The suffix-signature method for searching for phrases in text

by

Mei Zhou

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1997

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Finding all occurrences of a given word in a large static text is a well-studied problem. Most solutions, however, are not well-suited for phrase-searching. In this thesis, we investigate a new algorithm to find all occurrences of a given phrase in a large, static text, based on the data structure known as a *suffix array*. Using this algorithm, phrases of bounded length can be found with expected search time of one disk access to the text and one disk access to an index. To achieve this performance for phrases of up to five words in length requires an index having total size of approximately 120% of the size of the text. The algorithm guarantees a worst case search performance of 2 disk accesses to the text per phrase search.

The method augments a suffix array with a parallel signature array, so that every indexed phrase has an associated signature. To search for a phrase, we search a block of the index in memory to locate matching signatures. Then we read one or two phrases corresponding to matching signatures from disk and compare them to the target phrase to filter out false matches.

We present theoretical properties of the data structure and algorithm derived from a suitable model. The theoretical results have been validated through experimentation with actual data ranging in size from 6Mb to 550Mb and also including actual query patterns derived from logs of searches on the World Wide Web. These experiments show that the approach is applicable in practice to a variety of texts and realistic phrase searches.

Acknowledgements

I am most grateful to my supervisor, Dr. Frank W. Tompa, for his guidance, suggestions, encouragement, and patience; without these, I could not have completed this thesis. I would also like to thank my examining committee members: Dr. Gordon V. Cormack, Dr. George H. Freeman, Dr. Udi Manber of the University of Arizona, and Dr. Ian Munro for their valuable suggestions. Tim Snider provided me with much advice and help on writing software for text management. Open Text Corporation provided me with the tapes of web page data and queries, for which I am also grateful.

Finally I acknowledge the ongoing financial support I received from NSERC, the University of Waterloo, Open Text, and ITRC.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents a new access method to search for phrases in large static texts. In this chapter, we first review text search techniques. Then we briefly describe the proposed method. We conclude the chapter with an overview of the thesis.

1.1 Review of text searching

In text retrieval systems, word or phrase search is a very important operation. We can either scan all the text, which takes a long time if it is very long, or we can pre-process the text to build indices that can be used later when we search [Gon83]. Many text retrieval index methods have been studied and used, including inverted lists [HFBYL92], tries [Knu73], suffix arrays [GBYS92, MM93], and signature files [FC84].

Full text scanning can use the Knuth-Morris-Pratt algorithm [KMP77], Boyer-Moore algorithm [BM77], Karp-Rabin algorithm [KR87] and various improvements of these algorithms [Aho90]. Baeza-Yates presented detailed analyses of these al-

gorithms [BY89]. The advantage of full text scanning methods is that they require no space overhead and minimal effort on insertions and updates, because no indices have to be maintained and changed. The disadvantage is poor response time [Fal85].

Signature file methods are based on the idea of an inexact filter [Fal92], which quickly discards many of the unqualifying items. The qualifying items definitely pass the filter, but some additional items might also pass it coincidentally. Many methods on signature files have been suggested, trying to improve the response time and trading off space or insertion simplicity for speed [Fal92]. Signature methods are much faster than full text scanning. But they may be slow for large databases, since their response time remains linear in the number of items in the database.

An inverted index [Knu73] is a set of postings lists [HFBYL92, WMB94], each of which maps one keyword to a list of links to the documents containing that keyword. Inverted indices can be implemented as sorted arrays, B-trees, tries and various hashing structures [HFBYL92]. A special case of B-tree, the prefix B-tree, uses prefixes of words as primary keys in a B-tree index [BU77] and is particularly suitable for text databases. Tries are alternative recursive tree structures that use the digital decomposition of strings to represent a set of strings and to direct the searching [dlB59, Fre60]. The storage overhead for a word-level implementation is 30-100 percent of the original file size [GBYS92], or about 10 percent after removal of common words and applying compression [WMB94]. The advantage of inverted files is that they are fast and relatively easy to implement [Fal85].

A Patricia tree [Mor68, Knu73, FS86] is a binary trie with the additional constraint that single-descendant nodes are eliminated. A PAT[1] tree is a Patricia tree on all substrings of a text [Gon83, Gon88, Sha95, Cla96]. A PAT array keeps only

[1]PAT is a registered trademark of Open Text Corporation.

external nodes of a PAT tree. A PAT array is also called a suffix array, and was independently discovered by Gonnet [GBYS92] and Manber and Myers [MM93]. Gonnet used it in the PAT system to support fast text search for phrases, as part of the *Oxford English Dictionary* project [Tom91]. PAT also provides structures to support fielded searches, such as to find something specific within certain regions, or to find regions that include certain phrases or other regions [ST93, Li96]. To search for a phrase using a suffix array takes $\Theta(\log n)$ disk accesses to a text of n indexed phrases, which is a great improvement over using traditional postings lists when frequently occurring words are commonly used in phrases [GBYS92].

1.1.1 Conventional signature files

One important auxiliary structure for searching is the signature file [Fal92]. A text file on secondary storage consists of a sequence of blocks. Associated with each block is an integer signature, as described below. The text file is then represented by its signature file which contains all its block signatures and is much shorter than the original text file.

A signature function, similar to a hash function [Knu73], can be used to convert a word (a character string) to an integer. The signature of a block is obtained by bitwise *OR*-ing the signatures of all words in the block. Similarly, the signature of a phrase is obtained by bitwise *OR*-ing word signatures in the phrase. Thus, by definition, if a block contains a phrase p, the signature of the block must bitwise contain the signature of the phrase p.

Figure 1.1 is an example of a block signature. For this simple example, the block signature and word signatures are all 32 bits long. In our example the first word in the block has 3 bits set, the 2nd to the 5th words have 7, 6, 5, and 4 bits

set to 1 respectively. The block with phrase "book is red and old" has a signature 00000111 11010111 11010110 01100011. with 18 bits set to 1.

word		signature			
(1st: 3 bits set)	book	00000010	00010000	00010000	00000000
(2nd: 7 bits set)	is	00000010	00000011	01000010	00100001
(3rd: 6 bits set)	red	00000001	00010001	01000000	01100000
(4th: 5 bits set)	and	00000100	01000100	10000000	00000010
(5th: 4 bits set)	old	00000000	10000000	01000100	01000000
block signature		00000111	11010111	11010110	01100011

Figure 1.1: Illustration of a block signature

Instead of searching through a whole text file for a phrase. we go through its much shorter signature file to search for blocks that contain the signature of the phrase. Since there are false hits. the phrase is checked against matching blocks to determine which blocks actually have the phrase. If the number of blocks that are checked is much smaller than the total number of blocks in a text file. the search time will be shortened tremendously. The conventional signature approach is suitable for multiple word fields and partial match queries. which specify a few words in each field.

A disjoint coding method was studied by Faloutsos [Fal88], in which each block has several fields, each field has a field signature like the one described in Figure 1.1. and the block signature is the concatenation of field signatures. It has been shown that under optimal design, half of the bits in a field signature should be set to 1 [Fal88]. This is different from conventional hashing, where each field is restricted to one word, queries become exact match queries, and a false hit occurs only if two

signatures are *equal* to each other instead of one *containing* the other. We examine this in more detail in Section 3.4.2.

1.1.2 Suffix array

A suffix array [GBYS92, MM93] is a list of pointers representing lexicographically sorted phrases stored on secondary storage.

Assume that we have a text consisting of the following string:

```
the dog, the cat, the horse, the donkey and the chicken
```

Each position in it indicates a *suffix* or *semi-infinite string* (sistring), which goes to the end of the phrase. There are about 50 characters in the example; therefore, there are 50 semi-infinite strings. We look at only semi-infinite strings that start at the beginning of words. There are 11 such strings:

```
the dog, the cat, the horse, the donkey and the chicken
    dog, the cat, the horse, the donkey and the chicken
         the cat, the horse, the donkey and the chicken
             cat, the horse, the donkey and the chicken
                  the horse, the donkey and the chicken
                      horse, the donkey and the chicken
                             the donkey and the chicken
                                 donkey and the chicken
                                        and the chicken
                                            the chicken
                                                chicken
```

Table 1.1 lists the above sistrings in lexicographical order. We store corresponding pointers of sorted sistrings in the suffix array. Figure 1.2 illustrates the situation showing the above text and its suffix array. The numbers above the text show positions of starting characters of these sistrings in the text. Sistrings in the text are referenced by pointers in the suffix array, and pointers are sorted in the lexicographical order of the 11 sistrings. The numbers under the suffix array indicate relative orders of these sistrings. For example, the 8th sistring "the chicken" starts at the 45th character position of the text.

1	and the chicken
2	cat, the horse, the donkey and the chicken
3	chicken
4	dog, the cat, the horse, the donkey and the chicken
5	donkey and the chicken
6	horse, the donkey and the chicken
7	the cat, the horse, the donkey and the chicken
8	the chicken
9	the dog, the cat, the horse, the donkey and the chicken
10	the donkey and the chicken
11	the horse, the donkey and the chicken

Table 1.1: Sorted sistrings in an example

In large text databases, the text must be stored on disk. The suffix array corresponding to sistrings at every word typically requires $60 - 80\%$ of the size of the text. When the suffix array grows too big to fit in memory, it too is kept on disk. The suffix array is then divided into blocks and an index to the blocks can

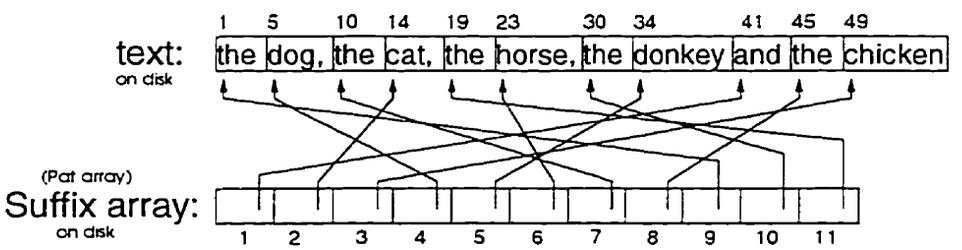


Figure 1.2: Suffix Array (PAT Array)

be used to reduce disk accesses. The block list contains a sorted list of words or phrases that delimit blocks. Normally, the block list is small enough to be kept in memory. Figure 1.3 illustrates the hierarchy.

Assume that we search for phrases starting with the word "the". From the block list, we could find out which block of the suffix array may contain the target phrases. Then we read that block of the suffix array into memory and perform a binary search on the suffix array block, accessing the text for each probe to determine whether to search to the right or the left within the array.

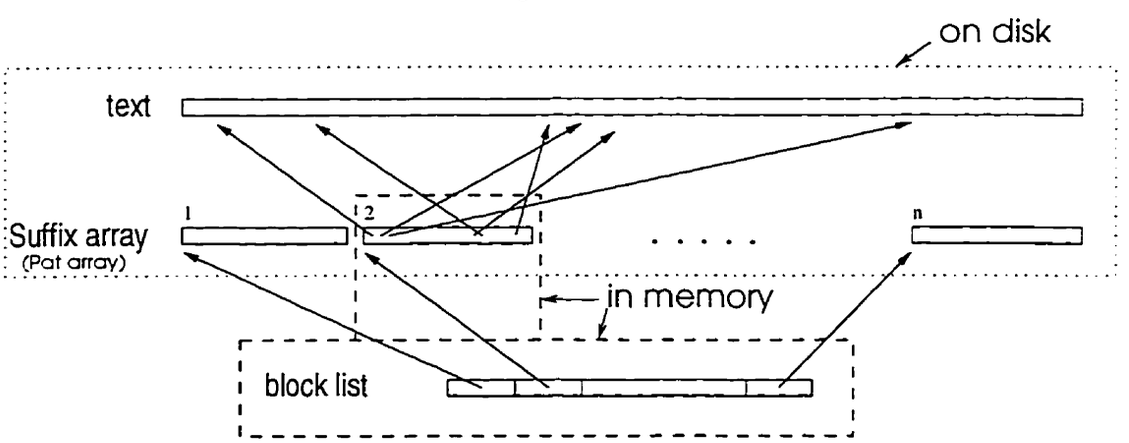


Figure 1.3: Hierarchy of Indices

Since search time is dominated by disk operation time, the number of disk accesses is critical. For a suffix array, search time is $\Theta(\log n)$ disk accesses, where

n is the number of entries in one block.

1.2 Proposed suffix-signature method

We propose a hash-based method to reduce the number of disk accesses to a text. The idea is to give a short "signature" to each phrase in a suffix array. When we search for a phrase, we first compute its signature, and then see if its signature matches any one in the signature file.

Optimistically, through the availability of a unique signature, a phrase search needs one disk access: after reading an appropriate piece of the signature file, perhaps located with the help of a block list, the phrase can be matched without access to the text itself. Thus, ideally, we want every phrase to have a short unique signature. The most trivial signature is, of course, the phrase itself, but these are too long.

If we simply give each phrase very "short" signatures, then we might have several different phrases with the same signature. If we design these signatures carefully, they will work most of the time and fail in some small number of cases. In these latter cases, we do a few more disk accesses, but on average, we still win.

This thesis aims at developing a method to search for a phrase with an expected time of one disk access to the index and no more than two disk accesses to the text, or two disk accesses to indices and no more than one disk access to the text. The space of the phrase signatures is about 20 bits per 5-word phrase after a simple compression.

In this section, we describe the outline of the suffix-signature method we are proposing. We briefly show each part of the method and how the method works in

general. In the following chapters, we study each part in detail.

1.2.1 Signature array and index hierarchy

Each phrase has an integer as its signature. If each phrase has a unique signature, only internal integer comparisons are needed to find out if a phrase exists in a block. But it is infeasible to have uniqueness, because words must be known in advance in order to find perfect signature functions, and adding one more word will probably make it necessary to find new functions.

Signature methods therefore give up the idea of uniqueness, permitting a small number of different words to have the same signature and using a special method to resolve collisions. Thus signature methods use hashing techniques as functions to generate signatures.

In the proposed method, there is a *signature array* in addition to the *text*, *suffix array*, and *block list*, as illustrated in Figure 1.4. The text and the suffix array are the same as in Figure 1.2. Elements in the signature array are integers generated by a hash function, and they are signatures of corresponding sistrings in the suffix array. Unlike a signature file as described earlier [Fal92], the signature array has one signature for each phrase in the suffix array. Signature functions are used at index building time to produce an integer as a signature for each phrase. In Figure 1.4, there are 8 different signature values. Some different sistrings have the same signature. For instance, the 9th and the 11th sistrings have S_8 as their signatures. In this example, the suffix array and the signature array are divided into two blocks. The first block has 5 sistrings and the second has 6 sistrings. The block list indicates where each block starts in the suffix array and the parallel signature array.

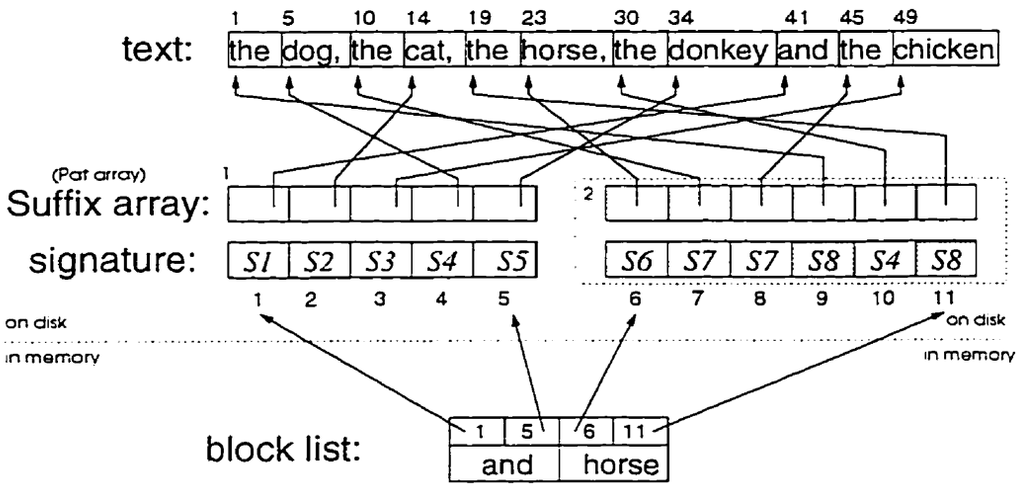


Figure 1.4: Suffix-Signature Method

The text, suffix array and signature array are on disk, whereas the block list is in memory. When we search for a particular phrase, we read a signature block into memory according to the start of that target phrase and the block list. Then we search the signature block for the same signatures as the target phrase, and read corresponding phrases from disk to determine whether they match the target phrase.

Assume that we search for a phrase p="the horse, the donkey and the chicken", which we will find starting at the 19th character in the text of Figure 1.4; this is the 11th entry in the suffix array and in block 2; it has $S8$ as its signature. Notice that in the same block, the phrase p'="the dog, the cat, the horse, the donkey and the chicken" has $S8$ as its signature too. By looking up the block list, we know that the phrase p is indexed in block 2, if it is in the text at all. We read the corresponding blocks of the suffix array and the signature array into memory from disk. We search the signature array block from the beginning for $S8$. We find p' after 1 disk access to the text, and then find p at the 2nd disk access to the text.

We continue to introduce the phrase signature structure in Section 1.2.2, and how to bound the number of disk accesses for searching for a phrase in Sections 1.2.3 and 1.2.4.

1.2.2 Signature structure

In our example, the 9th semi-infinite string in the suffix array, starting at the first character, has signature $S8$. Its prefixes, for instance "the dog", in general have different signatures. In order to search for prefixes of semi-infinite strings using the signature array, phrase signatures need the prefix property, that is, signatures of a prefix phrase are computable from the phrase signature.

The concatenation scheme, which satisfies the prefix property, computes the signature for each word in a phrase and concatenates all the word signatures to get the phrase signature.

A complete phrase signature array contains $\Theta(n * r)$ word signatures, where n is the number of words in a block and r is the length of the longest repeating phrase. To effectively use space, we let a phrase signature contain signatures for only the first several words, for instance, the first four words in our example of Figure 1.5.

Naively, each word signature may be of the same size, for example, 16 bits in Figure 1.5. To use space more effectively, we could use different signature sizes for different words in a phrase, and different phrases may use different word signature sizes depending on word distributions.

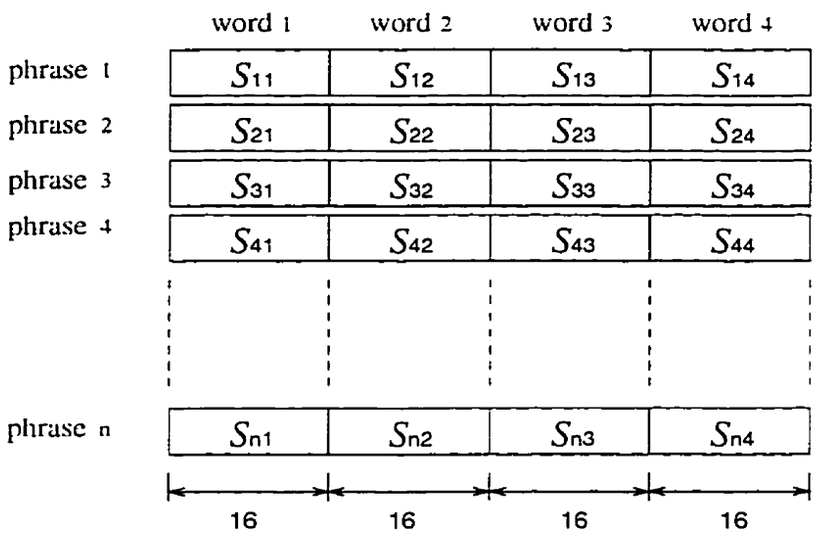


Figure 1.5: Signature Structure

1.2.3 Separate collisions

Assume that we search for the shaded phrase of Figure 1.6 which has a signature s. There are many phrases in the signature block having signature s. We could start searching for signature s from the beginning of the signature block. For a matching signature s, we need a disk access to read the corresponding phrase of the text into memory to verify if it is the target phrase. 8 such disk accesses are needed to get the shaded phrase in our example of Figure 1.6.

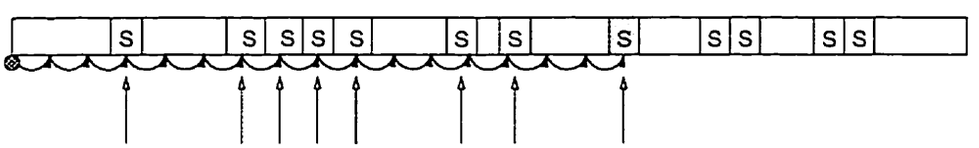


Figure 1.6: Search for Matching Signatures

To reduce the number of disk accesses for searching for a phrase, we start from the middle point of a range to look for matching signatures. Then, having found

one matching signature and having examined the corresponding text we could determine whether to look further to the right or left in the block. Therefore, only a logarithmic number of disk accesses is needed at worst to find the target phrase.

In Figure 1.7, we search for the shaded phrase. Using binary search to choose starting points for each scan, we start from the dot numbered 1 to search for matching signatures to the left and to the right at the same time. Using binary subdivision, dot 2 and dot 3 are the starting positions in the 2nd and the 3rd scans respectively. So, the 3rd disk access to text reaches the target phrase.

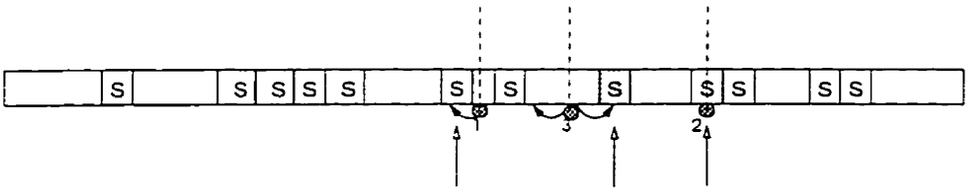


Figure 1.7: Start from the middle

1.2.4 Adjacent collisions

Phrases starting with the same first two words have the same 2-word prefix signature. All such phrases are adjacent in the suffix-array since it is ordered lexicographically, and thus also in the signature array. Therefore if we are searching for a 2-word phrase, we need to read only one of these phrases from disk to verify that all, or none, of the phrases match the target phrase. This reduces the number of disk accesses, but introduces the problem of separating *adjacent collisions*.

If two *different* 2-word phrases, for instance, "the cat" and "the chicken" in our example of Figure 1.4, are alphabetically adjacent and have the same signature, reading only one of the two adjacent phrases and using the above approach may miss the correct phrase.

To solve the adjacent collision problem, adjacent collision boundaries are stored in a table, called a look-aside table, exemplified in Figure 1.8. In every subrange of the signature array delimited by entries in the table, there are no adjacent collisions. Therefore we can safely read only one phrase of adjacent phrases sharing the same signature from disk to test for a match.

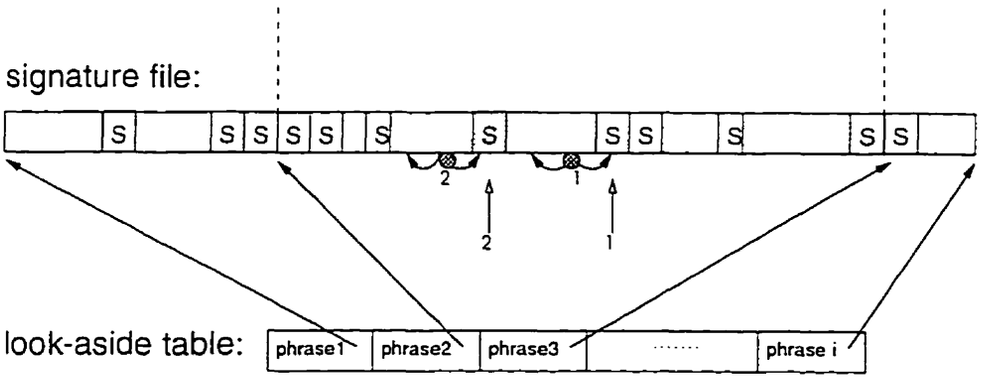


Figure 1.8: Look-aside Table

If phrases corresponding to such boundary points are stored in the look-aside table, searching on the signature array is done only on a smaller section, and no disk access to text is needed for phrases that appear in the table directly. The average number of disk accesses to search for a phrase thus is expected to become smaller.

The look-aside table may contain some other phrases as well. For instance, phrases that could not be found within the first two disk accesses may also be stored in the table. This guarantees that all the searches are done by at most two disk accesses to text; such stored phrases are called "guaranteeing phrases". Properties of the look-aside table and resulting algorithms form a major contribution of this thesis.

1.2.5 Simplified phrase search algorithm

Given a text and its suffix array, we divide the suffix array into blocks and create a block list. Assume that for each block there are a signature block based on the first 5 words of suffix phrases and a look-aside table. The look-aside table contains all the adjacent collision boundary phrases and some other sufficient phrases to guarantee a worst case of 2 disk accesses to the text for searching for a phrase up to 5 words. They are all on disk. Before searching starts, the block list is loaded into main memory.

Problem: find all the occurrences in a text of a phrase p of length up to 5 words.

Input: a phrase p;

block list: strings;

suffix array: integers;

signature array: integers;

look-aside table: (string, integer) pairs;

Output: the number of matches and the range in the suffix array pointing to matching places in the text.

Algorithm 1 (Phrase Search - a simplified version of Algorithm 3 from Chapter 4)

1. *Check the block list, find the block in which phrase p falls.*
2. *Read the corresponding block of the suffix array and the signature array into memory.*

3. *Check the adjacent collision boundary phrases in the look-aside table, and find the adjacent collision interval* $[l_1, l_2]$ *containing* p.

4. *Create the signature* sig_p *for phrase* p. *Then*

   (a) *Check the guaranteeing phrases in the look-aside table. If* p *matches a guaranteeing phrase* p', *pointers in the range around* p' *in* $[l_1, l_2]$ *that matches* sig_p *point to all occurrences of phrase* p *in the text. Otherwise, go to Step 4b.*

   (b) *Starting at position* $\frac{l_1+l_2}{2}$, *search up and down the signature array block alternatively to find a match for* sig_p. *If no match is found, then phrase* p *does not occur in the text. Otherwise, determine the first encountered interval* $[t_1, t_2]$ *that matches* sig_p.

   (c) *Pick any position* i *in* $[t_1, t_2]$. *Get the* i*th pointer in the suffix array block and read a phrase* p' *from text on disk. Compare phrases* p *and* p'.

   - *If they are identical, then pointers in the range* $[t_1, t_2]$ *of the suffix array point to all occurrences of phrase* p *in text.*
   - *If they are not identical, re-iterate from Step 4b using* $[l_1, t_1 - 1]$ *if* $p < p'$ *or* $[t_2 + 1, l_2]$ *if* $p > p'$.

Step 2 in Algorithm 1 costs one disk access per phrase search. Step 4c costs one disk access each time it gets executed, and because of storing guaranteeing phrases in the look-aside table, Step 4c will be executed at most twice per phrase search. Therefore, the number of disk accesses to text is at most 2.

1.3 Overview of the thesis

In this thesis, we study a suffix-signature method for searching for phrases in large static texts. We describe how the suffix-signature method works, investigate how to augment a suffix array with a parallel signature array and how to guarantee the worst case of 2 disk accesses to the text for searching for a phrase of bounded length. We also investigate how to create a look-aside table and how to control its size. We present theoretical analyses and report the results of experiments.

We will use the following assumptions in this thesis:

- *Data:* A text is a very large, static sequence of characters stored on a slow-access medium.
- *Queries:* Queries applied to the large text data are primarily substring queries, locating strings that start with a given phrase.
- *Fast and on-line access:* Queries to data are on-line and therefore responses are required to be very fast.
- *Storage space:* Only a small fraction of the text can be stored in main memory at one time, and auxiliary structures on secondary store are restricted in size to a small multiple of the size of the text.
- *Disk accesses:* The disk access time is the elapsed time from a reading request received by a disk driver to all the data having been transfered to memory. A disk access consists of two steps: a locating phase and a transferring phase. During a locating phase, the disk read-write head moves to the position where the transfered data starts. It includes the disk read-write head moving from the current track to the next required track and the required section rotating

to under the disk read-write head. Locating time depends on seek time and latency time. Transferring time depends on the transfer rate and the amount of data transfered. We limit the amount of data transfered in one disk access to be at most the amount of data that can be transfered in the average time needed to locate a randomly chosen point on the disk. This limits the block size.

In Section 1.2, we briefly described how the proposed method works in general. In the following chapters, we study each part of the method in detail.

In Chapter 2, we examine properties of adjacent collisions and review some related results on hashing. We study word signature functions, which take a word as input and generate an integer as the signature for that word. We present and analyze optimal word signature functions and discuss principles of word signature functions. Then we analyze the tradeoff between word signature size and the number of disk accesses required to search for a word.

In Chapter 3, we investigate phrase signature functions, which take a phrase as input and generate an integer as the signature for that phrase. We describe the prefix property of phrase signatures and the structure of phrase signatures. We study a concatenation scheme for phrase signatures, and contrast it with a more traditional superimposition scheme. We also discuss possibilities of using perfect hashing and compression techniques for phrase signatures.

In Chapter 4, we study searching with signatures. We define and show how to solve adjacent collisions and how to manage separate collisions (as described in Section 1.2). We describe a search algorithm on word signature arrays, and extend it to phrase signature arrays. Then we study how to balance the space requirements imposed by the look-aside table. We also discuss shortcomings of our

suffix-signature method and how to tackle these problems.

Throughout Chapters 3 and 4, we illustrate the properties of various schemes with small experiments. In Chapter 5, we present the implementation of our prototype system and report on more extensive experiments using the proposed suffix-signature method. We describe both the experimental model and experimental results.

In Chapter 6, we discuss some implementation issues. We discuss the creation of indexes, memory requirements and block lists.

In Chapter 7, we review the properties of the suffix-signature method, compare it with other methods, and discuss future related work.

In Appendix A, we describe the corpus used in our experiments, and in Appendix B, we give more detailed experimental results of the proposed suffix-signature method.

The main contribution of the thesis is a new phrase search method for large static texts. For a phrase of bounded length, it guarantees 2 disk accesses to a text in the worst case for both successful and unsuccessful searches. On average it requires one disk access to the text and one disk access to the index. Experimentally we show the method works well for real world data, using space that is 110% to 130% of the original text.

Chapter 2

Word Signatures

Since adjacent collisions influence search algorithms presented in the thesis, we examine properties of adjacent collisions in this chapter. We also review some related results on hashing.

We first define the data model used in this chapter. Next, we discuss issues related to designing good signature functions, and we describe some specific signature functions. Finally, we investigate some theoretical limitations for random signature functions and investigate a tradeoff between the signature size and the number of disk accesses to search for a word.

2.1 Data model

In this section, we define some terms and a simple word search model based on word signature matching. Our studies in this chapter are based on this model.

Definition 1 (Word) *A word is a sequence of characters.*

Definition 2 (Signature and signature function) *A signature is a sequence of bits. A signature function Sig takes a word w as input and generates an integer $Sig(w)$ as the signature for word w.*

Model 1 (Simple word search data model) *We are given an ordered set of words $W = \{w_i \mid w_i < w_j, 1 \leq i < j \leq n\}$, based on lexicographic ordering of the words, and a signature function Sig. For each word w_i in W, there is a corresponding value $Sig(w_i)$. As a result, an indexed set Q is associated with W, where $Q = \{< i, Sig(w_i) > \mid 1 \leq i \leq n\}$.*

In this simple data model, words are distinct. Since the number and the relative positions of adjacent collisions are not affected by repeating words, we will use this simplified data model in this chapter. We will define and use a more complex data model in later chapters of the thesis.

Definition 3 (Collision) *If two words have the same signature, we say that they collide with each other.*

It is because W is ordered that there are two kinds of collisions.

Definition 4 (Adjacent collision and separate collision) *Given an ordered set of words W, let w_i and w_j be words in W with $Sig(w_i) = Sig(w_j)$. If there is another word w_k in W between w_i and w_j, we say that w_i and w_j collide* separately*: if there is no such word w_k in W, w_i and w_j are said to collide* adjacently.

As presented in Section 1.2 the two kinds of collisions are handled differently.

2.2 Signature functions

Much research has been done regarding hash functions [Knu73, GBY91], which can be applied to words by interpreting the words as natural numbers. A simple way to interpret a word as a natural number is to consider a word as a number in a suitable radix notation. A word $w = s_0 s_1 \ldots s_t$ represented as ASCII characters, for instance, can be expressed as

$$s_0 * B^t + s_1 * B^{t-1} + \ldots + s_t * B^0.$$

And the following hash function could be used

$$h(w) = ((\sum_{i=0}^{t} s_i B^{t-i}) \bmod m) \bmod M,$$

where m is the maximum value in a computer word, the operation mod m is done by the hardware, and l with $M = 2^l$ is the number of bits in a word signature. For the function $h(w)$, the value $B = 131$ is recommended, as B^i has a maximum cycle mod 2^k for $8 \leq k \leq 64$ [GBY91].

Since the mod M operation has to be done to fit a word signature space and it is believed that $\sum s_i p_i$ might behave more randomly when p_i's are prime numbers, it might be a good idea to replace B^i in $h(w)$ by a prime number p_i. Also, the exclusive-or has the advantageous feature that if two strings have 0 or 1 bits occurring equally-probably and independently in each position, then the resulting bit string also has this property [Kno75]. Thus, the signature function $h'(w)$ that we used in our experiments for the suffix-signature method is the result of exclusive-*OR*ing various segments of

$$h'(w) = (\sum_{i=0}^{t} s_i p_i) \bmod m,$$

where p_i is the ith prime number in a vector of prime numbers, m is the maximum value in a computer word, and the size of a segment is the size of a word signature. Again since it is believed that $\sum s_i p_i$ might behave more randomly when p_i's are prime numbers, p_i's should be less than the size of a word signature, otherwise s_i's are essentially multiplied by non-prime numbers. Furthermore, if the word hashing space is very big, p_i's should be randomly spread in the hashing space to avoid clustering in some parts of the hashing space, and there should not be any relations among the p_i's, such as one being about twice as big as the previous one.

2.3 Analytical results

Signature functions use hashing techniques to map words (*i.e.* sequences of characters) into word signatures (*i.e.* sequences of binary bits). Following the definition from hashing we define the load factor to be the number of words divided by the size of the signature space. Collisions cause false hits, which need to be resolved by using more disk accesses. Given a particular signature, the fewer words mapped to this value by a signature function, the better the function behaves for retrieval. Since words are ordered and collisions are distinguished between adjacent collisions and separate collisions, signature functions have some characteristics that general hashing functions do not have. Since hashing techniques have been studied extensively by others [Knu73, GBY91], we will only concentrate on properties that are related to adjacent collisions and have not been studied before.

For a uniform query distribution, the more uniform a signature function, the better performance. Since word functions are chosen prior to knowing the words to be coded, the best expected performance is the one from truly random hash functions. In this section, we study, for random signature functions, the distribution

of the number of adjacent collisions and the distribution of the number of words having the same signature.

The reason that we study adjacent collisions is that the number of adjacent collisions influences the number of disk accesses used to retrieve words and the space used by the suffix-signature method, as discussed in detail later.

Lemma 1 (Distribution of the number of adjacent collisions) *If the hashing space has size m, the number of adjacent collisions i in a block of length n has a binomial distribution* $b(i; n-1, \frac{1}{m}) = \binom{n-1}{i}\frac{(m-1)^{n-i-1}}{m^{n-1}}$

Proof: Let us look at words in W, which are in lexicographical order. Signatures are repeated independent trials with only two possible outcomes for each signature, one is that a word collides with the word to its left and another possible outcome is that it does not collide. For any word, the probability to be hashed to the same value as the word to its left is $\frac{1}{m}$, and the probability to hash to a different value is $(\frac{m-1}{m})$. These probabilities remain the same throughout the block except for the first word. Thus, word signatures are Bernoulli trials [Fel50]. Therefore, the probability that i words collide with the words on their left is $b(i; n-1, \frac{1}{m}) = \binom{n-1}{i}\frac{(m-1)^{n-i-1}}{m^{n-1}}$ [Fel50].□

Lemma 2 (Expected number of adjacent collisions) *Assume that there are n keys to be hashed randomly in the range $[0, m-1]$. The expected number of adjacent collisions is $\frac{n-1}{m}$.*

Proof: The number of adjacent collisions has a binomial distribution $b(i; n-1, \frac{1}{m})$, as shown in Lemma 1. The expectation of the binomial distribution $b(i; n-1, \frac{1}{m})$ is $E(b(i; n-1, \frac{1}{m})) = \frac{n-1}{m}$ [Fel50]. Therefore, the expected number of adjacent collisions is $\frac{n-1}{m}$. □

Lemma 3 (Distribution of the number of collisions for a given signature) *Assume that there are n keys to be hashed randomly in the range $[0, m-1]$. The number of words that are mapped to a given signature has a binomial distribution $p_i(n) = b(i; n, \frac{1}{m}) = \binom{n}{i}\frac{(m-1)^{n-i}}{m^n}$.*

Proof: For a given signature value, the probability of a key to be hashed to it is $\frac{1}{m}$, and the probability that it is hashed to some other value is $(\frac{m-1}{m})$. All the keys are hashed independently with only two possible outcomes, one is that it is hashed to the given key and another is that it is not hashed to the given key. Again, these are Bernoulli trials. Therefore, the probability that i words collide with a given signature follows the Bernoulli distribution $p_i(n) = b(i; n, \frac{1}{m}) = \binom{n}{i}\frac{(m-1)^{n-i}}{m^n}$ [Fel50].□

When the load factor is 1 (*i.e.* $m = n$), the probability that i keys are hashed to a given signature is $p = \binom{m}{i}\frac{(m-1)^{m-i}}{m^m}$. For small i and large m, $p \approx \frac{1}{i!}\frac{(m-1)^{m-i}}{m^{m-i}} \approx \frac{1}{i!}(1-\frac{1}{m})^m$. Since $(1-\frac{1}{m})^m \approx \frac{1}{e}$ for large m [GBY91], we have $p \approx \frac{1}{e(i!)}$ for small i, large m, and $m = n$. This approximation agrees with Lemma 3 to three significant digits for $m = n = 2^{13}$ and $i \leq 6$.

