which uses a pre-determined model.

Our method is faster in compression and decompression than traditional LZ78 implementations, and is generally as fast as Snappy-like LZ77 methods. First, we avoid the running time cost of dynamically updating the dictionary. Second, we prunes dictionary

phrases to construct a cache-efficient variation of trie. In addition, we compress and decompress integer codes using existing work that exploits super-scalar capabilities of modern processors [33], reducing the cost of bit manipulation. Therefore, the running time of decompression is dominated by copying phrases from the dictionary, similar to LZ77. This makes our method as fast as Snappy in decompression.

2.3 Compression of String Collections

The problem of compressing a collection of strings has been studied in the areas of information retrieval for web collections [12, 24] and genome sequences [45, 37, 28, 14]. It is helpful to first introduce some previous works on the problem of selecting dictionary phrases for semi-static compression, which we refer to as *phrase selection*.

Phrase selection has been approached by several *grammar-based* methods, namely RE-PAIR [31], RAY [11], and SEQUITUR [36]. Their procedures are akin to generating a set of context-free grammar rules. RE-PAIR, for example, selects the most frequent symbol-pair that appears in the message, and then creates a new symbol in the alphabet to replace all appearances of this pair. It then updates the frequencies of symbol pairs. This procedure is executed iteratively until every distinct pair appears only once. RAY selects symbol pairs in a more complex manner, following the same intuition of repeatedly selecting the most heuristically compressible pair. These methods offer significantly better compression ratio than GZIP but incur great time and memory cost in compression. It is difficult to use them directly on large data sets.

XRAY [12] is an extension of RAY which compresses large string collections using a phrase book generated from a sample collection. The use of sampling scales the method to large data sets, and still provides better compression ratio than GZIP with comparable speed. Both XRAY and our method use sampling, and can be viewed as semi-static variations of LZ78. However there are two major differences. First, we approach phrase selection using frequent sequential patterns generated by the well-known algorithm *PrefixSpan* [27], instead of the greedy grammar-based approach of XRAY. Second, XRAY uses local lookahead similar to GZIP. Our method avoids local lookahead, and is much faster in compression. The authors of XRAY noted that compression is ineffective without local lookahead, but our method shows that a method without local lookahead still can achieve reasonably good compression. A possible explanation is related to the anti-monotonicity property of frequent patterns, which we will discuss in detail in the next 2 chapters.

Another related work is *relative Lempel-Ziv compression* [24], which builds a dictionary from a sample of the collection, and compresses each string by referencing the dictionary using LZ77-style factors. It achieves an impressive compression ratio and decompression speed, and supports random access of individual strings. However, the method targets information retrieval system storing large document collections (hundreds of Gigabytes)

with a dictionary of at least 500 MB large. The dictionary needs to be held in main memory during decompression. It is not clear yet how this method can be scaled down to compress small blocks of data with a low memory cost.

Compression of genome sequences is a unique problem in that two randomly selected human genomes are highly correlative. The state-of-the-art methods belong to a class known as *referential compression*. A single sequence is selected as a reference and all other sequences are compressed relative to the reference sequence [45, 37, 28, 14]. These methods can achieve a compression ratio of 400:1 or even more, as apposed to 4:1 to 8:1 using traditional compression methods [44]. For document collections and general string columnar data, a single string is not enough to capture the global repetitiveness of the collection. However, the intuition behind compressing genome sequences using a single reference sequence is similar to compressing other types of string collections using a sample collection.

A recurring theme in this line of works is the effective use of sampling. Sampling-based methods achieve favorable performance in different trade-off settings of compression speed, decompression speed, compression ratio and memory usage. Our method selects phrases from a sample, and targets fast compression and decompression of columnar data with relatively short fields, such as addresses and log messages. We approach this problem from a different way than the relative Lempel-Ziv compression method mentioned earlier. Specifically, our method is more similar to LZ78, constructing an explicit dictionary instead of using LZ77-style positional references. We note that the latter approach is a possible direction, and worth future investigations.

2.4 Compression and Frequent Pattern Mining

A few works link data compression to frequent pattern mining, by using the *minimum description length* (MDL) principle to select a set of sequential patterns that compresses data well [43, 29]. Compression is used as a means to reduce a large set of frequent patterns to a smaller set of interesting patterns with less redundancy. Our work focuses on using frequent patterns to efficiently compress textual data, and related work on this direction is rare. It should be noted that a smaller set of phrases are desirable as long as the compression ratio is preserved. This is because a smaller dictionary causes less cache misses, and speeds up compression. In this work, we use a simple pruning strategy to reduce the number of phrases. With a small loss in compression ratio, the pruning is shown to significantly speed up compression when there are a large number of frequent patterns.

2.5 Trie

A trie (prefix tree) is a multi-way tree for indexing strings. A basic implementation uses a pointer array to index children nodes. For example, to index byte sequences using one node per byte, each internal trie node represents a prefix string, and contains an array of 256 pointers to children nodes. Each child represents a string prefixed by the parent's string, with one appended byte.

Trie is often used in LZ78-style compression algorithms, since it is fast for looking up the longest dictionary phrase matching the input. However, the basic form of trie may leave many array entries empty, and is not space-efficient. With a large dictionary, the performance of frequent search operations may suffer from frequent CPU cache misses. Variations of trie have been proposed to address this issue. *Path compression* collapses consecutive single descendant nodes into one node, which stores the string segment represented by the replaced nodes [35]. This compressed version of trie is sometimes called a *compact trie*. *Burst-trie* uses alternative container (linked-list by default) instead of array when the number of children is limited [22]. *HAT-Trie* uses cache-conscious hash-table to replace sparse array [6]. *Adaptive radix tree* uses path compression, and dynamically adjust array size for trie nodes [32]. XRAY compression algorithm [12] uses ternary search tree, a trie variation with a fan-out of 3.

During our prototype stage, we found that compact trie and adaptive radix tree only speed up compression in a few limited cases where dictionary is large and cache inefficiency is severe. In other cases, they significantly slow down the compression because of the additional operations they add for traversing phrases. For example, compact trie needs a conditional test for each node examining whether it contains compressed path. This seems trivial but adds up in practice, and the cost is not sufficiently compensated by improvements in cache locality. Instead of using existing trie variations, we propose an optimization specifically designed to speed up compression, which is closely tied to the phrase pruning strategy mentioned in the previous section.

Chapter 3

Problem Definition

In this chapter, we introduce some preliminaries on frequent pattern mining for string columns. We then introduce a semi-static encoding format for string columns. Lastly, we formally define the problem of optimal compression with the aforementioned encoding format.

3.1 Columnar String Pattern Mining

Let $\Sigma = \{c_1, c_2, \ldots, c_m\}$ be an *alphabet*, a non-empty finite set of character symbols. A *string* s over Σ is an empty sequence, or a finite sequence of symbols $c'_1 c'_2 \cdots c'_l$, where $l > 0$ and $c'_i \in \Sigma$ for $1 \leq i \leq l$. l is called the *length* of the string, denoted by $|s| = l$. For example, a string of ASCII characters *abacababcab* has an alphabet Σ consisting of 128 ASCII characters, i.e. $|\Sigma| = 128$. In this thesis, we are mainly concerned with byte strings, where Σ consists of 256 byte values, i.e. $|\Sigma| = 256$. We denote by $s[i]$ the i-th symbol of s.

A non-empty string $s = c_1 c_2 \cdots c_l$ is a *substring* of a non-empty string $s' = c'_1 c'_2 \cdots c'_{l'}$ at r, denoted by $s \preceq_r s'$, if $l \leq l'$, $1 \leq r \leq l' - l + 1$ and $c_{j+1} = c'_{j+r}$ for all $0 \leq j \leq l - 1$. We also use $s'[r, r+l-1]$ to denote the substring $s \preceq_r s'$, where $l = |s|$. We call s' a *superstring* of s, and s' *supports* s. For example, $s = cab$ is a substring of $s' = abacababcab$, where we have $s \preceq_4 s'$ and $s \preceq_9 s'$ (the two appearances of substring s in s' start from the 4th and 9th symbols, respectively). For $s \preceq_r s'$, we say that s is a *prefix* of s' if $r = 1$, and s is a *postfix* or *suffix* of s' if $r = |s'| - |s| + 1$.

A *string column* $\mathcal{C}$ over Σ is a sequence of strings over Σ. $\mathcal{C}$ is defined as a sequence instead of a set, since the same string may appear several times in a column, and we want to preserve the order of a column. Note that the strings in a column may have different lengths. The number of strings in $\mathcal{C}$ is denoted by $|\mathcal{C}|$, and the i-th string in $\mathcal{C}$ is denoted by $\mathcal{C}[i]$ where $1 \leq i \leq |\mathcal{C}|$.

Definition 3.1 (Support and Relative Support)**.** The **support** of a string s in a string s', denoted by $Support(s, s')$, is the number of appearances of s in s', that is,

$$Support(s, s') = |\{r | s \preceq_r s'\}| \tag{3.1}$$

The **support** of a string s in a column $\mathcal{C}$, denoted by $Support(s, \mathcal{C})$ is the number of appearances of s in $\mathcal{C}$, that is,

$$Support(s, \mathcal{C}) = \sum_{1 \leq i \leq |\mathcal{C}|} Support(s, \mathcal{C}[i]) \tag{3.2}$$

The **relative support** of s in $\mathcal{C}$ is denoted by $RelSupp(s, \mathcal{C}) = \frac{Support(s, \mathcal{C})}{|\mathcal{C}|}$.

Note that $RelSupp(s, \mathcal{C})$ may be higher than 1. We often omit the string column $\mathcal{C}$ if it is clear from the context.

Definition 3.2 (Columnar String Pattern Mining)**.** Given a minimum threshold of relative support $minRelSupp > 0$ (or a minimum threshold of support $minSupport > 0$), the **problem of columnar string pattern mining** is to find all strings s such that $RelSupp(s, \mathcal{C}) \geq minRelSupp$ (or $Support(s, \mathcal{C}) \geq minSupport$, if $minSupport$ is given instead of $minRelSupp$). Every such string s is called a *string pattern*, and we say that s is *frequent*.

Example 3.1. Table 3.1 illustrates the string patterns of a column of 4 strings, where $minSupport = 2$, and $minRelSupp = 0.5$.

$\mathcal{C}$
abc
acd
bcd
abcab

(a) A string column $\mathcal{C}$

String pattern	Support
ab	3
bc	3
cd	2
abc	2

(b) string patterns with minimum support 2

Table 3.1: An example of string patterns

The *problem of sequential pattern mining* is a related problem defined in previous literature [4]. It has 3 major differences from columnar string pattern mining:

1. A *sequence* in sequential pattern mining is defined as a list of itemsets, while a string in columnar string pattern mining is defined as a list of items (symbols).

2. A *sequential pattern* is supported by a sequence if the pattern appears in the sequence with or without gaps. A string pattern s is supported by a string s' only if s appears in s' as a consecutive substring.

3. The relative support of a sequential pattern is no larger than 1, because each sequence in a sequential database contributes at most 1 to the support of a sequential pattern. The relative support of a string pattern s can be larger than 1, because all appearances of s in different positions of a string $\mathcal{C}[i]$ are counted towards the support.

The anti-monotonicity property of frequent sequential patterns also applies to string patterns.

Property 3.1 (Anti-monotonicity for String Patterns)**.** If s is a substring of s', then $Support(s) \geq Support(s')$. If s is infrequent, so is s'.

Proof. Assume s is a substring of s' at p, i.e. $s \preceq_p s'$. If for a string $\mathcal{C}[i]$ we have $s' \preceq_q \mathcal{C}[i]$, then by the definition of substring, $s \preceq_{q+p-1} \mathcal{C}[i]$. Then we have

$$\begin{aligned} Support(s, \mathcal{C}[i]) &= |\{r | s \preceq_r \mathcal{C}[i]\}| \\ &\geq |\{q+p-1 | s' \preceq_q \mathcal{C}[i]\}| = |\{q | s' \preceq_q \mathcal{C}[i]\}| = Support(s', \mathcal{C}[i]) \end{aligned}$$

Therefore,

$$Support(s, \mathcal{C}) = \sum_{1 \leq i \leq |\mathcal{C}|} Support(s, \mathcal{C}[i]) \geq \sum_{1 \leq i \leq |\mathcal{C}|} Support(s', \mathcal{C}[i]) = Support(s', \mathcal{C})$$

□

PrefixSpan [27] is a state-of-the-art algorithm for sequential pattern mining, and can be easily adapted to mine frequent string patterns, as discussed in the next chapter.

3.2 A Semi-static Encoding Format

In this section, we introduce a generic semi-static encoding format as the basis for the output of our compression algorithm. It is helpful to first introduce some compression-related concepts.

Definition 3.3 (Factorization)**.** A **factorization** of a string s, denoted by $s = s_1 s_2 \cdots s_k$ is a partition of s into a sequence of substrings $s_1, s_2, \ldots, s_k$. Each substring s_i for $1 \leq i \leq k$ is a **factor** of the factorization. A **factorization** of a column $\mathcal{C}$ is the sequence of factorizations of each string $\mathcal{C}[i]$ for all $1 \leq i \leq |\mathcal{C}|$.

Definition 3.4 (Cover and Dictionary)**.** A set of strings $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ is said to **cover** a string s, if there exists a factorization $s = s_1 s_2 \ldots s_k$ such that $s_i \in \mathcal{P}$ for all $1 \leq i \leq k$. $\mathcal{P}$ **covers** a column $\mathcal{C}$ if $\mathcal{P}$ covers $\mathcal{C}[i]$ for all $1 \leq i \leq |\mathcal{C}|$. A sequence that is an arbitrary ordering of $\mathcal{P}$ is called a **dictionary**, denoted by *Dict*. If $p \in \mathcal{P}$, then we say $p \in Dict$. Any string $p \in Dict$ is called a *phrase*. The index of a phrase p in *Dict* is

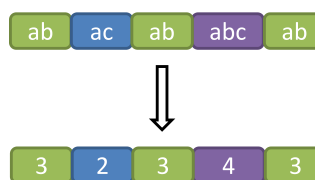
the *integer code* of p, also called a *dictionary index*, denoted by $d_{Dict}(p)$ (written as $d(p)$ whenever clear from the context). If $s = s_1 s_2 \cdots s_k$ is a factorization of s, and $s_i \in Dict$ for all $1 \leq i \leq k$, we say that the sequence $d(s_1), d(s_2), \ldots, d(s_k)$ is a *factorization* of s with $Dict$. Each integer code $d(s_i)$ in the factorization is alternatively called a *factor* of the factorization with $Dict$.

Note that a factor may refer to a substring, or the integer code of a substring, depending on the context.

Example 3.2. A dictionary for string *abacababcab* is shown in Figure 3.1. A factorization of s with this dictionary is $\langle 3, 2, 3, 4, 3 \rangle$, which creates a factoring by repeatedly finding the greedy longest match in the dictionary.

Dictionary Index	Phrase
0	*a*
1	*b*
2	*ac*
3	*ab*
4	*abc*

(a) A dictionary for string *abacababcab*



(b) A factorization of *abacababcab* with the dictionary

Figure 3.1: An example of string factorization by greedy matching

Throughout this thesis, we use the term *compression ratio* to refer to the fraction where the numerator is the byte length of the data in a compressed format, and the denominator is the byte length of the uncompressed data. The lower the compression ratio is, the better.

Table 3.2 outlines an encoding format for a compressed column block, where a column block is a consecutive subsequence of a column. We compress a large column in blocks so that a single block can be decompressed without decompressing the entire column. Both field 5 and field 6 are compressed sequence of integers. They are compressed using the Frame of Reference (FOR) method, which subtracts each integer with the minimum integer in the sequence, and compresses the differences with binary packing. Binary packing is a scheme that uses $\lceil \log_2 d_{max} \rceil$ bits to represent each integer in a sequence, where the maximum integer is d_{max}. For the integer codes (dictionary indexes) in field 6, since the minimum value is 0, the compression ratio of FOR should be very close to binary packing, where the bit length of a compressed integer code is $\lceil \log_2(|Dict|) \rceil$. In our implementation, we use SIMDPFor from the research library FastPFor [33]. It is an implementation of FOR using SIMD instructions whenever possible to speed up compression. Since one of our goals is to design a fast compression algorithm, we choose to use FOR instead of entropy coding such as Huffman coding [25]. Although Huffman coding can yield better compression ratio,

it incurs a slower compression and decompression speed, so it is usually not used in fast compression algorithms such as Snappy.

