The Suffix-Signature Method for Searching for Phrases in Text

Mei Zhou
Open Text Corporation
185 Columbia St. W.
Waterloo Ont. N2L 5Z5
Canada

Frank W. Tompa
Department of Computer Science
University of Waterloo
Waterloo Ont. N2L 3G1
Canada

February 15, 1998

Abstract

Dealing effectively with large volumes of semistructured data requires efficient retrieval. 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 paper, 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. The experiments with actual data ranging in size from 6Mb to 550Mb and actual query patterns derived from logs of searches on the World Wide Web show that the approach is applicable in practice to a variety of texts and realistic phrase searches.

In Section 1, we review text search techniques. In Section 2, we briefly describe the proposed method. We present theoretical results in Section 3 and some experimental results in Section 4. We compare the proposed method with some other methods in Section 5 and summerize in Section 6.

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 large, 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 algorithms [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 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].

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].

We will use the following assumptions in this paper:

- *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

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

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.

Note that these assumptions are valid for many semistructured databases.

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 a very "short" signature, 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 paper 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.

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.

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...", "dog, the cat...", *etc*).

In the proposed method, there is a *signature array* in addition to the *text*, a *suffix array*, and a *block list*, as illustrated in Figure 1. 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. 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 conventional 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, 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.

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.

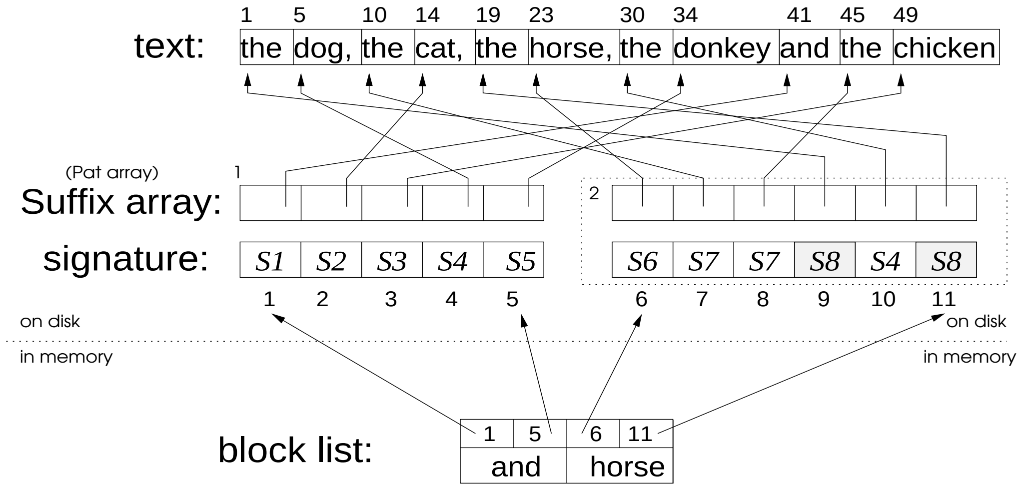

Figure 1: Suffix-Signature Method

Assume that we search for the phrase p=“the horse, the donkey and the chicken”, which we will find starting at the 19th character in the text of Figure 1; 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 2.2, and how to bound the number of disk accesses for searching for a phrase in Sections 2.3 and 2.4. We give a phrase search algorithm in Section 2.5.

2.2 Phrase signatures

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 from that of the complete phrase. 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 suffix array accommodates 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*:

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 $1 \leq i \leq n$.*

A stronger prefix property is as follows.

Strong Prefix Property: *The signatures of prefixes of a phrase P are prefixes of the signature of P.*

We use the concatenation of word signatures as our signature for phrases. This is described in more detail in Section 3.

2.3 Separate collisions

Assume that we search for the shaded phrase of Figure 2 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. Eight such disk accesses are needed to get the shaded phrase in our example of Figure 2.

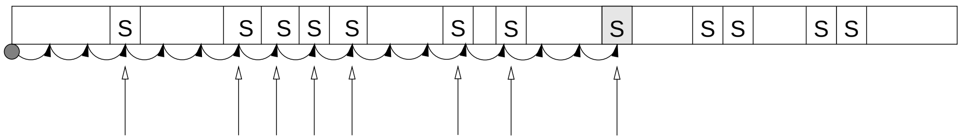

Figure 2: Search for Matching Signatures

Recall, however, that the suffix array references phrases in lexicographical order. 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 3, 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.

