# Indexing Compressed Text

by

Meng He

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2003

# Abstract

As a result of the rapid growth of the volume of electronic data, text compression and indexing techniques are receiving more and more attention. These two issues are usually treated as independent problems, but approaches of combining them have recently attracted the attention of researchers.

In this thesis, we review and test some of the more effective and some of the more theoretically interesting techniques. Various compression and indexing techniques are presented, and we also present two compressed text indices. Based on these techniques, we implement an compressed full-text index, so that compressed texts can be indexed to support fast queries without decompressing the whole texts. The experiments show that our index is compact and supports fast search.

# Acknowledgements

I am thankful to anyone who supported me during my master's studies. I would like to thank especially Professor J. Ian Munro, my supervisor, who have guided and aided me throughout my studies. I also thank Professor D. Toman and Professor A. Lopez-Ortiz, who served as readers of the thesis and offered several useful suggestions. Last but not least, I would like to thank my parents for their support and encouragement over the years.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation and Objectives

Text compression deals with "exploiting redundancies in the text to represent in less space" [14]. Interest in text compression continues to grow. Text compression saves computer storage space. It also accelerates data transmission between computers and reduces the transmission cost. Therefore, storing texts in compressed form is an attractive and economical choice, and sometimes, mandatory. Besides, text compression has a wide range of applications in some fields such as information retrieval, which we will discuss in Chapter 4, and data encryption [37].

Text Indexing deals with "building a data structure that will allow quick searching of the text" [9]. Computers are used mostly for the management and manipulation of textual data. As a result of the rapid growth of the amount of textual data available in databases and on the World Wide Web, traditional scan-based algorithms that perform sequential search on the entire text collection to find a keyword has become insufficient for many applications that require short response time. To facilitate searching, various indexing techniques have been developed. They usually pre-process the text and store auxiliary information in some data structures called indices, which will be used later to enable fast searching.

As the amount of textual data stored in compressed format increases, the problem of searching through compressed texts arises. The naive solution is to decompress the text first and then use a scan-based algorithm to search the result. However, this costs too much time for large texts. Some researchers therefore focused on *compressed matching problem* (i.e. designing algorithms that perform string matching through compressed text without decompression). Some of the research are on searching LZ77 compressed texts [24], LZ78 compressed texts [6] and Huffman coded texts [45, 46]. Although these algorithms improves the query efficiency, they still rely on a full linear scan on the compressed text and is inefficient over large texts. Thus approaches for combining indexing and compression techniques to facilitate searching in compressed text are nowadays receiving more and more attention.

The purpose of this thesis is to design and experiment with techniques for indexing compressed texts. To be more specific, we aim at designing techniques that store texts in compressed form, while at the same time the texts are appropriately indexed so that we can efficiently perform queries on them without decompressing the whole texts. Various related techniques will also be studied to achieve this goal.

## 1.2 Road Map

The rest of the thesis are organized as follows:

- **Chapter 2** gives a survey of various text compression algorithms. We also implement and test a Burrows-Wheeler compression algorithm for future experiments. During our experiments, we compare various techniques so that our resulting algorithm is both good in compression efficiency, and fast in compression speed.

- **Chapter 3** reviews some text indexing techniques. Two well-know indexing techniques are studied. They are inverted files and suffix trees. We also give a survey of various research work to reduce the space cost of suffix trees.

- **Chapter 4** is the most substantive contribution. It presents two techniques to combine compression and indexing techniques, which are SASE and the opportunistic index. We also design a new MTF method to implement the opportunistic index. Based on one implementation of the opportunistic index, we design and implement a technique to index compressed texts, determine its parameters, and improve its performance by adopting a caching technique.

- **Chapter 5** presents some conclusions and suggestions for future work.

# Chapter 2

# Text Compression

## 2.1 Introduction

There are two general approaches to compression: statistical coding and dictionary coding [14]. In statistical coding, we make use of the fact that different symbols usually occur at different frequencies. We assign shorter codes to symbols that occur more frequently, and longer codes to less frequently used symbols. In dictionary coding, we make use of the fact that certain groups of consecutive characters (i.e. phrases) occur more than once. We assign a code to a certain phrase.

During the past few decades, more and more coding and transformation algorithms have been developed for compression. From the many existing compression techniques, we will review those that are relevant to our topic, especially the Burrows-Wheeler Transform [18]. The purpose of this chapter is to design an algorithm with relatively "simple" methods, that achieves both compression performance close to more complex coders and fast compression speed. This is for our future study of combining compression and indexing techniques.

The contents of this chapter are organized as follows. In Section 2.2, we describe some basic terminology. In Section 2.3, we review some relative compression techniques. Finally, in Section 2.4, we implement and test some compression algorithms that will be used in further experiments.

## 2.2 Basic Terminology

**Text** We use the array $T[0, u-1]$ to denote a string or text of length $u$, which is drawn from a constant-size alphabet $\Sigma$. We assume $\Sigma = \{0, 1, 2, ..., |\Sigma| - 1\}$ unless otherwise specified. We denote the $i$'th symbol or character of the text $T$ as $T[i]$. We denote the probability of symbol $j$ in the text as $p_j$. $|\alpha|$ denotes the length of the string $\alpha$. In this paper, we all use $\lg n$ to denote $\log_2 n$.

**Entropy** We define *the zeroth order empirical entropy* of the text $T$, or *text entropy*, as

$$H_0(T) = \sum_{i=0}^{|\Sigma|-1} (p_i \lg \frac{1}{p_i}) = - \sum_{i=0}^{|\Sigma|-1} (p_i \lg p_i),$$

where lg denotes the logarithm base 2 and $0 \lg 0$ is interpreted as 0. An ideal compressor that uses $\lg \frac{1}{p_i}$ (or $-\lg p_i$) bits to code the symbol $i$ can compress the text $T$ to $|T|H_0(T)$ bits. This is the maximum compression we can achieve using a uniquely decodable coding scheme in which each alphabet symbol is assigned a fixed code word.

We define *the $k$-th order empirical entropy* of the text $T$ as

$$H_k(T) = \frac{1}{|T|} \sum_{w \in \Sigma^k} |w_T| H_0(w_T),$$

where $w_T$ is the string which is a concatenation of all the single characters immediately following one of the occurrences of $w$ in $T$ (For example, if $T = aabacabbabc$ and $w = ab$, then $w_T = abc$). The value $|T|H_k(T)$ is a lower bound in bits to the compression we can achieve using for each character a code that depends on only the $k$ characters preceding it.

## 2.3 Compression Techniques

### 2.3.1 Huffman Coding

One of the oldest and best known compression techniques is Huffman coding [34]. With Huffman coding, we maintain a forest of binary trees representing disjoint subsets of $\Sigma$ with weight equal to the sum of probabilities of the elements in the subset. Initially each symbol corresponds to a tree containing one node. We then select two trees with lowest weights and make their roots the left and right children of a new node thus combining the trees into a single tree whose weight is the sum of the two individual weights. We repeat the above process until a single tree remains. This is called the Huffman code tree. To form the code for any particular symbol, we traverse the tree from the root to that symbol, recording 0 for a left branch and 1 for a right branch. This results in a *prefix code*, that is one in which the code for no symbol is a prefix of the code for another. This feature makes it possible to encode a string as the concatenation of the codes for its individual symbols.

Huffman coding generates “minimum-redundancy” codes in the sense that the Huffman code produces the minimum average length prefix codes for the elements of an alphabet. The length of the code for symbol $i$ is approximately $-\lg p_i$. If each symbol in the alphabet occurs at a probability which is a power of 1/2, the average code length is exactly equal to the text entropy. It can be shown that the redundancy of Huffman codes (i.e. the average code length less the entropy), is bounded by $p + \lg[2(\lg e)/e] = p + 0.086$, where p is the probability of the most likely symbol [14]. As suggested, Huffman codes are used to compress strings. In this context, they are not optimal as:

1. The fact that they are prefix codes essentially forces the length of a code for a symbol to be “rounded up”. For example, if $\Sigma = \{a, b\}$, $p_a = 0.99$ and $p_b = 0.01$, we still use one bit per character.

2. No advantage is taken of higher order effects such as correlations between characters in a sequence. For example, the phrase “qu” in English texts.

Various methods have been developed to improve Huffman coding. A recent one is multiple-tables Huffman coding, or MTH, proposed by Wheeler[60]. It is motivated by the fact that an individual symbol will occur with different frequencies in various parts of the text. To take the advantage of this fact, Wheeler suggested using multiple tables. To be more specific, we keep several Huffman coding tables, and divide the whole text into small segments of the same size. For each segment, we use the table that is best for it (i.e. that achieves the best compression performance) to compress it. One algorithm used in [52] that can generate the multiple coding tables is described in the following:

1. Divide the symbols in the alphabet into $s$ groups, such that the sum of frequencies of the symbols in each group is roughly equal, where $s$ is the number of Huffman coding tables we are going to generate.

2. Assign a coding table to each group generated in Step 1. In each coding table, the symbols that occur in the corresponding group are assigned codes of constant length $l$, while those that do not occur are assigned codes of constant length $g$, where $g > l$. For example, we can set $l = \lceil \lg m \rceil$, where $m$ is the maximum value of the sizes of the groups, and $g = \lceil \lg |\Sigma| \rceil$. Note that we do not have to assign actual codes to the symbols at this step, but, instead, we only need the code lengths for future computation, and thus we can assign arbitrary constants to $g$ and $l$, as long as $g > l$. Now we have an initial set of coding tables.

3. Divide the original text into small segments of constant size.

4. To each coding table, we assign an array of size $|\Sigma|$, whose elements will store the frequencies of the alphabet symbols, but we initialize the elements of the arrays to be 0.

5. For each segment, we find the coding table that codes it most efficiently (On the initial pass, this will be the table corresponding to the alphabet subset that occurs most often in the segment). We increment the elements of the array that stores symbol frequencies for the selected coding table according to

how many times each symbol occurs in that segment. For example, if symbol $i$ occurs $n_i$ times, then we increment the $i$'th element of that array by $n_i$.

6. Generate Huffman coding tables for each of the groups according to the symbol frequencies associated to the current set of coding tables, which will be used as the set of coding tables in future iterations.

7. Iterate Step 4-6 up to some constant number of times and return the resulting set of Huffman coding tables.

After the above steps, we have a set of coding tables. To encoding the original text, we also need to store a selector for each group to indicate the specific coding tables we choose to encode it. The sequence of selectors can be encoded with the move-to-front encoding scheme, which will be further discussed in Section 2.3.4.

### 2.3.2 Arithmetic Coding

The basic idea of arithmetic coding scheme [62] is to represent a text by an interval of real number between 0 and 1. Before compressing, a model for alphabet of the text is built in which each symbol in the alphabet is assigned a range according to its probability of occurrence, and the range of the text is initialized to $[0, 1)$. During compressing, we process each symbol in the text sequentially and narrow the range of the text according to the range assigned to the symbol in the model.

For example, suppose we are going to encode $T = abaac$ with arithmetic coding. The symbols $a$, $b$, $c$ have frequencies of 0.6, 0.2, and 0.2, respectively, and are therefore assigned ranges of $[0, 0.6)$, $[0.6, 0.8)$, and [0.8, 1.0), respectively. Initially, the range of the text is $[0, 1)$. After seeing the first symbol $a$, the range is narrowed to $[0, 0.6)$. After seeing the second symbol $b$, the new range is narrowed to the fourth one-fifth of the original, since $b$ is assigned a range of $[0.6, 0.8)$. This results in a new range of $[0 + (0.6 - 0) * 0.6, 0.6 + (0.6 - 0) * 0.8) = [0.36, 0.48)$. Similarly, the next three symbols in the text narrow the range to $[0.36, 0.432)$, $[0.36, 0.4032)$, $[0.39456, 0.4032)$, respectively. This way we encode the text as $[0.39456, 0.4032)$. To decode the text, we first decide that the range $[0.39456, 0.4032)$ is in the range assigned to symbol $a$, and therefore the first symbol of the text is $a$. Then we decide that it is in the fourth one-fifth of the range assigned to symbol $a$, and therefore the second symbol is $b$. We repeat the process until we have the original text. Actually, we just need to use any number among the range $[0.39456, 0.4032)$, such as 0.39457, to encode the text, and we can decode the text by deciding which range contains the number. However, this means we need to append a end-of-file symbol to the original text so that we know when we should stop narrowing the range in the process of decoding.

Theoretically, the number of bits needed to encode a text with arithmetic coding is the same as the entropy of the text. However, in practice, some factors will make the coding less efficient. First, we need to add an additional symbol to indicate the end of the file. Second, for the sake of coding speed, we cannot use infinite-precision arithmetic. Instead, we usually use integer arithmetic instead, which makes the computation less accurate. Last, to prevent overflow of the variables that store the symbol frequencies, we have to adopt some scaling techniques which will further lower the accuracy. Despite of all these factors, arithmetic coding is superior to the original Huffman coding in coding efficiency, although it codes the texts at a slower speed than the latter since it requires much computation.

### 2.3.3 Ziv-Lempel Coding

Most practical dictionary coding algorithms belong to a family of algorithms, known as Ziv-Lempel coding (abbreviated as LZ coding), derived from Ziv and Lempel's work [63, 64]. These algorithms are based on the idea of replacing the strings in the text with a pointer to where they have occurred earlier in the text. This family of algorithms are generally derived from one of the two different approaches published in [63] and [64], namely LZ77 and LZ78, respectively. Some variants in the LZ77 family are: LZR [50], LZSS [12], LZB [13] and LZH [16]. Some variants in the LZ78 family are: LZW [59], LZC [54], LZT [55], LZMW [43] and LZJ [36]. LZFG [29] is based on both LZ77 and LZ78.

One particularly effective variant is that of Welch labeled LZW [59], and is derived from LZ78. In this algorithm, we first initialize a list of phrases (i.e. dictionary) which contains all the alphabet symbols. Then we parse the input text into phrases, where each phrase is the longest matching phrase seen previously plus one symbol. We output the index to the longest matching phrase in the list, and add the new phrase into the list. The extra symbol is encoded as the first symbol of the next phrase.

### 2.3.4 Move-to-front Coding

Move-to-front coding scheme [15] is based on the move-to-front heuristic for self-organizing sequential search on variable-length integer coding. In this scheme, the encoder maintains a word list (denoted as MTF list) that is initialized with all the words in the text to be compressed using the move to front heuristic: after a word is used, the number of the words preceding it in the list (which is a integer) is encoded and appended to the end of the compressed text, and then the word is deleted from its current position in the list and moved to the front of the list. When encoding the integers, we should use a variable-length encoding scheme in which small integers are encoded in fewer bits than larger integers are. One encoding method is to prefix the binary representation of the integer $i$ with $\lfloor \lg i \rfloor$ 0's. The decoder maintains an identical word list using the same heuristic for decoding. The most important property of this scheme is that a recently used word is near to the front of the list and therefore has a short code. It was proved that this scheme never performs much worse than Huffman coding and can perform substantially better [15].

There are many ways to divide texts into words. One simple method is to classify each symbol as a word, and we will use this method in future experiments. There are also many variations of the move-to-front schemes to maintain the word list, and we will introduce those that we need in our experiments. We label the original move-to-front scheme as MTF. Some variations are (assume the MTF list starts at position 0):

- *MTF1*: When we encounter a symbol that is not at the front of the MTF list, if it is at position 1 of the list, then we move it to the front, otherwise we move it to position 1. This scheme is suggested in [10], whose author label the scheme as MTF1 in [11].

- *MTF2*: When we encounter a symbol that is not at the front of the MTF list, if it is at position 1 of the list and the last output code is not 0, then we move it to the front, otherwise we move it to position 1. This scheme is suggested and labeled MTF2 in [11]. Actually, in this scheme, a symbol is moved to the front of the MTF list if and only if we encounter it twice in a row.
- *MTF3*: Move a symbol to the front of the MTF list if and only if we encounter it three times in a row. We label the scheme as MTF3.

Other options are clearly the simple exchange with the element ahead of the one in question and to move half way to the front of the list.

### 2.3.5 Run-Length Coding

Run-Length Coding, or RLE is a technique to compress a repeating string of symbols. The repeating string is called a *run*. A simple RLE coding scheme is to encode a run of symbols into two bytes, namely a *count* bye which indicates the length of the run, and a *symbol* byte which is the symbol in this run. This coding method is suitable to compress data that contains many runs, such as images, video and audio. There are many variants of this coding scheme designed to improve the coding efficiency.

RLE is often combined with other compression algorithms to achieve good compression. Therefore, there are also many variants of the scheme designed for many special situations. We will discuss one interesting situation that will be encountered in our experiments. In this situation, we need to encode texts that contain many long runs of the symbol 0, and we will only use RLE to encode the runs of 0's, and then compress the result with another coding method. Some schemes are:

- *RLE1*: When we encounter a run of $n$ 0's in the text ($n$ can be 1 which means we encounter a single 0), we first output the symbol 0, then we treat the value $n-1$ (since, obviously, $n$ cannot be 0) as a character and output it. If $n-1$ is greater than 255, which cannot be stored in one byte, we then divide the run into more than one sub-runs each of which is of length less than 256, and encode them.
- *RLE2*: When we encounter a single 0 in the text, we output it directly. But when we encounter a run of 0's whose length is $n$ where $n > 1$, we first output two 0's, then we treat the value $n-2$ as a character and output it. If $n-2$ is greater than 255, which cannot be stored in one byte, we then divide the run into more than one sub-runs each of which is of length less than 256, and encode them.
- *RLE3*: We append another 256 symbols, namely $|\Sigma|$, $|\Sigma|+1$, ... ,$|\Sigma|+255$ to the original alphabet, each of which represents runs of 0's of length 1, 2, ..., 256, respectively. When we encounter a run of 0's in the text, we output the corresponding appended symbol. Again, if we encounter a run of 0's whose length is greater than 256, we divide it into more than one sub-runs each of which is of length less than 256, and encode them.