ID	Field Name	Size (Bytes)	Description
1	Block size	8	64 bit integer representing the size of the current block.
2	Number of strings	4	32 bit integer representing the number of strings in the column block.
3	Dictionary	Variable	The dictionary *Dict* for the column block. Each phrase is prefixed by 16 bit integer (2 bytes) denoting its length. Two 0 bytes mark the end of this field.
4	Size of compressed string lengths	4	32 bit integer representing the size of the next field.
5	Compressed string lengths	Variable	A sequence of integers compressed using FOR. The integers represent the lengths of the strings in the column block.
6	Compressed integer codes	Variable	A sequence of integer codes compressed using FOR. The integer code sequence represents a factorization of the column block with *Dict*.

Table 3.2: A semi-static encoding format for a compressed column block

We distill the dictionary and the compressed integer codes from this concrete format into the following definition. The other fields are omitted because their sizes are not affected by the selection of dictionary phrases, and the factorization of the column.

Definition 3.5 (Semi-static Encoding Format $\mathcal{E}$)**.** Let the alphabet be all byte values. The **encoding format** $\mathcal{E}$ is a sequence of bytes consisting of two parts, M followed by D. M denotes a byte representation of a dictionary $Dict$. M consists of a sequence of elements, where each element is a byte representation of a phrase in $Dict$. The elements in M follow the same order as the phrases in $Dict$. The byte representation of each phrase $p \in Dict$ is the string of p prefixed by 2 bytes denoting the length of p. M ends with 2 empty bytes. D denotes a byte representation of a factorization of a string column $\mathcal{C}$ with $Dict$. D is a bit sequence, where every $\lceil \log_2(|Dict|) \rceil$ bits denote a dictionary index (a factor) in the factorization, following the order of the factorization.

3.3 The Problem of Optimal Compression with Encoding Format $\mathcal{E}$

With the encoding format specified, it is desirable to attain a compression ratio as small as possible and as efficiently as possible. We define an optimization problem for attaining the smallest compression ratio.

Definition 3.6 (Optimal Compression with Semi-static Encoding Format $\mathcal{E}$)**.** For a given string column $\mathcal{C}$, find a dictionary $Dict$ and a factorization of $\mathcal{C}$ with $Dict$ such that the byte length of $\mathcal{C}$ compressed in the format $\mathcal{E}$ is minimized.

We refer to this problem as *optimal compression* in the remaining of this thesis.

Definition 3.7 (Minimum Factorization with Static Dictionary)**.** Given a dictionary $Dict$ and a string s, find a factorization of s with $Dict$ such that the factorization contains the minimum number of factors. For such a factorization, we say that s is *minimally factorized.*

We refer to this problem as *minimum factorization* in the remaining of this thesis. Example 3.2 in Section 3.2 shows a minimum factorization. For this example, there is no other factorization resulting in less than 5 factors.

The following lemma states that a solution of optimal compression must be at the same time a solution of minimum factorization.

Lemma 3.1. Given an optimal dictionary for the optimal compression problem, the compression is optimal if and only if the factorization is minimum.

Proof. Assume a minimum factorization f with the given optimal dictionary does not yield optimal compression. Let f' be the factorization in an optimal compression. If we substitute f' with f, the number of factors does not increase. Since the bit length of each integer code is fixed to be $\lceil \log_2(|Dict|) \rceil$ where $Dict$ is given, the substitution does not increase the compressed size, so the compression is still optimal. This is a contradiction.

Assume in an optimal compression, the factorization is not minimum. Substituting the factorization with a minimum factorization would decrease the number of factors, and decrease the compressed size, so the given compression is not optimal, which is a contradiction. □

The following lemma states that the dictionary should not contain any unused phrase.

Lemma 3.2. In an exact solution to the optimal compression problem, where the dictionary is $Dict$, for any dictionary phrase $p \in Dict$, its integer code $d(p)$ appears in the factorization of a string s in $\mathcal{C}$ with $Dict$.

Proof. Given a solution to the optimal compression problem, if there is a phrase $p \in Dict$ not used in the factorization, then removing p from $Dict$ results in a smaller representation

of the dictionary without changing the factorization. In addition, it may result in shorter integer code, since $|Dict|$ is reduced by 1. Therefore the solution cannot be optimal. □

Finding an optimal dictionary in various encoding formats is NP-complete as proved in the literature [42]. However, solving minimum factorization is not hard, which we will discuss in more detail. Lemma 3.1 implies a possible heuristic direction for approaching the optimal compression problem. First, a heuristic dictionary is found, which captures the repetitiveness in the input column $\mathcal{C}$. Then, we find a minimum factorization for every string s in $\mathcal{C}$ with the dictionary.

In Chapter 4, we will give a dynamic programming algorithm for the minimum factorization problem, which runs in $O(\max_{p \in Dict}(|p|)|s|)$ time. The idea is that the minimum factorization of a string s with *Dict* can be obtained by examining every case of removing a suffix p of s where $p \in Dict$, and picking the case where the remaining string has the smallest minimum factorization. Another solution is to transform the problem into a shortest path problem on a graph, where each node represents a substring p of s where $p \in Dict$. Certain path on such a graph represents a concatenation into s. The details of this algorithm are left to the readers.

Some semi-static compression schemes use local lookahead methods for factorization [12]. One such method evaluates two cases: a greedy match starting from the current input symbol, and one starting from the next. If the latter match is longer, then the current input symbol is sent to the output as a literal, the input position is advanced by 1, and the comparison starts again [12]. The goal is to heuristically look for long matches (substring factors). This is similar to minimum factorization, because if the number of factors is minimized, then the average length of substring factors is maximized. Therefore, local lookahead can be viewed as a heuristic method for approaching the minimum factorization problem in linear time, i.e. $O(|s|)$ for an input string s. In Chapter 4, we show that if *Dict* only contains the alphabet and all frequent string patterns with a fixed *minSupport*, then the minimum factorization problem can be optimally solved in $O(|s|)$ with a simple greedy matching algorithm with no local lookahead. This is a nice property for using frequent string patterns for phrase selection in semi-static compression.

Finding a good dictionary is important for compression. If a large *Dict* is used, then even if the number of factors is minimized, the bit length of each integer code can be long, resulting in a larger compressed factor stream. Furthermore, a larger dictionary itself takes more space. It is desirable to have a small dictionary which at the same time has a small minimum factorization. In Chapter 4, we propose a sampling-based method using frequent string patterns as the dictionary for compression, with a focus on compression speed. Although the method does not have an optimality guarantee, it is shown in experiments to work well on many types of columnar data.

Chapter 4

Compressing Columnar Data with Frequent String Patterns

In this chapter, we develop our compression method for string columnar data. It is a semi-static compression method, where the dictionary phrases are frequent string patterns from a sample of the data. First, we illustrate a pattern mining method for string columnar data based on PrefixSpan [27]. Second, we describe a dynamic programming algorithm for the minimum factorization problem. Third, we describe a greedy factorization algorithm which we use in our compression method, and show some nice properties of the algorithm. Last, we describe a trie pruning method to improve compression speed.

4.1 PrefixSpan for Columnar String Pattern Mining

PrefixSpan is a pattern mining method for sequential databases based on the pattern-growth paradigm [27, 20]. Here we use a simple example to illustrate how to mine frequent string patterns using a variation of PrefixSpan for string columns. The major difference is that this method mine string patterns as defined in Section 3.1. That is, this variation does not consider gaps in a pattern, and counts multiple appearances of a substring p in a string s towards the support of p in s. The differences between string patterns and sequential patterns were discussed in more detail in Section 3.1.

In the original PrefixSpan algorithm for sequential patterns, a technique called *pseudo-projection* was introduced. This technique uses positional pointers to represent the *prefix-projection*, which examines only the postfix subsequences corresponding to the frequent prefix subsequences [27]. Here we also adopt pseudo-projection for mining string patterns. To store the frequent string patterns found, we construct a prefix tree (trie) on the fly.

Example 4.1. We use the string column from Example 3.1 as the input, denoted by $\mathcal{C}$, shown again in Table 4.1. Each field of this column is a string. The problem is to find all frequent string patterns with $minSupport = 2$. Note that as defined in Section 3.1, a string

pattern s is supported by a string s' only if s is a substring of s' without gaps. Also note that $Support(s, s')$ can be higher than 1 if s appears at several different positions in s'.

$\mathcal{C}$
abc
acd
bcd
abcab

Table 4.1: An example column for columnar string pattern mining

Step 1. Let a pair (i, j) denote the position in the column $\mathcal{C}$ of the j-th symbol in the i-th string, where i and j are 1-based indexes. For a substring p of a string s in $\mathcal{C}$, the position of p is the position of the first symbol of p in $\mathcal{C}$. For example, the substring *bc* in *abc* is at position $(1, 2)$. In the first step, for each symbol we gather all its positions into a list. We store these lists as the children of the root in a prefix tree, shown in Figure 4.1.

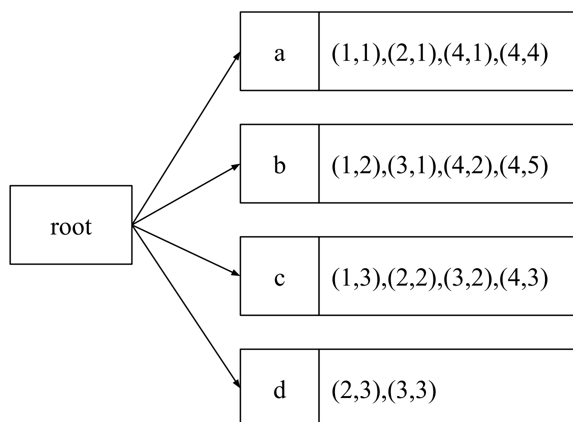


Figure 4.1: A prefix tree containing all symbols in $\mathcal{C}$ and their positions

Step 2. From Figure 4.1 we know that all symbols have a support higher than 2, so they are all frequent. Note that multiple appearances of a symbol in a string are counted multiple times towards the symbol's support. For example, a has a support of 4, where it appears twice in the string *abcab*. Then, we search the positions of a for all frequent length-2 patterns starting with a. To achieve this, we group the positions of a by different length-2 substrings starting with a. For example, the appearances of a at $(1, 1)$ and $(4, 1)$ are followed by b, so *ab* has positions $(1, 1)$ and $(4, 1)$. The grouped positions are stored as lists in the children of the node a in the prefix tree, shown in Figure 4.2. There are 2 such substrings, *ab* and *ac*. Only *ab* has a support greater than or equal to *minSupport*, so only *ab* is frequent. Since *ac* is infrequent, its node

in the prefix tree is removed. Here, the frequent string pattern ab is found by *pattern growth* on the prefix a. That is, we grow the frequent prefix a into a frequent string pattern ab by examining all extensions of a.

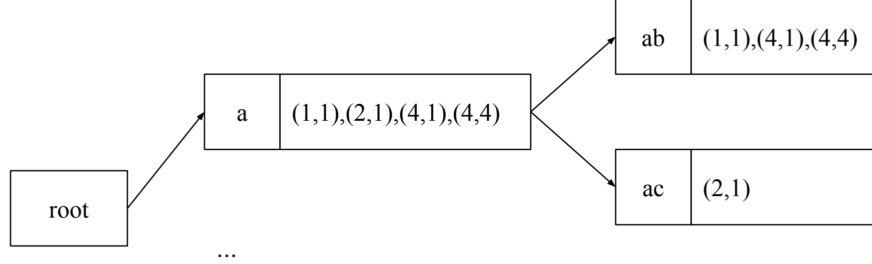


Figure 4.2: Grouping positions of a by length-2 substrings starting with a

Step 3. We search the positions of ab for frequent length-3 patterns starting with ab. Since 2 of the 3 appearances of ab are followed by c, the prefix ab is grown into a frequent string pattern abc. There is no frequent length-4 pattern starting with abc. There is no other frequent length-3 pattern starting with ab, or frequent length-2 pattern starting with a, so we go back to Step 2 to search for frequent string patterns starting with the next frequent symbol b. The algorithm terminates when all frequent extensions of each frequent symbol are explored. When the algorithm terminates, all frequent string patterns shown in Example 3.1 are found.

This is a recursive, depth-first search procedure producing a prefix-tree containing all frequent string patterns found. We refer to this algorithm as *PrefixSpan for columnar string pattern mining*. The pseudocode of this algorithm is listed in Algorithm 1 in the next page.

Lemma 4.1. PrefixSpan for columnar string pattern mining finds the complete set of frequent string patterns in a string column $\mathcal{C}$.

Proof. We prove that if a substring p is frequent in $\mathcal{C}$, then the algorithm identifies p as frequent, and finds all positions of p in $\mathcal{C}$.

If $|p| = 1$, i.e. p is a symbol, then all its positions are found in line 6-14 in Algorithm 1. The algorithm identifies p as frequent by calculating the support of p from the number of positions of p, by the definition of support. If $|p| > 1$, then by Property 3.1 (anti-monotonicity, page 16), all non-empty prefixes of p are frequent. Let us assume that, either the algorithm does not identify p as frequent (we say that p is *unidentified*), or there is a position of p not found by the algorithm (we say that there is an *unidentified position* of p). Then we claim that for all $0 \leq k < |p|$, either the frequent prefix $p[1, |p| - k]$ is unidentified, or there is an unidentified position of $p[1, |p| - k]$. This claim is proved by induction as follows.

Base Case. Let $k = 0$. For the prefix $p[1, |p|]$, i.e. p itself, the assumption specifies that either p is unidentified, or there is an unidentified position for p.

Algorithm 1 PrefixSpan for Columnar String Pattern Mining

Input: A string column $\mathcal{C}$, and a minimum support *minSupport*;
Output: A prefix tree T containing all frequent string patterns of $\mathcal{C}$;

```
1:  T ← empty prefix tree with only a root node
2:  root ← root node of T
3:  Let l(·) to denote the length of a string or list
4:  Let a pair (i, j) denote the position in C of the j-th symbol in the i-th string
5:  Let n[c] to denote a null pointer, or a child of node n in T, where n indexes a string s,
    and n[c] indexes the string s extended by the symbol c
6:  for all position pos in C do
7:      c ← the symbol at pos in C
8:      if root[c] is a null pointer then
9:          root[c] ← a new node
10:         root[c].pattern ← c
11:         root[c].Positions ← empty list
12:     end if
13:     append pos to root[c].Positions
14: end for
15: for all non-null child in children of root do
16:     if l(child.Positions) ≥ minSupport then
17:         DfsMinePatterns(child)
18:     else
19:         child ← a null pointer
20:     end if
21: end for
22:
23: function DfsMinePatterns(node)
24:     GroupPositionsByChildren(node)
25:     for all non-null child in children of node do
26:         if l(child.Positions) ≥ minSupport then
27:             DfsMinePatterns(child)
28:         else
29:             child ← a null pointer
30:         end if
31:     end for
32: end function
```

```
33: function GroupPositionsByChildren(node)
34:     for all pos ∈ node.Positions do
35:         if the position (pos.i, pos.j + l(pattern(node))) exists in 𝒞 then
36:             c ← the symbol at (pos.i, pos.j + l(pattern(node))) in 𝒞
37:             if node[c] is a null pointer then
38:                 node[c] ← a new node
39:                 node[c].pattern ← node.pattern appended with c
40:                 node[c].Positions ← empty list
41:             end if
42:             append pos to node[c].Positions
43:         end if
44:     end for
45: end function
```

Inductive Step. Let $0 < k < |p|$. The induction hypothesis is that for $p' = p[1, |p| - (k-1)]$, either p' is unidentified, or there is an unidentified position of p'. p' is frequent since it is a prefix of p. Let $p'' = p[1, |p| - k]$, i.e. p'' is the prefix of p' (and p) with length $|p'| - 1$. p'' is frequent since it is a prefix of p. Assume that the algorithm identifies p'' as frequent, and the algorithm identifies all positions of p''. Then, the algorithm would be able to search the positions of p'' and find all positions of p', because every position of p' must be a position of p''. Then, the algorithm would identify p' as frequent by calculating its support from the number of its positions. This contradicts the induction hypothesis, so either p'' is unidentified, or there is an unidentified position of p''.

