On the General Signature Trees

Yangjun Chen
Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
ychen2@uwinnipeg.ca

Abstract The signature file method is a popular indexing technique used in information retrieval and databases. It excels in efficient index maintenance and lower space overhead. Different approaches for organizing signature files have been proposed, such as sequential signature files, bit-slice files, S-trees, and its different variants, as well as signature trees. In this paper, we extends the structure of signature trees by introducing multiple-bit checkings. That is, during the searching of a signature tree against a query signature s_q, more than one bit in s_q will be checked each time when a node is encountered. This does not only reduce significantly the size of a signature tree, but also increases the filtering ability of the signature tree. We call such a structure a *general signature tree*. Experiments have been made, showing that the general signature tree uniformly outperforms the signature tree approach.

Keywords: *index*, *signature file*, *signature identifier*, *signature tree*, *information retrieval*

1. Introduction

An important question in information retrieval is how to create a database index which can be searched efficiently for the data one seeks. Today, one or more of the following techniques have been frequently used: full text searching, B-trees [3], inversion [14, 23] and the signature file [11, 12, 17]. Full text searching imposes no space overhead, but requires long response time. In contrast, B-trees, inversion and the signature file work quickly, but need a large intermediary representation structure (index), which provides direct links to relevant data. In this paper, we concentrate on the techniques of signature files and discuss a new approach for organizing signature files.

The signature file method was originally introduced as a text indexing methodology [11, 12]. Nowadays, however, it is utilized in a wide range of applications, such as office filing [7], hypertext systems [13], relational and object-oriented databases [6, 15, 18, 22], as well as data mining [1]. In comparison with the other index structures, it has mainly the following advantages:

- it can be used to efficiently evaluate set-oriented queries;
- it can handle insertion and update operations easily.

A typical query processing with the signature file is as follows: when a query is given, a query signature (a bit string) is formed from the query values. Then each signature in the signature file is examined over the query signature. If a signature in the file covers the query signature, the corresponding data object becomes a candidate that may satisfy the query. Such an object is called a drop. The next step of the query processing is the false drop resolution. Each drop is accessed and examined whether it actually satisfies the query condition. Drops that fail the test are called false drops while the qualified data objects are called actual drops.

Different approaches for organizing signature files have been proposed, such as sequential signature files, bit-slice files [15], S-trees [9], and its different variants [20, 21], as well as signature trees. In this paper, we introduce a new way to organize signature files

The author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

by extending the structure of signature trees. Instead of checking only one bit in the query signature each time when a node is encountered during the searching of a signature tree, multiple bits will be checked. This enables us both to

(i) decrease the size of a signature tree, and

(ii)increase the filtering ability of a signature tree.

Experiments are made, which show that the general signature tree is really beneficial in comparison with the signature tree approach.

The remainder of the paper is organized as follows. In Section 2, we show what is a signature file and what is a signature tree. In Section 3, we introduce the structure of general signature trees and discuss how they can be constructed. Section 4 is devoted to the maintenance of general signature trees. In Section 5, we report the experiment results. Finally, Section 6 is a short conclusion.

2. Signature files and signature trees

Intuitively, a signature file can be considered as a set of bit strings, which are called signatures. Compared to the inverted index, the signature file is more efficient in handling new insertions and queries on parts of words; and especially suitable for set-oriented query evaluation. But the scheme introduces information loss. More specifically, its output usually involves a number of false drops, which may be identified only by means of a full text scanning on every text block short-listed in the output. Also, for each query processed, the entire signature file needs to be searched [11, 12]. Consequently, the signature file method involves high processing and I/O cost. This problem is mitigated by partitioning a signature file, by introducing an auxiliary data structure, as well as by exploiting parallel computer architectures [8].

2.1 Signature files

Signature files are based on the inexact filter. They provide a quick test, which discards many of the nonqualifying elements. But the qualifying elements definitely pass the test although some elements which actually do not satisfy the search requirement may also pass it accidentally, *i.e*., there may exist “false hits” or “false drops” [11, 12]. In an object-oriented database, for instance, an object is represented by a set of attribute values. The signature of an attribute value is a hash-coded bit string of length *m* with *k* bits set to “1”. As an example, assume that we have an attribute value “professor”. Its signature can be constructed as follows. In terms of [4], the letter triplets in a word (or an attribute value) are the best choice for information carrying text segments in the construction of the signature for that word. So we decompose “professor” into a series of triplets: “pro,” “rof,” “ofe,” “fes,” “ess,” and “sor.” Using a hash function *hash*, we will map a triplet to an integer *p* indicating that the *p*th bit in the string will be set to 1. For example, assume that we have *hash*(pro) = 2, *hash*(rof) = 4, *hash*(ofe) = 8, and *hash*(fes) = 9. Then, we will establish a bit string: 010 100 011 000 for “professor” as its word signature (see [10] for a detailed discussion.) An object signature is formed by superimposing the signatures for all its attribute values. (By ‘superimposing’, we mean a bit-wise OR operation.) Object signatures of a class will be stored sequentially in a file, called a *signature file*. Fig. 1 depicts the signature generation and comparison process of an object having three attribute values: “John”, “12345678”, and “professor”.