- *RLE4*: This method was proposed by D. Wheeler [25]. In this algorithm, we increase the size of the alphabet by 1, and use symbol $k$ to represent symbol $k-1$ in the original alphabet, where $k > 1$. The symbol 0 and 1 are used to encode the length of runs of 0's and are given digit weight of 1 and 2. For example, a sequence of bits $x_0x_1x_2...x_n$ (least significant bit first) represents the length

$$\sum_{i=0}^{n-1}(1+x_i)2^i = \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} x_i 2^i = (2^n - 1) + \sum_{i=0}^{n-1} x_i 2^i$$

To further improve the coding efficiency, we can omit the most significant bit in the sequence since the length can never be 0. One efficient way of generating the coding of the length is to increment the length by one and encode the result as an ordinary binary number with least significant bit first, ignoring the most significant bit. In this method, the symbol 0 and 1 indicates the start of a run, which is terminated by symbols other than these two.

### 2.3.6 Burrows-Wheeler Compression

Burrows-Wheeler Compression algorithm was first discovered by Wheeler in 1983 and was proposed publicly by Burrows and Wheeler [18] in 1994. The algorithm achieves speed comparable to Ziv-Lempel coding algorithms, but obtains compression close to the best statistical modeling techniques. Unlike most other compression algorithms, this algorithm does not process the input text sequentially, but instead processes it block by block. The size of the blocks could be arbitrary, and in some cases, it could be the size of the entire text. Note that the size of the block influences both compression efficiency and compression speed: larger block size improves the compression efficiency but slows down the compression speed. In practice, we often need to choose moderate block size. To each block, we apply a reversible transformation to form a new block whose symbols are the same as those of the original blocks. The transformation is named Burrows-Wheeler transform, or BWT. The BWT tends to group the same symbols together and therefore produces a new block that is easier to compress with some fast locally-adaptive algorithms, such as MTF in combination with Huffman or arithmetic coding. We will first discuss the algorithms and properties of BWT and then introduce the general steps for BWT-based algorithms.

**The Burrows-Wheeler transform.** The BWT consists of two transformations: a forward transformation which is applied to a block of text before compressing it, and a backward transformation which performs the inverse operation. To illustrate the BWT, we give a running example, using the classical text $T[0, 10] = mississippi$ (an example taken from [25]) as the input text. We assume that the block size is equal to the size of the text so that we only process the text once. We first describe the forward transformation processing $T[0, u-1]$:

1. Append to the end of $T$ an EOF symbol (denoted by #) smaller than any other alphabet symbol.
   In our example, we get $T\# = mississippi\#$.

2. Form a conceptual $(u+1) \times (u+1)$ matrix $M$ whose elements are symbols, and whose rows are the cyclic shifts of $T\#$, sorted in lexicographic order.

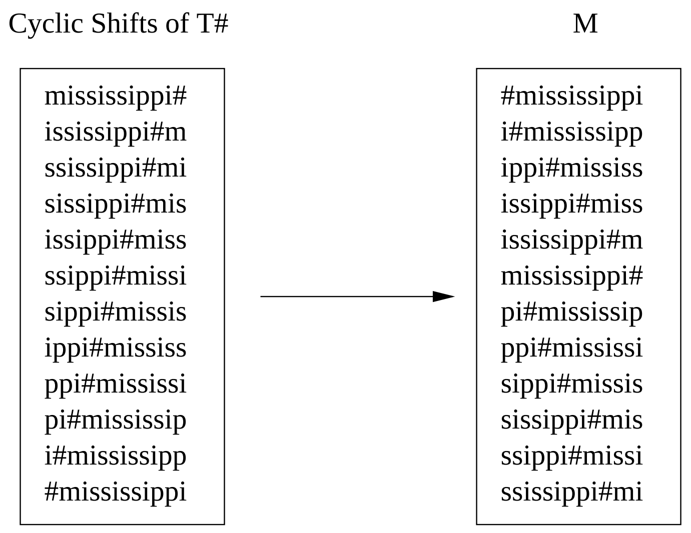

Figure 2.1: Sorting the cyclic shifts of $T\#$ to construct the matrix $M$ for the text $T = mississippi$.

Please refer to Figure 2.1 for the processing of our example.

3. Return the last column of $M$, which is the transformed text $L$.

   In our example, $L = ipssm\#pissii$.

Note that the process of sorting the cyclic shifts of $T\#$ is equivalent to the process of sorting suffixes of $T\#$. This is because the symbol # is smaller than any other alphabet symbol, and no character occurring after the symbol # is compared. Therefore, we can adopt algorithms for suffix sorting to perform the forward BWT. An experimental study of these algorithms can be found in [53].

A few easy-to-prove observations are crucial to design the backward BWT.

**Fact 1** *The matrix $M$ and the transformed text $L$ constructed by the forward BWT satisfy the following properties:*

a. *The first row of $M$ is always $\#T$.*

b. *The first column of $M$ (denoted by $F$) can be constructed from $L$ by lexicographically sorting the characters in $L$.*

c. *For the $i$th row of $M$, where $i = 0, 1, ..., u$, the characters $L[i]$ and $F[i]$ are its last and first characters, and $L[i]$ precedes $F[i]$ in $T\#$, except the row which ends with $\#$.*

d. *For any row $c\alpha$ of $M$, where $c$ is its first character and $\alpha$ is the rest of the string, we assume it is the $r$'th row of $M$ starting with $c$. Then we have the row $\alpha c$ is also the $r$'th row of $M$ ending with $c$.*

We also define a mapping among the rows of $M$, called Last-to-First mapping, or LF-mapping [26]. Given a row ending with a certain character $c$, we assume it is the $i$'th row of $M$, where $i = 0, 1, ..., u$. When we shift the row one character to the right to get a string that starts with $c$, we assume the new string is the $j$'th row of $M$. The LF-mapping shows the correspondence between such rows and we set $LF[i] = j$. Assume the first row that starts with $c$ is the $k$'th row in $M$, and $c$ occurs $r_i$ times in the prefix L[0, i-1], then according to observation (d) above, we have

$$LF[i] = k + r_i \tag{2.1}$$

Now we describe the backward BWT.

1. Construct $F$ by lexicographically sorting the characters in $L$ (see observation (b) above).

2. Compute the array $C[0...u-1]$, storing in $C[c]$ the number of occurrences of the characters #, 0, 1, ..., c-1 in $T\#$. It's obvious that $C[c]$ is the position of the first occurrence of $c$ in $F$, if $c$ occurs at least once in $T\#$, and we can make use of the fact to compute $C$ although $T$ is unknown to us by now.

3. Create an array $T[0..u-1]$ and set its last element to be the $L[0]$(see observation (a) above). Now we only know the suffix of length 1 of $T$. Set $s = 0$.

4. According to (2.1), we have $LF[s] = C[L[s]] + r_i$, where $r_i$ is the number of times character $L[s]$ occurs in the prefix $L[0, s-1]$ (set $r_i$ to 0 if $s = 0$). The character $LF[s]$ occurs before the currently known suffix of $T$ (see observation (c) above) and we set the value of the corresponding element of $T$. Now we set $s = LF[s]$.

5. Repeat Step 4 until all the elements of $T$ are computed. Return $T$.

**The BW-based compression algorithms.** Let $T^{bwt}$ denote the output of the BWT (i.e. the last column $L$ of $M$). As is seen in the steps of the forward BWT, the characters in $T^{bwt}$ are sorted according to the characters that appear after them in $T$. In other words, they are sorted according to their context. It is well-known that in most real texts, characters with the same or similar contexts tend to be the same. Therefore, the same characters tend to be grouped together in $T^{bwt}$ and this property is good for compression.

The BW-based compression algorithms are usually performed in the following steps:

1. Use a move-to-front coder, such as one of the MTF schemes in Section 2.3.4 to encode $T^{bwt}$. Let $T^{mtf}$ denote the result. Due to the nice property of $T^{bwt}$, $T^{mtf}$ are dominated by small numbers, especially 0's.

2. Encode each run of 0's in $T^{mtf}$ using one of the run-length coders discussed in Section 2.3.5. Let $T^{rl}$ denote the result.

3. Encode $T^{rl}$ with one of the statistical coders like Huffman coding or arithmetic coding.

Manzini [41] shows that there exists a constant $g_k$ such that the size of the compressed result does not exceed $5|T|H_k(T) + g_k \lg |T|$.

## 2.4 Implementation

We implemented the compression techniques discussed above and tested these techniques on the well-known Calgary Corpus [61].

### 2.4.1 Text Compression Without Burrows-Wheeler Transform

We first implemented and tested some algorithms to compress texts without the BWT. Please refer to Table 2.1 for the results. The column "Huffman" contains the results of text compression with Huffman coding. The column "MTH" contains the results of text compression with multiple-tables Huffman coding, using the implementation suggested in [52]. The column "arith" contains the results of

| File | Huffman | MTH | arith | LZW |
|---|---|---|---|---|
| *bib* | 5.239 | 5.167 | 5.234 | 3.872 |
| *book1* | 4.563 | 4.551 | 4.546 | 4.067 |
| *book2* | 4.825 | 4.708 | 4.777 | 4.538 |
| *geo* | 5.693 | 5.559 | 5.656 | 6.153 |
| *news* | 5.230 | 5.061 | 5.186 | 4.939 |
| *obj1* | 6.091 | 5.605 | 5.967 | 6.297 |
| *obj2* | 6.302 | 6.025 | 6.071 | 9.806 |
| *paper1* | 5.035 | 4.936 | 4.984 | 4.693 |
| *paper2* | 4.645 | 4.646 | 4.626 | 4.055 |
| *pic* | 1.664 | 1.656 | 1.166 | 1.095 |
| *progc* | 5.257 | 5.233 | 5.235 | 4.942 |
| *progl* | 4.812 | 4.623 | 4.759 | 3.899 |
| *progp* | 4.913 | 4.879 | 4.894 | 3.773 |
| *trans* | 5.579 | 5.220 | 5.492 | 4.316 |
| Avg. | 4.989 | 4.848 | 4.900 | 4.746 |
| Avg. (text) | 5.010 | 4.902 | 4.973 | 4.306 |

Table 2.1: Results of text compression (in bits/symbol) on Calgary Corpus without Burrows-Wheeler Transform.

text compression with arithmetic coding, and we followed the implementation in [62]. The column "LZW" contains the results of text compression with LZW, and our implementation is based on the sample code in [49]. Notice that the compression efficiency of our implementation for LZW can be further improved with more complicated techniques. The last row of the table stores the average of the results of compression on text files from Calgary Corpus (i.e. those except *geo*, *obj1*, *obj2* and *pic*), since text compression is our primary concern in this thesis.

From the results, we can see that MTH and arithmetic coding compresses better than the plain Huffman coding, while MTH outperforms arithmetic coding in most cases. LZW is more effective on files with many repetitive strings.

| File | Huffman | MTH | arith | LZW |
|---|---|---|---|---|
| *bib* | 2.386 | 2.361 | 2.298 | 2.455 |
| *book1* | 2.780 | 2.595 | 2.642 | 3.095 |
| *book2* | 2.448 | 2.346 | 2.300 | 2.603 |
| *geo* | 5.433 | 4.678 | 5.165 | 6.279 |
| *news* | 2.832 | 2.733 | 2.755 | 3.186 |
| *obj1* | 4.377 | 4.286 | 4.295 | 5.171 |
| *obj2* | 2.875 | 2.836 | 2.723 | 3.105 |
| *paper1* | 2.740 | 2.721 | 2.704 | 3.241 |
| *paper2* | 2.736 | 2.666 | 2.682 | 3.133 |
| *pic* | 1.582 | 1.567 | 0.912 | 1.058 |
| *progc* | 2.763 | 2.773 | 2.733 | 3.179 |
| *progl* | 2.091 | 2.122 | 1.929 | 2.193 |
| *progp* | 2.075 | 2.111 | 1.902 | 2.176 |
| *trans* | 1.911 | 1.935 | 1.658 | 1.899 |
| Avg. | 2.788 | 2.695 | 2.621 | 3.055 |
| Avg. (text) | 2.476 | 2.436 | 2.360 | 2.716 |

Table 2.2: Results of text compression (in bits/symbol) on Calgary Corpus preprocessed with BWT and MTF.

### 2.4.2 Efficiency of Various Coders On BW-transformed Texts

As mentioned above, performing the BWT and MTF on texts can generate texts dominated by small numbers, and these resulting texts are easier to compress. We compressed the resulting texts with the coders used in Section 2.4.1. For the results, please refer to Table 2.2. It is obvious that the BWT together with MTF improves the compression performance significantly. We also see that MTH and arithmetic coding are the best among the four, so we will use only these two in future experiments.

| File | *bib* | *book1* | *book2* | *geo* | *news* | *obj1* | *obj2* | *paper1* |
|---|---|---|---|---|---|---|---|---|
| frac 0 | 66.78 | 49.76 | 60.81 | 35.76 | 57.95 | 50.63 | 68.07 | 58.35 |
| File | *paper2* | *pic* | *progc* | *progl* | *progp* | *trans* | Avg. | |
| frac 0 | 55.37 | 87.40 | 60.35 | 72.86 | 74.03 | 79.24 | 62.67 | |

Table 2.3: The fraction (in percentage) of symbols encoded as 0 in $T^{mtf}$.

### 2.4.3 Efficiency of Various Run-length Coding Schemes

After performing the move-to-front coding on the BW-transformed text $T^{bwt}$, there are many 0's in the result $T^{mtf}$. Table 2.3 shows the fraction of symbols encoded as 0 in $T^{mtf}$. On average, 62.67% symbols in $T^{mtf}$ are 0's for the entire Calgary Corpus. Figure 2.2 shows that the fraction of symbols in $T^{mtf}$ drops dramatically as the value of the symbols increase. From these we can infer that if we apply a run-length coding scheme that encodes the runs of 0's to $T^{mtf}$ before we apply the MTH or arithmetic coding, we may achieve better results.

We tested the four run-length coding schemes designed for such situation in Section 2.3.5 and Table 2.4 shows the results. From the results, we can see that both MTH and arithmetic coding compress more efficiently after the run-length coding. MTH benefits much more than arithmetic coding does. This is because in MTH, each 0 is encoded by at least 1 bit, while in arithmetic coding, we can use one bit to encode more than one 0's. Although arithmetic coding can use 1 bit to encode a run of 0's, it still achieves better compression performance after MTF. This is because in our implementation, we have to use integer arithmetic instead of infinite-precision arithmetic to achieve acceptable compression speed, and we have to adopt some scaling techniques to prevent overflow, as mentioned in Section 2.3.2.