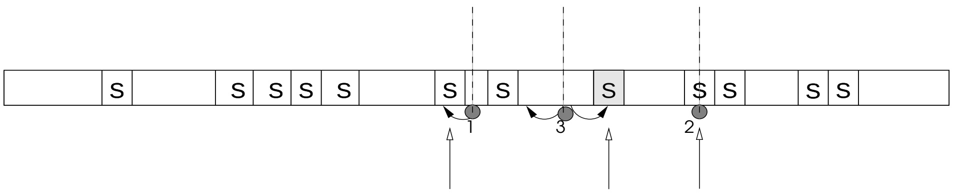

Figure 3: Start from the middle

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, 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 4. 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.

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.

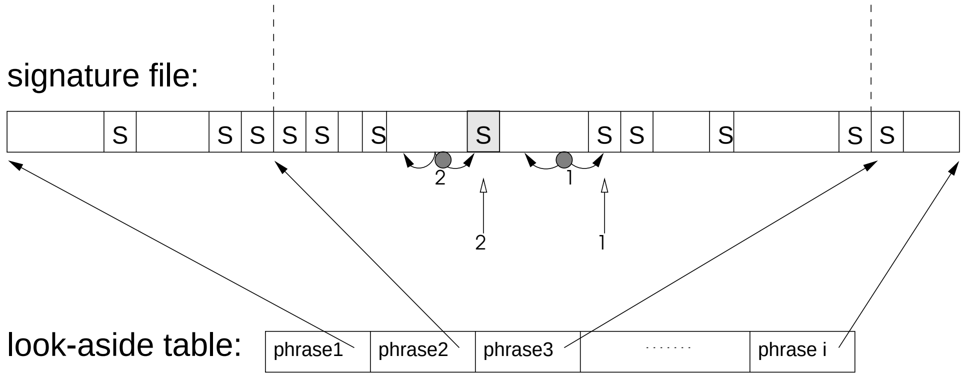

Figure 4: Look-aside Table

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 paper.

2.5 Phrase search algorithm

Given a text, at index time we create its suffix array and divide the suffix array into blocks to create a block list. For each block we then create 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. 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, continue with 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 ith 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 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.

3 Properties of phrase signatures and the look-aside table

We study the concatenation signature scheme in this section. We first define the notation:

- 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 5. 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.

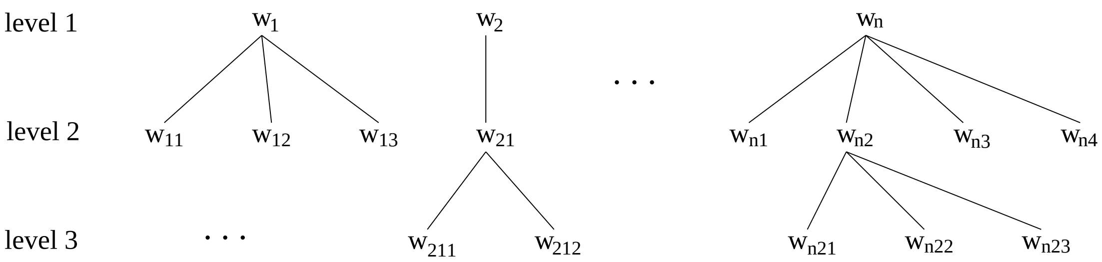

Figure 5: Levels of phrases

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.

3.1 Concatenation scheme

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

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

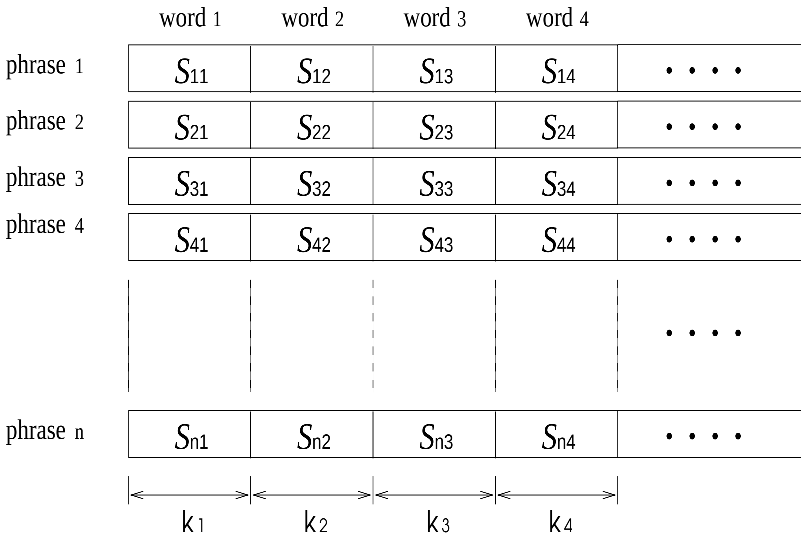

Figure 6: Signature Structure

A signature array of n phrases is shown in Figure 6, in which S_{ij} denotes $Sig_j(w_{ij})$ for phrase i.