object:	John	12345678	professor

attribute signature:		
John		010 000 100 110
12345678		100 010 010 100
professor	$\vee$	010 100 011 000
object signature (OS)		110 110 111 110

queries:	query signatures:	matching results:
John	010 000 100 110	match with OS
Paul	011 000 100 100	no match with OS
11223344	110 100 100 000	false drop

Fig. 1. Signature generation and comparison

When a query arrives, the object signatures are scanned and many nonqualifying objects are discarded. The rest are either checked (so that the “false drops” are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for attribute values. The query signature is then compared to every object signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 1: (1) the object matches the query; that is, for every bit set in s_q, the corresponding bit in the object signature s is also set (i.e., $s \wedge s_q = s_q$) and the object contains really the query word; (2) the object doesn't match the query (i.e., $s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the object in fact doesn't match the search criteria (false drop). In order to eliminate false drops, the object must be examined after the object signature signifies a successful match.

In addition, we can see that the signature matching is a kind of inexact matching. That is, s_q matches a signature s if for any bit set to 1 in s_q, the corresponding bit in s is also set to 1. However, for any bit set to 0 in s_q, it doesn't matter whether the corresponding bit in s is set to 1 or 0.

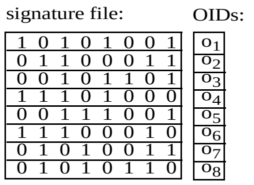
The purpose of using a signature file is to screen out most of the nonqualifying objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object access is prevented.

To determine the size of a signature file, we use the following formula [4]:

$$m \times \ln 2 = k \times D,$$

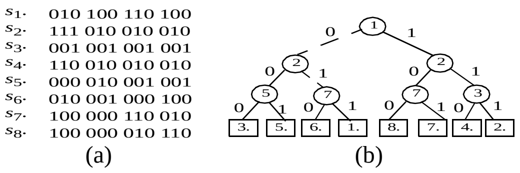
where D is the average size of a block. (In a relational or an object-oriented database, D can be considered to be the average number of attributes in a tuple or in an object.)

In a signature file, a set of signatures is sequentially stored, which is easy to implement and requires low storage space and low update cost. However, when a query is given, a full scan of the signature file is required. Therefore, it is generally slow in retrieval. Fig. 2 is a quite simple signature file. If more than one objects share the same signature, that



signature file:	OIDs:
1 0 1 0 1 0 0 1	o_1
0 1 1 0 0 0 1 1	o_2
0 0 1 0 1 1 0 1	o_3
1 1 1 0 1 0 0 0	o_4
0 0 1 1 1 0 0 1	o_5
1 1 1 0 0 0 1 0	o_6
0 1 0 1 0 0 1 1	o_7
0 1 0 1 0 1 1 0	o_8

Fig. 2. Illustration of sequential



s_1.	010 100 110 100
s_2.	111 010 010 010
s_3.	001 001 001 001
s_4.	110 010 010 010
s_5.	000 010 001 001
s_6.	010 001 000 100
s_7.	100 000 110 010
s_8.	100 000 010 110

Fig. 3. A signature file and its signature tree

signature will be associated with the identifiers of all those objects.

2.2 Signature trees

In [5], a new method was proposed to organize signature files to speed up a signature file scanning. Using this method, a tree over a signature file S, called a signature tree, is constructed with the following properties.

(1) Each node v is associated with a number (denoted $skip(v)$) to tell which bit in s_q to check when v is encountered during the tree searching.

(2) For each node, its left outgoing edge is labeled with 0 and its right outgoing edge is labeled with 1.

(3) Each path from the root to a leaf represents a signature identifier that uniquely identifies a signature in S just as a *position identifier* used to identify a substring [2]. A signature identifier is defined as follows. Let $S = s_1.s_2 \ldots .s_n$ denote a signature file. Let $s_i[j]$ represent the jth bit in s_i. The signature identifier for an s_i is a sequence of pairs: $(j_1, s_i[j_1])(j_2, s_i[j_2])\ldots(j_h, s_i[j_h])$ $(1 \leq j_k \leq m$; denoted $s_i(j_1, \ldots, j_h))$ such that for any $k \neq i$ $(1 \leq k \leq n)$ we have $s_i(j_1, \ldots, j_h) \neq s_k(j_1, \ldots, j_h)$.