Lemma 4 (Expected fraction of hash values associated with i words) *Assume that there are n keys to be hashed randomly in the range $[0, m-1]$. The expected fraction of hash values (signatures) with i keys per value is the same as the probability of any given hash value corresponding to i keys.*

Proof: The problem can be converted to a problem in which n balls are randomly thrown into m boxes. It is needed to prove that the expected fraction of boxes with i balls is the same as the probability of a box having i balls. Thus, for a box j,

let p_i^j be the probability of box j having i balls, and p_i be the expected fraction of boxes with i balls. We want to prove that $p_i^j = p_i$.

T is the total number of ways of placing n balls in m boxes. $B(i,j)$ is the number of ways in which box j has i balls. By definition, we have $p_i^j = \frac{B(i,j)}{T}$.

Each way throws balls in m boxes. T ways throw balls in mT boxes, out of which $\sum_{j=1}^{m} B(i,j)$ boxes have i balls. So, we have $p_i = \frac{\sum_{j=1}^{m} B(i,j)}{mT}$.

Since $B(i,j)$'s are equal for all $j = 1, 2, \ldots, m$, we have $p_i = \frac{mB(i,j)}{mT} = \frac{B(i,j)}{T} = p_i^j$. Therefore, we have that the expected fraction of hash values that have i keys is the same as the probability of any specific hash value having i keys. □

To illustrate the above lemma, let us look at all the ways 3 balls can be put into 3 boxes.

Table 2.1 is the sample space of 3 boxes and 3 balls. Table 2.2 lists values of $B(i,j)$, p_i^j, and p_i for the example.

Two keys producing identical signatures parallels two keys hashing to the same bucket. Thus studying the distribution of collisions is the same as studying the length of chains in external hashing [GBY91].

Lemma 5 (Expected fraction of words colliding with $(i-1)$ other words, or percentage of words in chains of length i) *Assume that there are n keys to be hashed randomly in the range $[0, m-1]$. The expected percentage of words in one of the chains of length i is $\frac{im}{n} b(i; n, \frac{1}{m})$.*

Proof: The expected fraction of hash values each of which has i keys is the same as the probability of a hash value being produced by i keys (lemma 4). The probability of a hash value having i keys is $b(i; n, \frac{1}{m})$ (lemma 3). So, $b(i; n, \frac{1}{m})$ is the expected

box 1	box 2	box 3
3	0	0
2	1	0
2	0	1
1	2	0
1	1	1
1	0	2
0	3	0
0	2	1
0	1	2
0	0	3

Table 2.1: Sample space of 3 boxes and 3 balls

fraction of hash values each of which has i keys. Then, $mb(i; n, \frac{1}{m})$ is the expected number of chains with length i. Then, $imb(i; n, \frac{1}{m})$ is the expected number of words in all the chains with length i. Therefore, $\frac{im}{n}b(i; n, \frac{1}{m})$ is the expected percentage of words in all the chains with length i. □

Note that for small i, big m and $m = n$, the expected percentage of words in all the chains with length i is $\frac{1}{e(i-1)!}$. So most words will appear on very short chains.

Lemma 6 *For $m = n > 1$, the expected lengths of non-empty chains is smaller than $2 + \frac{3}{e(m-1)}$.*

Proof: We use the formulas $(1 - \frac{1}{m})^m < \frac{1}{e}$[GBY91] and $\sum_{i=1}^{\infty} \frac{i}{(i-1)!}\frac{1}{e} = 2$.[1]

[1]Since $\sum_{i=0}^{\infty} \frac{i+1}{i!} - \sum_{i=0}^{\infty} \frac{i}{i!} = \sum_{i=0}^{\infty} \frac{1}{i!} = e$ and $\sum_{i=0}^{\infty} \frac{i}{i!} = \sum_{i=1}^{\infty} \frac{1}{(i-1)!} = \sum_{i=0}^{\infty} \frac{1}{i!} = e$, we have $\sum_{i=1}^{\infty} \frac{i}{(i-1)!}\frac{1}{e} = \frac{1}{e}\sum_{i=0}^{\infty} \frac{i+1}{i!} = 2$.

$B(3,j)=1$	$p_3^j = 1/10$	$p_3 = 3/30$
$B(2,j)=2$	$p_2^j = 2/10$	$p_2 = 6/30$
$B(1,j)=3$	$p_1^j = 3/10$	$p_1 = 9/30$
$B(0,j)=4$	$p_0^j = 4/10$	$p_0 = 12/30$

Table 2.2: Values of $B(i,j)$, p_i^j, and p_i

Let the expected lengths of non-empty chains be l. From Lemma 5, we have $l = \sum_{i=1}^{m} \frac{i^2 m}{n} b(i; n, \frac{1}{m})$. So for $m = n$,

$$l = \sum_{i=1}^{m} i^2 \binom{m}{i} \frac{(m-1)^{m-i}}{m^m} = \sum_{i=1}^{m} \frac{i}{(i-1)!} \frac{m!}{(m-i)!} \frac{(m-1)^{m-i}}{m^m}.$$

Let x_i be $\frac{i}{(i-1)!} \frac{m!(m-1)^{m-i}}{(m-i)!m^m} = \frac{i}{(i-1)!} \frac{m!}{(m-i)!(m-1)^i} \frac{(m-1)^m}{m^m} < \frac{i}{(i-1)!} \frac{m(m-1)(m-2)\ldots(m-i+1)}{(m-1)^i} \frac{1}{e}$. Thus we have $x_1 < \frac{m}{m-1}\frac{1}{e}$ and $x_2 < \frac{m}{m-1}\frac{2}{e}$. Since $\frac{m(m-1)(m-2)\ldots(m-i+1)}{(m-1)^i} < 1$ for $i > 2$, we have $x_i < \frac{i}{(i-1)!}\frac{1}{e}$ for $i > 2$.

Thus we have

$$\begin{aligned} l &< \frac{m}{m-1}\frac{3}{e} + \sum_{i=3}^{m} \frac{i}{(i-1)!}\frac{1}{e} \\ &< \frac{m}{m-1}\frac{3}{e} + \sum_{i=3}^{\infty} \frac{i}{(i-1)!}\frac{1}{e} \\ &= \frac{m}{m-1}\frac{3}{e} + 2 - \frac{1}{e} - \frac{2}{e} \\ &= 2 + \frac{3}{e(m-1)}. \end{aligned}$$

□

The upper bound of l goes to 2 as m increases. When $m = 1$, 2, 3 and 4, l is 1, 1.5, 1.67 and 1.75 respectively. Figure 2.1 illustrates expected lengths of non-empty chains when the load factor is 1. The numbers on the x axis represent numbers of

bits in each signature, and numbers on the y axis are expected counts of words in chains. We notice that the expected lengths converge to 2 as m goes to infinity. Based on this, in later chapters we choose to guarantee at most 2 disk accesses to a text when searching for a phrase.

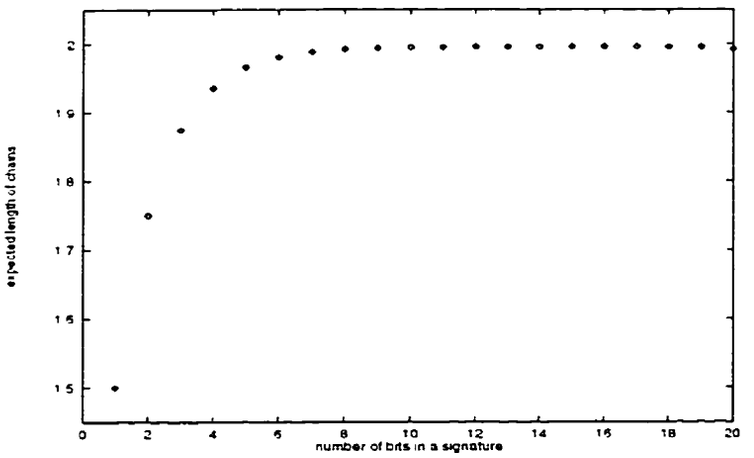


Figure 2.1: Expected length of non-empty chains when m=n

Corollary 1 *Assume that W includes n words to be hashed randomly in the range $[0, m-1]$. The expected number of matches to a given signature is $\frac{n}{m}$. For a given word w in W, the expected number of matches to the signature of w is $\frac{n-1}{m}+1$.*

Proof: Let i be the number of words in W that have a given signature s as their signatures. Then, the expected number of matches is $\sum_{i=0}^{n} i * p_i(n) = \sum_{i=0}^{b} i * b(i; n, \frac{1}{m}) = \frac{n}{m}$.

Let s be the signature of a given word w in W. The number of other words having s as their signatures has a binomial distribution $p'_i(n) = b(i; n-1, \frac{1}{m}) = \binom{n-1}{i}\frac{(m-1)^{n-1-i}}{m^{n-1}}$, for $0 \leq i \leq n-1$. There are $(i+1)$ words having s as their signatures if there are i other words that have s as their signatures. So, the expected number

of matches is $\sum_{i=0}^{n-1}(i+1)*p_i'(n) = \sum_{i=0}^{n-1} i*p_i'(n) + \sum_{i=0}^{n-1} p_i'(n)$. Since $\sum_{i=0}^{n-1} i*p_i'(n)$ is the expectation of the binomial distribution $p_i'(n) = b(i; n-1, \frac{1}{m})$. We have $\sum_{i=0}^{n-1} i*p_i'(n) = \frac{n-1}{m}$ [Fel50]. Since $\sum_{i=0}^{n-1} p_i'(n) = 1$. We have the expected number of matches is equal to $\frac{n-1}{m} + 1$. □

In the suffix-signature method, signature comparisons are integer comparisons in memory, whereas comparing the target word against a word in the text having the same signature requires a random disk access to the text. So, the number of string (word) comparisons is indicative of the number of disk accesses.

Intuitively, the more bits in word signatures, the fewer collisions result and hence the fewer string comparisons are needed. There is a tradeoff between the length of signatures and the number of string comparisons.

Lemma 7 (Space-Time trade-offs) *Assume signature matching and binary search probing are used to search for a word in an ordered word set W. If $|W| = n$ and $(\log n - i)$ bits are used for each word signature, the expected number of string comparisons to search for a word chosen from W with all words in W equally likely to be chosen is $\Theta(i)$.*

Proof: Each probe of a binary search cuts a range to half. So, the first probe on a signature file is over a range of size n, the second is over a range of size $n/2$, and the ith is over a range of $n/2^{i-1}$. If the desired word has not been found after i probes, we are left with a range of size $n/2^i$ in which the desired word falls.

Since each word signature has $(\log n - i)$ bits, there are $2^{\log n - i} = n/2^i$ distinct word signature values. Thus, since this is the same size as the range after i probes, each signature in W is expected to appear 1.5 to 2 times, as illustrated in Figure 2.1. That is, for each signature in W, the expected number of string com-

parisons on the resulting range converges to 2. Therefore, the expected number of string comparisons to search for a word is about i. □

Lemma 7 describes the relationship between the number of bits in word signatures and the number of string comparisons for our suffix-signature method using binary probing. The following corollaries give two extreme cases. One is a lower bound on the number of bits needed to get $\Theta(1)$ string comparisons and another is the case when the size of word signatures is 0.

Corollary 2 *If $|W| = n$ and word signatures have $\log n$ bits, the expected number of string comparisons to search for a word is $\Theta(1)$.*

Corollary 3 *If we do not use signatures to filter probes, that is, a word signature has 0 bits, the expected number of string comparisons to search for a word is $\Theta(\log n)$. This is a direct binary search on an ordered list of words without using the suffix-signature method.*

2.4 Summary

The contributions of this chapter are studies of the properties of adjacent collisions. We also reviewed some related results on hashing. We discussed issues related to practical word signature functions. We analyzed behaviors of random signature functions for the suffix-signature method, in particular, behaviors of adjacent collisions. We studied the relationship between the signature size and the number of disk accesses to search for a word.

Interestingly, when the number of words in W is approximately the size of the signature space, most words share signatures with at most one other word.

As the number of words in W increases, more words share signatures: but then the probability of adjacent collisions increases as well. Since adjacent collisions are stored in a look-aside table, W is effectively partitioned into subranges, thus reducing the load factor for each subrange. As will be seen in Chapters 3 and 4, this forms the basis of the effectiveness of the suffix-signature approach.

Chapter 3

Extension To Phrase Signatures

In Chapter 2, we explored signature functions for words. We will study phrase signatures in this chapter.

We first discuss desirable properties of phrase signature. Then we study the concatenation signature scheme. We investigate various phrase signature structures and discuss some parameters of the concatenation scheme. We next compare its performance to an alternative phrase signature scheme based on superimposition. We also study the possibilities of using other techniques, namely perfect hashing and compression, in the suffix-signature method.

3.1 Notations

We first define the notation that is used in this chapter:

- w_i is a word.
- $P = w_1 w_2 \ldots w_n$ is a phrase.

- $Sig(P)$ is the signature of phrase P.
- $P = w_1 w_2 \ldots w_n$ is an indexed phrase in a text if P is a prefix of a semi-infinite string listed in the suffix array.

Consider a set of phrases $P_1, \ldots, P_m$. They can be partitioned according to the first word of each phrase, w_1 through w_n. All phrases beginning with w_1 can again be partitioned by their second words, $w_{11}, w_{12}, \ldots, w_{1k}$. Repeating this partitioning, the set of phrases can be considered to form levels much like a word-based trie [Fre60], as illustrated in Figure 3.1. For the one-word prefix w_2, there are the phrases 'w_2' at the first level, '$w_2\ w_{21}$' at the second, and the two three-word phrases '$w_2\ w_{21}\ w_{211}$' and '$w_2\ w_{21}\ w_{212}$' at the third.

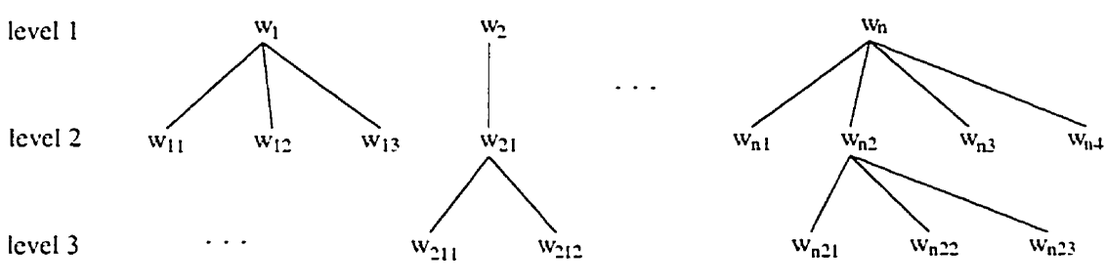


Figure 3.1: Levels of phrases

The number of nodes on level i is equal to the total number of distinct i-word phrases in the database, and the branching factor between level $(i-1)$ and level i represents the number of i-word phrases that share a common $(i-1)$-word prefix.

3.2 Prefix property

If a phrase $P = w_1 w_2 \ldots w_n$ is indexed, its signature can be stored in the signature array. However, we also wish to search for phrases that are the prefixes of P. For

example. we may want to find all phrases starting with $P_2 = w_1 w_2$. a prefix of the phrase P. The suffix array accommodates such prefixes naturally [GBYS92]. However we cannot afford to store independent signatures for all the prefixes of P because they would take too much space.

To find prefixes of such a phrase P using the signature file of a text without storing all the prefix signatures. we need the signature of P to have a *prefix property*:

Definition 5 (Prefix Property) *The signatures of prefixes of a phrase P ($P = w_1 w_2 \ldots w_n$) can be extracted algorithmically from the signature of P. That is, there exist a family of functions $\{f_i \mid 1 \leq i \leq n\}$, such that, $Sig(P_i) = f_i(Sig(P))$ for $P_i = w_1 w_2 \ldots w_i$, where $1 \leq i \leq n$.*

Sometimes. a weaker prefix property is useful. It does not require that prefix signatures of a phrase can be extracted from the phrase signature. Instead. it requires that a function be defined that returns *true* if a given prefix matches the target signature. Just as the signature of a prefix can also be the signature of a phrase that is not a prefix, this function may also return true for some phrases that are not prefixes.

Definition 6 (Weak Prefix Property) *There exist a test and a real value ϵ ($0 < \epsilon < 1$) such that, given an input phrase P_i and a phrase signature Sig, the test returns 'true' if P_i is a prefix of some phrase P having Sig as its phrase signature. and returns 'false' with probability at least ϵ when it is called with randomly chosen signatures as parameters. That is, there exist a family of functions $\{f_i | 1 \leq i \leq n\}$ such that $f_i(Sig(P), Sig(P_i)) = True$ for $P_i = w_1 w_2 \ldots w_i$ where $1 \leq i \leq n$, and $f_i = False$ for some other input phrases.*

A stronger prefix property is as follows.

Definition 7 (Strong Prefix Property) *The signatures of prefixes of a phrase P are prefixes of the signature of P.*

Clearly, any signature scheme satisfying the prefix property satisfies the weak prefix property, and any one satisfying the strong prefix property satisfies the prefix property.

3.3 Concatenation scheme

We present the concatenation scheme for phrase signatures in this section. We investigate phrase signature structures and study adjacent collisions. We discuss the expected number of adjacent collisions in a block, the condition of minimizing the total number of adjacent collisions, and how likely it is that a boundary between two distinct phrases has adjacent collisions from various levels of phrases. Using these results, we discuss how to partition bits among words for the concatenation scheme, and discuss related parameters, for instance, the total number of bits in a phrase signature, the number of words to be encoded in a phrase signature, and the block size to be used as a unit of I/O.

3.3.1 Structure of phrase signature

The idea of the concatenation scheme is that we concatenate all the word signatures of a phrase to form a phrase signature

$$Sig(P) = Sig_1(w_1)Sig_2(w_2)\ldots Sig_k(w_k).$$

A signature array of n phrases is shown in Figure 3.2, in which S_{ij} denotes $Sig_j(w_{ij})$ for phrase i.

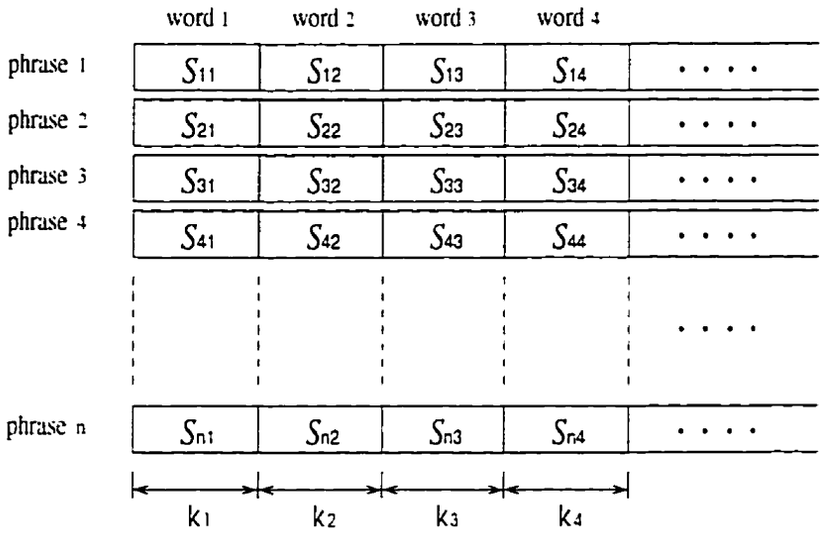


Figure 3.2: Signature Structure

Word signatures are basic units on which phrase and prefix signatures are built. Assume that there are k_i bits for $Sig_i(w_i)$ $(1 \leq i)$, and that $P_i = w_1 w_2 \ldots w_i$, $(1 \leq i)$.

- Prefix signature functions are:

$$Sig(P_i) = \sum_{d=1}^{i} Sig_d(w_d) * 2^{\sum_{t=d+1}^{i} k_t} = \sum_{d=1}^{i} (Sig_d(w_d) << \sum_{t=d+1}^{i} k_t)$$

  for $(1 \leq i)$.

- The functions converting a phrase signature into its word signatures are:

$$Sig_i(w_i) = (Sig(P_k) / \sum_{j=i+1}^{k} k_j) \bmod 2^{k_i} = (Sig(P_k) >> \sum_{j=i+1}^{k} k_j) \bmod 2^{k_i}$$

  for $(1 \leq i \leq k)$.

By definition, the concatenation scheme satisfies the Strong Prefix Property.

3.3.2 Improved structure of phrase signature

There are some strategies to maintain collections of phrase signatures:

- Ideally, signatures of all the prefixes of phrase P could be obtained from $Sig(P)$. Storing signatures for semi-infinite strings requires $O(nm)$ space, where n is the number of indexed phrases and m is the number of words in the text.

- To reduce space, we reduce the number of distinct prefix signatures that come from $Sig(P)$. Let $Sig(P)$ contain signatures for only P_1, P_2, ..., P_l, where P_l is the shortest prefix of P which appears only once in the text. Therefore, different phrases might have different shortest prefixes, which leads to a variable length for phrase signatures in the scheme and may still need a lot of space.

- To further reduce space, $Sig(P)$ contains signatures for only the first t prefixes of P, that is, P_1, P_2, ... P_t for some constant t. This is equivalent to the constraint that $Sig_i(w_i)$ has no bits for $i > t$. Thus significantly many more different phrases may have the same signatures.

We will use this third approach in the thesis.

In Figure 3.2, it appears that word signatures may be all of the same size. However, to use space more effectively, we use different signature sizes for different words in a phrase. So as to minimize collisions, the i-th words of phrases should have bigger signatures than the j-th words if there are more distinct i-word phrases that share the first $(i-1)$ words than distinct j-word phrases that share the first $(j-1)$ words. To use space even more effectively, different phrases may use different

numbers of bits for signatures of the ith words; that is, $Sig_i(w_i)$ need not be of constant size for all values of w_i. In fact, we will choose signature sizes independently for each block of the signature array.

3.3.3 Adjacent collisions

Let us first develop a concrete idea of the cost to store adjacent collisions. Assume that adjacent collisions occur for about 1% of the phrases, or 100 times for a block of 10,000 phrases, which is approximately what is expected for a 7-bit signature using the model from Chapter 2. An English word is about 4.5 characters in length on average, and each character is represented with 8 bits. Let a, p, w, c and b represent the number of adjacent collisions, the size of a pointer, the number of words to be encoded in a phrase signature, the number of characters per word, and the number of bits per character or space, respectively. Storing adjacent collision boundary phrases and their pointers without compression uses $a * (p + w * c * b + (w - 1) * b)$ bits. This implies about $100 * (14 + 4 * 4.5 * 8 + 3 * 8)/10,000 = 1.82$ bits per phrase, if a phrase signature is based on the first 4 words; and $100 * (14 + 5 * 4.5 * 8 + 4 * 8)/10,000 = 2.26$ bits per phrase, if a phrase signature is based on the first 5 words.

In this section, we study adjacent collisions for the concatenation scheme. We first determine a formula for the expected number of adjacent collisions in a block. Then we investigate how to minimize the expected number of adjacent collisions for a block. Finally, we determine the probability of a particular distinct i-word phrase boundary having an adjacent collision due to phrases of length greater than or equal to i.

Expected number of adjacent collisions

Assume that signature functions hash words uniformly, and that phrase signatures are based on the first t words. Let the number of distinct i-word phrases be n_i and the number of bits used for word signatures for the ith word of a phrase be k_i.

Lemma 8 *The expected total number of adjacent collisions is* $exp(A) = \sum_{i=1}^{t} \frac{n_i - n_{i-1}}{2^{k_i}}$.

Proof: The word signature space for the ith word has size $m_i = 2^{k_i}$. There are n_{ij} distinct i-word phrases for the jth $(i-1)$-word phrases. According to Lemma 2 in Section 2.3, the expected number of adjacent collisions for these n_{ij} distinct i-word phrases is $\frac{n_{ij}-1}{m_i}$.

There are n_{i-1} $(i-1)$-word phrases, so the expected number of adjacent collisions for i-word phrases is

$$\begin{aligned} & \sum_{j=1}^{n_{i-1}} \frac{n_{ij} - 1}{m_i} \\ = & \frac{\sum_{j=1}^{n_{i-1}} n_{ij} - n_{i-1}}{m_i} \\ = & \frac{n_i - n_{i-1}}{m_i} \\ = & \frac{n_i - n_{i-1}}{2^{k_i}}. \end{aligned}$$

Therefore, we have the expected total number of adjacent collisions

$$exp(A) = \sum_{i=1}^{t} \frac{n_i - n_{i-1}}{2^{k_i}}.$$

□

Notice that the expected total number of adjacent collisions is affected by the number of repetitions of any of the phrases, *i.e.*, how bits are divided among words

and how distinctions among phrases occur at different levels. It is not affected by the distribution of distinct i-word phrases within level i.

Minimize the number of adjacent collisions

Given the number of bits we are willing to allocate to each phrase signature and the count of distinct i-word phrases in the text, how should we choose word signature sizes for each word to get the minimum total expected number of adjacent collisions?

Assume that phrase signatures are based on the first t words of phrases, the number of bits in a phrase signature is L, and the number of distinct i-word phrases is n_i.

Lemma 9 *When the expected numbers of i-word adjacent collisions are equal for all values of i, the expected total number of adjacent collisions is minimum.*

Proof: From Lemma 8, we have the expected total number of adjacent collisions $exp(A) = \sum_{i=1}^{t} \frac{n_i - n_{i-1}}{2^{k_i}}$, where n_i is the number of distinct i-word phrases, and k_i is the number of bits of word signatures for the ith word of a phrase. To simplify the description, let $x_i = \frac{(n_i - n_{i-1})}{2^{k_i}}$, then we have

$$exp(A) = \sum_{i=1}^{t} x_i.$$

Since the number of bits in a phrase signature $L = \sum_{i=1}^{t} k_i$ is fixed, $\prod_{i=1}^{t} 2^{k_i}$ is a constant, and therefore since n_i are also fixed, $\prod_{i=1}^{t} x_i = \prod_{i=1}^{t}((n_i - n_{i-1})/2^{k_i})$ is a constant. Let $\prod_{i=1}^{t} x_i$ be C. So, we have

$$\prod_{i=1}^{t} x_i - C = 0.$$

Let λ be a new variable, and

$$f(x_1, x_2, \ldots, x_t, \lambda) = \sum_{i=1}^{t} x_i - \lambda(\prod_{i=1}^{t} x_i - C).$$

According to the method of Lagrange multipliers [Swo81], the values of x_i which give the extrema of $exp(A) = \sum_{i=1}^{t} x_i$ are among the simultaneous solutions of $f'_{x_i} = 0$ and $f'_{\lambda} = 0$ for $i = 1, 2 \ldots t$, where f'_{x_i} and f'_{λ} are the first partial derivatives of f. So, $exp(A) = \sum_{i=1}^{t} x_i$ reaches its minimum when $x_i = \lambda C$ for $i = 1, 2 \ldots t$.

Therefore, when the expected numbers of i-word adjacent collisions are equal for $i = 1, 2 \ldots t$, the expected total number of adjacent collisions has its minimum value. □

Lemma 10 *When the number of bits for the ith word signature is* $k_i = \log_2 \frac{n_i - n_{i-1}}{C}$, *the expected total number of adjacent collisions is minimum and the number of i-word adjacent collisions is* C *for* $i = 1, 2 \ldots t$, *where* $C = \sqrt[t]{\frac{\prod_{i=1}^{t}(n_i - n_{i-1})}{2^L}}$ *and* L *is the total number of bits in a phrase signature.*

Proof: According to Lemma 9, the expected total number of adjacent collisions is minimum when the expected numbers of i-word adjacent collisions are equal for all values of i. Assume that the expected number of i-word adjacent collisions is

$$C = \frac{n_i - n_{i-1}}{2^{k_i}}$$

for $i = 1, 2 \ldots t$. So, we have

$$C^t = \frac{\prod_{i=1}^{t}(n_i - n_{i-1})}{2^{\sum_{i=1}^{t} k_i}}.$$

Therefore, when the number of bits for the ith word signature is

$$k_i = \log_2 \frac{n_i - n_{i-1}}{C}.$$

the expected number of i-word adjacent collisions is

$$C = \sqrt[t]{\frac{\prod_{i=1}^{t}(n_i - n_{i-1})}{2^L}}.$$

where L is the length of a phrase signature. □

Corollary 4 *Under the conditions in Lemma 10, the expected number of adjacent collisions in a block having n_i distinct i-word phrases for $1 \leq i \leq t$ is tC.*

Probability of an i-word phrase being listed in a look-aside table because of adjacent collisions

As indicated in Section 1.2.4, the search algorithm to be used depends on storing adjacent collisions in a look-aside table. Adjacent collisions partition a list of phrases into sublists and searches are performed on one of these smaller sublists. Therefore, the number of adjacent collisions is an important performance factor for estimating search time.

For a list of i-word phrases, adjacent collisions affecting search performance are not only from the ith words of these phrases, but from higher levels of words as well. For example, in Figure 3.3, word w_1 has 8 distinct following words. Words w_{13} and w_{14} in the shaded area have the same signature. So, an adjacent collision occurs between phrases 'w_1 w_{13}' and 'w_1 w_{14}'. Since the phrase 'w_1 w_{14}' is recorded in the look-aside table because of this adjacent collision, the word w_1 is effectively also listed in the look-aside table. Therefore, an i-word phrase may be listed in the look-aside table because of adjacent collisions occurring at the ith or higher level.

In this section, we study cross-impacts of adjacent collisions from different levels, more specifically, the probability of an i-word phrase being listed in a look-aside table because of an adjacent collision at the ith or higher level.

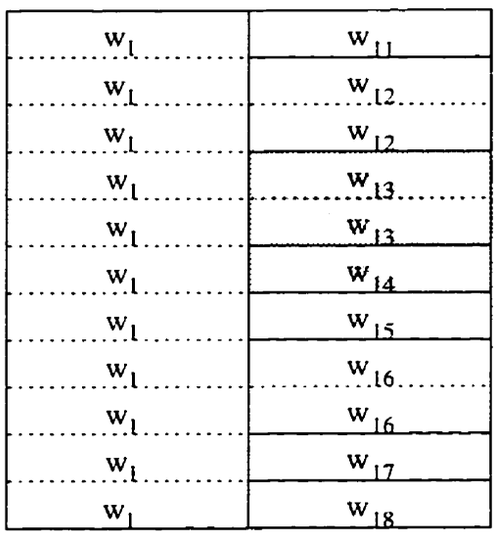


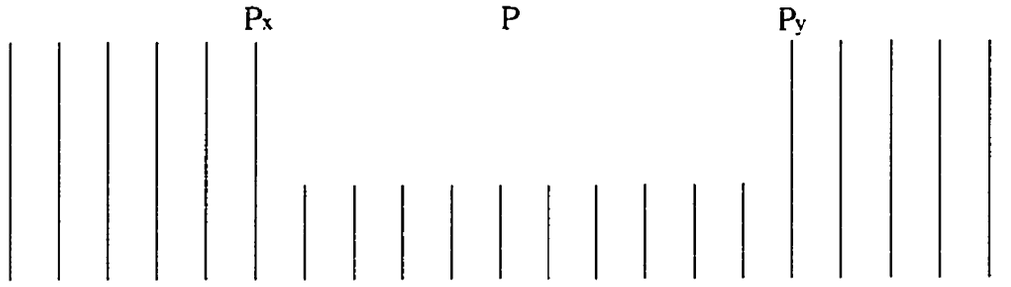
Figure 3.3: Adjacent collisions

Definition 8 (Collision probability and no-collision probability of two phrases) *The collision probability of two adjacent phrases having the first $i-1$ words in common is the probability p_i that the signatures of the ith words collide with each other. The no-collision probability is $1-p_i$.*

If bits are allocated to word signatures to minimize adjacent collisions, then the expectation of an adjacent collision is independent of the level of the collision. Therefore, $p_i = \frac{c}{n_i - n_{i-1}}$.

Assume phrase signatures are based on the first t words of phrases. In the following figure, phrases are in alphabetic order, 10 shadowed phrases in the center have P as their i-word prefixes, and, P_x and P_y are the left and the right adjacent phrases whose i-word prefixes are not P.

If P_x and P collide, P will be listed in the look-aside table. If P and P_y collide, P_y will be listed in the look-aside table. Thus, in order to not list P in the look-aside table, P_x and P should not collide and all the adjacent pairs of the 10 phrases of P



should not collide. From this we see that, the probability of an i-word phrase P not appearing in the look-aside table because of adjacent collisions is the no-collision probability between P and its left adjacent phrase times no-collision probabilities of adjacent pairs of all occurring t-word phrases having P as the i-word prefix.

Lemma 11 *Assume an i-word phrase P differs from phrase P_x first in word j, and there are N_k distinct k-word phrases having prefix P, for $j \leq i < k \leq t$. If bits are allocated to word signatures to minimize adjacent collisions, P will appear in the look-aside table with an expected value*

$$1 - (1 - \frac{C}{n_j - n_{j-1}}) \prod_{k=i+1}^{t} (1 - \frac{C}{n_k - n_{k-1}})^{N_k - N_{k-1}}.$$

Proof: If bits are allocated to word signatures to minimize adjacent collisions, the collision probability of two adjacent phrases having the first $k-1$ words in common is $p_k = \frac{C}{n_k - n_{k-1}}$. Among the phrases having a prefix P, there are $N_k - N_{k-1}$ boundaries at which adjacent pairs have the first $k-1$ words in common and differ at the kth words. Then the probability of P not appearing in the look-aside table is $(1 - \frac{C}{n_j - n_{j-1}}) \prod_{k=i+1}^{t} (1 - \frac{C}{n_k - n_{k-1}})^{N_k - N_{k-1}}$. □

3.3.4 Compressing phrase signatures

We notice that, in most texts, phrases distribute in such a way that many words or phrases occur multiple times. When phrases are ordered lexicographically, some

prefixes are listed repeatedly. Subsequently, signatures of these prefixes are repeated multiple times if signatures are stored word-wise instead of phrase-wise.

Therefore, to save storage costs, we substitute consecutively repeated word signatures by the word signature and the number of repetitions. So, word-wise signatures consist of word signatures and counts. An extra bit is used to indicate whether it is a word signature or a count.

Since the compression uses an extra bit to distinguish signatures and counts, compressions are not going to be performed when numbers of bits in word signatures are small.

As a second heuristic, at lower levels (like the first or the second word in a phrase), we could use more bits in word signatures if we prefer to distinguish phrases as early in levels as possible. So, numbers of bits used for words at lower levels are relatively big, and word signatures at these levels can be compressed to a large degree.

3.3.5 Experimental results

In this section we discuss the allocation of bits among words, the number of bits and the number of words in a phrase signature, and the block size for the proposed method.

Allocate bits among words

The length of a phrase signature is L bits. How best to divide L bits among words of a phrase depends not only on the phrase distribution in a block but also on the expected pattern of phrase searches.

Since adjacent collisions need space to be stored, the number of total adjacent collisions in a block may be considered as a factor in choosing the numbers of bits for word signatures. If L bits are divided among words in such a way that the number of total adjacent collisions is minimum, the size of the look-aside table is nearly minimized. We shall call this strategy *Balance*. Lemma 10 in Section 3.3.3 says when the number of bits for the ith words k_i satisfies $2^{k_i} = \frac{n_i - n_{i-1}}{C}$, the total number of adjacent collisions is minimum and the number of adjacent collisions at each level is C, where $C = \sqrt[t]{\frac{\prod_{i=1}^{t}(n_i - n_{i-1})}{2^L}}$ and L is the number of bits in a phrase signature. The number of i-word phrases is influential.

