Performance evaluation of fast integer compression techniques over tables

by

Ikhtear Md. Sharif Bhuyan

Bachelor of Computer Science and Engineering, Khulna University , 2006

A Dissertation Submitted in Partial Fulfilment of the Requirements for the Degree of

Master of Computer Science

in the Graduate Academic Unit of Computer Science

Supervisors:	Hazel Webb, Ph.D. Computer Science Daniel Lemire, Ph.D. Engineering Mathematics Owen Kaser, Ph.D. Computer Science
Examining Board:	Christopher Baker, Ph.D. Computer Science, Chair Weichang Du, Ph.D. Computer Science
External Examiner:	Tim Alderson, Ph.D. Mathematics

This dissertation is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

November 2013

Abstract

Compression is used in database management systems to improve the performance by preserving memory and storage. Compression and decompression requires substantial processing by the Central Processing Unit (CPU). Queries in large databases such as Online Analytical Processing (OLAP) databases involve processing huge amounts of data. Database compression not only reduces disk space requirements, but also increases the effective Input/Output (I/O) bandwidth since more data is transferred in compressed form to cache for query processing. As a result, database compression transforms I/O-intensive database operations into more CPU-intensive operations.

We are most likely to benefit from compression if encoding and decoding speeds significantly exceed I/O bandwidth. Hence, we seek compression schemes that can be used in databases with good compression ratios and high speed.

We examined the performance of several compression schemes such as Variable-Byte, Binary Packing/Frame Of Reference (FOR), Simple9 and Simple16 which have reasonable compression ratio with fair decompression speed over sequences of integers. As variations on Binary Packing, we also studied patched schemes such as NewPFD, OptPFD and FastPFOR: they have good compression ratios and decompression speed though they need more computational time during compression than Binary Packing.

In our study, we aim to quantify the trade-offs of fast integer compression schemes with respect to compression ratio and speed of compression and decompression. We are able to decompress data at a rate of around 1.5 billion of integers per second (in Java) while

sometimes beating Shannon's entropy. In our tests, Binary Packing is significantly faster than all other alternatives. Among the patched schemes we tested, the recently introduced FastPFOR is most competitive. Hence, we found that it is worth using a patching scheme because we get both a good compression ratio and a high decompression speed. However, the higher compression and decompression speed of Binary Packing makes it a better choice when speed is more important than compression ratio.

We also assessed the effects that row ordering and sorting have on compression performance. We discovered that sorting can significantly improve the performance of compression. We obtained around 15% gain in decompression speed and 12% gain in compression ratio by sorting compared to random shuffling. Additionally, we found that sorting on the highest cardinality column was more effective than sorting on lower cardinality columns.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many applications, such as search engines and relational database systems, deal with large amounts of data and these data are often stored in the form of arrays of integers. Some data warehouses and some statistical database systems have indefinite retention and total database size will only grow [26]. As the amount of data to be handled by large databases increases, their performance becomes constrained by the speed at which data can be read or written. As a result, queries in large databases often generate a huge amount of Input/Output (I/O) since these queries may access a big chunk of data to retrieve the desired information. Processing in the CPU cache is faster than in main memory or disk [7], but the CPU cache is smaller in size and more expensive than other types of storage. As a result, we can not fit all data in the CPU cache during query processing. We have to access disk to retrieve data from large databases and copy those data into main memory. Therefore, main memory access is a performance bottleneck for many database operations [7]. Compression reduces the physical size of the database, saving storage cost and reducing the transfer time of data to and from primary and secondary memory [2]. Keeping data in a compressed format can preserve memory and allow the transfer of data to and from memory in a shorter time with lower processing cost and enhance I/O bandwidth [16]. I/O time is a vital factor for query processing in large databases. Compression can also enhance

index structures like B-trees and R-trees which improves the overall performance [14].

Applications like relational database systems deal with a huge amount of data that needs to be organized as tuples. Tables are the basic building blocks for relational database systems. Each table is made up of rows and columns. Every tuple has different attributes and is stored into a row. Tables in database have been traditionally stored row-wise. Tuples are stored together on disk. Recently, column-wise has become a popular alternative storage model. In a column store database, the values in each column of a table are stored sequentially in disk. In our study, we focus on the performance of compression over tables. There are some factors which influence compression performance like the order of data, data distribution and cardinality of data [14, 14, 23, 48]. Data distribution also influences the performance of compressed databases [23].

Our long-term goal is to assess the performance of compression and other factors such as row order to enhance the overall performance of database management systems. For the purpose of this thesis, we focus on in-memory compression applied to sequences of integers. Database tables are mapped to such sequences of integers.

1.1 Light-weight compression

On the one hand, the performance of many applications like relational database systems is affected by access to slower storage devices like disk or tape. On the other hand, a modern CPU is much faster than the disk, tape or main memory. Poor storage performance becomes a limiting factor for many applications such as information retrieval systems [19]. Compression in a database will allow storing more data in cache during query processing, which can result in faster operation [48]. Compression schemes must have both high compression ratio and decompression speed to enhance the performance of databases. Also, lossless compression is needed for most database operations. Indeed, the compression scheme should permit recovery of the original data from its compressed form and it

should not hamper the regular operations of the database system [26]. To achieve this, light-weight compression methods are used [37]. These methods aim to compress while minimizing CPU usage. In contrast, a scheme like arithmetic coding has better compression ratios but it has slow compression and decompression speed [45].

Some of the most common light-weight compression methods over arrays of integers include Binary Packing/Frame Of Reference (FOR) [4, 14], Delta coding [14, 25, 48], Variable-Byte Coding [8, 33, 43], Simple9 [47], and Simple16 [46]. These coding techniques are often used in inverted index compression for search engines as well as in database engines. Recently, patched versions of FOR and Delta coding known as PFOR and PForDelta [47] have been used for inverted list and database compression. Patched compression schemes use the super-scalar facility of a modern CPU [46]. As a result, these techniques provide aggressive compression and decompression speed as well as a modest compression ratio. Our goal is to make an experimental comparison among these techniques with respect to compression and decompression speed in a database setting.

Note that we use Binary Packing and FOR as equivalent terms. In the literature(See Section 3.1.8), Binary Packing and FOR refer to similar ideas and different authors called FOR different names.

1.2 Contribution

Incorporating compression in relational database systems or information retrieval systems is not a new idea. There are several compression algorithms that have been devised in recent years to compress database tables. High speed schemes are known to boost the performance of relational database systems [1, 9, 37] and information retrieval systems [3, 30, 33, 46].

Our approach is to compress tables on a column basis. Generally, we can get higher compression ratios in column stores because consecutive entries in a column are often

quite similar to each other [1, 24, 34]

Tables in relational database systems contain data of different types such as integer, string etc. We restrict our discussion to integer compression since a simple dictionary encoding can be used to map strings to integers in case of relational database systems [5, 10, 17, 24, 28]. Several compression algorithms are available to compress integer values. We are particularly interested in recently introduced new techniques based on "patching" [48]. These techniques have become a standard among integer coding techniques. However, we do not know of any study that compared them in a database setting with other competitive alternatives. The main objective of this study is to quantify the trade-offs of the patched schemes like NewPFD, OptPFD, FastPFOR in a database setting with respect to compression speed, compression size and decompression speed. We wish to arrive at an analysis comparing patched schemes with other schemes. We limit our study to compressed column values. The objectives of this study include:

- Examining and comparing the performance of patched schemes with other methods with respect to compression ratio, decompression speed and compression speed.
- Assessing the effect of different factors such as row order.

1.3 Organization

In the next chapter, we discuss important concepts necessary for understanding database operations. We present characteristics of compression as well as the techniques that can be used in a database context. We also present the comparison characteristics and a statistic to evaluate the compression techniques. This statistical characteristic helps us to determine the compressibility of any compression scheme.

In Chapter 3, we explore several integer compression schemes considering different factors as mentioned in our first objective (See Section 1.2) to get an optimized result. We describe different types of compression techniques based on the nature of their input and

output. We also discuss the functionalities, the work flow and pros and cons of different compression techniques.

Chapter 4 describes our experimental setup, our hardware configuration, the nature of the hardware and some experimental preliminaries. We also describe basic information on our implementation and fast integer compression schemes that we use in our study.

In Chapter 5, we discuss experiments on synthetic datasets that validate some of our theoretical results. The experiments on these synthetic datasets will provide us with a reference. We ran several identical trials varying the number of integers of different distributions (Clustered and Uniform) to measure the performance of different compression schemes with respect to compressed size, compression speed and decompression speed.

Our work with experiments based on real data is described in Chapter 6. In order to benchmark the performance of patched schemes over other schemes, we used datasets based on real data that are freely available. We identify the characteristics of each dataset and store the results. We present our overall experimental results. In the analysis, we measure the performance of the compression schemes on integers. This allows us to determine the trade-off between compression ratio and decompression speed among different schemes.

Finally, we summarize our study, review our conclusions and discuss possible future work in Chapter 7. This study focuses on the performance of different fast integer compression techniques, but it is not sufficient for a decision on when to use a specific compression scheme in database setting. To measure the workload performance, one could use these compression algorithms and evaluate the performance of query processing inside a database engine. Nevertheless, our work identifies some of the most promising techniques.

Chapter 2

Preliminaries

Databases must cope with large datasets, and many queries involve parsing large amounts of data [26]. To retrieve any information, these queries need to read data from disk and copy those to memory and then to the cache. As a result, the time spent to complete these data-intensive queries mainly depends on the time required for input/output operations to be completed.

To alleviate these problems, databases need to keep more of the data in RAM and cache. Unfortunately, RAM is expensive and often scarce. Hence, it is natural to attempt to compress the data in RAM. However, the data decoding speed can then become a bottleneck.

In this chapter, we review a few factors that influence database operation. These include: memory access for database operations, the basic prerequisites for database compression, the basic differences between data compression in general and database compression in particular, comparison characteristics of compression algorithms and the characteristics of modern processors.

2.1 Memory access

Memory access is very important in the case of large databases. Usually, data are stored in flat file form or record-oriented form in a sequential manner. Those files are stored on

disk. Usually when we are trying to retrieve any data we need to access those files from disk, so the speed of the disk can be important. There will be a lot of disk I/O operations to access a large file or large amount of data that cannot fit in RAM. Unfortunately, modern disk speed is very slow compared to CPU speed [41].

The memory hierarchy of computer storage can be categorized depending on response time. The major storage levels are

- Processor Cache
- Main memory
- Disk

Each stage of memory has a different latency. The cache is the fastest to process data and the disk is the slowest. In the worst case, to answer a query, we need to access the disk and move data to memory and finally to cache. As a result, the CPU spends most of its time in waiting to complete disk I/O to get data from disk. However, the CPU cache is not large compared to the disk. So, we cannot fit a large amount of data in the cache in one CPU cycle. Compression can help us to fit more data in the cache and RAM so that any database operation can be done in a shorter time compared to dealing with uncompressed data. In order to show the query results to the end user we need to decompress the compressed form, so decompression speed is important for database operations.

In practice, main memory is also slower than the CPU. It will take around 30–100 CPU cycles to send a memory word (i.e. 4 bytes for 32-bit CPU) from main memory to CPU [9]. To alleviate this problem we can employ a caching technique which is to store the most recently used data in the cache. Different caching techniques such as Most Recently Used (MRU) and Least Recently Used (LRU) have been devised and used by many applications for performance enhancement. In the case of LRU, it discards the least recently used items first from the cache. Meanwhile, MRU discards the most recently used items first [38].

2.2 Types of compression

According to the characteristics of compressed data and data recovery from compressed form, compression techniques can be divided into 2 categories. We will discuss those in next sections

2.2.1 Lossy Compression

In lossy compression schemes, some information has been discarded while compressing the original data. The main objective of this approach is to reduce the size by keeping the overall structure of the file and eliminating some detail or redundant or unnecessary information of the original data. Therefore, it generates an approximation of original data in exchange for a smaller size. Lossy compression techniques are generally used in audio, video or image compression [40].

2.2.2 Lossless Compression

Lossless compression allows recovering the original data from its compressed form without any information loss. Lossless compression reduces the size as well as preserving all information of the original file [39]. Lossless compression techniques are often used with text documents.

Often, lossless compression techniques generate a statistical model in an initial phase. Afterwards, the statistical model is used to store the data more efficiently. For example, frequent values might be stored using fewer bits than infrequent values [22].

2.3 Database compression

Database compression is different from generic data compression. In the case of data compression, we are mainly concerned about the compression ratio. However, in the case

of database compression we need to consider the following factors:

- The compression technique has to be lossless so that it allows recovering the original data without any information loss from its compressed form. Lossless compression techniques will preserve data integrity of a database.

- As much as possible, queries should be executed directly on the compressed data, thus bypassing a separate decompression step [16]. So, the compression technique should allow the query processing engine to operate on compressed data.

- We have to take into consideration the compression and decompression speed as well as the compression ratio. The compression and decompression speeds should be related to the I/O bandwidth: compression techniques for data in RAM need to be faster than compression techniques designed for disk.

Not all compression techniques can satisfy all the above requirements. Therefore, we cannot use all types of compression techniques in database settings to achieve optimal benefits.

Most database compression techniques are based on some conventional data compression schemes. We will discuss in detail different compression algorithms in the next chapter.

2.4 Comparison Characteristics

Data compression helps to load and keep more data in RAM. However, we cannot get the benefit of compression if it takes more CPU cycles to decompress the data during query processing. To evaluate a compression scheme, we need to consider compression ratio and low CPU usage.

Comparing the compression schemes is not an easy task. There are many characteristics including compression ratio, compression speed and decompression speed. In order to get

better performance, we need to preserve memory and also minimize processing. So, we evaluate compression schemes mainly by compression ratio and decompression speed.

2.5 Shannon Entropy

In the case of lossless compression algorithms, the compressed form of data has the same information using fewer bits. As a result, every bit has more information which means it has higher entropy. However, the compressed form is unpredictable because we may have random values in any column of the database tables. Shannon entropy offers an upper bound on the compression performance of a compression scheme that compresses each value as if it were independent. To evaluate the performance of any compression scheme with respect to compression ratio, we used the frequentist interpretation of Shannon entropy. We compute Shannon entropy for every column of the datasets. The formula that we use to calculate Shannon entropy is as follows:

$$-\sum_{i} p(y_i) \log_2 p(y_i) \tag{2.1}$$

To calculate the probability $p(y_i)$ of any integer y_i in any column, we take the number of occurrences of that integer y_i in that particular column divided by the total number of elements (including duplicates) in the column. In the case of database tables column values are uncertain. In order to calculate the overall uncertainty, we sum up individual uncertainties. We can achieve the additivity characteristic of individual uncertainty using the logarithm. We can measure the efficiency of any compression scheme by identifying how close we are to the entropy value.

2.6 Benefits from modern CPU

There has been a radical improvement of CPU speed in the last two decades. Previously, CPUs executed only one machine instruction at a time. After completion of the current instruction, CPUs fetched the next instruction from memory or cache. As a result, CPUs spend a lot of idle time since there are several sub-units such as the instruction decoder, the integer arithmetic unit, Floating Point (FP) arithmetic unit, etc. involved in executing any instruction. During the execution of any instruction only one sub-unit is busy at a time.

To mitigate the problem, pipelining has been introduced. The basic concept of pipelining is to decode or fetch the next instruction when the earlier instruction has been moved to an execution sub-unit. In the case of pipelining, execution of any instruction is divided into several stages and in every clock cycle all the instructions move forward by one stage towards the execution stage. Thus, we could theoretically increase the instruction throughput without increasing the CPU clock. This only works flawlessly when there are no conditional jumps in the code. Pipelines may face delays in execution of any instruction in two ways:

- Data hazard: It happens when any instruction needs the result of a previous instruction as input. In this case, the second instruction has to wait for completion of the first instruction.

- Control hazard: This happens when there is an *if-then-else* statement in the program. In this case the CPU cannot decide ahead of time which instruction it needs to execute next.

Superscalar CPUs with multiple execution subunits are able to do the same thing in parallel [42]. CPUs employ *branch prediction* techniques. This will distribute instructions across fully parallel units of CPUs while ensuring the results are the same as if the instructions had been executed sequentially. In the case of an instruction *if-A-then-B-else-C*,

modern CPUs try to predict the outcome of *A* based on the previous branching behaviour. It starts working on the prediction and after a few stages if it has been found that the prediction is wrong it flushes the pipeline and starts over executing the other option. Some modern CPUs execute both of the instructions (*then* and *else*) at the same time and discard one after having the result of the *if* statement.

To take advantage of modern CPUs, compression schemes can avoid branching during encoding and decoding. In practice, this often means that we regroup the data into blocks: instead of uncompressing one integer at a time, we uncompress blocks of them.

In the next Chapter, we study several compression algorithms and classify them based on the nature of the output of encoding. We also describe the pros and cons of these compression methods.

Chapter 3

Background

Compression is used in relational database systems to enhance the overall performance of database operations. Compression reduces the size of data thereby improving all I/O operations. It reduces the seek times and transfer time from disk to memory or cache [16, 37, 46, 47]. It allows more data to be kept in RAM or cache. As a result it increases the buffer hit rate which means more data can be found in cache for processing. We discussed I/O intensive database operations in Chapter 2. In the case of I/O bound queries, we can get the benefit of compression over CPU overhead of decompression by improving disk I/O time. In our work we are mostly interested in fast techniques well suited for compressing data in RAM.