For example, the tree shown in Fig. 3(b) is a signature tree for the signature file shown in Fig. 3(a).

In the tree shown in Fig. 3(b), each path represents an identifier for some signature. For instance, the path from the root to the leaf labeled with s_6 (see the dashed line) represents the signature identifier for s_6. It is because $s_6(1, 2, 7) = (1, 0)(2, 1)(7, 0)$ and for any $i \neq 6$ we have $s_i(1, 2, 7) \neq s_6(1, 2, 7)$.

In addition, we point out that this signature tree is constructed using an algorithm different from that discussed in [5], which generates a signature tree for a signature file like a *Pat-tree* for a long bit string [16, 19] and needs $O(n \cdot min(m, \log n))$ time. However, the algorithm used to generate the tree shown in Fig. 3(b) needs $O(n \cdot m \cdot \log n)$ time, worse than the algorithm proposed in [5]. But it can create a more balanced tree. Below is the formal description of this algorithm, in which we consider a signature file $S = s_1.s_2 \ldots .s_n$ as a boolean matrix and use $S[i]$ to represent the ith column of S.

```
Algorithm balanced-tree-generation(file)
input: a signature file.
output: a signature tree.
begin
let $S = file$; $n \leftarrow |S|$;
if $n > 1$ then {
   choose $j$ such that $|w(S[j]) - \frac{1}{2}n|$ is minimum;
   let $g_1 = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ with each $s_{i_l}[j] = 0$ $(l = 1, \ldots, k)$;
   let $g_2 = \{s_{i_{k+1}}, s_{i_{k+2}}, \ldots, s_{i_n}\}$ with each $s_{i_h}[j] = 1$ $(h = k + 1, \ldots, n)$
   generate a tree containing a root $r$ and two child nodes marked with $g_1$ and $g_2$, respectively;
   $skip(r) \leftarrow j$;
   replace the node marked $g_1$ with balanced-tree-generation($g_1$);
   replace the node marked $g_2$ with balanced-tree-generation($g_2$);}
else return;
end
```

The idea of the algorithm is simple. First, we calculate the weight of each $S[i]$, *i.e.*, the number of 1s appearing in $S[i]$, denoted $w(S[i])$. This needs $O(n \cdot m)$ time. Then, we choose an j such that $|w(S[i]) - \frac{1}{2}n|$ is minimum. Here, the tie is resolved arbitrarily. Using this j, we divide S into two groups $g_1 = \{s_{i_1}, s_2, \ldots, s_{i_k}\}$ with each $s_{i_l}[j] = 0$ $(l = 1, \ldots, k)$ and $g_2 = \{s_{i_{k+1}}, s_{i_{k+2}}, \ldots, s_{i_n}\}$ with each $s_{i_h}[j] = 1$ $(h = k + 1, \ldots, n)$; and generate a tree as shown in Fig. 4(a). In a next step, we consider each g_i $(i = 1, 2)$ as a single signature file and perform the same operations as above, leading to two trees generated for g_1 and g_2, respectively. Replacing g_1 and g_2 with the corresponding trees, we get another tree as illustrated in Fig. 4(b). We repeat this process until the leaf nodes of a generated tree cannot be divided any more.

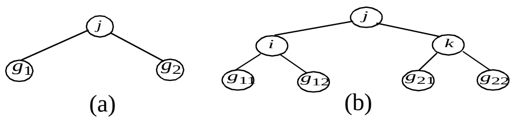


Fig. 4. Illustration of generation of balanced signature trees

The searching of a signature tree against a query signature can be done in the same way as discussed in [5], by means of which the behavior of a signature file as a filter is modeled as below. Let s_q be a query signature. The ith position of s_q is denoted as $s_q[i]$. During the traversal of a signature tree, the inexact matching is done as follows:

(i) Let v be the node encountered and $s_q[i]$ be the position to be checked.
(ii) If $s_q[i] = 1$, we move to the right child of v.
(iii) If $s_q[i] = 0$, both the right and left child of v will be explored.

3. On the general signature trees

In this section, we extend the above signature tree structure by assigning each internal node v a sequence: $i_1, i_2, \ldots, i_l$ for some l to tell that the i_1th, i_2th, ..., and i_lth bits in s_q will be checked when v is encountered during the searching of a signature tree against s_q. In this way, the size of a signature tree can be significantly reduced.