Minimizing the total number of adjacent collisions does not always guarantee a good search performance, because the presence of adjacent collisions reduces the range of the array over which searches are actually conducted.

Let us look at another strategy of allocating bits. The more distinct words there are, the bigger hash space is needed to reduce collisions. So, the maximum number of distinct words following all $(i-1)$-word prefixes is an important factor for allocating bits to the ith word signature. Thus, one relevant measure is the maximum load factor $\hat{n}_i/m_i$, where $\hat{n}_i$ is the maximum number of distinct words following any $(i-1)$-word prefix and m_i is the size of the signature space of the ith word of a phrase. We will call this way of allocating bits *ByMax* in the following discussion.

Some experiments were done on the text of the *Bible* by using the two different bit allocation algorithms. The *Bible* is approximately 5.6Mb and has around 1.13M indexed phrases (see Appendix A). Parameters that affect performance are the number of words in a phrase signature, the size of a block, and the maximum number of bits in a phrase signature. Each different experiment takes a different combination of parameter values, which are fixed in each experiment. The number

of words in a phrase ranges from 3 to 7. The size of a block is 1k, 5k, or 10k indexed phrases. The maximum length of a phrase signature is 16 bits, 24 bits, or 32 bits respectively.

If the maximum number of bits in a phrase signature is x and there are not many distinct phrases in a block, the number of bits in a phrase signature will be chosen to be fewer than x bits in this block. For the i-word phrase having the biggest number of distinct following words (the $(i+1)$th words), the lower bound of load factors of the word signature space for the $(i+1)$th words is set to be around 1.

Search performance and space usages of the two algorithms are plotted in the error-bar style (one of *Gnuplot*'s displaying styles [WK]). The values of the algorithm *Balance* (to minimize the total number of adjacent collisions) are represented by "$\diamond$", and the values of the algorithm *ByMax* (to use the maximum numbers of distinct i-word phrases for $(i-1)$-word phrases) are represented by "$-$". The values of the two algorithms using the same group of parameters are connected by a vertical line. The longer the line, the bigger the difference between values of the two algorithms.

Let us look at space usages first. The x-axis represents primarily the size of the signature: the major divisions mark the number of words in a phrase signature, which ranges from 3 to 7; in each interval, these minor divisions represent the maximum lengths of 16, 24, and 32 bits for a phrase signature. In each subinterval, there are three samples corresponding to the block sizes of 1k, 5k, and 10k respectively. The y-axis represents the numbers of bits per index point.

Figure 3.4 shows numbers of bits used in a phrase signature. Figure 3.5 shows numbers of bits used for storing adjacent collision boundaries. We observe that

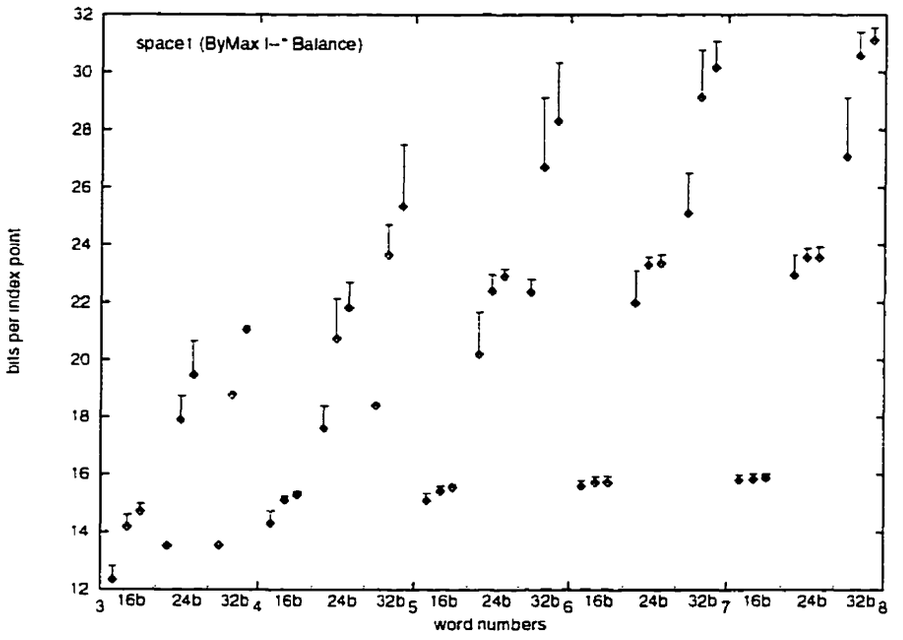


Figure 3.4: Bits in a phrase signature

Balance generally uses less space than *ByMax* both for a phrase signature and for adjacent collisions. So, the total space used per index point, for the signature array and the look-aside table, is smaller using *Balance* than *ByMax*, as shown in Figure 3.6. Note that the space used by the look-aside table includes adjacent collisions and block boundary phrases, as shown in Figure 1.4.

Although *ByMax* uses more space for a phrase signature, its word signatures are more compressible than *Balance*, as shown in Figure 3.7 which shows numbers of bits in a phrase signature after compression (as described in Section 3.3.4) is applied to the signature array. This results from the fact that *Balance* recognizes how phrases distribute between different levels, but ignores how phrases distribute locally within a level. In fact, Lemma 10 in Section 3.3.3, on which *Balance* is based, actually assumes the most skew phrase distribution, in which all the i-word phrases except one have only one distinct following word. Normally the numbers

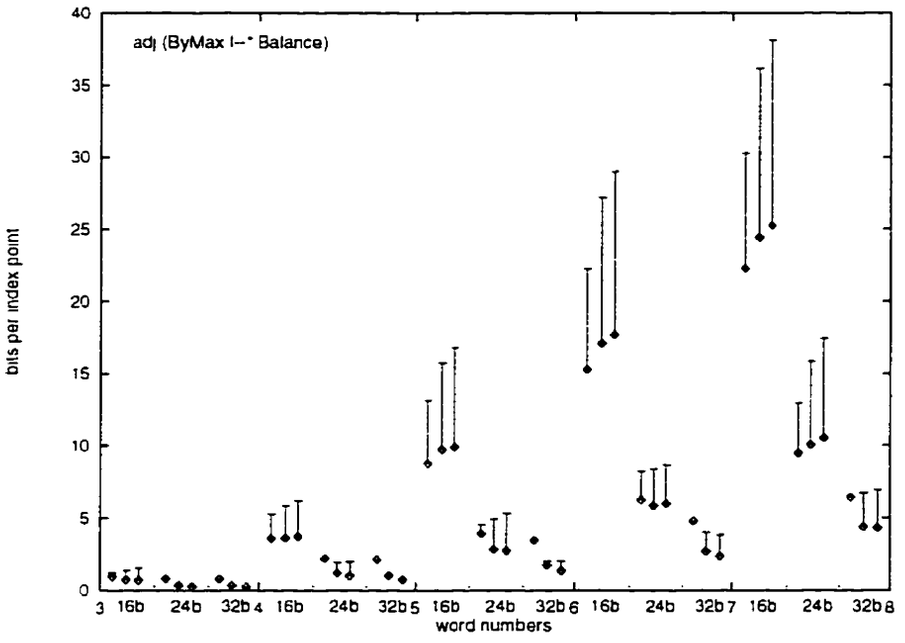


Figure 3.5: Bits for adjacent collisions

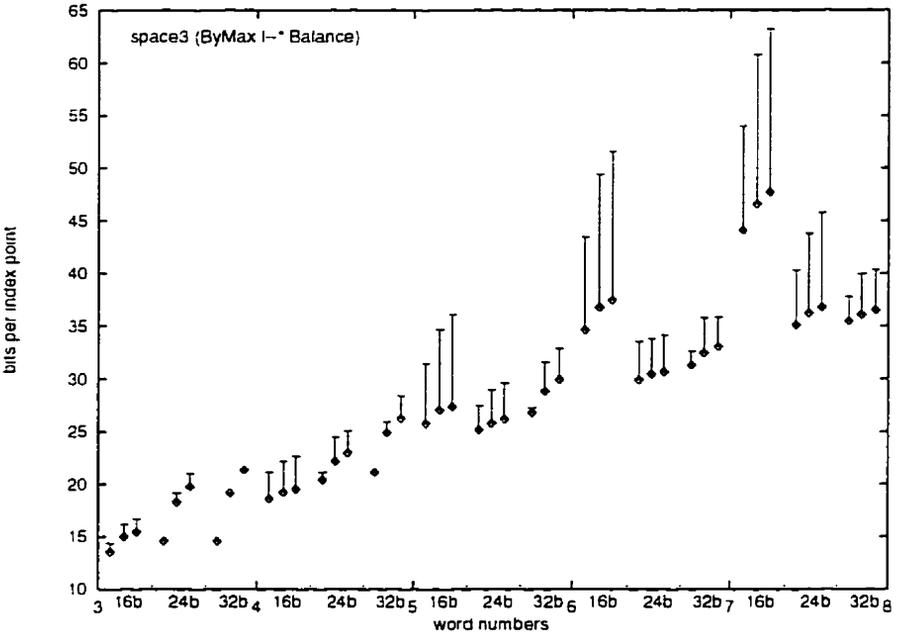


Figure 3.6: Total bits per index point

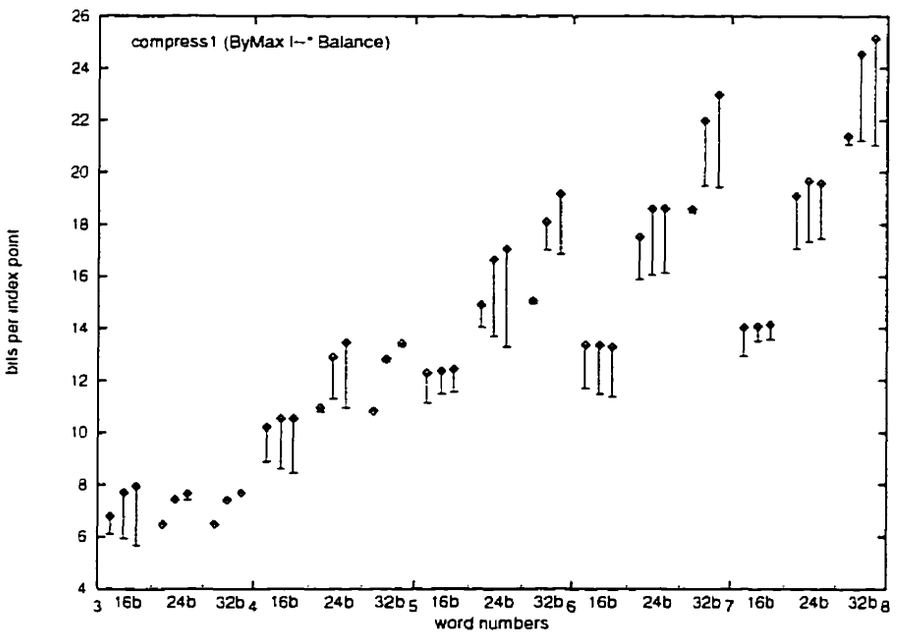


Figure 3.7: Bits for a phrase signature after compression

of distinct following words of i-word phrases are all much smaller than $(n_i - n_{i-1})$. Therefore, *Balance* in general gives a relative high load factor at the first level and lower load factors at other levels.

But, if compression is applied to the signature array, the two algorithms use a similar amount of total space for the signature array and the look-aside table combined, except when the number of bits in a phrase signature is too small for the number of words, such as 16 bits for 6 words as shown in Figure 3.8.

Figures 3.6 and 3.8 indicate that for a fixed number of bits in a phrase signature, the difference in the total space between the two algorithms, both before and after compression, increases as the number of words in a phrase signature increases.

Figure 3.9 plots the search performance for the cases where a phrase signature is based on 5 words and the maximum number of bits in a phrase signature is 32.

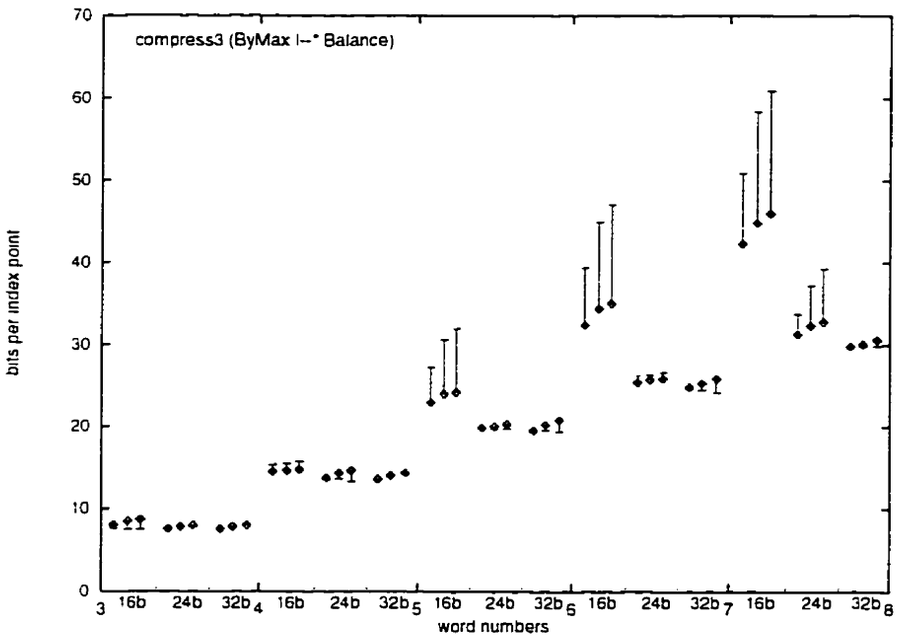


Figure 3.8: Total bits per index point after compression

assuming each distinct i-word phrase is searched once using the algorithm described in Section 1.2. The x-axis represents the number of words of a phrase. There are three points in each interval on the x-axis, corresponding to the block sizes of 1k, 5k, and 10k respectively. The y-axis is the expected number of disk accesses to a text using binary search and scanning as shown in Figure 1.7. We observe that *ByMax* and *Balance* are very close for phrases longer than 2 words. For one or two word phrases, *ByMax* is faster than *Balance*. For all cases, the expected number of disk accesses to a text to search for a phrase of up to 5 words is less than 1.1.

Generally, we could assign bits to lower levels of words (*i.e.*, the first or the second word in a phrase) more generously in order to distinguish phrases as early as possible. This improves the performance of short phrases, which are most commonly requested in practice. Furthermore, a better performance at lower levels makes the performance at higher levels better as well, since more phrases that have different

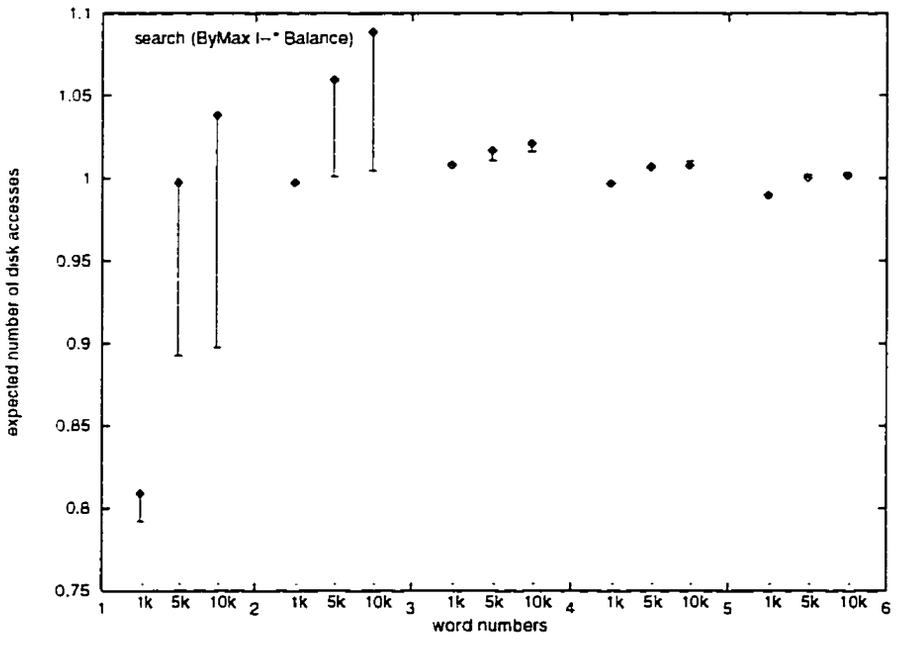


Figure 3.9: Search performance for 32 bits 5-word signatures

prefixes but the same following words are more likely distinguished already by the signatures of their prefixes. Thus, factors other than minimizing the total number of adjacent collisions might influence how best to allocate bits for individual word signatures.

In summary, our experiments show that the search performance of *Balance* is good enough. Although, *Balance* uses a similar amount of space as *ByMax* after compression for a reasonable combination of the number of bits and the number of words in a phrase signature, it uses less space than *ByMax* without the signature array compression, as predicted by the theory. We suggest using *Balance* to allocate bits among words in a phrase.

The number of bits and the number of words in a phrase signature

How many bits to use in a phrase signature and how many words are included in a phrase signature are determined by search requirements and space requirements. In general, the more words a phrase signature contains, the more bits a phrase signature needs to maintain a reasonable load factor at each level.

The two numbers affect each other. Some other factors, for example, the block size and the number of adjacent collisions we can accept, may affect the two parameters as well.

The experiments described in the previous section use different combinations of the maximum number of bits, the number of words, the block size, and a bit allocation algorithm. Let us look at the results from a different perspective.

As shown in Figure 3.5, the amount of space used for storing adjacent collisions increases as the number of words included in a phrase signature increases. When the number of words in a phrase signature is no more than 3, 5, and 6 words for the maximum allocation of 16 bits, 24 bits, or 32 bits respectively, the space to store adjacent collisions is less than 5 bits per index point.

Figure 3.6 shows that when the number of words in a phrase signature is no more than 3, 5, and 6 words for the maximum allocation of 16 bits, 24 bits, or 32 bits respectively, the total space is around or less than 16 bits, 24 bits, or 32 bits for strategy *Balance*.

Figure 3.8 indicates that if run-length compression (as described in Section 3.3.4) is used on the signature array, a longer phrase signature uses less space than a shorter phrase signature as the number of words in a phrase signature increases. This is because for these measurements the look-aside table is not compressed.

Too many adjacent collisions indicates that the total number of bits in a phrase signature is too small, or alternatively that too many words are included in a phrase signature, assuming that bits are properly divided among words. We may need to increase the number of bits in a phrase signature or reduce the number of words in a phrase signature.

On the other hand, if the number of adjacent collisions is very small, we could include more words in a phrase signature, or reduce the total space by reducing the total number of bits in a phrase signature with a tolerable increase in the number of adjacent collisions.

As a final experiment, total space was examined for the suffix-signature method, taking into account subdivision of the text and of the suffix array into blocks as described in Figure 1.3. The total space used by the signature array and storing adjacent collision boundaries and block boundaries for different block sizes is shown in Figure 3.10. The x-axis represents the block size, and the y-axis represents the number of total bits per index point. Values before compression of the signature array are represented by "$\diamond$", and values after the compression are represented by "$+$". We observe that when the size of a block is small, bits can be allocated among words more suitably to each particular block, but more space is required for the block list to store block boundaries.

3.4 Alternative phrase signature schemes

In this section, we study two alternatives to the concatenation scheme to create a phrase signature from word signatures. One is the Chinese remainder approach. Another is the superimposition scheme.

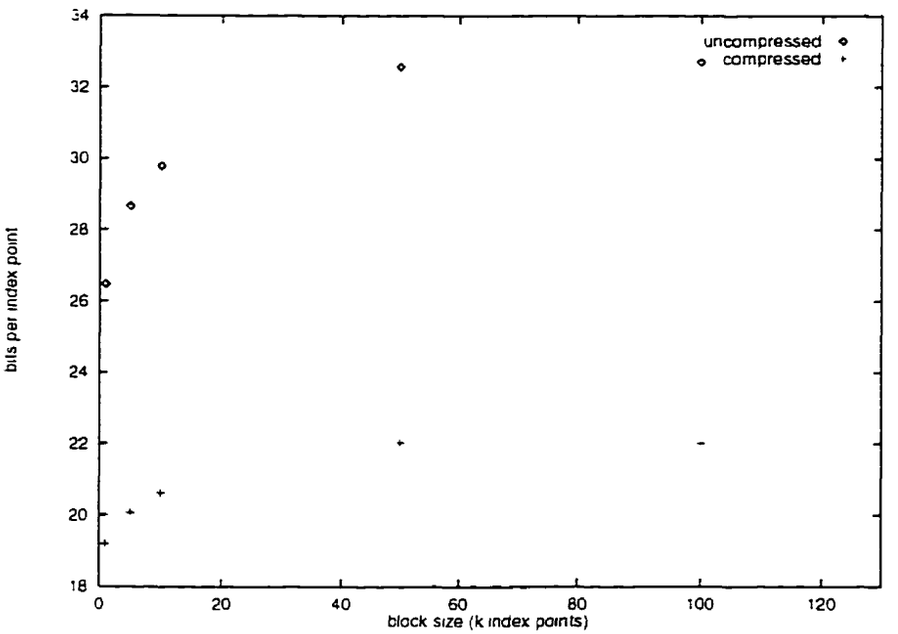


Figure 3.10: Total bits used by different block sizes

3.4.1 Chinese remainder approach

In this section, we describe an implementation of phrase signatures based on the Chinese remainder theorem and modular arithmetic.

Chinese remainder theorem [AHU74]: *Assume that* $p_1, p_2, \ldots, p_n$ *are a set of pairwise relatively prime integers, and* $u_1, u_2, \ldots, u_n$ *are a set of residues. Let* $p = \prod_{i=1}^{n} p_i$, $c_i = p/p_i$, *and* $d_i = c_i^{-1}$ *modulo* p_i *(that is,* $d_i c_i = 1$ *modulo* p_i*). Then we have* $u = \sum_{i=1}^{n} c_i d_i u_i$ *modulo* p, *and* $u_i = u$ *modulo* p_i.

Let us look at an example [AHU74]. Assume that $(p_1, p_2, p_3, p_4) = (2, 3, 5, 7)$, and $(u_1, u_2, u_3, u_4) = (1, 2, 4, 3)$. We have that

$d_1 = (3 * 5 * 7)^{-1}$ modulo 2 = 1, since $1 * 105 = 1$ modulo 2.

$d_2 = (2 * 5 * 7)^{-1}$ modulo 3 = 1, since $1 * 70 = 1$ modulo 3.

$d_3 = (2 * 3 * 7)^{-1}$ modulo 5 = 3, since $3 * 42 = 1$ modulo 5.

$d_4 = (2 * 3 * 5)^{-1}$ modulo $7 = 4$, since $4 * 30 = 1$ modulo 7.
So, $u = (105 * 1 * 1 + 70 * 1 * 2 + 42 * 3 * 4 + 30 * 4 * 3)$ modulo $210 = 59$, and $(u$ modulo p_1, u modulo p_2, u modulo p_3, u modulo $p_4) = (1, 2, 4, 3)$.

Assume that $Sig(P)$ contains the information of the first k prefixes of phrase P. The implementation of phrase signatures based on the Chinese remainder theorem is as follows:

- Choose k pairwise relatively prime integers, $p_1, p_2, \ldots, p_k$, that are closest to selected signature sizes of words in phrases.
- Let $c_i = (\Pi_{j=1}^{k} p_j)/p_i$ and $d_i = c_i^{-1}$ modulo p_i.
- The phrase signature function is

$$Sig(P) = \sum_{i=1}^{k} Sig(w_i) c_i d_i \text{ modulo } p$$

- The functions converting a phrase signature into its word signatures are

$$Sig(w_i) = Sig(P) \text{ modulo } p_i$$

  for $(1 \leq i \leq k)$.

To search for a phrase P, we search for a signature x in the signature file such that $Sig(w_i) = x$ modulo p_i for each word w_i in P, where $(1 \leq i \leq k)$.

Like the concatenation approach, the Chinese remainder approach satisfies the phrase prefix property. The phrase signature space S has k dimensions and is the Cartesian product of k word signature spaces $S_1, S_2, \ldots, S_k$. The size of space S_i is 2^{k_i} in the concatenation approach or p_i in the Chinese remainder approach. Thus there is more flexibility in allocating bits to words in the latter approach.

In the concatenation approach, prefixes of phrase signatures are identical for different phrases having identical prefixes. In the Chinese remainder approach, two phrases do not necessarily share any bits even if they have an identical prefix. So, signature files built on the concatenation approach have more repeated patterns, and therefore are more compressible than the ones built on the Chinese remainder approach. Furthermore, it is much cheaper to extract word or prefix signatures from phrase signatures in the concatenation approach than in the Chinese remainder approach.

3.4.2 Superimposition scheme

In this section, we study an alternative based on a more conventional phrase signature scheme. Instead of concatenating word signatures, we superimpose word signatures to get a phrase signature [Fal92].

First, we describe the superimposition scheme. Then we study the relationship between the number of adjacent collisions and the word signature sizes. We compare the superimposition scheme and the concatenate scheme, including some experimental results on the *OED* (which is described in Appendix A).

Word and phrase signatures

In the superimposition scheme, phrase signatures are created differently from the concatenation scheme. Word signatures are bit patterns of the same size and are bitwise *OR*-ed together to form a phrase signature. Faloutsos has shown that under optimal design, half of the bits in the phrase signature should be set to 1 [Fal88]. We will investigate the possibility of using the superimposition scheme in phrase searches and study adjacent collisions for the scheme.

Word signatures and phrase signatures are of the same length. Word signatures may be formed using any suitable hashing scheme. The signature of the ith word has b_i bits set to 1. We superimpose (using bitwise OR) the signatures of the first k words in a phrase as the phrase signature. In total, we wish to have about half of the bits set to 1 in a phrase signature.

Figure 3.11 is an example showing how a phrase signature is created. The phrase signature and word signatures are all 32 bits long. In our example the first word in the phrase has 3 bits set, the 2nd to the 5th words have 7, 6, 5, and 4 bits set to 1 respectively. The phrase "book is red and old" has a signature 00000111 11010111 11010110 01100011, with 18 bits set to 1.

word		signature			
(1st: 3 bits set)	book	00000010	00010000	00010000	00000000
(2nd: 7 bits set)	is	00000010	00000011	01000010	00100001
(3rd: 6 bits set)	red	00000001	00010001	01000000	01100000
(4th: 5 bits set)	and	00000100	01000100	10000000	00000010
(5th: 4 bits set)	old	00000000	10000000	01000100	01000000
phrase signature		00000111	11010111	11010110	01100011

Figure 3.11: Illustration of a phrase signature

Definition 9 (Contain) *Let s_1 and s_2 be two signatures of the same size. The signature s_1 contains signature s_2 if $s_1 = s_1|s_2$, where '$|$' represents bitwise boolean or.*

By definition, we have that in the superimposition scheme a phrase signature contains the signatures of all its prefixes. Thus, given a phrase P_i and a phrase signa-

ture Sig. if P_i is a prefix of any phrase P having Sig as its phrase signature. the signature of P_i must be contained by Sig.

Lemma 12 *The superimposition scheme satisfies the Weak Prefix Property of phrase signatures.*

Word signature spaces are bigger in the superimposition scheme than in the concatenation scheme. But in the concatenation scheme two phrases collide only when they have the same signature. while in the superimposition scheme they may collide even when they do not have the same signatures. For example. if the signatures of words w_1.w_2 and w_3 are 01001001. 11000010. and 11001001 respectively. then the signature of the phrase w_1 w_2 is 11001011. which contains the signature of w_3 although the signature of neither w_1 nor w_2 contains it.

Adjacent collisions

Is the superimposition scheme better or worse than the concatenation scheme in terms of numbers of collisions? Let us look at adjacent collisions for the superimposition scheme.

Assume that signatures are of n bits. a phrase signature is constructed from superimposing signatures of the first k words. and the number of bits to be randomly set at the ith word is b_i.

Let S be an integer of n bits with each bit initially 0. First. we set b_1 bits of S according to the signature of the first word of a phrase P. Then. we set b_2 bits of S based on the signature of the second word. We continue to set bits in S like this until the kth word of the phrase P. So. the value in S is the i-word

phrase signature $Sig(P_i)$ after processing the ith word signature, and it is the phrase signature $Sig(P)$ after processing k words.

Lemma 13 *The probability of a bit set to* 1 *in a* k*-word phrase signature is*

$$1 - \frac{\prod_{i=1}^{k}(n - b_i)}{n^k}.$$

The expected number of bits set in a k*-word phrase signature is*

$$n(1 - \frac{\prod_{i=1}^{k}(n - b_i)}{n^k}).$$

Proof: b_i is the number of 1's in the signature of the ith word. So, with a probability of $\frac{n-b_i}{n}$, a bit in a phrase signature would not be set by the ith word. Thus, with a probability of $\frac{\prod_{i=1}^{k}(n-b_i)}{n^k}$, a bit would not be set by the k words. This proves that in a k-word phrase signature the probability of a bit set is $1 - \frac{\prod_{i=1}^{k}(n-b_i)}{n^k}$, and the expected number of bits set is $n(1 - \frac{\prod_{i=1}^{k}(n-b_i)}{n^k})$. □

Lemma 14 *If* m *of* n *bits are initially set and* j *of* n *bits are randomly chosen to be also set, the probability of the total number of bits set being* s *is* $P(s, j, m, n) = \binom{m}{j+m-s} * \binom{n-m}{s-m}/\binom{n}{j}$ *if* $\max(m, j) \leq s \leq \min(n, m + j)$, *and 0 otherwise.*

Proof: Assume that m of n bits are set in S_1, and j of n bits are set in S_2. $S_1|S_2$ denotes S_1 bitwise *OR*ed with S_2. Then, i of n more bits are set in $S_1|S_2$ than in S_1 with probability $\binom{m}{j-i} * \binom{n-m}{i}/\binom{n}{j}$ if $|\min(m - j, 0)| \leq i \leq \min(j, n - m)$, and 0 otherwise. Because the total number of bits set is $s = i + m$, we substitute $s - m$ for i to prove the lemma. □

Figure 3.12 plots the probability of a signature containing j bits that are randomly chosen, that is, $P(s, j, m, n)$ in Lemma 14 for $s = m$. A signature is $n = 32$

bits long, and initially has m bits set to 1 for $m = 14, \ldots, 18$ (about $n/2$). The x-axis represents j, the number of bits chosen randomly, and the y-axis is the probability. Results are denoted by "$\diamond$", "$+$", "$\square$", "$\times$", and "$\triangle$" for m, the numbers of bits set initially, of 14, 15, 16, 17, and 18 respectively.

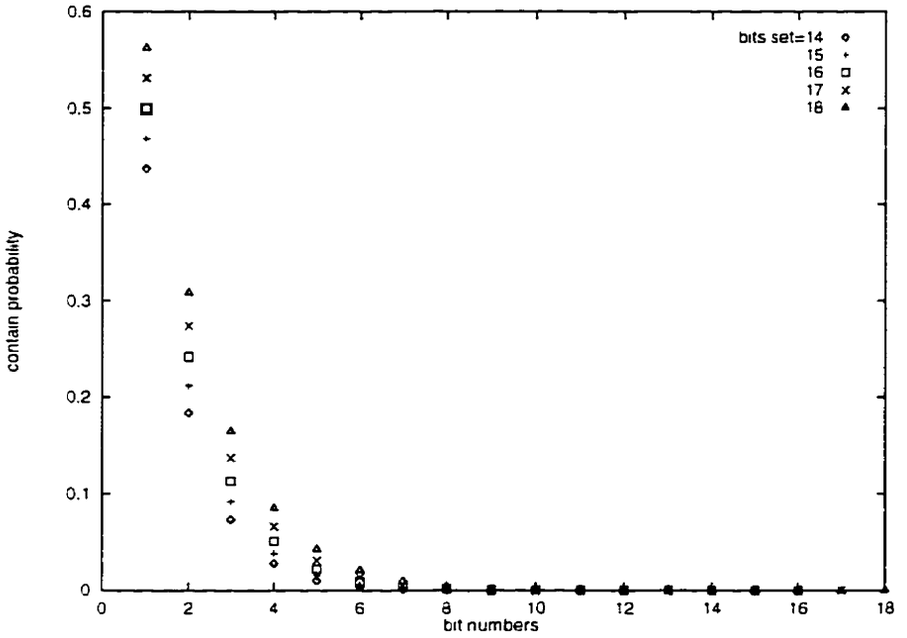


Figure 3.12: Probability of containing j bits

Let us look at the probability that s bits are set in a phrase signature.

Corollary 5 *Assume that $b_i < n$ bits are set for each word w_i in a phrase. The probability that s bits are set in S after the 1st word is*

$$newp_1(s, b_1, n) = \begin{cases} 1. & \text{if } s = b_1. \\ 0. & \text{otherwise.} \end{cases}$$

The probability of s bits being set after the first i words is

$$newp_i(s, b_1, b_2, \ldots, b_i, n) = \begin{cases} \sum_{j=\hat{d}_{i-1}}^{d_{i-1}} P(s, b_i, j, n) * newp_{i-1}(j, b_1, b_2, \ldots, b_{i-1}, n), \\ \qquad\qquad\qquad\qquad \text{if } \hat{d}_i \le s \le d_i, \\ 0, \qquad\qquad \text{otherwise.} \end{cases}$$

where $\hat{d}_i = \max_{1 \le m \le i} b_m$, $d_i = \min(n, \sum_{m=1}^{i} b_m)$, and n is the total number of bits in a signature.

Proof: Since S is initially 0 at every bit and there are b_1 bits set for the first word, we have

$$newp_1(s, b_1, n) = \begin{cases} 1, & \text{if } s = b_1, \\ 0, & \text{otherwise.} \end{cases}$$

So, the probability of s bits set after the 2nd word is $newp_2(s, b_1, b_2, n) = P(s, b_2, b_1, n)$, where $\max(b_1, b_2) \le s \le \min(n, b_1 + b_2)$. Let the probability that j bits are set after $(i-1)$ words set be $newp_{i-1}(j, b_1, b_2, \ldots, b_{i-1}, n)$, where j is in $[\hat{d}_{i-1}, d_{i-1}]$. Notice that

$$\sum_{j=\hat{d}_{i-1}}^{d_{i-1}} newp_{i-1}(j, b_1, b_2, \ldots, b_{i-1}, n) = \sum_{j=0}^{d_{i-1}} newp_{i-1}(j, b_1, b_2, \ldots, b_{i-1}, n) = 1$$

for $i > 1$. Thus, the probability of s of n bits being set afterwards is

$$newp_i(s, b_1, b_2, \ldots, b_i, n) = \sum_{j=\hat{d}_{i-1}}^{d_{i-1}} P(s, b_i, j, n) * newp_{i-1}(j, b_1, b_2, \ldots, b_{i-1}, n),$$

for $\hat{d}_i \le s \le d_i$; and 0 otherwise. □

Lemma 15 *Assuming perfect signature functions, the probability that a k-word phrase signature S contains a given word signature for level i is*

$$contain(i, b_1, b_2, \ldots, b_k, n) = \sum_{j=b_i}^{d_k} [\binom{j}{b_i} / \binom{n}{b_i} * newp_k(j, b_1, b_2, \ldots, b_k, n)].$$

and the probability that S contains the signature of a given i-word phrase is

$$contain'(i, b_1, b_2, \ldots, b_k, n) = \sum_{j=\hat{d}_k}^{d_k} (\prod_{t=1}^{i} [\binom{j}{b_t} / \binom{n}{b_t}]) * newp_k(j, b_1, b_2, \ldots, b_k, n).$$

where $\hat{d}_k = \max_{1 \le m \le k} b_m$. *and* $d_k = \min(n, \sum_{m=1}^{k} b_m)$.

Proof: b_i is the number of bits set for signatures of words at the level i. $\binom{j}{b_i} / \binom{n}{b_i}$ is the probability of j bits containing b_i bits for $j \ge b_i$, 0 otherwise. $\prod_{t=1}^{i} [\binom{j}{b_t} / \binom{n}{b_t}]$ is the probability of j bits containing a given i-word phrase for $j \ge \hat{d}_k$, 0 otherwise. □

Lemma 16 *The number of adjacent collisions occurring because of the ith words for a list of phrases is*

$$adj\_num(i, b_1, b_2, \ldots, b_i, n) = (2c_i - c_i * c_i) * (n_i - n_{i-1}).$$

where $c_i = contain(i, b_1, b_2, \ldots, b_k, n)$. *and* n_i *is the number of distinct i-word phrases with* $n_0 = 1$.