In this chapter, we review different compression techniques in detail and discuss their pros and cons. We also discuss the effect of data order on compressibility.

3.1 Compression Techniques

There are different granularities of compression like file level, block level, tuple level and field level [37]. Compression can be done on various types of attributes like strings, numbers, dates and times. In our study, we are mainly concerned with compression of column values of database tables so that we can reduce the size of data and enhance the

overall performance of database. In recent years, several compression schemes have been developed which have fast decompression speed with moderate compression ratio. Our goal is to achieve significant benefit from compression in database settings.

We can classify integer compression into two primary types depending on the nature of the output of encoding: schemes that use a variable number of bytes or bits to compress every integer and those that use a fixed number of bytes or bits to compress a variable number of integers.

- **Variable length output:** It takes a fixed input length such as a single integer and processes it into variable length output depending on the input value. This can be classified into three sub-categories:
  - Byte-oriented compression: Integers are coded in units of bytes. A variable number of bytes is used to encode a single integer. Variable-Byte, discussed in Section 3.1.4 is an example of this category.
  - Per-integer bit-oriented compression:This kind of compression schemes uses a variable number of bits to encode a single integer. Golomb-Rice coding, discussed in Section 3.1.5 is an ideal example of this kind of coding. Other variable length bit oriented compression schemes are available such as Elias gamma coding [8, 13, 36], k-gamma [31] and binary interpolative coding [25]. Bit oriented compression schemes result in low decompression speed since bit by bit look up imposes a high processing burden on computer architecture [32].
  - Block-based compression: These schemes use a fixed number of input integers and output a variable number of bytes. Frame Of Reference and PForDelta are examples of this kind of scheme. These schemes are discussed in Sections 3.1.8 and 3.1.9 respectively.
- **Fixed length output:** In this compression scheme, each step takes a variable number of integers and produces a compressed form of those integers using a fixed num-

ber of bits as a unit. The basic strategy is to pack as many integers as possible from input into a fixed length codeword. The number of input sequences can be identified by using a flag which can be placed in front of every codeword. Simple9 (S9) coding and Simple16 (S16) coding are examples discussed in Sections 3.1.6 and 3.1.7 respectively.

We will briefly discuss a few of the compression techniques that we found useful in order to enhance the performance of database systems in the following sections.

3.1.1 Block coding

Data of any table is typically organized in blocks consisting of a few rows [15, 26]. Every block will contain a number of rows [26, 28]. It is also called a database page [12], though some authors might consider a page to be larger than a block. The size of the block varies according to the policy used in the database engine [28, 48]. Some of them fix the size of the block according to the number of rows other use the number of bytes to fix the size. Compression allows us to store more rows in a single block [28]. Block-oriented compression techniques can be used to reduce the disk I/O bottleneck for large databases. Block size may influence the compression since large block size may have negative effects like high memory usage.

3.1.2 Attribute coding

There are various types of attributes in a database such as string, number, time and so on. It would be beneficial to compress all those attribute values irrespective of their occurrence. We replace all attribute values in any column by non-negative integers. Coding can be done in various ways like domain coding, frequency coding or random coding [20, 24].

Name	City	Tax Amount
Robert	Saint John	235
Maria	Saint John	365
Robert	Montreal	400.5
William	Saint John	265.7
Xan	Saint John	400.5
William	Montreal	365
Robert	Toronto	235
Maria	Toronto	365
Robert	Toronto	400.5
Maria	Saint John	400.5

(a)

Name	City	Tax Amount
2	2	1
1	2	3
2	1	4
3	2	2
4	2	4
3	1	3
2	3	1
1	3	3
2	3	4
1	2	4

(b)

Figure 3.1: Example of a domain-coded table, Fig. 3.1a is the original table. Fig. 3.1b is its corresponding domain coded table

Domain coding

In domain coding [24], values of any column are mapped to integer values according to their canonical order (e.g., the alphabetical order for strings). It preserves the natural order of all attribute values. In the case of number values, we map the lowest number to 1 (see Fig. 3.1).

Frequency coding

In the case of frequency coding, values are mapped to integers in decreasing order according to their frequency [20]; i.e., the most frequent value is mapped to 1 (see Fig. 3.2). Frequency coding improves compression ratio more effectively compared to domain or random coding [30]. Domain coding is more beneficial than frequency coding when using range queries [20, 30].

Random coding

In case of random coding, we map attribute values to integers according to their occurrence; i.e., the the attribute value which occurred first will be coded as 1 (See Fig. 3.3).

Name	City	Tax Amount
Robert	Saint John	235
Maria	Saint John	365
Robert	Montreal	400.5
William	Saint John	265.7
Xan	Saint John	400.5
William	Montreal	365
Robert	Toronto	235
Maria	Toronto	365
Robert	Toronto	400.5
Maria	Saint John	400.5

(a)

Name	City	Tax Amount
1	1	3
2	1	2
1	3	1
3	1	4
4	1	1
3	3	2
1	2	3
2	2	2
1	2	1
2	1	1

(b)

Figure 3.2: Example of a frequency-coded table, Fig. 3.2a is the original table. Fig. 3.2b is its corresponding frequency coded table

Random coding does not improve compression substantially compared to frequency coding (See Section 6.3.1).

3.1.3 Difference coding

This coding mechanism encodes a set of integers by deducting successive values. This sort of coding has been used in the field of data compression for years [14, 25, 48]. The main idea is to code the first value as it is and the remaining values will be coded as the difference between successive values. We can achieve a good compression ratio if the differences between successive values are small and we have a technique to compress sequences of small integers.

Delta coding is one of the popular techniques for integer coding. It codes integer values by subtracting successive values using simple arithmetic, $\delta = x_i - x_{i+1}$ (sometimes called deltas). The result can be either positive or negative. If the difference between two consecutive integers is small then we can code the difference with few bits, but one large difference can adversely affect the total compression ratio. Consider a sequence of integers like $24, 32, 43, 25, 25, 55, 77$. According to difference, we will store the first value as

Name	City	Tax Amount
Robert	Saint John	235
Maria	Saint John	365
Robert	Montreal	400.5
William	Saint John	265.7
Xan	Saint John	400.5
William	Montreal	365
Robert	Toronto	235
Maria	Toronto	365
Robert	Toronto	400.5
Maria	Saint John	400.5

(a)

Name	City	Tax Amount
1	1	1
2	1	2
1	2	3
3	1	4
4	1	3
3	2	2
1	3	1
2	3	2
1	3	3
2	1	3

(b)

Figure 3.3: Example of a random-coded table, Fig. 3.3a is the original table. Fig. 3.3b is its corresponding random coded table

it is 24. We will store the differences of successive values rather than storing the original values. As a result, the coded sequence will be $24, 8, 11, -18, 0, 20, 22$. We can revert back to the original sequence by computing the prefix sum $x_i = x_1 + \sum_{k=2}^{i} \delta_i$. In the case of random access to an integer in a certain location will require summing up all the delta values which may make differential coding inefficient.

3.1.4 Variable-Byte coding

An integer can be compressed as a sequence of bytes using Variable-Byte coding. Different authors referred it variously such as v-byte, variablebyte [43], var-byte, vbyte [8], varint,Vint,VB [33] or Escaping [35]. This is a byte oriented coding technique. For every byte, the first 7 bits are used to store part of the binary representation of the integer, and the last bit is used for a status bit which indicates whether the next byte is part of this integer [47]. For example, an integer $k = 312$ and its binary representation is 100111000. Using variable byte coding k can be represented by two bytes: $10000010\ 00111000$. The status bit 0 in the first byte (00111000) indicates that the next byte is also part of the integer. In total, it took 16 bits to encode the number 312. While decoding, we read byte

Number	Divisor	Quotient	Remainder	Code
5	4	1	1	0101
10	4	2	2	00110
15	4	3	3	000111
61	8	7	5	00000001101
64	8	8	0	000000001000

Figure 3.4: Golomb-Rice coding

by byte. We discard the eighth bit if it is 0 and continue reading the next bytes till we get 1 in the eighth bit for any byte to record any integer. Variable-Byte coding is easy to implement and known to be significantly faster than traditional bit oriented methods such as Golomb-Rice [47].

The main drawback of this method is that it will take at least 1 byte to store even a small integer. In the case of encoding an integer of a single bit such as 1, we have to store it using one byte and the byte looks like 10000001. As a result, we waste 7 bits. Thus, it can make Variable-Byte coding inefficient in terms of compression ratio.

3.1.5 Golomb-Rice coding

In Golomb coding, we code an integer (i), by the the quotient (q) and remainder (r) of division by the divisor (d). We write the quotient $\lfloor i/d \rfloor$ in unary notation and the reminder i mod d in binary notation [47]. We need a stop bit after the quotient. We can use 1 as stop bit if the quotient is written as 0 to represent the unary form. In case of Rice coding, we use the divisor as a power of 2. For example if we are coding a number 15 with divisor 4, the code will be 000111 (See Fig. 3.4). Golomb-Rice coding will achieve good compression ratio. However, the decompression speed is quite slow due to the leading unary values as we need to check a single bit at a time during decompression [3, 44, 46]. We are not including this compression scheme in our experiments, since decompression speed is very vital for relational database system to retrieve information.

3.1.6 Simple9 (S9) coding

The basic idea of this coding technique is to pack as many integers as possible within 32 bit words. Simple9 divides each block into 4 status bits and 28 data bits. These 28 bits actually store the binary representation on input blocks. These data bits can be divided in 9 different ways: 28 1-bit numbers, 14 2-bit numbers,7 4-bit numbers, 4 7-bit numbers, 2 14-bit numbers, 1 28-bit number, 9 3-bit numbers(1 bit unused), 5 5-bit numbers(3 bits unused), 3 9-bit numbers(1 bit unused). The 4 status bits of a 32-bit word stores which of the 9 cases is used. During decompression, switch cases will be used on status bits to identify each of the 9 cases and apply a bit-mask to exact the desired bits from 32 bit word. Thus, we can recover the original integers [47].

3.1.7 Simple16 (S16) coding

Simple16 is a modified version of Simple9. Simple9 wastes a few status bits by representing only 9 cases using 4 status bits. However, using 4 bits, 16 cases can be represented. Several cases can be included like 2 4-bit numbers followed by 4 5-bit numbers, 2 5-bit numbers followed by 3 6-bit numbers, etc. This will result in better compression than Simple9 [46]. For example, if all of the next 4 numbers take no more than 7 bits each, then those 4 numbers can be encoded in four 7-bit binary representations. Thus, 28 bits will be occupied and 4 bits will be used to select which combination has been used. Fig. 3.5 shows all the 16 possible combinations to store integers using 28 bits.

For an example, if we have a sequence of integers

$$\{2, 15, 6, 106, 2, 186, 369, 3, 22, 86, 467, 8194\}, \tag{3.1}$$

the numbers of bits required to represent these integers are $\{2, 4, 3, 7, 2, 8, 9, 2, 5, 7, 9, 14\}$. Now assume we want to code in 32-bit words using Simple16. The first block will consist of $\{2, 15, 6, 106\}$ according to the bit combination $\{4 \times 7\}$. The next block $\{2, 186, 369\}$

Serial No	Selector	Bit combination (Numbers $\times$ Length of codes)	Total number coded
1	0000	28×1	28
2	0001	7×2 , 14×1	21
3	0010	7×1 , 7×2 , 7×1	21
4	0011	14×1 , 7×2	21
5	0100	14×2	14
6	0101	1×4 , 8×3	9
7	0110	1×3 , 4×4 , 3×3	8
8	0111	4×5 , 2×4	6
9	1000	2×4 , 4×5	6
10	1001	3×6 , 2×5	5
11	1010	2×5 , 3×6	5
12	1011	4×7	4
13	1011	7×4	7
14	1100	1×10 , 2×9	3
15	1110	2×14	2
16	1111	1×28	1

Figure 3.5: Bit combination of Simple16 algorithm

will be coded according to the bit combination $\{1 \times 10, 2 \times 9\}$. The next partition $\{3, 22, 86\}$ will use bit combination $\{1 \times 10, 2 \times 9\}$. The last partition will consist of $\{467, 8194\}$ and will be coded with bit combination $\{2 \times 14\}$.

3.1.8 Frame Of Reference

Frame Of Reference (FOR) maps a set of large integers to relatively small integers. The basic approach is to identify the range (maximum m and minimum value M) from the set of integers. The smallest number will be coded as 0 and we will store the difference between the smallest number and the original number rather than storing the original set [14]. Each integer can be stored using $\lceil \log_2(M - m + 1) \rceil$ bits. Alternatively, if the integers are small, we may simply store the number of bits ($\lceil \log_2(M - m + 1) \rceil$) required to store each integer: we call this variation Binary Packing. Different authors have called FOR by different names: Anh and Moffat called it PackedBinary [4] whereas Delbru et al [11]

called their 128-integer binary packing FOR and their 32-integer binary packing AFOR-1. The first step of implementing this scheme is to partition the array of values into blocks (e.g., 128 integers). We have to compute the range of any particular partition and code the range accordingly. Afterwards, all values in a certain block are coded in reference to the range values. Consider a sequence of integers : $67, 78, 85, 96, 98$. We can find that the numbers range from 67 to 98. Using FOR, instead of storing the original sequence we can subtract 67 from each of the values and code the difference only. It transforms the sequence into $0, 11, 18, 29, 31$. We can now code each of the offset values using a maximum of 5 bits. We still need to store the minimum value 67 in full and we need 3 bits to store the fact that 5 bits have been used to store differences.

The main drawback of this approach is that a single outlier value might have a significant adverse effect on compression.

3.1.9 Patched coding

As discussed in earlier section 3.1.8, the performance of Binary Packing is negatively impacted by outliers. Consider the integer sequence $7, 78, 89, 99, 108$. These integers can be represented by $5 * 7 = 35$ bits. However, if the set has a large integer like $7, 78, 89, 99, 2147483647$ then it will require $31 * 5 = 155$ bits, because it needs 31 bits to store the difference between the last integer and the minimum value (7). As a result, we need to use 31 bits to store each integer. To alleviate this problem, patching has been introduced [48].

The main idea of patched schemes is to code the large values (called exceptions) of an integer block in a different location. PFOR is quite similar to FOR but it takes into consideration the exceptions. In patching, we first identify the value of b, such that, most of the values (e.g., 90% or so) will be less than 2^b. Any sequence of integers is coded in a block with an extra array to store exceptions. Integers greater than 2^b are considered exceptions. Each of the values smaller than 2^b is moved into its corresponding b-bit slot. The unused

b-bit slots will be used to construct a linked list, such that the b-bit slot of one exception stores the offset to the next exception. We get the offset value by calculating the difference between the index of the next exception and the index of the current exception minus one. If the offset value is greater than 2^b, we need to force the exception in between the two slots. We store all the values of exceptions in separate locations which can be called the exception table. There will be two phases for decompression. In the first phase b-bit values will be restored. In the second phase, big numbers will be restored according to the linked list of the exceptions.

We can use PFOR in conjunction with differential coding and the result is sometimes called PForDelta and PFD [47]. It encodes the difference of successive values. In case of PForDelta, we do decompression first then compute prefix sum to recover the original value from the delta coded integers.

We divide the input array into some partitions, which can be further subdivided into sub-arrays that have a fixed maximal size (e.g., 16 MB). We call each such subarray a page. In case of PFOR compression, all the integers within a page are coded by one single bit width. It is very important to identify the best base value b. To identify the base value, we need to create a sample from the integer sequence such as 2^{16} integers. Afterwards, we can test various bit widths until we get the best compression ratio [21] and choose the base value b.

Consider the sequence of integers in binary notation:

$$10, 01, 10, 01, 11, 101, 101011, 111, 110, 111000, \ldots$$

In this scenario, we consider that the base value is 3 ($b = 3$) and we have exception at index positions 6, 9. Therefore, the offset is $9 - 6 - 1 = 2$. As a result we will store 10, 01, 10, 01, 11, 101, **10**, 111, 110, ... Here, we store number 10 at index position 6. Thus, we can identify that the next exception is in 2 index positions ahead. We need to maintain two codewords to record the position of the first exception of the integer sequence and the

first exception value in the exception table.

NewPFD and OptPFD

Since PFOR does not compress the exception values and uses fixed base value b to encode the whole sequence of integers in any page, Yan et al [46] proposed two new schemes called NewPFD and OptPFD to enhance compression. In their implementation they used blocks of 128 integers. In this approach, they use bit width b for every block of integers rather than for the whole sequence in any page. Moreover, they do not use bit packed blocks to store the offsets of exceptions. They store the first b bits of exceptions in the bit packed block. They store the higher $32-b$ bits and the offset of exceptions in two separate arrays. These two arrays are compressed by some other compression method afterwards. Consider the integer sequence:

$$10, 01, 10, 01, 11, 101, 101011, 111, 110, 111000, \ldots$$

Assume we use base value $b = 3$. According to the new schemes, we store the integer sequence as

$$10, 01, 10, 01, 11, 101, \mathbf{11}, 111, 110, \mathbf{0}, \ldots$$

So, for each block of integers we have main data followed by compressed exceptions(in our example $101, 111, \ldots$). We need to store bit width b, the number of exceptions and the storage requirement of compressed exceptions as well. This coding method will achieve fast decompression speed and small compressed size [46]. Fast decompression can be achieved by these patched schemes as branching is avoided during decompression. NewPFD tries to get base value b as small as possible such that not more than 10% of the integers in a block are exceptions. On the other hand OptPFD takes the base value b to maximize the compression.