By the principle of mathematical induction, for all $0 \leq k < |p|$, either the frequent prefix $p[1, |p| - k]$ is unidentified, or there is an unidentified position of $p[1, |p| - k]$. This holds for $k = |p| - 1$, where $p[1, |p| - k] = p[1, 1]$, which is the first symbol of p. However, this contradicts the fact that all positions of each symbol are identified by the algorithm in lines 6-14, and all frequent symbols are identified as frequent. By this contradiction, we proves the initial claim that the algorithm identifies p as frequent, and identifies all positions of p. The completeness of the algorithm is proved. □

Lemma 4.2. PrefixSpan for columnar string pattern mining does not identify any infrequent pattern as frequent.

Proof. We prove that for a substring p identified by the algorithm as frequent in $\mathcal{C}$, and (i, j) identified as a position of p, p indeed appears in $\mathcal{C}$ at (i, j). We say that (i, j) is *correctly identified* for p. Because the algorithm calculates the support of p from the number of identified positions of p, if every identified position of p is correct, then the calculated support is no larger than the true support, so p must be frequent.

If $|p| = 1$, i.e. p is a symbol, then every position found for p in Algorithm 1 is correctly identified. If $|p| > 1$, let us assume that there is an *incorrectly identified* position (i, j)

for p, i.e. (i, j) is identified as a position of p by the algorithm, but p does not appear at (i, j). Then we claim that for all $0 \leq k < |p|$, (i, j) is incorrectly identified for the prefix $p[1, |p| - k]$. This claim is proved by induction as follows.

Base Case. Let $k = 0$. For the prefix $p[1, |p|]$, i.e. p itself, the assumption specifies that (i, j) is incorrectly identified for p.

Inductive Step. Let $0 < k < |p|$. The induction hypothesis is that for $p[1, |p| - (k - 1)]$, (i, j) is incorrectly identified. Let $p' = p[1, |p| - (k - 1)]$. Let $p'' = p[1, |p| - k]$, i.e. the prefix of p' (and p) with length $|p'| - 1$. It follows that the algorithm identifies p' as frequent when exploring extensions of p'', and the symbol at $(i, j + |p''|)$ is the same as the last symbol of p'. Also, the algorithm identifies p'' as frequent. Since all identified positions of p' are found by searching the identified positions of p'', (i, j) must be an identified position of p''. Assume (i, j) is correctly identified for p''. Then the algorithm would correctly identify (i, j) for p' because the symbol at $(i, j + |p''|)$ is the same as the last symbol of p', but this contradicts the induction hypothesis. Therefore, (i, j) must be incorrectly identified for p''.

By the principle of mathematical induction, for all $0 \leq k < |p|$, (i, j) is incorrectly identified for the prefix $p[1, |p| - k]$. This holds for $k = |p| - 1$, where $p[1, |p| - k] = p[1, 1]$, which is the first symbol of p. This contradicts the fact that all the positions identified for each symbol in Algorithm 1 are correct. Thus we proves the initial claim that every identified position of p is correct. The correctness of the algorithm is proved. □

Combining Lemma 4.1 and Lemma 4.2, we have the following theorem.

Theorem 4.3. A substring p is a frequent string pattern if and only if PrefixSpan for columnar string pattern mining says so.

We analyze the running time as follows. For a position (i, j) in $\mathcal{C}$, let p be the longest frequent string pattern at (i, j). That is, the substring at (i, j) of length $|p| + 1$ does not exist, or is infrequent. During pattern growth, $O(|p|)$ running time is spent on appending the position (i, j) into the position list of every prefix of p, i.e., the positions lists of $p[1], p[1, 2], \ldots, p[1, |p| - 1], p$. Overall, the algorithm runs in $O(l_{max} \sum_{1 \leq i \leq |\mathcal{C}|} |\mathcal{C}[i]|)$ time, where l_{max} is the length of the longest frequent string pattern(s). One worst-case scenario is that every symbol in $\mathcal{C}$ is the same. In this case, the running time of the algorithm is $O(\sum_{1 \leq i \leq |\mathcal{C}|} |\mathcal{C}[i]|^2)$.

Since we derive the frequent string patterns from a sample of the input column, only a small proportion of the compression time is spent on pattern mining. The majority of the compression time is spent on the factorization of the entire input column, which is described in Section 4.3. In a practical implementation, it may be desirable to impose a constraint on the maximum pattern length to further reduce the cost of pattern mining.

4.2 Solving the Minimum Factorization Problem with Dynamic Programming

In Section 3.3, we defined the optimal compression problem, which tries to find an optimal dictionary to compress an input string column into the smallest size. We also defined the minimum factorization problem: given a set of dictionary phrases, partition an input string into the least amount of *substring factors*, where each substring factor is a dictionary phrase. It was proved that, under our encoding format (with fixed-length codes), *if an optimal dictionary is given*, then the minimum factorization of the input column yields the optimal compression (Lemma 3.1).

Note the condition in the above property. If a non-optimal dictionary is given, then the minimum factorization does not necessarily yield the best compression among all possible factorizations. Consider the simple case of compressing a string s, with a dictionary containing s itself as a phrase. The minimum factorization has 1 factor, which is s itself. But this does not yield any compression, although another factorization with some other dictionary phrases may compress the string s. However, Lemma 3.1 still points to a heuristic direction for approaching the optimal compression problem. First, we find a heuristic dictionary that succinctly captures the repetitiveness of the input column. Then, we obtain a minimum factorization of the input column to heuristically approach compression, given that the bit length of each integer code is constrained by the dictionary.

In this section, we present a dynamic programming solution to optimally solve the minimum factorization problem. The basic idea comes from the following observation: given a string s, if $s = p_1 p_2 \cdots p_m$ is a minimum factorization of s (where each p_i is a phrase in the given dictionary), then $s' = p_1 p_2 \cdots p_{m-1}$ must be a minimum factorization of s'. This recurrence relation is formally described in Lemma 4.4 with a proof. Example 4.2 shows an example of a dynamic programming algorithm based on the recurrence relation.

Lemma 4.4. Given a dictionary *Dict*, let $OPT(s)$ denote the minimum number of factors that can be achieved by a factorization of s with *Dict*. If s is not an empty string, and *Dict* does not cover s, then let $OPT(s) = \infty$. Then the following recurrence relation holds:

$$OPT(s) = \begin{cases} 0 & \text{if } |s| = 0, \\ \infty & \text{if } |s| > 0 \text{, and } Dict \text{ does not cover } s, \\ \min\limits_{p \in Dict, s = s'p} (1 + OPT(s')) & \text{otherwise.} \end{cases} \tag{4.1}$$

Proof. The case where s is an empty string is trivial. If s is not empty, and *Dict* covers s, let us assume that $OPT(s) \neq \min_{p \in Dict, s = s'p} (1 + OPT(s'))$.

Case 1. $OPT(s) > \min_{p \in Dict, s = s'p} (1 + OPT(s'))$. Then the right-hand side cannot be ∞, and there exists p' such that $p = p' \in Dict$, $s = s''p'$ achieves minimization on the right-

hand side. Then we have $OPT(s) > 1 + OPT(s'')$, and $OPT(s'') \neq \infty$ (i.e. there is a factorization of s''). If we factorize s such that the last factor is p', and the remaining prefix s'' is factorized corresponding to $OPT(s'')$, then the number of factors in such a factorization of s is exactly $1 + OPT(s'')$, which contradicts $OPT(s) > 1 + OPT(s'')$.

Case 2. $OPT(s) < \min_{p \in Dict, s=s'p} (1 + OPT(s'))$. Then $OPT(s) \neq \infty$. Let the factorization corresponding to $OPT(s)$ be $s = p_1 p_2 \cdots p_m$, where $OPT(s) = m \geq 1$. Let s'' be such that $s = s'' p_m$. If $m = 1$, then s'' is an empty string, so $m - 1 = OPT(s'') = 0$. If $m > 1$, then $s'' = p_1 p_2 \cdots p_{m-1}$, i.e. there is an factorization of s'' with $m - 1$ factors, so $m - 1 \geq OPT(s'')$. In both cases, $m - 1 \geq OPT(s'')$. Since $OPT(s) = m$, we have $OPT(s) \geq 1 + OPT(s'') \geq \min_{p \in Dict, s=s'p} (1 + OPT(s'))$, which is a contradiction.

□

Note that for the case where s is not empty and *Dict* does not cover s, we specify $OPT(s) = \infty$. In practice, we should either allow the dictionary to contain the alphabet so that the input is always covered, or have certain literal encoding scheme to deal with the case that the input is not covered.

Example 4.2. Let the input string be $s = abacd$. Let $Dict = \langle a, b, c, d, ab, bac \rangle$. Note that *Dict* is a sequence of distinct phrases (so that each phrase is associated with an index). Also note that *Dict* contains every symbol in s, so *Dict* covers every substring of s. Let an empty string be denoted by e.

Step 1. It is trivial to show that $OPT(a) = 1$, where the dictionary phrase a is used to factorize the input string a into 1 factor.

Step 2. For ab, there are two suffixes that are in the dictionary, b and ab, and the corresponding remaining prefixes are a and an empty string, respectively. It follows that $OPT(ab) = \min(1 + OPT(a), 1 + OPT(e)) = \min(2, 1) = 1$.

Step 3. For aba, there is only one suffix that is in the dictionary. This suffix is a. It follows that $OPT(aba) = min(1 + OPT(ab)) = 2$. That is, the minimum factorization of aba is $\langle ab, a \rangle$.

Step 4. For $abac$, two suffixes are in the dictionary, c and bac. The corresponding remaining prefixes are aba and a, respectively. Since $OPT(aba) = 2 > OPT(a) = 1$, we have $OPT(abac) = \min(1 + OPT(aba), 1 + OPT(a)) = 2$, where bac is chosen as the last factor, and the factorization $\langle a, bac \rangle$ is chosen over $\langle ab, a, c \rangle$. However, if we use a greedy method where we immediately choose the longest available factor as the next, we would have the factorization $\langle ab, a, c \rangle$, which is not the minimum factorization of $abac$. This greedy method however has other attractive properties, which are discussed in detail in the next section.

Step 5. For $abacd$, i.e. the entire input string s, the only suffix that is in the dictionary is d. $OPT(abacd) = \min(1 + OPT(abac)) = 3$. The minimum factorization of the string s is $\langle a, bac, d\rangle$.

The algorithm can be implemented as follows. A pointer array A of length $|s|$ is created, where each $A[j]$ for $1 \leq j \leq |s|$ points to a set of indexes. Each index i in the set pointed by $A[j]$ is such that $1 \leq i \leq j$, and the substring $s[i, j]$ is a dictionary phrase, i.e. $s[i, j] \in Dict$. $A[j]$ should contain every such index i for j. We scan the input string s from left to right to create the array A. Note that we do not need to test the cases where $j - i > \max_{p \in Dict} |p|$, i.e. $s[i, j]$ is longer than the longest dictionary phrase. Thus creating the array A runs in $O(|s| \max_{p \in Dict} |p|)$ time, if we index $Dict$ with a prefix tree. Then, for computing the minimum factorization of each prefix $s[1, j]$ for $1 \leq j \leq |s|$, we iterate through every index i in the set pointed by $A[j]$. We examine each case of removing such a suffix $s[i, j]$, and compute the minimum factorization of $s[1, j]$ according to equation 4.1. Overall, the running time is $O(|s| \max_{p \in Dict} |p|)$.

This algorithm is presented in pseudocode as follows. Note that we also maintain an array $MinFact$ of minimum factorizations for each prefix of s. In a proper implementation, the time complexity of maintaining $MinFact$ should be the same as maintaining OPT. The indexes for A, OPT and $MinFact$ are 0-based.

Algorithm 2 Dynamic Programming for Minimum Factorization

Input: An input string s, and a dictionary $Dict$;
Output: A minimum factorization $MinFact$ of s with $Dict$;

```
1:  A ← array of size |s| + 1, where each element is an empty set
2:  OPT ← array of size |s| + 1, where each element is initialized with 0
3:  MinFact ← array of size |s| + 1, where each element is an empty factorization
4:  for all (i, j) such that s[i, j] ∈ Dict do
5:      append (i, j) to the set in A[j]
6:  end for
7:  for all 0 < j ≤ |A| do
8:      if A[j] is empty then
9:          OPT[j] ← ∞
10:     else
11:         (i*, j) = arg min_{(i,j)∈A[j]} (1 + OPT[i − 1])
12:         OPT[j] = 1 + OPT[i* − 1]
13:         if OPT[j] ≠ ∞ then
14:             MinFact[j] = MinFact[i* − 1] appended with s[i*, j]
15:         end if
16:     end if
17: end for
```

4.3 Greedy Matching for Factorization and Phrase Selection

In the previous section, we presented a dynamic programming solution to the minimum factorization problem, which runs in $O(|s| \max_{p \in Dict} |p|)$ time. For data compression, we have not seen this method used in practice (and presume it is rare). In this section, we discuss two alternative methods for factorization, a *greedy matching* algorithm, and *local lookahead* methods. We focus the discussion on the former, which is used in our compression method.

Factorization refers to parsing an input column top-to-bottom and left-to-right, and partitioning each string into a sequence of substrings such that each substring is a dictionary phrase. Each such substring can be replaced by a dictionary index to achieve compression. Such a substring for replacement is called a *match*. We also say it is a *substring factor* of the factorization. The idea of greedy matching is that, the longest match starting from the current input symbol is found, and is immediately replaced by its dictionary index. Later, we will present the pseudocode of this algorithm, which has more implementation subtleties. For now, let us demonstrate the basic idea by reusing the input in Example 4.2.

Example 4.3. Let the input string be $s = abacd$. Let $Dict = \langle a, b, c, d, ab, bac \rangle$. Perform a greedy matching to factorize s with $Dict$.

Step 1. The current input string is $abacd$. The longest prefix such that the prefix is a dictionary phrase is ab. We say that such a prefix is the *longest match*. The algorithm immediately declares ab as the first factor. The remaining input string is acd.

Step 2. The longest match for acd is a, so the factorization is appended by a new factor a, and becomes $\langle ab, a \rangle$. The remaining string is cd.

Step 3. The longest match for cd is c, so the factorization is appended by a new factor c, and becomes $\langle ab, a, c \rangle$. Similarly, in step 4 the last longest match d is found. The complete factorization of s is $\langle ab, a, c, d \rangle$.

This greedy algorithm is in contrast to a local lookahead method [12]: alternative matches starting from the following input symbols are also evaluated, and a certain measure (e.g. lengths of matches) is used to decide whether to defer the replacement of a match. The following example demonstrates one such method, where a single alternative match is evaluated.

Example 4.4. In this example, we demonstrate a local lookahead method for factorization. We use the same input as Example 4.3.

Step 1. The input string is currently $abacd$. The longest match is ab. We evaluate an alternative match: the longest match starting from the next input symbol b. This

alternative match is *bac*. Since the alternative match is longer, we abandon the former match *ab*, and use the current input symbol a as a factor (or send a to the output as a literal, depending on the details of the compression algorithm). The factorization so far is $\langle a \rangle$, and the remaining input string is *bacd*. Note that we do not immediately declare *bac* as the next factor. The decision of factoring is *deferred*.

Step 2. The input string is currently *bacd*. We already know that the longest match is *bac* from step 1. The alternative match starting from the next input symbol a is a. Since the alternative match is not longer, we use the current match *bac* as a factor. The factorization becomes $\langle a, bac \rangle$, and the remaining input string is d.

Step 3. The last factor d is found, and the complete factorization is $\langle a, bac, d \rangle$. Interestingly, this gives the same optimal solution to the minimum factorization problem as the dynamic programming example in the previous section.