3.1 Definition

Assume that $S = s_1.s_2 \ldots .s_n$ be a signature file. For each s_i, we denote it as $s_i = s_i[1]s_i[2] \ldots s_i[m]$, where each $s_i[j] \in \{0, 1\}$ $(j = 1, \ldots, m)$.

Definition 1. (*general signature tree*) A general signature tree with respect to an integer l for a signature file $S = s_1.s_2 \ldots .s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = m$ for $k = 1, \ldots, n$, is a tree $T(l)$ such that

1. Each internal node v is associated with a sequence: $i_1, i_2, \ldots, i_l$ for some l, denoted

$c(v)$, to tell that the i_1th, i_2th, ..., and i_lth bits in the query signature will be checked when v is encountered.

2. For each internal node of $T(l)$, the number of its outgoing edges is bounded by 2^l. Each edge e is labeled with a different bit string $b_1b_2....b_l$, denoted *label*(e).
3. $T(l)$ has n leaves labeled 1, 2, ..., n, used as pointers to n different positions of s_1, s_2 ... and s_n in S. Let v be a leaf node. Denote by $p(v)$ the pointer to the corresponding signature.
4. Let v_1, ..., v_h be the nodes on a path from the root to a leaf v labeled i (then, this leaf node is a pointer to the ith signature in S, *i.e.*, $p(v) = i$). Let $\{ i_1^j, i_2^j, ..., i_l^j \}$ be the sequence associated with v_j ($1 \le j \le h - 1$). Let e_1, ..., $e_{h\text{-}1}$ be the edges on the path and let $b_1^j\, b_2^j \ldots b_l^j$ be the bit string labeling e_j ($1 \le j \le h - 1$). Then, $(i_1^1, b_1^1) \ldots (i_l^1, b_l^1) \ldots (i_1^{h-1}, b_1^{h-1}) \ldots (i_l^{h-1}, b_l^{h-1})$ makes up a signature identifier for s_i, $s_i(i_1^1, ..., i_l^1, ..., i_1^{h-1}, ..., i_l^{h-1})$.

Example 1. In Fig. 5(a), we show a general signature tree with $l = 2$, generated for the signature file shown in Fig. 3(a). It is easy to see that this tree contains less nodes than the tree shown in Fig. 3(b).

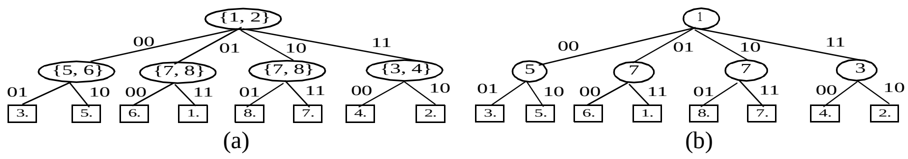


Fig. 5. Illustration for the construction of general signature trees

In addition, we notice that if the sequence associated with each node is contiguous, we need to store only one integer for a sequence. For example, the tree shown in Fig. 5(a) can be stored as shown in Fig. 5(b), in which a contiguous sequence is implicitly implemented.

The searching of a general signature tree against a query signature s_q can be done in a way similar to that of a signature tree, but different in the label checkings as described below:

(i) Let v be the node encountered. Assume that the sequence associated with it is i_1, i_2, ..., i_l for some l. Then, $s_q[i_1]$, ..., $s_q[i_l]$ will be checked.

(ii) Let e be an edge outgoing from v and labeled with a bit string $b_1b_2....b_l$. Then, if $b_1b_2...b_l$ matches $s_q[i_1]$, ..., $s_q[i_l]$, explore e. Recall that by “matching” we mean that for every j ($1 \le j \le l$) if $s_q[j] = 1$, we have $b_j = 1$; if $s_q[j] = 0$, b_j can be 1 or 0.

Example 2. Consider the signature file shown in Fig. 3(a) once again. The general signature tree for it is shown in Fig. 6(a). Assume s_q = 100 110 010 000. Then, only part of the signature tree (marked with thick edges in Fig. 6(a)) will be searched. On reaching a leaf node, the signature pointed to by the leaf node will be checked against s_q.

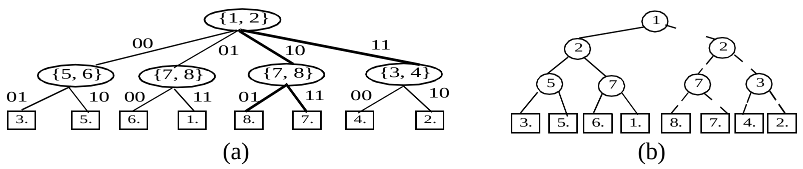


Fig. 6. Illustration for searching general signature trees and signature trees

We also notice that when we search the signature tree established for the same file, more edges will be accessed. (See the dashed edges in Fig. 6(b).)