Proof: Let P_1 and P_2 be two adjacent phrases that have the identical first $(i-1)$-words and differ at the ith words. An adjacent collision occurs between P_1 and P_2 if either the signature of P_1 contains the signature of the ith word of P_2, or the signature of P_2 contains the signature of the ith word of P_1. Thus, the probability of an adjacent collision occurring between P_1 and P_2 is $2 * c_i - c_i * c_i$, where $c_i = contain(i, b_1, b_2, \ldots, b_k, n)$. Since the number of boundaries of distinct i-word phrases is $(n_i - 1)$, the number of distinct boundaries introduced by the ith-word level is $(n_i - 1) - (n_{i-1} - 1) = n_i - n_{i-1}$. Therefore, the number of adjacent collisions occurring because of the ith-words is $adj\_num(i, b_1, b_2, \ldots, b_i, n) = (2 * c_i - c_i * c_i) * (n_i - n_{i-1})$. □

The superimposition scheme versus the concatenation scheme

We compare the superimposition scheme and the concatenation scheme in this section. We will compare them in terms of collisions, more specifically, in terms of adjacent collisions, because storing the boundaries of adjacent collisions needs extra space.

We list some differences between the two schemes as follows.

- In the superimposition scheme, each word signature has the same number of bits as a phrase signature, whereas a word signature has fewer bits in the concatenation scheme.
- In the superimposition scheme, two phrase signatures collide if one is any subset of the other, whereas in the concatenation scheme, they collide only if the prefixes are identical.
- In the superimposition scheme, if most bits are 1 or most bits are 0 in a phrase signature, it will collide with many other phrase signatures, because it contains or is contained by many phrase signatures. To maximize the information content of a phrase signature, approximately half the bits should be set [Fal88]. Thus the superimposition scheme does not use the full space of 2^L, where L is the number of bits in a phrase signature. In the concatenation scheme, each word could use its full, albeit smaller word signature space.
- In the superimposition scheme, a phrase signature does not keep the order information of words of a phrase, and words in different positions affect each other. The signature of the third word of one phrase, for instance, may contain the signature of the second word signature of another, or the superimposition of the third and the fourth words may contain the signature of the second

word. In the concatenation scheme, two words collide only when they have the same signature and appear in the same positions in their phrases.

To compare the two phrase signature schemes, we calculated the theoretical and the experimental numbers of adjacent collisions for various combinations of bits allocated or set for word signatures on some blocks of data from the *OED* (see Appendix A).

The concatenation scheme

We experimented with various combinations of word signature sizes on some blocks of data in the *OED*. Each block has about 2^{13} indexed phrases such that phrases in each block start with the same word. Thus phrase signatures are based on the 2nd to the 5th words of phrases.

For most of these blocks, *ByMax*, a bit allocation strategy limiting the maximum load factor at each level (see Section 3.3.5), gives 29 bits in total for a phrase signature, so the total number of bits in a phrase signature was chosen to be 29 in our experiments.

The number of bits in a signature of the ith word in a phrase is k_i. We used different combinations of (k_2, k_3, k_4, k_5) with k_i in the range of $[1, 26]$ to calculate the theoretical numbers of adjacent collisions using Lemma 8 (Section 3.3.3). We chose 100 combinations that have the smallest theoretical numbers of adjacent collisions, and used these 100 combinations on the blocks chosen from the *OED* to calculate the empirical numbers of adjacent collisions. Thus we counted the actual numbers of adjacent collisions for the 100 bit-allocations that had the best theoretical performance.

We observed that theoretical and experimental results are very close, as illustrated below.

The superimposition scheme

We conducted similar experiments using the superimposition signature scheme. We varied the combinations of the numbers of bits set for words in a phrase to see the relationship between the numbers of bits set and the number of adjacent collisions.

To compare with the concatenation scheme, we used the same conditions as used in the previous experiments. We used the same blocks of lexicographically consecutive phrases from the *OED*. Again, since the first words of all the phrases in each block are the same, we did not use the first words in the phrase signatures. Thus, again phrase signatures are structured from the $2nd$, $3rd$, $4th$, and $5th$ words, and the number of bits in word signatures and phrase signatures is 29 bits. The number of bits set in a signature of the ith word in a phrase is k_i. We used different combinations of (k_2, k_3, k_4, k_5) with k_i in the range of $[1, 16]$ to calculate the theoretical numbers of adjacent collisions using Lemma 16 (Section 3.4.2). Again we chose 100 combinations that have the smallest theoretical numbers of adjacent collisions, and used these 100 combinations on the blocks chosen from the *OED* to calculate the empirical numbers of adjacent collisions.

We observed that theoretical results are better than experimental results, but they are not far from experimental values in general. When the expected total number of bits set to 1 is between 12.5 and 16.5, the number of adjacent collisions is small, except when three words in a phrase have small numbers of bits set and one word has many bits set. When the expected total number of bits set is less than 10.5 or bigger than 18.5, the number of adjacent collisions increases substantially. This observation matches Faloutsos' optimal analyses; that is, the total number of bits set in a phrase signature should be close to half of the bits in a phrase signature.

Contrast

Tests on all blocks of data had similar results. Figure 3.13 and Table 3.1 show the relationship between numbers of adjacent collisions and percentages of combinations of bits allocated to words by both the concatenation scheme and the superimposition scheme for the block of phrases starting with the word "book". They are the results of the best 100 combinations of word signature sizes in terms of adjacent collisions for each scheme.

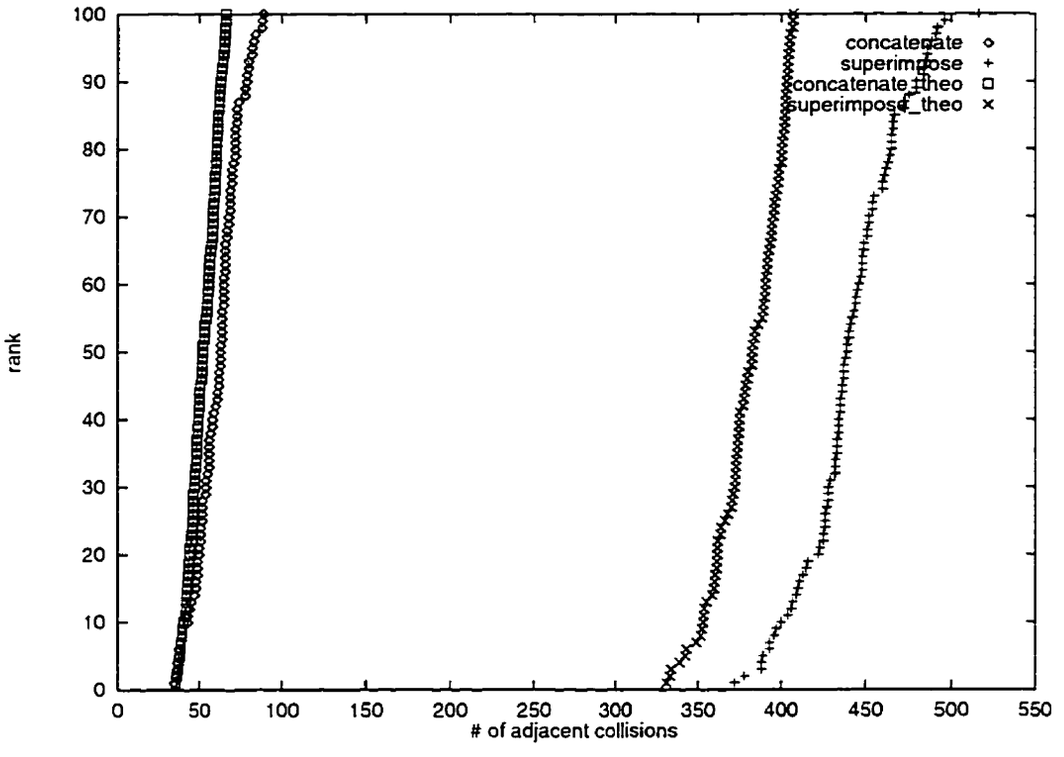


Figure 3.13: Concatenation versus superimposition

In Figure 3.13, the x-axis represents the number of adjacent collisions, and the y-axis indicates the theoretical rank for the combination of bits allocated (*i.e.*, point 30 represents the 30th best combination of word signature sizes as predicted by the theoretical model). There are four parts in Table 3.1, representing the

concatenation-theoretical							
rank	1	10	25	50	75	90	100
adj #	35.3	40.2	46.0	52.3	59.6	63.2	66.3
concatenation-empirical							
rank	1	10	25	50	75	90	100
adj #	34	43	52	63	70	79	89
superimposition-theoretical							
rank	1	10	25	50	75	90	100
adj #	331.1	353.2	366.1	382.4	398.1	403.3	407.1
superimposition-empirical							
rank	1	10	25	50	75	90	100
adj #	372	400	426	439	460	480	516

Table 3.1: Concatenation versus superimposition

theoretical and empirical results of the two schemes. In each part, the first row represents the rank of the combination of bits allocated or set, and the second row indicates the number of adjacent collisions. In our experiments, the total number of adjacent collisions is under 100 for every combination of word signature sizes using the concatenation scheme, and above 300 for the superimposition scheme. These indicate that the superimposition scheme has about 3 to 4 times as many adjacent collisions as the concatenation scheme.

Assume there are $10,000 (\approx 2^{13})$ indexed phrases in a block, 14 bits are needed for a pointer, 4.5 characters in a word on average, adjacent collision boundary phrases are 2.5 words on average because the first word is constant across the block

and it need not be stored, and one character requires 8 bits. In Table 3.1, there are 66 adjacent collisions in the 100th best combination for the concatenation scheme, which needs 66*(14+2.5*4.5*8+1.5*8)/10000 = 0.7656 bits per indexed phrase; and 331 adjacent collisions in the number 1 ranked combination for the superimposition scheme, which needs 331*(14+2.5*4.5*8+1.5*8)/10000 = 3.8396 bits per indexed phrase. Thus, adjacent collisions cost at least 3 more bits per indexed phrase in the superimposition scheme than in the concatenation scheme.

A phrase signature is used as a filter to limit disk accesses. Because the fewer the words, the fewer bits are set, it is more likely that the signature of a short prefix is contained by signatures of other phrases. Thus the superimposition scheme has the further disadvantage that the expected number of disk accesses might be even bigger to search for shorter, likely more common, prefixes than for longer ones.

3.4.3 Perfect hash functions for phrase signatures

In the next two subsections we briefly examine two other possible approaches.

A hash function h is *perfect* for a set of keys if for any x_i and x_j in the set, we have $h(x_i) = h(x_j)$ iff $i = j$. Assume that there are n words and m hash values. Witten, *et al.* give a trial and error algorithm to find a mapping function g such that a hash function $h(w) = g(h'(w)) +_n g(h''(w))$ is perfect, where $+_n$ means addition modulo n [WMB94]. It is required that $m > 2n$ in order to get such a perfect hash function h after a constant number of trials on average. They also give an algorithm for a perfect hash function $h(w) = g(h'(w)) +_n g(h''(w)) +_n g(h'''(w))$. It requires $m > 1.23n$ in order to get h. In either approach, the mapping function g occupies $m \log n$ bits.

Assume that we use the concatenation scheme for phrase signatures. We create

a perfect hash function h_i for the ith words of each $(i-1)$-word phrase. The concatenation of $h_1..h_i$ is then a perfect hash function for i-word phrases. For this approach, we must find and record a distinct function for every $(i-1)$-word prefix. This clearly needs a lot of space.

Alternatively function h_i can be chosen for all distinct ith words independently of the first $(i-1)$ words in the phrase. For this approach, n (the number of keys) would be quite large. Assume phrase signatures are based on the first k words. Let n_i be the number of distinct i-word phrases, and λ be the load-factor of function g_i. Then, the number of distinct ith words might be comparable to $(n_i - n_{i-1})$. So, $\frac{(n_i - n_{i-1})}{\lambda}$ hashing slots are needed in the mapping table g_i of h_i, and the total number of bits for mapping tables is about $\sum \frac{(n_i - n_{i-1})}{\lambda} \log_2(n_i - n_{i-1})$.

Assume there is a block of 10,000 phrases with $n_1, \ldots, n_5$ being 1000, 2000, 3000, 4000, 5000 respectively. Let a load factor λ be 1.5. Then, space for mapping tables is about $5 * 1000 * 10/1.5$ bits, which is about 4,200 bytes (3.3 bits per phrase). Since there are about $10*5$ bits for each phrase signature, the total is about 53 bits per phrase. As will be seen in Chapter 4, this expected size of 53 bits per phrase is significantly more space than is needed for the approach we propose.

3.4.4 Compressed text as phrase signatures

In text compression, one or more characters are represented by a short code. As a result, a text with n Latin characters is represented by an encoding that is shorter than n bytes. Because compression is typically used for text streams, text prediction models are used for coding [BCW90].

Table 3.2 summarizes compression measures used to estimate the entropy of

ordinary English [BCW90]. Shannon's results are that using 0-gram model [1], ordinary English for 26 characters could be compressed to about 4.70 bits per character: decreasing to 4.14, 3.56, 3.3 bits per character for 1, 2, 3-gram respectively; and 2.62 bits per character for the word model, or 11.79 bits per word assuming 4.5 characters per word. Evidence suggests that compression better than one bit per character is not likely to be achieved [WMB94].

size of	letter models with gram:								word	
alphabet	0	1	2	3	4	8	12	> 100	model	source
from statistical analysis of text										
26	4.70	4.14	3.56	3.3					2.62	Shannon(1951)
26+1	4.75	4.03	3.32	3.1					2.14	
26	4.70	4.12							1.65	Barnard(1955)
26+1	4.75	4.09	3.23	2.85	2.66	2.43	2.40			Newman and Waugh(1960)
from experiments with subjects' best guesses										
26+1	4.75									
upper bound		4.0	3.4	3.0	2.6	2.1	1.9	1.3		Shannon(1951)
lower bound		3.2	2.5	2.1	1.8	1.2	1.1	0.6		
26+1	4.75				2.2	1.8	1.8	1.7		Jamison and Jamison(1968)
from experiments with subjects using gambling										
26+1	4.75							1.25		Cover and King(1978)

Table 3.2: Estimates of the storage requirement of ordinary English (bits/char)

[1] A n-gram model uses statistical information of n characters.

Let us consider a very different signature scheme in which we apply compression algorithms to a phrase, and then take the first 32 bits of the compressed code as the signature of the phrase.

It is generally accepted that an English word is about 4.5 characters in length on average. Considering the space character, a word is generally then about 5.5 characters. At present, compression techniques reach about 2 bits per characters on average, and a phrase of 5 words (and 4 spaces) uses about $45 - 53$ bits, in other words, a 32 bit signature contains about $2.9 - 3.6$ words. If a phrase could be compressed to one bit per character, a 5-word phrase, on average uses 22.5 to 26.5 bits for phrase signature. However, even compression to 2 bits per character is achieved only under a good probability prediction, which needs a very good world model that requires a lot of extra space.

An alternative is to compress words instead of phrases. Word compression can then produce different lengths. We can keep a certain number of bits from each word signature, and then concatenate them to form a phrase signature. With this approach, 32 bits can then contain some informations for 5 words, for example.

If we maintain a prediction model of some kind, we would in principle get a good compression code for a phrase, but the model itself would occupy a large amount of space.

Bell, *et al.* report that for a 0-gram character model, we need about 4.7 bits per character, which is about 106 bits per phrase, with no model space cost [BCW90]. A 4-gram model with 26 letters, uses 2.2 bits per character, needs at least 26^4 nodes and gives about 50 bits per phrase of 5 words. A word model uses 11.79 bits per word or 59 bits per phrase of 5 words, but we need to maintain a word probability list.

In conclusion, using compression techniques to form a signature for a phrase is conceptually simple. Such a phrase signature contains any prefixes of a phrase up to the cutoff point for the phrase signature; phrase signatures are order-preserving if an order-preserving coding is used, so binary searches could be performed; and there are no collisions if compressions are lossless. Like perfect hashing, the disadvantage is that such an approach uses much more space.

3.5 Conclusion

In this chapter, we investigated the concatenation phrase signature scheme. We studied strategies to balance parameters, such as the number of bits in a phrase signature, the number of bits in each word, the size of a block, and the number of words in a phrase signature. We also compared the concatenation scheme and the superimposition scheme for phrase signatures, and explored the possibilities of using perfect hashing technique or a compression technique as the basis for the suffix-signature method.

We choose the concatenation scheme for phrase signatures. This scheme is based on simple hashing, and, unlike modifications based on perfect hashing or compression, it is not overly space intensive.

A superimposition scheme is an alternative, but there will likely be more collisions, because a phrase signature can contain a given word signature even though none of its words have that word signature. We also notice that signature arrays based on the concatenation scheme are more easily compressed using standard run-length techniques than those based on the superimposition scheme, since ordered phrases result in repeated prefixes in the signatures.

Chapter 4

Searching With Signatures

We have explored signature functions for words in Chapter 2, and signature structures for phrases in Chapter 3. In this chapter, we develop an algorithm for searching using the signature array.

We first study searches for simple words; then, we extend the algorithm to phrases. We next study ways to improve search performance for the proposed method. We also discuss long phrase searches and range searches.

4.1 Data model

Chapter 2 was based on a model that assumes no words repeat. The model in this chapter is based on repeated words. This relaxation on words comes from the fact that individual words occur in several places in a text. The remainder of the thesis assumes this more realistic text model.

Model 2 (Practical word search data model) *There is an ordered set of words* W, *where* $W = \{w_i \mid w_i \leq w_j, 1 \leq i < j \leq n\}$ *based on lexicographic ordering of the*

words. Also there is a signature function Sig. For each word w_i in W, there is a corresponding value $Sig(w_i)$, such that if $w_i = w_j$, $Sig(w_i) = Sig(w_j)$. So, we have a set of pairs Q, where $Q = \{< i, Sig(w_i) > | \ 1 \leq i \leq n\}$.

This word search model is the same as the simple word search model except that two lexicographically adjacent words in set W may be equal. Like the simple word search model, words in set W are sorted, and the signature function Sig takes a word from W as input and generates an integer $Sig(w_i)$ as the signature for that word. When two words are the same, their signatures are same.

We continue to assume that operations on signature set Q are much cheaper than on word set W.

4.2 Search Based On Word Signatures

Since word signature functions that have no collisions either have no reduction in the address space, require a large model, or are very expensive to support updates, word signature functions that are used in the suffix-signature method yield collisions. In this section we investigate collision handling issues in more depth. Next, we give a search algorithm, which is extended to phrase searching in Section 4.3.

4.2.1 Collision handling

Since we assume that operations on signature set Q are much cheaper than on word set W, we do most searching on Q and do as little as possible on W.

The first step to search for a word w is to look at signature set Q and identify entries $< i, Sig(w_i) >$ such that $Sig(w_i)$ are equal to $Sig(w)$. In the worst case,

we compare word w with those w_i's picked in the first step. Because W is ordered, the search of Q can be done in any order and unsuccessful searches can be stopped as soon as we find a range (j, k) for which $w_j < w < w_k$ and $Sig(w_i) \neq Sig(w)$ for $j < i < k$.

To improve performance, we choose not to search for words having signature Sig_w one by one from w_1 to w_n in set W. Instead we start in the middle of W and compare signatures in a widening range alternating towards w_1 and towards w_n. That is, we examine $w_{\frac{n}{2}}, w_{\frac{n}{2}+1}, w_{\frac{n}{2}-1}, \ldots$. Once a matched signature is found, we compare the corresponding word with word w. Then we can determine in which part of set W to continue the search, based on the ordering of W.

We know that a word in W might be repeated several times. Since W is ordered, repeated identical words are adjacent and have the same signature. Assuming for the moment that there are no adjacent collisions, all the identical signatures that are adjacent correspond to the same word. Thus having compared any word from a group of adjacent words having identical signatures to the target word, we can determine that all those words match w or none do. In the latter case, we can skip the whole set of adjacent words having signatures equal to $Sig(w)$.

However, as discussed in Chapters 2 and 3, there are almost unavoidably adjacent collisions: a block of adjacent identical signatures might correspond to different words. Thus to avoid comparing an identical word with w repeatedly, we store all the adjacent collision boundaries in a look-aside table (as illustrated in Figure 1.8). As a result, adjacent non-identical words having the same signature are divided into different sub-blocks. Within these sub-blocks there are therefore no adjacent collisions.

For each adjacent collision boundary, the look-aside table includes the lexico-

graphically larger word and its index. Words that appear in the look-aside table can be found without going to word set W. We may realistically assume that operations on the look-aside table are much cheaper than on word set W, since it is much smaller in general, as proven in Chapters 2 and 3. In addition to that, such boundary words describe limits that reduce search ranges, which therefore reduce search cost on word set W for other words as well.

4.2.2 Word search algorithm

A set of postings lists and a signature array are built over a text file *text*. The postings lists contain pointers to words in the *text*. In the postings list corresponding to a word, there are as many pointers as that word occurs in *text*. Following the strategy used in suffix arrays [GBYS92], the postings lists are concatenated in lexicographical order. Thus, in the resulting postings array, pointers are ordered lexicographically by the words they reference. The signature array contains signatures of corresponding words represented in the postings array.

Assume that *text*, the postings array, and the signature array are very large and are stored on disk. The postings array and the signature array are partitioned into blocks so that each block fits in memory.

Normally, a word falls inside a single block. But it might be possible that a word occurs so often that it spans several blocks, as exemplified in Figure 4.1. In this case, all middle blocks correspond to this particular word, and one block at each end might contain this word as well. In our example, the shaded area represents a word w, which occupies the last part of the second block, the whole third block, and the beginning of the fourth block. Notice that the shaded areas in blocks 2 and 4 are necessarily within the nearest adjacent collision points in both blocks.

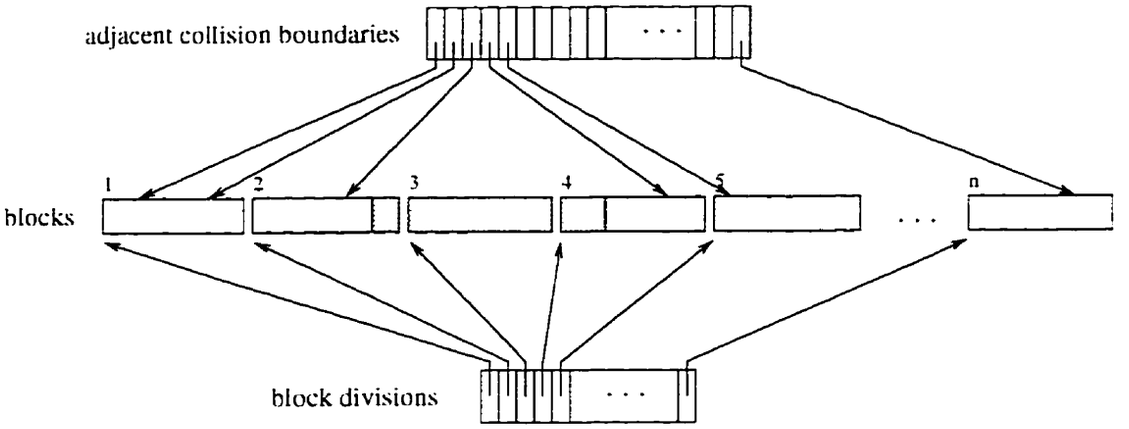


Figure 4.1: Blocks and look-aside table

We assume that a block list contains block division points (*i.e.* word values) that divide the postings array and the signature array into blocks. This list is assumed to be sufficiently small to fit in memory. We store the fragment of the look-aside table corresponding to each block together with the signature array and postings array fragment for that block. The look-aside table is used to store all adjacent collision boundary words and their indices in the postings array. Inside a block, binary search is used to find a word and identify its two end boundaries.

The text, the postings array, the signature array, and the block list are stored on disk. Before searching starts, the block list is loaded into main memory. Following is a word search algorithm to search for all occurrences of a word w, by using word signatures.

Problem: find all the occurrences in a text of a word w.

Input: a word w;

block list: strings;

postings array: integers;

signature array: integers:

look-aside table: (string, integer) pairs:

Output: the number of matches and the range in the postings array pointing to matching places in the text.

Algorithm 2 (Word Search)

1. *Calculate the signature sig_w for word w.*

2. *If w is not a block division point, all matches to w fall within one block. Read this block into memory. Determine the adjacent collision interval containing w by searching in the look-aside table.*

   - *If w is listed in the table as an adjacent collision boundary word, then the corresponding pointer in the postings array represents the smallest index i for which $w_i = w$. Adjacent pointers for $w_i, w_{i+1}, w_{i+2}, \ldots$ having the same signature and before the next adjacent collision boundary reference all occurrences of word w in text.*

   - *If w is not listed in the table, find the pair of adjacent words in the table surrounding word w, and get the corresponding range $[l_1, l_2]$ of the postings array.*

     (a) *Starting at position $\frac{l_1+l_2}{2}$, search up and down the signature array alternatively to find a match for sig_w. If no match is found, then word w does not occur in the text. Otherwise, determine the first encountered interval $[t_1, t_2]$ that matches sig_w.*

     (b) *Pick any position i in $[t_1, t_2]$. Get the ith pointer of the postings array and read a word w' from text on disk. Compare words w and w'.*

- *If they are identical, then pointers in the range $[t_1.t_2]$ on the postings array point to all occurrences of word w in text.*
- *If they are not identical, re-iterate from Step 2a using $[l_1.t_1 - 1]$ if $w < w'$ or $[t_2 + 1.l_2]$ if $w > w'$.*

3. *If w is listed as a block division point, word w appears in some adjacent blocks. Center blocks contain only word w. We read the two end blocks into memory.*

   - *In the higher end block, the beginning range that has the matching signature sig_w and is before the first adjacent collision boundary corresponds to word w.*
   - *In the lower end block, if the highest adjacent collision boundary word matches w, pointers from it to the end of the block correspond to word w. Otherwise, find the smallest position t which is greater than the highest adjacent collision boundary word and from which to the end of the block all positions have signature sig_w. Get the pointer on the postings array for any position i in that interval and read the corresponding word from text on disk and compare it with w. If they are identical, the range from t to the end of th block corresponds to word w.*

Note that the asymmetrical test of blocks in step 3 results from our convention that adjacent collisions always cause the lexicographically greater word to be stored in the look-aside table. In Step 2b, picking *any* position in an interval spares reading word w' from text on disk if word w' is coincidentally already in memory, which will potentially reduce cost.

4.2.3 Experimental results

Let us look at the search speed of all words in experiments described in Section 3.3.5. Although phrase signatures are stored, we are only concerned with the first word of each phrase. The experiments use different combinations of the block size, the number of words and the maximum number of bits in a phrase signature, and a bit allocation algorithm on the *Bible*. The graph on the left in Figure 4.2 plots the search performance for the block size 10k, assuming every distinct word is searched once. The graph on the right plots the number of bits per index point used for the signature array and the look-aside table. The x-axis represents the number of words in a phrase signature, which ranges from 3 to 7. The y-axis represents the expected number of disk accesses to a text on the left graph, and the number of total bits on the right.

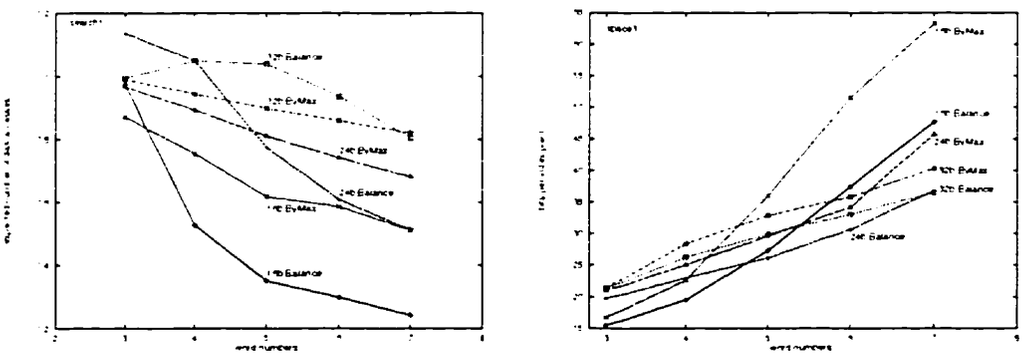


Figure 4.2: Speed vs. space

We observe that the expected numbers of disk accesses to the text are always less than 1.2. In general, the expected number of disk accesses decreases and the total space increases as the number of words increases. This is because the number of adjacent collisions increases as the number of words in a phrase signature in-

creases and the maximum number of bits in a phrase signature remains unchanged. Therefore, more space is required, and searches are performed in smaller ranges, which potentially reduces search costs. It also shows that when the number of words in a phrase signature is big enough, the expected number of disk accesses for the maximum allocation of 16 bits is smaller than 24 bits, which in turn is smaller than 32 bits. It again is due to the fact that there are more adjacent collisions for fewer bits than more bits. Thus, although sometimes more space is used, the higher likelihood of finding a word in the look-aside table and the smaller ranges for searching for other words, saves disk accesses on average. This is a tradeoff between space and time, and this motivates Section 4.4.1.

4.3 Phrase search algorithm

Having presented an algorithm to search for words by using word signatures, we now give an extended algorithm to search for phrases by using phrase signatures.

Again assume that *text* is a sequence of words. Phrases consist of subsequences of words from *text*. We create a suffix array and a signature array for phrases in the text, where the suffix array contains pointers to phrases in the text. Pointers in the suffix array are ordered by the lexicographical order of the repeated phrases. The signature array is a parallel array of signatures, one for each phrase in the text. The order of phrase signatures is the same as the order of the corresponding pointers to phrases.

The phrase signature is based on the first k words of the phrase, such that word signatures are concatenated to form a phrase signature. Thus, phrases having the same first l words have the same l-word prefix signatures. We again use a look-aside table to store all the adjacent collisions and a block list to store block boundaries.

An adjacent collision is said to be order i if the adjacent collision occurs at the ith words of the two adjacent phrases. Notice that two adjacent j-word phrases at an i-order adjacent collision point are not equal for $j = i$ and equal for $j < i$.

The text, the suffix array, the signature array, and the block list are stored on disk. Before searching starts, the block list is loaded into main memory. The following algorithm supports searches for all the phrases whose l-word prefixes match $p = w_1 w_2 \ldots w_l$, $l \leq k$.

Problem: find all the occurrences in a text of a phrase p of length up to k words.

Input: a phrase p containing at most k words;

block list: strings;

suffix array: integers;

signature array: integers;

look-aside table: (string, integer) pairs;

Output: the number of matches and the range in the suffix array pointing to matching places in the text.

Algorithm 3 (Phrase Search)

This algorithm outputs a flag and a range of the suffix array. The flag indicates if a search is successful or unsuccessful. The range includes all phrases whose l-word prefixes are p when the flag is "successful", and when the flag is "unsuccessful", the range is the last range during a search in which p falls.

1. *Create the signature sig_p for phrase $p = w_1 w_2 \ldots w_l$.*

2. *If p is not a prefix of a block division point, all matches to p fall within one block. Read this block into memory. Determine the adjacent collision interval containing p by searching in the look-aside table.*

   - *If p is listed in the table as a prefix of an adjacent collision boundary phrase p_i:*

     - *If the order of this adjacent collision is less than or equal to l, then the corresponding pointer in the suffix array represents the smallest index i for which $p_i = p$. Adjacent pointers for $p_i, p_{i+1}, p_{i+2}, \ldots$ that have signature sig_p, and before the next adjacent collision boundary of an order less than or equal to l, correspond to all occurrences of phrase p in text.*

     - *If the order of this adjacent collision is greater than l, adjacent pointers for $\ldots, p_{i-2}, p_{i-1}, p_i, p_{i+1}, p_{i+2}, \ldots$ that have signature sig_p, and are after the previous adjacent collision boundary of an order less than or equal to l but before the next adjacent collision boundary of an order less than or equal to l, correspond to all occurrences of phrase p in text.*

   - *If p is not listed in the table, find the corresponding range $[l_1, l_2]$ in which phrase p falls.*

     (a) Starting at position $\frac{l_1+l_2}{2}$, search up and down the signature array alternatively to find a match for sig_p. If no match is found, then phrase p does not occur in the text. Set the return flag to false, and return the range $[l_1, l_2]$. Otherwise, determine the first encountered interval $[t_1, t_2]$ that matches sig_p.

     (b) Pick any position i in $[t_1, t_2]$. Get the ith pointer in the suffix array

and read a phrase p' from text on disk. Compare phrases p and p'.

- *If they are identical. then pointers in the range $[t_1.t_2]$ of the suffix array point to all occurrences of phrase p in text. Set the return flag to true. and return the range $[t_1.t_2]$.*
- *If they are not identical. re-iterate from Step 2a using $[l_1.t_1-1]$ if $p < p'$ or $[t_2+1.l_2]$ if $p > p'$.*

3. *If p is listed as a prefix of a block division point. phrase p appears in some adjacent blocks. Center blocks all correspond to phrase p. As in Algorithm 2. we read the two end blocks into memory.*

   - *In the higher end block. the beginning range that has the matching signature sig_p and is before the first adjacent collision boundary of an order less than or equal to l corresponds to phrase p.*
   - *In the lower end block:*
     - *If p is a prefix of the highest adjacent collision boundary phrase. the last range that has the matching signature sig_p and is after the highest adjacent collision boundary of an order less than or equal to l corresponds to phrase p.*
     - *If p is not a prefix of the highest adjacent collision boundary phrase. as in Algorithm 2. find the smallest position t which is greater than the highest adjacent collision boundary phrase and from which to the end of the block all positions have signature sig_p. Get the pointer on the suffix array for any position i in that interval and read the corresponding phrase from text on disk and compare it with p. If they are identical. the range from t to the end of th block corresponds to phrase p.*

During searching, there are $O(\log x_b)$ comparisons within the block list, $O(\log x_l)$ comparisons within the look-aside table, and $O(x_s)$ comparisons in a search range, where x_b is the number of entries in a block list, x_l is the total number of adjacent collision phrases in a block, and x_s is the size of a search range of matching signatures.

4.3.1 Experimental results

Some further experiments were conducted on the text of the *Bible* (see Appendix A) to determine the effects of block size on performance.

We assume that all distinct existing i-word phrases have the same probability to be queried, and non-existing phrases are not queried in these experiments. A phrase signature has 32 bits and is based on the first 5 words. The strategy *Balance* is used to allocate bits among words in a phrase.

Figure 4.3 shows the expected number of disk accesses to the text to search for an i-word phrase with block sizes of 1k, 5k, 10k, 50k, and 100k. The x-axis represents the number of words in a phrase, and the y-axis represents the expected number of disk accesses to the text. Samples are represented by "$\diamond$", "$+$", "$\square$", "$\times$", and "$\triangle$" for the block sizes of 1k, 5k, 10k, 50k, and 100k respectively. Again, the average number of disk accesses to the text under all parameter settings is less than 1.2. We observed that searching for phrases of one or two words is noticeably faster on smaller blocks than bigger blocks.

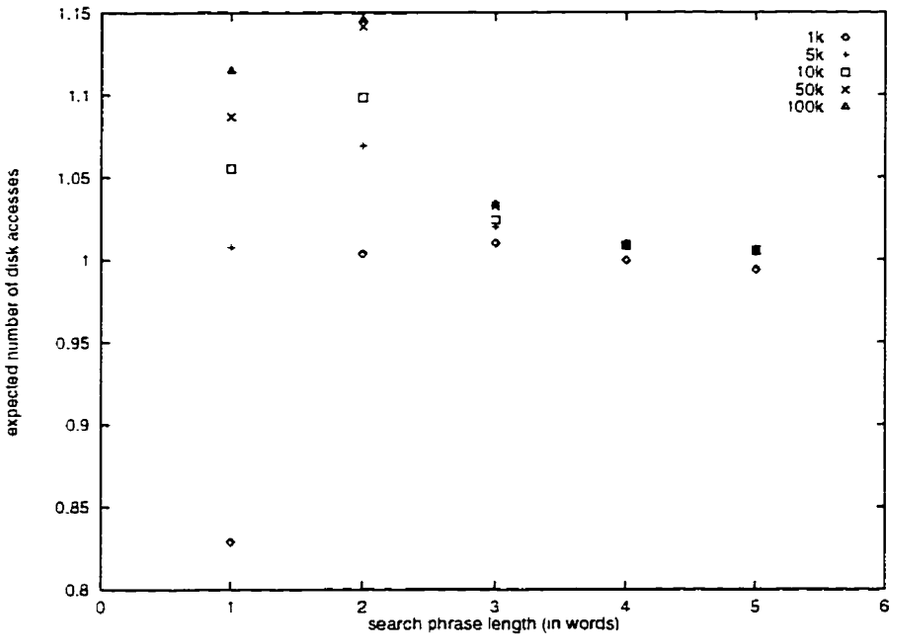


Figure 4.3: Search performances of different block sizes

4.4 Selecting phrases for look-aside tables

We can reduce the average search time by reducing the load factor for one or more of the component word signature spaces. There are several ways to reduce a load factor. We could increase bits allocated to the ith words by increasing the total number of bits in a phrase signature. Or, given a fixed number of bits in a phrase signature, we could give one more bit to the ith words by reducing one bit from the jth words. Or without increasing the number of bits to the ith words, a load factor can be decreased by reducing the number of words to be stored within a single extent of the signature array.