Greedy matching may not be as effective as local lookahead for compression [12]. Local lookahead methods based on match lengths try to achieve a similar goal to the dynamic programming algorithm for minimum factorization. Specifically, the dynamic programming algorithm achieves minimum number of factors, and therefore maximum average length of matches. Local lookahead methods also try to find long matches and use them as factors, and therefore heuristically improve compression. It is reported in previous literature that with a dictionary generated with certain methods, local lookahead achieves effective compression, while greedy matching with no local lookahead does not [12].

However, local lookahead methods are also slow, as we will show in the experiments. Let us examine one particular local lookahead method [12], which is already demonstrated in Example 4.4. Let the longest match starting from the current input symbol be the *left match*. Let the longest match starting from the next input symbol be the *right match*. If the left match is longer, then it is immediately replaced by a dictionary index, and the input position is advanced to skip the left match. Otherwise, the current input symbol is encoded as a literal, the input position is advanced by 1 symbol, and the previous right match is used as the left match for the next comparison. This represents a local lookahead scheme with a search window of length 2. In the best case, every left match is longer than their corresponding right match, and the running time is $O(|s|)$ (in this case, local lookahead performs a less exhaustive search than the dynamic programming algorithm). In the worst case, every right match is longer, and the running time is the same as the dynamic programming algorithm, i.e. $O(|s| \max_{p \in Dict} |p|)$. Even in the best case, such a method may still spend a significant amount of time on evaluating alternative matches, while the greedy matching algorithm does not.

A major idea of our work is to use a set of frequent string patterns as the heuristic dictionary. Later we will prove that, if a complete set of string patterns are used as the

dictionary, then the greedy matching algorithm achieves the minimum number of factors with the dictionary. The main implication is that, since local lookahead cannot further decrease the number of factors with such a dictionary, greedy matching may be sufficient for effective compression. This is confirmed in our experiments: if we use frequent string patterns as the dictionary, local lookahead methods based on match lengths not only are much slower than greedy matching, but also offer no significant improvement in terms of compression ratio.

The basic idea of the greedy matching algorithm was shown in Example 4.3. We again describe the algorithm with pseudocode in Algorithm 3. There is a subtle complication here. According to Lemma 3.2, the dictionary $Dict$ stored in the compressed data needs to contain only the set of phrases that are used in the factorization of the input column $\mathcal{C}$. Depending on the factorization method, some phrases from the dictionary may not be used, so the input dictionary may be a superset of $Dict$, i.e. the input dictionary is a *candidate dictionary*. Therefore, upon completing the greedy matching algorithm with the given candidate dictionary, we can then reduce the candidate dictionary into only those phrases that are used in the factorization. That is, the term *phrase selection* means a two-pass process in our method: the selection of a candidate dictionary by pattern mining, and the selection of a final dictionary by factorization. Algorithm 3 uses a candidate dictionary $CandDict$ for factorization, and selects a $Dict$ from $CandDict$. Note that the integer code of a phrase p is the index of p in $Dict$, not $CandDict$. To ensure $CandDict$ covers the input string column, $CandDict$ must contain at least the alphabet of the column.

Algorithm 3 Greedy Matching for Factorization

Input: A string column $\mathcal{C}$, and a candidate dictionary $CandDict$ containing at least the alphabet of $\mathcal{C}$;

Output: A dictionary $Dict \subseteq CandDict$, and a factorization of $\mathcal{C}$ with $Dict$;

1: $Dict \leftarrow$ empty list
2: $factorization \leftarrow$ empty list
3: **for all** string s in $\mathcal{C}$ **do**
4:     **while** s is not an empty string **do**
5:         $p \leftarrow$ the longest prefix of s where the prefix is in $CandDict$
6:         **if** $p \notin Dict$ **then**
7:             append p to $Dict$
8:         **end if**
9:         append $d_{Dict}(p)$ to $factorization$
10:         remove the prefix p from s
11:     **end while**
12: **end for**

Example 4.5. Let the input column contain just one string $s = abacd$, and let $CandDict = \langle a, b, c, d, ab, bac \rangle$ (same as the previous examples except that here the dictionary is treated as a candidate). The complete factorization with Algorithm 3 is $\langle ab, a, c, d \rangle$, same as Example

4.3. There is one difference: $Dict$ is constructed from $CandDict$ along with the factorization, and contains only the used phrases, so $Dict = \langle ab, a, c, d \rangle$ where the unused candidate phrases b and bac are not included. This decreases the bit length of a integer code from $\lceil \log_2(|CandDict|) \rceil = 3$ to $\lceil \log_2(|Dict|) \rceil = 2$.

A proper implementation of Algorithm 3 runs in $O(\sum_{1 \leq i \leq |\mathcal{C}|} |\mathcal{C}[i]|)$ time complexity. Finding the longest match in line 5 uses linear time to the length of the match, if $CandDict$ is indexed by a trie. When a phrase p is appended to $Dict$ in line 7, $d_{Dict}(p)$ (the index of p in $Dict$) can be stored in the trie node of p in $CandDict$, so that for future matches of p, $d_{Dict}(p)$ can be retrieved in constant time. In addition, we can use a positional pointer to implement line 10 instead of creating a copy of s with the prefix p removed.

The following lemma shows a nice property of using frequent string patterns as the heuristic dictionary, as discussed earlier in the section.

Lemma 4.5. *Given a string column $\mathcal{C}$, and a candidate dictionary $CandDict$ containing only the alphabet of $\mathcal{C}$ and a set of all frequent string patterns of a given $minSupport$, then the greedy matching algorithm achieves the minimum number of factors of every string $s \in \mathcal{C}$ with $CandDict$.*

Proof. For a string $s \in \mathcal{C}$, let the factorization produced by the greedy matching algorithm be $s = p_1 p_2 \cdots p_k$, where $p_i \in CandDict$ for $1 \leq i \leq k$. Note k is the number of factors in this factorization. Let a minimum factorization of s be $s = q_1 q_2 \cdots q_m$, where $q_j \in CandDict$ for $1 \leq j \leq m$. Note m is the minimum number of factors achievable, so $k \geq m$. Our goal is to prove $k = m$. Let us first prove such a claim: for all $1 \leq r \leq m$, we have $\sum_{1 \leq i \leq r} |p_i| \geq \sum_{1 \leq j \leq r} |q_j|$, i.e. $|p_1 p_2 \cdots p_r| \geq |q_1 q_2 \cdots q_r|$. We prove it by induction.

Base Case. Let $r = 1$. Since p_1 is the longest first factor that can be selected, we have $|p_1| \geq |q_1|$, i.e. $\sum_{1 \leq i \leq r} |p_i| \geq \sum_{1 \leq j \leq r} |q_j|$ is true when $r = 1$.

Inductive Step. Let $r > 1$. As the induction hypothesis, assume the statement to be true for $r-1$, i.e. $\sum_{1 \leq i \leq r-1} |p_i| \geq \sum_{1 \leq j \leq r-1} |q_j|$. Then either $\sum_{1 \leq i \leq r-1} |p_i| \geq \sum_{1 \leq j \leq r} |q_j|$, in which case we have $\sum_{1 \leq i \leq r} |p_i| \geq \sum_{1 \leq j \leq r} |q_j|$, or $\sum_{1 \leq i \leq r-1} |p_i| < \sum_{1 \leq j \leq r} |q_j|$, as shown in Figure 4.3. Since $q_r \in CandDict$, q_r is a symbol or a frequent string pattern. If q_r is a symbol, then every non-empty substring of q_r (i.e. only q_r itself) is in $CandDict$. If q_r is a frequent string pattern, then by Property 3.1 (anti-monotonicity, page 16), every non-empty substring of q_r is in $CandDict$. As shown in Figure 4.3, our available selections for p_r include the suffix q'_r of q_r, where q'_r is obtained by removing the prefix $p_1 p_2 \cdots p_{r-1}$ from the string $q_1 q_2 \cdots q_r$. Since we select p_r to be the longest one in all available options, including q'_r, we have $\sum_{1 \leq i \leq r} |p_i| \geq \sum_{1 \leq j \leq r} |q_j|$. This completes the inductive step.

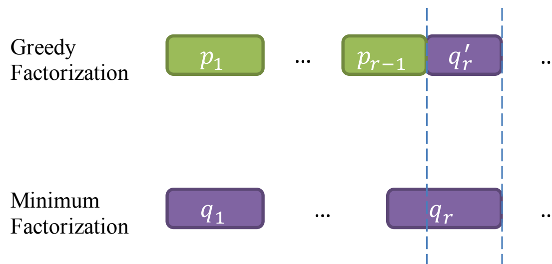


Figure 4.3: Towards proving Lemma 4.5

Then we prove $k = m$ by contradiction. Since we know $k \geq m$, let us assume $k > m$. We already know that $\sum_{1 \leq i \leq m} |p_i| \geq \sum_{1 \leq j \leq m} |q_j| = |s|$. Since $m < k$, we have $\sum_{1 \leq i \leq m} |p_i| < \sum_{1 \leq i \leq k} |p_i| = |s|$, which is a contradiction. Therefore, $k = m$. □

The following example uses the greedy matching algorithm with a complete set of frequent string patterns as the dictionary.

Example 4.6. We use the input string $s = abacd$ again. But in this example, we augment the CandDict in Example 4.5 such that it mimics a complete set of frequent string patterns. We do so by adding all substrings of each $p \in CandDict$ into $CandDict$. After the augmentation, we have $CandDict = \langle a, b, c, d, ab, bac, ba, ac \rangle$ (two new phrases ba and ac are added, which are substrings of bac).

It is easy to see that the greedy matching algorithm produces $\langle ab, ac, d \rangle$ as the factorization, and $\langle ab, ac, d \rangle$ as the output dictionary $Dict$. Using the local lookahead algorithm demonstrated in Example 4.4 with $CandDict$ still yields $\langle a, bac, d \rangle$ as the factorization. Local lookahead does not reduce the number of factors, and does not increase the average length of matches.

Using the dynamic programming algorithm achieves either of these factorizations, depending on how a tie is broken.

The frequent string patterns used as a dictionary does not have to be from the input column $\mathcal{C}$. For efficient and effective compression, we use the frequent string patterns from a sample of $\mathcal{C}$, based on the assumption that the repetitiveness of $\mathcal{C}$ can be captured in the form of the string patterns from a sample. Our experiments show that using a sampling rate of 0.5% for a 4MB block of real world or synthetic columnar data yields satisfactory compression, outperforming Snappy [13], a LZ77-based compression program widely used in the industry.

We discussed earlier that, in our method, local lookahead based on match lengths may not improve compression ratio over greedy matching. However, it is possible for local lookahead with alternative measures to effectively improve compression ratio. In particular, if such a measure facilitates selection of a smaller $Dict$ (e.g. prefers dictionary phrases that are already in $Dict$ over others), then compression may be improved by shorter bit length of

each integer code due to less phrases in $Dict$, since each code is represented by $\lceil \log_2(|Dict|) \rceil$ bits. This is a possible future direction to trade compression speed for compression ratio.

Is a set of string patterns a suitable heuristic dictionary for columnar data? In our experiments, we compare string patterns with another phrased selection method Re-Pair [31], which is a grammar-based method. It is shown that given the same sampling settings and encoding format, phrase selection using string patterns achieves similar compression ratio as Re-Pair. The advantage of string patterns is that a pruning heuristic based on support can be naturally incorporated into pattern mining, which enables trading a small amount of compression ratio for compression speed. This pruning method is described in the next section.

4.4 Pruning Trie

In Section 4.1, we illustrated columnar string pattern mining using a variation of PrefixSpan, where the output string patterns are stored in a prefix tree. A prefix tree can be implemented as an uncompressed trie, where the children of a tree node are indexed by an array of length $|\Sigma|$, Σ being the alphabet. Using an uncompressed trie for greedy matching is very fast, except when the number of the patterns is large and there are many long patterns. In such cases, trie search suffers from frequent cache misses and deteriorating performance. Experiments in Chapter 5 show that for two synthetic datasets, compression speed is negatively affected if an uncompressed trie is used. One dataset contains user agent strings, and the other one contains lorem ipsum strings (fake text). Both datasets contain long patterns. Figure 4.4 shows an uncompressed trie containing 3 strings *abc*, *abd*, and *bcd* and all their substrings, where many array entries on the lower levels of the trie are left empty. The wastage of space in a trie leads to poor CPU cache performance for searching long string patterns.

A possible remedy is to use a compact trie, introduced in Section 2.5. This trie variation uses path compression to mitigate cache inefficiencies. However, experiments show that using a compact trie leads to faster compression in some cases, and slower in others. The problem is that additional CPU cycles are required when traversing a compact trie. It needs at least a check for whether the current node represents a collapsed path of single descendant nodes. In our experiments, about 500 to 1500 trie nodes are created to index the frequent string patterns from a sample (0.5% of 4MB data). The trie is large enough to cause cache inefficiencies, but not so large that using a compact trie is consistently faster. Instead of using existing variations of trie, we design a pruning heuristic to produce a smaller trie to speed up compression. The pruning heuristic is described as follows.

After pruning, the trie has no more than 3 levels below the root. The first 2 levels are exactly the same as the original trie. On the third level, each node n may contain a suffix. This suffix is selected from the descendants of n in the original trie. To select a single suffix

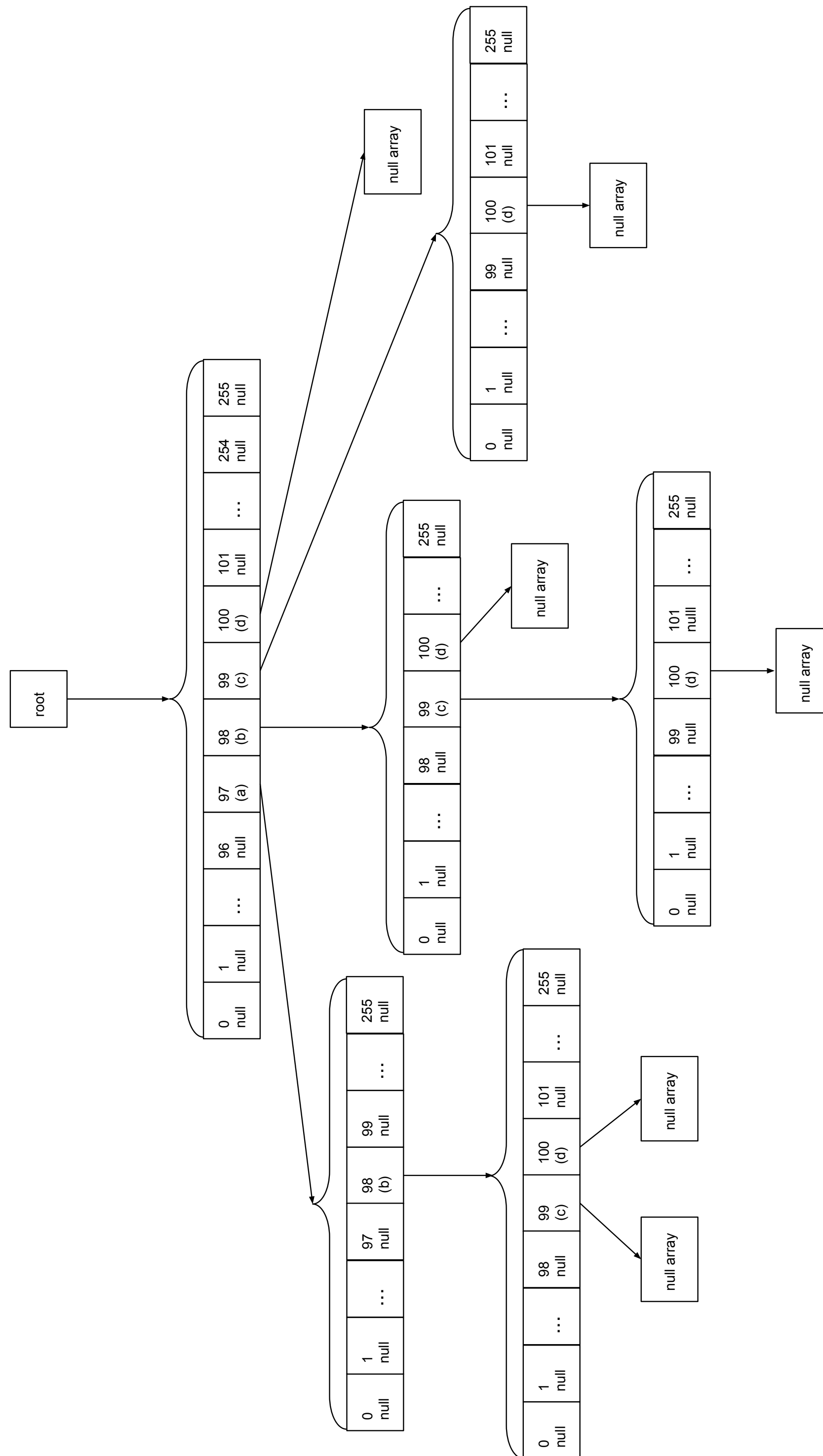


Figure 4.4: Uncompressed trie containing *abc*, *abd*, and *bcd* and all their substrings. A null array refers to an array of length 256 containing only null pointers.

among the descendants, a best-first heuristic is used. On each level below the third, only the most frequent extension is selected.