Word signatures are basic units on which phrase and prefix signatures are built. Word signatures can be created using any appropriate hashing functions. Assume that there are k_i bits for $Sig_i(w_i)$ $(1 \leq i)$, and that $P_i = w_1 w_2 \dots w_i$, $(1 \leq i)$.

- 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.

3.2 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 from a random model for a 7-bit signature. 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 $100*(14+5*4.5*8+4*8)/10,000 = 2.26$ 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.

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.

Theorem *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. If the hashing function produces word signatures with uniform probability, 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

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

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

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.

Theorem *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 the previous theorem, 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

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

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. □

Corollary 1 *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 the above theorem 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 2 *Under the conditions in Corollary 1, the expected number of adjacent collisions in a block having n_i distinct i-word phrases for $1 \leq i \leq t$ is tC.*

In our experience, based on theoretical models of text as well as several large texts (such as the 550Mb *Oxford English Dictionary*) and large text collections (such as six months of news stories from the *Ottawa Citizen* and a 200Mb extraction from the World Wide Web), for 5 word phrases, tC is approximately $60 - 70$ for a block of $10,000$ indexed phrases. Thus the look-aside table to store adjacent collisions remains small enough to make the suffix-signature method practical. Experimental results are presented in the next section.

4 Experiments

Experiments were conducted to validate the suffix-signature method with respect to both space and searching speed on 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.

4.1 Prototype system

Figure 7 illustrates the hierarchy of indices. The *suffix array* and the *signature array* are divided into fixed sized blocks. Block addresses and corresponding boundary 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.

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*.

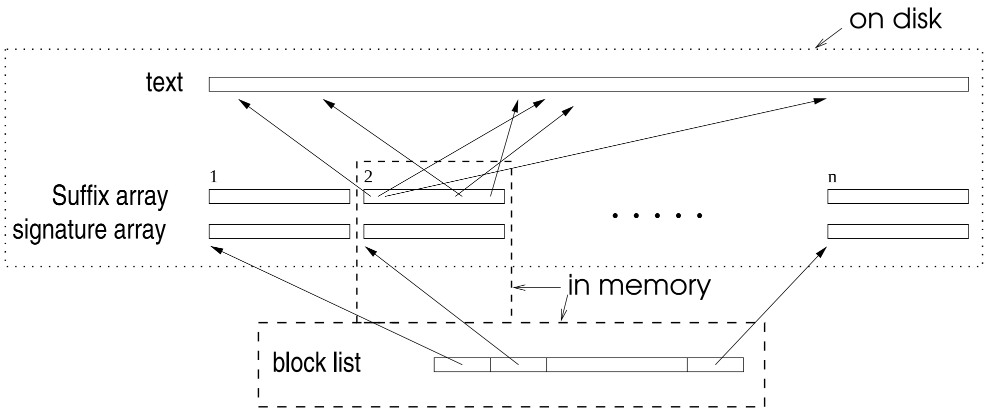

Figure 7: Hierarchy of Indices

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.

4.2 Design of experiments

The set of data sources used in our experiments is described in Appendix A. 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.

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 in such a way that the number of total adjacent collisions in a block is minimum (see Section 3.2). Thus the size of the look-aside table is nearly minimized.

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

Recall that adjacent collisions seem to subdivide the interval to be searched into smaller units. If too many phrases occur between two successive adjacent collisions, we insert additional phrases (breaking points) into the look-aside table to speed up searches without costing too much space.

Phrases that could not be found within the first two disk accesses to a text are also stored in the look-aside table. 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-*OR*s 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.

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. We will 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). We will also measure space usage for each text in our experiments.

4.3 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 (see Appendix C).