From the above example, we can see that in comparison with the signature trees, the general signature trees have the following two advantages:

(1) A general signature tree tends to have fewer nodes.

(2) When searching a general signature tree, fewer edges will be visited.

3.2. Construction of general signature trees

Now we discuss how a general signature tree is constructed for a given signature file S.

Given an integer l, we choose, from S, the i_1th, i_2th, ..., and i_lth columns to divide the whole S into j ($\leq 2^l$) groups: $g_1 = \{s_1^1, s_2^1, ..., s_{k_1}^1\}$, ..., $g_j = \{s_1^j, s_2^j, ..., s_{k_j}^j\}$ such that

1. In each g_k ($1 \leq k \leq j$), for any two signatures s_a^k and s_b^k we have $s_a^k[i_1] = s_b^k[i_1]$, ..., and $s_a^k[i_l] = s_b^k[i_l]$.
2. For any two different groups g_x and g_y, there exists at least an $i_z \in \{i_1, i_2, ..., i_l\}$ such that for any $s_1 \in g_x$ and $s_2 \in g_y$, we have $s_1[i_z] \neq s_2[i_z]$.
3. $\max\{|g_1|, ..., |g_j|\} - \min\{|g_1|, ..., |g_j|\}$ is minimized, which guarantees that S is divided as evenly as possible.

Then, we can generate a tree T_S of two levels with the root labeled with a sequence $\{i_1, i_2, ..., i_l\}$ and j leaf nodes with each labeled with a g_k. For instance, for the signature file shown in Fig. 3(a), we can generate a tree as shown in Fig. 7(a). In this tree, $g_1 = \{s_3, s_5\}$, $g_2 = \{s_1, s_6\}$, $g_3 = \{s_7, s_8\}$ and $g_2 = \{s_2, s_4\}$. In a next step, we consider each g_k ($k = 1, ..., j$) as a single signature file with i_1th, i_2th, ..., and i_lth columns removed, and perform the same operations as above. Assume that T_{g_k} ($k = 1, ..., j$) is the tree generated for g_k. Replacing g_k with T_{g_k} for each k in T_S, we get another tree which is three levels high. For example, for the signature file shown Fig. 3(a), a tree as shown in Fig. 7(b) can be created, in which $g_{11} = \{s_3\}$, $g_{12} = \{s_1\}$, $g_{21} = \{s_6\}$, $g_{22} = \{s_1\}$, $g_{31} = \{s_8\}$, $g_{32} = \{s_7\}$, $g_{41} = \{s_4\}$, and $g_{42} = \{s_2\}$.

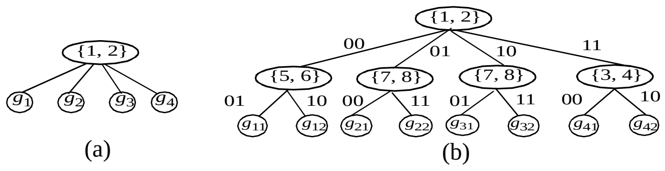


Fig. 7. Illustration of generation of general signature trees

This process will be repeated until the leaf nodes of a generated tree cannot be divided any more.

Below is a formal description of the above process.

```
Algorithm general-tree-generation(file, l)
input: file - a signature file; l - an integer.
output: a general signature tree.
begin
let $S = file$; $n \leftarrow |S|$;
if $n > 1$ then {
  choose the $i_1$th, $i_2$th, ..., and $i_l$th columns to divide the whole $S$ into $j$ ($\leq 2^l$) groups: $g_1 = \{s_1^1, s_2^1, ..., s_{k_1}^1\}$, ..., $g_j = \{s_1^j, s_2^j, ..., s_{k_j}^j\}$ as described above;
  generate a tree containing a root $r$ and $j$ child nodes marked with $g_1, ..., g_j$, respectively;
  $c(r) \leftarrow \{i_1, i_2, ..., i_l\}$;
  for ($i$ = 1 to $j$) do
     {replace the node marked $g_i$ with general-tree-generation($g_i$, $l$);}
else return;
end
```

By applying this algorithm with $l = 2$ to the signature file shown in Fig. 3(a), a general signature tree as shown in Fig. 5(a) will be created. Since $O(\binom{m - l \cdot i}{l} \cdot l \cdot n)$ time is needed to generate the nodes at level i in the tree, the time complexity of the whole process is on the order of $\sum_{i=1}^{\frac{1}{l}\log N} \binom{m - l \cdot n}{l} \cdot l \cdot n$.

In the above discussion, a very important issue has not yet been addressed. That is, for a file containing n signatures, what l should be chosen?