Example 4.7. Figure 4.5 illustrates an example of the pruning heuristic. Assume the frequent string pattern *abcd* has the highest support among all frequent length-4 patterns prefixed by *abc*. Then, all such patterns are pruned except *abcd*. Assume *abcdf* has the highest support among all length-5 patterns prefixed by *abcd*. Then, all such patterns are pruned except *abcdf*. Assume *abcdf* has no frequent extensions, then the pruning below the node *abc* stops. There is only a single suffix *df* left below the node *abc*. Insead of using 2 extra nodes for the suffix *df*, we can store a string *df* in the node *abc* to further improve cache locality (this is the *lazy expansion* technique of a compact trie, which is path compression with the collapsed path inside a leaf).

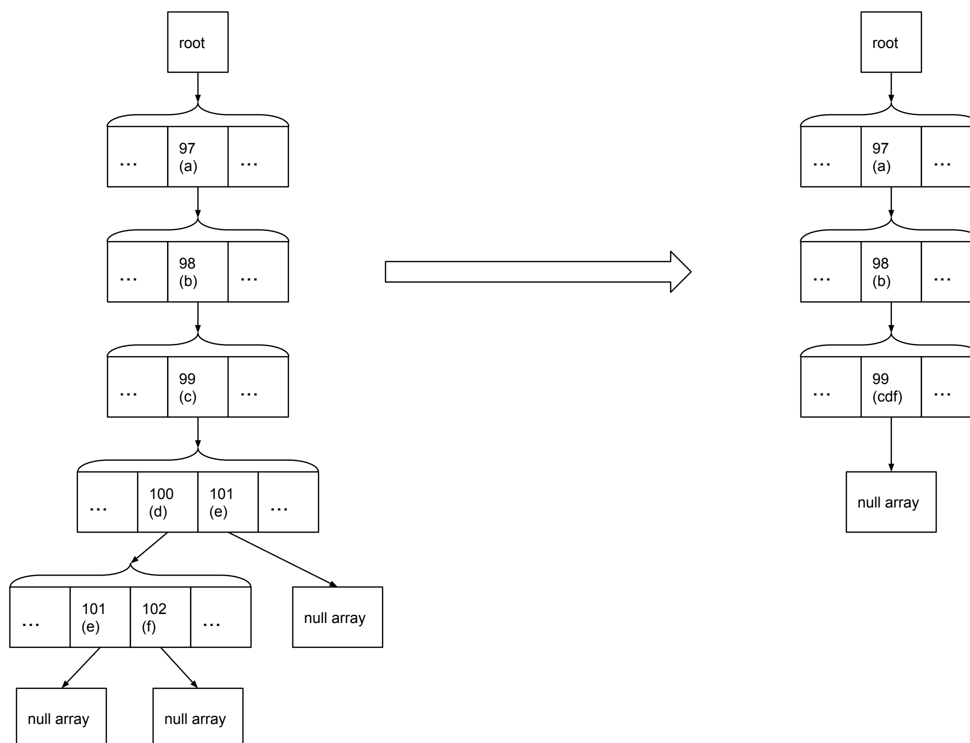


Figure 4.5: An example of pruning trie to improve CPU cache performance in compression. In the top 3 levels below the root, no pruning is performed. Among the children of a 3rd level node, only the node with the highest support is kept. The pruning is performed recursively until a leaf is reached. E.g., *abcd* is the most frequent children of *abc*, so all the siblings of *abcd* are pruned. Again, *abcdf* is the most frequent children of *abcd*, so all the siblings of *abcdf* are pruned. Since only a single suffix *df* is left below *abc*, we remove the lower levels altogether and keep the suffix *df* in the node *abc*.

The pruning may result in increased compression ratio since less phrases are available for compression, but it tries to keep the more frequent phrases with the assumption that they

are more beneficial for compression. We maintain a fixed number of untouched levels to simplify the operations involved in trie search as much as possible. For example, if a leaf is on the second level, there is no need to test whether it contains a suffix, since only the third level nodes may contain suffixes. The number 3 is chosen because it strikes a reasonable balance of compression speed and ratio in our experiments. Note that with pruning, it is no longer guaranteed that for a phrase $p \in CandDict$ all suffixes of p are also in $CandDict$. Among the substrings of p, only those of length 3 or less are guaranteed to be in $CandDict$. Therefore, the greedy matching algorithm may no longer produce a minimum factorization with $CandDict$.

An interesting comparison can be made between the pruned trie and the sliding window dictionary in Snappy [13]. Snappy indexes each 4-byte substring of the window, and if there is a conflict, the right-most substring (which is closest to the current input position) is indexed. The assumption is that a closer match is likely to be a longer match. While for the pruned trie, each 3-byte frequent pattern from a sample indexes a single dictionary phrase. Conflicts are resolved by a best-first heuristic, and the assumption is that, on average, more frequent phrases may produce better matches. This assumption works well on columnar data, as demonstrated in the experiments.

Algorithm 4 uses an alternative recursive function to replace the DfsMinePatterns function in Algorithm 1, to incorporate the heuristic pruning into pattern mining.

4.5 Summary

In this chapter, we introduced our compression method which consists of two phases. First, we described how to mine frequent patterns as dictionary phrases from a sample. Second, we introduced the greedy matching method for identifying input phrases for replacement by dictionary indexes. We provided a partial justification of this method, as apposed to other more complicated factorization methods. Finally, we described a pruning method to improve search performance of the dictionary trie. A final note is that we do not further compress the dictionary stored in the compressed data, although it is possible to do so, since there may be repetitions in the dictionary. Further compressing the dictionary may not yield large compression benefits, since we observe that the size of the dictionary is usually less than 1% of the entire compressed block. We keep the method simple for the current prototyping, and leave this kind of detailed improvements for future work.

Algorithm 4 Mining String Patterns with Heuristic Pruning

```
1: function DfsMinePatternsWithPruning(node, level)
2:     if level = 3 then
3:         SelectSuffix(node)
4:         return
5:     end if
6:     GroupPositionsByChildren(node)
7:     for all non-null child in children of node do
8:         if l(child.Positions) ≥ minSupport then
9:             DfsMinePatternsWithPruning(child, level + 1)
10:        else
11:            child ← a null pointer
12:        end if
13:    end for
14: end function
15:
16: function SelectSuffix(node)
17:    GroupPositionsByChildren(node)
18:    c* ← the symbol c with maximum l(node[c].Positions)
19:    if l(node[c*].Positions) ≥ minSupport then
20:        node.pattern ← node[c*].pattern
21:        node.Positions ← node[c*].Positions
22:        SelectSuffix(node)
23:    end if
24:    all children of node ← null pointers
25: end function
```

Chapter 5

Experiments

In this chapter, we evaluate our method in terms of compression ratio, compression speed, and decompression speed, on both real-world and synthetic columnar data. First, we give an overall comparison among our method, GZIP [16], and Snappy [13] on all the available datasets. Second, we evaluate our method under different sampling and minimum support settings on a single dataset. Two other sampling-based methods are included for comparison.

We implement our algorithms using C++. The experiments are conducted on a MacBook Pro with 8 GB RAM, and Intel(R) Core(TM) i5-4278U CPU, 2.60GHz, with 3 MB L3 cache size. The programs are compiled with Apple Clang in Xcode 7.3.1. The code is publicly available [1].

5.1 Datasets

We use both synthetic and real-world datasets in our experiments. The synthetic datasets are generated by the Python Faker package [2]. The package generates synthetic data such as names, addresses, and dates. Table 5.1 gives examples of the synthetic datasets that we use. We generate 5GB data for each category. For the real-world datasets, we have *fine_foods* [3] [34] and *wiki-links* [4] [39]. The former consists of Amazon reviews of fine foods, amounting to 248.5 MB in size. The latter consists of web page urls whose contents contain links to the English Wikipedia, amounting to 694.9 MB. Table 5.2 gives examples of these two datasets. Table 5.3 shows some statistics of the datasets, where $\overline{|\mathcal{C}|}$ is the average number of strings in a 4MB column block, and $\overline{|s|}$ is the average string length in a dataset.

[1] https://github.com/superxiao/FrequentPatternCompressor
[2] https://github.com/joke2k/faker
[3] http://snap.stanford.edu/data/web-FineFoods.html
[4] https://code.google.com/archive/p/wiki-links/

Synthetic Datasets	**Example**
name	Liza Kemmer
address	284 Reichert Canyon Suite 070 Port Aron, FL 54862
phone_number	(257)123-0907
email	lurline.schiller@kris.com
iso8601	2004-09-30T21:13:29
credit_card_number	3337689104646870
credit_card_full	JCB 15 digit Alex Jewess 180098847840358 09/24 CVC: 464
uri	http://schuppehintz.com/main.html
user_agent	Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.01; Trident/4.0)
sha1	0e96cbfc74b05421f9ad54933c8a638ea7c60b77
text	Sint velit iste laudantium blanditiis. Fugiat quos nostrum esse iste.

Table 5.1: Examples of synthetic datasets

Real-world Datasets	**Example**
fine_foods	This product is great. Gives you so much energy and...
wiki-links	http://www.nytimes.com/2009/07/20/arts/20funny.html

Table 5.2: Examples of real-world datasets

	name	address	phone_number	email	iso8601	credit_card_number
$\overline{\lvert \mathcal{C} \rvert}$	241,165	86,657	241,397	169,021	199,728	242,827
$\overline{\lvert s \rvert}$	15.39	46.40	15.38	22.82	19.00	15.27

	credit_card_full	uri	user_agent	sha1	text	fine_foods	wiki-links
$\overline{\lvert \mathcal{C} \rvert}$	67,853	114,174	45,990	99,864	28,151	9,571	64,712
$\overline{\lvert s \rvert}$	59.81	34.74	89.20	40.00	147.00	436.23	62.82

Table 5.3: Dataset statistics

5.2 Comparisons to GZIP and Snappy

In this section, we will give a set of experiments evaluating the compression efficiency and effectiveness of our method, compared to two baselines, GZIP [16] (from boost 1.6.0) and Snappy [13] (version 1.1.3). These two programs are widely used in industry for textual compression in databases. For each dataset, we compress every 4 MB of data as a block to simulate real-world column-oriented storage where data is often stored in blocks [1]. For each dataset, we repeatedly compress it until a total volume of 10 GB (2,560 blocks) is compressed and then decompressed. For example, we have 5 GB data for the *email* dataset, so we compress the dataset 2 times to reach the 10 GB volume. We exclude disk I/O time

from the compression and decompression time, so that we can evaluate the computing efficiency of the various algorithms more accurately.

For our method, two versions are experimented with. The first version is referred to as *frequent*. This version uses an uncompressed trie to index string patterns for textual replacement during compression, as described in Chapter 4. The second version, referred to as *frequent(pruned)*, uses the pruning scheme described in Section 4.4 to reduce the number of patterns and improve CPU cache performance in trie search. When compressing a 4 MB block, both methods derive string patterns from a sample of the block. Two parameters need to be set. The first one is the number of strings to include in a sample (or the sampling rate). The second one is the minimum relative support (or the minimum support) for pattern mining. As will be shown in the next section, these parameters affect the trade-off between compression speed and ratio. For the experiments in this section, we use the following simple rules. We use a sampling rate of 0.5% for both versions. For example, if a 4 MB block of the *fine_food* dataset contains 9,571 reviews, then a sample contains $\lceil 0.5\% \times 9,571 \rceil = 48$ reviews. We use a minimum relative support of 3% or a minimum support of 4, whichever is the higher, for mining frequent string patterns. For example, for a sample of 48 *fine_food* reviews, we set $minSupport = \max(3\% \times 48, 4) = 4$. The minimum support threshold 4 is to avoid deriving too many patterns, because a large number of patterns may significantly slow down the compression. Figures 5.1, 5.2, 5.4 show the comparisons in compression ratio (see page 17), compression speed, and decompression speed of the two versions of our method and the two baselines. For each dataset, the three values are computed for every block, and the average and the standard deviation over all the 2,560 blocks are displayed. Figure 5.3 shows the average number of candidate dictionary phrases mined from a sample, along with the standard deviation. Figure 5.5 shows the memory usage of each method during compression.

Overall, our methods obtain good compression ratios, which are significantly lower (better) than Snappy on most datasets, and are close to GZIP on a few datasets. Speed-wise, our methods compress faster than Snappy on some datasets, and slower on others. For decompression, our methods are faster than Snappy on most datasets (note that decompression speed is measured in terms of decompressed bytes produced per second). While GZIP has overall the best compression ratio, it is very slow compared to our methods and Snappy.

One may wonder why the *frequent* method can be even faster than Snappy on certain datasets. We provide several observations here. First, we observe that the running time of pattern mining in these experiments is usually less than 10% of the overall running time of compression, due to sampling. The compression time is dominated mainly by matching the input with the dictionary trie and compressing the integer codes. For certain datasets, such as *phone_number*, the dictionary trie is small and very fast for matching. Second, we use an integer compression library using SIMD instructions [33], which accounts for 20-30%