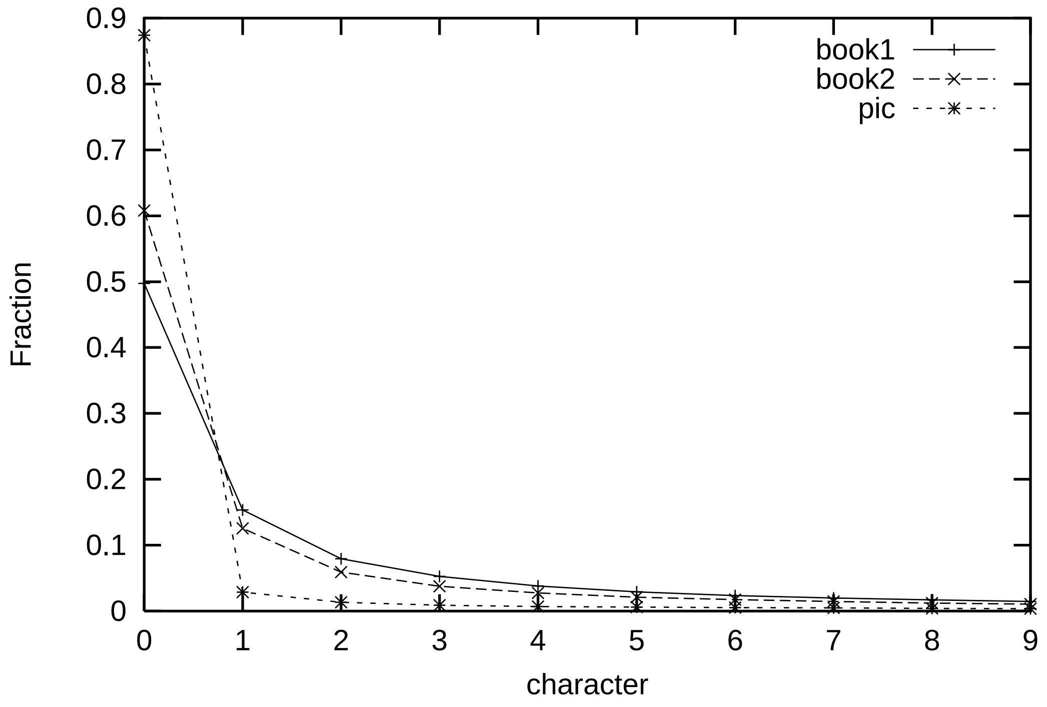

Figure 2.2: Fraction of characters [0,9] in $T^{mtf}$.

| File | MTH | | | | arith | | | |
|---|---|---|---|---|---|---|---|---|
| | RLE1 | RLE2 | RLE3 | RLE4 | RLE1 | RLE2 | RLE3 | RLE4 |
| *bib* | 2.085 | 2.069 | 2.076 | 1.995 | 2.100 | 2.087 | 2.096 | 2.031 |
| *book1* | 2.563 | 2.509 | 2.474 | 2.423 | 2.589 | 2.548 | 2.553 | 2.484 |
| *book2* | 2.184 | 2.163 | 2.122 | 2.064 | 2.201 | 2.178 | 2.179 | 2.111 |
| *geo* | 4.505 | 4.445 | 4.505 | 4.439 | 4.850 | 4.785 | 4.781 | 4.767 |
| *news* | 2.649 | 2.601 | 2.599 | 2.515 | 2.664 | 2.622 | 2.641 | 2.562 |
| *obj1* | 4.115 | 4.066 | 4.169 | 3.986 | 4.042 | 3.994 | 4.006 | 3.935 |
| *obj2* | 2.584 | 2.562 | 2.548 | 2.473 | 2.587 | 2.568 | 2.568 | 2.499 |
| *paper1* | 2.658 | 2.617 | 2.661 | 2.532 | 2.635 | 2.614 | 2.644 | 2.549 |
| *paper2* | 2.591 | 2.550 | 2.576 | 2.466 | 2.592 | 2.565 | 2.585 | 2.505 |
| *pic* | 0.845 | 0.845 | 0.820 | 0.801 | 0.838 | 0.839 | 0.821 | 0.802 |
| *progc* | 2.692 | 2.682 | 2.726 | 2.582 | 2.672 | 2.645 | 2.682 | 2.576 |
| *progl* | 1.883 | 1.865 | 1.899 | 1.771 | 1.875 | 1.861 | 1.881 | 1.781 |
| *progp* | 1.891 | 1.888 | 1.909 | 1.767 | 1.857 | 1.850 | 1.875 | 1.764 |
| *trans* | 1.650 | 1.640 | 1.643 | 1.539 | 1.624 | 1.624 | 1.630 | 1.531 |
| Avg. | 2.493 | 2.464 | 2.481 | 2.382 | 2.509 | 2.484 | 2.496 | 2.421 |
| Avg. (text) | 2.285 | 2.258 | 2.269 | 2.165 | 2.281 | 2.259 | 2.277 | 2.189 |

Table 2.4: Comparison of compression ratios (in bits/symbol) among BWT-based compression algorithms with various run-Length coding Schemes.

It is also easy to observe that on this data RLE4 is the best for compression, and MTH compresses better than arithmetic coding after run-length coding. For this reason, we always use RLE4 and MTH in future experiments.

### 2.4.4 Efficiency of Various Move-to-front Coding Schemes

Now we test the efficiency of various move-to-front coding schemes introduced in Section 2.3.4. The results are shown in Table 2.5. The conclusion is that MTF and MTF2 are the best among the four. Also, we can see that although the average performance of MTF is better than MTF2 for the Calgary Corpus, MTF2 is better

| File | MTF | MTF1 | MTF2 | MTF3 |
|---|---|---|---|---|
| *bib* | 1.995 | 2.022 | 2.021 | 2.060 |
| *book1* | 2.423 | 2.391 | 2.370 | 2.381 |
| *book2* | 2.064 | 2.056 | 2.045 | 2.064 |
| *geo* | 4.439 | 4.420 | 4.419 | 4.409 |
| *news* | 2.515 | 2.535 | 2.535 | 2.569 |
| *obj1* | 3.986 | 4.010 | 4.000 | 4.054 |
| *obj2* | 2.473 | 2.524 | 2.528 | 2.551 |
| *paper1* | 2.532 | 2.556 | 2.555 | 2.591 |
| *paper2* | 2.466 | 2.475 | 2.468 | 2.499 |
| *pic* | 0.801 | 0.765 | 0.759 | 0.752 |
| *progc* | 2.582 | 2.613 | 2.611 | 2.656 |
| *progl* | 1.771 | 1.811 | 1.819 | 1.861 |
| *progp* | 1.767 | 1.820 | 1.846 | 1.880 |
| *trans* | 1.539 | 1.607 | 1.615 | 1.657 |
| Avg. | 2.382 | 2.400 | 2.399 | 2.427 |
| Avg. (text) | 2.165 | 2.189 | 2.189 | 2.222 |

Table 2.5: Comparison of compression ratios (in bits/symbol) among BWT-based compression algorithms with various move-to-front coding Schemes.

than MTF for some of the files. We will use only MTF and MTF2 in future experiments.

MTF2 works better with $T^{bwt}$ in the case when we encounter a string which is a sequence of the same characters separated by a single different character. If we use MTF, we need to use one non-zero value to encode the different character followed by another non-zero value which encodes the first character of the second run, but if we use MTF2, we just need to use one non-zero value to encode that character, and then we start another run of 0's. This way we tend to increase the percentage of 0's in $T^{mtf}$ and the average length of runs of 0's. MTF1 and MTF3 work in a similar way. Table 2.6 shows the fraction of 0's and average length of runs of 0's in

| File | frac. of 0's | | | | avg. length of runs of 0's | | | |
|---|---|---|---|---|---|---|---|---|
| | MTF | MTF1 | MTF2 | MTF3 | MTF | MTF1 | MTF2 | MTF3 |
| *bib* | 66.78 | 63.98 | 63.92 | 60.53 | 6.151 | 6.654 | 6.696 | 7.245 |
| *book1* | 49.76 | 51.43 | 52.82 | 51.86 | 3.768 | 3.776 | 3.841 | 3.905 |
| *book2* | 60.81 | 60.48 | 61.22 | 59.32 | 5.039 | 5.194 | 5.266 | 5.442 |
| *geo* | 35.76 | 35.17 | 35.17 | 35.54 | 5.783 | 6.349 | 6.365 | 6.219 |
| *news* | 57.95 | 54.49 | 54.44 | 51.25 | 4.250 | 4.712 | 4.769 | 5.062 |
| *obj1* | 50.63 | 47.71 | 47.63 | 44.02 | 4.863 | 5.471 | 5.572 | 6.382 |
| *obj2* | 68.07 | 63.51 | 63.00 | 59.21 | 6.109 | 7.224 | 7.312 | 8.220 |
| *paper1* | 58.35 | 54.74 | 54.58 | 50.65 | 4.104 | 4.365 | 4.446 | 4.644 |
| *paper2* | 55.37 | 53.36 | 53.93 | 50.94 | 4.130 | 4.279 | 4.325 | 4.466 |
| *pic* | 87.40 | 88.56 | 88.75 | 88.82 | 27.015 | 27.040 | 26.941 | 27.354 |
| *progc* | 60.35 | 55.89 | 55.37 | 51.27 | 4.147 | 4.547 | 4.591 | 4.791 |
| *progl* | 72.86 | 68.52 | 67.71 | 63.59 | 5.839 | 6.497 | 6.620 | 7.034 |
| *progp* | 74.03 | 68.76 | 67.59 | 62.82 | 5.814 | 6.517 | 6.618 | 7.648 |
| *trans* | 79.24 | 73.18 | 72.60 | 67.81 | 7.041 | 8.097 | 8.272 | 8.757 |
| Avg. | 62.67 | 59.98 | 59.91 | 56.97 | 6.718 | 7.194 | 7.260 | 7.655 |
| Avg. (text) | 63.55 | 60.48 | 60.42 | 57.00 | 5.028 | 5.464 | 5.544 | 5.900 |

Table 2.6: The fraction (in percentage) of 0's and average length of runs of 0's in $T^{mtf}$.

$T^{mtf}$ for the four move-to-front schemes.

### 2.4.5 Alphabet Encoding

Most compression programs do not encode the set of characters which actually occur in the input files. In our case, this means we deal with an alphabet of size 256. However, the actual size of the alphabet may be much smaller. Table 2.7 shows the actual alphabet size for each file in Calgary Corpus. If we assume the alphabet size to be 256 for those files of a smaller alphabet, we will build Huffman trees with more levels in the process of MTH. This way we may assign codes of more bits to

| File | *bib* | *book1* | *book2* | *geo* | *news* | *obj1* | *obj2* |
|---|---|---|---|---|---|---|---|
| frac 0 | 81 | 82 | 96 | 256 | 98 | 256 | 256 |
| File | *paper1* | *paper2* | *pic* | *progc* | *progl* | *progp* | *trans* |
| frac 0 | 95 | 91 | 159 | 92 | 87 | 89 | 99 |

Table 2.7: The actual size of the alphabet of the files in Calgary Corpus.

some characters that occur in the input file. If we encode the actual alphabet, we can achieve better compression performance. We encode the actual alphabet by storing a bitmap of 256 bits which indicates whether the corresponding character occurs in the input file. To further reduce the overhead, we store the bitmap in two levels. The first level consists of 16 bits, and the $i$'th bit indicates whether any of the characters from $(i-1)*16$ to $i*16-1$ occurs in the file. If any of them occurs, we set the corresponding bit in the first level to be 1 and store another 16 bits in the second level to indicate whether each of the 16 characters occurs. Otherwise we set the corresponding bit to be 0.

The results are shown in Table 2.8. The columns labeled with "N" shows the results of encoding the files without alphabet encoding while the columns labeled with "Y" shows the results of encoding the files with alphabet encoding. It's clear that alphabet encoding improves the compression efficiency for files whose actual alphabet size is smaller than 256.

### 2.4.6 Final Results

Now we compare our algorithms with other compressors. Table 2.9 shows the compression results for the following methods:

| File | MTF | | MTF2 | |
|---|---|---|---|---|
| | N | Y | N | Y |
| *bib* | 1.995 | 1.971 | 2.021 | 1.996 |
| *book1* | 2.423 | 2.420 | 2.370 | 2.367 |
| *book2* | 2.064 | 2.061 | 2.045 | 2.042 |
| *geo* | 4.439 | 4.442 | 4.419 | 4.422 |
| *news* | 2.515 | 2.509 | 2.535 | 2.528 |
| *obj1* | 3.986 | 3.999 | 4.000 | 4.013 |
| *obj2* | 2.473 | 2.474 | 2.528 | 2.529 |
| *paper1* | 2.532 | 2.481 | 2.555 | 2.506 |
| *paper2* | 2.466 | 2.438 | 2.468 | 2.439 |
| *pic* | 0.801 | 0.799 | 0.759 | 0.757 |
| *progc* | 2.582 | 2.521 | 2.611 | 2.564 |
| *progl* | 1.771 | 1.740 | 1.819 | 1.781 |
| *progp* | 1.767 | 1.726 | 1.846 | 1.789 |
| *trans* | 1.539 | 1.513 | 1.615 | 1.592 |
| Avg. | 2.382 | 2.364 | 2.399 | 2.380 |
| Avg. (text) | 2.165 | 2.138 | 2.189 | 2.160 |

Table 2.8: Comparison of compression ratios (in bits/symbol) between BWT-based compression algorithms with alphabet encoding and those without it.

- gzip: standard *gzip* program with option -9 for maximum compression (this is an implementation of the LZ77 algorithm [63],
- B97: the best version of Prediction for Partial Marching (PPM) compression algorithms proposed by Bunton [17], and these algorithms predict with Markov modeling the probability of a given character based on a given number of characters that immediately precede it,
- VW98: the switching method which can be used to combine two sequential universal source coding algorithms, proposed by Volf and Willems [56, 57],
- BW94: the original Burrows-Wheeler compression algorithm [18],
- D00: the best version of BWT-based compression algorithms proposed by Deorowicz [22],
- BWT: the BWT compression algorithms implemented in this paper with the original MTF scheme,
- BWT2: the BWT compression algorithms implemented in this paper with the MTF2 scheme.

From the results we can conclude that the BWT-based algorithms can achieve compression performance close to that of the best statistical compressors. Although B97 and VW98 achieve better compression performance, they are computationally expensive. Besides, we have no idea how to index texts compressed by them. The compression performance of our implementation is close to the best version of BWT-

| File | gzip | B97 | VW98 | BW94 | D00 | BWT | BWT2 |
|---|---|---|---|---|---|---|---|
| *bib* | 2.510 | 1.786 | 1.714 | 2.07 | 1.896 | 1.971 | 1.996 |
| *book1* | 3.250 | 2.184 | 2.150 | 2.49 | 2.274 | 2.420 | 2.367 |
| *book2* | 2.700 | 1.862 | 1.820 | 2.13 | 1.958 | 2.061 | 2.042 |
| *geo* | 5.345 | 4.458 | 4.526 | 4.45 | 4.152 | 4.442 | 4.422 |
| *news* | 3.063 | 2.285 | 2.210 | 2.59 | 2.409 | 2.509 | 2.528 |
| *obj1* | 3.839 | 3.678 | 3.607 | 3.98 | 3.695 | 3.999 | 4.013 |
| *obj2* | 2.628 | 2.283 | 2.245 | 2.64 | 2.414 | 2.474 | 2.529 |
| *paper1* | 2.791 | 2.250 | 2.152 | 2.55 | 2.403 | 2.481 | 2.506 |
| *paper2* | 2.887 | 2.213 | 2.136 | 2.51 | 2.347 | 2.438 | 2.439 |
| *pic* | 0.817 | 0.781 | 0.764 | 0.83 | 0.717 | 0.799 | 0.757 |
| *progc* | 2.678 | 2.291 | 2.195 | 2.58 | 2.431 | 2.521 | 2.564 |
| *progl* | 1.805 | 1.545 | 1.482 | 1.80 | 1.670 | 1.740 | 1.781 |
| *progp* | 1.812 | 1.531 | 1.460 | 1.79 | 1.672 | 1.726 | 1.789 |
| *trans* | 1.611 | 1.325 | 1.256 | 1.57 | 1.452 | 1.513 | 1.592 |
| Avg. | 2.695 | 2.177 | 2.123 | 2.43 | 2.249 | 2.364 | 2.380 |
| Avg. (text) | 2.511 | 1.927 | 1.858 | 2.21 | 2.051 | 2.138 | 2.160 |

Table 2.9: Comparison of compression algorithms for Calgary Corpus.

based algorithms that is heavily toned by ad hoc methods and entropy coder that is very slow.

# Chapter 3

# Text Indexing

## 3.1 Introduction

There are two general categories of text indices: *word indices* and *full-text indices* [51]. For word indices, we assume that texts consist of multi-symbol objects or words. We extract keywords from the texts and organize them in some data structures that enable fast keyword search. This technique is suitable for many types of texts that can be easily parsed into a set of words, such as English texts. However, it is not suitable for biological data or texts in some far eastern languages. The queries are also restricted to keyword search, or word search if we treat all the words as keywords. For full-text indices, we treat the text as a single long string. Then we usually organize the starting positions of the suffixes in some data structures to enable queries for any substring of the text. This way we can build an index on each character and queries are not restricted to words. However, such indices generally require much space, which makes them impractical in many applications. Therefore, techniques to reduce the space occupancy have attracted much attention.

The contents of this chapter are organized as follows. In Section 2.2, we describe some basic terminology. In Section 3.3, we review some indexing techniques, where we fist review inverted files, and then we review suffix trees and some related data structures.

## 3.2 Basic Terminology

Besides the terminology adopted in the previous chapter, we adopt the following terminology in this chapter:

**Machine Model** We adopt the standard unit cost RAM model.

**Query** The queries discussed in this paper mainly deal with searching text for strings. A *query term*, or a *pattern*, is the string to be searched for in the text. We use $P[0..p-1]$ to denote a pattern of length $p$. We are mainly interested in the following types of queries:

- *existential query*: An existential query searches for $P$ in text $T$, and returns true if $P$ is found in $T$, and false otherwise.
- *counting query*: An counting query computes the number of times of occurrences of $P$ in $T$. We use *occ* to denote the result.
- *enumerative query*: An enumerative query lists all the positions of occurrences of $P$ in $T$.

## 3.3 Indexing Techniques

### 3.3.1 Inverted File

Inverted Files [38] have been the most popular word indices. An inverted file is a sorted list (index) of keywords, with each keyword having links to the records containing that keyword in the text [33]. It is obvious from the description that an inverted file contains two main parts: an *index file* containing all the keywords, and a *posting file* containing the links associated with each keyword.

The general steps to build a inverted file for an text are:

1. Divide the text into records. How to divide the text depends on your application. A record can be a fixed-length block of the text, a paragraph or chapter if the text is a book, or even a document if the text is a collection of documents.

2. Assign a list of keywords to each record. The keywords can be extracted manually or automatically from the records. If it is done automatically, we need to create a set of rules that decide the beginning of an indexable word or character sequence. We usually also need to create a list of *stopwords* (i.e. very common words) that will not be indexed. *Stemming* is another usual technique, which enables us to use one keyword to represent a set of morphologically similar indexing and search terms [30].

3. Compute the set of distinct keywords and store them in the index file. For each keyword, we compute its *inverted list* [65], storing the identifiers of the records containing the value. Store all the inverted lists into the posting file

and record the starting position and the size of each list (i.e. the number of occurrences for each keyword) in the index file.