In the following, we discuss a heuristics for this task.

Consider a complete balanced signature tree T with the outdegree of each internal node $k = 2^l$, constructed for a signature file containing n signatures. Let $v_1, v_2, ...,$ and v_k be the child nodes of a node v in T, and $e_1 = (v, v_1), e_2 = (v, v_2), ...,$ and $e_k = (v, v_k)$ be the outgoing edges from v. If k is not so large, we can arrange an array A of size k to accommodate these edges in such a way that each entry $A[j]$ stores a link to a node v_i iff $label(e_i) = j$. So when we meet v during the searching of T against s_q, all those child nodes, which should be further explored, can be easily located. Assume that $c(v) = \{i_1, i_2, ..., i_l\}$ and $s_q[i_1] ... s_q[i_l] = b_1... b_l$. Then, any entry $A[j]$ with j equal to the value of a bit string $b_1'... b_l'$ should be explored if for any i with $b_i = 1$ we have $b_i' = 1$. Then, it is easy to show that the average number of entries in A, which may be explored, is

$$\frac{1}{2^l}\left(2^l + \binom{l}{1}2^{l-1} + \binom{l}{2}2^{l-2} + \ldots + 1\right) = \left(\frac{3}{2}\right)^l.$$

Therefore, the average number of the nodes, which must be visited during the searching of T against a query signature, can be estimated by $O\left(\frac{\left(\frac{3}{2}\right)^{\lceil \log_2 n \rceil} - 1}{\left(\frac{3}{2}\right)^l - 1}\right)$.

However, if k is large, we can not store the children of a node in an array as above since it can be quite sparsely populated, leading to a high space overhead. In this case, we need to store them in a linked list to avoid wasting space. In this way, to locate the child nodes to be explored, the linked list has to be scanned and at average $O(2^{l-1})$ time is needed. So in this case the average number of the nodes to be checked is estimated by $O\left(2^{l-1}\frac{\left(\frac{3}{2}\right)^{\lceil \log_2 n \rceil} - 1}{\left(\frac{3}{2}\right)^l - 1}\right)$.

Assume that when $k \le 2^{l_0}$ for some l_0, the child nodes are stored in arrays while when $k > 2^{l_0}$, they are stored in linked lists. Then, the average number of the nodes to be checked when searching a general signature tree is of the pattern shown in Fig. 8.

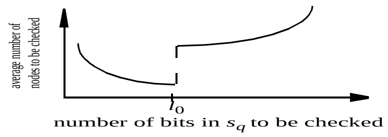


Fig. 8. Average number of the nodes to be checked

In practice, we can try different l's with the child nodes stored in arrays until the size of the general signature tree becomes larger than a given threshold. For instance, one of the goals of the general signature tree approach is to reduce the tree size. However, if, due to the sparse population of child links in the arrays, the size of a general signature tree with respect to an integer l becomes larger than the corresponding signature tree for the same signature file, we should set l_0 to be an integer smaller than l.

4. Maintenance of general signature trees

In this section, we consider the maintenance of general signature trees. Concretely, we discuss how a general signature tree is changed when a new signature is inserted into the signature file or when a signature is removed from it.

- *inserting a signature*

When a signature s is inserted into a signature file, we will first search the corresponding signature tree as described in 3.1. The searching stops when one of the following two conditions is satisfied:

(i) The searching meets a node v with $c(v) = \{i_1, i_2, ..., i_l\}$ and none of its outgoing edge matches $s[i_1]s[i_2]...s[i_l]$.

(ii)The searching reaches a leaf node u with $p(u) = i$.

In case (i), we simply generate a new leaf node v' with $p(v')$ pointing to s and connect v and v' using an edge labeled with $s[i_1]s[i_2]...s[i_l]$. In case (ii), we will compare s and the signature s_i pointed to by $p(u)$ and find i_1', i_2', ..., i_l'such that $s[i_1']s[i_2']...s[i_l'] \neq s_i[i_1']s_i[i_2']...s_i[i_l']$. Then, we generate a new internal node v' with $c(v') = \{i_1', i_2', ..., i_l'\}$, and a new leaf node v'' with $p(v'')$ pointing to s. In addition, we *replace* u with v'. By "replace", we mean that the position of u in the tree is occupied by v' and u becomes one of its children. v'' is set to be another child node of v'. (See Fig. 9(a) for illustration.)