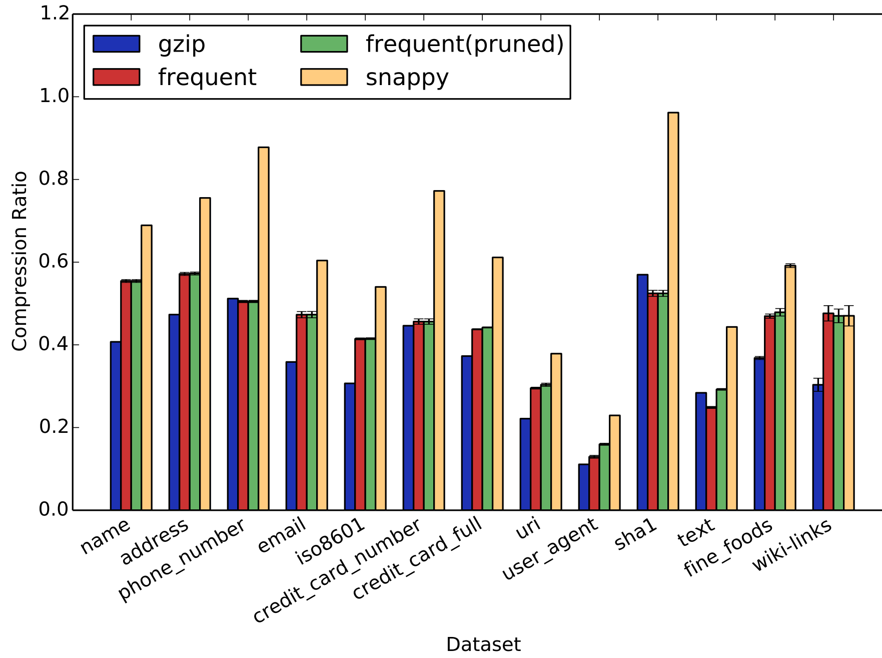


Figure 5.1: Compression ratio of our methods compared to GZIP and Snappy

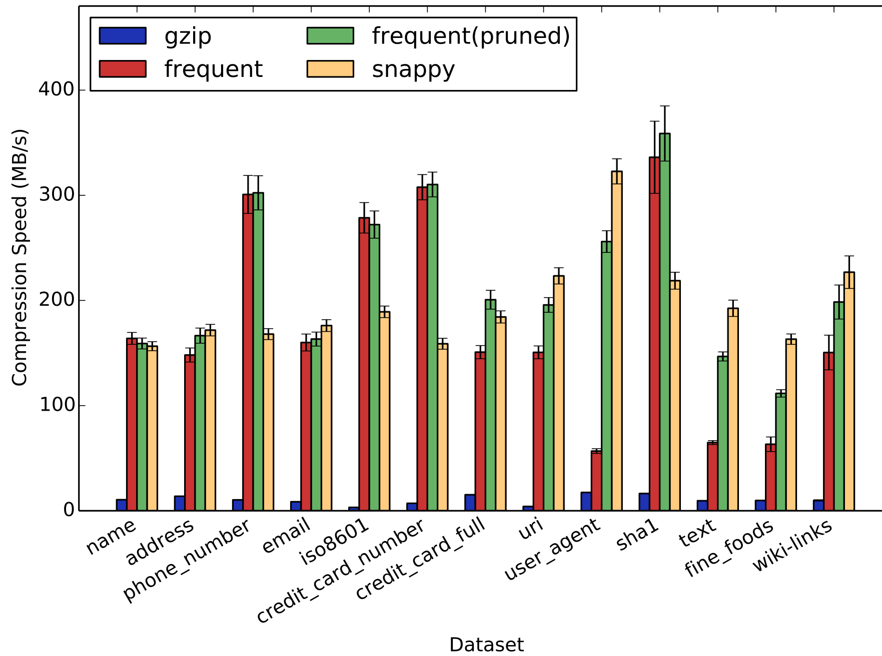


Figure 5.2: Compression speed of our methods compared to GZIP and Snappy

improvements in compression speed, on top of our own bit packing implementation. Snappy currently does not use such vectorized code. Third, our method is semi-static while Snappy is adaptive. More specifically, Snappy adaptively maintains a lookup table for finding the

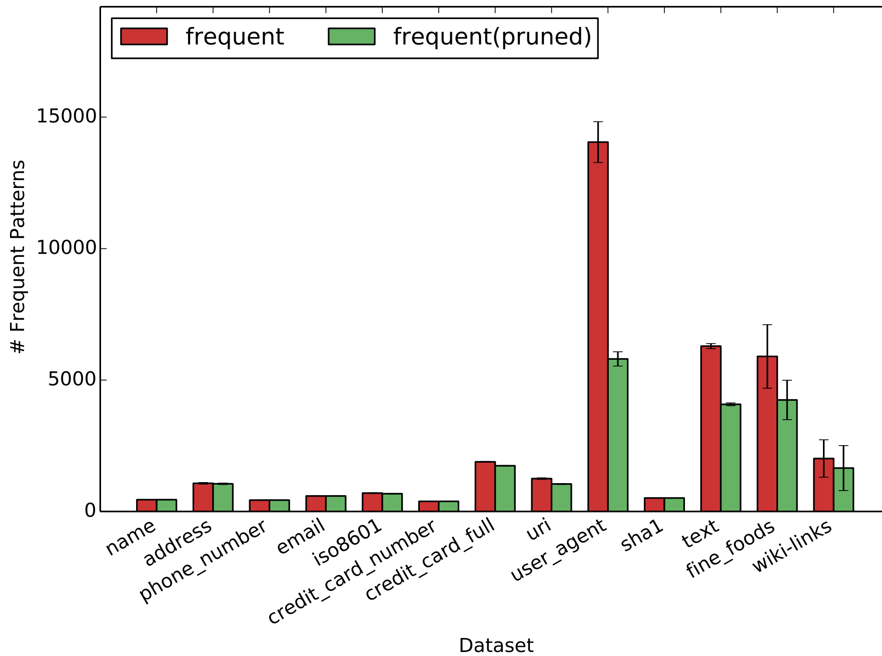


Figure 5.3: Average number of candidate dictionary phrases from a sample

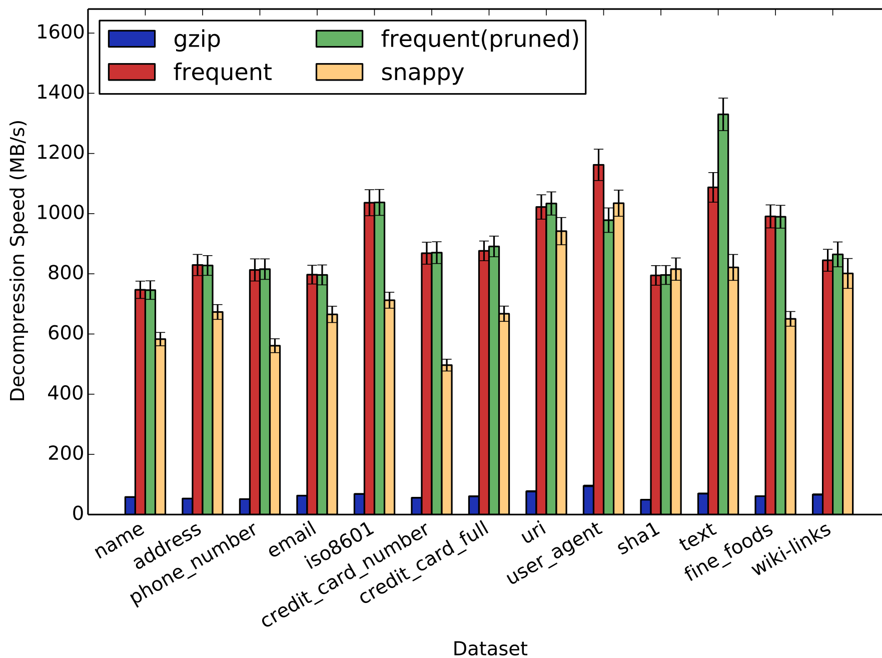


Figure 5.4: Decompression speed of our methods compared to GZIP and Snappy

positions of matches in the sliding window. Our method does not pay the cost of maintaining an adaptive dictionary, because the dictionary trie does not change once it is constructed from the sample.

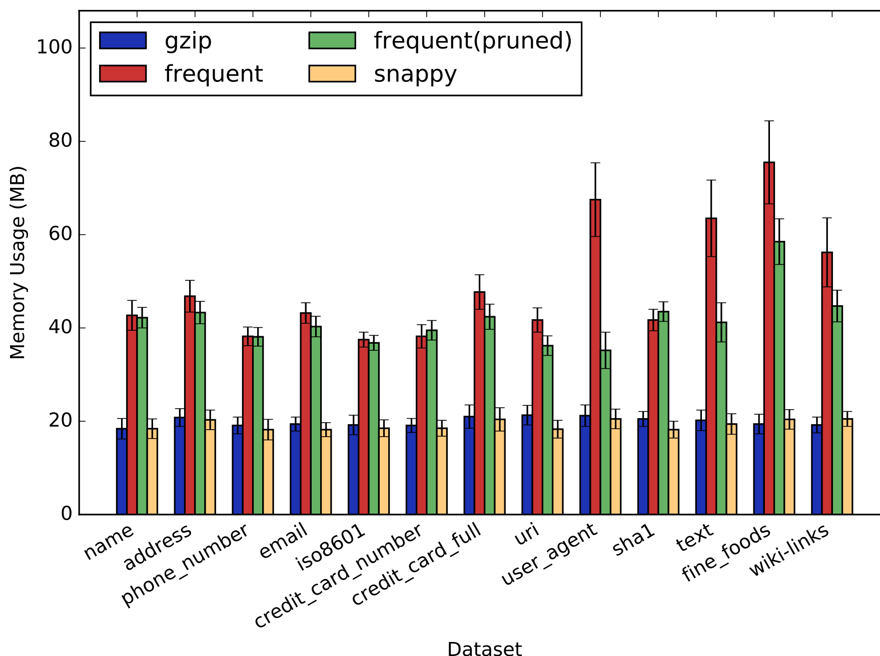


Figure 5.5: Memory usage compared to GZIP and Snappy

For the two versions of our method, the compression ratio of *frequent(pruned)* is slightly worse than *frequent* on some datasets. However, on the datasets *user-agent*, *text*, *fine_foods* and *wiki-links*, *frequent(pruned)* compresses much faster than *frequent*. Figure 5.3 shows that, for these 4 datasets, the numbers of frequent string patterns are large. This causes severe cache inefficiencies and explains why *frequent* is slow. Meanwhile, the pruning heuristic removes a large number of dictionary phrases on these datasets, speeding up the compression to a level comparable to Snappy, without affecting the compression ratio too much. A question is, can we simply increase the minimum (relative) support to decrease the number of string patterns to achieve the same purpose? This is not as effective as the pruning heuristic in preserving the compression ratio, as will be demonstrated in the next section.

Both versions of our method have similar decompression speed. This is because decompression is a relatively simple process without involving a trie. We only need to read the dictionary from a compressed block, and index it as an array of strings for textual replacement in decompression. In terms of decompression speed, our method is even faster than Snappy on most datasets. This shows that the encoding method we use is suitable for read-only analytical tasks in column stores.

Another observation is that, compared to Snappy, our method is particularly effective on datasets containing random sequences of symbols, such as phone numbers and dates. This is because such datasets do not contain long repeats of strings. They are compressible mainly because they have a small alphabet. Therefore they are not suitable for the byte-

oriented LZ77 algorithms such as Snappy. However, these datasets can be effectively and efficiently compressed by our method because we pack integer codes in their bit lengths. In addition, we use an integer compression library using SIMD instructions [33], further speeding up the packing and unpacking of integer codes.

For the *iso8601* dataset, it should be noted that date time values can be stored as integers instead of strings in databases. For example, unix time can be stored in a single 32-bit integer [48]. This would give us around 21% comrpession ratio, much better than all the different algorithms shown in Figure 5.1, including ours. With more prior knowledge of the data, a more specific algorithm can compress the data better.

For the *wiki-links* dataset, our method does not compress better than Snappy. This is because this dataset contains consecutive urls from the same domain, and such local repetitions can be more effectively captured by the LZ77 algorithms. Since our method is based on sampling a large number of strings, the local repetitions may not be captured effectively. We also observe that *frequent(pruned)* has a slightly better compression ratio than *frequent* on *wiki-links*, unlike other datasets. This shows that having more patterns in the dictionary does not necessarily translates to better compression. These observations demonstrate that different kinds of data may be compressed more or less effectively by our method, depending on the characteristics of the data.

Figure 5.5 shows the memory usage of our experiment program. These measurements include the memory usage of the compression algorithms, as well as holding the input blocks in memory and other experiment-related cost. Our methods have a memory usage ranging from 40 MB to 80 MB, significantly more than Snappy and GZIP. *Frequent(pruned)* generally has a smaller memory footprint than *frequent*, resulting from a pruned trie. It is possible to further improve memory usage on top of our current prototype implementation.

5.3 Comparisons to Local Lookahead and Re-Pair Under Different Parameters

In this section, we evaluate our method under different parameters settings. There are two parameters to set. The first one is the sampling rate, and the second one is the minimum support. Note that we experiment with the minimum support instead of the minimum relative support in the following experiments, for clarity reasons. We still use a 4 MB block size for all experiments, and compress a 10 GB volume under each parameter setting.

GZIP and Snappy are again included as baselines, for which two horizontal lines are shown in each figure, since they are not relevant to the sampling rate or minimum support. Again, the two versions of our method, *frequent* and *frequent(pruned)*, are evaluated in these experiments. Besides, there are 2 other sampling-based competitors. The first one, referred to as *frequent(local lookahead)*, is the same as *frequent*, except that it uses local lookahead instead of greedy matching for the factorization of the input data (both local

lookahead and greedy matching were discussed in Section 4.3). This method is included so that we can evaluate whether a simpler and faster greedy matching algorithm would have worse compression ratio than a local lookahead method. For this purpose, we implemented a more exhaustive version of the local lookahead scheme [12], which uses a search window of the same size as the left match (see the discussion on local lookahead in Section 4.3 for more details). The second method we experimented against, referred to as *re-pair*, is a sampling-based method using the Re-Pair method [31] to select phrases. As introduced in Section 2.3, Re-Pair is a grammar-based method which repeatedly selects the most frequent symbol-pair that appears in the data, and then replaces the pair with an unused symbol. This method was shown to have very good compression ratio, although the compression is slow. The original Re-Pair method does not use sampling, and comes with its own coding scheme with entropy codes [31]. We use only the phrase selection part of Re-Pair together with our sampling settings and encoding format. We include *re-pair* in the experiments to evaluate the compression performance of phrase selection using frequent string patterns, as apposed to a more complicated phrase selection method. We implemented *re-pair* using the discussed method in the original paper [31].

The dataset *fine_foods* is chosen for the following experiments since it represents a common kind of string data, English text. Most of the findings on *find_food* is representative of the results we see on other datasets. There are a few exceptions, which we will point out in later discussion. Figures 5.6 to 5.8 respectively show the compression ratio, compression speed and decompression speed of the various methods under different sampling rates. Figure 5.9 shows the number of trie nodes in the sampling based methods (this is the number of dictionary phrases for *frequent* and *frequent(pruned)*, but not *re-pair*). Figure 5.10 shows the percentages of CPU cycles spent on serving L3 cache misses during compression. This metric is recommended by Intel for measuring the performance impact on L3 cache [40]. The minimum support is set to 5 in these experiments. Figures 5.11 to 5.15 show the results under different minimum support settings. The sampling rate is set to 0.5% in these experiments.

We have the following observations from these results.

1. A higher sampling rate leads to slower compression speed (Figure 5.7) and better compression ratio (Figure 5.6) for all the sampling-based methods. This shows that the repetitiveness of columnar data can be captured more effectively with a larger sample. At the same time, larger sample size increases the cost of phrase selection, and also increases the number of phrases selected (Figure 5.9). This leads to worse cache performance during the textual replacement phase (Figure 5.10). This also implies that our method may benefit from a larger CPU cache size, e.g. from a server machine. If fast compression is required, then a sampling rate under 0.5% seems to give a good trade-off between compression speed and ratio.