After we build the inverted file, the searching for a keyword are usually performed in two steps:

1. Search for the keyword in the index file.

2. If we find the it, retrieve its corresponding inverted list in the posting file to get the records containing it.

In order to efficiently search for a keyword in the index file, we organize the index file in a second index structure, such as a sorted array, a prefix B-tree, or a hash table [33]. Figure 3.1 illustrates an inverted file implemented using a sorted array [33], in which keyword search can be performed with a binary search.

Inverted files support keyword query efficiently and is widely used in many text database systems. However, their support for searching is quite limited. First, it is obvious that only keywords are searchable in the index. Second, some kinds of searches are inefficient on inverted files, such as search for phrases, regular expression searching and approximate string searching. Third, inverted files usually need a high storage overhead. For example, the size of an inverted file varies from 50% to 300% of the size of the original text, if we treat each word in the text as a keyword [23].

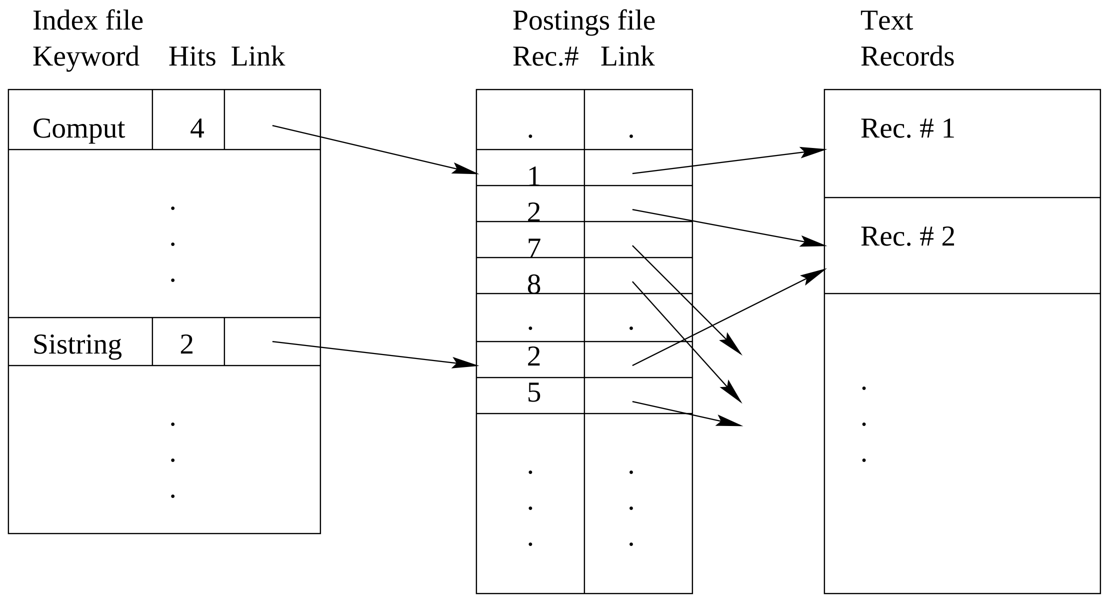

Figure 3.1: An inverted file implemented using a sorted array.

## 3.3.2 Suffix Trees and Related Data Structures

Almost all the practical full-text indices are derived from suffix trees, which were first discovered by Weiner [58]. For a suffix tree index, we see the text as one long string. The *suffix*, or *semi-infinite string*, is the substring starting from the given position in the text and continuing to the end of the text. Each suffix has a shortest prefix that distinguishes it from the other suffixes. A suffix tree is constructed over the suffixes as a tree-based data structure, which enables us to perform a query by searching the suffixes of the text. Some of the variants of the suffix tree structure are *suffix trie* [58], *prefix tree* [42], and *PAT tree* [31].

Among these variants, we only review PAT tree [31]. A PAT tree is a PATRICIA tree [44] constructed over the suffixes of a text. To build a PAT tree, we need to

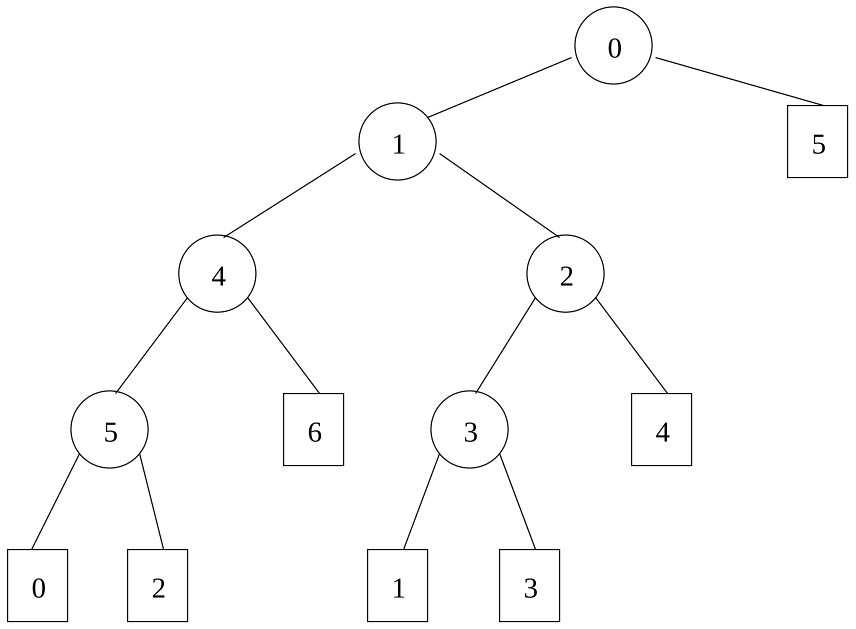

Figure 3.2: PAT tree.

append a special end-of-file symbol # to the end of the text $T$. We also need to give a binary encoding for each symbols in $\Sigma \cup \{\#\}$, which is used to encode suffixes and query terms. In a PAT tree, the individual bits of the suffixes decide on the branching. To be more specific, a 0 bit in the suffix causes a branch to the left subtree, and a 1 bit to the right. Each internal node is labeled with the position of the first bit that distinguishes suffixes contained in the left sub-tree from those in the right sub-tree, while each leaf contains a suffix. Figure 3.2 is a PAT tree constructed over the text $T = ababbcab$, in which the encoding $a = 00$, $b = 01$, $c = 10$ and $\# = 11$ is used. Note that the leaves store the starting positions of the suffixes (the positions are starting from 0) instead of the suffix strings.

The general steps of searching for a pattern $P[0, p-1]$ with a PAT tree index constructed over text $T[0, u-1]$ are:

1. Encode $P$ in binary. We use $B$ to denote the result.

2. Process the bits of $B$ sequentially. At each pass, we check the bit in $B$ at the position indicated in the internal node, and move on to the left or right sub-tree according to whether the bit is 0 or 1, respectively. Repeat the process until we reach a leaf or we run out of the bits of $B$.

3. Get the suffix stored in one of the leaves of the sub-tree rooted in the current node and compare it with $P$. If their first $m$ characters are the same, then we return all the positions of the suffixes stored in the leaves of the sub-tree rooted in the current node as the positions of each occurrence of $P$ in the text. Otherwise we return that $P$ does not occur in $T$ at all.

It is obvious from the above description that the cost for existential query is $O(p)$, and counting and enumerative query costs additional $O(occ)$ time to retrieve all the occurrences. This is very efficient. However, the space occupancy of PAT tree is between $4u$ and $5u$ words for the text [31], which is impractical for many applications.

Note that it is also possible to build word indices with a suffix tree. All we need to do restrict the indexing point to the start position of keywords (i.e. construct suffix tree over the suffixes starting with one of the keywords).

Various techniques have been developed to reduce the space cost of suffix trees. Some of them are:

1. *Suffix Arrays* [40, 31]: Manber and Myers and Gonnet et al. independently

proposed suffix arrays, or PAT arrays. This idea is to organize the suffix offsets in a sorted list using the suffixes as sort keys instead of organizing them in a tree. This way it only cost $u$ words or $u \lg u$ bits to store a suffix array. A search for a pattern can be performed over suffix arrays using a binary search, which costs $O(p \lg u)$ time since each comparison requires $O(p)$ time. To improve the efficiency, Manber and Myers developed a secondary structure to boost the searching speed to $O(p + \lg u)$. This secondary structure stores some kind of pre-computed longest common prefixes and is stored in two arrays, each of which costs $u \lg u$ bits. They also proposed another auxiliary structure, which costs $\frac{u}{4} \lg u$ bits, to boost the searching speed to $O(p)$ average time.

2. *Compact PAT trees* [21]: A recent line of research has been built upon the idea of *succinct data structures* [47]. Jacobson first proposed the idea and method for representing static data structures such as trees and graphs succinctly [35] (i.e. close to the theoretical lower bound of the space cost to represent the structures), while at the same time allows the navigating operations to be performed efficiently. Based on such ideas, Clark and Munro proposed the method to represent PAT trees succinctly [21] and achieve the result of representing the PAT tree at the cost of less than $3.5+\lg u+\lg\lg u+O(\frac{\lg\lg\lg u}{\lg u})$ bits of expected size per index point, while achieving efficient query time. The representation of the binary tree itself was reduced to the information theoretic limit of $2 + o(1)$ bits per node by Munro and Raman [48].

3. *Compressed Suffix Arrays* [32]: The compressed suffix array structure also

falls in the framework of succinct data structures. It was motivated by the observation that there are totally $O(2^u)$ possibles distinct strings of fixed length $u$ on a given alphabet of constant size, therefore there is a canonical way to represent suffix arrays in $O(u)$ bits. Grossi and Vitter proposed a method to build a compressed suffix array, which cost $O(u)$ bits and supports an access to any element of the original suffix array in $O(\lg^\epsilon u)$ time for any fixed constant $0 < \epsilon < 1$ without computing the entire original suffix array, which allows efficient query.

# Chapter 4

# Indexing Compressed Texts

## 4.1 Introduction

Text compression and indexing are usually treated as independent problems. Recently researchers have tried to combine them in order to allow efficient searching through compressed text. This is motivated by the observation that certain compression and indexing methods have something in common, such as commonality in some of the ideas, algorithms or data structures used in these methods. We can exploit such commonality to index compressed texts.

The contents of this chapter are organized as follows. In Section 4.2, we review some techniques to index compressed texts, namely the techniques used in shrink and search engine and the opportunistic index. In Section 4.3, we describe some ideas of implementing the opportunistic index directly. We design a new MTF method to reduce the space cost of auxiliary information. In Section 4.4, we describe one practical implementation of the opportunistic index: FM-index. Finally, in

Section 4.5, we implement and test the indexing and searching algorithms of FM-index, together with some improvements.

## 4.2 Related Work

### 4.2.1 Shrink and Search Engine

Shrink and Search Engine [19, 20], or SASE, is a compressed text search engine. It exploits the fact that both inverted file index and dictionary coding require a dictionary structure: An inverted file is typically maintained as a dictionary of all the distinct words or keywords in the text with a linked list of occurrence pointers associated with each word, while some of the dictionary coding methods such as LZW introduced in Section 2.3.3 maintain as a dictionary of the words in the text with a distinct numerical identifier for each word, and replace each word in the text with the identifier to reduce the redundancy in the text. Hence the dictionary used in the inverted file can be reused in dictionary coding to integrate both techniques.

It is obvious from the above ideas that the key issue is how to maintain such a dictionary that can be used both in inverted file index and dictionary coding. In SASE, three dictionaries are maintained. These are for common words, uncommon words and literals, respectively. The dictionary for literals consists of several sub-dictionaries. We assign an identifier to each word in the dictionaries. The identifier for each common word is 1 byte long, while the identifier for each uncommon word or literal is two bytes long. The dictionaries are organized as hash tables.

Before compressing and indexing the original text, we process the text off-line and compute the *compression benefit factor* (i.e. the product of a word's frequency and the compression gain of replacing it by a single numerical identifier) of each distinct word in the text, and partition the words into three groups according to their compression benefit factors. The words in the group with the highest compression benefit factors are stored in the dictionary for common words, the words in the group with second highest compression benefit factors are stored in the dictionary for uncommon words, and the rest are stored in the dictionary for literals. Then we partition the text into records and process it word by word. For each word in the text, we replace it with its corresponding identifiers to achieve dictionary coding at the word granularity. At the same time, for all the occurrences of a distinct word in a certain block, we record the block number, number of occurrences and the position of the first occurrence in the corresponding record for the word in its dictionary. In this way, inverted file index is constructed over the dictionaries.

Searching for a pattern $P$ with SASE are generally performed in the following steps:

1. Search for $P$ in the dictionaries.

2. If $P$ is found, then we have the identifier of $P$ and ID's of the blocks that contains $P$. For each block that contains $P$, we also have how many times $P$ occurs in the block, and the position of the first occurrence of $P$ in the block.

3. Search the blocks that contains $P$, staring from the position of the first occurrence of $P$ in the block. We only need to compare the identifiers of words instead of performing string matching. This further reduces the search cost.

SASE's are suitable for building word indices for texts consisting of words of limited distinct values, such as natural English texts. The performance result shows that SASE archives compression ration comparable to that of Gzip, and supports efficient keyword query. However, the support for query of SASE are limited to that of word indices, especially inverted lists.

### 4.2.2 Opportunistic Index

Opportunistic data structures were proposed by Ferragina and Manzini [26]. They are called opportunistic because their "space occupancy is decreased when the input text is compressible and the space reduction is achieved at no significant slowdown in the query performance" [26]. Opportunistic indices make use of the fact that the conceptual matrix $M$ stores all the suffixes of $T$ in sorted order, as mentioned in Section 2.3.6, while, on the other hand, a suffix tree is constructed over all the suffixes.

An opportunistic index consists of two parts: a compressed representation (denoted by $Z$) of the Burrows-Wheeler transformed string $T^{bwt}$, and some auxiliary information. Part of the auxiliary information is stored to enable the following operation to be executed in constant time: $Occ(c, 0, k)$, which is to compute the number of occurrences of $c$ in $T^{bwt}[0, k]$. We first assume that the $Occ(c, 0, k)$ operation is implemented efficiently in order to introduce the key algorithms for searching with an opportunistic index, and then we introduce how to organize the auxiliary information to provide an efficient implementation of $Occ(c, 0, k)$.

**Searching Algorithms** Searching for a pattern $P[0..p{-}1]$ through text $T[0..u{-}1]$ with an opportunistic index requires two operations: the operation *count* which returns the number of occurrences of $P$ in $T$ if $P$ occurs in $T$ at all, and returns "pattern not found" otherwise; and the operation *locate* which returns the position of a given occurrence of $P$ in $T$.

The following is the algorithm for the *count* operation [26]:

```
Algorithm count(P[0..p-1])
1. c = P[p - 1], i = p - 1;
2. sp = C[c], ep = C[c + 1] - 1;
3. while ((sp <= ep) and (i >= 1) do
4.   c = P[i - 1];
5.   sp = C[c] + Occ(c, 0, sp - 1);
6.   ep = C[c] + Occ(c, 0, ep) - 1;
7.   i = i - 1;
8. if (ep < sp) return "pattern not found"
   else return "found (ep - sp + 1) occurrences"
```

During the execution of the above algorithm, the variable *sp* and *ep* record the first and last rows of $M$ prefixed by $P[i, p-1]$, respectively. To achieve this, we first initialize them via array $C$. As mentioned in Section 2.3.6, $C[c]$ is the position of the first occurrence of $c$ in $F$, so before iteration, $C[c]$ is the position of the first occurrence of $P[p-1]$ in $F$, and $C[c+1]-1$ is the last since all the characters in $F$ are sorted in increasing order. In each pass of the while loop, we update *sp* and *ep* in Step 5-6 using the LF-mapping.

To show the correctness of Step 5-6, we take the updating of *sp* as an example. According to (2.1) in Section 2.3.6, Step 5 computes the row in $M$ corresponding to the first row in $M$ that starts with $P[i..p-1]$ and ends with $P[i-1]$, under the LF-mapping. Assume the above two rows are $M[s] = P[i-1]P[i..p-1]\alpha$ and $M[t] = P[i..p-1]\alpha P[i-1]$, respectively. Because the $M[t]$ is the first row in $M$ that starts with $P[i..p-1]$ and ends with $P[i-1]$, we have the $M[s]$ is the first row in $M$ that starts with $P[i-1]P[i..p-1]$. This is true because otherwise, assume we have another row $M[v] = P[i-1]P[i..p-1]\beta$ such that $v < s$. Then we have $\beta < \alpha$ in lexicographic order, so the $M[w] = P[i..p-1]\beta P[i-1] < M[t] = P[i..p-1]\alpha P[i-1]$ in lexicographic order, which is contradictory to the assumption that $M[t]$ is the first row in $M$ that starts with $P[i..p-1]$ and ends with $P[i-1]$.

It is obvious the *count* operation costs $O(p)$ time.