In this section, we show how to limit the number of disk accesses to a text. We will discuss how to control load factors while maintaining the same hash space size for the ith words of all $(i-1)$-word phrases, and how to guarantee a worst case for

search performance by putting some additional phrases in the look-aside table. We investigate the trade-off that arises between different parts of the look-aside table in terms of search speed and space.

4.4.1 Breaking points

The strategy *ByMax* or *Balance* described in Section 3.3.5 could be used to allocate bits in a phrase signature among the component words. If phrases are uniformly distributed, each i-word sub-space corresponding to different $(i-1)$-word prefixes would have similar load factors. But in reality, different words may have very different numbers of following words appearing in natural language phrases. For example, word w_6 in Figure 4.4 has many more distinct second words than words w_1 through w_5. Thus load factors for some i-word phrases may be much bigger than a desired load factor.

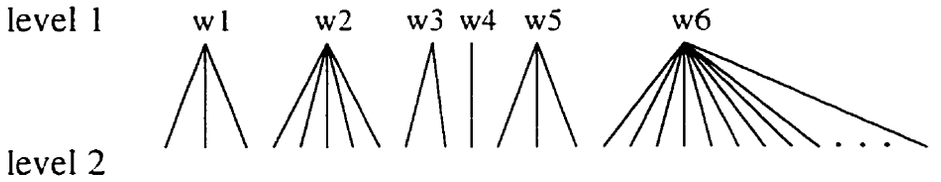


Figure 4.4: Far-outness of word distributions

Dividing the suffix array and signature array into blocks enables us to use different numbers of bits to represent the ith words of phrases in different blocks. Moreover, within each block, load factors can be further controlled while the ith words of phrases are represented by the same number of bits. The approach is to pick some phrases, to store them in the look-aside table, and to use them to narrow search ranges, like adjacent collision boundary phrases. This approach lowers load

factors for some phrases, and is motivated by the observation from Figure 4.2 that search performance improves when the number of adjacent collisions increases.

Control load factors

We explore the idea to lower load factors for some phrases if necessary by using an idea similar to extendible hashing [FNPS79].

Figures 4.5 and 4.6 illustrate extendible hashing. Items are hashed into a hash space with each hash value pointing to a bucket in which items are stored, as illustrated in Figure 4.5. Depths (in circles) indicate sizes of a hash space for each bucket. When a particular bucket is too full by some standard, the bucket will be split into two buckets. For example, the third bucket is split into two buckets, as illustrated in Figure 4.6. Items in the original bucket will be divided into the two new buckets. Depths of a hash space and buckets may change as insertions and deletions go on. Thus, each bucket could have a controllable load factor.

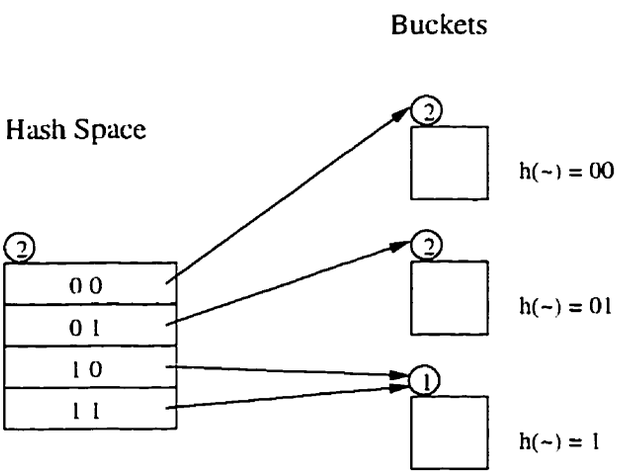


Figure 4.5: Before split (extendible hashing)

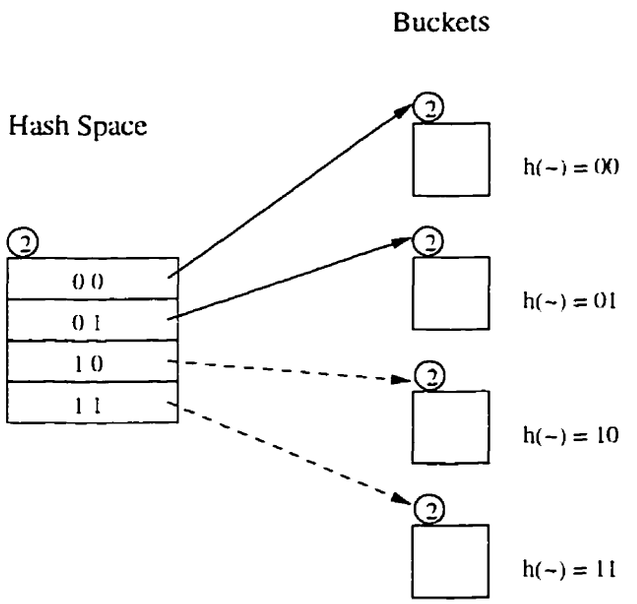


Figure 4.6: After split (extendible hashing)

Let us look back at our suffix-signature method. Assume there are x adjacent collisions that are of orders less than or equal to i. Then these x adjacent collision boundaries divide i-word phrases in a block into $(x + 1)$ buckets. Assume that each $(i - 1)$-word phrase has the same hash space size for following words. Then each bucket for the ith words has the same hash space size, but a different load factor. A bucket having more distinct i-word phrases has a higher load than a bucket having fewer distinct i-word phrases. If a bucket gets too full, an i-word phrase can be chosen to split the space into two parts, and the two new buckets will have lower loads than the original bucket. The chosen i-word phrase is kept in the look-aside table and serves as a separator of the two new buckets, which is similar to the idea in external hashing with limited internal storage [GL82, LK84]. Whereas extendible hashing determines which sub-bucket to use based on a hash, this approach chooses sub-buckets based on the keys themselves. Therefore, while

the same number of bits are used for the ith words of all phrases. controllable load factors are obtained by inserting additional breaking points in the look-aside table. Which bucket a given i-word phrase falls in depends on its lexicographic order in relation to all phrases in the look-aside table.

Strategies of inserting breaking points

When the number of distinct i-word phrases that have the same first $(i-1)$ words exceeds some threshold. phrases can be inserted into a look-aside table so that the new load factor is around some number p. We will call this *check0*. Figure 4.7 illustrates picking breaking points. There are 12 distinct words. or 19 words if repetitions are counted as well. So. either $w6$. the middle word of the 12 distinct words. or $w10$. the middle word of the 19 words. could be picked as a breaking point phrase. Although the size of hash spaces remains unchanged. load factors in the two new subspaces are expected to be half as big as the one in the original space since the number of elements in each new sublist is about half as many as in the original one.

Instead of using an overall load factor to determine when to split buckets. limiting the number of collisions in a single bucket could be used. as implied by extendible hashing [FNPS79]. Thus breaking points could be picked so that between two adjacent look-aside phrases no more than 2 or 3 distinct i-word phrases have the same signature. We will call this *check1*. Then. all phrase searches are guaranteed to be found within 2 or 3 disk accesses to a text. Figure 4.8 has a signature array for a list of distinct 2-word phrases. w and w' are 2 distinct first words having the same signature s. Signatures of their second words are s_i. s_1 occurs four times after w and three times after w'. Assume that we pick every third occurrence of the same signature as a breaking point. Then. we pick x_1 and x_2 as breaking points because

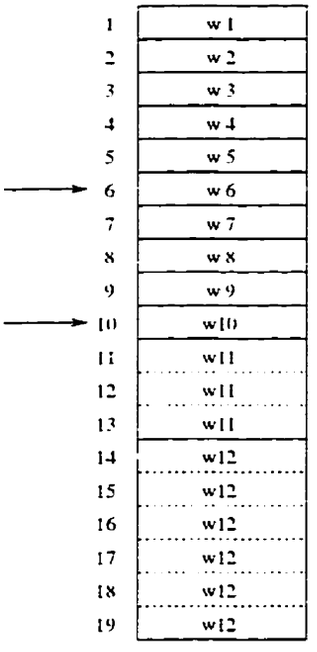


Figure 4.7: Choose breaking points to reduce load factor

the signature '$s\ s_1$' occurs the third time at x_1 from the point a and at x_2 after the point x_1. Although s_8 occurs four times, x_3 will not be a breaking point because s_8 occurs fewer than three times after x_2 is picked as a breaking point.

As a third alternative, breaking points could also be picked so that (for all values of $i \leq t$) between two adjacent look-aside phrases *any individual* $(i-1)$*-word phrase* has no more than 2 or 3 distinct following words having the same signature. This will be called *check2*. As a result, very few phrases need more than 2 disk accesses to a text. Note that unlike for *check1*, however, the same i-word signature can occur between breaking points more than 2 or 3 times in total because distinct $(i-1)$-word phrases might have the same signature. So, in Figure 4.8, positions y_1 and y_2 are picked as breaking points where the signature '$s\ s_1$' occurs the third time in the ranges of w and w', respectively. Although s_8 occurs three times in the

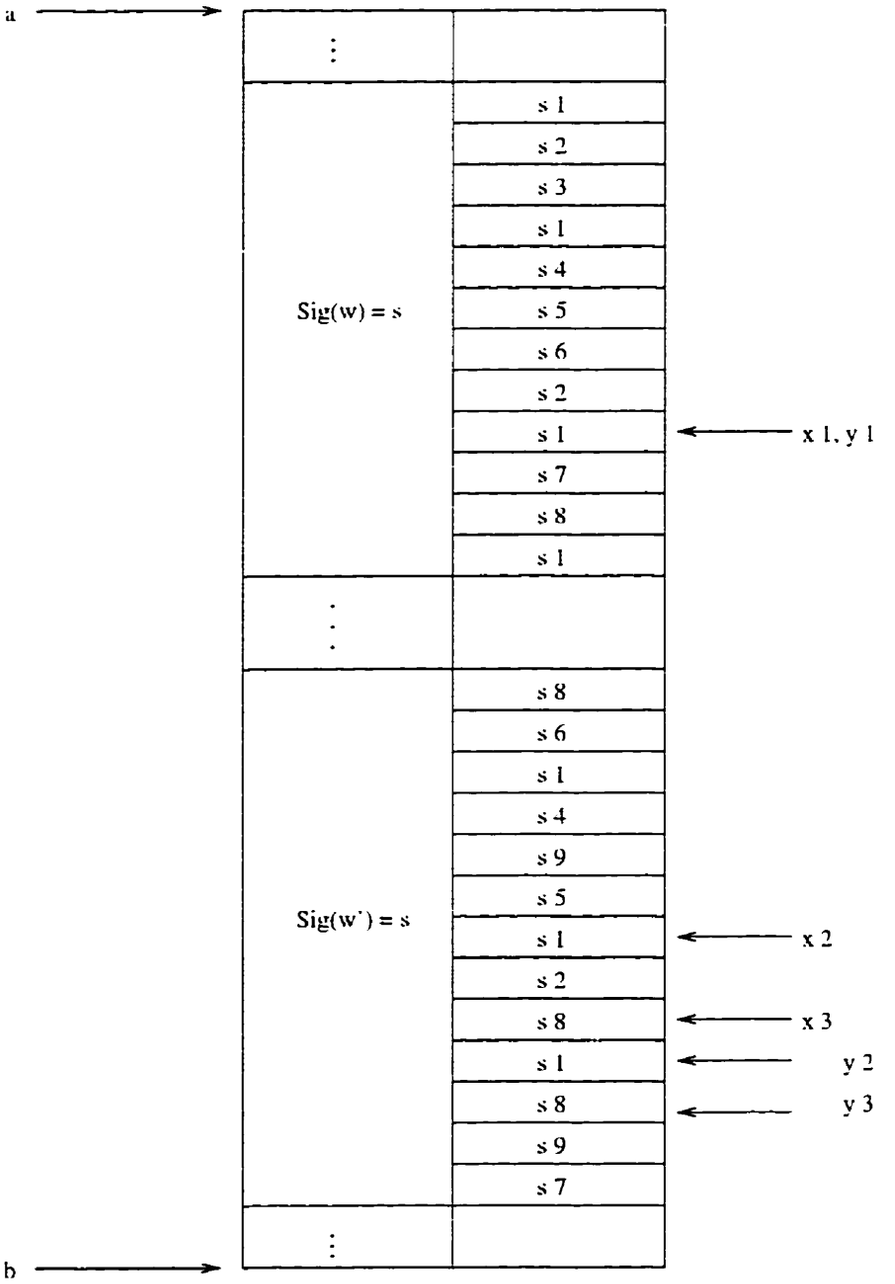


Figure 4.8: Choose breaking points to limit the length of a collision list

range of the word w'. y_3 will not be a breaking point because s_8 occurs fewer than three times after y_2 is picked as a breaking point.

Summary

Phrases that are picked to lower load factors are stored in the look-aside table and are used like adjacent collisions to narrow search ranges. Inserting phrases in the look-aside table essentially gives extra bits to those phrases while most other phrases use the assigned number of bits, which is smaller. Such an approach guarantees a controllable load factor, and in that sense it has the property of a variable size scheme. However, it does not introduce new complexities, but instead uses the look-aside table, which matches the style of our suffix-signature method.

4.4.2 Guaranteeing upper bounds on search time

In the proposed suffix-signature method, a look-aside table is introduced to store adjacent collision boundary phrases to solve the problem of adjacent collisions. As discussed in Section 4.4.1, supplementary breaking points can also be inserted to balance load factors for different i-word phrases. Since phrases in the look-aside table reduce the range of the array in which searches are performed, the result of using the look-aside table is improved search performance.

If most phrase searches can be solved by no more than β disk accesses, a β-disk access performance can be guaranteed by pre-loading the rest of the occurring phrases in another look-aside table. At indexing time, after a look-aside table is built to accommodate adjacent collisions and breaking points, we perform the following algorithm to build a second look-aside table L:

For $p \in$ text and $p \leq k$ words

if p is not found within β accesses using Algorithm 3,

then insert p into L.

Therefore, at search time, if a phrase p is not in L and using Algorithm 3 we have not found a match within β disk accesses, then p does not exist in the text. As a result, both successful and unsuccessful searches can be done within at most β disk accesses to a text. Step 2a in Algorithms 2 and 3 of Sections 4.2.2 and 4.3 is therefore repeated at most β times. Based on Figure 2.1 of Section 2.3, we choose $\beta = 2$. Since phrases in L need space to store, the number of guaranteeing phrases should be kept small. If it is found that too many phrases need to be stored, the value of β should be increased to reduce the number of guaranteeing phrases. Phrases in L are stored as *strings*, and binary search is used to search L.

When we search for a phrase using Algorithm 3, we start from the middle of a range and move up and down towards both endpoints of the block to find a matching signature. As in binary search, if the phrase corresponding to a matched signature is not the one being sought, we iterate in the appropriate subinterval. Unfortunately, this is *not* a binary search on matching signatures. So, if there are, for instance, three matching signatures in a range, it is not guaranteed that one of them is found at the first iteration, and two are found at the second iteration. In Figure 4.9, for example, there are three matching points at positions x, y, and z. x will be found at the first iteration, and the other two will be found at the second and third iteration, respectively.

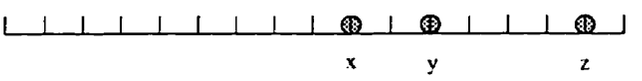


Figure 4.9: Non-binary

Why must L be kept separately? Paradoxically, reducing the size of a range may make some matching signatures require *more* iterations. For example in Figure 4.10, there are three matching signatures in the range $[A, B]$. Match point 2 could be found at the first iteration, and either point 1 or 3 would be found at the second iteration, depending on which lexicographical subinterval is chosen. If the range $[A, B]$ is reduced to $[A, B']$, match point 1 will be found at the first iteration, and the other two points will be found at the second and third iteration respectively. Thus, in this example, *reducing* the size of the interval *increases* the average and maximum numbers of probes for a successful search for this particular signature! Inserting a guaranteeing phrase into the first look-aside table may therefore negatively affect phrases already confirmed to take at most β accesses.

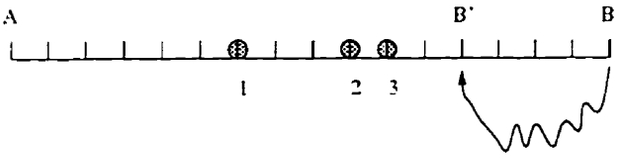


Figure 4.10: Reducing a range

So, guaranteeing phrases are not used for reducing search ranges. Instead, the look-aside table consists of two parts. The first part contains all the adjacent collision boundary phrases and breaking point phrases. They can be used to reduce search ranges. The second part contains "guaranteeing phrases" so that all phrase searches are guaranteed to require at most β disk accesses to a text. Adjacent collision boundary phrases make the suffix-signature method work. Breaking point phrases and guaranteeing phrases speed up phrase searches.

Generally, the more phrases used to reduce search ranges, the better search performance, and therefore the fewer guaranteeing phrases are needed. There is a trade-off between the two parts of a look-aside table, and we prefer more phrases

in the first part of the table than in the second part, given a fixed size of a look-aside table. Thus, optimizing a look-aside table overall could be done by inserting breaking point phrases into a look-aside table to reduce the number of guaranteeing phrases, and thus hopefully to reduce the size of the whole table.

Theorem 1 *Using the suffix-signature method with signatures stored for all k-word phrases, if the look-aside table fits in the same block as the corresponding segment of the suffix array and the signature array, then all searches of length up to k words can be found in at most* 3 *disk accesses.*

Proof: For a successful search, if the target phrase is not a block boundary phrase, it requires one disk access to get the corresponding index block and at most two disk accesses to the text. If the target phrase is a block boundary phrase, we must consider the blocks on both sides: at the higher end block, it requires one disk access to get the corresponding index block and no disk accesses to text (matching signatures at the end of the block are all for matching phrases); at the lower end block, it requires one disk access to get the corresponding index block and at most one disk access to the text to check if matching signatures at the end of the block match the target phrase. If the target phrase appears for more than one block boundary, the same arguments hold for the uppermost and lowermost blocks; every phrase in every intervening block must be a match, so there is no need to access either those index blocks nor the text. Therefore, the algorithm requires at most 3 disk accesses for a successful search.

For an unsuccessful search, the target phrase must fall within one block. So, it requires one disk access to get the corresponding index block and at most two disk accesses to the text, depending on how many matching signatures are encountered

during the search. By design, if a matching phrase is not found within two accesses to the text, there is no match.

Therefore, under all conditions, the algorithm requires at most 3 disk accesses to search for a phrase of length up to k words. □

If a look-aside table is too big to fit in the same block as the segment of the suffix-array and the signature array, we reduce the number of index points in the block to make the complete structure fit in one block. If as a result, there are too many blocks for the block list to fit in memory, the block list has to be stored in two levels. This requires one more disk access to get to the corresponding part of the block list. We discuss the two-level block list in Section 6.3. In the worst case, all index phrases in a block are adjacent collision phrases. Then the look-aside table becomes an inverted list on all k word phrases. We will discuss inverting all distinct k word phrases in Section 7.2.

4.4.3 Experimental results

Experiments have been done to evaluate several ways to insert breaking points, as described in Section 4.4.1. We choose $\beta = 2$, *i.e.* we wish to guarantee at most two disk accesses to the text for any search, which is motivated by Figure 2.1 of Section 2.3. We contrast the following approaches:

- *nobreak_noguar:* Neither breaking points nor guaranteeing phrases are inserted into the look-aside table.
- *nobreak:* No breaking points are inserted into the look-aside table, but guaranteeing phrases are inserted into the second look-aside table.

- *check0:* Breaking points are inserted when the load factor of i-word phrases that have the same first $(i-1)$ words exceeds 1. In our experiments, load factors after breaking points are inserted are limited to be at most 0.8. Guaranteeing phrases are inserted into the second look-aside table.

- *check1:* A breaking point is inserted when the third appearance of the same i-word signature for distinct i-word phrases is encountered (therefore avoiding the need to store *any* guaranteeing phrases).

- *check2:* A breaking point is inserted when the third appearance of the same i-word signature for distinct words following a single $(i-1)$-word phrase is encountered. Guaranteeing phrases are inserted into the second look-aside table.

Again we assume that all distinct existing i-word phrases have the same probability to be queried, and non-existing phrases are not queried in these experiments. A phrase signature has 32 bits and is based on the first 5 words. The strategy *Balance* is used to allocate bits among words in a phrase.

The following figures present experimental results for each of the different ways of inserting breaking points for the texts of *Bible* and *News* (see Appendix A). Samples are represented by "$\diamond$", "$+$", "$\square$", "$\times$", and "$\triangle$" for *nobreak_noguar*, *nobreak*, *check0*, *check1*, and *check2*, respectively.

Figures 4.11, 4.12, and 4.13 display the expected number of disk accesses to the text to search for an i-word phrase on the text of 5.6Mb *Bible* for block sizes of 1k, 5k, and 10k indexed phrases respectively. Figure 4.14 shows the search performance on the text of 85Mb *News* (for block size 10k). The x-axis represents the number of words in a phrase, and the y-axis represents the expected number of disk accesses to the text.

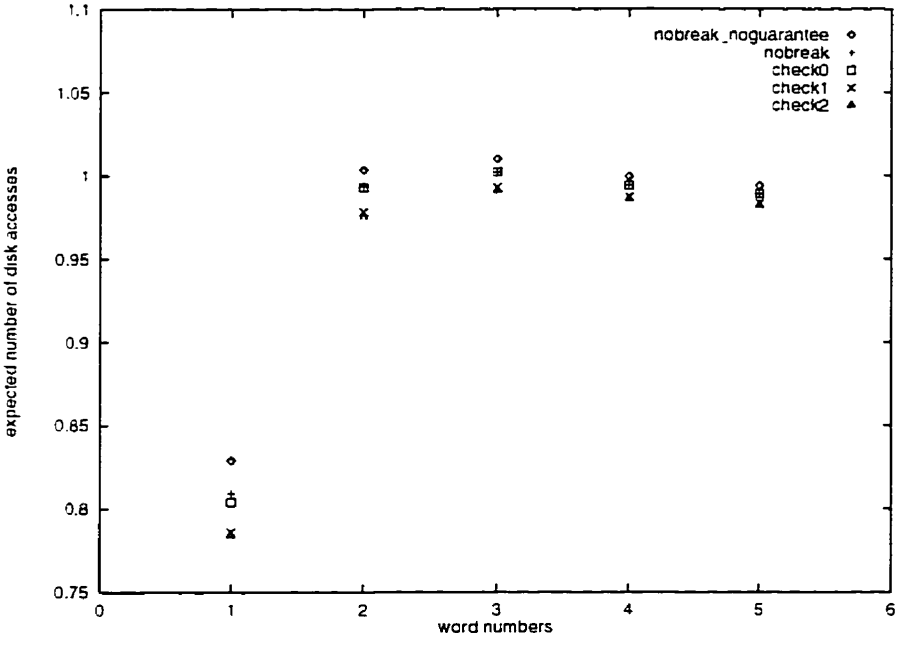


Figure 4.11: Search performance (*Bible*/1k)

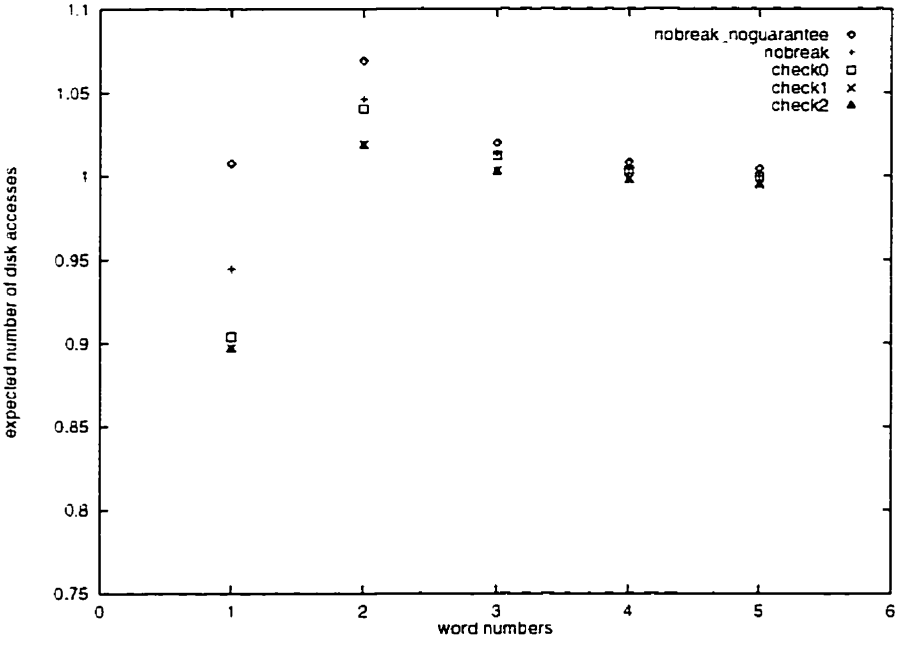


Figure 4.12: Search performance (*Bible*/5k)

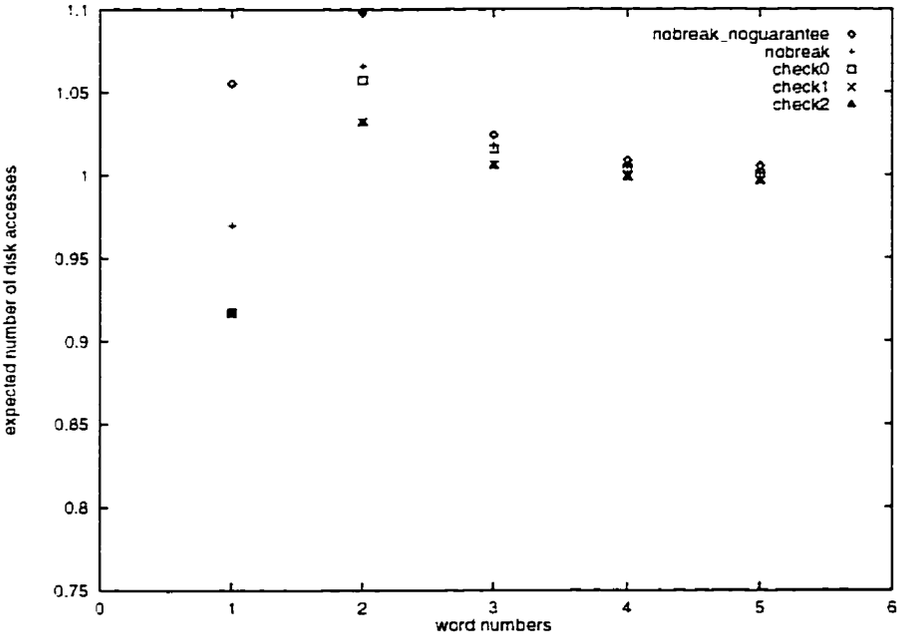


Figure 4.13: Search performance (*Bible*/10k)

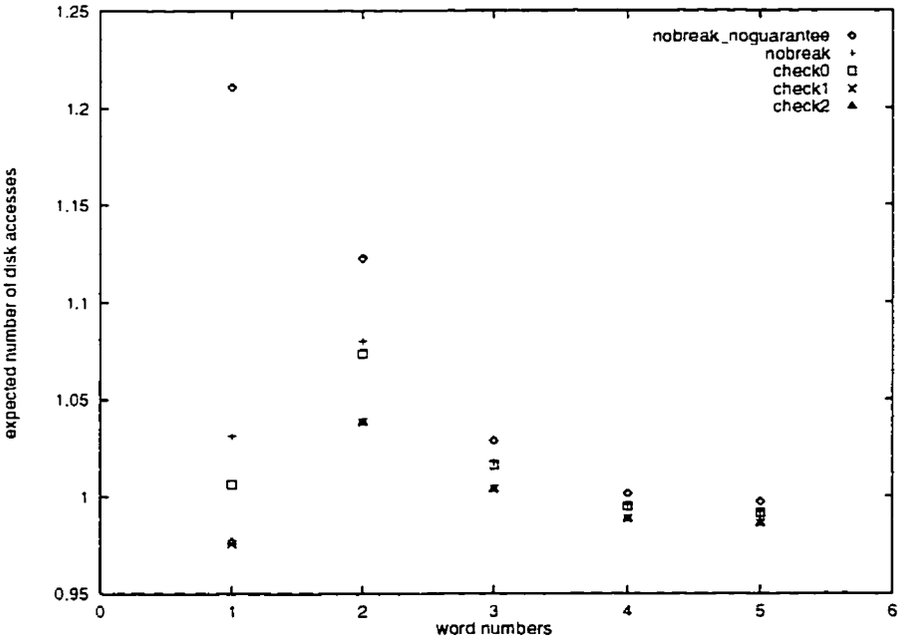


Figure 4.14: Search performance (*News*/10*k*)

Figure 4.15 is the total space used by the suffix-signature method for the *News* and the *Bible* for block sizes of 1k, 5k, and 10k respectively. The y-axis represents the number of bits per index point on average for storing the signature array, adjacent collision boundaries, block boundaries, and/or breaking points and guaranteeing phrases.

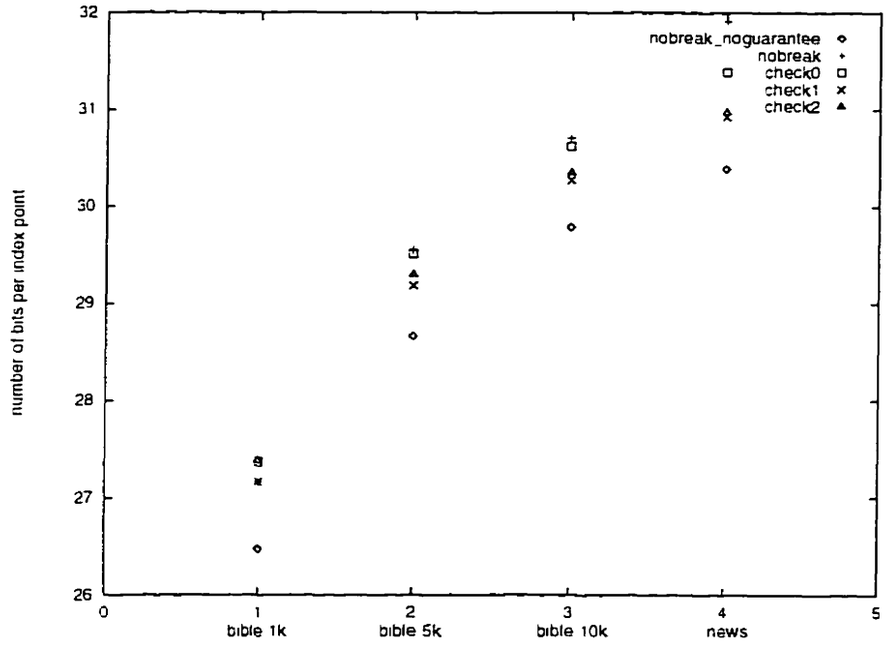


Figure 4.15: Total space

Observations

Guaranteeing phrases improve search performance at the cost of space. In general, all three ways of inserting breaking points (*check0*, *check1*, and *check2*) improve search performance and use less space than *nobreak*, which puts guaranteeing phrases in the look-aside table but does not insert breaking points. Shorter phrases get more search improvement than longer phrases. The results of *check1* and *check2*

are close, and they have the most improvement and use the least space. Since *check1* requires no guaranteeing phrases, it gives a faster index creation due to no need to check if any phrase is a guaranteeing phrase. For many blocks *check2* requires significantly fewer breaking points. Therefore, we suggest using either *check1* or *check2*.

Smaller blocks are better than bigger ones in terms of search performance and space because they are more adaptive for allocating bits among words, but a larger block list must be kept in memory.

4.5 Long phrase search

From Chapter 3, we know that a phrase signature does not contain all the information of a phrase and does not always contain enough information to distinguish it from other phrases, because a phrase signature normally is based on the first k words. Since a phrase signature contains no information about words after the kth word, phrases that share the first k words cannot be distinguished from each other by just checking their signatures. So, we need to handle searches for phrases of longer than k words differently.

Again, the text, the suffix array, the signature array, and the block list are stored on disk. Before searching starts, the block list is loaded into main memory.

Problem: find all the occurrences in a text of a phrase p containing more than k words.

Input: a phrase p containing $l > k$ words;

block list: strings;

suffix array: integers:

signature array: integers:

look-aside table: (string, integer) pairs:

Output: the number of matches and the range in the suffix array pointing to matching places in the text.

Algorithm 4 (Long Phrase Search)

1. *Use Algorithm 3 (Phrase Search) to search for phrases containing the first k words of s.*

2. *If the result set S_1 contains a few phrases, we may simply check phrases in S_1 directly. Otherwise, go to the next step.*

3. *Use Algorithm 3 again to search for phrases containing the next k words of s. The result set is S_2.*

4. *Find phrases in S_1 that are followed by a phrase in S_2 by comparing their pointers. Assume that $|S_1| = l_1$ and $|S_2| = l_2$.*

   - *If one of the sets is very small, say S_2, we sequentially go through S_1 and for each phrase pointer in S_1 we check S_2. Time is $O(l_1 l_2)$.*
   - *Otherwise sort S_1 and S_2. Then sequentially go through S_1 and S_2 to compare positions. Time is $O(l_1 \log l_1 + l_2 \log l_2)$.*

5. *Let the result set of the last step be S_1, and go to Step 2.*

Let l be the length of the target phrase (the number of words) and k be the number of words in a phrase signature. The above algorithm divides a long phrase

into $\lceil \frac{l}{k} \rceil$ k-word fragments (the last one might have fewer words), then combines search results of these fragments. The total number of disk accesses is $2\lceil \frac{l}{k} \rceil$ on average.

If a k-word fragment does not appear in the block list, it can be found within at most 3 disk accesses. If it appears $x > 0$ times in the block list, as in Figure 4.16, the number of disk accesses to get all the pointers from a suffix array is at most $x + 2$ (x for blocks 1 to x and two for block 0).

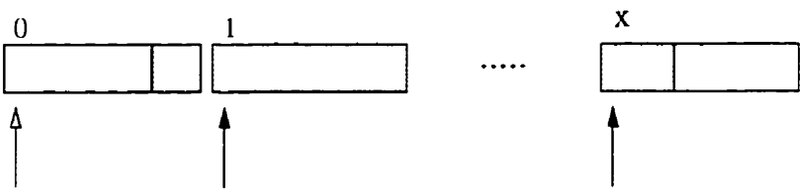


Figure 4.16: A phrase covers more than two blocks

Let t be the number of k-word fragments that are not listed in the block list. Then these t k-word fragments need at most $3t$ disk accesses. Let b be the total number of times that k-word fragments are listed in the block list. Then these $(\lceil \frac{l}{k} \rceil - t)$ k-word fragments need at most $2(\lceil \frac{l}{k} \rceil - t) + b$ disk accesses. In the worst case, the total number of disk accesses is $2\lceil \frac{l}{k} \rceil + t + b$.

4.6 Range searches

There are some issues that need to be further addressed.

- Interval searches: How can the structure be used to support a search for all phrases that are lexicographically between two given phrases?

- Incomplete phrase searches: How can the structure be used to support a search for all phrases that share a common prefix ending in the middle of a word?

We will use Algorithm 3 (Phrase Search) of Section 4.3 to get some intermediate results and then use a binary search on a result set of Algorithm 3. We assume that the binary search will take a suffix array range and a phrase s, and return "successful" with a range corresponding to s, or return "unsuccessful" with a pointer to the phrase in the text that is lexicographically greatest among all those smaller than s.

Interval search

An interval search finds all the phrases that are lexicographically between two given phrases p_1 and p_2. If both p_1 and p_2 exist, the phrases whose index positions are between p_1 and p_2 in the suffix array are lexicographically between p_1 and p_2. But if either or both of p_1 and p_2 do not exist in a text, we need to find the smallest phrase in a text that is bigger than p_1, and/or the biggest phrase in a text that is smaller than p_2.

Again, the text, the suffix array, the signature array, and the block list are stored on disk, and the block list is loaded into main memory before searching starts.

Problem: find all the phrases in a text that are lexicographically between two phrases p_1 and p_2.

Input: two phrases p_1 and p_2;

block list; strings;

suffix array: integers:

signature array: integers:

look-aside table: (string, integer) pairs:

Output: the number of phrases in a text that are lexicographically between p_1 and p_2 and the range in the suffix array pointing to matching places in the text.