FastPFOR

All patched schemes split the input integers into pages based on certain numbers of integers. Those pages can be broken down into several blocks. PFOR uses single base value b for a full page, while newer methods such as NewPFD and OptPFD select base value b for every block. In the case of PFOR, it stores exceptions on a per page basis, whereas NewPFD and OptPFD record exceptions on a per block basis. New methods such as NewPFD or OptPFD achieve better compression than PFOR. However, PFOR is faster. To minimize the trade-off between speed and compression size among the patched schemes, Lemire et al proposed a new scheme called FastPFOR [21]. This scheme stores the exceptions on a per page basis, but selects the base value b on a per block basis. FastPFOR stores the location of exceptions using a single byte. Consider a sequence of integers in binary notation:

$$11, 1, 10, 11, 11, 111111, 10, 11, 1, 10, 110001, 1, 100101, 11, 1, 111111$$

In this sequence we have a maximum bit of 6. To determine the base value b, FastPFOR uses a formula to measure the cost of using any particular bit width. The formula is

$$b \times blocksize + (8 + (MAXbitwidth - b)) \times numberofexception \tag{3.2}$$

For every bit width we calculate the cost and choose the lowest cost. In our example, we have 16 integers. Among them, 8 are 2 bits long, 4 integers are 1 bit long and 4 are 6 bits long. The cost of $b = 1$ is $172(b \times 16 + (14 - b) \times c)$. For other values $b = 2, 6$, the costs are 80 and 96 respectively. As a result, we choose $b = 2$. We first store the integer using 2 bits. So, the sequence looks like

$$11, 1, 10, 11, 11, \mathbf{11}, 10, 11, 1, 10, \mathbf{01}, 1, \mathbf{01}, 11, 1, \mathbf{11}$$

We store these packed values for every block one after another. For every block of 128 integers we have a different base value b. During compression, we record different information such as the base value b, the maximum number of bits, the number of exceptions and the exception location in a byte array for each block. In our example we have base value $b = 2$ which can be stored by using 1 single byte, the maximum bit width is 6 (we can store it in 1 byte), the number of exceptions is 4 (it takes 1 byte) and the location of the exceptions are $5, 10, 12, 15$(1 byte to store each location). On top of all this information we need 16×2=32 bits (4 bytes) to store the packed integers. Thus, we need $3 + 3 + 4 = 10$ bytes in total to store 16 integers. Finally, we need to store the exception values ($1111, 1100, 1001, 1111$) as well which can be bit packed later on.

In the next chapter, we describe our experimental setup, hardware configuration, implementation detail and choose different fast integer compression schemes to run our experiment.

Chapter 4

Experimental Setup

The main objective of this study is to examine and compare the performance of patched schemes over other competitive methods. Mainly, we are comparing different schemes based on compression ratio and decompression speed. We conduct our experiment in different series. The first sequence of our experiment is based on synthetic data sets which will be discussed in Chapter 5. We can reproduce synthetic data at any time and perform testing on those data. We run different trials by varying different parameters such as integer size, number of integers, etc. Finally, we perform a thorough experiment using large realistic data sets (see Chapter 6).

4.1 Hardware

In order to run all of our experiments on both synthetic and realistic datasets we use a machine equipped with Intel Core $i5$-2400 processor and 8 GB of RAM. This is a quad core processor, each running at a clock speed of 3.1 GHz (with a Max Turbo Frequency of up to 3.4 GHz) and delivering four-way multi-core processing via parallelism. Each core has 256 KB of L2 cache and there is 6MB shared L3 cache. This processor is equipped with 4×32 KB of instruction caches and 4×32 KB of data caches. Each processor is capable of accessing 8 GB of memory with 2 memory channels. However, accessing

the memory depends on memory type and accessing the remote memory would induce a penalty for reads and writes for the corresponding processor.

4.2 Software

To quantify the trade-offs among different compression algorithms, we have to implement several compression schemes and apply them to randomly generated data and also real data sets. We use some code for compression and decompression of integers written in Java which is available as open source under the Apache License 2.0 (https://github.com/lemire/JavaFastPFOR). Along with the implementation of the core compression algorithm, we need to develop our own testing strategy to test and verify all the compression schemes. All the implementations have been written in Java using SDK version 1.7.0. In order to improve the memory usage, we set *-server* flag during run-time. It also activates compiler optimizations, such as method in-lining, which improves the performance of long-running software.

4.3 Experimental Preliminaries

We tried to run our experiments on low CPU load since CPU and memory usage would have an effect on performance. We record CPU time, and we only consider the in-memory processing time. When applicable, we include the overhead due to difference coding. All of our experiments are conducted in single-thread mode and without any parallelism or concurrency.

In our experiment we focus on compression of integer values. In the case of relational database systems, there are column values which are not in integer format initially. We can transform the column values in relational database systems into integer values by dictionary coding [5, 24, 28, 30]. In order to improve the compressibility, we create a hash table for the all entries of any particular column and map the most frequent values to the

smallest integer [6]. We call this frequency coding, as described in Section 3.1.2. We ignore the overhead due to the mapping between actual database values and the corresponding integers.

4.4 Compression schemes

In the case of databases, compression should achieve both space reduction and lower disk traffic. During query processing time, we need to fetch the column value from the disk into a main memory and cache which makes decompression speed the critical factor for performance. Compression in database should make the total time of transferring a compressed chunk of data from disk and then decompressing it in memory less than the time required to transfer the same chunk of data in uncompressed form. Fast decompression will enhance the cache utilization as well. We mainly compare the trade-offs of patched schemes with other schemes in database setting. We use different schemes from different categories like byte wise compression, fixed length encoding and different types of patched coding so that we can benchmark our experimental result. In our experiment, we use following schemes to evaluate the performance of compression of integer values.

- Variable-Byte 3.1.4
- Binary Packing or FOR 3.1.8
- Simple9 (S9) 3.1.6
- NewPFD 3.1.9
- FastPFD 3.1.9
- OptPFD 3.1.9

In the next chapter, we discuss experiment on synthetic data. We describe the nature of synthetic dataset and assess the result.

Chapter 5

Synthetic Data

To evaluate the performance of different compression schemes we performed thorough experiments on synthetic data. We varied different parameters such as integer size, number of integers, etc. We analyzed the effect of different aspects of datasets on compression. Again, size of integers may influence the performance of compression. Some coding schemes (i.e. Variable-Byte, Simple9) will take more bits to compress one large integer. We can save space by compressing small integers into smaller memory space compared to larger integers. Moreover, the difference between the consecutive integers also influences the compression ratio when we use delta coding or differential coding. The cardinality of the data may influence the compression rate.

We have identified different factors that can influence the compression size. Our final goal is to examine and compare the performance of different compression schemes on databases. In our experiment we examined not only compressed size but also compression and decompression speed. We examined how different factors can influence the time to compress and decompress data.

To evaluate the performance we ran several identical trials varying characteristics such as the number of integers and the maximum length of each integer. In the case of delta coding, the cardinality of data can influence the compression speed. In the case of low

cardinality and sorted sequence, delta values will be smaller. So, we can examine the compression speed by varying the cardinality of data.

In this chapter, we discuss the generation of our synthetic data. Afterwards, we describe our experiment varying different characteristics of data. Finally, we analyze the result.

5.1 Generation of data

To perform the experiment, we first generate data with two different distributions namely Uniform and Clustered Data from Anh and Moffat [3]. In the case of Uniform distribution we are generating integers randomly from zero to a certain limit. In the case of Clustered distribution we are also generating values from zero to a certain limit but in clustered fashion. This means, similar values will be clustered together. In the case of database tables, we find similar values in certain columns, resembling Clustered distributed data. However, these synthetic tests do not model accurately the columns of database tables. For example, in our tests, all sequences of integers are sorted. We use differential coding in this instance whereas we do not find such kind of data distribution in the case of realistic database tables. However, these synthetic tests provide us with a reference. In the case of synthetic data test, we can only measure the performance of different compression algorithm based on the comparison characteristics (See Section 2.4), but we can not make the actual assessment of performance.

As discussed in the previous section we varied different characteristics of data. Therefore, our first approach was to generate Uniform and Clustered data varying the size of the integers and run our test. In this experiment we have two scenarios first we ran our test varying integer size with Uniform distribution and later on we run the similar test on Clustered distribution.

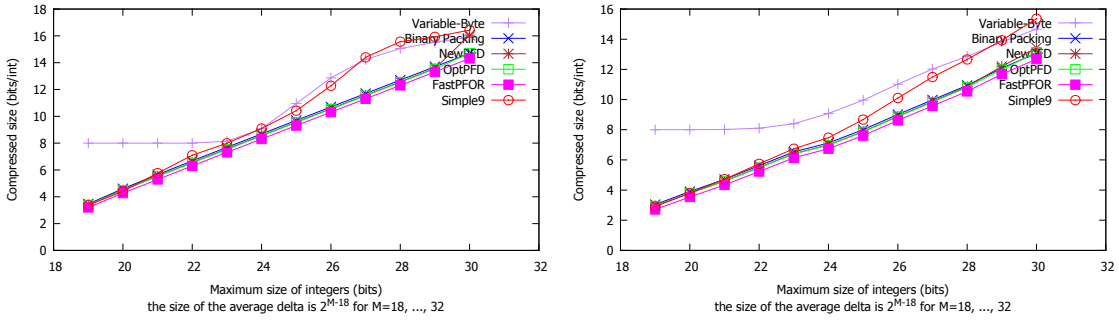
While generating synthetic data, we generated Uniform data first. We gave two parameters, N and Max. Our program generated randomly N distinct integers from 0 to Max

and stored it to an array of integers in sorted order. We used differential coding (See Section 3.1.3) on the input values and stored the delta values in an array of integers. This set of delta values was used as an input to evaluate different compression schemes later. In our implementation we took an delta coded integer array as an input and ran different compression schemes and recorded the compression ratio, compression speed and decompressing speed for every scheme.

In the case of differential coding, we are storing the difference between successive integers together with initial value: $(x_1, \delta_2 = x_2 - x_1, \delta_3 = x_3 - x_2, \ldots)$ instead of storing the original array of integers in sorted order such as $x_1, x_2, \ldots$ where $x_{i+1} \geq x_i$. This will result in non-negative integers that are typically much smaller than the original integers. Thus, we will be getting benefit for compressing a sequence of smaller numbers rather than the original one.

Our second approach was to generate a sequence of integers like Clustered distribution. In the case of Clustered distribution, similar data are generated together rather than uniform distribution [4]. Unlike the Uniform distribution, the Clustered distribution create clusters of similar values. Hence, we expect better compressibility with Clustered. We believe that it is more representative of real data found in database systems. We first took a few parameters such as total number of integers to be generated (N), the array where we stored integers $array[]$ and range of the integers Min and Max. In the case of the parameter N being 10, we generated the value in a uniform manner. When the value of N was greater than 10, we recursively called the same function varying the total number of integers and the range in random manner (see Anh and Moffat for details [3]). Thus, we generated data in Clustered distribution. Finally, we used delta coding and used delta values as input for compression.

In all of our experiments, we measured the timing based on CPU time. In the case of compression and decompression, we ran our every trial for 10 times and took the average of the recorded time for every trial. While measuring the timing we consider all the encoding



(a) Uniform Data

(b) Clustered Data

Figure 5.1: Compression Size Uniform and Clustered data

and decoding operations including delta coding and prefix sum.

5.1.1 Varying integer size

We generated our synthetic data sets with random integers in the range $[0, 2^{30})$ for both Uniform and Clustered distributions. We generated integers from zero to the integer with the specified number of bits, i.e., the range will be from 0 to a maximum 30 bit integer which means the interval is from 0 to $1,073,741,824$. Our test includes the integer range $[0, 2^{19})$, $[0, 2^{20})$, $[0, 2^{21})$ and so on. We generated $262,144$ number of distinct integers for every interval in sorted order and stored them into an array for further processing. Finally, we stored the delta values. We will get larger delta values with increase of the range, since we are generating fixed number$(262,144)$ of distinct integers in a particular interval. In this case, we have the size of the average delta between successive integers is 2^{M-18} for $M = 18, \ldots, 32$. We analyze the effect of integer size on compression. If we have all very small integers we can fit those into small memory space. We have listed the test result for both Uniform and Clustered distribution.

In our experiment with synthetic data, we try to avoid reading or writing data on disk. Processing will be slower if we store or retrieve data on disk.

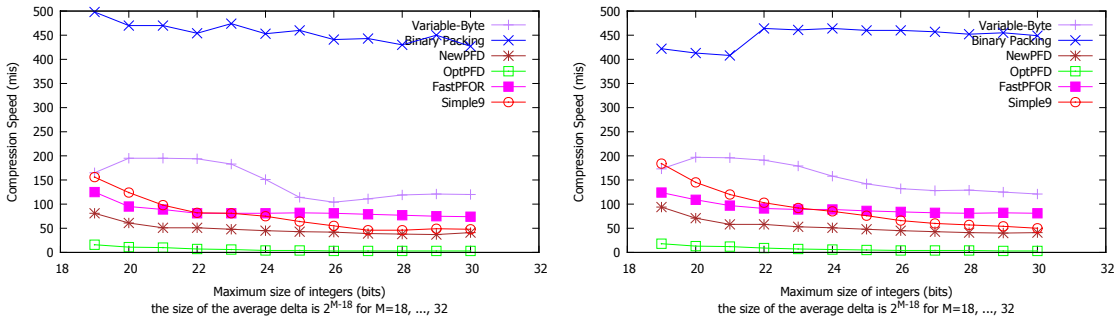
Since we are trying to evaluate the performance of different compression schemes, we tried to measure the compression ratio, compression speed and decompression speed for

those schemes (See Sec 4.4).

We used the same strategy to test Uniform and Clustered data. We have plotted the test results in Fig. 5.1. , Fig. 5.1a shows the result of Uniform distribution and Fig. 5.1b shows the result of Clustered distribution. We found the Clustered distribution is more compressible compared to Uniform distribution. In this test we changed the size of integer to some certain limit. Clustered distribution generates chunks of more similar data so it becomes more compressible. When we generate integers within small range such as $[0, 2^{19})$, it is obvious that we will get small delta values compared to larger range like $[0, 2^{30})$.Therefore, when we have lot of small integers, we can compress them into smaller memory space. For both Uniform and Clustered data in Fig. 5.1, we found small integers can be compressed into fewer bits compared to large numbers. In the case of maximum 19 bit integers, we can compress every integer within 2.91 bits (approximately) for Clustered distribution and within 3.37 bits (approximately) for Uniform distribution on average. There is an exception for Variable-Byte, since it compresses in a byte oriented manner. Therefore, Variable-Byte needs at least 8 bit to compress any integer 3.1.4.

Size of integers has some effect on compression speed as well. Compression speed decreases with increase of size of integer. In Fig. 5.2, we found compression speed is higher for small integers. Compression speed decreases with increment of integers size. We measure our compression speed in million of integers per second (mis). In every case Binary Packing outperformed all other schemes. The OptPFD has the lowest compressing speed compared to all other schemes.

In the case of large databases such as data warehouse or statistical database, there will be little update on data. As a result, we need not compress data over and over again [37, 48]. In most of the cases, we use compressed data for query processing and data retrieval. To evaluate the applicability of using any compression algorithm, we are mostly concerned about the compressed size and decompression speed of the scheme. As a result, most of the algorithms we used have lower compression speed compared to decompression speed.



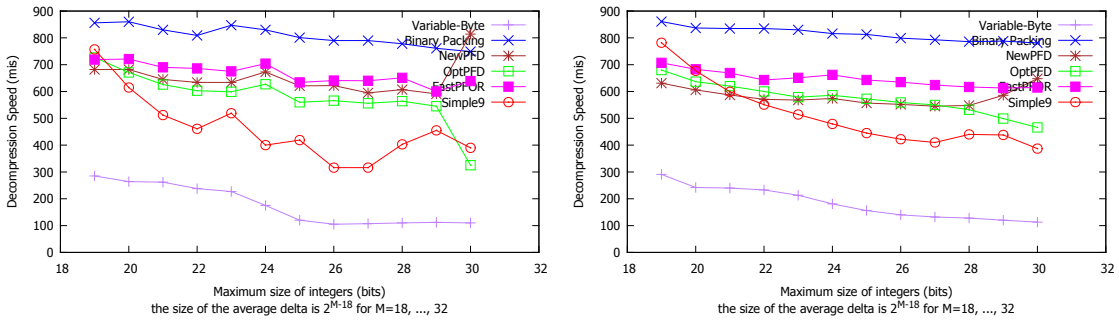
(a) Uniform Data (b) Clustered Data

Figure 5.2: Compression Speed Uniform and Clustered data

However, efficient compression is also important for database compression since we need to re-compress the data in case of updates.