Now we look into the *locate* operation. The basic idea is to logically mark parts of the rows of $M$, and store the starting positions of the corresponding text suffixes explicitly in some auxiliary data structures when constructing the index. Then we use the following algorithm to compute the position of each occurrence of $P$ (assume that $s$ is a row in $M$ whose prefix is $P$, and we use $pos(s)$ to denote its position) [26]:

```
algorithm locate(s)
1. t = s, v = 0;
2. while row M[t] is not marked do
3.   c = L[t];
4.   t = C[c] + Occ(c, 0, t) - 1;
```

```
5.    v = v + 1;
6. return pos(t) + v;
```

The above algorithm first checks whether row $M[s]$ is marked. If it is, then we have the position. Otherwise, in Step 2-5, we find the row before it that is marked. To be more specific, in Step 3-4, we find the row that corresponds to it under the LF-mapping, which is prefixed with $T[pos[s]-1..u-1]$. $v$ records how many times we iterate the above process before we find a marked row $t$, so in Step 6 we have the $pos(s) = pos(t) + v$. Note that in Step 3, we do not have the string $L = T^{bwt}$. In order to compute $L[t]$, for each character $e$ in $\Sigma$, we compute $Occ(e, 0, t-1)$, and $Occ(e, 0, t)$. It is obvious that $Occ(e, 0, t-1) \neq Occ(e, 0, t)$ if and only if $e = L[t]$. This costs $O(|\Sigma|) = O(1)$ time since $\Sigma$ is of constant size.

To implement the above algorithm, we need to design a scheme to mark rows. One approach is to mark the rows corresponding to text positions having the form $1 + i\eta$, where $i = 0, 1, ..., u/\eta$. Ferragina and Manzini showed in [27] how to make use of a Packet B-tree [7] to make it possible to check whether a row is marked in constant time, when $\eta = \Theta(\lg^2 u)$, and the additional space cost is $O(\frac{u}{\lg u})$ bits per input symbol. Besides, since the loop in the operation is executed at most $\eta$ times, the algorithm costs $O(\lg^2 u)$ time. To list all the occurrences, we simply need to perform the above operations *occ* times, which cost $O(occ \lg^u)$ time, and the space cost remains the same.

In [27], Ferragina and Manzini also designed another more complicate method to implement the *locate* operation in $O(\lg^\epsilon u)$ time, at additional space cost of $O(\frac{\lg\lg u}{\lg^\epsilon u})$ bits per input symbol.

**Index Structures** Now we introduce how to organize the index structures except the additional information required by the *locate* operation, in order to implement the function $Occ(c, 0, k)$, which costs constant time.

The text is compressed with the BW-based algorithm described in [41]. We logically partition the transformed string $T^{bwt}$ into substrings, or *buckets*, of length $l$ each, and denote them by $BT_i$, for $i = 0, 1, ..., u/l-1$. We denote the corresponding compressed buckets by $BZ_i$, for $i = 0, 1, ..., u/l - 1$. Then we group each of the succeeding $l$ buckets into a *superbucket*, and denote the superbuckets by $BS_i$, for $i = 0, 1, ..., u/l^2-1$. In the opportunistic index, we store all the $BZ_i$'s consecutively. We assume that each run of 0's are contained in one bucket, and the case when this is not true is more complex and is outlined in [26].

From the above we can see that $Occ(c, 0, k)$ equals to the sum of the following three: the number of occurrences of $c$ in the superbuckets preceding the superbucket that contains $T^{bwt}[k]$ (denoted by $occ1$), the number of occurrences of $c$ in the buckets preceding the bucket that contains $T^{bwt}[k]$ but existing in the same superbucket (denoted by $occ2$), and the number of occurrences of $c$ from the start of the bucket that contains $T^{bwt}[k]$ till position $k$ (denoted by $occ3$). To compute the above three values efficiently, we store some auxiliary information, consisting of three parts accordingly:

- To compute $occ1$, we store:
  - The two-dimensional array $NO[0..u/l^2 - 1, 0..|\Sigma| - 1]$ that stores in the entry $NO[i, c]$ the number of occurrences of the character $c$ in all the superbuckets preceding $BS_i$.

  - The array $W[0..u/l^2 - 1]$ that stores in the entry $W[i]$ the position of the end of the compressed image of superbucket $BS_i$ in the index.

- To compute *occ*2, we store:
  - The two-dimensional array $NO'[0..u/l - 1, 0..|\Sigma| - 1]$ that stores in the entry $NO'[i, c]$ the number of occurrences of the character $c$ in the buckets preceding bucket $BT_i$, but existing in the same superbucket that contains $BT_i$. Note that all the values in the array are smaller than or equal to $l^2$.
  - The array $W'[0..u/l - 1]$ that stores in the entry $W'[i]$ the starting position of the compressed image of bucket $BT_i$ (i.e. $BZ_i$) in the index.

- To compute *occ*3, we store:
  - The array MTF[0..u/l-1] that stores in the entry $MTF[i]$ the state of the MTF list at the beginning of the coding of $BT_i$. Note that each entry costs $|\Sigma| \lg |\Sigma|$ bits, which is of constant size.
  - The table $S$ that stores in the entry $S[c, j, b, m]$ the number of occurrences of $c$ among the first $j$ characters of the compressed string $b$, and $m$ is the state of MTF list at the beginning of the procedure to produce $b$.

With the above index structure, we can compute $Occ(c, 0, k)$ in the following steps in constant time:

1. Determine the superbucket $BS_j$ that contains $T^{bwt}[k]$. This can be done by one single computation since $j = \lceil k/l^2 - 1 \rceil$. Then we have $occ1 = NO[j, c]$.

2. Determine the bucket $BZ_h$ that contains $T^{bwt}[k]$. This can be done by one single computation since $h = \lceil k/l - 1 \rceil$. Then we have $occ2 = NO'[h, c]$.

3. Retrieve $BZ_j$ from the index, whose starting position can be determined from $W$ and $W'$. Assume position $k$ is the $g$'th position in $BZ_h$. Then we have $occ3 = S[c, g, BZ_h, MTF[h]]$.

4. Sum $occ1$, $occ2$ and $occ3$, and return the result.

In [26], Ferragina and Manzini showed how to choose value for $l$ to make the call to $Occ(c, 0, k)$ cost constant time, at the additional space cost of $O((u/\lg u) \lg\lg u)$ bits. The non-trivial issue in their design is how to choose parameters for the table $S$. Assume that the maximum length of the compressed buckets is $l'$ bits, and according to [41], $l' = (1 + 2\lfloor \lg \Sigma \rfloor)l$. Then table $S$ has $|\Sigma| l 2' |\Sigma|! = O(l 2^{l'})$ entries, and each entry costs $O(\lg l)$ bits. We choose $l = \Theta(\lg u)$ bits so that $l' = c \lg u$, where $c < 1$. Then the size of table $S$ is $O(u^c \lg u \lg\lg u)$ bits, which is $o(u)$.

In summary, the overall result is that an enumerative query can be performed with an opportunistic index in $O(p + occ \lg^{\epsilon} u)$ time, and the space occupancy is $O(H_k(T) + \frac{\lg\lg u}{\lg^{\epsilon} u})$ bits per symbol (including the compressed text).

However, we note that the above method cannot be implemented directly. One notable fact is, in the table $S$, the fourth dimension contains all the possible statuses of the MTF list, and there are $|\Sigma|!$ possible statuses. In the above discussion, $|\Sigma|!$ is treated as a constant value, but in practice, for a vocabulary of moderate size such as English texts, this is overwhelmingly large.

## 4.3 One Attempt to Implement Opportunistic Index

From the discussion in Section 4.2.2, we know that in most cases it is impossible to implement that opportunistic index directly (i.e. as its theoretical design). One notable fact is that there are $|\Sigma|!$ possible statuses of the MTF list. In this section, we try to reduce the number of possible statuses of the MTF list by designing some special MTF scheme. We label the MTF schemes introduced here as MTFk.

In this scheme, we first set a constant number $k \leq \Sigma$. Initially the MTF list contains all the alphabet symbols in increasing order. When we encounter a symbol that is not at the front of the MTF list, we move it to the front. Then we move the $k$'th symbol in the MTF list (assume that the positions in the MTF list starts at 0) backward, until we encounter a symbol that is greater than it. In other words, we maintain the MTF list in such a way that the last $\Sigma - k$ symbols in the list are in sorted order. For example, $\Sigma = a, b, c, d, e$ and $k = 2$. Assume that before we encounter a symbol (assume it is $c$), the MTF list is *ebacd*. Then after when encounter $c$, the MTF list is changed to *ceabd*. Note that when $k = \Sigma$, this method is the same as the original MTF scheme.

Since the last $\Sigma - k$ symbols in the MTF list are sorted, the MTF list has $P^k_{|\Sigma|} = O(|\Sigma|^k)$ possible statuses. Table 4.1 and Figure 4.1 show the compression ratios for different values of $k$ on the Calgary Corpus. They show that we achieve

| File | $k = 1$ | $k = 2$ | $k = 3$ | $k = 4$ | $k = 5$ | $k = \|\Sigma\|$ |
|---|---|---|---|---|---|---|
| *bib* | 2.232 | 2.147 | 2.085 | 2.082 | 2.068 | 1.971 |
| *book1* | 2.915 | 2.730 | 2.655 | 2.598 | 2.563 | 2.420 |
| *book2* | 2.457 | 2.291 | 2.212 | 2.181 | 2.155 | 2.061 |
| *geo* | 4.425 | 4.499 | 4.509 | 4.521 | 4.523 | 4.442 |
| *news* | 2.836 | 2.727 | 2.663 | 2.628 | 2.614 | 2.509 |
| *obj1* | 4.169 | 4.108 | 4.063 | 4.048 | 4.049 | 3.999 |
| *obj2* | 2.784 | 2.662 | 2.619 | 2.591 | 2.578 | 2.474 |
| *paper1* | 2.779 | 2.668 | 2.623 | 2.601 | 2.579 | 2.481 |
| *paper2* | 2.757 | 2.625 | 2.573 | 2.560 | 2.537 | 2.438 |
| *pic* | 0.843 | 0.861 | 0.868 | 0.867 | 0.868 | 0.799 |
| *progc* | 2.846 | 2.670 | 2.639 | 2.613 | 2.593 | 2.521 |
| *progl* | 1.939 | 1.858 | 1.826 | 1.809 | 1.792 | 1.740 |
| *progp* | 1.970 | 1.853 | 1.818 | 1.797 | 1.783 | 1.726 |
| *trans* | 1.686 | 1.605 | 1.565 | 1.565 | 1.552 | 1.513 |
| Avg. | 2.617 | 2.522 | 2.480 | 2.462 | 2.447 | 2.364 |
| Avg. (text) | 2.442 | 2.317 | 2.266 | 2.243 | 2.224 | 2.138 |

Table 4.1: Comparison of compression ratios (in bits/symbol) for different values of $k$ with MTFk.

better compression performance when the value of $k$ increases. Besides, one notable fact is, even when $k = 1$, BWT with MTFk compresses better than gzip (see Table 2.9 for the results of gzip).

MTFk reduces the number of possible statuses of the MTF list, and the compression it achieves is reasonable, but it still costs too much additional space (mainly for table $S$) to build an opportunistic index for a vocabulary of moderate size when the value $k$ increases. For example, Assume $u = 1,000,000,000$ and $|\Sigma| = 64$. If we choose $c = \frac{1}{2}$, then after we omit some lower-order items and use $|\Sigma|^k$ as an upper bound for $P^k_{|\Sigma|}$, we estimate the size of table $S$ to be $u^{\frac{1}{2}}\Sigma^{k+1}$. To ensure that the size of $S$ is reasonable, we must have $u^{\frac{1}{2}}\Sigma^{k+1} \ll u$, which in turn requires

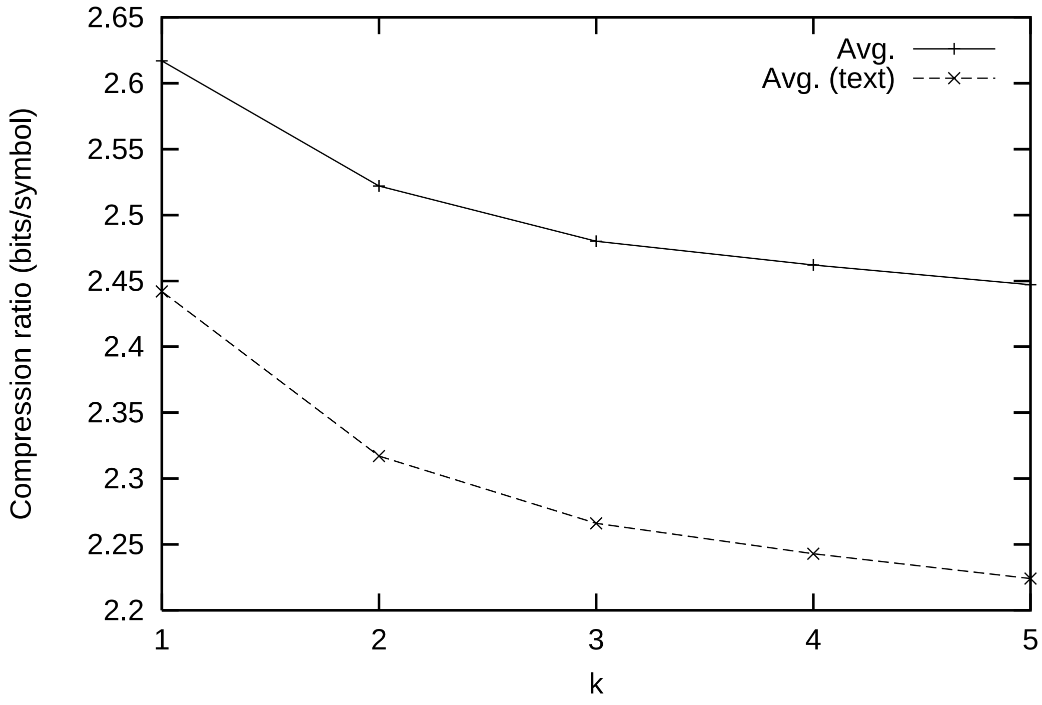

Figure 4.1: Compression ratio (bits/symbol) as a function of $k$ with MTF4.

$\Sigma^{k+1} \ll u^{\frac{1}{2}}$. With the above given parameters, it means $k < 2$. This clearly shows that our method is not able to reduce the space occupancy of $S$ to an acceptable level in most cases, although it is suitable for vocabulary of small size such as that of biological sequences. Besides, there are other considerations for the theoretical design. For example, the theoretical design requires the size of bucket to be smaller than a machine word and is not practical. Therefore, we will not use MTFk in future experiments. However, we still report it in this thesis since it is possible it might prove to be useful in certain other applications.

## 4.4 Implementation

Ferragina and Manzini proposed and tested one method to implement the opportunistic index in [28]. They name the result FM-index. In this section, we base our implementation mainly on FM-index, though we also include some of our ideas and go into the details of implementation.

**Compression Scheme** As what is discussed in the theoretical design of the opportunistic index, FM-index also encodes the BW-transformed texts bucket by bucket. The difference is that in the theoretical design, we encode $T^{mtf}$ with one statistic coder bucket by bucket, while in FM-index, we encode $T^{bwt}$ with MTF and one statistic coder bucket by bucket. To be more specific, when constructing FM-index, after we performed the BWT, we divide $T^{bwt}$ into buckets, and for each bucket, we perform MTF coding. This method will lower the compression efficiency, but we will not need to store the status of MTF list at the beginning of each bucket as what is required in the theoretical design.

According to the experiments in Section 2.4, we use MTH to encode each bucket after MTF coding. As for the MTF scheme itself, we will test MTF and MTF2 and compare the results, since both of them seem promising for compressing texts.

**Data Structures for the function** *occ* To implement the function *occ*, we store the compressed result as a two-level structure. First, $T^{bwt}$ is partitioned into superbuckets of size $l_s$. Second, each superbucket is partitioned into buckets of size $l_b$ provided that $l_b$ divides $l_s$. As mentioned above, each bucket is individually encoded with a MTF coder and a MTH coder, such that we can decompress each compressed bucket to get its corresponding bucket in $T^{bwt}$ without decompressing the whole text. We also store some auxiliary information so that the index is organized as the following:

- *Header*: It contains the starting positions of the three parts following the header in the FM-index.
- *Superbuckets Section*: For each superbucket, we store the number of occurrences of every alphabet symbol in the previous superbuckets. This section is implemented as an array, and each entry is also an array itself that corresponds to each superbucket (with the exception of the first superbucket since there is obviously no superbuckets preceding it), whose entries correspond to the alphabet symbols.
- *Bucket Directory*: In this directory, we store the starting positions of each compressed bucket in the body of FM-index. It is also implemented as an

array.

- *body*: It contains the compressed buckets, each of which in turn consists of two part: a *header* containing the number of occurrences of each alphabet symbol from the beginning of the superbuckets but before the current bucket, and a compressed image of the bucket itself.

With the above structure, we can perform $Occ(c, 0, k)$ in the following steps:

1. Locate the superbucket that contains $T^{bwt}[k]$, and from the superbucket section, retrieve the number of occurrences of $c$ in the previous superbuckets.

2. Locate the position of the compressed bucket that contains $T^{bwt}[k]$ from the bucket directory, and retrieve the compressed bucket with a single disk operation. From its header, retrieve the number of occurrences of $c$ from the beginning of the superbuckets, to (but not including) the beginning of the bucket itself.

3. Decompress the compressed image of the bucket to get its corresponding bucket in $T^{bwt}$. Then we count the occurrences of $c$ from the beginning of the bucket to $T^{bwt}[k]$.