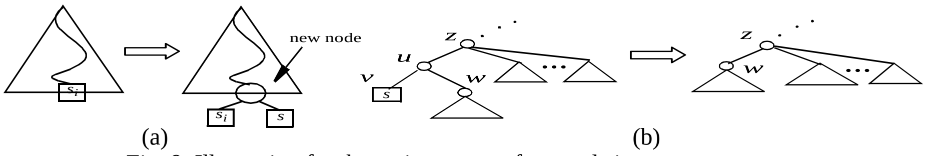


Fig. 9. Illustration for the maintenance of general signature trees

- deleting a signature

When a signature s is removed from a signature file, the corresponding signature tree may be changed in one of the following two ways:

(i) Let v be leaf node with $p(v)$ pointing to s. Let u be the parent of v. If u has more than two child nodes, v will be simply removed.

(ii) If u has exactly two child nodes v (to be removed) and w, replace u with the subtree rooted at w. (See Fig. 9(b) for illustration.)

After some insertions and deletions, a general signature tree may become unbalanced. So a tree should be reconstructed using the algorithm discussed in 3.2 periodically.

5. Experiments

We have implemented a test bed in C++, with our own buffer management (with first-in-first-out replacement policy). The computer was Intel Pentium III, running standalone. The capacity of the hard disk is 4.95 GB and the amount of the main memory available is 46 MB.

We have tested the signature tree approach (ST) and the general signature tree approach (GST). For the GST, only two versions are tested: two contiguous bit checking (TwoCBC) and three contiguous bit checking (ThreeCBC). By the TwoCBC, each time when a node in encountered, two contiguous bits in the query signature will be checked, while by ThreeCBC, each time three contiguous bits in the query signature will be checked. They are applied to different signature queries against the signature files of different sizes. All the signatures are created randomly using a uniform distribution for the positions that will be set to 1. The performance measure was considered to be the number of page accesses required to satisfy a query. For each query, an average of 20 measurements was taken.

For the experiment purpose, all the trees are stored page-wise as illustrated in Fig. 10.

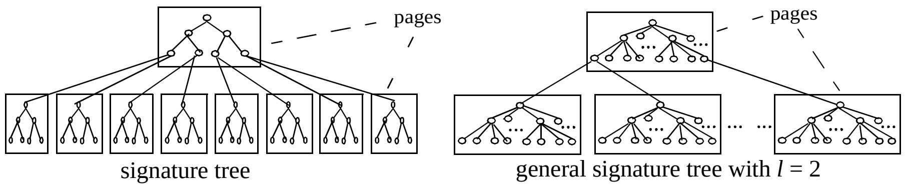


Fig. 10. Illustration for tree storage

The considered parameters and the tested values for each parameter are given in Table 1.

Tabel 1:

parameters \ data	groupI	groupII	groupIII	groupIV
number of signatures (×1024)	100	200	100	200
signature size/weight (in bits)	64/32	64/16	128/64	128/32
page size (in KB)	1	2	1	2

For all the methods implemented, an entry in a signature file contains two fields: a signature and an object identifier as shown in Fig. 11(a). Each internal node structure for a signature tree contains three fields: an integer to indicate which bit of a query signature will be checked, and two pointers to the left and the right child of a node, respectively. (See Fig. 11(b) for illustration.) Similarly, each internal node of a general signature tree with $l = 2$ has an integer to indicate a contiguous bit string of length 2 to be checked, and 4 pointers to its child nodes. (See Fig. 11(c) for illustration.)

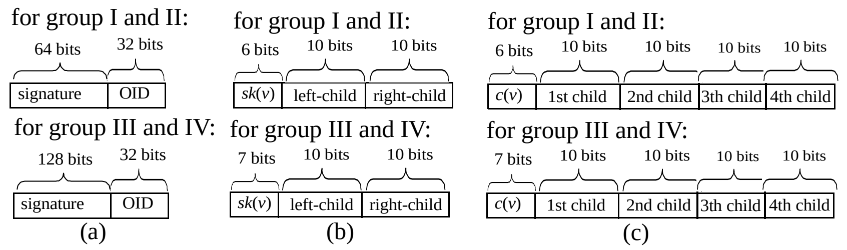


Fig. 11. Illustration for storing signature file entries and internal nodes in signature trees

Fig. 12 shows the test results for group I. The query signatures are generated randomly with all those positions to be set 1 uniformly distributed. Each of the queries is evaluated by different strategies.

From this figure, we can see that TwoCBC is much better than ST. But ThreeCBC is not much better than TwoCBC as we expect. It is because although the tree size of ThreeBCB is smaller than that of TwoCBC, a tree generated by ThreeCBC may not be so balanced as a tree generated by TwoCBC. However, as the length of signatures increases, we have more chance to find a balanced tree for ThreeCBC. So the discrepancy between ThreeCBC and TwoCBC increases as shown in Fig. 13.