Decompression speed was also influenced by integer size. Binary Packing outperformed all other schemes in case of Clustered and Uniform distribution 5.3. There is little difference in decompression speed between Uniform and Clustered distribution. Clustered data has better decompression speed over Uniform data in most of the schemes.

We used the same number of integers in every trial. We found the integer size influenced the performance of compression in every trial. In the case of the first trial, we have maximum number of bits is 19. Therefore, our range of random generated integer is in between $[0, 2^{19})$. The difference between consecutive integers is small for a small range of integers. In the case of a large range such as $[0, 2^{30})$, the difference between consecutive integers is large and we found it takes more bits to compress, such as around 15.35 bits (approximately) for Uniform distribution and 13.72 bits (approximately) for Clustered distribution. We can compress the integers into a smaller number of bits when we have small integers and also compression and decompression time will be significantly high for small integers compared to large integers.



(a) Uniform Data

(b) Clustered Data

Figure 5.3: Decompression Speed Uniform and Clustered data

5.1.2 Varying the number of integers

Our next approach to testing was to vary the number of integers and evaluate the performance. In this case, we generated data sets of random integers for both Uniform and Clustered distribution as before. Rather than generating a fixed number of integers like $262,144$, we generated a variable number of integers in each run. In every trial, we generated 20 integer arrays. Each of them containing same number of integers in a certain pass and in every pass we changed the size of the array by changing the number of input integers. We have done our experiments in two phases. In our first phase, we generated random integers within the same range of $[0, 2^{28})$ but varying the number of integers i.e., we generated 2^{13} integers for each input array in the first pass. We continued our trial by generating more integers than in the previous trial by a power of 2. We generated 2^{14} to 2^{20} integers in our test. Since the range is fixed, the average difference among random generated numbers will be higher when we generate a small number of integers.

In our experiment, we divided our test case into two parts. We called it short array and long array. In the case of short array, in the first pass, we generated 2^{13} integers. In the next passes we generated 2^{14} integers till 2^{16} integers. In every pass, we generated integers within the same range $[0, 2^{28})$.

In the case of long array, we started generating 20 arrays with 2^{17} integers. We generated same number of arrays with 2^{18}, 2^{19} and 2^{20} integers. We followed the same strategy such

Table 5.1: Average compression size, compression and decompression speed of Uniform and Clustered distribution for short array

	Uniform Distribution			Clustered Distribution		
Coding Scheme	Compressed Size	Compression Speed	Decompression Speed	Compressed Size	Compression Speed	Decompression Speed
Variable-Byte	15.18	176.00	195.50	13.50	182.50	198.50
Binary Packing	13.20	685.00	939.00	11.65	711.25	962.25
NewPFD	13.44	51.00	863.00	11.69	92.00	747.75
OptPFD	13.08	4.25	703.75	11.55	5.50	684.75
FastPFOR	12.81	239.5	816.25	11.24	231	818.75
Simple9	15.58	86.75	361.50	13.54	91.75	369.00

as the range is still the same $[0, 2^{28})$. However, the average difference between successive integers was small in the case of long arrays compared to short arrays.

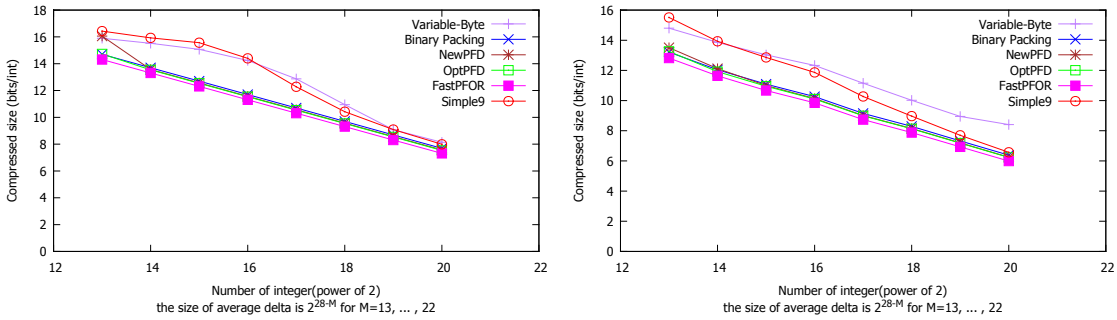
In our test, we generated data with Uniform and Clustered distribution for both short and long arrays. The result of compression size, compression speed and decompression speed of short array and long array has been listed in Table 5.1 and Table 5.2 respectively.

In Table 5.1, we record average of compression size , compression speed and decompression speed for short array. In this scenario, we ran 4 identical trials and generated 2^{13}, 2^{14}, ... 2^{16} integers. In every trial, we generated random numbers in the range $[0, 2^{28})$. The difference among consecutive numbers in short array is large, since we are generating fewer of integers on a fixed large range($[0, 2^{28}$)). The difference between consecutive integers are known as delta values. In Table 5.1, we found that to compress integers with large delta values takes more bits. Moreover, it takes more time to decompress. In case of long array, we ran 4 identical trials. We generate 2^{17}, 2^{18}, ... 2^{20} integers in the same range $[0, 2^{28})$. However, delta values will be lower in long arrays. As a result, we can achieve very good compression ratio and also good decompression speed, which is recorded in Table 5.2. Clustered distributed integers are more compressible than Uniform distributed integers in both short array and long array.

In Fig. 5.4, we can see the effect of number of integers on compression size. Compression

Table 5.2: Average compressed size, compression and decompression speed of Uniform and Clustered distribution for long array

	Uniform Distribution			Clustered Distribution		
Coding Scheme	Size	Compression	Decompression	Size	Compression	Decompression
Variable-Byte	10.26	224.00	232.50	9.63	243.00	254.50
Binary Packing	9.19	728.75	941.25	7.78	732.00	956.75
NewPFD	9.07	55.25	787.50	7.65	114.25	726.50
OptPFD	9.03	6.00	771.25	7.63	8.75	752.50
FastPFOR	8.81	248.00	837.25	7.38	239.50	832.50
Simple9	9.94	109.00	394.25	8.37	121.00	433.25



(a) Uniform Data

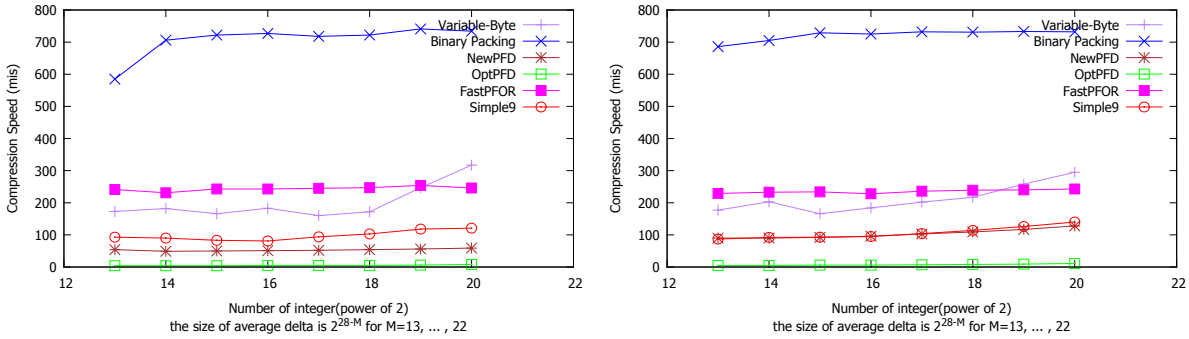
(b) Clustered Data

Figure 5.4: Compressed Size Uniform and Clustered data

size is decreasing whenever we are generating more integers on a fixed range $[0, 2^{28})$. We will get small delta values when we have more integers. As a result we can compress integers with fewer bits.

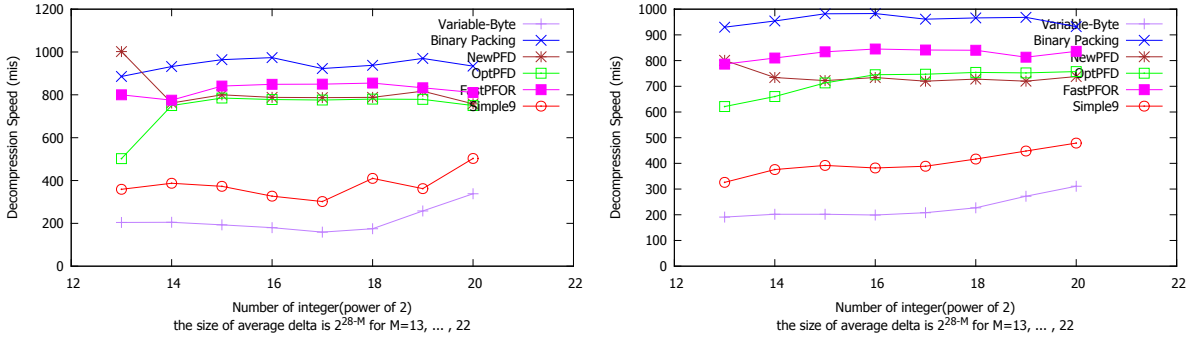
Fig. 5.5 shows the compression speed. Fig. 5.6 shows the decompression speed. We can see the effect of the number of integers on compression speed in Fig. 5.5. In the case of Binary Packing the compression speed varies a lot. Compression speed increases with the increment of number of integers, since delta value decreases as the number of integers increases. Binary Packing is significantly faster than any other method in terms of compression speed. Moreover, Clustered distributed integer has faster compression speed than Uniform distributed integer set.

We can see in Fig. 5.6, decompression speed increases with the increase in number of



(a) Uniform Data (b) Clustered Data

Figure 5.5: Compression Speed Uniform and Clustered data



(a) Uniform Data (b) Clustered Data

Figure 5.6: Decompression Speed Uniform and Clustered data

integers. Binary Packing is faster in terms of decompression in most cases.

In our synthetic data test, we generated a fixed number of integers and also a variable number of integers, but in both of our cases we have seen the size of integers influences the compression ratio and speed of compression and decompression. However, Binary Packing is clearly the winner for speed of compression and decompression but it does not compress as effectively as OptPFD and FastPFOR. In most of the cases FastPFOR compresses very well. Compression ratio for OptPFD is also very good, but it has the worst compression speed. FastPFOR has the best speed of compression and decompression among patched schemes and Simple9 but it never beats Binary Packing.

In this chapter we analyzed the result of compression ratio, speed of compression and

decompression on synthetic data sets. To quantify the trade-offs of patched schemes with other schemes, we need to experiment on real data sets as well.

Experiments in the next chapter analyze the performance of different compression schemes and assess effect of row order on compression on real datasets. This will conclude our experimental study.

Chapter 6

Real Data

We measured the performance of different compression schemes based on synthetic data in Chapter 5. Our synthetic data set is not an exact simulation for real life scenario, so we conduct experiments using datasets that have realistic data. We use realistic data to compare the performance of different compression schemes in datasets with real data.
In this chapter we will first discuss the characteristics of real datasets on which we performed our experiments. Afterwards, we will discuss our strategy to run the experiment. Finally, we discuss the results that we got from our experiment to evaluate the performance.

6.1 Datasets

We have chosen three data sets (Census-Income [18], Census1881 [22,29] and Star Schema Benchmark [27]) to evaluate the performance of different integer compression schemes and summarize the response for compression ratio and decompression speed. All of the datasets Census-Income, Census 1881 and Star Schema Benchmark are derived from real data. All rows are distinct in every table. All columns have different cardinalities. Column values will be mapped into 32-bit integers using frequency coding where the most frequent value will be mapped to 1. These data sets are in comma-separated-value (CSV) files. In

CSV files, every field is actually the value of the column and every row is separated by newline.

6.1.1 Census-Income

This data set contains different demographic and employment-related variables which have been extracted from the 1994 and 1995 Current Population Surveys conducted by the U.S. Census Bureau [18]. Census-Income has a larger number of columns. It has 42 columns and the number of records is $199,523$. Census-Income has $103,419$ distinct values. The column cardinalities for Census-Income is in the range from 2 to $99,800$.

6.1.2 Census1881

In the case of Census1881, it has 7 columns and $4,277,807$ rows and it has $343,422$ distinct values. The column cardinalities for Census1881 is in between 138 and $152,882$. It is publicly available at [22, 29]. It contains different individual level information on all household members.

6.1.3 Star Schema Benchmark

We generated the SSB fact table using a version of the DBGEN software modified by O'Neil [27]. We used a scale factor of 40 to generate it: that is, we used command dbgen -s 40 -T l. The SSB table has column cardinalities ranging from 1 to $3,345,588$. (The fact table has a column with a single value in it: zero.).

All columns have different cardinalities. The characteristics of the data sets are summarized in Table 6.1.

	rows	columns	distinct values
Census-Income	199 523	42	103 419
Census1881	4 277 807	7	343 422
SSB	6 001 171	17	6 001 171

Table 6.1: Characteristics of realistic data sets. We include the total number of distinct values over all columns.

6.2 Pre-processing

All our real datasets used in the experiment are comma-separated files. We followed a few preprocessing strategies and then ran our experiments to evaluate the performance of different compression schemes.

6.2.1 Create Frequency Coded File

The first step of our experiment was to create a frequency coded file for every dataset that we used. All the attribute values were coded by non-negative integers. The procedure for frequency coding has been described in Section 3.1.2. During our experiment, we made different hash tables for every column of each real dataset and mapped the most frequent value to 1. We used this frequency coded normalized file for further experiment. Frequency coding is unaffected by row order in such a way that, irrespective of the row order, the same integer codes will be attributed to the same attribute values. Note however that the compression ratios of a frequency coded table might be affected by row orders: the order in which the values appear matters for compression. We encode the file with integer values and data were stored in binary flat files, using 32 bits per integer.

6.2.2 Create Random Coded File

We tried to test all the real data sets with random coded files as well. Random coding is affected by row order in such a way that, depending on the row order, the same integer

Name	City	Tax Amount
Maria	Saint John	365
Robert	Saint John	235
Robert	Montreal	400.5
William	Saint John	265.7
Xan	Saint John	400.5
William	Montreal	365
Robert	Toronto	235
Maria	Toronto	365
Robert	Toronto	400.5
Maria	Saint John	400.5

(a)

Name	City	Tax Amount
1	1	1
2	1	2
2	2	3
3	1	4
4	1	3
3	2	1
2	3	2
1	3	1
2	3	3
1	1	3

(b)

Figure 6.1: Example of a random-coded table with different row order for the same table of Fig. 3.3, Fig. 6.1a is the original table. Fig. 6.1b is its corresponding random coded table

codes will be attributed to the different attribute values. In the case of shuffled files we might geta different integer code for any column values. In order to create a random coded file, we created a hash table for every column of the table according to the occurrence of every value in the column. In this case, the first value is mapped to 1 (See Section 3.1.2). We replaced the value of each column according to the hash-table entry that we made in our earlier step. Thus, we created the random coded file. As a result, we can see that random coding is affected by row order. In the case of a different row order for the table in Fig. 3.3, column values will be coded in different integer values. We can see the effect of different row order for the same table in Fig 6.1.

6.2.3 Shuffling

Rather than testing on the original file, we first shuffled the both the random and the frequency coded file. We used the Unix command *shuf* to shuffle both files created from each data set. We shuffled the files to evaluate the performance of compression size, compression speed and decompression speed on data with random order.

6.2.4 Sorting

One of our primary goals is to assess the effect of row order on performance of compression. In order to examine the effect of sorting on any column, we first shuffle the file and considered two cases. First, we sorted the shuffled file with the high cardinality column and ran different compression schemes on the sorted file. Again, we sorted the shuffled file with the low cardinality column and ran the same experiments to evaluate the performance.

6.3 Experiments with compression

To benchmark the performance of patched schemes over other schemes we ran several identical trials on data sets with real data.. We can quantify the variance of different compression schemes when the trials are actually identical. We can conclude our survey and determine how well patched schemes can work to improve the performance of databases for information retrieval.

In the case of synthetic data in Chapter 5, we generated random data and used those random data to evaluate the performance of different compression schemes. We tried to generate data in Uniform and Clustered distributions. We used those data as an input and measured the performance of compression. In the case of real data, we cannot use this strategy because we are not generating those data.

To evaluate the performance of different compression schemes we considered four test cases for every dataset. First, we ran the compression algorithms on the original file and examined its performance. (The files were given to us with the rows in some predetermined order: we don't expect this order to be random.) Secondly, we shuffled the file and repeated the test. Next, we reran the same experiment on the file which has been sorted on the high cardinality column. Lastly, we tested the compression performance using the file which has been sorted with low cardinality column. In every test case we

used two types of coding: Random coding (See Section 3.1.2) and Frequency coding (See Section 3.1.2). As a result, our data sets include only the integer coded values. Therefore, we can evaluate the performance of fast integer compression schemes over tables. In our experiment, we ran several trials and all the trials are identical. We ran 10 identical trials on every experimental configuration and recorded the average values. We tried to measure the performance of compression on different comparison characteristics(See Section 2.4) which includes compression ratio, compression speed and decompression speed. So, in our experiments we exclude the time to read the file.