Algorithm 5 (Interval Search)

1. *Use Algorithm 3 (Phrase Search) to search for phrases whose prefixes are* p_1. *returning a flag and a range.*

2. *If the flag is "successful", let the index of the first phrase in the range be* r_1. *If the flag is "unsuccessful", use a binary search in the range for* p_1. *which returns an index* x *that references the greatest phrase smaller than* p_1. *and let* $x + 1$ *be* r_1.

3. *Use Algorithm 3 to search for phrases whose prefixes are* p_2. *returning a flag and a range.*

4. *If the flag is "successful", let the index of the last phrase in the range be* r_2. *If the flag is "unsuccessful", use a binary search in the range for* p_2. *which returns an index* x *that references the greatest phrase smaller than* p_2. *and let* x *be* r_2.

5. *If* $r_1 > r_2$. *there is no phrase between* p_1 *and* p_2. *Otherwise, phrases from* r_1 *to* r_2 *are between* p_1 *and* p_2.

Incomplete phrase search

Assume that we search for phrases with first word "the", and a prefix of the second word is "surg". That is, in the target phrase p ="the surg", the last word w ="surg" is an incomplete word. So "the surge", "the surgeon", "the surgery", and "the surgical" are all eligible. We call this search an *incomplete phrase search*. Unfortunately if w is a proper prefix of the last word of a matching phrase, the signature of p and the signature of a matching phrase may not be the same.

We will search for phrases that contain all the words in a target phrase p except the last word. Then in the matching range, we resort to a binary search for p.

Again, the text, the suffix array, the signature array, and the block list are stored on disk, and the block list is loaded into main memory before searching starts.

Problem: find all the occurrences in a text of a phrase p with an incomplete last word.

Input: a phrase p with an incomplete last word:

block list: strings:

suffix array: integers:

signature array: integers:

look-aside table: (string, integer) pairs:

Output: the number of matches and the range in the suffix array pointing to matching places in the text.

Algorithm 6 (Incomplete Phrase Search)

1. *Use Algorithm 3 (Phrase Search) to search for phrases whose prefixes are p except the last word, and get a flag and a range.*

2. *If the flag is "unsuccessful", there is no matching phrase. If the flag is "successful", use a binary search in the range for p. This returns "successful" or "unsuccessful", which is the value to return for the phrase search.*

4.7 Summary

In this chapter, we developed an algorithm to search for words using word signatures, extended it to phrases using phrase signatures, and discussed long phrase searches and range searches.

We also discussed inserting breaking points and using a second look-aside table to improve search performance for the proposed suffix-signature method. For a non-variable signature size scheme, inserting breaking points makes load factors controllable. The performance of at most β-disk accesses to a text is guaranteed by putting guaranteeing phrases in a second look-aside table. This reduces expected and worst-case search time without too much extra space being required.

Chapter 5

Experimental Validation

We have proposed a suffix-signature method for fast phrase searching on a large static text. In this chapter we describe the implementation of our prototype system and extensive experiments conducted to test the suffix-signature method based on the concatenation of word signatures of the first k words in a phrase. The goal of the experiments is to validate the suffix-signature method with respect to both space and searching speed.

The searching speed is measured in terms of numbers of disk accesses to search for a phrase. The extra space that the method uses, in addition to a suffix array and a list of blocks for the suffix array, includes the signature array and a look-aside table.

5.1 Prototype system

Figure 5.1 illustrates the hierarchy of indices. The *suffix array* and the *signature array* are divided into fixed sized blocks. Block addresses and corresponding bound-

ary phrases are kept in a *block list*. The *text*, the *suffix array*, *signature array*, and look-aside tables are on disk. The *block list* is in memory. During searching, a particular block of the *suffix array* and the *signature array*, including the corresponding part of the look-aside table, is loaded into memory using one read operation.

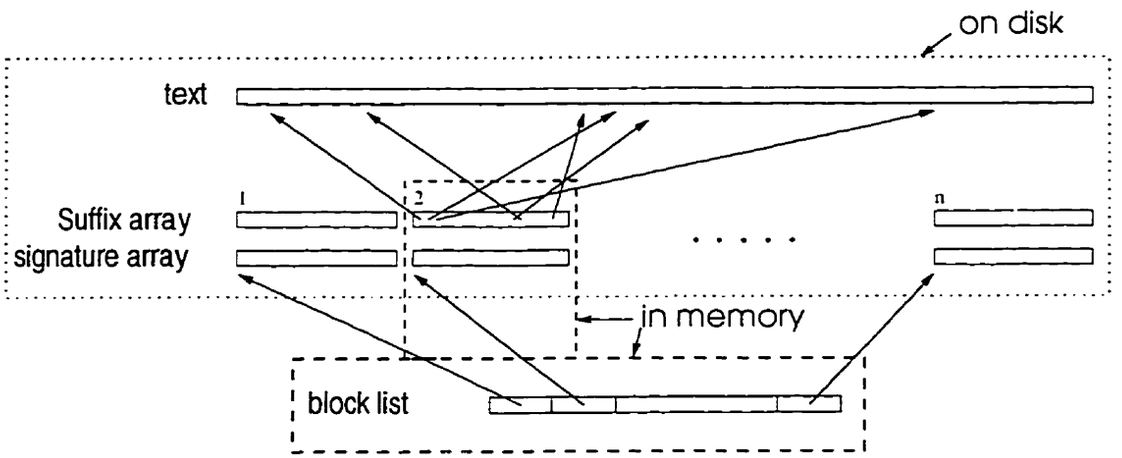


Figure 5.1: Hierarchy of Indices

The prototype system supporting the suffix-signature method has been implemented in C, with about 6000 lines of code. It includes two parts: *buildindex* and *searchphrase*.

Buildindex builds a block list, a signature array and look-aside tables based on a text and its suffix array. *Searchphrase* provides only a simple user interface. It takes a phrase search query from a user and prints the number of matches and a few matching samples.

Experiments with the prototype system were run on a shared *DEC alpha* 3000/50S machine, using the *Bible* and the *News* (see Appendix A) as data sources. The number of words in a phrase signature was chosen to be 5, and the number of phrases in a block was 10000. 32 bits were used for a suffix array pointer, and for simplicity, all phrase signatures, regardless of length, were stored in 32 bit fields.

About 800Kb of memory was required for data structures used by *buildindex*.

For *searchphrase*, the times were measured from the moment that a query was issued to the moment that a list of answers was ready. They do not include retrieving the final matching phrases from disk.

For the *Bible*, *buildindex* took an elapsed time of 10 minutes. The *Bible* is 5.6Mb and the suffix array is 4.5Mb. The block list, the signature array, and the look-aside tables use about 4.9Mb without any compression, and 3.0Mb after compressing the signature array (as described in Section 3.3.4). Selected phrases were searched against the *Bible* using *searchphrase*, which required about 110Kb of memory for its data structures. A typical phrase search required about $0.004 - 0.008$ seconds of cpu time and $0.05 - 0.14$ seconds of elapsed time. Some search examples are in Table 5.1.

	matches	cpu (sec)	elapsed
both	458	0.004	0.064
today	0	0.004	0.078
tomorrow	3	0.004	0.047
Egypt	658	0.004	0.064
and so	168	0.005	0.060
and there was	145	0.006	0.059
in the beginning	20	0.008	0.136
an east wind to	1	0.006	0.105

Table 5.1: Search examples in Bible

For the *News*, *buildindex* took an elapsed time of 155 minutes. The *News* is 84.7Mb and the suffix array is 61.3Mb. The block list, the signature array, and

the look-aside tables use about 66.0Mb without any compression, and 43.7Mb after compressing the signature array (as described in Section 3.3.4). Again selected phrases were searched against the *News* using *searchphrase*, which required about 170Kb of memory for its data structures. A typical phrase search required about $0.005 - 0.010$ seconds of cpu time and $0.05 - 0.15$ seconds of elapsed time. Some search examples are in Table 5.2.

	matches	cpu (sec)	elapsed
both	6281	0.009	0.132
today	7212	0.010	0.130
tomorrow	334	0.006	0.085
Egypt	388	0.005	0.061
and so	524	0.005	0.067
we have a	377	0.005	0.048
and there was	135	0.005	0.095
in the beginning	40	0.009	0.149
Canadian government has	23	0.005	0.098
an east wind to	0	0.006	0.081
where did he come from	0	0.004	0.064

Table 5.2: Search examples in News

5.2 Design of experiments

5.2.1 Experiment corpus

In order to verify that the suffix-signature method works well on real texts, experiments were conducted on a collection of different types of texts. Specifically, the texts include the *Bible*, *News*, the *OED*, and data from the *World Wide Web*, as described below and more fully in Appendix A.

Bible is about 5.6Mb and has approximately 1.1M indexed phrases.

News is a half year (Jan 1 1990 - July 31 1990) collection of articles from the *Ottawa Citizen* newspaper. It is roughly 84.7Mb and has about 15.3M indexed phrases.

The *OED* is the *Oxford English Dictionary, second edition*. It is approximately 545.6Mb and has around 1.5M distinct words. Its suffix array is roughly 439.3Mb and has about 109.8M indexed phrases.

WWW1, *WWW2*, and *WWW3* are collections of *HTML* pages from the *World Wide Web*. They are roughly 5.1Mb, 50.1Mb and 201.6Mb in size, and have about 0.83M, 7.8M and 34.4M indexed phrases, respectively.

5.2.2 Parameters

Lexicographical ordering is based on a case-insensitive indexing: most punctuation characters are mapped to blanks, and multiple consecutive blanks are treated equivalently to a single blank.

text	# bytes	# index points	avg word length
Bible	5.6M	1.1M	3.96 (char)
News	84.7M	15.3M	4.54 (char)
OED	545.6M	109.8M	3.97 (char)
WWW1	5.1M	0.83M	5.14 (char)
WWW2	50.1M	7.8M	5.42 (char)
WWW3	201.6M	34.4M	4.86 (char)

Figure 5.2: Experiment corpus

Block size

A block is a unit that is loaded into memory from disk and in which searches are performed. For the purposes of the experiments, the number of total indexed phrases in each block is fixed to be 10,000.

Number of words in a phrase signature

Phrase signatures are based on 5 words of text. This parameter is the same for all blocks in a text.

Phrase and word signature size

The number of bits for signatures of the ith words in a phrase is chosen independently for each individual block. Bits of a phrase signature are divided among words using the strategy *Balance*, as described in Section 3.3.5. *Balance* tries to minimize the total number of adjacent collisions in a block.

The number of bits in a phrase signature is at most 32. If the total number of bits for a phrase is more than 32 bits after the initial bit assignment, the number of bits is reduced proportionally among all words. For the i-word phrase having the biggest number of distinct following words (the $(i+1)$th words), the lower bound of load factors of the word signature space for the $(i+1)$th words is around 1. Thus, if there are not many distinct phrases in a block, the number of bits in a phrase signature will be fewer than 32 bits in this block.

Look-aside table

Strategy *check2* (Sections 4.4.3 and 4.4.1) is used to choose breaking points to lower load factors for some phrases. The value of β is chosen to be 2 (see Section 4.4.2). So, all the searches for phrases up to 5 words can be done by at most 2 disk accesses to a text, therefore by at most 3 disk accesses in total.

Word signature function

The word signature function used in our experiments is the composition of $f(w)$ and g, defined as follows. Function $f(w)$ is $\Sigma(w[j] * prime[j])$, the summation of the multiplication of the jth character $w[j]$ of a word and a prime number $prime[j]$. Function g divides the result of $f(w)$ into several parts of l bits and exclusive-ORs them together, where l is the number of bits in the word signature.

Simple compression

An i-word phrase may repeat several times, especially when i is 1 or 2. So, if signatures are stored word-wise, many word signatures repeat adjacently at lower

levels. Run-length encoding is used to compress word signatures at these lower levels.

Let x be the number of adjacent repetitions of a word signature s. When x is greater than a certain number, the x repetitions of the signature s are represented by a pair (s, x). For our purposes, the cutoff value for the run-length encoding is 4. An extra bit is needed for each word signature s to indicate if s is followed by a repetition number.

When a fragment of a signature array is read into memory, the phrase signatures are decompressed so that they can be processed more easily.

5.2.3 Performance metrics

By construction, searching for a phrase up to 5 words can be done by at most 2 disk accesses to a text. Since one disk access to index is needed for each search, the total number of disk accesses is at most 3.

Theoretical models have shown that this can be achieved with reasonable space. However we wish to see that space is also manageable in practice, *i.e.* that not too many phrases need to be stored in look-aside tables.

Thus, we will measure space usage for each text in our experiments. We will also measure the average number of disk accesses to a text (so with one access to the index we can know the total average number of disk accesses).

Space

We show the space used by a signature array, the block list, and a look-aside table, in terms of bits per indexed phrase in a text. The size of a phrase signature is

measured by the actual number of bits in it. The space is shown for both states of the signature array: before and after simple compression (see Section 5.2.2).

Speed

The experiments are done by using the proposed method with a binary search on a block of the suffix array. They are repeated for i-word phrases for $i = 1, 2, \ldots, 5$, where 5 is the number of words on which a phrase signature is based.

The experimental results are shown for all indexed phrases (to estimate time for successful searches) and for all phrase signatures (to estimate time for unsuccessful searches), as described in Section 5.2.4.

Search performance is in terms of the expected numbers of disk accesses to the text and the percentages of searches being satisfied by j disk accesses.

5.2.4 Query distributions

Experiments are all done for i-word phrases, for i between 1 and 5.

Successful searches

Searches for indexed phrases are successful searches. Three query distributions for the indexed phrases are used in the experiments.

- Each distinct i-word phrase is queried exactly once, which will be labeled as *uniform phrases*.
- An i-word phrase is queried as many times as it occurs in a text, which will be labeled as *proportional to occurrences*.

- An i-word phrase to be queried is selected according to the *DeFazio* distribution [DeF93], as follows:
  - For a given value of i, consider the i-word phrases in decreasing order of the numbers of occurrences. Calculate the cumulative numbers of occurrences from the top.
  - Those phrases accounting for the first 90% of occurrences are the "high" frequency phrases. Those in the next 5% are the "medium" frequency phrases. Those in the last 5% (which will typically be many, many phrases) are the "low" frequency phrases.
  - In choosing a phrase to be searched, select from each of the "high", "medium", and "low" categories with 1/3 probability, and select uniformly within each category.

Unsuccessful searches

Searches for unindexed phrases are unsuccessful searches. We do not actually search for unindexed phrases in most of our experiments. Instead, we assume every phrase *signature* is equally likely to be requested since an arbitrary phrase has an equal probability of having each signature.

Let the total number of distinct signatures of the indexed i-word phrases of a block be $N_{indexed}$, and the total number of possible signatures of i-word phrases be $N_{possible}$.

Let S be the set of signatures of all the indexed i-word phrases of a block. Partition S into $S_0 \cup S_1 \cup S_2 \ldots \cup S_t$ such that for a signature s in S_x, x is the maximum number of disk accesses to search for indexed i-word phrases having s as signature.

If the signature of an i-word phrase is not in S, no disk access to the text is needed to confirm that a phrase having that signature does not appear. If the signature of an i-word phrase is in S_x, then at most x disk accesses to a text are needed for this unindexed i-word phrase.

Let $|S_x|$ be the size of S_x. The expected percentage of searches without extra disk access to a text is $\frac{N_{possible}-N_{indexed}+|S_0|}{N_{possible}}$, and the expected percentage of searches with at most x disk accesses to a text is $\frac{|S_x|}{N_{possible}}$ for $x > 0$, which is the expected worst case performance of unsuccessful searches for i-word phrases and is labeled as *uniform i-word signatures* in the following tables.

Similarly, for each $(i-1)$-word prefix, we can get the expected worst case performance of unsuccessful searches for following words, assuming every word signature of the ith level is equally likely to be requested. This corresponds to a search for an i-word phrase for which only the ith word does not match the text. The average of the worst case performance over all $(i-1)$-word prefixes is labeled as *uniform word signatures*. The maximum of the worst case performance of all $(i-1)$-word prefixes is labeled as *max(uniform word signatures)*.

5.2.5 Experimental procedure

The suffix array and the signature array are created for a text. They are divided into blocks, where each block has a fixed number of indexed phrases.

The look-aside table contains three types of information: adjacent collision boundary phrases, breaking point phrases, and guaranteeing phrases.

Each i-word adjacent collision boundary is represented by the index position (an integer) and the first 5 words of the lexicographically greater colliding phrase. Each breaking point is represented by an index position and a 5-word phrase. Each

i-word guaranteeing phrase is represented by an index position together with the i-word phrase. For a breaking point or guaranteeing phrase, the smallest position of a repeating i-word phrase is stored.

Phrases are in lexicographical order in each part of the look-aside table.

Experiments are conducted through blocks one by one. All distinct i-word phrases in each block are searched using the suffix-signature method. Each block has its local result for i-word phrases, for $i = 1, 2, \ldots, 5$. A text has a global experimental result, obtained by aggregating the local results.

For the query distribution of querying each distinct i-word phrase exactly once, we calculate the percentage of the distinct i-word phrases that need x disk accesses as a fraction f'_x of the total number of all the distinct i-word phrases. Expected search time then is $\sum f'_x * x$.

For the query distribution of querying each i-word phrase as many times as it occurs, we calculate the percentage of i-word phrases with repetitions that need x disk accesses as a fraction f''_x of the total number of occurrences of all i-word phrases, and use $\sum f''_x * x$.

For the pth distinct i-word phrase in a block, we keep its signature, d_p (the number of disk accesses used to search for it), and r_p (the number of its repetitions). For the *DeFazio* distribution, we sort (d_p, r_p) pairs in descending order of r_p, and divide the ordered pairs in three adjacent groups so that the first group has 90% of total r_p's, and the 2nd and the 3rd 5% each. Then, we calculate f'''_x, the percentage of i-word phrases that need x disk accesses, which is the average of $\frac{\#\text{ of pairs in the 1st group such that } d_p=x}{\#\text{ of pairs in the 1st group}}$, $\frac{\#\text{ of pairs in the 2nd such that } d_p=x}{\#\text{ of pairs in the 2nd group}}$, and $\frac{\#\text{ of pairs in the 3rd such that } d_p=x}{\#\text{ of pairs in the 3rd group}}$. Expected search time then is $\sum f'''_x * x$.

We calculate the expected worst case performance of unsuccessful searches as

described in the subsection "Unsuccessful searches" of Section 5.2.4.

5.3 Experimental results

We briefly give experimental results in this section. A more detailed report of search performance is found in Appendix B.

The column labeled "i-words" in tables represents i-word phrases. Search performance is given in rows for the suffix-signature method and contrasting PAT search [GBYS92], averaged over all blocks in the text.

5.3.1 *Bible*

Table 5.3 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. The successful search performance is presented under the three different query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	0.92	1.03	1.01	1.00	1.00
proportional to occurrences	0.09	0.51	0.78	0.92	0.97
DeFazio	0.43	0.90	0.94	0.97	0.97
Suffix-Signature(unsuccessful)					
uniform i-word signatures	1.23	0.77	0.05	0.02	0.00
uniform word signatures	1.23	0.37	0.03	0.04	0.03
max(uniform word signatures)	2.00	2.00	1.094	1.03	0.84
PAT					
uniform phrases	16.56	15.21	14.75	14.55	14.48
proportional to occurrences	17.19	18.61	16.11	15.44	14.70
DeFazio	14.15	14.88	14.02	14.26	14.04

Table 5.3: Expected numbers of disk accesses to a text (*Bible*)

The average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 or 16 disk accesses to the text using binary

search when the size of a block is $10K$ as used in the experiment.

Table 5.4 summarizes the space usage. The number of bits in a phrase signature is 28.31 on average. The simple compression method described in Section 5.2.2 reduces this to 19.15. There are 1.45, 0.55, 0.02 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 30.36 bits which is compressed to 21.20 bits per indexed phrase.

	# of bits
phrase signature	28.31
phrase signature(compressed)	19.15
adjacent collisions	1.45
break points	0.55
guaranteeing phrases	0.02
block boundaries	0.03
total	30.36
total(compressed)	21.20

Table 5.4: Space (*Bible*)

5.3.2 *News*

Table 5.5 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. As before, the successful search performance is presented under the same three query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	0.98	1.04	1.00	0.99	0.99
proportional to occurrences	0.08	0.54	0.82	0.92	0.97
DeFazio	0.36	0.90	0.96	0.96	0.97
Suffix-Signature(unsuccessful)					
uniform i-word signatures	1.00	0.78	0.08	0.01	0.01
uniform word signatures	1.00	0.50	0.06	0.03	0.03
max(uniform word signatures)	2.00	1.86	1.51	1.84	1.81
PAT					
uniform phrases	15.72	15.03	14.58	14.36	14.32
proportional to occurrences	13.94	17.73	16.34	15.12	14.76
DeFazio	11.46	14.17	14.38	14.18	14.23

Table 5.5: Expected numbers of disk accesses to a text (*News*)

Again, the average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 or 15 disk accesses to the text

using binary search.

Table 5.6 summarizes the space usage as before. The number of bits in a phrase signature is 28.95 on average. The simple compression method described in Section 5.2.2 reduces this to 20.81. There are 1.41, 0.54, 0.03 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 30.96 bits which is compressed to 22.82 bits per indexed phrase.

	# of bits
phrase signature	28.95
phrase signature(compressed)	20.81
adjacent collisions	1.41
break points	0.54
guaranteeing phrases	0.03
block boundaries	0.03
total	30.96
total(compressed)	22.82

Table 5.6: Space (*News*)

5.3.3 *OED*

Table 5.7 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. As before, the successful search performance is presented under the same three query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	1.06	1.03	1.01	1.00	1.00
proportional to occurrences	0.08	0.35	0.61	0.80	0.94
DeFazio	0.29	0.65	0.82	0.91	0.97
Suffix-Signature(unsuccessful)					
uniform i-word signatures	0.63	0.53	0.12	0.04	0.03
uniform word signatures	0.63	0.43	0.11	0.06	0.06
max(uniform word signatures)	2.00	2.00	1.85	1.93	1.97
PAT					
uniform phrases	15.18	14.87	14.65	14.55	14.50
proportional to occurrences	8.28	14.60	16.44	16.67	16.62
DeFazio	6.39	11.50	13.73	14.46	14.75

Table 5.7: Expected numbers of disk accesses to a text (*OED*)

As for the previous experiments, the average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 to 15 disk

accesses to the text using binary search.

Table 5.8 summarizes the space usage. The number of bits in a phrase signature is 25.75 on average. The simple compression method described in Section 5.2.2 reduces this to 14.90. There are 0.87, 0.49, 0.02 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 27.16 bits which is compressed to 16.31 bits per indexed phrase.

	# of bits
phrase signature	25.75
phrase signature(compressed)	14.90
adjacent collisions	0.87
break points	0.49
guaranteeing phrases	0.02
block boundaries	0.03
total	27.16
total(compressed)	16.31

Table 5.8: Space (*OED*)

5.3.4 *WWW1*

Table 5.9 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. As before, the successful search performance is presented under the same three query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	1.04	1.02	1.00	0.99	0.99
proportional to occurrences	0.33	0.78	0.91	0.96	0.98
DeFazio	0.81	1.03	1.00	0.99	0.99
Suffix-Signature(unsuccessful)					
uniform i-word signatures	1.52	0.43	0.01	0.00	0.00
uniform word signatures	1.52	0.23	0.01	0.03	0.04
max(uniform word signatures)	2.00	1.70	1.34	1.38	1.78
PAT					
uniform phrases	15.59	14.83	14.60	14.51	14.46
proportional to occurrences	18.28	17.94	16.37	15.84	15.57
DeFazio	14.20	14.71	14.46	14.41	14.39

Table 5.9: Expected numbers of disk accesses to a text (*WWW1*)

Again, the average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 or 15 disk accesses to the text

using binary search.

Table 5.10 summarizes the space usage as before. The number of bits in a phrase signature is 29.94 on average. The simple compression method described in Section 5.2.2 reduces this to 20.92. There are 1.55, 0.52, 0.01 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 32.05 bits which is compressed to 23.03 bits per indexed phrase.

	# of bits
phrase signature	29.94
phrase signature(compressed)	20.92
adjacent collisions	1.55
break points	0.52
guaranteeing phrases	0.01
block boundaries	0.03
total	32.05
total(compressed)	23.03

Table 5.10: Space (*WWW1*)

5.3.5 *WWW2*

Table 5.11 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. As before, the successful search performance is presented under the same three query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	1.05	1.03	1.00	0.99	0.99
proportional to occurrences	0.22	0.74	0.90	0.94	0.96
DeFazio	0.67	1.00	0.99	0.98	0.98
Suffix-Signature(unsuccessful)					
uniform i-word signatures	1.31	0.48	0.02	0.00	0.00
uniform word signatures	1.31	0.33	0.02	0.03	0.05
max(uniform word signatures)	2.00	1.87	1.52	2.00	1.94
PAT					
uniform phrases	15.49	14.89	14.63	14.47	14.20
proportional to occurrences	16.83	17.76	16.05	15.60	15.32
DeFazio	13.01	14.56	14.32	14.36	14.41

Table 5.11: Expected numbers of disk accesses to a text (*WWW2*)

Again, the average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 or 15 disk accesses to the text

using binary search.

Table 5.12 summarizes the space usage as before. The number of bits in a phrase signature is 29.62 on average. The simple compression method described in Section 5.2.2 reduces this to 20.72. There are 1.30, 0.50, 0.01 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 31.46 bits which is compressed to 22.56 bits per indexed phrase.

	# of bits
phrase signature	29.62
phrase signature(compressed)	20.72
adjacent collisions	1.30
break points	0.50
guaranteeing phrases	0.01
block boundaries	0.03
total	31.46
total(compressed)	22.56

Table 5.12: Space (*WWW2*)

5.3.6 *WWW3*

Table 5.13 summarizes the expected numbers of disk accesses to the text to search for an i-word phrase by using the proposed suffix-signature method and the pure PAT search. As before, the successful search performance is presented under the same three query distributions, and the unsuccessful search performance is measured by using the two unsuccessful models.

	1-word	2-words	3-words	4-words	5-words
Suffix-Signature					
uniform phrases	1.03	1.04	1.01	1.00	0.99
proportional to occurrences	0.08	0.58	0.87	0.96	0.99
DeFazio	0.40	0.97	1.00	0.99	0.99
Suffix-Signature(unsuccessful)					
uniform i-word signatures	1.02	0.76	0.06	0.00	0.00
uniform word signatures	1.02	0.57	0.04	0.02	0.04
max(uniform word signatures)	2.00	1.82	1.80	1.61	1.81
PAT					
uniform phrases	15.57	15.04	14.73	14.59	14.52
proportional to occurrences	13.38	18.89	16.77	15.80	15.48
DeFazio	10.64	14.92	14.47	14.40	14.40

Table 5.13: Expected numbers of disk accesses to a text (*WWW3*)

Again, the average search time for a phrase is about 1 disk access to the text using the suffix-signature method, and about 14 or 15 disk accesses to the text

using binary search.

Table 5.14 summarizes the space usage as before. The number of bits in a phrase signature is 29.48 on average. The simple compression method described in Section 5.2.2 reduces this to 19.44. There are 1.19, 0.49, 0.01 and 0.03 bits per phrase on average to store adjacent collisions, breaking phrases, guaranteeing phrases and block boundaries, respectively. Therefore, the total is about 31.20 bits which is compressed to 21.16 bits per indexed phrase.

	# of bits
phrase signature	29.48
phrase signature(compressed)	19.44
adjacent collisions	1.19
break points	0.49
guaranteeing phrases	0.01
block boundaries	0.03
total	31.20
total(compressed)	21.16

Table 5.14: Space (*WWW3*)

5.3.7 *WWWQ* on *WWW1*, *WWW2*, and *WWW3*

In order to further test our approach in practice, we perform one more sequence of experiments, this time using a real query distribution. *WWWQ*, as described in Appendix A.4.4, contains a trace of 2.7 million phrase search queries posed against the World Wide Web in the first two months of 1997. Query phrases in *WWWQ* are searched against the *WWW1*, *WWW2* and *WWW3* databases by using the suffix-signature method. Tables 5.15, 5.16 and 5.17 summarize the search performance for both successful searches and unsuccessful searches for the *WWW1*, *WWW2* and *WWW3*, respectively.

For successful searches, these tables list the number of successful i-word queries, the percentage of successful i-word queries over all successful queries, the average number of disk accesses to a text for a successful i-word query, and the average number of disk accesses to a text for a successful search over all lengths of successful searches. Similarly, they list the information for unsuccessful queries.

In the experiments, the average number of disk accesses to a text for a successful search is less than 1. The average number of disk accesses to a text for an unsuccessful search is less than 0.2, which is significantly less than we had expected from the earlier experiments.

5.4 Summary

This chapter presented the implementation of our prototype system, experiments and results of the proposed suffix-signature method on real world data.

The required number of disk accesses to a text to search for an i-word phrase using only a suffix array is about 14 or 15. The suffix-signature method reduces this to about 1 for a variety of texts and query distributions. Unsuccessful searches are particularly fast in experiments using real query patterns obtained from the World Wide Web. In addition to accesses to a text, 1 disk access is needed to a suffix array block in the pure suffix array method, and 1 or 2 disk accesses are needed to a suffix array block and a signature block in the suffix-signature method.

Table 5.18 summarizes space used by a signature array and auxiliary tables in terms of bits per index point for the *Bible*, *News*, *OED*, *WWW1*, *WWW2*, and *WWW3*. Table 5.19 summarizes the total space usage, where percentages are based on sizes of original texts. Interestingly, for web data, the total space for the index is merely 15% more than the space for the text itself.

	successful searches			unsuccessful searches		
length	#	%	disk accesses	#	%	disk accesses
1	1095368	95.4222	0.860	761125	48.3449	0.325
2	47086	4.1019	0.911	503860	32.0040	0.038
3	4735	0.4125	0.982	192386	12.2199	0.009
4	607	0.0529	0.932	73967	4.6982	0.003
5	70	0.0061	0.986	25229	1.6025	0.001
6	34	0.0030	1.618	9843	0.6252	0.002
7	14	0.0012	2.500	4072	0.2586	0.004
8	0	0.0000	0.000	1833	0.1164	0.005
9	2	0.0002	2.000	861	0.0547	0.002
10	0	0.0000	0.000	429	0.0272	0.002
11	1	0.0001	2.000	246	0.0156	0.000
12	0	0.0000	0.000	154	0.0098	0.000
13	0	0.0000	0.000	85	0.0054	0.000
14	0	0.0000	0.000	53	0.0034	0.000
15	0	0.0000	0.000	54	0.0034	0.000
>15	0	0.0000	0.000	169	0.0107	0.018
	total	expected disk accesses		total	expected disk accesses	
	1147917	0.863		1574366	0.171	

Table 5.15: *WWWQ* searches on (*WWW1*)

	successful searches			unsuccessful searches		
length	#	%	disk accesses	#	%	disk accesses
1	1504638	89.7994	0.633	351855	33.6147	0.331
2	150217	8.9652	0.872	400729	38.2839	0.090
3	17416	1.0394	0.901	179705	17.1682	0.023
4	2452	0.1463	0.944	72122	6.8902	0.009
5	333	0.0199	1.000	24966	2.3851	0.004
6	399	0.0238	2.115	9478	0.9055	0.007
7	74	0.0044	1.797	4012	0.3833	0.013
8	19	0.0011	2.053	1814	0.1733	0.019
9	2	0.0001	2.500	861	0.0823	0.015
10	3	0.0002	2.333	426	0.0407	0.028
11	1	0.0001	4.000	246	0.0235	0.028
12	0	0.0000	0.000	154	0.0147	0.019
13	0	0.0000	0.000	85	0.0081	0.012
14	0	0.0000	0.000	53	0.0051	0.000
15	0	0.0000	0.000	54	0.0052	0.000
>15	0	0.0000	0.000	169	0.0161	0.012
	total	expected disk accesses		total	expected disk accesses	
	1675554	0.658		1046729	0.151	

Table 5.16: *WWWQ* searches on (*WWW2*)

	successful searches			unsuccessful searches		
length	#	%	disk accesses	#	%	disk accesses
1	1629345	84.2269	0.732	227148	28.8328	0.425
2	255276	13.1962	0.817	295670	37.5305	0.142
3	39996	2.0675	0.844	157125	19.9445	0.036
4	7386	0.3818	0.873	67188	8.5284	0.016
5	1399	0.0723	0.969	23900	3.0337	0.008
6	680	0.0352	2.222	9197	1.1674	0.015
7	250	0.0129	1.668	3836	0.4869	0.024
8	101	0.0052	1.871	1732	0.2198	0.047
9	19	0.0010	1.842	844	0.1071	0.056
10	8	0.0004	2.000	421	0.0534	0.100
11	10	0.0005	3.900	237	0.0301	0.046
12	0	0.0000	0.000	154	0.0195	0.078
13	1	0.0001	3.000	84	0.0107	0.071
14	0	0.0000	0.000	53	0.0067	0.019
15	0	0.0000	0.000	54	0.0069	0.000
>15	0	0.0000	0.000	169	0.0215	0.053
	total	expected disk accesses		total	expected disk accesses	
	1934471	0.747		787812	0.185	

Table 5.17: *WWWQ* searches on (*WWW3*)

	uncompressed	compressed
Bible	30.36	21.20
News	30.96	22.82
OED	27.16	16.31
WWW1	32.05	23.03
WWW2	31.46	22.56
WWW3	31.20	21.16

Table 5.18: Signature space for *Bible*, *News*, *OED*, *WWW1*, *WWW2*, *WWW3*

	# bytes	Suffix array	Signature array	compressed Sig
Bible	5.6M	4.5M (81%)	4.3M (77%)	3.0M (54%)
News	84.7M	61.3M (71%)	59.3M (70%)	43.7M (52%)
OED	545.6M	439.3M (81%)	372.9M (68%)	224.0M (41%)
WWW1	5.1M	3.3M (66%)	3.3M (66%)	2.4M (47%)
WWW2	50.1M	31.3M (62%)	30.7M (61%)	22.0M (44%)
WWW3	201.6M	137.4M (69%)	134.0M (67%)	90.8M (45%)

Table 5.19: Total space (bits per index point)

Chapter 6

Implementation Issues

In this chapter, we investigate some implementation issues. We discuss the creation of indexes. We study the relationship among the number of index points, the memory size required by the method and the block size. We also discuss a two-level block list.

6.1 Creating a signature array

In the proposed suffix-signature method, both a suffix array and a signature array of a text are needed. In this section we examine briefly how to generate the data structures. The signature array and its look-aside table can be created by using the suffix array of a text, or they can be created at the same time as the suffix array of a text is created.

In using a pre-existing suffix array to create a signature array, we read in each block of a suffix array sequentially. All the indexed phrases in a block addressed by the pointers in its suffix array block are read into memory. There are n random

disk accesses to read phrases if there are n pointers in the suffix array in total. The number of random disk accesses can be reduced as follows [GBYS92]: a certain number of blocks of suffix array pointers are read, and sorted by location in the text, then a large number of phrases are read from the text in a sequential pass, and then written to temporary disk space. Then each block of phrases from the temporary phrase file is read to create signatures. About 97% of comparisons are within 48 characters in the *OED* [GBYS92]. If 48 characters are read into memory for each indexed phrase, it is possible to read 600,000 phrases into a 32Mb memory. For a 600Mb text, this requires that we expect to read 1 phrase per kilobyte, so we can use sequential I/O. If the disk transfer rate is 5Mb per second, each reading of the file needs about 150 seconds. For 120,000,000 index points, it takes 200 passes, or approximately 8.3 hours.

Alternatively consider how to create a suffix array and a signature array at the same time. If a text is too huge for a suffix array to be created in memory, its suffix array could be created by merging several smaller suffix arrays, as studied and described in [GBYS92]. The algorithm suggested there uses a heap to organize pointers for each suffix array to be merged, reads phrases associated with each pointer and compares them. As indicated above, the number of random disk accesses can be reduced by writing a sufficiently large number of phrases to temporary disk space. These phrases are then merged by making a sequential pass over all the temporary phrase files. Thus it requires about 200 passes of reading a file of 600Mb, which takes about 8.3 hours. When these phrases are compared and merged, we essentially have all the phrases in memory block by block sequentially. So, we can create signatures for a block after the block is formed from the merge.

Bits in a phrase signature are allocated among words based on the phrase distribution of the block. Word signatures are created by some hashing functions,

and phrase signatures are created by using the concatenation scheme. While the signatures are created, the adjacent collision boundary phrases and the breaking point phrases are recorded into the look-aside table. The number of disk accesses needed to search for a particular phrase will not be known until the signature array block is completely created. To find the guaranteeing phrases, each indexed i-word phrase of the block is checked after the signature array block is created.

Therefore, the cost to create a suitable signature array is comparable to the cost to create the corresponding suffix array.