4. Sum the above three results and return the result.

**Data Structures for the *locate* operation** The key of implementing the *locate* operation is to design a scheme to mark the rows. We adopt the simple scheme used in [28]: only mark the rows of $M$ that ends with a certain character $d$, and store their starting positions in $T$ sequentially in an array. With this scheme, to check whether row $M[i]$ is marked, we simply compare its last character with $d$. If it is, then the corresponding position is the $Occ(c, 0, i)$'th entry of the array.

This scheme takes advantage of the fact that the occurrences of each character in most real texts are roughly equally spaced. It highly relies on the structure of the text and cannot ensure good worst-case performance. However, it greatly reduces the space overhead since we do not need any other auxiliary information besides the arrays that stores positions of the marked rows. We can also choose the percentage of the rows to be marked and choose a symbol $d$ whose frequency is the closest to it.

**Caching** In *count* and *locate* locations, we will call the function *occ* many times, depending on the times of occurrences of the pattern we are searching. This means that we may decompress some buckets more than once during searching. We design a caching scheme to take the advantage of this.

In this scheme, we maintain a fixed-size cache in memory that stores the buckets that have been decompressed. We organize the cache as a double list, and for the replacement policy, we choose the least recently used (LRU) policy. For each bucket, we also store a pointer that points to its decompressed result in the cache, and the pointer is NULL if its decompressed result is not in cache. Then in function *occ*, when we need to decompress a bucket, we fist check whether its decompressed result is stored in the cache. If it is, then we retrieve the result from the cache without decompressing it.

**Implementation Details** During implementation, we also designed some techniques to reduce the space cost and/or to improve the search efficiency. They are listed in the following:

1. For each number of occurrences stored in the superbucket section, or each position stored for the marked rows, we use $\lg u$ bits to represent it. For each number of occurrences stored in the header of each compressed bucket, we use $\lg l_s$ bits to represent it. Although this means we will have more bit operations and thus the query will be slowed down, this scheme significantly reduces the space cost.

2. In the header, we store a bitmap of the characters occurring in the text. Then, in the superbucket section, we only need to store number of occurrences of the characters actually appear in the text. This reduces the space cost, and also speeds up the computation of *occ*, since if a character $c$ in the pattern does not occur in the text at all, we can determine from the bitmap that $Occ(c, 0, k) = 0$. For each superbucket, we also store a bitmap of the characters occurring in the superbucket, and similarly, this can reduce the space cost and speed up the computation of *Occ*. These bitmaps can be organized with the structure introduced in Section 2.4.5.

3. In the function $Occ(c, 0, k)$, when we need to decompress a bucket, we actually do not have to decompress the whole bucket. Instead, we just need to decompress until we reach $T^{bwt}[k]$. This further speeds up the computation of *Occ*. However, when we adopt the caching scheme, we have to decompress the whole bucket so that we can cache the result.

| Name | Size in Bytes | Content |
|---|---:|---|
| *world192* | 2,473,400 | 1992 CIA world fact book |
| *canterbury* | 2,821,120 | Canterbury corpus |
| *bible* | 4,047,392 | King James Bible |
| *E.coli* | 4,638,690 | DNA sequence |
| *shakespeare* | 7,648,372 | Shakespeare XML database |
| *ohsumed.87* | 60,303,307 | MEDLINE references from 1987 |
| *gutenberg* | 63,035,670 | A collection of texts from Project Gutenberg |
| *dblp* | 127,074,908 | The DBLP database |
| *rfc* | 146,744,697 | Internet RFC documents |
| *jdk14* | 146,963,536 | html and java sources in the jdk1.4 documentation |

Table 4.2: Files used in our experiments.

4. For step 3 of the *locate* operation, we check for each character $e$ in $\Sigma$ whether $Occ(e, 0, k-1) = Occ(e, 0, k)$ to compute $L[k]$ in Section 4.2.2 to avoid decompressing buckets. However, in our implementation of $Occ$, we have to decompress buckets when necessary. Therefore, in our implementation of *locate*, we decompress the bucket to compute $L[k]$ in Step 3. Since in Step 4, we need to decompress the same bucket when $Occ$ is called, we combine these two steps together so that we just need to decompress the same bucket only once.

## 4.5 Performance Results

### 4.5.1 Large Texts Used in Experiments

We now test the performances of the implementation in Section 4.4 on several input files. We chose files in table 4.2 to represent different types of input texts, as they are of different types of content, format or style. For their sizes and content, please refer to Table 4.2. The following is a more detailed description:

- *world192* : The 1992 CIA world fact book, which is included in the Canterbury Corpus[8]. It is in the plain-text format.
- *canterbury* : A tar archive containing the small files of the Canterbury Corpus[8], which includes both binary files and text files in various formats (plain text, html, c and list code).
- *bible* : The King James Bible, which is included in the Canterbury Corpus[8]. It is in the plain-text format.
- *E.coli* : Complete gnome of the E.coli bacterium, which is included in the Canterbury Corpus[8].
- *shakespeare* : A collection of Shakespeare lyrics that have been converted into XML, from [4].
- *ohsumed.87* : MEDLINE references from 1987. MEDLINE is an online medical information database,consisting of titles and/or abstracts from 270 medical journals over a 5-year period (1987-1991). This file is from the TREC-9 document collection used in the TREC-9 filtering track[5].
- *gutenberg* : A collection of texts from Project Gutenberg[2], which consists of various literature work and references.
- *dblp* : The DBLP database[39], which contains bibliographical references for databases and logical programming research. The underlying data is stored

in XML.

- *rfc* : The entire Internet RFC (the Request for Comments) documents, which is available in [3].
- *jdk14* : The concatenation of html and java sources in the jdk1.4 documentation[1].

As for the test sets of patterns to be searched for, we randomly chose 1000 English words of length between 4 and 8 (for *E.coli* we generated 1000 random DNA sequences of length between 8 and 15), as in [28].

The following experiments mainly follow the experimental part in [28]. However, we also test some of our own ideas. All the experiments were run on a Sun Ultra Enterprise 450 machine, with 4 CPUs and 4 Gb RAM, installed with SunOS 5.8.

### 4.5.2 Size of Superbuckets

It can be seen from Section 4.2.2 and Section 4.4 that the purpose of superbuckets is to reduce the auxiliary information stored in buckets (recall that the number of occurrences for each character costs $\lg l_s$ bits). If we choose to use smaller superbuckets, the auxiliary information stored in buckets will cost less space, but the superbucket section will cost more space. Therefore we try to determine the appropriate size of superbuckets via experiments.

Table 4.3 and Table 4.4 reports the results of our experiments on *bible* and *ohsumed.87*. In our experiments, we set the bucket size to be 1Kb, and the fraction of marked rows is 2%. The row labeled MTF shows the results when we use the original MTF scheme, while the row labeled MTF2 shows the results when

| | Superbucket size | 2K | 4K | 8K | 16K | 32K | 64K |
|---|---|---|---|---|---|---|---|
| MTF | Compression ratio | 3.080 | 2.750 | 2.623 | 2.607 | 2.651 | 2.730 |
| | Ave. count time | 2.36 | 2.65 | 2.66 | 2.48 | 2.61 | 2.56 |
| | Ave. locate time | 19.30 | 21.06 | 20.50 | 19.79 | 19.65 | 20.24 |
| MTF2 | Compression ratio | 3.058 | 2.727 | 2.601 | 2.585 | 2.628 | 2.708 |
| | Ave. count time | 2.38 | 2.40 | 2.48 | 2.50 | 2.51 | 2.36 |
| | Ave. locate time | 19.79 | 19.72 | 19.85 | 19.81 | 19.44 | 19.52 |

| | Superbucket size | 128K | 256K | 512K | 1024K | 2048K | 4096K |
|---|---|---|---|---|---|---|---|
| MTF | Compression ratio | 2.832 | 2.955 | 3.108 | 3.233 | 3.350 | 3.500 |
| | Ave. count time | 2.42 | 2.28 | 2.48 | 2.38 | 2.52 | 2.44 |
| | Ave. locate time | 19.35 | 19.06 | 19.97 | 18.92 | 18.77 | 19.97 |
| MTF2 | Compression ratio | 2.809 | 2.932 | 3.085 | 3.211 | 3.328 | 3.477 |
| | Ave. count time | 2.41 | 2.51 | 2.48 | 2.44 | 2.47 | 2.44 |
| | Ave. locate time | 19.51 | 19.26 | 19.23 | 19.09 | 19.14 | 18.83 |

Table 4.3: Compression ratio (bits/symbol) and average time (milliseconds) for the *count* and *locate* operations as a function of the superbucket size (bytes) for *bible.*

we use MTF2. From the results we can see that the size of superbuckets does not significantly influence the efficiency of *count* and *locate*, while the compression ratios change significantly with it. Figure 4.2 clearly shows the change of compression ratios. From it we can see that our method compresses the best when the size of superbucket is 16 times as large as that of buckets, no matter whether we adopt MTF or MTF2. Therefore, in our future experiments, we always ensure that $l_s : l_b = 16 : 1$.

Table 4.5 shows the results of compressing texts with MTF and MTF2 schemes. The size of superbuckets and buckets are 16Kb and 1Kb, respectively, and the fraction of marked rows is 2%. From the results, we can see that MTF2 is better than MTF on most of the input files, so we will therefore use MTF2 in the rest of our experiments.

| | Superbucket size | 2K | 4K | 8K | 16K | 32K | 64K |
|---|---|---|---|---|---|---|---|
| MTF | Compression ratio | 3.440 | 2.908 | 2.695 | 2.649 | 2.696 | 2.799 |
| | Ave. count time | 3.32 | 3.33 | 3.36 | 3.38 | 3.42 | 3.25 |
| | Ave. locate time | 20.14 | 20.17 | 20.58 | 20.51 | 20.45 | 21.35 |
| MTF2 | Compression ratio | 3.413 | 2.881 | 2.668 | 2.623 | 2.669 | 2.773 |
| | Ave. count time | 3.31 | 3.49 | 3.33 | 3.40 | 3.42 | 3.45 |
| | Ave. locate time | 19.78 | 20.10 | 19.95 | 19.89 | 19.81 | 19.73 |
| | Superbucket size | 128K | 256K | 512K | 1024K | 2048K | 4096K |
| MTF | Compression ratio | 2.931 | 3.079 | 3.247 | 3.411 | 3.570 | 3.721 |
| | Ave. count time | 3.36 | 3.40 | 3.38 | 3.07 | 3.27 | 3.31 |
| | Ave. locate time | 19.96 | 20.19 | 19.87 | 19.78 | 20.08 | 20.06 |
| MTF2 | Compression ratio | 2.904 | 3.053 | 3.221 | 3.384 | 3.544 | 3.694 |
| | Ave. count time | 3.08 | 3.45 | 3.54 | 3.31 | 3.09 | 3.48 |
| | Ave. locate time | 19.65 | 19.72 | 19.83 | 19.59 | 19.63 | 19.69 |

Table 4.4: Compression ratio (bits/symbol) and average time (milliseconds) for the *count* and *locate* operations as a function of the superbucket size (bytes) for *ohsumed.87*.

| File | *world192* | *canterbury* | *bible* | *E.coli* | *shakespeare* |
|---|---|---|---|---|---|
| MTF | 2.714 | 3.636 | 2.607 | 8.003 | 2.401 |
| MTF2 | 2.712 | 3.615 | 2.585 | 7.999 | 2.370 |
| File | *ohsumed.87* | *gutenberg* | *dblp* | *rfc* | *jdk14* |
| MTF | 2.649 | 2.921 | 1.862 | 2.682 | 1.369 |
| MTF2 | 2.623 | 2.925 | 1.861 | 2.680 | 1.380 |

Table 4.5: Comparison of the compression ratios (bits/symbol) between two move-to-front schemes.

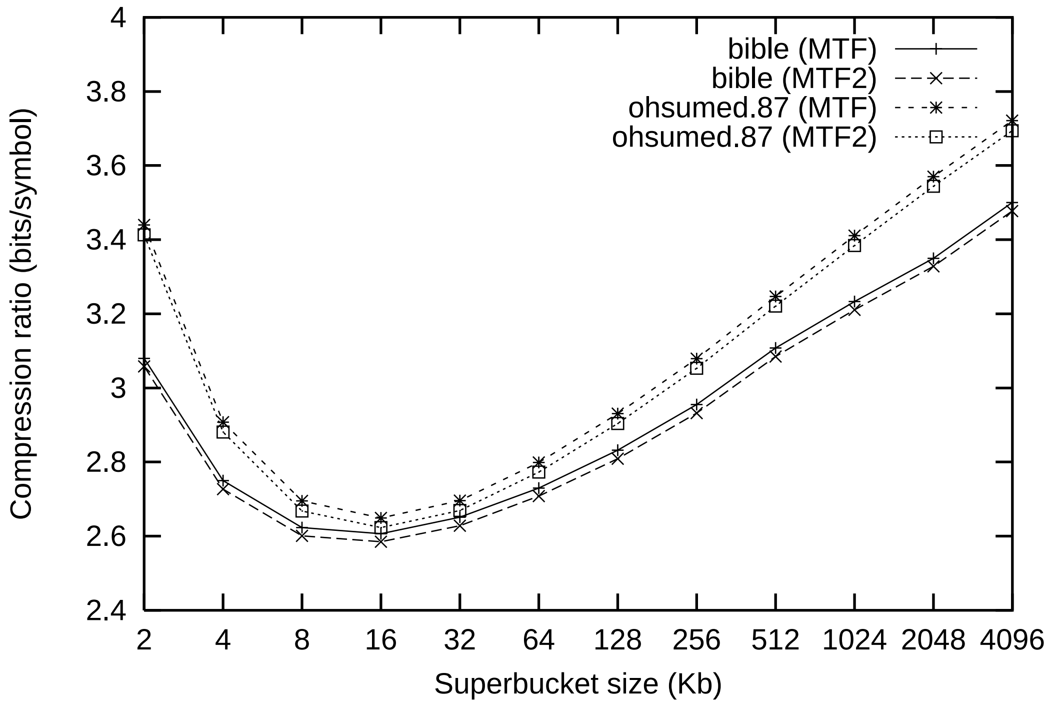

Figure 4.2: Compression ratio (bits/symbol) as a function of the superbucket size.

| | *bible* | | | | *ohsumed.87* | | | |
|---|---|---|---|---|---|---|---|---|
| Bucket size | 1K | 2K | 4K | 8K | 1K | 2K | 4K | 8K |
| Compression ratio | 2.585 | 2.341 | 2.198 | 2.111 | 2.623 | 2.335 | 2.164 | 2.061 |
| Ave. count time | 2.50 | 3.45 | 4.90 | 8.25 | 3.40 | 4.14 | 5.76 | 9.29 |
| Ave. locate time | 19.81 | 27.74 | 43.80 | 75.48 | 19.89 | 26.68 | 41.54 | 70.62 |

Table 4.6: Compression ratio (bits/symbol) and average time (milliseconds) for the *count* and *locate* operations as a function of the bucket size (bytes).

### 4.5.3 Size of Buckets

Size of buckets influences both search time and compression efficiency. Smaller buckets cost less time to decompress, and therefore they enable more efficient computation of the function *Occ*. On the other hand, larger buckets improve compression efficiency by reducing the auxiliary information associated with the buckets, and making the coding of MTF and MTH more efficient.

Table 4.6, Figure 4.3, Figure 4.4, and Figure 4.5 report the results. We can see that as the bucket size increases, the execution time for *count* and *locate* operations increase dramatically, while the compression efficiency drops less significantly. Therefore we choose to use small bucket size, which is 1Kb in our experiments.

### 4.5.4 Percentage of Marked Rows

It is clear that this parameter introduces a trade-off between compression efficiency and searching speed: If we choose to mark more rows, then the index we built will cost more space, but the *locate* operation will be executed more efficiently.

Table 4.7 reports the results of our experiments on FM-index with different percentage of marked rows and different bucket sizes. The average time of *count*

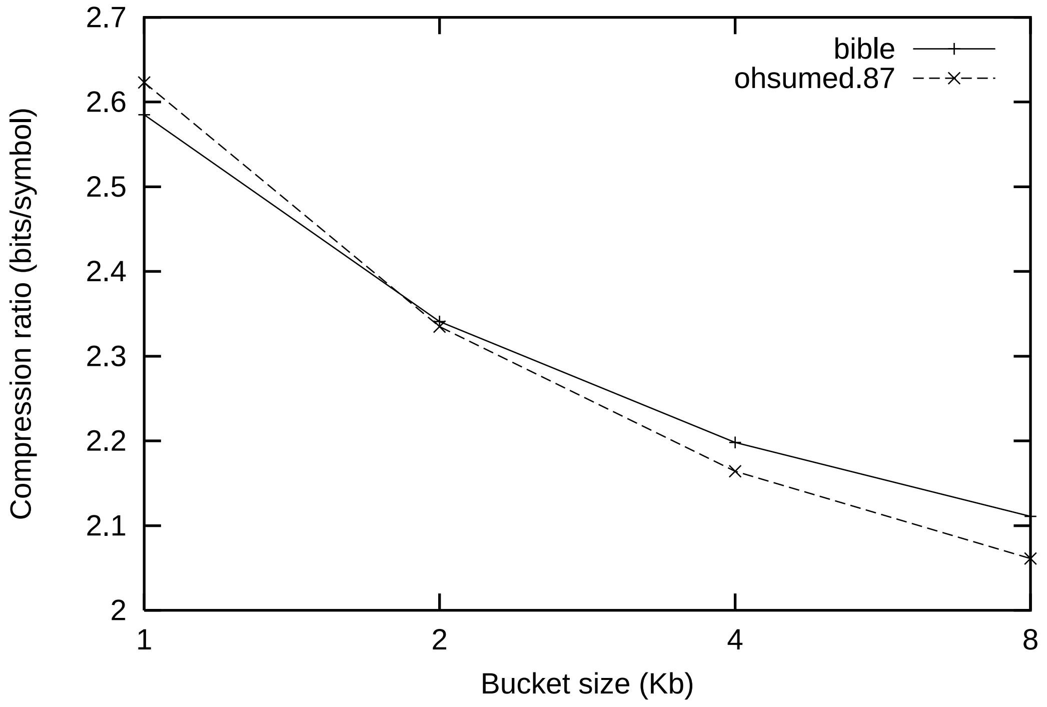

Figure 4.3: Compression ratio (bits/symbol) as a function of the bucket size.