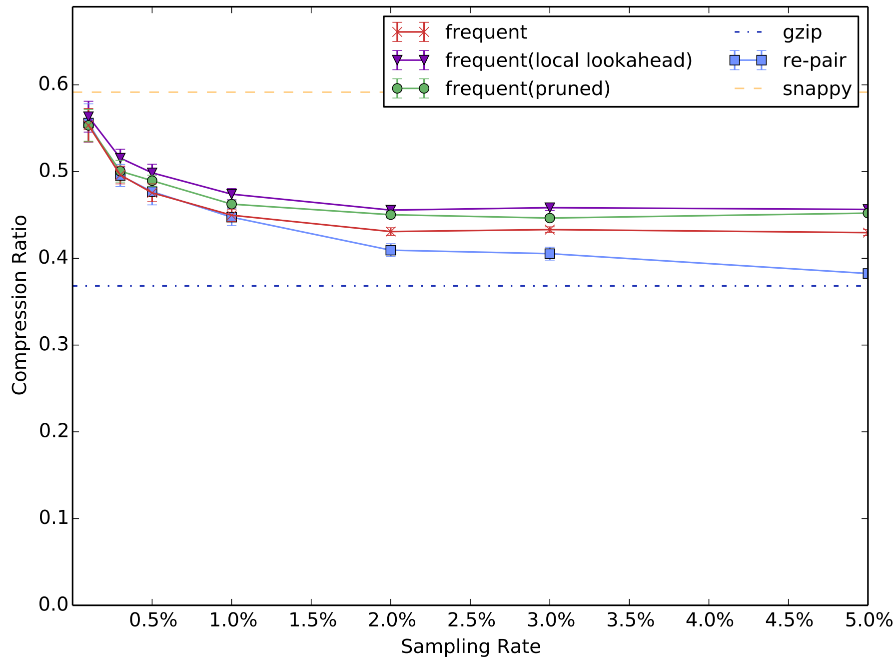


Figure 5.6: Comparison of compression ratio for *fine_foods* with $minSupport = 5$

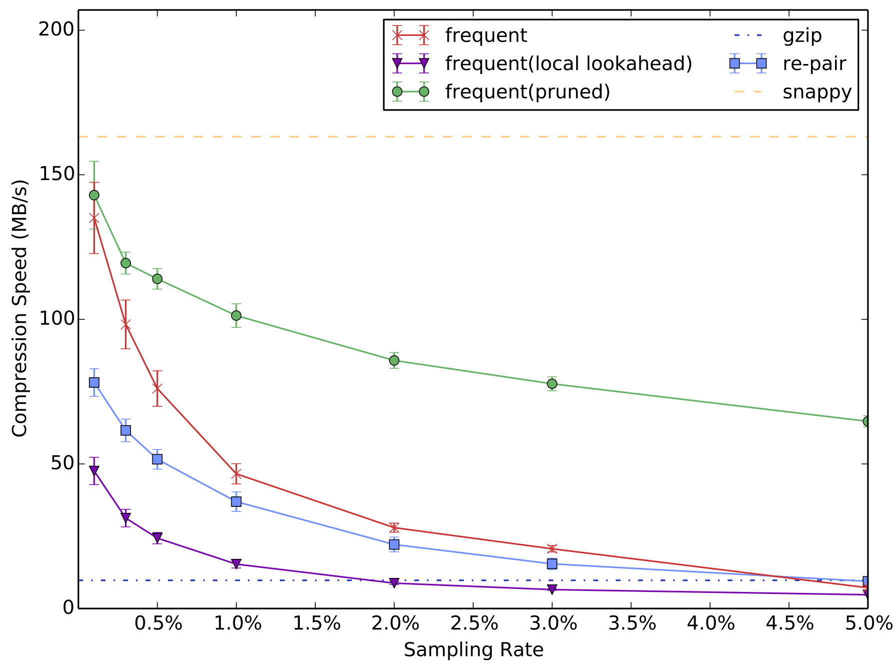


Figure 5.7: Comparison of compression speed for *fine_foods* with $minSupport = 5$

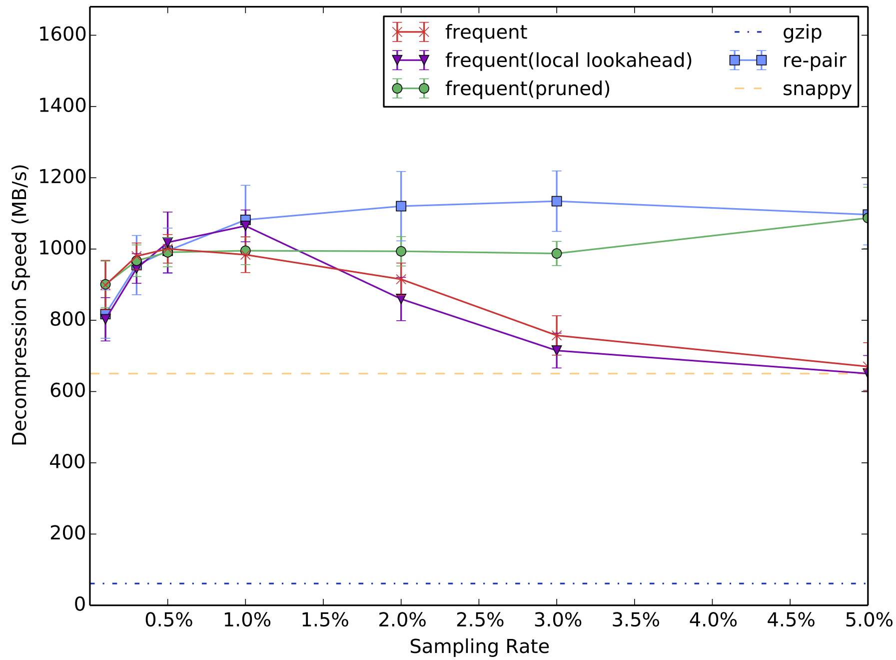


Figure 5.8: Comparison of decompression speed for *fine_foods* with $minSupport = 5$

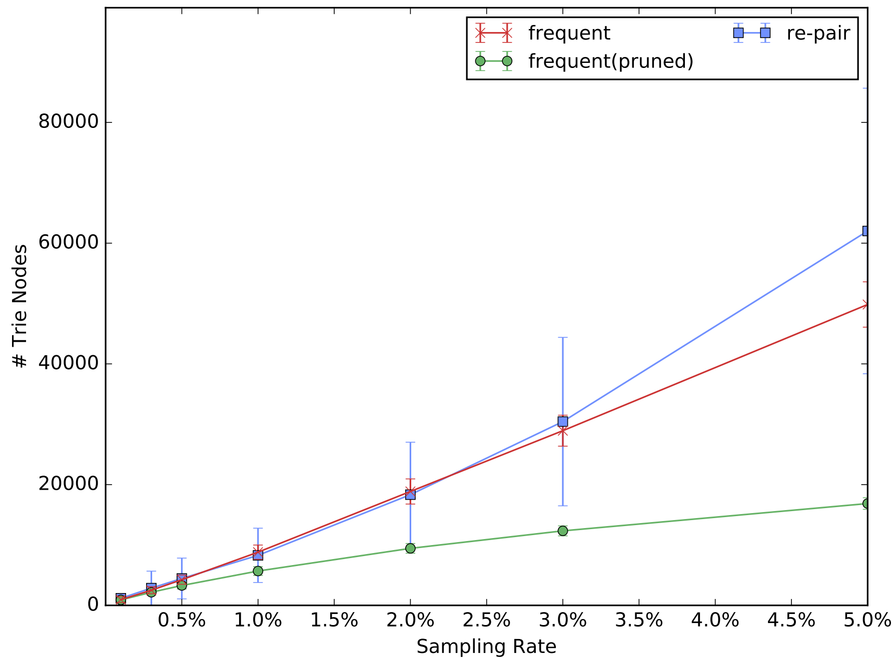


Figure 5.9: Comparison of number of trie nodes in compression, with $minSupport = 5$. This is the same as the number of dictionary phrases for *frequent* and *frequent(pruned)*, since every node represents a frequent string pattern, and therefore a dictionary phrase. This does not hold for *re-pair*.

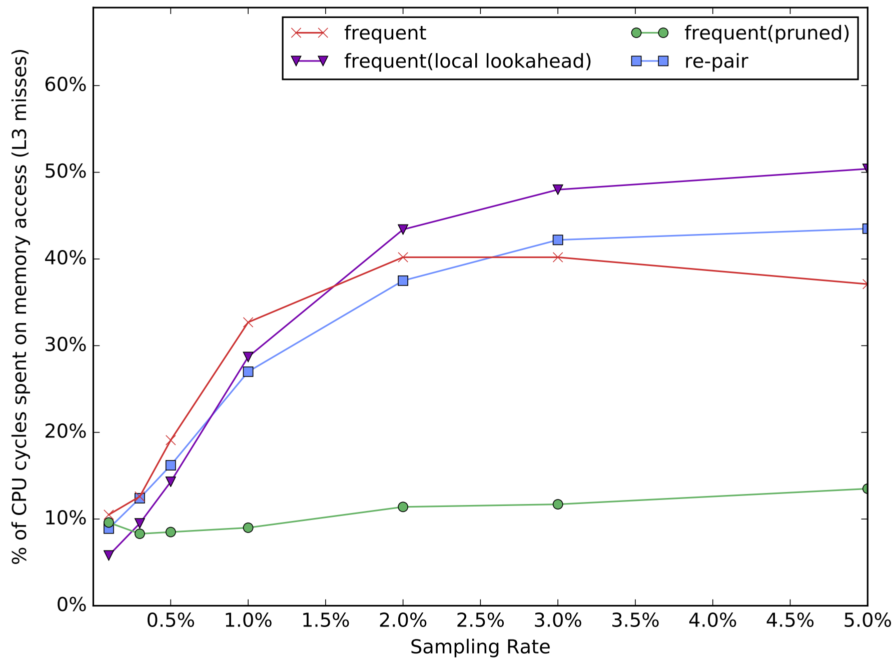


Figure 5.10: Comparison of percentages of CPU cycles spent on memory access (L3 cache misses) in compression, with $minSupport = 5$

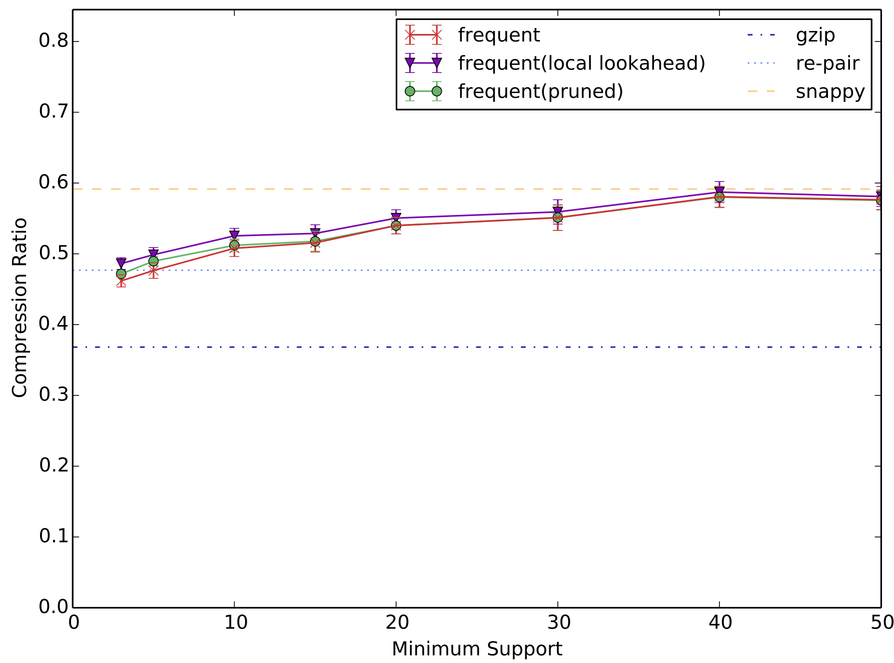


Figure 5.11: Comparison of compression ratio for *fine_foods* with sampling rate 0.5%

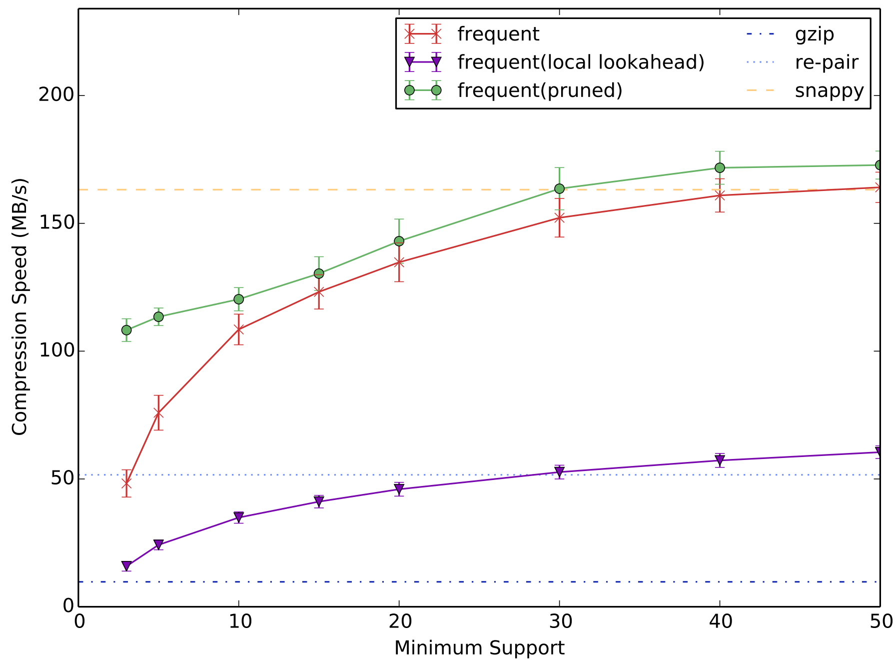


Figure 5.12: Comparison of compression speed for *fine_foods* with sampling rate 0.5%

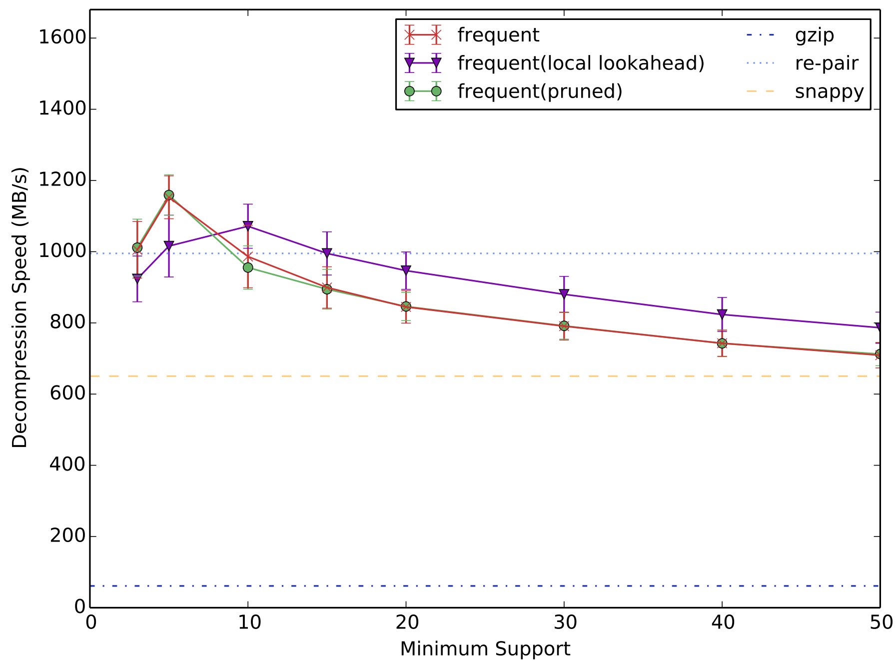


Figure 5.13: Comparison of decompression speed for *fine_foods* with sampling rate 0.5%

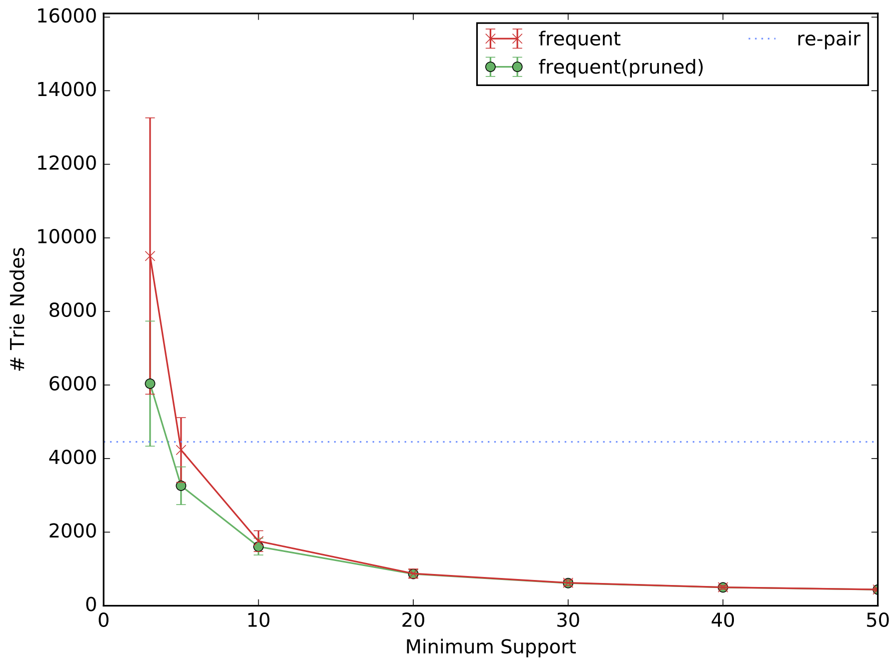


Figure 5.14: Comparison of number of trie nodes in compression, with sampling rate 0.5%