6.3.1 Test on whole file

In the case of real data set testing, we ran our first test on the full file. We measured the compression size, compression speed and decompression speed for the whole files of each data set. In every data set we have two types of coded files and for every coded file we have four scenarios that includes, original file, shuffled file, sorted with high cardinality column and sorted with low cardinality column. We will discuss the compression size, compression and decompression speed from our experiments in this section.

Compression Size

As discussed previously, we had four scenarios for every integer-coded file from our datasets except for SSB dataset. We sort the SSB dataset with second lowest cardinality column(Column no 7), since it has one column with a single value(See Section 6.1.3). The compression sizes that we got for frequency coded file of Census1881, Census-Income and SSB are shown in Table 6.2, Table 6.3 and Table 6.4 respectively. In all of our data sets we found that the shuffled file had the worst compression ratio for every scheme. We got good compression result in the case of file sorted with high cardinality column. In the case of row ordering there are more likely to be long sequences of similar values. It clusters the large integers together, which makes the column more compressible. Sorting improves

Table 6.2: Result of compression (bits per integer) on Census1881 with frequency coded file. The table is either in its original order, in shuffled order, sorted by a highest cardinality column or by a lowest cardinality column.

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	10.62	10.62	10.62	10.62
Binary Packing	9.38	11.12	9.59	10.56
NewPFD	8.82	9.86	8.75	9.46
OptPFD	8.25	9.31	8.41	8.94
FastPFOR	8.18	9.28	8.32	8.93
Simple9	9.24	10.75	9.97	10.38

Table 6.3: Result of compression (bits per integer) on Census-Income with frequency coded file

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	8.31	8.31	8.31	8.31
Binary Packing	4.00	4.00	3.83	3.96
NewPFD	3.57	3.57	3.43	3.52
OptPFD	3.41	3.41	3.33	3.38
FastPFOR	3.20	3.20	3.11	3.18
Simple9	3.86	3.86	3.67	3.82

the compression ratio. We get significant improvement in compression for file sorted with high cardinality column compared to shuffled input.

Our second experiment was to run the same four scenarios with random-coded files for each of the datasets. The compression size for Census1881, Census-Income and SSB are recorded in Table 6.5, Table 6.6 and Table 6.7 respectively. We got similar results in the case of random-coded files as well. Shuffled input results in worst compression size and files sorted with high cardinality column results in good compression size.

In both frequency and random-coded files we found Variable-Byte has fixed compression size in all the four scenarios. Variable-Byte is unaffected by row order because integers are compressed one-by-one and not in blocks. Variable-Byte encodes integers from approximately 8–15 bits. This is not unexpected because Variable-Byte compresses integers in a

Table 6.4: Result of compression (bits per integer) on SSB with frequency coded file

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	15.00	15.00	15.00	15.00
Binary Packing	11.37	11.42	11.15	11.37
NewPFD	13.06	13.19	12.32	13.14
OptPFD	11.84	11.85	11.80	11.80
FastPFOR	11.27	11.29	11.06	11.24
Simple9	15.75	15.90	15.72	15.84

Table 6.5: Result of compression (bits per integer) on Census1881 with random coded file

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	12.25	12.25	12.25	12.25
Binary Packing	10.77	12.77	11.29	12.21
NewPFD	10.63	12.36	10.79	11.94
OptPFD	10.00	11.15	10.17	10.87
FastPFOR	9.74	10.96	9.96	10.72
Simple9	12.32	13.79	12.92	13.50

Table 6.6: Result of compression (bits per integer) on Census-Income with random coded file

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	8.34	8.34	8.34	8.34
Binary Packing	4.12	4.15	4.02	4.10
NewPFD	3.66	3.71	3.61	3.66
OptPFD	3.48	3.49	3.44	3.47
FastPFOR	3.26	3.28	3.23	3.26
Simple9	4.09	4.10	3.94	4.06

Table 6.7: Result of compression (bits per integer) on SSB with random coded file

Coding Scheme	Original	Shuffled	High Card.	Low Card.
Variable-Byte	15.10	15.10	15.10	15.10
Binary Packing	11.16	11.43	11.16	11.38
NewPFD	12.43	13.60	12.43	13.56
OptPFD	12.14	12.14	12.03	12.09
FastPFOR	11.06	11.30	11.06	11.25
Simple9	15.82	15.97	15.82	15.91

byte-oriented way, and it takes at least one byte to compress any integer. This results in a bad compression ratio compared to other schemes that we used. We find all the patched schemes and Binary Packing compress better than Variable-Byte or Simple9. FastPFOR is the winner for compression size in each of the four scenarios and both frequency and random-coded files. FastPFOR takes approximately 1 few bit than Binary Packing to compress any integer.

The compression ratio of random-coded files is worse than for frequency-coded files. This is not surprising, because frequency coding maps column values according to their frequencies in the table (See Section 3.1.2). As a result, we find lots of small integers in a frequency-coded file and those small integers can be compressed in smaller number of bits. On the other hand, we have more larger integers in random-coded files. We need more bits to compress large integers. This result is quite similar to our synthetic data set.

In our experiment, SSB has more records than Census1881 and Census-Income. It is expected that we will have more entries in the hash table to create frequency or random-coded files in the case of SSB compared to Census1881 or Census-Income. This results in more bits to compress in SSB file than in other files. In our results we can see that it takes approximately 2–4 more bits to compress any integer in frequency-coded SSB than in Census1881 and 7–12 more bits in Census-Income. In the case of random-coded files, we get quite similar variations. On average, it takes 2–4 more bits to compress any integer in SSB than in Census1881 and 7–12 more bits in Census-Income.

Table 6.8: Result of compression and decompression speed on Census1881 with frequency coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	33	31	33	31	165	197	214	186
Binary Packing	729	711	746	732	1151	1089	1151	1135
NewPFD	52	36	40	34	709	615	729	689
OptPFD	6	3	5	4	421	357	482	381
FastPFOR	104	76	89	84	776	707	763	730
Simple9	78	60	69	64	488	377	447	398

Compression and Decompression Speed

We recorded compression and decompression speed in all four scenarios (original row order, shuffled, sorted with high cardinality column and sorted with low cardinality column) for SSB, Census1881 and Census-Income. We found quite similar results of compression and decompression speed for sorted and unsorted files. However, shuffled files had worse compression and decompression speed compared to other scenarios. In most of the cases, we observed good compression and decompression speed for files sorted with high cardinality column. We have listed our results for compression and decompression speed for frequency coded SSB, Census1881 and Census-Income file in Table 6.10,Table 6.8 and Table 6.9. In neither of these two cases, does Variable-Byte have good compression and decompression speed. Binary Packing is always the winner in terms of compression and decompression speed. OptPFD results in reasonable compression ratio but it has the worst speed of compression. Decompression speed of OptPFD is much slower than other alternatives.

We do not get the same speed of compression and decompression for all real data sets. Due to different distribution of data, we may get different results for speed of compression and decompression [33]. We get higher speed of compression and decompression in SSB than in Census1881 or Census-Income. Compression and decompression speed for all four scenarios of random-coded SSB, Census1881 and Census-Income are shown in Table 6.11, Table 6.12 and Table 6.13. Simple9 has better speed of compression and

Table 6.9: Result of compression and decompression speed on Census-Income with frequency coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	55	55	58	59	280	343	291	289
Binary Packing	639	647	522	644	1347	1353	1344	1348
NewPFD	93	89	101	89	807	805	816	816
OptPFD	14	17	16	16	764	761	783	772
FastPFOR	134	137	139	133	943	932	943	926
Simple9	159	161	170	161	956	959	1009	963

Table 6.10: Result of compression and decompression speed on SSB with frequency coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	143	142	143	141	178	180	183	179
Binary Packing	915	914	928	926	1348	1350	1382	1392
NewPFD	68	65	70	67	1286	1340	1346	1335
OptPFD	8	7	9	7	834	838	849	845
FastPFOR	309	308	315	310	1252	1240	1270	1256
Simple9	79	77	78	79	298	296	301	300

decompression than Variable-Byte but it cannot beat the speed of Binary Packing in any case. However, Simple9 sometimes beats NewPFD and OptPFD with speed of compression and decompression. FastPFOR has better speed of compression and decompression than any other scheme except Binary Packing. Binary Packing is clearly the winner with respect to speed of compression and decompression.

Table 6.11: Result of compression and decompression speed on SSB with random coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	143	142	143	141	164	176	182	178
Binary Packing	924	933	931	929	1376	1396	1389	1396
NewPFD	71	87	94	88	1364	1405	1392	1408
OptPFD	10	7	10	8	840	714	948	700
FastPFOR	321	324	326	327	1295	1305	1281	1311
Simple9	76	78	81	79	302	295	301	296

Table 6.12: Result of compression and decompression speed on Census1881 with random coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	32	31	32	31	161	160	178	160
Binary Packing	801	749	773	757	1216	1125	1164	1130
NewPFD	51	35	41	42	840	719	769	827
OptPFD	5	3	5	3	468	381	449	414
FastPFOR	115	78	94	83	838	710	823	721
Simple9	66	55	63	54	464	355	440	383

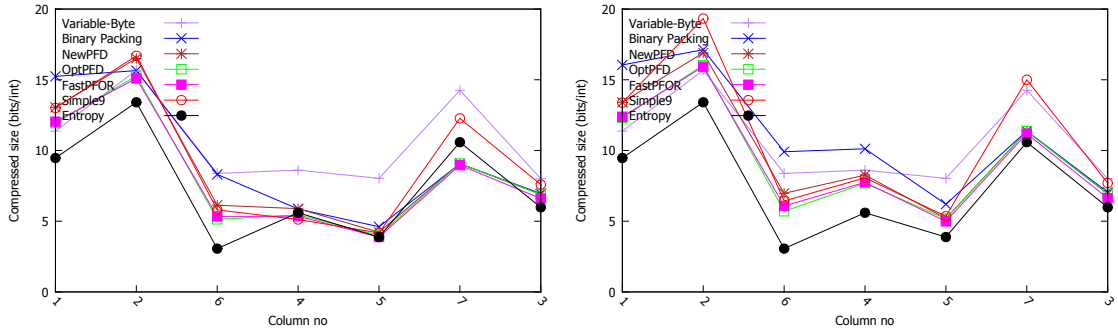
Table 6.13: Result of compression and decompression speed on Census-Income with random coded file

	Compression speed				Decompression speed			
Coding Scheme	Orig.	Shuf.	High	Low	Orig.	Shuf.	High	Low
Variable-Byte	60	59	56	62	311	278	296	289
Binary Packing	648	592	665	621	1386	1368	1384	1367
NewPFD	90	91	96	92	844	843	853	862
OptPFD	14	15	16	15	763	746	765	747
FastPFOR	135	133	134	134	931	893	947	945
Simple9	158	156	164	158	981	978	1010	986

6.3.2 Performance of column-wise compression

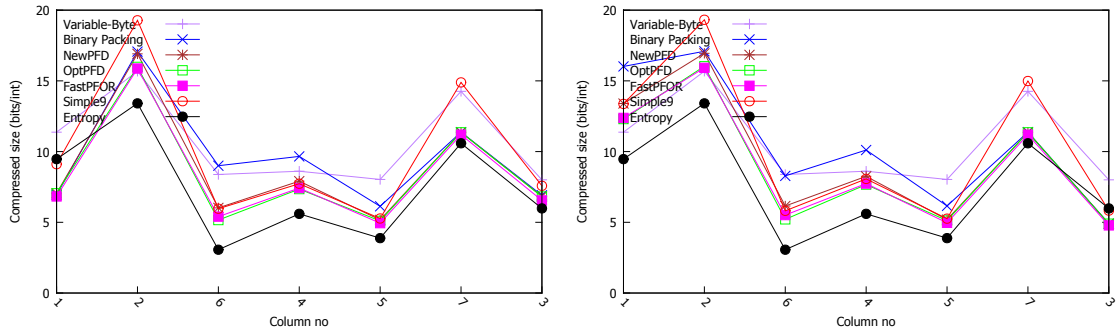
We tested the performance on column-wise compression. Rather than run compression on whole datasets we ran our compression schemes on every column and recorded the results. In evaluating column-based compression, we ran our experiment on four scenarios for both frequency and random-coded files. We compared the compression size with the Shannon entropy (Section 2.5) value of every column and plotted it. We plotted the column-wise compression size from frequency-coded and random coded files of Census1881 are shown in Fig. 6.2 and Fig. 6.3 respectively.

Performance of the compression scheme always depends on the nature of the data. Binary Packing or patched schemes are very much sensitive to the order of the items in a column. In the case of sorted column we got very good compression ratio than unsorted column in all the data sets. We can see the effect of sorting in Fig 6.5 and Fig 6.4 for SSB data set. We find similar result of sorting in Census1881 data set as well (See Fig 6.2 and 6.3). One



(a) Original File

(b) Shuffled File



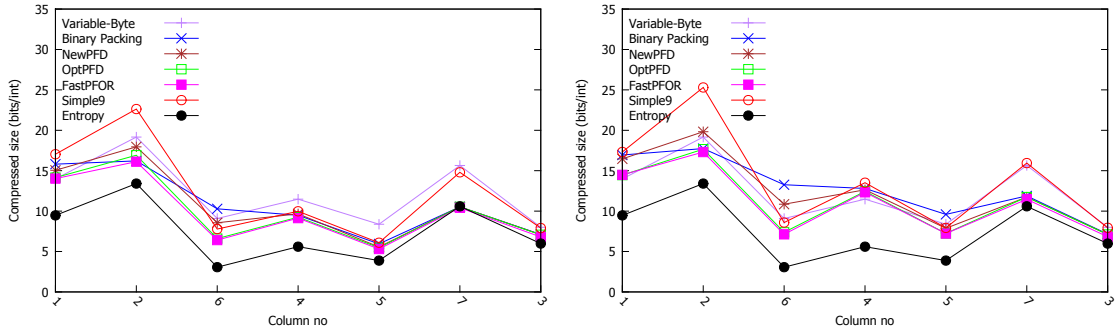
(c) Sort High Cardinality Column (column 1)

(d) Sort Low Cardinality Column(column 3)

Figure 6.2: Column-wise compression size for Census1881 of frequency coded file

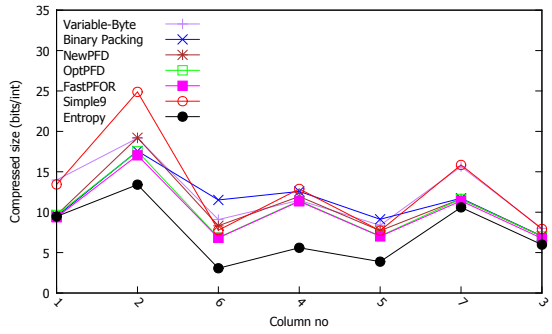
might object that Shannon entropy, which assumes that there is statistical independence between the different values, is not applicable to sorted data. While this is correct, Shannon entropy still offers an upper bound on the compression performance of a compression scheme that compresses each value as if it were independent like Variable-Byte. That is, we know that Variable-Byte cannot beat the entropy even if the data is sorted.

In Fig. 6.2a compression size of Binary Packing, NewPFD, OptPFD, FastPFOR for few columns beats entropy values for frequency coded files which appears to us as a surprisingly good result. However, in the case of random-coded original file in Fig. 6.3a, no compression schemes can beat entropy. We see similar kind of result for SSB dataset as well (See Fig. 6.5a and Fig. 6.4a). We assume that those column values might come in sorted order. So, we tried to shuffle the file and again ran the test. In Fig. 6.2b and 6.3b, we see no compression schemes can beat entropy value which is not unexpected. In the case of shuffled files, the compression ratio is not good compared to that of the original file.