Unsuccessful searches

Searches for unindexed phrases are unsuccessful searches. We do not actually search for unindexed phrases in most of our experiments. Instead,

	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 1: Signature bits per phrase for various data sources

we assume every phrase *signature* is equally likely to be requested since an arbitrary phrase has an equal probability of having each signature. If the signature of an i-word phrase is not in the applicable segment of the signature array, no disk access to the text is needed to confirm that a phrase having that signature does not appear. Similarly, if the signature appears once, confirming that the phrase does not occur will require one access to the text. For simplicity, if the signature appears more than once, we assume that at least two disk accesses will be required. However, because of the presence of guaranteeing phrases in the look-aside table, no successful query requires more than two disk accesses, and thus confirmation of non-occurrence requires at most two disk accesses as well.

4.4 Experimental results

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.

Table 1 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 2 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.

Table 3 gives a more detailed space usage for the data source *WWW3*. The number of bits in a phrase signature is 29.48 on average. The simple

	# 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 2: Total space requirements

compression method described in Section 4.2 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 3: Space (*WWW3*)

Speed

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 (described in Section 4.3). 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.

We first consider the *News* data and report the execution times for index building and searching. About 800Kb of memory was required for data structures used by *buildindex* which took an elapsed time of 155 minutes on a shared DEC *alpha* 3000/50S machine. 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.

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 4.

	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 4: Search examples in News

A second set of queries, *WWWQ*(see Appendix B), contains a trace of 2.7 million phrase search queries posed to the Open Text Index against the World Wide Web in the first two months of 1997. Query phrases in *WWWQ* were searched against the *WWW1*, *WWW2* and *WWW3* databases by using the suffix-signature method. Table 5 summarizes the search performance for both successful searches and unsuccessful searches for the *WWW3*.

For successful searches, the table lists 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. Note that since phrases longer than 5 words do not have stored signatures, these might require two disk accesses per 5-word segment.

Also note that in the experiments, the average number of disk accesses to a text for a successful search for phrases up to 5 words in length 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.

	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: *WWWQ* searches on (*WWW3*)

5 Comparison to other structures

The proposed suffix-signature method is very efficient for phrase searches (see [Zho97] for more details). 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 $O(\log n)$ [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 2), 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, 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), but faster for phrase searches. It uses more space than an inverted list.

Inverting five word phrases

Since the number of distinct five word phrases is comparable to the number of words in the text, an 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.

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 [WMB94]. 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.

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 [Zho97] 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. Unsuccessful searches are on average much faster for the suffix-signature method than for a Compact Pat tree.

Superimposition scheme

In the superimposition scheme, word signatures and phrase signatures are of the same length. 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 [Fal92, Zho97].

Similarly to the concatenation scheme, extra space is needed to store the boundaries of adjacent collisions. 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, 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, 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 conducted experiments using the concatenation scheme and the superimposition scheme for various combinations of bits allocated or set for word signatures on some blocks of data from the *OED* (see Appendix A). We calculated the theoretical and the experimental numbers of adjacent collisions. Our experiments indicate that the superimposition scheme has about 3 to 4 times as many adjacent collisions as the concatenation scheme [Zho97].

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.

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.

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[2] 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

[2]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.

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.

6 Summary

In this paper, we presented a suffix-signature method for searching for phrases in large static texts. We described how the suffix-signature method works, and investigated 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 presented some theoretical analyses and reported the results of experiments. Further details can be found elsewhere [Zho97].

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.

The suffix-signature method is useful for searching in semistructured data. For instance, we want to search for certain patterns in DNA sequences or musical note sequences.

The suffix-signature method might be particularly useful in languages, such as Japanese or Chinese. 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. 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.

A Experiment data

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*.

Bible is an SGML encoded version of the King James Version of the *Bible* – the Old Testament, Apocrypha, and the New Testament. This is about 5.6Mb and has approximately 1.1M indexed phrases.

News is an SGML version of the *Ottawa Citizen* newspaper from July 1990 to December 1990. It is roughly 84.7Mb and has about 15.3M indexed phrases.

The *OED* is a tagged version of the *Oxford English Dictionary, second edition*. It is approximately 545.6Mb, and has around 1.5M distinct words and about 109.8M indexed phrases.

WWW1, *WWW2*, and *WWW3* are collections of *HTML* pages from the *World Wide Web*. They were taken from different portions of the pages. 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.

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 8: Experimental corpus

B WWW queries

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 6 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 6: 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 6 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 7 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 7: Distribution of search lengths in *WWWQ*

C DeFazio distribution

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.

References

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Swo81] E. W. Swokowski. *Calculus*. Prindle, Weber & Schmidt, 1981.

[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.

[Zho97] M. Zhou. The suffix-Signature Method for Searching for Phrases in Text. Technical report, Department of Computer Science, University of Waterloo, 1997.