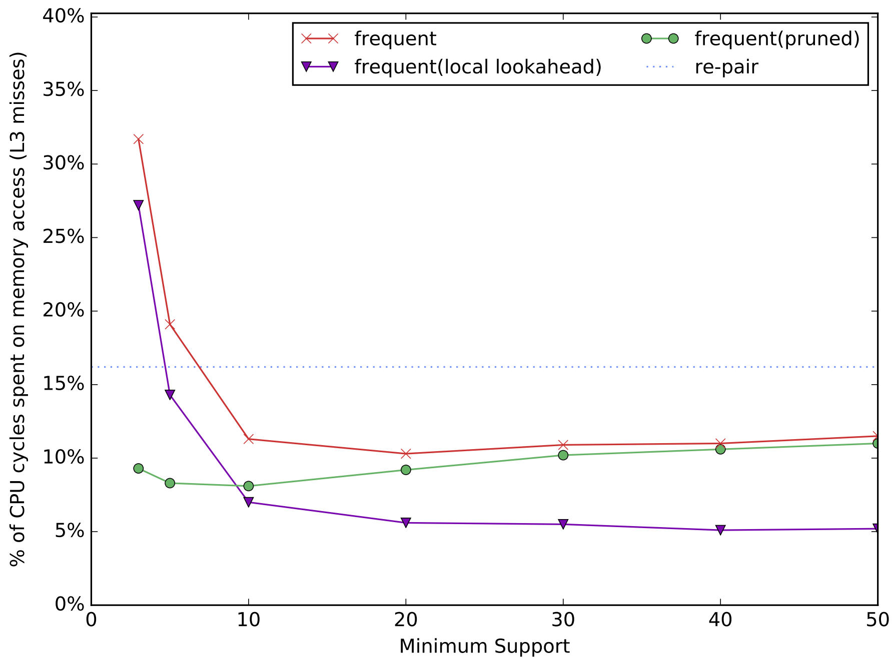


Figure 5.15: Comparison of percentages of CPU cycles spent on memory access (L3 cache misses) in compression, with sampling rate 0.5%

2. Among the sampling-based methods, *frequent(pruned)* is overall the fastest in terms of compression. It obtains an impressive speed-up on top of *frequent*, especially in high sampling rates (Figure 5.7) or low *minSupport* settings (Figure 5.12). In these settings, *frequent* tends to produce a larger number of dictionary phrases (Figure 5.9 and 5.14), which results in worse cache performance (Figure 5.10 and 5.15). *Frequent(pruned)* restricts the number of dictionary phrases, so the compression speed is faster. The speed-up comes at a cost of compression ratio, which is in an acceptable range (Figure 5.6 and Figure 5.11).

3. Among the sampling-based methods, the best compression ratio is obtained by *re-pair* (Figure 5.6). Such a grammar-based method is shown to exploit the compressibility of the data more effectively than frequent string patterns. However, on the *fine_foods* dataset, this advantage only comes under a large sampling rate and slow compression speed. Below a sampling rate of 1%, *re-pair* does not compress better than *frequent*. In addition, *re-pair* compresses significantly slower than both *frequent* and *frequent(pruned)* (Figure 5.7). One reason is that phrase selection using Re-Pair can be more costly than string pattern mining, which was observed in our profiling. Another reason is that greedy matching for *re-pair* is more complicated. For example, if the Re-Pair method finds a dictionary phrase $p = abacd$, then the prefix $abac$ is not necessarily also a dictionary phrase (unlike frequent string patterns). Let us assume that only the prefix ab is found as a phrase. When a match ab is found during factorization, we must keep looking for a possible match $abacd$. If we find a match $abac$ but not $abacd$, we still have to use ab as a factor, at a cost of extra lookahead. The *frequent* and *frequent(pruned)* methods do not have such a cost. We also experimented with another version of *re-pair* where every prefix of a phrase can be used for factorization. This method is faster but has worse compression ratio than *re-pair*. In the comparison for another dataset *user_agent* (figures are not shown for brevity), *re-pair* not only is faster than *frequent*, but also compresses better, although *frequent(pruned)* is still the fastest. This demonstrates that such grammar-based methods may be more suitable for the types of data containing long repetitions. In short, Re-Pair or other similar methods can be good options for sampling-based compression, but some careful engineering is needed to speed up compression.

4. Increasing the minimum support has the effect of speeding up compression (Figure 5.12), at a cost of worse compression ratio (Figure 5.11), because it reduces the length of dictionary phrases, and reduces the number of phrases in the dictionary. By comparing *frequent(pruned)* with $minSupport = 3$ and *frequent* with $minSupport = 5$, we can see that the former not only has slightly better compression ratio (Figure 5.11), but also is significantly faster (Figure 5.12). This shows that the pruning heuristic can be a more effective strategy for trading a small amount of compression ratio

for compression speed, compared to raising minimum support. It is even more so for datasets containing long repetitions, such as *user_agent*, where the speed-up achieved by *frequent(pruned)* is much more significant (Figure 5.2).

5. *Frequent(local lookahead)* not only compresses much slower than *frequent* (Figure 5.7 and 5.12), but also has worse compression ratio (Figure 5.6 and 5.11). This confirms our speculation in Section 4.3 that the faster and simpler greedy matching algorithm is sufficient to obtain good compression ratio with frequent string patterns as the dictionary.

6. For decompression speed, Figure 5.8 shows that as the sampling rate increases over 1%, *frequent* decompresses slower, while *frequent(pruned)* does not. We presume that this is because, for *frequent*, a higher sampling rate leads to more dictionary phrases being stored in a compressed block. This leads to worse cache performance for seeking and copying the phrases in decompression. For *frequent(pruned)*, since the pruning restricts rapid growth of dictionary phrases as the sampling rate increases, decompression speed does not decrease too much. We also observe in Figure 5.13, as the minimum support increases, the decompression speed first increases, then decreases. We presume that the initial speed-up is due to a smaller dictionary and improved cache performance in decompression. The following slow-down may be because the dictionary phrases (frequent patterns) become shorter, and the number of integer codes (factors) becomes larger, resulting in higher cost in decompressing the integer codes and replacing them with the phrases.

7. It should be noted that Figure 5.10 and 5.15 only show the performance impact on L3 cache, not the entire cache hierarchy. For example, as the sampling rate increases, *frequent(pruned)* becomes significant slower, although the performance impact on L3 cache does not change much. One explanation is that the cost of pattern mining is higher. Another one is that the performance impact on L1 and L2 caches, not included in the experiments, may become worse.

Finally, the following Figure 5.16 shows the memory usage of our methods under different sampling rates, with the dataset *fine_foods*. It can be seen that memory usage of *frequent* increases almost linearly up to a sampling rate of 5%, while the memory usage of *frequent(pruned)* increases much slower under large sampling rates. This is consistent to Figure 5.9 which shows the numbers of trie nodes under different sampling rates.

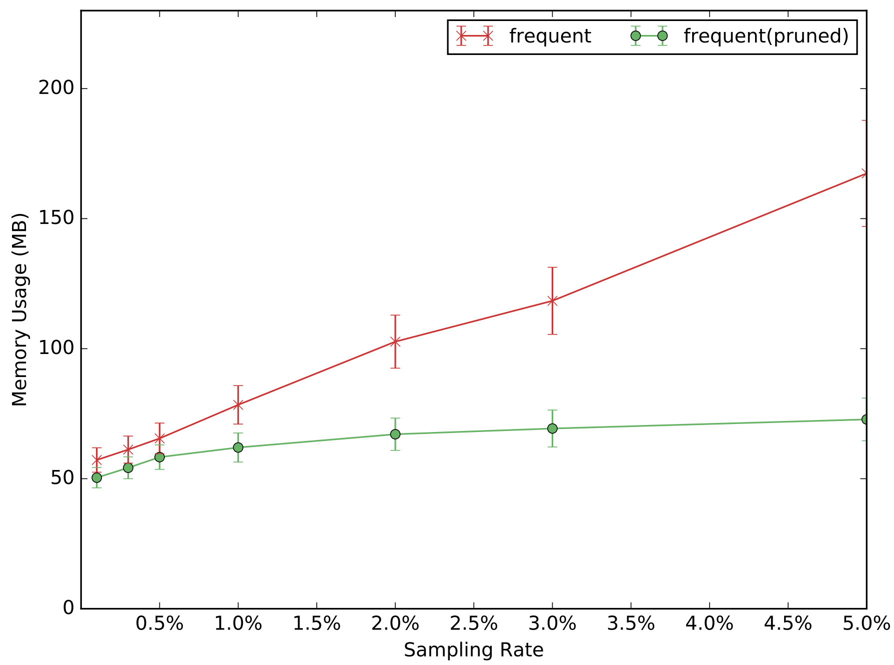


Figure 5.16: Comparison of memory usage with $minSupport = 5$

Chapter 6

Conclusions

Compression is important for the performance of analytical tasks in column stores. We observe that existing compression methods used in column stores, such as dictionary compression and run length encoding [2], may not effectively compress certain kinds of string data such as street addresses, log messages, etc. Another popular option is to use LZ77-based methods such as GZIP [16] and Snappy [13], but these methods do not exploit the columnar nature of the data. We develop a sampling-based method, which derives frequent string patterns from a sample of an input column, and uses these patterns as the dictionary phrases for compression. We use the PrefixSpan algorithm [27] to find frequent string patterns. We discussed several methods for partitioning an input column into a sequence of dictionary phrases that can be replaced by dictionary indexes (i.e. methods for factorization). We inferred that, given a complete set of frequent string patterns as the dictionary, the simple and fast greedy matching algorithm may be sufficient for obtaining good compression ratio, as apposed to a more complicated local lookahead method [12]. This is confirmed on the datasets used in our experiments. In addition, we develop a simple pruning heuristic to reduce the number of dictionary phrases (string patterns), in order to improve CPU cache performance in compression. The pruning heuristic is experimentally shown to significantly speed up compression with a reasonable loss in compression ratio. Overall, our method is shown to have significantly better compression ratio than Snappy on many types of columnar data, with comparable compression and decompression speed.

For future work, the following directions can be considered.

- *Developing a better pruning method, or choosing alternative data structures for efficient and effective compression.* Currently, we use frequent string patterns with a simple pruning heuristic to achieve fast compression speed and reasonable compression ratio. A better pruning method may potentially improve cache performance, speed up compression, and yield better compression ratio. A possible direction is to prune on the transitions on possible word breaks, such as transitions between different character classes (alphabetic, numeric, and punctuation marks, etc). We have also evaluated

other pattern mining methods such as GoKrimp, which mines patterns by how well a pattern compresses the input data [29]. This method turns out to be less effective in terms of compression ratio than simple frequent patterns in our compression settings. A possible explanation is that GoKrimp may not work well with the greedy matching method which we use. However, there are other possible alternatives worth further evaluation, such as closed sequential patterns [46] or sequence generators [18]. Finally, we can evaluate data structure choices other than trie, such as suffix array.

- *Experimenting with LZ77-like factorization with a dictionary constructed with frequent string patterns.* Currently, we use a sequence of phrases as the dictionary, and use the dictionary indexes as codes. An alternative is to construct the dictionary as a block of data from a sample of the input, and use two numbers, an offset and a length, to refer to a dictionary phrase in this block. This is similar to LZ77, and can be considered as a combination of sampling and traditional fast compression algorithms, such as Snappy. The idea has been explored for large string collections such as web collections [24] and genome sequences [45, 37, 28, 14]. We can test this idea for collections of shorter strings, such as the kinds of columnar data targeted by our method. Especially, we can construct the dictionary block with a pattern mining based approach.

- *Automatic selection of sampling rate and minimum support.* In our experiments, we used some simplistic rules for selecting these two parameters in Section 5.2. It is desirable to have a more stable method for selecting the parameters that yield a more consistent trade-off between compression ratio and speed.

- *Extending our method for NoSQL, such as JSON document stores.* Since JSON documents can be longer than the strings we experimented with, it may be beneficial to first apply a certain kind of vertical partitioning on the documents, and use each partition as a column input for our method. Since each partition may have its unique set of repetitions, it is possible to achieve better compression ratio and speed than compressing the documents as a single column.

- *Extending our method in streaming settings.* If we expect each block of a columnar data stream to be similar, then the dictionary phrased mined from an earlier block can be used to compress later blocks, saving the running time spent on repeated pattern mining.

Bibliography

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[3] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475. IEEE, 2007.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[5] Gennady Antoshenkov. Dictionary-based order-preserving string compression. *The VLDB Journal*, 6(1):26–39, February 1997.

[6] Nikolas Askitis and Ranjan Sinha. Hat-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, pages 97–105, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[7] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, et al. Business analytics in (a) blink. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, page 9, 2012.

[8] Timothy Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, December 1989.

[9] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 283–296, New York, NY, USA, 2009. ACM.

[10] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[11] Adam Cannane and Hugh E. Williams. General-purpose compression for efficient retrieval. *Journal of the American Society for Information Science and Technology*, 52(5):430–437, 2001.

[12] Adam Cannane and Hugh E. Williams. A general-purpose compression scheme for large collections. *ACM Transactions on Information Systems*, 20(3):329–355, jul 2002.

[13] Jeff Dean, Ghemawat Sanjay, and Steinar H. Gunderson. google/snappy: A fast compressor/decompressor. `https://github.com/google/snappy`.

[14] Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, November 2011.

[15] L Peter Deutsch. Deflate compressed data format specification version 1.3, 1996.

[16] L Peter Deutsch. Gzip file format specification version 4.3, 1996.

[17] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database–an architecture overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, page 28, 2012.

[18] Chuancong Gao, Jianyong Wang, Yukai He, and Lizhu Zhou. Efficient mining of frequent sequence generators. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1051–1052, New York, NY, USA, 2008. ACM.

[19] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[20] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[21] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20:192–223, 2002.

[23] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 389–400, New York, NY, USA, 2007. ACM.

[24] Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment*, 5(3):265–273, November 2011.

[25] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[26] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[27] Jian Pei, Jiawei Han, B. Mortazavi-Asl, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 215–, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th International Conference on String Processing and Information Retrieval*, SPIRE'10, pages 201–206, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] Hoang Thanh Lam, Fabian Mörchen, Dmitriy Fradkin, and Toon Calders. Mining compressing sequential patterns. *Statistical Analysis and Data Mining*, 7(1):34–52, 2014.

[30] Per-Ake Larson, Eric N. Hanson, and Susan L. Price. Columnar storage in sql server 2012. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, page 15, 2012.

[31] Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the Conference on Data Compression*, DCC '99, pages 296–, Washington, DC, USA, 1999. IEEE Computer Society.

[32] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49. IEEE Computer Society, 2013.

[33] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[34] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[35] Donald R. Morrison. Patricia–practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[36] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[37] Armando J. Pinho, Diogo Pratas, and Sara P. Garcia. Green: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, 2011.

[38] J. Rissanen and G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27(1):12–23, Jan 1981.

[39] Sameer Singh, Fernando Pereira, and Andrew Mccallum. Wikilinks: A large-scale cross-document coreference corpus labeled via links to wikipedia, 2012.

[40] Intel Software and Services Group. Using intel vtune to tune software on the 4th generation, 2013. [Online; accessed 29-June-2016].

[41] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[42] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[43] Nikolaj Tatti and Jilles Vreeken. The long and the short of it: Summarising event sequences with serial episodes. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 462–470, New York, NY, USA, 2012. ACM.

[44] Sebastian Wandelt, Marc Bux, and Ulf Leser. Trends in genome compression. *Current Bioinformatics*, 9(3):315–326, 2014.

[45] Sebastian Wandelt and Ulf Leser. Fresco: Referential compression of highly similar sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(5):1275–1288, Sept 2013.

[46] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, page 79, Washington, DC, USA, 2004. IEEE Computer Society.

[47] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[48] Wikipedia. Unix time — Wikipedia, the free encyclopedia, 2016. [Online; accessed 23-June-2016].

[49] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.

[50] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[51] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sep 1978.

[52] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 59, Washington, DC, USA, 2006. IEEE Computer Society.