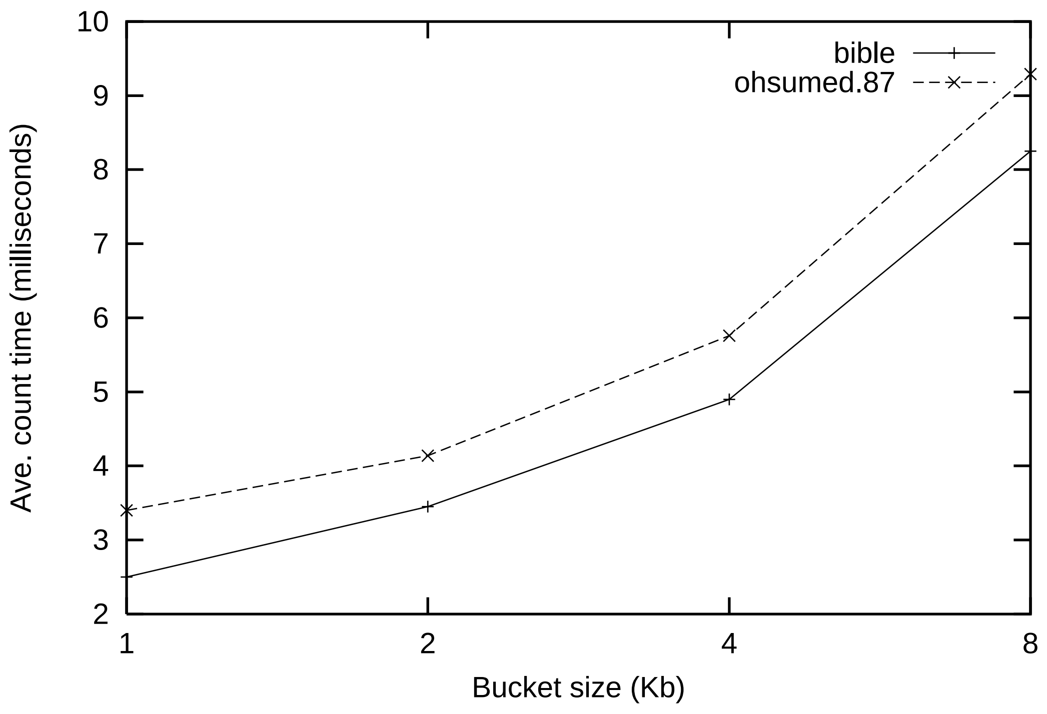

Figure 4.4: Average time (milliseconds) for the *count* operation as a function of the bucket size.

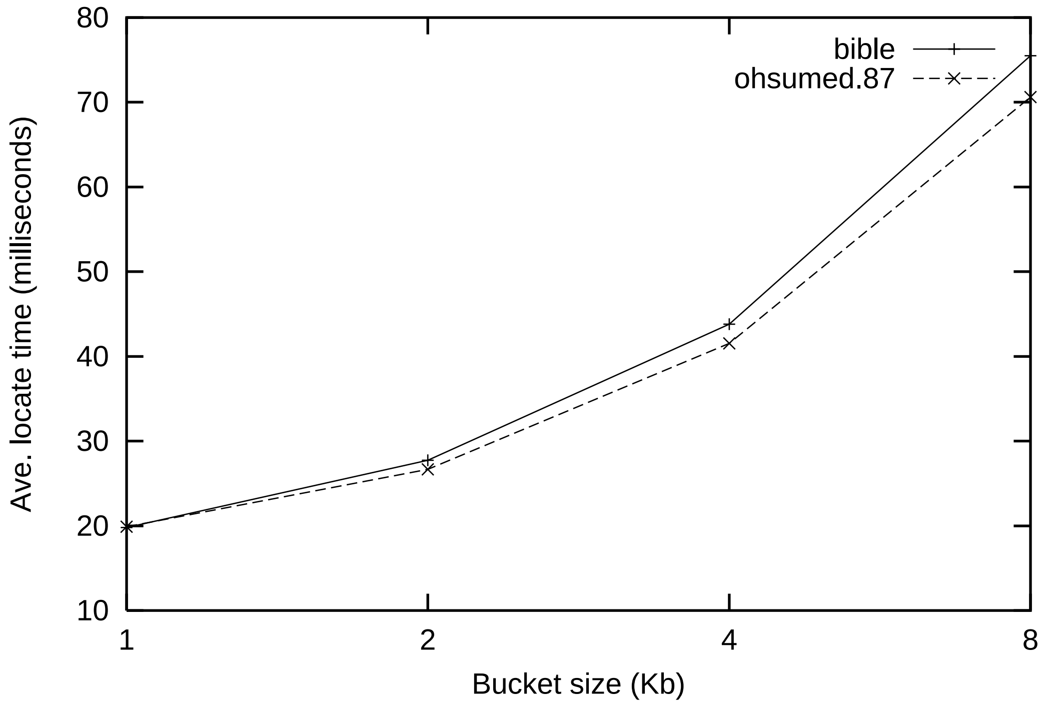

Figure 4.5: Average time (milliseconds) for the *locate* operation as a function of the bucket size.

| | | *bible* | | | |
|---|---|---|---|---|---|
| | Bucket size | 1Kb | 2Kb | 4Kb | 8Kb |
| 1% | Compression ratio | 2.363 | 2.119 | 1.975 | 1.889 |
| | Ave. locate time | 46.02 | 66.81 | 105.04 | 181.09 |
| 2% | Compression ratio | 2.585 | 2.341 | 2.198 | 2.111 |
| | Ave. locate time | 19.81 | 27.74 | 43.80 | 75.48 |
| 5% | Compression ratio | 3.317 | 3.073 | 2.930 | 2.844 |
| | Ave. locate time | 6.51 | 8.92 | 14.22 | 24.73 |
| 10% | Compression ratio | 4.298 | 4.055 | 3.911 | 3.825 |
| | Ave. locate time | 3.15 | 4.40 | 7.00 | 11.98 |
| | | *ohsumed.87* | | | |
| | Bucket size | 1Kb | 2Kb | 4Kb | 8Kb |
| 1% | Compression ratio | 2.312 | 2.025 | 1.853 | 1.750 |
| | Ave. locate time | 51.74 | 70.38 | 109.41 | 185.69 |
| 2% | Compression ratio | 2.623 | 2.335 | 2.164 | 2.061 |
| | Ave. locate time | 19.89 | 26.68 | 41.54 | 70.62 |
| 5% | Compression ratio | 3.478 | 3.191 | 3.019 | 2.916 |
| | Ave. locate time | 6.56 | 9.32 | 14.38 | 24.38 |
| 10% | Compression ratio | 4.279 | 3.992 | 3.820 | 3.718 |
| | Ave. locate time | 4.11 | 5.81 | 8.99 | 15.33 |

Table 4.7: Compression ratio (bits/symbol) and average time (milliseconds) for the *count* and *locate* operations as a function of the bucket size (bytes) and percentage of marked characters.

operation is not reported since it is not influenced by the percentage of marked rows. The results suggest that it is preferable to use small buckets. One suitable choice is to set the size of buckets to be 1Kb, and the fraction of marked rows to be 2%. We will continue to use this setting in the rest of our experiments.

### 4.5.5 More Results

We also test the FM-index on all of our input files in Table 4.2, and Table 4.8 summarizes the results. Among the results, one exception is *E.coli*, its compression ratio is 7.999 bits/symbol, much higher than any other files. This is because it has an alphabet size of 4, and the frequencies of the four alphabet symbols are close to each other. When we choose to mark 2% of the marked rows, we actually choose a symbol whose frequency is close to 25%. Although we can use a certain simple scheme to solve the problem (For example, we only mark part of the rows that end with the chosen character, such as those corresponding result returned from function $Occ(c, 0, k)$ is divided by a given constant number), we choose not to further work on the file. The reason is that BWT is not a suitable method to compress DNA sequences. From Table 2.9 we see that the compression ratio for *E.coli* with BWT is more than 2 bits/symbol, while a simple coding scheme that encode each symbol with 2 bits (since the alphabet size is 4) only costs 2 bits/symbol. For all the other files, the results are mostly consistent, although the *locate* operation on *jdk14* is much worse than others. This is because the occurrences of a given character are less evenly distributed in *jdk14*.

### 4.5.6 Comparisons

First we compare the compression ratio. Table 4.9 lists the results. The row labeled FM-index shows the results with the FM-index we implemented. The row labeled BWT shows the results with the BWT compression algorithm we implemented in Section 2.4, with MTF2 scheme. The row labeled bzip2 shows the results with the

| File | *world192* | *canterbury* | *bible* | *E.coli* | *shakespeare* |
|---|---|---|---|---|---|
| Compression ratio | 2.712 | 3.615 | 2.585 | 7.999 | 2.370 |
| Ave. count time | 2.36 | 3.24 | 2.50 | 6.99 | 2.88 |
| Ave. locate time | 33.66 | 8.26 | 19.81 | 1.67 | 19.61 |
| File | *ohsumed.87* | *gutenberg* | *dblp* | *rfc* | *jdk14* |
| Compression ratio | 2.623 | 2.925 | 1.861 | 2.680 | 1.380 |
| Ave. count time | 3.40 | 3.05 | 3.41 | 3.23 | 1.94 |
| Ave. locate time | 19.89 | 11.75 | 15.87 | 22.77 | 43.41 |

Table 4.8: Compression ratio (bits/symbol) and average time (milliseconds) for the *count* and *locate* operations on different types of input files.

bzip2 program [52], which is an implementation of BWT compression algorithm. The row labeled gzip shows the results with gzip. We can see that the construction and decompression speeds of our index are comparable to those of the other schemes, while the compression ratio is close to and sometimes better than that of gzip.

We also compared the search time of FM-index with other search programs: grep, zgrep (i.e. grep over gzipped files) and bzgrep(i.e. grep over bizipped files). These programs use scan-based algorithms. Table 4.10 reports the results. From the reports we can see that the counting queries can be performed with FM-index very efficiently. However, the average search time of FM-index is often worse than that of grep, and sometimes, even worse than that of zgrep. This is because for our choice of test sets, one pattern may occur many times in a text, so the average search time of FM-index is not the sum of average time for *count* and *locate* operations. Combined with other results, we can see that FM-index is very efficient for existential and counting query, but may not be efficient for enumerative query. However, this is not a serious problem in most applications. First, in most cases, users will not be able to browse the results in a single screen if there are many occurrences, while

| File | | *world-192* | *canterbury* | *bible* | *E.coli* | *shakespeare* |
|---|---|---|---|---|---|---|
| FM-index | Compression ratio | 2.712 | 3.615 | 2.585 | 7.999 | 2.370 |
| | Construction speed | 2.76 | 2.27 | 2.55 | 3.16 | 3.80 |
| | Decompression speed | 1.04 | 1.02 | 1.07 | 1.28 | 1.14 |
| BWT | Compression ratio | 1.387 | 1.655 | 1.559 | 2.116 | 1.374 |
| | Construction speed | 2.37 | 2.00 | 2.53 | 2.75 | 3.44 |
| | Decompression speed | 0.99 | 0.96 | 1.04 | 1.27 | 1.05 |
| bzip2 | Compression ratio | 1.584 | 1.619 | 1.671 | 2.158 | 1.539 |
| | Construction speed | 1.45 | 0.97 | 1.17 | 1.27 | 1.45 |
| | Decompression speed | 0.43 | 0.38 | 0.56 | 0.52 | 0.42 |
| gzip | Compression ratio | 2.333 | 2.088 | 2.326 | 2.240 | 2.212 |
| | Construction speed | 0.73 | 4.66 | 1.43 | 9.07 | 0.922 |
| | Decompression speed | 0.08 | 0.09 | 0.08 | 0.09 | 0.08 |

| File | | *ohsumed.87* | *gutenberg* | *dblp* | *rfc* | *jdk14* |
|---|---|---|---|---|---|---|
| FM-index | Compression ratio | 2.623 | 2.925 | 1.861 | 2.680 | 1.380 |
| | Construction speed | 4.29 | 4.85 | 7.33 | 5.00 | 13.82 |
| | Decompression speed | 1.60 | 1.69 | 1.31 | 1.66 | 1.23 |
| BWT | Compression ratio | 1.406 | 1.672 | 0.730 | 1.244 | 0.304 |
| | Construction speed | 3.59 | 4.18 | 6.64 | 4.52 | 13.10 |
| | Decompression speed | 1.46 | 1.55 | 1.22 | 1.56 | 1.17 |
| bzip2 | Compression ratio | 1.931 | 2.160 | 0.957 | 1.542 | 0.446 |
| | Construction speed | 1.22 | 1.27 | 1.85 | 1.11 | 2.39 |
| | Decompression speed | 0.45 | 0.49 | 0.36 | 0.39 | 0.31 |
| gzip | Compression ratio | 2.663 | 2.969 | 2.623 | 2.607 | 0.731 |
| | Construction speed | 0.70 | 1.12 | 0.47 | 1.50 | 0.31 |
| | Decompression speed | 0.09 | 0.10 | 0.07 | 0.08 | 0.05 |

Table 4.9: Compression ratio (bits/symbol) and compression/decompression speed (microseconds/symbol) of the FM-index compared with those of gzip (with option -9 for maximum compression) and bzip2 (version 1.0.2).

computing the occurrences that can be listed in a single screen is very fast. Second, in practical applications, users usually search for more sensible phrases that seldom occur thousands of times, while a pattern that occurs thousands of times in large texts such as *jdk14* is very common in our test set.

### 4.5.7 Caching

Table 4.11, Figure 4.6, Figure 4.7 and Figure 4.8 reports the search time and hit ratios for different cache sizes for *bible* and *ohsumed.87*. The average search time decreases and the hit ratio increases, as the cache size increases. We achieve better performance with the caching scheme. However, the *count* operation may not be improved. This is because in a single *count* operation, we only call *Occ* a few times, while since we need to cache the decompressed results, we have to decompress the whole bucket when needed. Recall that we only decompress part of a bucket if we do not implement the caching techniques. From the figures, we can also see that the improvement of hit ratios and searching efficiency increases less rapidly as the size of the memory used for caching increases.

We choose the cache size to be 2% of the size of the original text and test our caching techniques on the other files. The results are reported in Table 4.12. We can see that the average search speed on each file is improved notably.