The size of a phrase signature should be the same as the size of a stored *integer* value, for example 32 bits. The number of words in a phrase signature is picked based on search needs. Usually, 5 or 6 is enough. Block size could be 1k to 10k index points. Smaller block sizes are better than bigger ones in terms of search performance and space, but a larger block list must be kept in memory during searching. For guaranteeing phrases, choose $\beta = 2$ to guarantee at most 2 disk accesses to a text for searching for a phrase. These parameters could be adjusted after creating test signatures for some random blocks. Strategy *Balance* described in Section 3.3.5 should be used to allocate bits among words for a block. Strategies *check1* or *check2* described in Sections 4.4.3 and 4.4.1 should be used to pick breaking phrases.

6.2 Choosing block sizes

Memory required for searching includes a *block list*, a block of *suffix array*, a block of *signature array* and its look-aside table. In memory, both *suffix array* block and *signature array* block are integer arrays. A look-aside table contains adjacent collision information, breaking phrases, and guaranteeing phrases. Adjacent collision

information includes adjacent collision boundary phrases (strings), their relative positions in a block (integers), and word positions (integers) at which collisions occur. Breaking phrases and guarantee phrases are strings, and their relative positions in a block are integers. Searches on look-aside table phrases are binary searches on strings. The *block list* contains all the block boundary phrases (strings) and starting positions (integers) of all the blocks. Searches on the *block list* are binary searches on strings.

Throughout this thesis it has been assumed that when the search engine opens a text file, the block list is loaded into memory and stays there. During each query, one block of the suffix array and the signature array and the corresponding look-aside table are loaded into memory as a single unit.

If a block is very big, the buffers for the suffix array and the signature array will occupy a lot space. Furthermore, data transfer time is longer for larger blocks. But if a block is very small, there will be many blocks, and therefore the block list will occupy a lot more space. In this section we will study how the memory size, the total number of index points and the block size affect each other.

First, we define some notation.

T: the size of a text

N: the total number of index points in a text

M: the memory allowed for data

n: the number of index points in a block

b: the number of blocks in a text

l: the number of characters in an entry of the block list

l_p: the number of bytes for an entry of the suffix array

l_s: the number of bytes for an entry of the signature array

l_b: the number of bytes for a pointer in an entry of the block list

So, we have $N = bn$. The maximum number of blocks is $b_{max} = 2^{8*l_b}$, since there are $8 * l_b$ bits of addressing available, and the minimum number of blocks is $n_{min} = \frac{N}{b_{max}}$.

Since we wish to calculate memory requirements to support signature arrays, we must examine the data as it will reside in main memory. The search algorithms invoke random probes into the block list and into each block of the signature array, and therefore these are stored in an uncompressed form in memory, even if they are compressed when stored on disk. Thus for example, we will assume 32 bits per stored signature instead of 21 to 23 bits.

Clearly we have

$$b(l + l_b) + n(l_p + l_s) \leq M.$$

Since $b = \frac{N}{n}$, we have

$$\frac{N}{n}(l + l_b) + n(l_p + l_s) \leq M.$$

So,

$$n^2(l_p + l_s) - nM + (l + l_b)N \leq 0.$$

Therefore, the number of index points in a block has to satisfy

$$n_1 \leq n \leq n_2,$$

where $n_1 = \frac{M - \sqrt{M^2 - 4(l_p+l_s)(l+l_b)N}}{2(l_p+l_s)}$ and $n_2 = \frac{M + \sqrt{M^2 - 4(l_p+l_s)(l+l_b)N}}{2(l_p+l_s)}$.

Since $M^2 - 4(l_p + l_s)(l + l_b)N$ is inside the square root, it has to be at least 0. Thus the memory size for data has to be at least

$$M_{min} = 2\sqrt{(l_p + l_s)(l + l_b)N}.$$

From the equation, we see that the minimum memory size increases as the number of index points increases.

Assume that $l_p = 4$ bytes, $l_s = 4.25$ bytes which includes 32 bits of a phrase signature and 2 bits for the look-aside table per index point on average, $l = 15$ characters and $l_b = 2$ bytes. Table 6.1 lists the minimum memory sizes for *OED*, *News* and *Bible*.

	# bytes	# index points	M_{min}(bytes)
OED	546M	110M	248.4K
News	84.7M	15.3M	91.7K
Bible	5.6M	1.13M	25.2K

Table 6.1: Minimum memory sizes for *OED*, *News*, *and Bible*

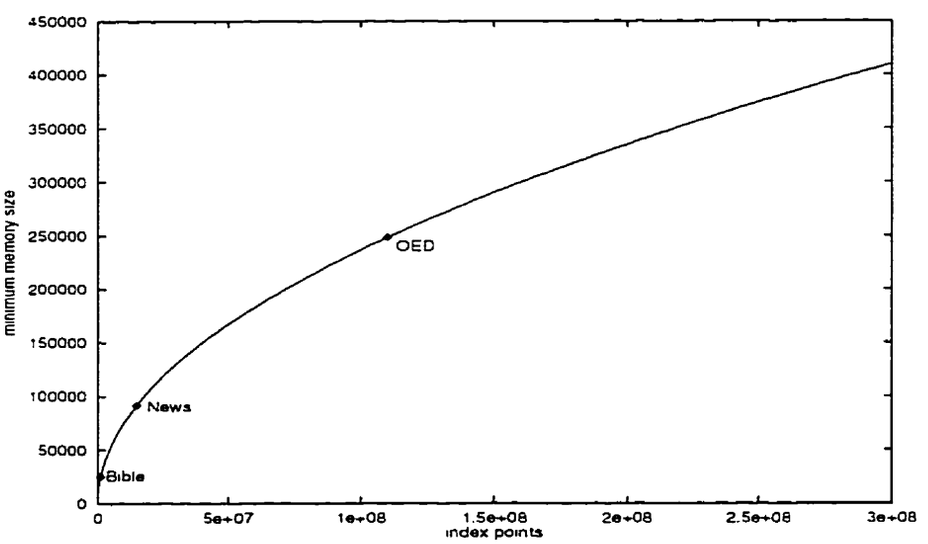


Figure 6.1: Minimum memory size

Figure 6.1 shows how the minimum memory size changes with the total number

of index points. The x-axis represents the number of index points while the y-axis stands for the minimum memory size. According to this figure, at least 237K and 411K bytes are required for 100M and 300M index points, respectively.

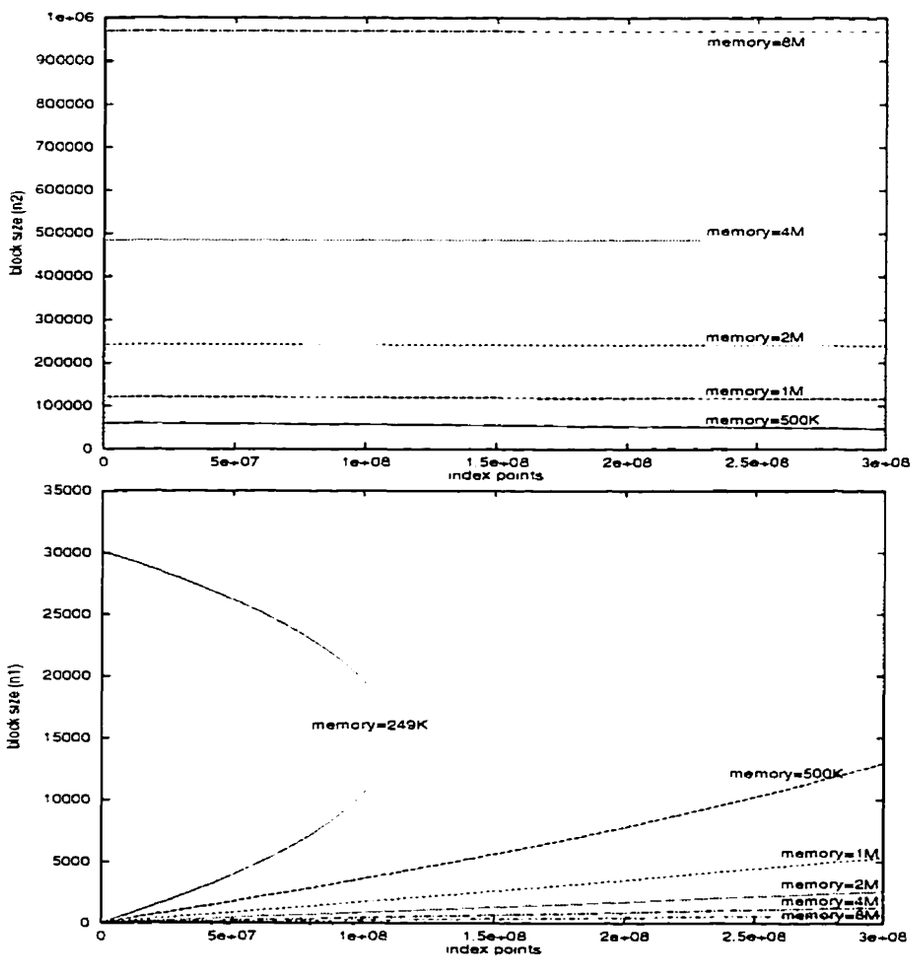


Figure 6.2: Block size vs. number of index points

We have shown that the size of a block has to be in a certain range. Figure 6.2 shows how the range of possible block sizes responds to the total number of index points and the memory size. The top part shows the upper bounds of ranges and the bottom part shows the lower bounds of ranges. The x-axis is the total number

of index points, and the y-axis shows the number of index points in a block. Curves for memory sizes of 249K (the minimum size for *OED*), 500K, 1M, 2M, 4M and 8M bytes are shown in the figure.

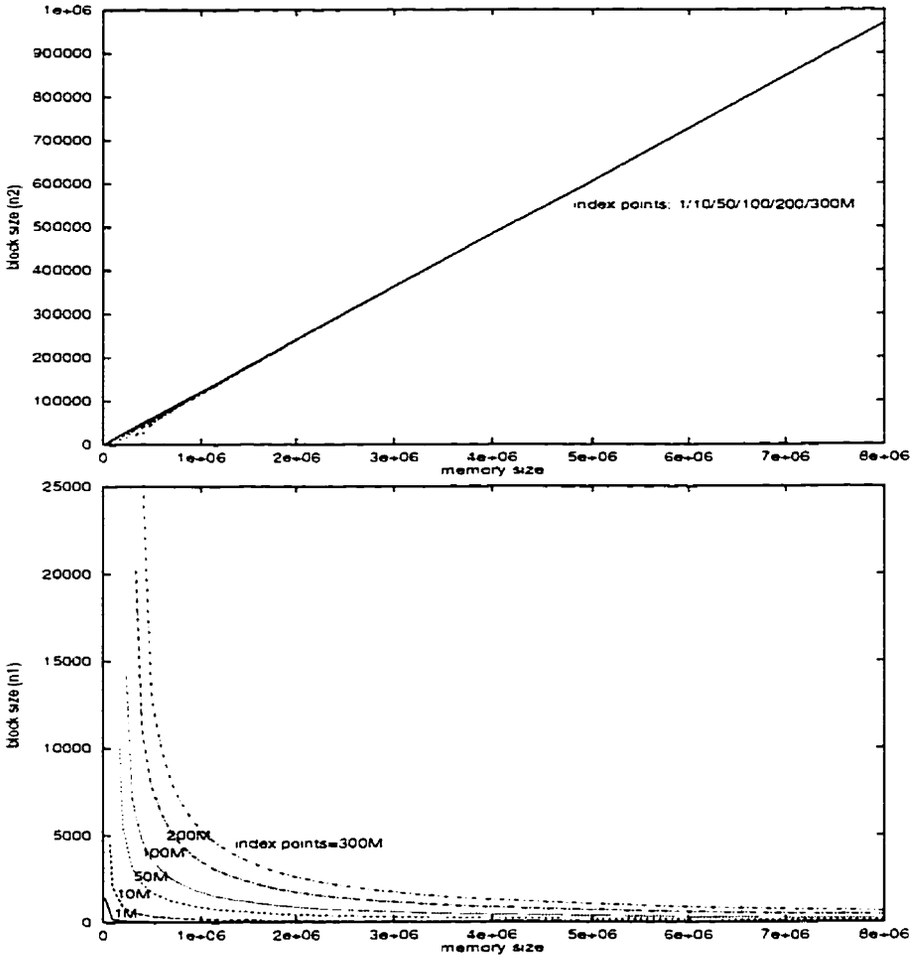


Figure 6.3: Block size vs. memory size

Figure 6.3 shows essentially the same data as Figure 6.2, but from a different vantage point. Again the top part shows the upper bounds of ranges and the bottom part shows the lower bounds of ranges. The x-axis represents the memory size, and the y-axis represents the range of possible block sizes. Curves for the total

numbers of index points of 1M, 10M, 50M, 100M, 200M and 300M are shown in the figure.

The reason that the size of a block has to be in a certain range is that the suffix array and the signature array will need a lot of memory space if each block is big, and if each block is small, more space will be required for the block list. We observe that given a memory size, the range of possible block sizes shrinks as the number of total index points increases. Also, given a total number of index points, the range of possible block sizes gets bigger as the memory size increases.

6.3 Hierarchical block list

Since the minimum memory size required increases as the number of index points increases, we may come to a point when the minimum memory requirement cannot be satisfied. Since the data size in memory is affected by the size of a block and the number of blocks, a hierarchical structure of block lists may have to be used.

In Figure 6.4, the block list is a tree of height 2. Only the root stays in memory all the time. A corresponding leaf block of the tree is loaded into memory for each search query. This will reduce the data size in memory, but as a consequence, the number of disk accesses to search for a phrase will be increased by 1 due to visiting a corresponding leaf of the block list.

We use the same notations defined in Section 6.2, and let c be the number of sub-blocks into which the block list is divided. Assume that a block of the suffix-array and the corresponding block of signature array together use memory of a full page size, that is, page size $= n(l_p + l_s)$.

Since the same memory space can be used for a leaf block of the block list and

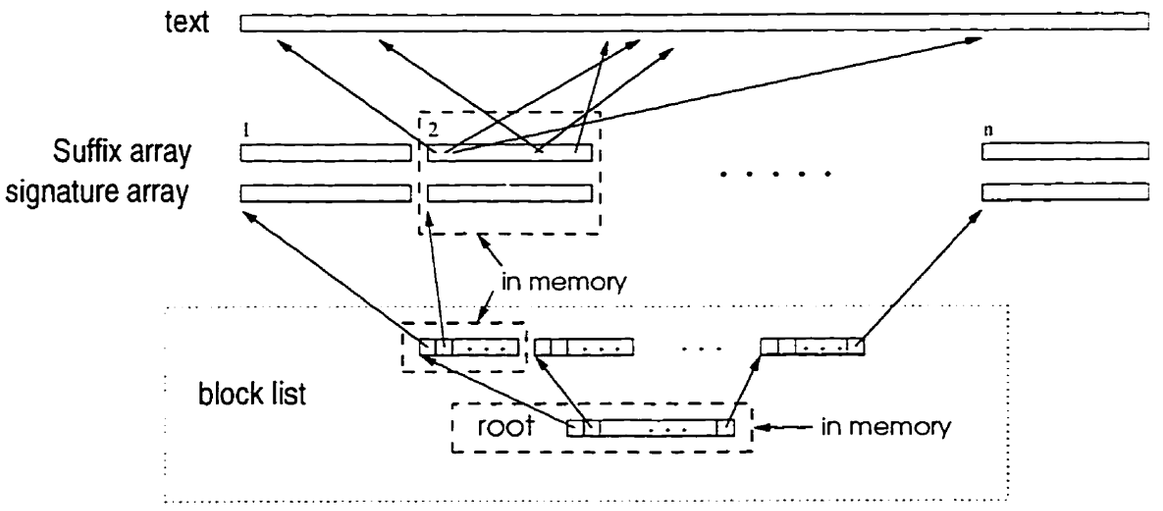


Figure 6.4: Hierarchical block list

corresponding blocks of the suffix array and the signature array at different times, we have

$$\frac{N}{cn}(l + l_b) \leq M - c(l + l_b) \tag{6.1}$$

$$n(l_p + l_s) \leq M - c(l + l_b) \tag{6.2}$$

Let a leaf block of the block list be at most a full page size, we have

$$\frac{N}{cn}(l + l_b) \leq n(l_p + l_s). \tag{6.3}$$

Therefore, n, the number of index points in a block, has to satisfy Equations 6.2 and 6.3.

From Equation 6.3, we have

$$n \geq \sqrt{\frac{N(l + l_b)}{c(l_p + l_s)}} = n_1.$$

and from Equation 6.2, we have

$$n \leq \frac{M - c(l + l_b)}{l_p + l_s} = n_2.$$

Thus, the number of index points in a block has to satisfy

$$n_1 \leq n \leq n_2.$$

Thus, $\sqrt{\frac{N(l+l_b)}{c(l_p+l_s)}} \leq \frac{M-c(l+l_b)}{l_p+l_s}$. Therefore, we have

$$M_{min}(N, c) = c(l + l_b) + \sqrt{\frac{N(l + l_b)(l_p + l_s)}{c}}. \tag{6.4}$$

Again assume that $l_p = 4$ bytes, $l_s = 4.25$ bytes which includes 32 bits of a phrase signature and 2 bits for the look-aside table per index point on average, $l = 15$ characters and $l_b = 2$ bytes. Figure 6.5 shows how the minimum memory size relates to the number of sub-blocks into which the block list is divided, and the total number of index points. The y-axis represents the minimum memory size, and the x-axis represents the number of sub-blocks for the block list. Curves for block sizes of 1M, 10M, 50M, 100M, and 200M index points are shown in the figure.

We observe that the minimum memory size depends not only on the total number of index points, but also on the number of sub-blocks of the block list. When the number of sub-blocks of the block list is in a certain range, the minimum memory size is at its valley. As the number of sub-blocks of the block list increases or decreases away from its valley range, the minimum memory size increases. This is because, for the two level block list, more memory space is required to load a sub-block of the block list during each query if the number of sub-blocks of the block list is too small, and if the number of sub-blocks of the block list is very big, more space is needed to keep the root of the block list in memory.

Figure 6.5 also indicates that for a given number of total index points N, the memory size gets its minimum at a particular value of c. Let $\frac{\partial M_{min}(N,c)}{\partial c} = 0$, we have $c = (\frac{N(l_p+l_s)}{4(l+l_b)})^{\frac{1}{3}}$. Substituting it for c in Equation 6.4, we have the minimum

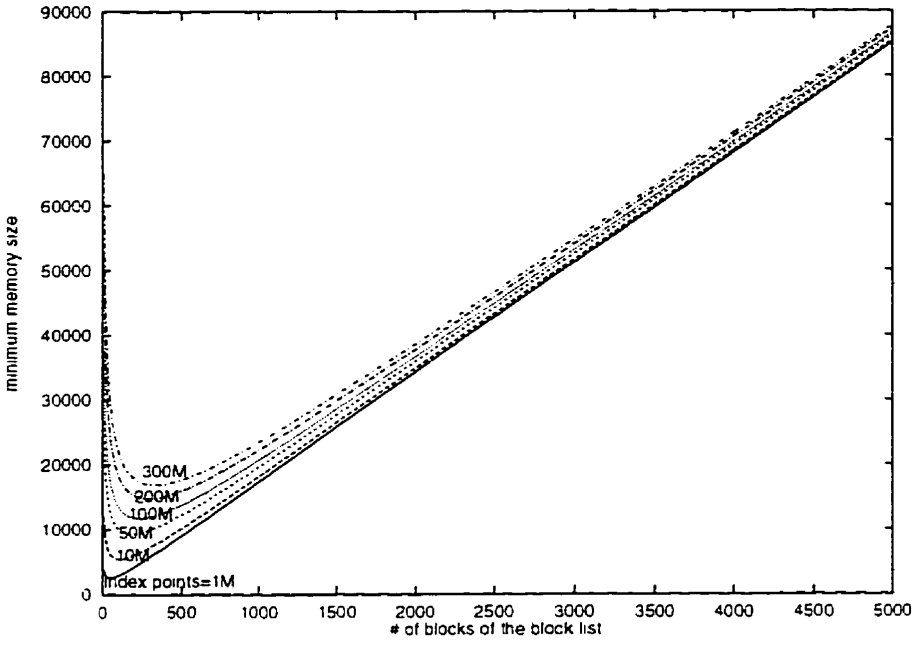


Figure 6.5: Minimum memory size for given N and c

memory required for a given N.

$$\tilde{M}_{min}(N) = (4^{-\frac{1}{3}} + 4^{\frac{1}{6}})N^{\frac{1}{3}}(l_p + l_s)^{\frac{1}{3}}(l + l_b)^{\frac{2}{3}}. \tag{6.5}$$

Figure 6.6 shows how the minimum memory size $\tilde{M}_{min}(N)$ changes with the total number of index points N. The x-axis represents the number of index points while the y-axis stands for the minimum memory size. According to this figure, at least 2.63K, 6.23K and 12.10K bytes are required for the *Bible*, *News* and *OED*, respectively.

Figure 6.7 shows, for 110M total index points (*OED*), how the range of possible block sizes relates to M (the memory size) and c (the number of sub-blocks that a block list is divided into). The bottom part of the figure shows in detail the lower-left corner of the top part. The x-axis is the memory size, and the y-axis is the number of index points in a block. Figure 6.7 shows "n-M" curves for $c = 1$.

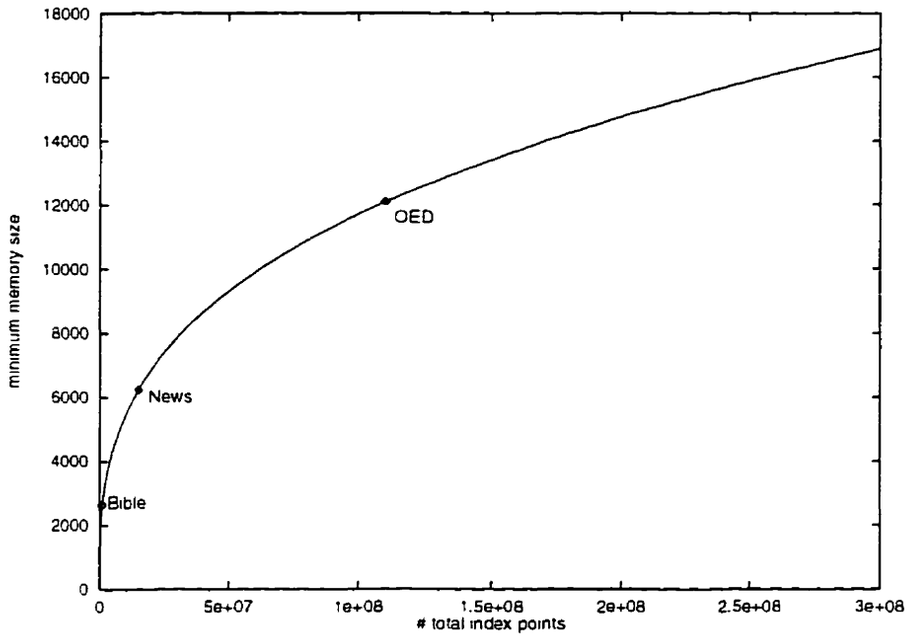


Figure 6.6: Minimum memory size for a given number of index points

50, 100, 240, 1000, and 2000. In the bottom part, it also plots curve $M_{min}(N, n)$ for N = 110M, derived from $M_{min}(N, c)$ by substituting c by $c = \frac{N(l+l_b)}{n^2(l_p+l_s)}$ (from Equation 6.3).

We observe that for a given number of sub-blocks of a block list, the "n-M" curve is a straight line starting at a point (x, y) on curve $M_{min}(110 * 10^6, n)$. It means that for a given number of sub-blocks of a block list, the minimum memory size is x, the minimum number of index points in a block is y, and the number of index points in a block is restricted in a range. The minimum number of index points in a block is restricted by the total number of index points, and the maximum number is restricted by the memory size and the page size. The bigger the available memory, the larger the possible range of the block size.

When the number of sub-blocks is very small, the number of index points in

	# bytes	# index points	M_{min}(bytes)
OED	546M	110M	12.10K
News	84.7M	15.3M	6.23K
Bible	5.6M	1.13M	2.63K

Table 6.2: Minimum memory sizes for *OED*, *News*, and *Bible*

a block cannot be very small, otherwise there would be too many blocks, which would make big sub-blocks of the block list. The maximum block size decreases as the number of sub-blocks increases. This is because more space is required by the root of the block list.

The bottom part of Figure 6.7 highlights points corresponding to page sizes of 2k, 4k, and 8k bytes on curve $M_{min}(110 * 10^6, n)$. The top part highlights the point corresponding to page size 1k. The numbers of index points in a block for page sizes of 1k, 2k, 4k and 8k are at most 121, 242, 484 and 970, respectively. The minimum memory sizes required for page sizes of 1k, 2k, 4k and 8k bytes are about 263.8k, 67.7k, 20.4k and 12.1k bytes, respectively.

6.4 Summary

In this chapter we studied the relationship among the size of memory required, the number of total index points, and the size of a block for both one-level and two-level block lists. The reported results are used for picking parameters for the method. The total number of index points determines the minimum size of memory. For a given size of memory, the block size must fall within a certain range. If sufficient memory is not available, a two-level block list must be used, which costs one more

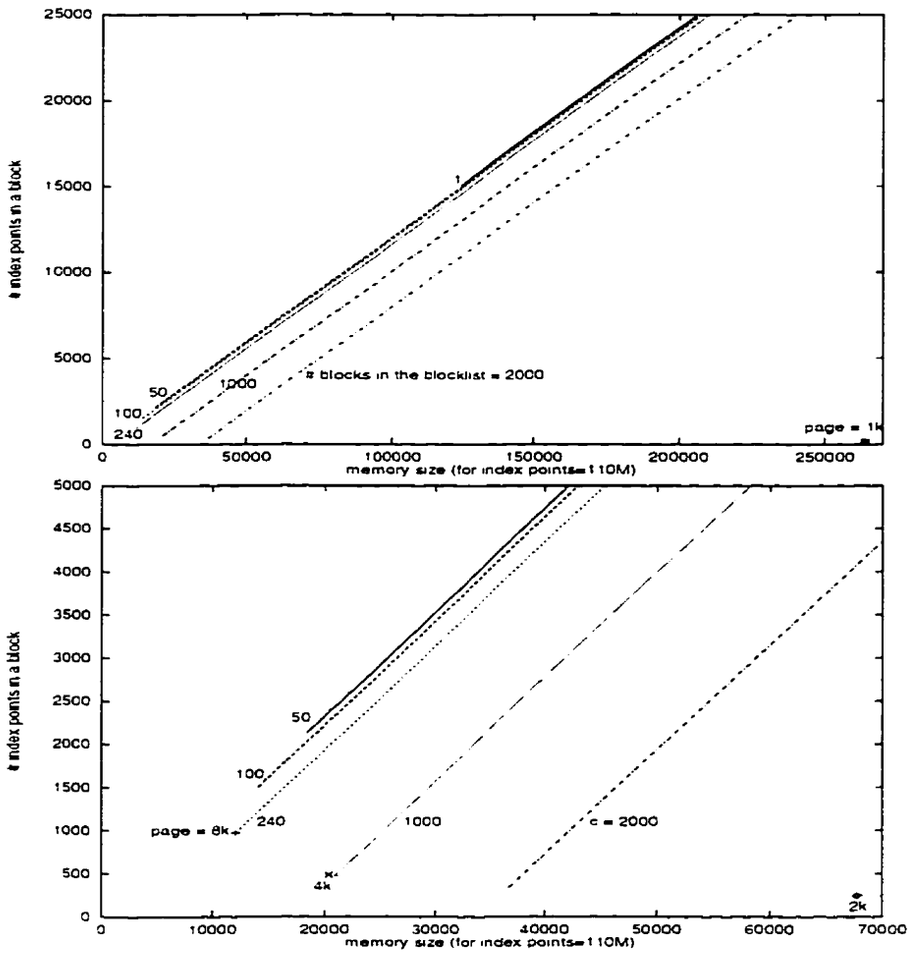


Figure 6.7: Block size vs. memory size (*OED*)

disk access for all searches.

Chapter 7

Conclusions

In this chapter, we summarize the results of the thesis, compare the suffix-signature method with some other structures, and outline future work.

7.1 Summary

We have proposed the suffix-signature method for searching for all the occurrences of a given phrase in large static texts. Using this method, phrase searches can be done very efficiently, as shown theoretically and experimentally. Assume that phrase signatures are based on the first k words of phrases. For a phrase of length $l \leq k$, the method requires two disk accesses on average (one to the index and one to the text). The average number of disk accesses for unsuccessful searches approaches 1 as the number of words in a phrase increases. For both successful and unsuccessful searches, the method requires at most two disk accesses to text in the worst case to search for a phrase of length $l \leq k$.

We studied word signature properties. In particular, we investigated the dis-

tribution of the number of adjacent collisions, the distribution of the number of collisions to a given word, and the trade-off between signature size and the number of disk accesses. We gave solutions to address adjacent collisions and to reduce impacts of separate collisions.

We investigated phrase signature schemes and proposed the concatenation scheme for phrase signatures. We studied and compared strategies to allocate bits among words in a phrase for the concatenation scheme. We also investigated some alternative phrase signature schemes. We compared the concatenation scheme against the superimposition scheme for phrase signatures, and concluded the superimposition scheme has far more adjacent collisions than the concatenation scheme. We studied the possibilities of using text compression techniques for phrase signatures.

We gave search algorithms for the method and proposed a way to guarantee two disk accesses to the text in the worst case to search for a phrase of bounded length. We also discussed balancing different parts of the look-aside table and the search performance. We showed that under many distributions, most searches require only two disk accesses in total, and no search for a phrase of bounded length requires more than three disk accesses in total.

We gave a model to calculate the minimum memory space and to choose the block size for a one-level block list. We also studied a hierarchical block list, and a model for calculating the minimum memory space, choosing the block size, and dividing the block list.

7.2 Comparison to other structures

The proposed suffix-signature method is very efficient for doing phrase searches. It requires two disk accesses on average and three disk accesses at most to search for a phrase of bounded length. In particular, it is very fast for unsuccessful searches because it can quickly filter out unqualified phrases. In our experiments using real queries from the World Wide Web, unsuccessful one-word searches need $0.3 - 0.4$ disk accesses to text on average, and less than 0.2 disk accesses over all phrases of all lengths.

It also handles range searches well. Since it is based on a suffix array, and any operation on a Patricia tree can be simulated on a suffix array at the cost of a factor of $O(\log_2 n)$ [GBYS92], the method has a potential use in other kinds of searches, such as, *proximity search* [MBY91], *most frequent search*, *longest repetition search* (with additional supporting bits), and *regular expression search* [BY89].

For a word level implementation of phrases up to five words, the total size of the suffix array and the signature array is about 110% to 130% of the original text (see Table 5.19), which includes $60 - 80\%$ of the original text for the suffix array and $40 - 55\%$ for the signature array and look-aside table, depending on the text.

Inverted lists

As we discussed in Section 1.1, an inverted list is very well suited for one-word searches, and the storage overhead for a word-level implementation is 30% to 100% of the original file size [GBYS92]. Searching for a phrase requires set operations involving all postings for each word in the phrase. If words in a target phrase are very common, intersecting several result sets of words would require many disk

accesses. The number of disk accesses to search for a phrase is the total number of blocks involved in each word of a phrase.

As compared with other structures, the suffix-signature method is slower than an inverted list for one-word searches (unless a word list is used in place of the block list - see below), but faster for phrase searches. It uses more space than an inverted list.

Inverting five word phrases

Inverting all distinct five word phrases requires fewer disk accesses to search for a phrase and more disk space than the suffix-signature method used on the first five words of all suffixes.

Speed

Since the number of distinct five word phrases is comparable to the number of words in the text, the inverted phrase list needs to be divided into blocks. Assume that a block list fits in memory, then one disk access is required to get to the block in which the target phrase falls. For a phrase of up to five words, no more disk accesses are needed, therefore one disk access in total is required to search for a phrase of length up to five words.

Space

Inverting all distinct five word phrases is at best a lossless compression of the text, and the signature array on five words of all phrases is a lossy compression. From an information theoretical point of view, the suffix-signature method should in principle use less space than inverting all distinct phrases containing the same number of words.

The inverted list has two parts, a phrase list and postings. A suffix array gives all the postings. The phrase list consists of all the distinct five word phrases. Since the distinct five word phrases are sorted alphabetically, each phrase can be represented by (t_1, t_2, s), where t_1 is the length of the common prefixes of this phrase as compared to the alphabetically previous one, t_2 is the length of the remaining part of this phrase, and s is the remaining part. Remaining parts of all distinct five word phrases could be compressed by using a text compression scheme. Assuming a compression of 30% for the remaining strings, they would require about 60% of the space used by the original texts for our experimental data files: *Bible*, *News*, *OED* and web page files. Since compressing short text strings may not have a compression rate as good as 30%, the phrase list very likely uses much more than 60% of the space of original texts.

Therefore, inverting all distinct five word phrases requires fewer disk accesses for searches, but more space, than using the suffix-signature method on the first five words of all suffixes.

Conventional signature files

Conventional signature files are similar to the superimposition phrase signature scheme. This suits partial matches, but does not perform as well as the suffix-signature method for exact phrase searches, as shown in Section 3.4.2 due to the number of collisions.

Tries

In addition to supporting *string search*, trie structures support other kinds of searches, such as, *proximity search*, *most frequent search*, *longest repetition search*

(with additional supporting bits), and *regular expression search* [BY89].

A PaTrie [Sha95] is a pointer-less representation of a binary Patricia trie. Experimentally, it requires 5 to 7 disk accesses when using 1k byte pages for searching a text with 100 million index points. The worst case might be 46 disk accesses. It is unlikely to be well extended to larger block sizes, since it is processor intensive (it scans every bit when traversing a block). The space is reported to be 0.65 to 0.78 words (*i.e.* 20.80 to 24.96 bits) per index point.

A Compact Pat tree [Cla96] requires at most 5 disk accesses when using 1k byte pages for searching the *OED*, and requires 3 disk accesses using 8k byte pages. Its average number of disk accesses is close to its worst case. It handles dynamic texts. Experimentally, the space of the static Compact Pat tree on the text of the complete works of *Sherlock Holmes* (238.6Kb), the *Bible* (5.6Mb), and the *OED* (546Mb) are about 60%, 88%, and 99% of the original text, respectively. The space of the dynamic version on a collection of 161 documents of size 21Kb to 1.4Mb is about 134% of the original text.

The suffix-signature method requires no more disk accesses than a trie structure, and given sufficient memory can be designed to use fewer disk accesses. For example, a hierarchical suffix-signature structure requires 3 disk accesses on average and 4 in the worst case when using 1k byte pages for searching a text of 110M index points, given a 264Kb memory. It uses more space than a static Compact Pat tree, but less space than dynamic trie structures.

Some prototype systems

MG (*Managing Gigabytes*) [WMB94] is a full-text retrieval system. It compresses texts and images and indexes them using inverted lists. It supports boolean queries

and ranked queries. A compressed text is about 25% of the original text, and an index is about 10%. On a Sun SPARC 10 model 512, it takes about 4 hours and 39 minutes of cpu time to invert the 2Gb *TREC* collection. Using plain compressed inverted lists, typical queries of five to ten terms are resolved in 3 to 4 seconds of cpu time, and slightly more elapsed time. Skipping[1] allows them to be resolved 4 to 6 times faster.

Glimpse [MW93] provides indexing and query schemes for personal file systems. It uses a two-level indexing and searching scheme. An index is an inverted list of distinct words followed by a list of blocks. It searches the index and then searches each block on the list. The index is about 2-4% of the original text. It allows boolean queries, approximate matching,and searching for regular expressions. On a DEC 5000/240 workstation, it took 4.9 minutes of cpu time (9 minutes of elapsed time) to index a file system containing 69Mb of text. A typical search takes about 2-10 seconds cpu time.

Compared with these prototype systems, the proposed suffix-signature method uses more space for faster phrase searches.

7.3 Future work

The suffix-signature method might be particularly useful in languages, such as Japanese or Chinese, where word boundaries are not well defined.

Japanese Industrial Standards have 6353 characters for general use [YM91]. It has been showed that the length of words is 2.51 characters for *Kanji* and 4.39

[1] A sequence of pointers are interleaved with blocks of a compressed inverted list entry to provide random access into the inverted list entry for faster searching.

for *Katakana* on average. Since words are not separated by spaces in a sentence in Japanese, inverted lists based on words are not easily created for Japanese. Therefore many Japanese document retrieval systems use character-based indexing [YM91]. Because merging postings lists of inverted lists based on single characters incurs significant cost, indexing strings of two, or sometimes more, characters has been proposed for Japanese.