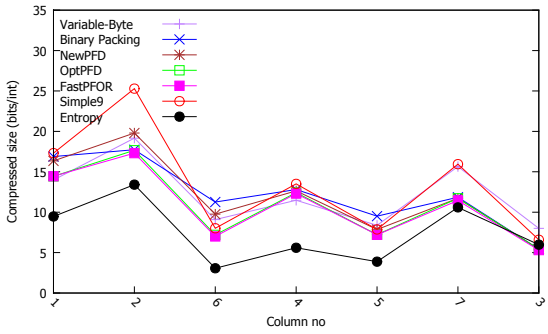


(a) Original File

(b) Shuffled File



(c) Sort High Cardinality Column (column 1)

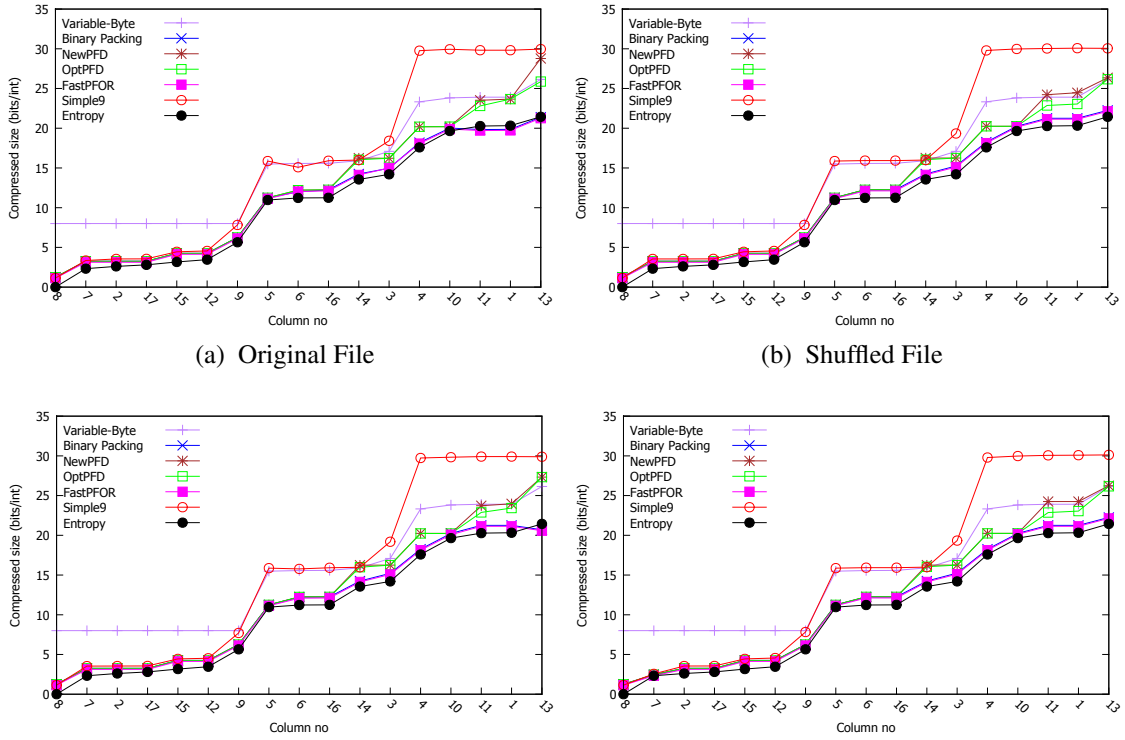


(d) Sort Low Cardinality Column(column 3)

Figure 6.3: column-wise compression size for Census1881 of random coded file

This is similar to our previous results of compression for whole file. To assess the effect of row order we tried to sort different column values. In our experiment we have two scenarios of sorting. First, we sort the high cardinality column and measure the performance of compression ratio, speed of compression and decompression. The second scenario is to sort the low cardinality column. In Fig. 6.2c and Fig. 6.3c, we can see that Binary Packing, NewPFD, OptPFD and FastPFOR fall below the entropy values for high cardinality column which is sorted. In the case of SSB dataset, we found that Binary Packing and FastPFOR have beaten the entropy value (See Fig. 6.5c and Fig. 6.4c). However, in the case of SSB file NewPFD and OptPFD cannot beat the entropy values. In Fig. 6.2d and Fig. 6.3d we see few schemes beat entropy values for sorted column. Therefore, we can conclude that row order has some positive effect on compression.

With regard to the compression speed, we find that Binary Packing always outperformed rest of the schemes in both frequency-coded (See Fig. 6.6, Fig. 6.8), and random-coded



(a) Original File

(b) Shuffled File

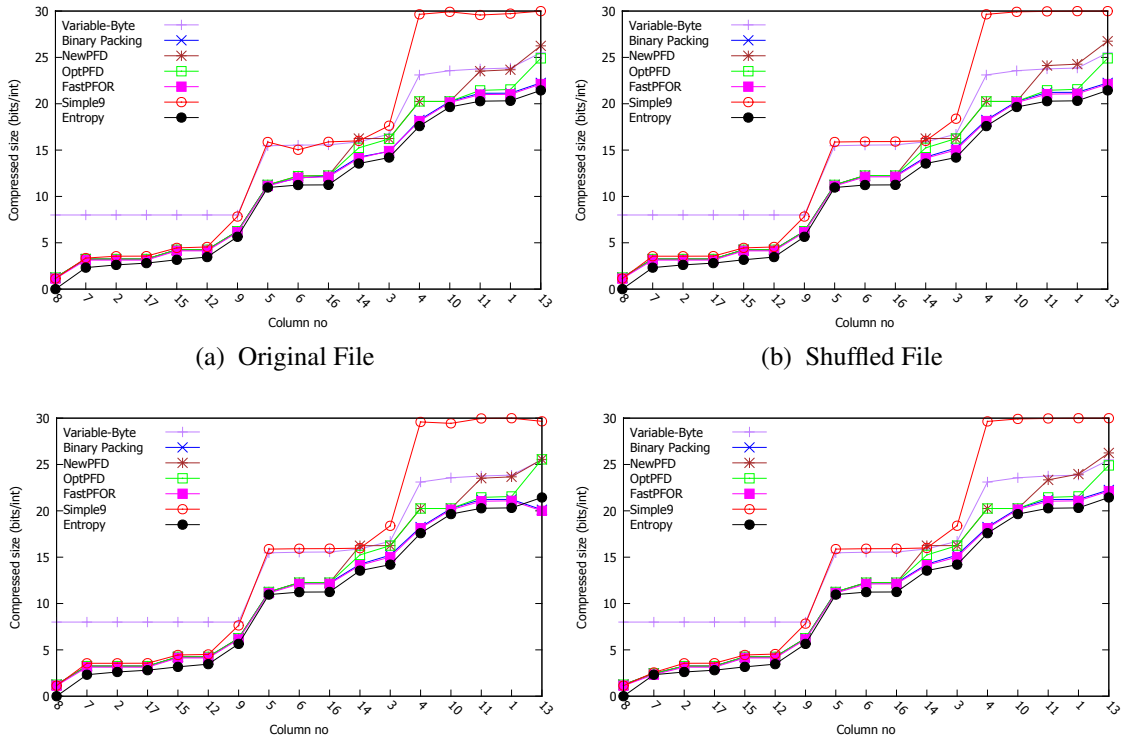
(c) Sort High Cardinality Column (column 13)

(d) Sort Low Cardinality Column (column 7)

Figure 6.4: Column-wise compression size for SSB of random coded file

file(See Fig. 6.7, Fig. 6.9). We find compression speed results similar to those for synthetic data and the whole file. We see the effect of row ordering in both SSB and Census1881 dataset. In Fig. 6.6c and Fig. 6.6d we see the effect of row order of frequency coded files for Census1881. We see the same effect of SSB dataset as well (See Fig. 6.8c and Fig. 6.4c). We are getting good compression speed for sorted input compared to original and shuffled files. We get similar degree of improvement of compression on sorted files for random coded files as well(See Fig. 6.7c and Fig. 6.7d). In the case of sorted column, we get good speed for compression for every scheme that we used.

Binary Packing is also the winner for decompression speed. In both random and frequency coded files and all the four scenarios of our experiment, we see speed of decompression of Binary Packing is significantly higher than that of any other schemes for Census1881 (See Fig. 6.10 Fig. 6.11) and SSB dataset(See Fig. 6.12 and Fig. 6.13). Variable-Byte has the worst decompression speed compared to other schemes that we used. FastPFOR has



(a) Original File

(b) Shuffled File

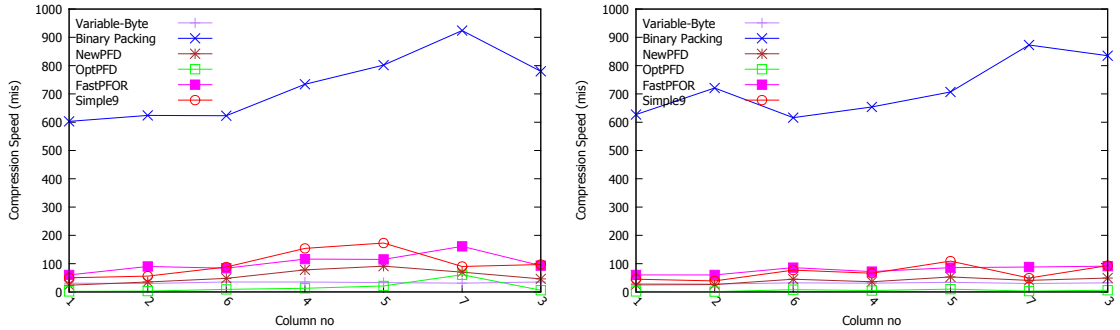
(c) Sort High Cardinality Column (column 13)

(d) Sort Low Cardinality Column (column 7)

Figure 6.5: Column-wise compression size for SSB of frequency coded file

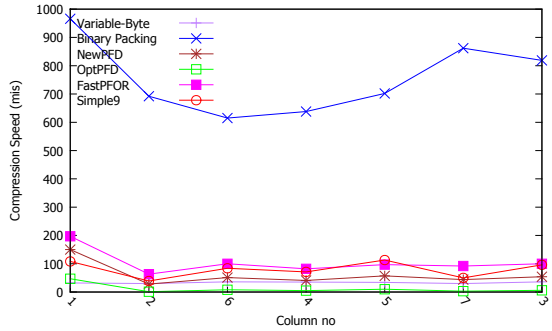
reasonably good decompression speed compared to other patched schemes and Simple9. However, decompression speed of FastPFOR is considerably less than that of Binary Packing. We see substantial effect of row order in decompression speed. Sorted column can be decompressed faster than original or shuffled file(See Fig. 6.10 Fig. 6.11, Fig. 6.12 and Fig. 6.13). In the case of frequency coded file for Census1881(See Fig. 6.10c, Fig. 6.10d) and SSB(See Fig. 6.12c and Fig. 6.12d), we see the sorted column can be decompressed in faster speed than other column. We see similar effect of sorting on decompression in the case of random coded file(See Fig. 6.11c, Fig. 6.11d, Fig. 6.13c and Fig. 6.13d).

We see the performance of compression varied from column to column. We found different compression ratio, compression speed and speed of decompression from column to column. Different columns may have different data distributions. As a result, the performance varies from column to column. However, in all the case of sorted column we found better compression ratio, compression speed and decompression speed compared to

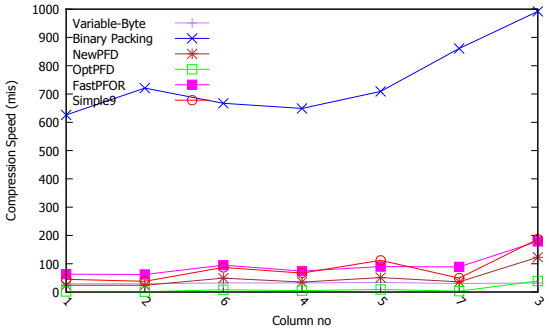


(a) Original File

(b) Shuffled File



(c) Sort High Cardinality Column (column 1)



(d) Sort Low Cardinality Column(column 3)

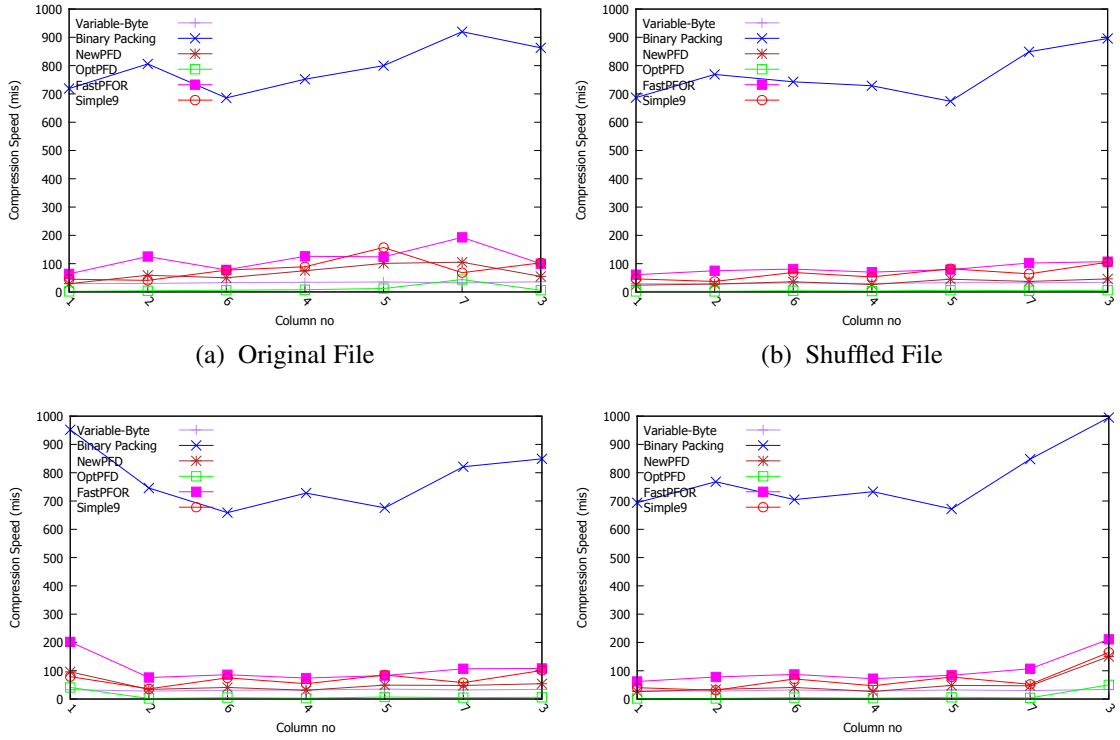
Figure 6.6: Column-wise compression speed for Census1881 of frequency coded file

original row order.

6.4 Effect of Row Order

Row order has a substantial effect on compression. In the case of sorted input we found better compression ratio compared to shuffled input. In our experiment we sorted table based on the cardinality of the column to assess to effect of sorting. We have two cases, one of them is to sort the table with high cardinality column and other one is to sort the table with low cardinality column. In Fig. 6.14, we see that sorting with a high cardinality column results in best compression size. However, sorted input with a low cardinality column also reduces size compared to a shuffled input.

Variable-Byte always results in fixed compression size for all the three cases. In the case of shuffled input Binary Packing compress integers poorly compared to other schemes.



(a) Original File

(b) Shuffled File

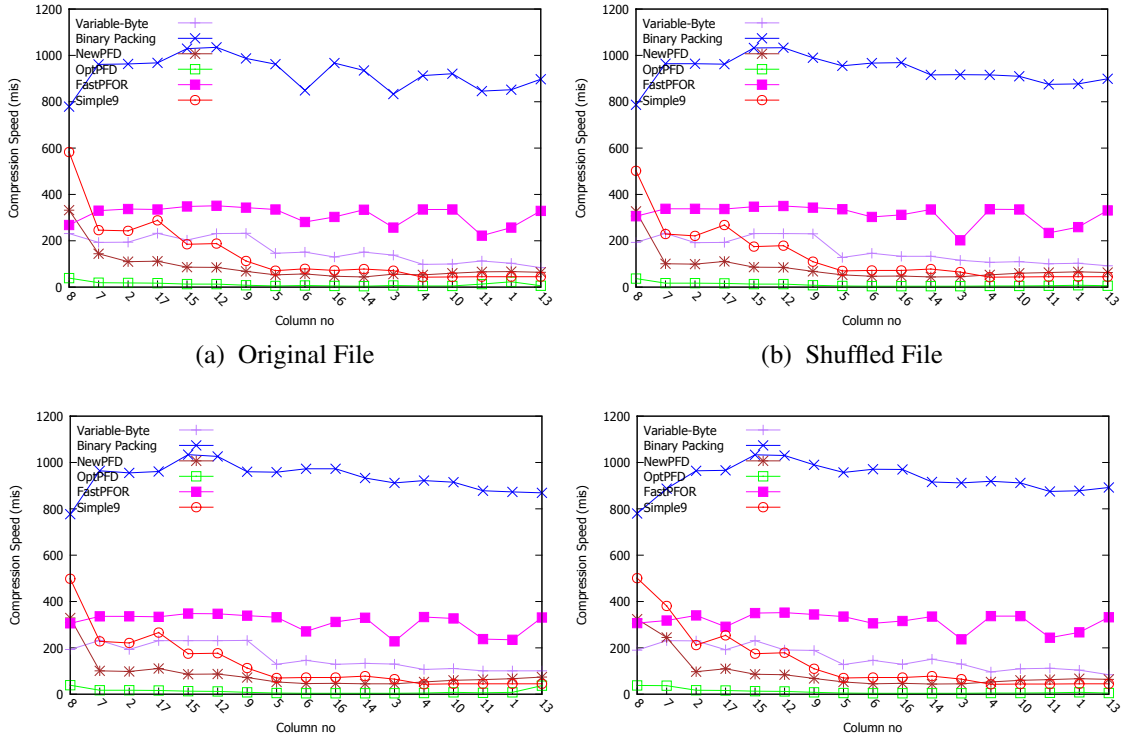
(c) Sort High Cardinality Column (column 1)

(d) Sort Low Cardinality Column(column 3)

Figure 6.7: Column-wise compression speed for Census1881 of random coded file

FastPFOR is always the winner in case of compression size. All the patched schemes performed well with respect to compression ratio for random and sorted input. Simple9 did not compress well compared to the patched schemes.