### 4.5.8 Summary

From our experiments, we suggest to use the following parameters:

1. Size of superbuckets: 16Kb,

| File | | *world-192* | *canterbury* | *bible* | *E.coli* | *shakespeare* |
|---|---|---|---|---|---|---|
| FM-index | Compression ratio | 2.712 | 3.615 | 2.585 | 7.999 | 2.370 |
| | Ave. count time | 2.36 | 3.24 | 2.50 | 6.99 | 2.88 |
| | Ave. locate time | 33.66 | 8.26 | 19.81 | 1.67 | 19.61 |
| | Ave. search time | 457.81 | 76.79 | 498.44 | 23.58 | 580.59 |
| zgrep | Compression ratio | 2.333 | 2.088 | 2.326 | 2.240 | 2.212 |
| | Ave. search time | 323.69 | 341.25 | 487.39 | 784.20 | 894.18 |
| bzgrep | Compression ratio | 1.584 | 1.619 | 1.671 | 2.158 | 1.539 |
| | Ave. search time | 1033.93 | 973.85 | 1666.03 | 2633.69 | 3128.41 |
| grep | Compression ratio | 8.000 | 8.000 | 8.000 | 8.000 | 8.000 |
| | Ave. search time | 125.85 | 118.65 | 175.33 | 431.23 | 388.97 |

| File | | *ohsumed.87* | *gutenberg* | *dblp* | *rfc* | *jdk14* |
|---|---|---|---|---|---|---|
| FM-index | Compression ratio | 2.623 | 2.925 | 1.861 | 2.680 | 1.380 |
| | Ave. count time | 3.40 | 3.05 | 3.41 | 3.23 | 1.94 |
| | Ave. locate time | 19.89 | 11.75 | 15.87 | 22.77 | 43.41 |
| | Ave. search time | 5184.79 | 4870.05 | 20007.00 | 17712.30 | 26104.10 |
| zgrep | Compression ratio | 2.663 | 2.969 | 2.623 | 2.607 | 0.731 |
| | Ave. search time | 6830.12 | 7788.04 | 12179.80 | 15528.10 | 11372.90 |
| bzgrep | Compression ratio | 1.931 | 2.160 | 0.957 | 1.542 | 0.446 |
| | Ave. search time | 25466.50 | 28761.00 | 43026.30 | 53600.40 | 41657.60 |
| grep | Compression ratio | 8.000 | 8.000 | 8.000 | 8.000 | 8.000 |
| | Ave. search time | 2707.13 | 2777.68 | 6224.82 | 6828.23 | 6570.32 |

Table 4.10: Compression ratio (bits/symbol) and average search time (milliseconds) of the FM-index compared with other searching tools.

| | *bible* | | | | | |
|---|---|---|---|---|---|---|
| Cache size | 1 | 2 | 3 | 4 | 5 | 10 |
| Ave. count time | 2.10 | 2.11 | 2.14 | 2.13 | 2.17 | 2.22 |
| Ave. locate time | 17.45 | 17.07 | 17.02 | 16.81 | 16.58 | 16.39 |
| Ave. search time | 439.09 | 429.78 | 428.56 | 423.30 | 417.61 | 413.09 |
| Hit ratio | 19.25 | 25.80 | 30.54 | 34.64 | 38.35 | 53.19 |
| | *ohsumed.87* | | | | | |
| Cache size | 1 | 2 | 3 | 4 | 5 | 10 |
| Ave. count time | 3.12 | 2.97 | 3.10 | 3.19 | 3.33 | 3.26 |
| Ave. locate time | 18.12 | 18.11 | 18.08 | 17.71 | 17.63 | 17.49 |
| Ave. search time | 4726.54 | 4725.05 | 4718.06 | 4623.77 | 4603.77 | 4572.39 |
| Hit ratio | 26.28 | 32.95 | 38.07 | 42.33 | 46.01 | 59.32 |

Table 4.11: Average search time (milliseconds) and hit ratio (percentage) as a function of the cache size (percentage of the size of the original file).

| File | *world192* | *canterbury* | *bible* | *E.coli* | *shakespeare* |
|---:|---:|---:|---:|---:|---:|
| Ave. count time | 2.37 | 3.10 | 2.11 | 7.19 | 2.93 |
| Ave. locate time | 33.21 | 8.19 | 17.07 | 1.57 | 20.77 |
| Ave. search time | 451.84 | 76.18 | 429.78 | 22.84 | 615.26 |
| Hit ratio | 34.81 | 30.65 | 25.80 | 44.85 | 20.05 |
| File | *ohsumed.87* | *gutenberg* | *dblp* | *rfc* | *jdk14* |
| Ave. count time | 2.97 | 3.36 | 3.65 | 3.42 | 2.46 |
| Ave. locate time | 18.11 | 10.95 | 14.10 | 20.15 | 35.28 |
| Ave. search time | 4725.05 | 4547.30 | 17772.9 | 15681.5 | 21218.2 |
| Hit ratio | 32.95 | 50.76 | 69.91 | 65.51 | 95.84 |

Table 4.12: Average search time (milliseconds) and hit ratio (percentage) on different types of input files when cache size is 2% of the original text.

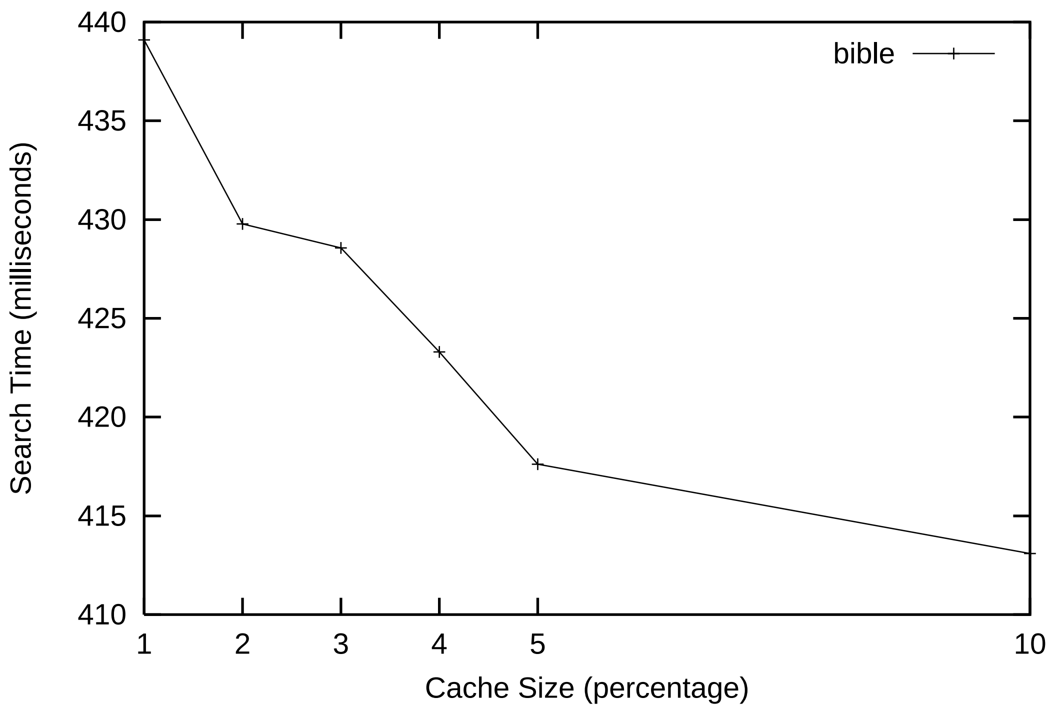

Figure 4.6: Average search time (milliseconds) as a function of the cache size (percentage) for *bible*.

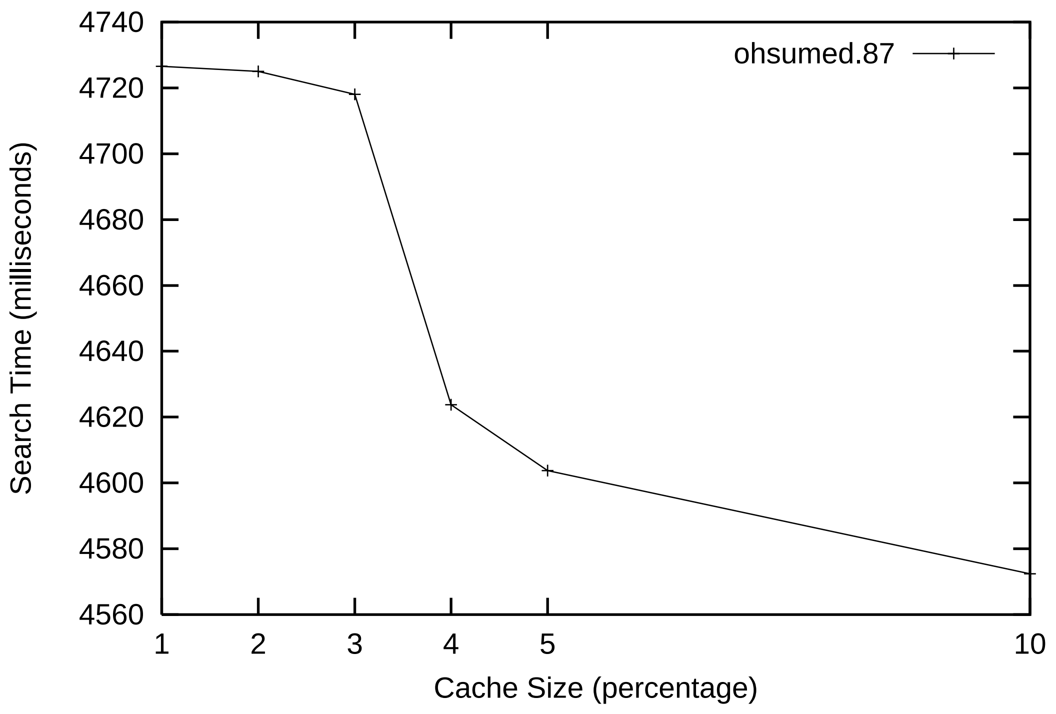

Figure 4.7: Average search time (milliseconds) as a function of the cache size (percentage) for *ohsumed.87*.

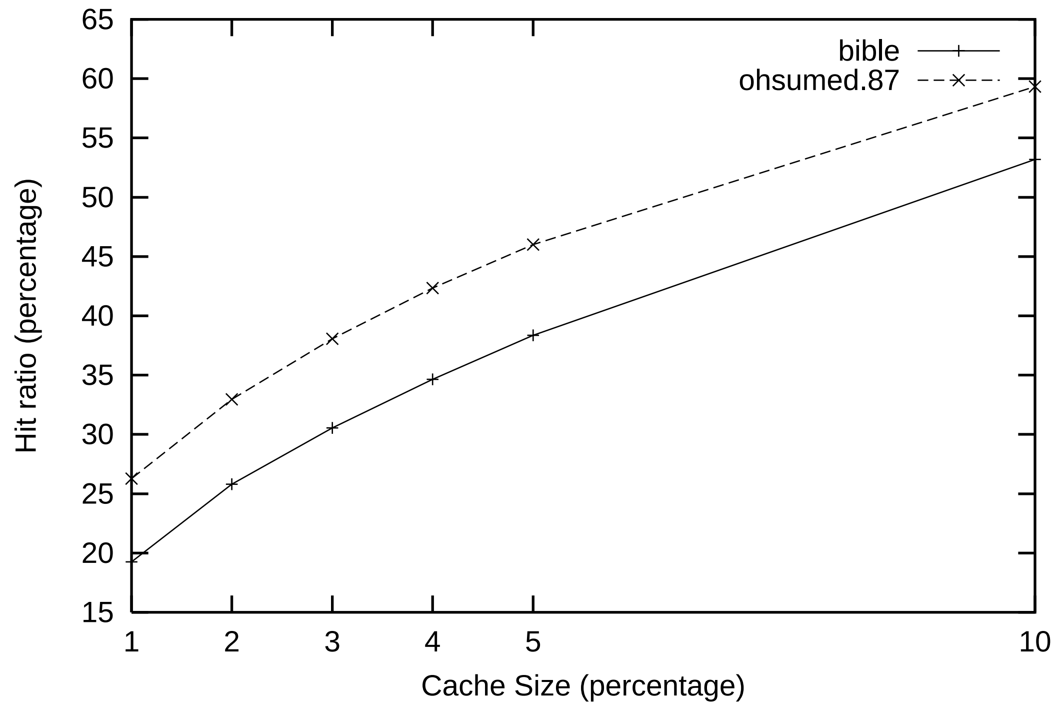

Figure 4.8: Hit ratio (percentage) as a function of the cache size (percentage).

2. Size of buckets: 1kb,

3. Fraction of marked rows in $M$: 2%,

4. Cache size: 2% of the size of the original text.

This set of parameters prove to be efficient in our experiments, though other choices may also be suitable.

We also conclude that MTF2 is superior to MTF in most cases.

Our experiments also show that FM-index is compact and supports efficient query.

# Chapter 5

# Conclusion

In this thesis, we have studied three topics on text processing: text compression, text indexing, and techniques to combine text compression and indexing. The first are usually studied as two independent techniques: one is for reducing the storage cost for storing texts, the other is for facilitating searching through the texts. We have given a survey on various compression techniques and implemented an efficient BWT-based compression algorithm through experiments. We have also presented two most used indexing techniques: inverted files and suffix trees. The third technique is motivated by the need to facilitate searching through compressed texts. We have reviewed the SASE, which is a compressed word index, and the opportunistic index, which is a compressed full-text index. We also designed a new MTF scheme when considering the implementation of the opportunistic index.

These studies led to the design and implementation of a method to build indices for texts that have been compressed with Burrows-Wheelers compression algorithm. The method is mainly based upon the FM-index proposed and studied in [28], which is an implementation of the opportunistic index, but we have also tested some of our ideas. We have determined a set of suitable parameters for the organization of FM-index, and designed a caching technique to improve the performance. The FM-index we built achieves compression performance close to, and sometimes better than, that achieved by gzip, but supports fast query at the same time, especially existential and counting queries.

Some future development includes:

- Design more efficient methods for marking rows.
- Implement more complex queries, such as regular expression queries and approximate queries.
- Design more caching techniques, such as caching some search results in memory for future queries.

# Bibliography

[1] JDK 1.4 documentation. http://java.sun.com/j2se/1.4/.

[2] Project Gutenberg. http://promo.net/pg/.

[3] RFC collection. http://www.rfc-editor.org/download.html/.

[4] Shakespeare xml database. http://www.ibiblio.org/xml/examples/shakespeare/.

[5] TREC collection. http://trec.nist.gov/data.html.

[6] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[7] A. Andersson. Sorting and searching revisited. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, pages 185–197, 1996.

[8] R. Arnold and T. Bell. The Canterbury corpus home page. http://corpus.canterbury.ac.nz.

[9] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[10] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the burrows and wheeler data compression algorithm. In *Data Compression Conference*, pages 188–197.

[11] B. Balkenhol and Y. M. Shtarkov. One attempt of a compression algorithm using the bwt, 1999.

[12] T. C. Bell. Better OPM/L text compression. *IEEE Transactions on Communications*, 34:1176–1182, December 1986.

[13] T. C. Bell. *An unifying theory and improvements for existing approaches to text compression*. PhD thesis, Department of Computer Science, University of Canterbury, New Zealand, 1987.

[14] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1990.

[15] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[16] R. P. Brent. A linear algorithm for data compression. *Australian Computer Journal*, 19(2):64–68, May 1987.

[17] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2-3):76–93, 1997.

[18] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[19] T. Chiueh and S. Varadarajan. Compression-Domain text indexing and retrieval. Technical Report ECSL-TR-25, Experimental Computer Systems Lab, Deparment of Computer Science, State University of New York at Stony Brook, 1997.

[20] T. Chiueh and S. Varadarajan. SASE: implementation of a compressed text search engine. Technical Report ECSL-TR-41, Experimental Computer Systems Lab, Deparment of Computer Science, State University of New York at Stony Brook, 1997.

[21] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. of the 7th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[22] Sebastian Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software Practice and Experience*, 32(2):99–111, 2002.

[23] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, March 1985.

[24] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[25] P. Fenwick. Block sorting text compression – final report. Technical Report 130, University of Auckland, 1996.

[26] P. Ferragina and G. Manzini. Opportunistic data structures with applications. Technical Report TR-00-03, University of Pisa, 2000.

[27] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[28] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. of ACM-SIAM SODA*, pages 269–278, 2001.

[29] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[30] W. B. Frakes. Stemming algorithms. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 8, pages 131–160. Prentice-Hall, 1992.

[31] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[32] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[33] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 3, pages 28–43. Prentice-Hall, 1992.

[34] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the IRE*, 40(9), 1952.

[35] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[36] Jakobsson. Compression of character strings by an adaptive dictionary. *BIT*, 25(4):593–603, 1985.

[37] S. T. Klein, A. Bookstein, and S. DeerWester. Storing text retrieval systems on CD-Room: compression and encryption considerations. *ACM Transaction on Information Retrieval Systems*, 7(3):230–245, July 1989.

[38] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2 edition, 1998.

[39] M. Ley. DBLP bibliography. http://www.informatik.uni-trier.de/ ley/db/index.html.

[40] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[41] G. Manzini. An analysis of the Burrows-Wheeler transform. Technical Report B4-99-13, University of Pisa, 1999.

[42] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[43] V. S. Miller and M. N. Wegman. Variations on a scheme by ziv and lempel. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 131–140. Springer-Verlag, Berlin, 1984.

[44] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–524, October 1968.

[45] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–306, 1998.

[46] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[47] J. I. Munro. Succinct data structures. In *Proceedings of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science.* Springer-Verlag LNCS n. 1738, 1999.

[48] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[49] M. Nelson. LZW data compression. *Dr. Dobb's Journal*, pages 29–36, 86–87, October 1989.

[50] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

[51] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix arrays. In *Proceedings of ISACC'00*, number 1969, pages 410–421, 2000.

[52] J. Seward. The bzip2 and libbzip2 official home page. http://sources.redhat.com/bzip2/.

[53] J. Seward. On the performance of BWT sorting algorithms. In *Proceedings of the IEEE Data Compression Conference*, pages 28–30.

[54] S. W. Thomas, J. Mckie, S. Davies, K. Turkowski, J. A. Woods, and J. W. Orost. Compress (version 4.0) program and documentation, 1985.

[55] P. Tischer. A modified Lempel-Ziv-Welch data compression scheme. *Australian Computer Science Communications*, 9(1):262–272, 1987.

[56] P. A. J. Volf and F. M. J. Willems. Switching between two universal source coding algorithms. In *Proceedings of the IEEE Data Compression Conference*, pages 491–500.

[57] P. A. J. Volf and F. M. J. Willems. The switching method: elaborations. In *Proceedings of the 19'th Information Theory in Benelux*, pages 13–20.

[58] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11.

[59] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17:8–19, 1984.

[60] D. Wheeler. Improving Huffman coding. ftp://ftp.cl.cam.ac.uk/users/djw3/, 1997.

[61] I. Witten and T. Bell. The Calgary corpus. ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus.

[62] I. H. Witten, R. M. Neal, and J. G. Gleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[63] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, May 1977.

[64] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, September 1978.

[65] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.