Yasushi and Masajirou [YM91] also show that for *Kanji*, 73.57% of distinct words are of length equal to or fewer than 5 characters, and 99.37% are of length equal to or fewer than 10 characters; for *Katakana*, the corresponding numbers are 26.80% and 88.97%; and on average, these numbers are 68.71% and 98.29% for 5 and 10 characters respectively. The suffix-signature method therefore could be used effectively to index Japanese documents. The approach to take is to index every character and to store signatures for corresponding multi-character strings, therefore effectively storing word signatures as if they were phrase signatures. Because of the prefix property, searching for individual words in Japanese text could then be guaranteed to take no more than 3 disk accesses without requiring Japanese word segmentation.

Whether for Japanese or for Western languages, one possible extension of this work is to use ideas of the suffix-signature method in combination with other data structures to achieve fast searches. For instance, k-word signatures of phrases could be stored with postings in an inverted list. This additional data can be used to filter out unqualified elements in a postings list in answering a phrase query. As a result, set operations would be done on significantly smaller sets for phrase searches.

We studied a two-level block list for the situation when the number of blocks is too big for the size of an available memory. With such an approach, one disk access is needed to find the block in which a given phrase falls. Alternatively, a

word list of distinct first words of block boundary phrases can be used. A word on the word list that appears too many times in a text might need another word list on the second words to reduce the size of its blocks, as illustrated in Figure 7.1. So, the block list is uneven in height. Some phrases need no disk accesses on the block list, and some need one disk access, assuming that the first level of the block list is in memory and the second level is on disk. If the block list is based on a word list (like in the experiment for Section 3.4.2), then the first word need not be represented in the signature, saving space, reducing disk accesses, or increasing the effective number of words represented by a phrase signature.

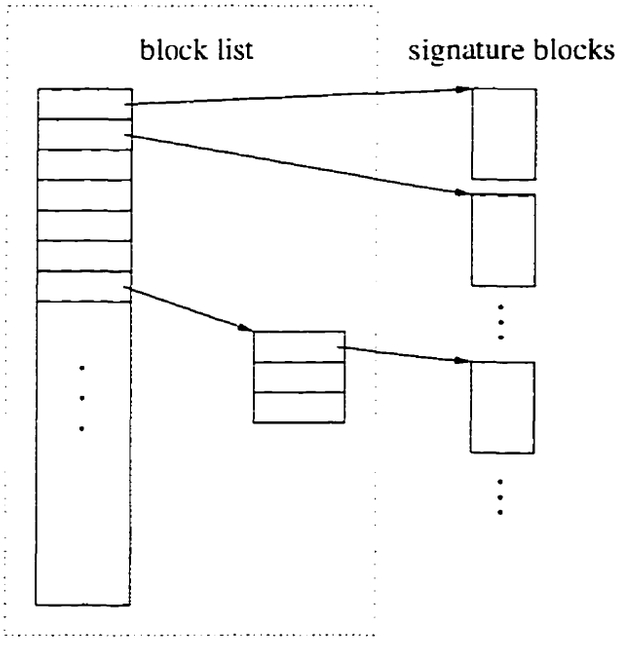


Figure 7.1: A block list of uneven height

Appendix A

Experiment Corpus

In this appendix, we describe the texts used in our experiments. They are the *Bible*, *News*, *OED*, and three different sizes of texts from the World Wide Web.

A.1 *Bible*

This is an SGML encoded version of the King James Version of the *Bible* - the Old Testament, Apocrypha, and the New Testament. Following is a piece from the *Bible*:

<bible><book><NAME>Gen</NAME><chap><C>Gen 1.</C><V><L> 1.</L> In the beginning God created the heaven and the earth.</V>
<V><L>2.</L> And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.</V>
<V><L>3.</L> And God said, Let there be light: and there was light.
</V>

The text of the *Bible* is about 5.6Mb. There are roughly 1.13M indexed phrases in its suffix array. The suffix array and the signature array are divided into 113 blocks. each block having 10K indexed phrases. Table A.1 characterizes the *Bible* phrases.

	1-word	2-words	3-words	4-words	5-words
m_i	14.666	180.965	498.667	740.433	886.727

Table A.1: Numbers of distinct i-word phrases (*Bible*)

A.2 *News*

This text is an SGML version of the *Ottawa Citizen* newspaper from July 1990 to December 1990. Following is a piece from the *News*:

```
<I><K#>1178128</K#><CS>Ready</CS><PA>Ottawa Citizen</PA><PD>Tue 31
Jul 90</PD><D>900731</D><DAY>Tue</DAY><ED>Final</ED><SEC>BUSINESS
</SEC><PG>D5</PG><HL>Consumer spending in U.S. up 1.0 per cent in
June</HL><SRC>AP</SRC><DL>WASHINGTON</DL><ST>NEWS</ST><L>354</L><CN>
LK</CN><DOB>900731</DOB><UP>900731</UP><AKN>1178128</AKN><T> <P>---
Consumer spending in U.S. up 1.0 per cent in June     ---  </P><P>
WASHINGTON (AP) -- Consumer spending in the United States jumped
1.0 per cent in June, the largest gain in five months, while
personal in comes rose 0.4 per cent, the government said Monday.</P>
```

The text of the *News* is about 84.7Mb. There are roughly 15.3M indexed phrases in its suffix array. The suffix array and the signature array are divided into 1534

blocks, each block having 10K indexed phrases. Table A.2 characterizes the *News* phrases.

	1-word	2-words	3-words	4-words	5-words
n_i	246K	3.074K	7.938K	11.227K	12.868K

Table A.2: Numbers of distinct i-word phrases (*News*)

A.3 *OED*

This is a tagged version of the *Oxford English Dictionary, second edition.* Following is a piece from the *OED*:

```
<e><hg><hw>A</hw> <pr><ph>eI</ph></pr></hg>, <s0>the first letter of
the Roman Alphabet, and of its various subsequent modifications
(as were its prototypes Alpha of the Greek, and Aleph of the
Ph&oe.nician and old Hebrew); representing originally in English,
as in Latin, the &oq.low-back-wide&cq. vowel, formed with the widest
opening of jaws, pharynx, and lips.&es.The plural has been written
<cf>aes</cf>, A's, <cf>As</cf>. &es.<il>from A to Z</il>: see <xr>
<x>Z</x> <xs>3</xs>. </xr><qp><q><qd>&c. 1340</qd><a>Hampole</a>
<w>Pr. Consc.</w> <lc>481</lc> <qt>And by &th.at cry men knaw
&th.an Whether it be man or weman, For when it es born it cryes
swa.&es.If it be man it says a! a!&es.That &th.e first letter is of
&th.e nam Of our forme-fader Adam. </qt></q> <q><qd>&c.1386</qd>
```

The text of the *OED* is about 545.6Mb. There are roughly 109.8M indexed phrases in its suffix array. The suffix array and the signature array are divided into about 11K blocks, each block having 10K indexed phrases. Table A.3 characterizes the *OED* phrases.

	1-word	2-words	3-words	4-words	5-words
m_i	2.670K	15.834K	36.512K	53.660K	65.062K

Table A.3: Numbers of distinct i-word phrases (*OED*)

A.4 Data from *World Wide Web*

These are HTML pages from the World Wide Web. Three texts of sizes 5.1Mb, 50.1Mb, and 201.6Mb were taken from different portions of the pages. A search query log from World Wide Web was collected from 1996 for about 2 months. Following is a piece from the data:

```
<OTData>
<TITLE>RADIOACTIVA - LIAM NEESON</TITLE>
<TABLE><IMG><H3>Liam Neeson</H3><H4>(1953 - )</H4>Actor Fecha de
Nacimiento: 1953, Ballymena, Irlanda del Norte Educacion: Lyric
Players' Theatre, Belfast; Abbey Theatre, Dublin, Irlanda</TABLE>
1981 EXCALIBUR actor 1984 THE BOUNTY actor 1985 THE INNOCENT actor
1985 LAMB actor 1986 DUET FOR ONE actor 1986 LA MISION/ THE MISSION
actor 1987 A PRAYER FOR THE DYING actor 1987 SUSPECT actor 1988 THE
DEAD POOL actor 1988 THE GOOD MOTHER actor 1988 HIGH SPIRITS actor
```

1988 SATISFACTION/GIRLS OF SUMMER actor 1989 NEXT DE KIN actor 1990 DARKMAN actor 1991 CROSSING THE LINE/THE BIG MAN actor 1992 MAR IDOS Y ESPOSAS/ HUSBANDS AND WIVES actor 1992 SALTO DE FE/ LEAP DE FAITH actor 1992 REVOLVER actor 1992 UN DESTELLO EN LA OSCURIDAD/ SHINING THROUGH actor 1992 BAJO SOSPECHA/ UNDER SUSPICION actor 1993 DECEPTION actor 1993 ETHAN FROME actor 1993 LA LISTA DE SCHINDLER/ SCHINDLER'S LIST actor<H3>Oscar </H3>Nominado por Mejor Actor 1993 : SCHINDLER'S LIST<>[RADIOACTIVA - Cine] [RADIOACTIVA - Menu] </OTData></OTDoc><OTDoc>

A.4.1 *WWW1*

The text of *WWW1* is about 5.1Mb. There are roughly 834K indexed phrases in its suffix array. The suffix array and the signature array are divided into 84 blocks, each block having 10K indexed phrases. Table A.4 characterizes the *WWW1* phrases.

	1-word	2-words	3-words	4-words	5-words
m_i	60K	314K	508K	594K	637K

Table A.4: Numbers of distinct i-word phrases (*WWW1*)

A.4.2 *WWW2*

The text of *WWW2* is about 50.1Mb. There are roughly 7.8M indexed phrases in its suffix array. The suffix array and the signature array are divided into 782 blocks, each block having 10K indexed phrases. Table A.5 characterizes the *WWW2* phrases.

	1-word	2-words	3-words	4-words	5-words
m_i	405K	2.646K	4.657K	5.460K	5.802K

Table A.5: Numbers of distinct i-word phrases (*WWW2*)

A.4.3 *WWW3*

The text of *WWW3* is about 201.6Mb. There are roughly 34.4M indexed phrases in its suffix array. The suffix array and the signature array are divided into 3436 blocks, each block having 10K indexed phrases. Table A.6 characterizes the *WWW3* phrases.

	1-word	2-words	3-words	4-words	5-words
m_i	635K	6.923K	16.546K	22.329K	24.742K

Table A.6: Numbers of distinct i-word phrases (*WWW3*)

A.4.4 *WWWQ*

Experiments were conducted on the *WWW1*, *WWW2*, and *WWW3*, by using real search queries in the search query log *WWWQ* from *WWW*. The query log began on approximately Dec 27, 1996 and covered about 2 months. The queries in the log include those posed to the Open Text Index as simple search queries and as power search queries (see http://index.opentext.net).

One simple search query might consist of several word searches, or one phrase search. For example, a compound word search "tourism Malaysia" searches for

web pages containing words "tourism" and "Malaysia". A simple phrase search "American Political Science Association" is to search for web pages which have the phrase "American Political Science Association". A user can request a phrase search, even if only one word is specified.

A power search query is more complicated than a simple search query, it might consist of several word searches and phrase searches. For example, the power search in Table A.7 is to search for web pages which have "Hampshire" somewhere other than the titles of pages, have "Randall", and have no "New Hampshire".

	Hampshire	*Anywhere*
And	Randall	*Anywhere*
But not	Hampshire	*Title*
But not	New Hampshire	*Anywhere*

Table A.7: An example of power searches

The trace includes in total about 3.4 million simple search queries and power search queries. The percentages of word search queries of simple search queries, phrase search queries of simple searches, and power search queries are 54.0%, 15.5%, and 30.5%, respectively.

In our experiments, a power search is converted to several simple phrases. For example, the power search query in Table A.7 is converted to four simple searches, "Hampshire", "Randall", "Hampshire", and "New Hampshire". The *WWWQ* includes phrase searches of simple search queries, and word or phrase searches converted from power search queries. As a result from this trace, the total number of phrases in *WWWQ* is about 2.7 million. Table A.8 is the distribution of search length of *WWWQ*. It lists the number of i-word queries, its percentage over all

queries, and the accumulative percentage of queries of lengths from 1 to i.

In order to estimate the unsuccessful search rate of *WWWQ* queries on a real search index, we randomly picked query phrases from *WWWQ*, and then searched them on the full *Open Text Index* as it existed in April 30, 1997. For 1000 randomly selected phrases from the trace, the average unsuccessful search rate was 17.6%. We also randomly picked, from *WWWQ*, query phrases that have 200 1-word phrases, 200 2-word phrases, 200 3-word phrases, 200 phrases of 4 or 5-words, and 200 phrases longer than 5 words. The average unsuccessful rate for those 1000 phrases was 50.9%.

length	#	%	accumulative %
1	1856493	68.196%	68.196%
2	550946	20.238%	88.434%
3	197121	7.241%	95.675%
4	74574	2.739%	98.414%
5	25299	0.929%	99.343%
6	9877	0.363%	99.706%
7	4086	0.150%	99.856%
8	1833	0.067%	99.923%
9	863	0.032%	99.955%
10	429	0.016%	99.971%
11	247	0.009%	99.980%
12	154	0.006%	99.986%
13	85	0.003%	99.989%
14	53	0.002%	99.991%
15	54	0.002%	99.993%
>15	169	0.007%	100.000%

Table A.8: Distribution of search lengths in *WWWQ*

Appendix B

Detailed Experimental Results

This appendix gives detailed search performance of experiments described in Chapter 5 on the *Bible*, *News*, *OED*, *WWW1*, *WWW2*, and *WWW3*. Experiments are presented by percentages of searches of 0, 1, 2, and 3 disk accesses to a text. Three query distributions, *uniform phrases*, *proportional to occurrences*, and *DeFazio*, as described in Section 5.2.4, are used for successful searches. Two models *Uniform i−word signatures* and *uniform word signatures*, as described in Section 5.2.4, for unsuccessful searches are also used in our experiments. We also list the numbers of adjacent collisions, breaking points, and guaranteeing phrases.

B.1 *Bible*

Phrase signatures are structured from the first 5 words of phrases. Table B.1 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	2.34	6.30	7.40	6.40	5.87	28.3108
adjacent collisions	1036	1113	1387	2525	1958	8019
breaking points	286	1071	701	615	614	3287
guaranteeing phrases	0	36	27	16	18	97

Table B.1: Signatures and the look-aside table (*Bible*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	20.190%	4.337%	2.094%	1.534%	1.314%
1 disk access	67.919%	88.140%	95.262%	97.092%	97.752%
2 disk accesses	11.891%	7.503%	2.639%	1.371%	0.932%
3 disk accesses	0.000%	0.020%	0.005%	0.002%	0.002%
4 disk accesses and over	0.000%	0.000%	0.000%	0.001%	0.000%
0,1,2 disk accesses	100.000%	99.980%	99.995%	99.997%	99.998%

Table B.2: Uniform phrases (*Bible*)

Tables B.2, B.3, and B.4 describe search performances for indexed phrases under

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	91.729%	52.234%	24.083%	10.262%	4.111%
1 disk access	7.612%	44.430%	73.621%	87.962%	94.469%
2 disk accesses	0.659%	3.331%	2.293%	1.774%	1.418%
3 disk accesses	0.000%	0.005%	0.003%	0.002%	0.002%
4 disk accesses and over	0.000%	0.000%	0.000%	0.000%	0.000%
0,1,2 disk accesses	100.000%	99.995%	99.997%	99.998%	99.998%

Table B.3: Proportional to occurrences (*Bible*)

three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.2 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.3 is for querying a phrase as many times as it appears in the *Bible*. Table B.4 is obtained by using the DeFazio distribution to query indexed phrases. The three results are relatively close. 88%, 93%, 97%, 98% and 99% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.5 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.6 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	62.080%	16.880%	8.809%	4.660%	3.947%
1 disk access	33.126%	75.983%	88.560%	93.581%	94.969%
2 disk accesses	4.793%	7.122%	2.627%	1.758%	1.081%
3 disk accesses	0.000%	0.015%	0.004%	0.001%	0.002%
4 disk accesses and over	0.001%	0.000%	0.000%	0.000%	0.001%
0,1,2 disk accesses	99.999%	99.985%	99.996%	99.999%	99.997%

Table B.4: Under the DeFazio distribution (*Bible*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	31.416%	41.901%	95.317%	98.233%	99.706%
1 disk access	14.546%	39.522%	4.472%	1.524%	0.253%
2 disk accesses	54.038%	18.577%	0.211%	0.243%	0.041%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.5: Uniform i-word signatures (*Bible*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	31.416%	73.396%	97.202%	96.408%	97.505%
1 disk access	14.546%	15.838%	2.698%	3.309%	2.411%
2 disk accesses	54.038%	10.766%	0.100%	0.283%	0.084%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.6: Uniform word signatures (*Bible*)

B.2 *News*

Phrase signatures are structured from the first 5 words of phrases. Table B.7 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	2.13	6.53	7.76	6.86	5.66	28.95
adjacent collisions	11,536	16,410	22,462	27,512	27,832	105,752
breaking points	3,998	17,721	11,640	7,599	5,937	46,895
guaranteeing phrases	9	426	352	306	817	1,910

Table B.7: Signatures and the look-aside table (*News*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	13.290%	3.363%	2.238%	2.322%	2.067%
1 disk access	75.589%	89.428%	95.150%	96.519%	97.257%
2 disk accesses	11.118%	7.195%	2.608%	1.157%	0.670%
3 disk accesses	0.000%	0.012%	0.003%	0.001%	0.001%
4 disk accesses and over	0.003%	0.002%	0.001%	0.001%	0.005%
0,1,2 disk accesses	99.997%	99.986%	99.996%	99.998%	99.994%

Table B.8: Uniform phrases (*News*)

Tables B.8, B.9, and B.10 describe search performances for indexed phrases

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	92.784%	49.712%	20.660%	9.476%	4.167%
1 disk access	6.649%	46.948%	76.898%	89.127%	94.919%
2 disk accesses	0.567%	3.335%	2.439%	1.395%	0.908%
3 disk accesses	0.000%	0.005%	0.003%	0.001%	0.001%
4 disk accesses and over	0.000%	0.000%	0.000%	0.001%	0.005%
0,1,2 disk accesses	100.00%	99.995%	99.997%	99.998%	99.994%

Table B.9: Proportional to occurrences (*News*)

under three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.8 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.9 is for querying a phrase as many times as it appears in the *News*. Table B.10 is obtained by using the DeFazio distribution to query indexed phrases. The three results are quite close. 89%, 93%, 97%, 99% and 99% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.11 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.12 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	68.352%	16.527%	6.489%	4.983%	3.730%
1 disk access	27.807%	76.571%	90.779%	93.705%	95.435%
2 disk accesses	3.841%	6.893%	2.729%	1.311%	0.835%
3 disk accesses	0.000%	0.008%	0.003%	0.001%	0.000%
4 disk accesses and over	0.000%	0.001%	0.000%	0.000%	0.000%
0,1,2 disk accesses	100.00%	99.991%	99.997%	99.999%	100.00%

Table B.10: Under the DeFazio distribution (*News*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	44.172%	42.272%	92.538%	99.043%	99.580%
1 disk access	11.225%	37.457%	6.994%	0.901%	0.389%
2 disk accesses	44.603%	20.271%	0.468%	0.055%	0.031%
0,1,2 disk accesses	100.000%	100.000%	100.00%	99.999%	100.00%

Table B.11: Uniform i-word signatures (*News*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	44.172%	65.510%	94.623%	97.591%	96.660%
1 disk access	11.225%	18.862%	5.010%	2.308%	3.259%
2 disk accesses	44.603%	15.628%	0.367%	0.101%	0.081%
0,1,2 disk accesses	100.000%	100.000%	100.00%	100.000%	100.00%

Table B.12: Uniform word signatures (*News*)

B.3 *OED*

Phrase signatures are structured from the first 5 words of phrases. Table B.13 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	1.79	4.89	6.59	6.55	5.93	25.75
adjacent collisions	50,957	65,246	86,235	118,046	140,219	460,703
breaking points	46,733	86,503	63,847	54,955	48,754	300,792
guaranteeing phrases	13	778	958	1,386	6,483	9,618

Table B.13: Signatures and the look-aside table (*OED*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	6.945%	2.965%	1.792%	1.387%	1.203%
1 disk access	80.379%	90.642%	95.328%	96.940%	97.607%
2 disk accesses	12.675%	6.388%	2.877%	1.671%	1.180%
3 disk accesses	0.000%	0.005%	0.002%	0.001%	0.001%
4 disk accesses and over	0.001%	0.000%	0.001%	0.001%	0.009%
0,1,2 disk accesses	99.999%	99.995%	99.997%	99.998%	99.990%

Table B.14: Uniform phrases (*OED*)

Tables B.14, B.15, and B.16 describe search performances for indexed phrases

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	92.806%	67.093%	41.105%	21.896%	8.115%
1 disk access	6.554%	31.023%	57.117%	76.503%	90.164%
2 disk accesses	0.640%	1.883%	1.776%	1.599%	1.713%
3 disk accesses	0.000%	0.001%	0.002%	0.002%	0.003%
4 disk accesses and over	0.000%	0.000%	0.000%	0.000%	0.005%
0,1,2 disk accesses	100.000%	99.999%	99.998%	99.998%	99.992%

Table B.15: Proportional to occurrences (*OED*)

under three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.14 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.15 is for querying a phrase as many times as it appears in the *OED*. Table B.16 is obtained by using the DeFazio distribution to query indexed phrases. The three results are quite close. 87%, 94%, 97%, 98% and 98% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.17 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.18 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	74.476%	39.402%	21.314%	11.556%	5.498%
1 disk access	22.253%	55.971%	75.887%	86.255%	92.473%
2 disk accesses	3.271%	4.625%	2.796%	2.187%	2.025%
3 disk accesses	0.000%	0.002%	0.002%	0.002%	0.004%
4 disk accesses and over	0.000%	0.000%	0.001%	0.000%	0.000%
0,1,2 disk accesses	100.000%	99.998%	99.997%	99.998%	99.996%

Table B.16: Under the DeFazio distribution (*OED*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	65.029%	60.988%	88.532%	95.943%	97.543%
1 disk access	7.143%	25.274%	10.602%	3.746%	2.182%
2 disk accesses	27.828%	13.737%	0.866%	0.310%	0.275%
0,1,2 disk accesses	100.000%	99.999%	100.000%	99.999%	100.000 %

Table B.17: Uniform i-word signatures (*OED*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	65.029%	69.087%	89.521%	94.329%	94.540%
1 disk access	7.143%	18.587%	9.688%	5.302%	5.107%
2 disk accesses	27.828%	12.326%	0.791%	0.369%	0.353%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.18: Uniform word signatures (*OED*)

B.4 *WWW1*

Phrase signatures are structured from the first 5 words of phrases. Table B.19 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	4.73	7.83	7.06	5.60	4.73	29.95
adjacent collisions	984	1072	1459	1628	1490	6633
breaking points	919	939	456	303	245	2862
guaranteeing phrases	1	23	15	6	5	50

Table B.19: Signatures and the look-aside table (*WWW1*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	7.305%	2.503%	1.769%	1.587%	1.518%
1 disk access	81.003%	92.943%	96.725%	97.744%	98.065%
2 disk accesses	11.690%	4.547%	1.503%	0.667%	0.416%
3 disk accesses	0.000%	0.007%	0.003%	0.001%	0.001%
4 disk accesses and over	0.002%	0.000%	0.000%	0.001%	0.000%
0,1,2 disk accesses	99.998%	99.993%	99.997%	99.998%	99.999%

Table B.20: Uniform phrases (*WWW1*)

Tables B.20, B.21, and B.22 describe search performances for indexed phrases

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	69.651%	25.239%	10.377%	5.166%	2.323%
1 disk access	27.858%	71.397%	88.066%	94.100%	97.111%
2 disk accesses	2.489%	3.360%	1.551%	0.733%	0.564%
3 disk accesses	0.001%	0.004%	0.005%	0.001%	0.000%
4 disk accesses and over	0.001%	0.000%	0.001%	0.000%	0.002%
0,1,2 disk accesses	99.998%	99.996%	99.994%	99.999%	99.998%

Table B.21: Proportional to occurrences (*WWW1*)

under three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.20 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.21 is for querying a phrase as many times as it appears in the *WWW1*. Table B.22 is obtained by using the DeFazio distribution to query indexed phrases. The three results are quite close. 88%, 95%, 98%, 99% and 99% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.23 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.24 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	27.703%	2.432%	1.481%	1.493%	1.587%
1 disk access	63.266%	92.414%	97.012%	97.830%	98.032%
2 disk accesses	9.030%	5.148%	1.505%	0.676%	0.380%
3 disk accesses	0.000%	0.006%	0.002%	0.001%	0.001%
4 disk accesses and over	0.001%	0.000%	0.000%	0.000%	0.000%
0,1,2 disk accesses	99.999%	99.994%	99.998%	99.999%	99.999%

Table B.22: Under the DeFazio distribution (*WWW1*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	15.495%	64.429%	98.629%	99.968%	99.999%
1 disk access	17.281%	27.832%	1.325%	0.032%	0.001%
2 disk accesses	67.224%	7.739%	0.046%	0.000%	0.000%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.23: Uniform i-word signatures (*WWW1*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	15.495%	83.261%	98.652%	97.288%	95.727%
1 disk access	17.281%	10.818%	1.324%	2.693%	4.257%
2 disk accesses	67.224%	5.921%	0.024%	0.019%	0.016%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.24: Uniform word signatures (*WWW1*)

B.5 *WWW2*

Phrase signatures are structured from the first 5 words of phrases. Table B.25 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	4.10	7.82	7.29	5.74	4.66	29.61
adjacent collisions	6814	8160	12242	13822	10656	51694
breaking points	6516	10150	4997	2552	1583	25798
guaranteeing phrases	1	134	95	68	93	391

Table B.25: Signatures and the look-aside table (*WWW2*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	6.897%	2.382%	1.592%	1.423%	1.369%
1 disk access	80.902%	92.247%	96.664%	97.911%	98.273%
2 disk accesses	12.201%	5.366%	1.742%	0.665%	0.357%
3 disk accesses	0.000%	0.005%	0.002%	0.001%	0.001%
4 disk accesses and over	0.000%	0.000%	0.000%	0.000%	0.000%
0,1,2 disk accesses	100.000%	99.995%	99.998%	99.999%	99.999%

Table B.26: Uniform phrases (*WWW2*)

Tables B.26, B.27, and B.28 describe search performances for indexed phrases

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	79.936%	29.342%	11.438%	6.742%	4.392%
1 disk access	18.294%	66.892%	86.855%	92.544%	95.190%
2 disk accesses	1.770%	3.763%	1.705%	0.711%	0.416%
3 disk accesses	0.000%	0.003%	0.002%	0.001%	0.001%
4 disk accesses and over	0.000%	0.000%	0.000%	0.002%	0.001%
0,1,2 disk accesses	100.000%	99.997%	99.998%	99.997%	99.998%

Table B.27: Proportional to occurrences (*WWW2*)

under three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.26 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.27 is for querying a phrase as many times as it appears in the *WWW2*. Table B.28 is obtained by using the DeFazio distribution to query indexed phrases. The three results are quite close. 88%, 94%, 98%, 99% and 99.6% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.29 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.30 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	40.168%	5.482%	3.037%	2.518%	1.960%
1 disk access	52.306%	88.799%	95.140%	96.781%	97.639%
2 disk accesses	7.526%	5.714%	1.821%	0.700%	0.399%
3 disk accesses	0.000%	0.004%	0.002%	0.000%	0.001%
4 disk accesses and over	0.000%	0.001%	0.000%	0.001%	0.001%
0,1,2 disk accesses	100.000%	99.995%	99.998%	99.999%	99.998%

Table B.28: Under the DeFazio distribution (*WWW2*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	26.518%	61.923%	97.852%	99.659%	99.785%
1 disk access	16.163%	28.274%	2.046%	0.308%	0.204%
2 disk accesses	57.319%	9.802%	0.102%	0.033%	0.011%
0,1,2 disk accesses	100.000%	99.999%	100.000%	100.000%	100.000%

Table B.29: Uniform i-word signatures (*WWW2*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	26.518%	75.714%	98.303%	97.514%	95.465%
1 disk access	16.163%	15.999%	1.624%	2.439%	4.509%
2 disk accesses	57.319%	8.287%	0.073%	0.047%	0.025%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	99.999%

Table B.30: Uniform word signatures (*WWW2*)

B.6 *WWW3*

Phrase signatures are structured from the first 5 words of phrases. Table B.31 describes the signature array and the look-aside table. It gives the average number of bits assigned for word signatures, the total number of adjacent collisions, the total number of breaking points for i-word phrases, and the total number of guaranteeing phrases of 2 disk accesses.

	1-word	2-words	3-words	4-words	5-words	total
bits in word signatures	2.42	7.12	8.01	6.67	5.26	29.48
adjacent collisions	21201	29240	39248	53932	54899	198520
breaking points	12707	39306	23712	15038	9185	99948
guaranteeing phrases	34	410	471	327	281	1523

Table B.31: Signatures and the look-aside table (*WWW3*)

	1-word	2-words	3-words	4 words	5-words
0 disk accesses	10.370%	3.027%	1.672%	1.339%	1.234%
1 disk access	76.283%	90.018%	95.885%	97.637%	98.271%
2 disk accesses	13.342%	6.949%	2.440%	1.023%	0.494%
3 disk accesses	0.000%	0.006%	0.003%	0.001%	0.001%
4 disk accesses and over	0.005%	0.000%	0.000%	0.000%	0.000%
0,1,2 disk accesses	99.995%	99.994%	99.997%	99.999%	99.999%

Table B.32: Uniform phrases (*WWW3*)

Tables B.32, B.33, and B.34 describe search performances for indexed phrases

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	92.344%	45.904%	15.074%	5.233%	1.992%
1 disk access	6.935%	50.718%	82.617%	93.606%	97.369%
2 disk accesses	0.720%	3.376%	2.306%	1.159%	0.638%
3 disk accesses	0.000%	0.002%	0.003%	0.002%	0.001%
4 disk accesses and over	0.001%	0.000%	0.000%	0.000%	0.000%
0,1,2 disk accesses	99.999%	99.998%	99.997%	99.998%	99.999%

Table B.33: Proportional to occurrences (*WWW3*)

under three different query distributions. They list percentages of i-word phrases that are found by 0 disk accesses (by looking up the look-aside table), 1, 2, 3, 4 and more disk accesses. Table B.32 is the result of each indexed distinct i-word phrase being queried exactly once. Table B.33 is for querying a phrase as many times as it appears in the *WWW3*. Table B.34 is obtained by using the DeFazio distribution to query indexed phrases. The three results are quite close. 87%, 93%, 97%, 99% and 99% of queries of 1, 2, 3, 4 and 5-word phrases respectively could be done by 0 or 1 disk access.

Table B.35 is the worst case search performance of unsuccessful i-word searches under the assumption that all the i-word phrase signatures are queried with the same probability, as described under "Unsuccessful searches" in Section 5.2.4. Table B.36 is the expected worst case search performance of the unsuccessful ith word searches, of all existing $(i-1)$-word prefixes.

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	65.194%	10.505%	2.541%	1.872%	1.588%
1 disk access	30.148%	82.230%	94.827%	97.046%	97.888%
2 disk accesses	4.657%	7.260%	2.630%	1.080%	0.524%
3 disk accesses	0.000%	0.004%	0.002%	0.001%	0.000%
4 disk accesses and over	0.001%	0.001%	0.000%	0.001%	0.000%
0,1,2 disk accesses	99.999%	99.995%	99.998%	99.998%	100.000%

Table B.34: Under the DeFazio distribution (*WWW3*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	43.667%	43.239%	94.651%	99.701%	99.855%
1 disk access	11.119%	37.688%	5.066%	0.286%	0.133%
2 disk accesses	45.214%	19.073%	0.283%	0.013%	0.012%
0,1,2 disk accesses	100.000%	100.000%	100.000%	100.000%	100.000%

Table B.35: Uniform i-word signatures (*WWW3*)

	1-word	2-words	3-words	4-words	5-words
0 disk accesses	43.667%	59.619%	96.591%	98.221%	96.553%
1 disk access	11.119%	24.137%	3.229%	1.749%	3.418%
2 disk accesses	45.214%	16.244%	0.180%	0.029%	0.029%
0,1,2 disk accesses	100.000%	100.000%	100.000%	99.999%	100.000%

Table B.36: Uniform word signatures (*WWW3*)

Bibliography

[Aho90] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume A*, chapter 5, pages 255–300. Elsevier and MIT Press, 1990.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[BU77] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[BY89] R. A. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Department of Computer Science, University of Waterloo, 1989.

[Cla96] D. Clark. *Compact Pat Trees*. PhD thesis, Department of Computer Science, University of Waterloo, October 1996.

[DeF93] S. DeFazio. Overview of the full-text document retrieval benchmark. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 435–487. M. Kaufmann Publishers, second edition, 1993.

[dlB59] R. de la Briandais. File searching using variable length keys. In *AFIPS Western JCC*, pages 295–298, San Francisco, Calif., 1959.

[Fal85] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, March 1985.

[Fal88] C. Faloutsos. Signature files: An integrated access method for text and attributes, suitable for optical disk storage. *Bit*, 28:736–754, 1988.

[Fal92] C. Faloutsos. Signature files. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 44–65. Prentice Hall, 1992.

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on office information systems*, 2(4):267–288, October 1984.

[Fel50] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, Inc., third edition, 1950.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing–a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[Fre60] E. Fredkin. Trie memory. *C.ACM*, 3:490–499, 1960.

[FS86] P. Flajolet and R. Sedgewick. Digital search trees revisited. *SIAM J. on Computing*. 15:748–767. 1986.

[GBY91] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison Wesley. second edition. 1991.

[GBYS92] 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. A. Baeza-Yates. editors. *Information Retrieval: Data Structures & Algorithms*. pages 66–82. Prentice Hall. 1992.

[GL82] G. H. Gonnet and P.-Å. Larson. External hashing with limited internal storage. *Proceedings of the ACM Symposium on Principles of Database Systems*. pages 256–261. 1982.

[Gon83] G. H. Gonnet. Unstructured data bases or very efficient text searching. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium Principles of Database System*. pages 117–124. March 1983.

[Gon88] G. H. Gonnet. Efficient searching of text and pictures (extended abstract). 1988. UW Center for the New OED and Text Research. University of Waterloo. Technical Report OED-88-02.

[HFBYL92] D. Harman. E. Fox. R. A. Baeza-Yates. and W. Lee. Inverted files. In W. B. Frakes and R. A. Baeza-Yates. editors. *Information Retrieval: Data Structures & Algorithms*. pages 28–43. Prentice Hall. 1992.

[KMP77] D. E. Knuth. J. H. Morris. and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*. 6(2):323–350. June 1977.

[Kno75] G. D. Knott. Hashing functions. *The Computer Journal*. 18(3):265–278. 1975.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. volume 3. Addison-Wesley. 1973.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithm. *IBM J Res. Development*. 31(2):249–260. March 1987.

[Li96] S. Li. Using Inverted Lists to Match Structured Text. Technical report. Department of Computer Science. University of Waterloo. 1996.

[LK84] P.-Å. Larson and A. Kajla. File organization: implementation of a method guaranteeing retrieval in one disk access. *Commun. ACM*. 27(7):670–677. July 1984.

[MBY91] U. Manber and R. A. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*. 37(3):113-136. February 1991.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*. 22(5):935–948. October 1993.

[Mor68] D. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *J.ACM*. 15:514–534. 1968.

[MW93] U. Manber and S Wu. Glimpse: A tool to search through entire file systmes. Technical report. Department of Computer Science. University of Arizona. October 1993.

[Sha95] H. Shang. *Trie Methods for Text and Spatial Data on Secondary Storage.* PhD thesis. Department of Computer Science. University of Helsinki. January 1995.

[ST93] A. Salminen and F. Wm. Tompa. Pat expressions: an algebra for text search. *Acta Linguistica Hungarica.* 41(1-4):277–306. 1992-1993.

[Swo81] E. W. Swokowski. *Calculus.* Prindle. Weber & Schmidt. 1981.

[Tom91] F. Wm. Tompa. An overview of Waterloo's database software for the OED. In *Proc. Symp. on Historical Dictionary Databases and Data Retrieval Requirements.* pages 123–143. Toronto. October 1991. in CCH (Centre for Computing in the Humanities) Working Papers 2 (1992).

[WK] T. Williams and C. Kelley. *Gnuplot.* an interactive plotting program.

[WMB94] I. H. Witten. A. Moffat. and T. C. Bell. *Managing Gigabytes.* Van Nostrand Reinhold. 1994.

[YM91] O. Yasushi and I. Masajirou. A new character-based indexing method using frequency data for Japanese documents. In *ACM SIGIR 95.* pages 261–275. December 1991.