In this Chapter, we ran different tests on real data sets and analyze the result of different compression schemes with respect to compression size, speed of compression and decompression. In the case of speed of compression and decompression we found Binary Packing always outperformed any other scheme and FastPFOR is the best space-efficient scheme. In our study, we are mostly concerned about the speed of decompression since it is important in the case of data retrieval. We found Binary Packing is both fast and space-efficient. It used only ≈ 1 bit more than our most space-efficient scheme FastPFOR.



(a) Original File

(b) Shuffled File

(c) Sort High Cardinality Column (column 13)

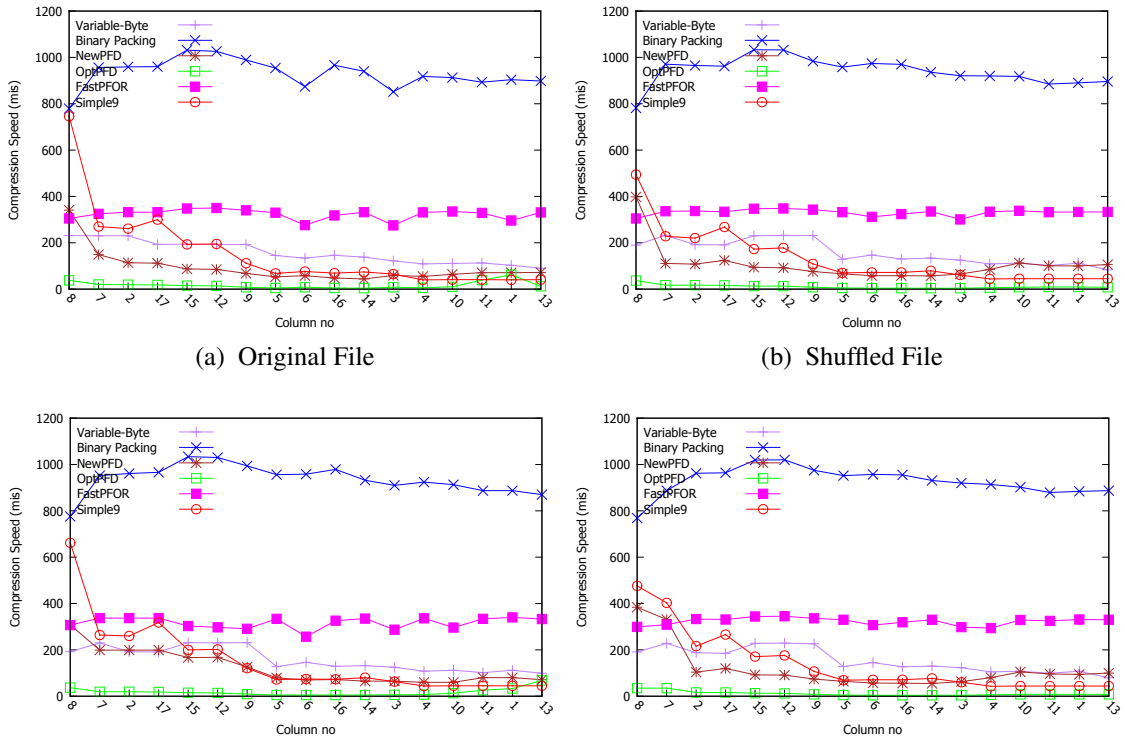
(d) Sort Low Cardinality Column (column 7)

Figure 6.8: Column-wise compression speed for SSB of frequency coded file

6.5 Experimental Evaluation

In our study, we divided our experiment in two cases. First, we tried to compare different compression schemes with synthetic data. Our other approach is to test the compression schemes with real data sets so that we can evaluate the performance of compression algorithms in a real world scenario.

In the case of a compressed database system, data is stored in compressed format on disk or in memory. Data compression helps to keep more data in main memory and cache. As a result, compression will reduce processing cost and save disk storage. This data needs to be decompressed during query processing to show the result to end users. Unfortunately, compression is not always beneficial. Decompression may use more CPU cycles to restore data. The performance gain by reducing I/O can be eaten up by higher CPU usage. As a result, decompression speed is also an important factor for performance



(a) Original File

(b) Shuffled File

(c) Sort High Cardinality Column (column 13)

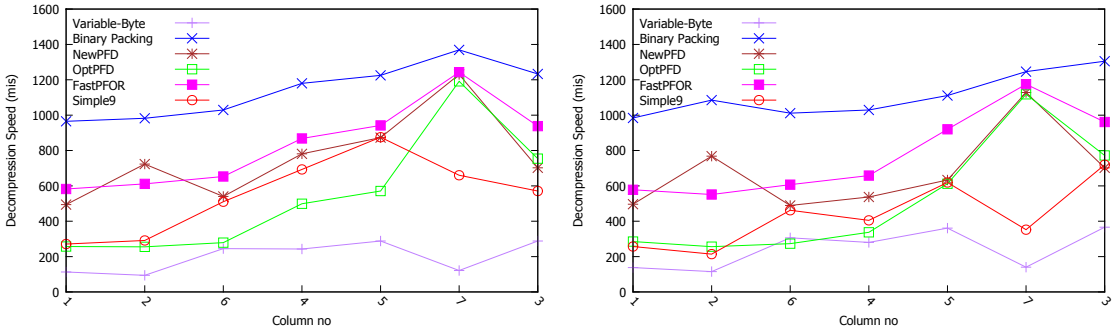
(d) Sort Low Cardinality Column (column 7)

Figure 6.9: Column-wise compression speed for SSB of random coded file

evaluation. In Section 2.4, we discussed the comparison characteristics to evaluate the compression methods. In this section, we will discuss the performance of different compression schemes based on the results of our study.

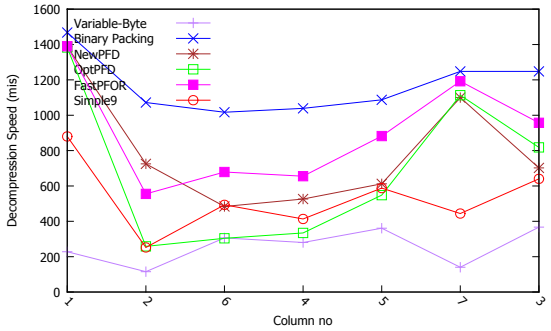
6.5.1 Comparison of different methods

Comparing compression is not an easy task. We identified comparison characteristics based on compression size, compression speed and decompression speed. In the case of database management systems, decompression speed and compression size are the most important characteristics. From the test result of our real dataset, we plot both the characteristics for all of the schemes that we used in our study in Fig. 6.15. According to the plot, we can summarize that the most competitive schemes are FastPFOR and OptPFOR with respect to compression size. However, Binary Packing is the most competitive because of

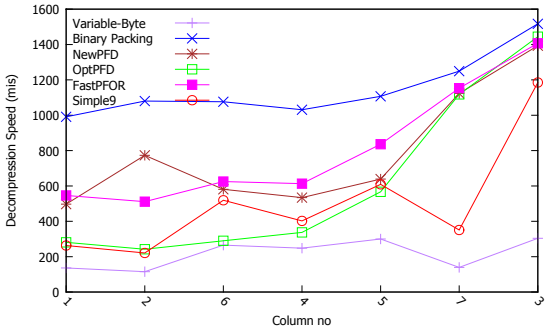


(a) Original File

(b) Shuffled File



(c) Sort High Cardinality Column (column 1)



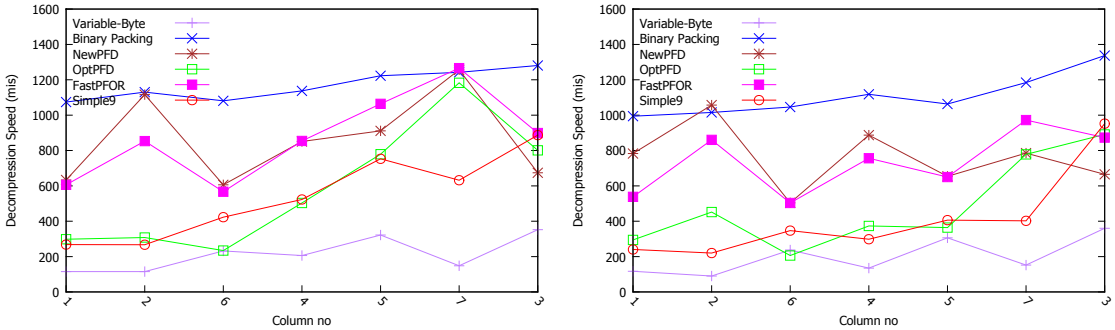
(d) Sort Low Cardinality Column(column 3)

Figure 6.10: Column-wise decompression speed for Census1881 of frequency coded file

its decompression speed and space efficiency. We see that it is less space-efficient compared to patched schemes but it is faster than patched schemes. Binary Packing uses about ≈ 1 extra bit compared to our most space-efficient schemes such as FastPFOR/OptPFD. We believe that the good performance of a scheme like Binary Packing can be explained by its relative low count of mispredicted branches which helps with instruction pipelining. In this thesis, we do not include processor-level metrics such as cache faults and branching. This could be the subject of future work.

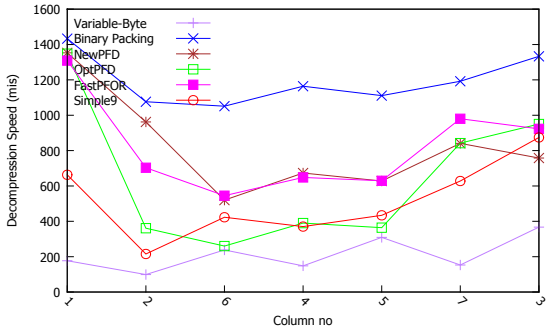
6.6 Effect of CPU family on performance

We claim that algorithms like Binary Packing are well suited to modern processors as they are unlikely to have many mispredicted branches and can benefit from pipelining. Of course, modern CPUs are sophisticated and an extensive study of how various CPU

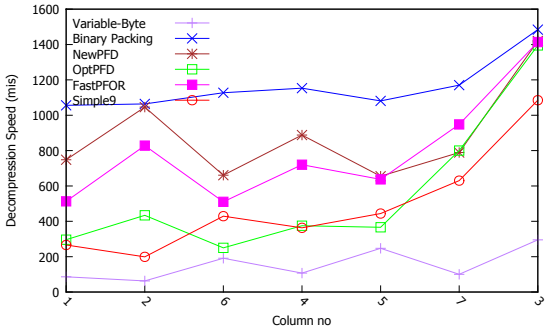


(a) Original File

(b) Shuffled File



(c) Sort High Cardinality Column (column 1)

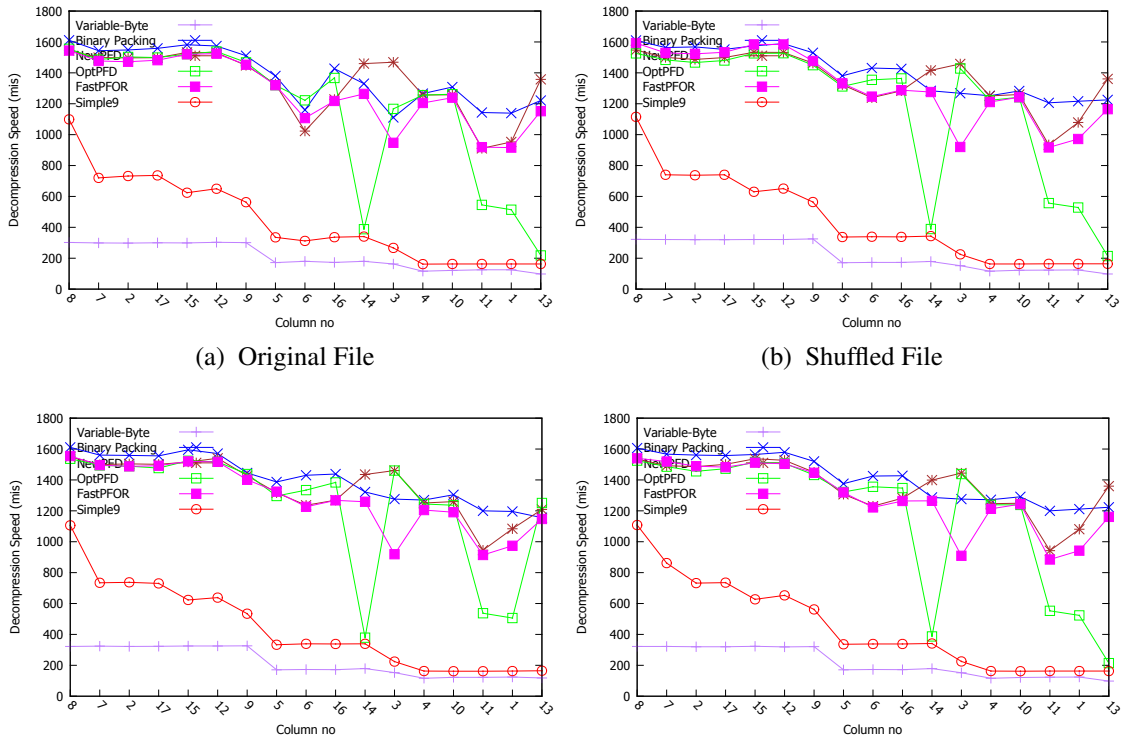


(d) Sort Low Cardinality Column(column 3)

Figure 6.11: Column-wise decompression speed for Census1881 of random coded file

features impact performance is outside our scope. We can, however, see how changing the CPU family impacts performance.

In our study we ran all our experiments on a machine equipped with Intel Core $i5$-2400 processor and 8GB of RAM (DDR3 Non-ECC 1333MHz) (See Section 4.1). To assess the effect of the CPU family, we ran our experiments on an another machine equipped with Intel Core $i7$-3770 processor and 16GB of RAM (DDR3 Non-ECC 1600MHz). This is a quad core processor, running at a clock speed of 3.5GHz (with a Max Turbo Frequency of up to 3.9GHz) and delivering eight-way multi-core processing via parallelism. It is difficult to compare the two machines. They have different clock speeds. Since the clock speed of the Core $i7$ processor is 10% higher than the Core $i5$, we expect better results for Core $i7$. Moreover, Core $i7$ processor has larger cache size than Core $i5$. Core $i5$ has 6MB of L3 cache and in the case of Core $i7$ we have 8MB of L3 cache. Both of our processors are equipped with quad core. We compare the basic specifications of these two machine



(a) Original File

(b) Shuffled File

(c) Sort High Cardinality Column (column 13)

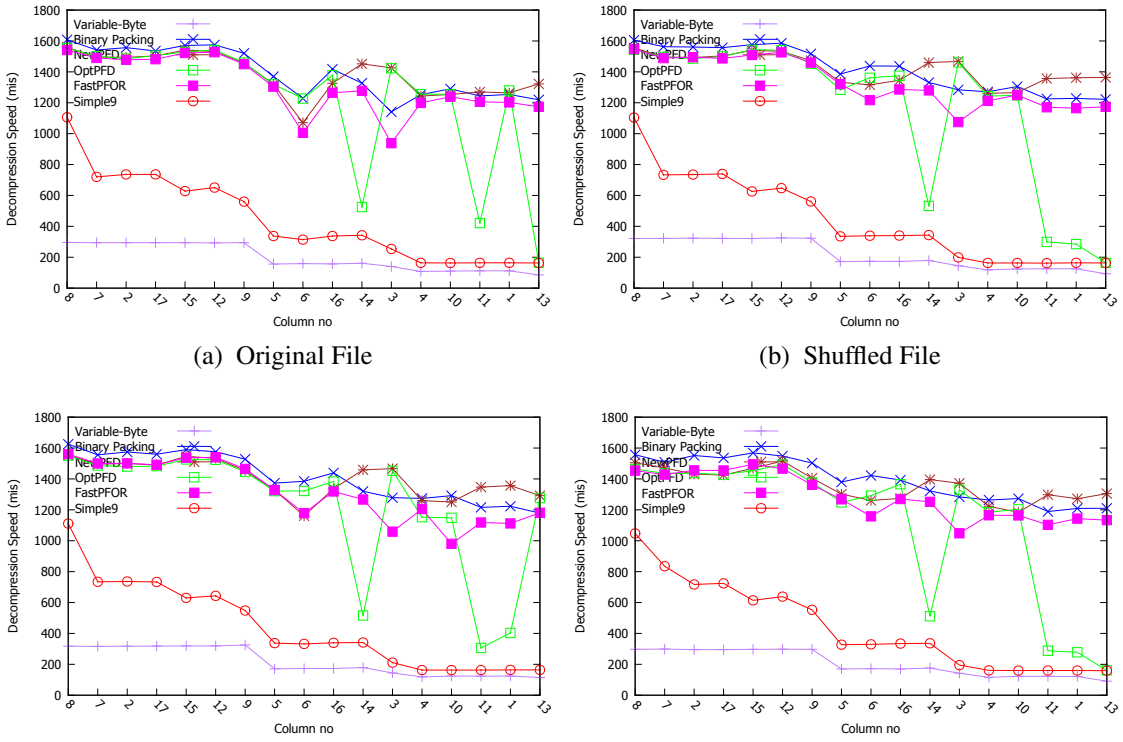
(d) Sort High Cardinality Column (column 7)

Figure 6.12: Column-wise decompression speed for SSB of frequency coded file

in the Table 6.14.

We found similar results for the compression ratios. Indeed, compression ratio does not depend on CPU, it depends on the algorithm that we are using. So, we did not plot for compression ratio. However, in the case of compression speed we got significant improvement for Core $i7$. In some cases we got 2-fold improvement for compression speed (See Fig. 6.16). In the case of decompression speed there is no major gain. We found around 10–20% gain (See Fig. 6.17), which can be explained by better clock speed and cache size. It is the compression speed that gains from the better CPU (i7). We theorize that it is the better CPU cache size that boosts the compression speed. In our implementation, we did not use parallelism. So, thread performance will not influence the speed. However, the CPU cache size reduces page faults which is an expensive operation.

In the next Chapter, we will summarize our study. We discuss the performance of different compression schemes based on different compression characteristics. Additionally, we



(a) Original File

(b) Shuffled File

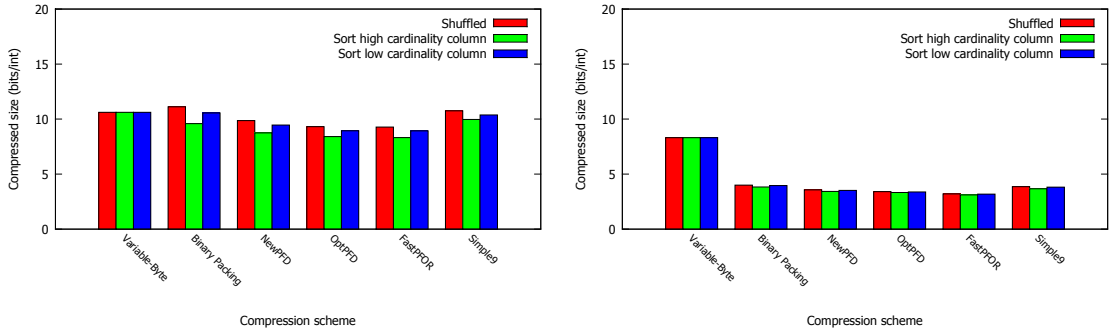
(c) Sort High Cardinality Column (column 13)

(d) Sort Low Cardinality Column (column 7)

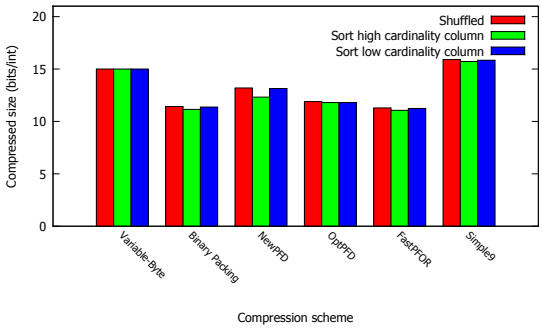
Figure 6.13: Column-wise decompression speed for SSB of random coded file

Table 6.14: Comparison between two machines

	Intel Core $i7$-3770	Intel Core $i5$-2400
	Processor's Specifications	
No of Cores	4	4
No of Threads	8	4
Clock Speed	3.5GHz	3.1GHz
Max Turbo Frequency	3.9GHz	3.4GHz
Cache	8MB L3	6MB L3
Max Memory Size	32GB	32GB
Memory Types	DDR3-1333/1600	DDR3-1066/1333
No of Memory Channels	2	2
	RAM Specifications	
Memory Clock Speed	1600MHz	1333MHz
Peak Transfer Rate	12.8GB/s	10.6GB/s



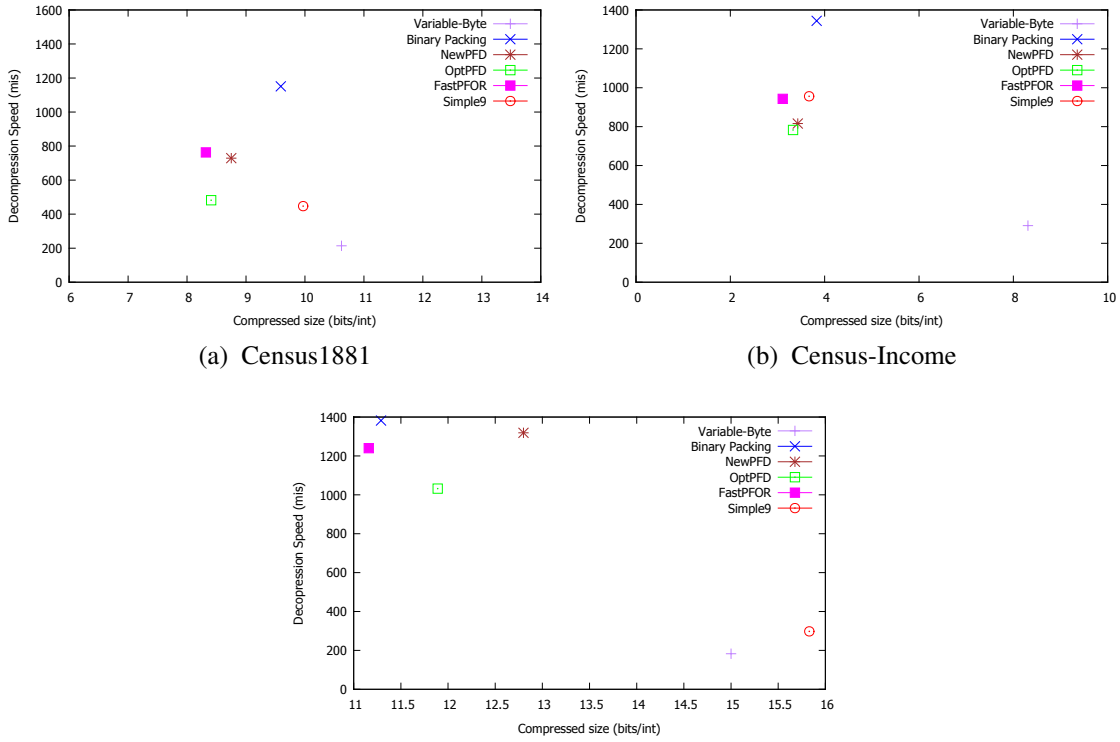
(a) Census1881 (b) Census-Income



(c) SSB

Figure 6.14: Histogram of compression size (bits/int)

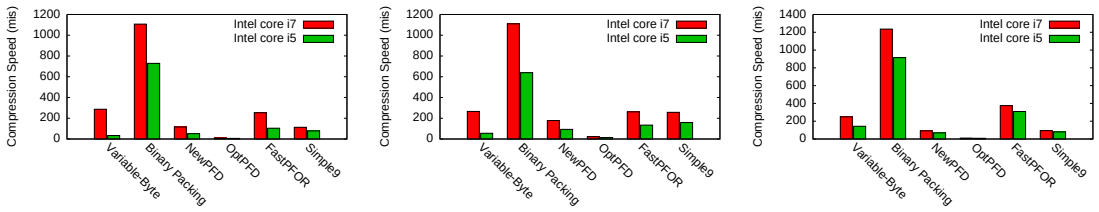
discuss some possible extensions of our study.



(a) Census1881 (b) Census-Income

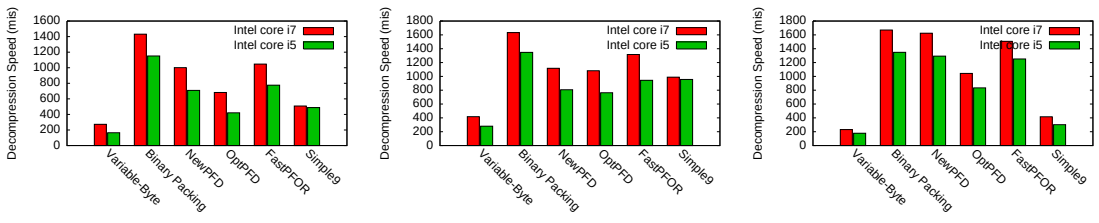
(c) SSB

Figure 6.15: Scatter plots comparing different schemes on decompression speed (mis) and compression size (bits/int)



(a) Census1881 (b) Census-Income (c) SSB

Figure 6.16: Histogram of compression speed (mis) on different processor



(a) Census1881 (b) Census-Income (c) SSB

Figure 6.17: Histogram of decompression speed (mis) on different processor

Chapter 7

Conclusions and Future Work

In this Chapter, we summarize our work and review our experimental results and the contribution of our study. Moreover, we explore the possible extension of our work.

7.1 Summary

We are interested in measuring the performance of fast integer compression over tables. Our main criteria to evaluate different compression schemes are compression ratio, compression speed and decompression speed. We are mostly concerned about the compression ratio and decompression speed.

In Chapter 1, we discussed the effect of compression on performance enhancement of relational database systems. Moreover, we discussed different preliminaries of database compression in Chapter 2 and pros and cons of different compression schemes in Chapter 3. We have tested different compression schemes on synthetic data and also on real datasets. We recorded the results from our experiments on synthetic data in Chapter 5 and real data in Chapter 6.

7.2 Contributions

Various authors have discussed patched schemes in previous studies but very few of them considered Binary Packing. However, Ann and Moffat [3] achieved good results for speed of decompression with Binary Packing. Delbru et al. [11] also recorded good results for Binary Packing over other schemes. They found good speed of compression and decompression using highly-optimized routines which avoid branching conditions. We use similar routines though we ran our experiments in Java and not C and we used database tables and not inverted indexes. Our results are consistent with previous findings: Binary Packing is fast and it offers good compression.

According to the nature of the database, we can use different compression schemes. The scheme which is beneficial for data warehouse or OLAP processing might not produce the same results in OLTP processing. In the case of data warehouse or OLAP processing, there will be few updates on databases and most of the queries will be I/O bound as these queries need to retrieve information from huge amounts of data. As a result, it would be prudent to use a compression scheme with high compression ratio, because decompression time in CPU will be negligible over I/O bound operations. In contrast, for OLTP processing, we have to consider both compression ratio as well as time to decompress the data in CPU because there will be frequent access of data due to updates and even for information retrieval. In this case, light-weight compression schemes will give good results.

Depending on the requirement of decompression speed we can choose any compression scheme from Fig. 6.15. If we need a higher decompression speed, we should choose Binary Packing. We got very fast speed of decompression for all of our real datasets. The cost of compression of Binary Packing is not very high. It takes approximately 1 more bit to compress any integer compared to our most space-efficient scheme FastPFOR.

In our study, we tried to evaluate the effect of row order as well. In the case of real data test in Chapter 6 we see that sorting of high cardinality columns results in very good compression size. Additionally, row order has a substantial effect on speed of compression

and decompression. Sorted columns can be compressed and decompressed faster than original or shuffled order. We found that sorting with a high cardinality column results in very good compression size in the case of whole table. It induced a good speed as well.

7.3 Future Work

We have tested the performance of different compression schemes in the case of some table values by compressing columns and merging the compressed columns. In future we can evaluate the outcome of these schemes by incorporating a query engine to asses real world performance. We are interested in studying the use of parallelism and the use of Single Instruction Multiple Data (SIMD) instructions of modern CPUs in compression algorithms since CPU technology is improving day by day in the case of speed and instruction sets. However, in our implementation we did not use parallelism but it is certainly a logical next step in this work. Further extensions could consider CPU processing as well. In this thesis, we do not include processor-level metrics such as cache faults and branching. This could be an another subject of future work.

Bibliography

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pages 671–682, New York, NY, USA, 2006. ACM.

[2] P. Alsberg. Space and time savings through large data base compression and dynamic restructuring. Proceedings of the IEEE, 63(8):1114 – 1122, August 1975.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. Information Retrieval, 8(1):151–166, 2005.

[4] V. N. Anh and A. Moffat. Index compression using 64-bit words. Softw. Pract. Exper., 40(2):131–147, 2010.

[5] C. Binnig, S. Hildenbrand, and F. 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, pages 283–296, New York, NY, USA, 2009. ACM.

[6] T. A. Bjørklund, N. Grimsmo, J. Gehrke, and O. Torbjørnsen. Inverted indexes vs. bitmap indexes in decision support systems. In CIKM '09, pages 1509–1512, 2009.

[7] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999. VLDB Endowment.

[8] S. Buttcher, C. Clarke, and G. Cormack. Information retrieval: Implementing and evaluating search engines. The MIT Press, 2010.

[9] S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In CIKM'07, pages 761–770, 2007.

[10] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. SIGMOD Rec., 30(2):271–282, May 2001.

[11] R. Delbru, S. Campinas, and G. Tummarello. Searching web data: An entity retrieval and high-performance indexing model. Web Semantics: Science, Services and Agents on the World Wide Web, 10:33–58, 2012.

[12] W. Effelsberg and T. Haerder. Principles of database buffer management. ACM Transactions on Database Systems, 9(4):560–595, 1984.

[13] P. Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2):194–203, 1975.

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In ICDE '98, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[15] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In Data Engineering, 1998. Proceedings., 14th International Conference on, pages 370 –379, Feb. 1998.

[16] G. Graefe and L. Shapiro. Data compression and database performance. In Applied Computing, 1991., [Proceedings of the 1991] Symposium on, pages 22 –27, Apr. 1991.

[17] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. CoRR, abs/1208.0225, 2012.

[18] S. Hettich and S. D. Bay. The UCI KDD archive. http://kdd.ics.uci.edu (checked 01-02-2012), 2000.

[19] S. J and N. Drach-Temam. Memory bandwidth: The true bottleneck of simd multimedia performance on a superscalar processor. In Euro-Par 2001,Parallel Processing, Lecture Notes in Computer Science, volume 2150, pages 439–447, Berlin / Heidelberg, 2001. Springer.

[20] O. Kaser and D. Lemire. Attribute value reordering for efficient hybrid OLAP. Information Sciences, 176(16):2304–2336, 2006.

[21] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. Software: Practice and Experience, 43, 2013.

[22] D. Lemire and O. Kaser. Reordering columns for smaller indexes. Information Sciences, 181(12):2550–2570, June 2011.

[23] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. Data & Knowledge Engineering, 69(1):3–28, 2010.

[24] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores. In DaWak '10, pages 117–129. Springer, Berlin, Heidelberg, 2010.

[25] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. Information Retrieval, 3(1):25–47, 2000.

[26] W. Ng and C. Ravishankar. Block-oriented compression techniques for large statistical databases. IEEE Transactions on Knowledge and Data Engineering, 9(2):314–328, 1997.

[27] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In Performance Evaluation and Benchmarking (LNCS 5895), pages 237–252, Berlin, Heidelberg, 2009. Springer.

[28] M. Poess and D. Potapov. Data compression in Oracle. In VLDB'03, Proceedings of the 29th International Conference on Very Large Data Bases, pages 937–947, Berlin, Germany, 2003. VLDB Endowment.

[29] Programme de recherche en démographie historique. PRDH 1881. http://www.prdh.umontreal.ca/census/en/main.aspx, 2009. last checked 2011-12-19.

[30] V. Raman and G. Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In VLDB'06, Proceedings of the 32nd International Conference on Very Large Data Bases, pages 858–869, 2006.

[31] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using simd instructions. In Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN '10, pages 34–40, New York, NY, USA, 2010. ACM.

[32] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In SIGIR'02, pages 222–229, New York, NY, USA, 2002. ACM.

[33] A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. Simd-based decoding of posting lists. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.

[34] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: a column-oriented DBMS. In VLDB'05, Proceedings of the 31st International Conference on Very Large Data Bases, pages 553–564, San Jose, CA, USA, 2005. VLDB Endowment.

[35] F. Transier and P. Sanders. Engineering basic algorithms of an in-memory text search engine. In ACM Transactions on Information Systems, pages 2:1–2:37. ACM, Dec 2010.

[36] J. Walder, M. Krátký, R. Bača, J. Platoš, and V. Snášel. Fast decoding algorithms for variable-lengths codes. Inf. Sci., 183(1):66–91, Jan. 2012.

[37] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. SIGMOD Record, 29(3):55–67, September 2000.

[38] Wikipedia. Cache algorithms. http://en.wikipedia.org/wiki/Cache_algorithms, 2013. Last checked on 06/09/2013.

[39] Wikipedia. Lossless compression. http://en.wikipedia.org/wiki/Lossless_compression, 2013. Last checked on 06/09/2013.

[40] Wikipedia. Lossy compression. https://en.wikipedia.org/wiki/Lossy_compression, 2013. Last checked on 06/09/2013.

[41] Wikipedia. Memory hierarchy. http://en.wikipedia.org/wiki/Memory_hierarchy, 2013. Last checked on 06/09/2013.

[42] Wikipedia. Superscalar CPU. http://en.wikipedia.org/wiki/Superscalar, 2013.

[43] H. Williams and J. Zobel. Compressing integers for fast file access. The Computer Journal, 42(3):193–201, 1999.

[44] I. H. Witten, A. Moffat, and T. C. Bell. Managing gigabytes (2nd ed.): compressing and indexing documents and images. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[45] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. Commun. ACM, 30(6):520–540, June 1987.

[46] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In WWW '09, pages 401–410, 2009.

[47] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In WWW '08, pages 387–396, 2008.

[48] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In ICDE '06, Washington, DC, USA, 2006. IEEE Computer Society.