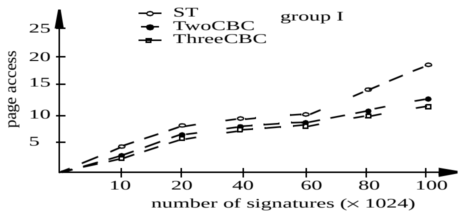


Fig. 12. Test results of group I

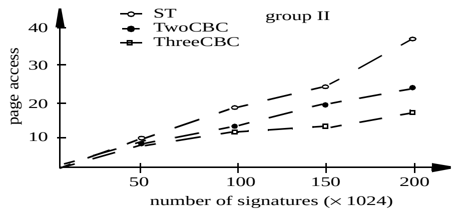


Fig. 13. Test results of group II

In Fig. 14 and Fig. 15, we show the results of Group III and Group IV, respectively. These results also confirm the above analysis.

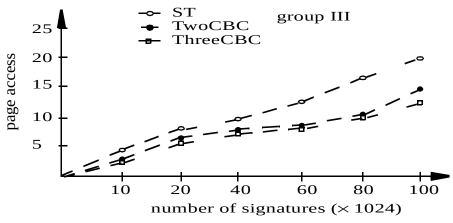


Fig. 14. Test results of group III

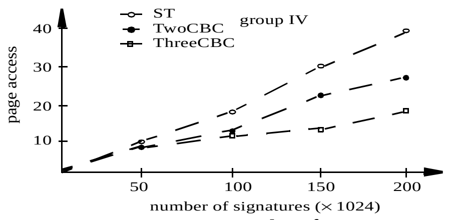


Fig. 15. Test results of group IV

In addition, the weight of a query signature (*i.e.*, the percentage of 1-bits in a query signature) affects both signature trees and general signature trees greatly. Fig. 16 shows the number of page access when the three methods are used to search a signature file containing 100×1024 signatures to locate query signatures with different weights.

From this, we can see that as the weight of a query signature increases, the searching time of both the signature trees and the general signature trees reduces. It is because each bit set to 1 in the query signature may cut off a subtree. However, more bits set to 1 in a query signature impacts the general signature trees more than it does to the signature trees, which shows that the filtering ability of a general signature tree is stronger than a signature tree.

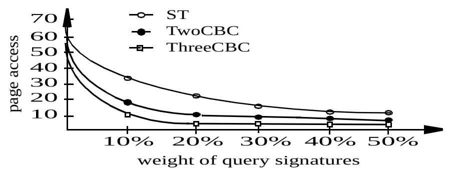


Fig. 16. Test results

6. Conclusion

In this paper, we extend the structure of signature trees by checking more than one bits in a query signature s_q when encountering a node during the searching of a signature tree against s_q. In this way, we can not only reduce the size of a signature tree, but also increase its filtering ability. Experiments have been done, which shows that the general signature tree uniformly outperforms the signature tree approach.

References

[1] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte and J. Simeon, “Querying documents in object databases,” *Int. J. on Digital Libraries*, Vol. 1, No. 1, Jan. 1997, pp. 5-19.

[2] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Com., London, 1974.

[3] R. Bayer and K. Unterrauer, “Prefix B-tree,” *ACM Transaction on Database Systems*, 2(1), 1977, pp. 11-26.

[4] S. Christodoulakis and C. Faloutsos, “Design consideration for a message file server,” *IEEE Trans. Software Engineering*, 10(2) (1984) 201-210.

[5] Y. Chen, Signature Files and Signature Trees, *Information Processing Letters*, Vol. 82, No. 4, March 2002, pp. 213-221.

[6] W.W. Chang, H.J. Schek, A signature access method for the STARBURST database system, in: *Proc. 19th VLDB Conf.*, 1989, pp. 145-153.

[7] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa and A. Pathria, Multimedia document presentation, information extraction and document formation in MINOS - A model and a system, *ACM Trans. Office Inform. Systems*, 4 (4), 1986, pp. 345-386.

[8] P. Ciaccia and P. Zezula, Declustering of key-based partitioned signature files, *ACM Trans. Database Systems*, 21 (3), 1996, pp. 295-338.

[9] U. Deppisch, S-tree: A Dynamic Balanced Signature Index for Office Retrieval, ACM SIGIR Conf., Sept. 1986, pp. 77-87.

[10] D. Dervos, Y. Manolopulos and P. Linardis, “Comparison of signature file models with superimposed coding,” *J. of Information Processing Letters* 65 (1998) 101 - 106.

[11] C. Faloutsos, “Access Methods for Text,” *ACM Computing Surveys*, 17(1), 1985, pp. 49-74.

[12] C. Faloutsos, “Signature Files,” in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 44-65.

[13] C. Faloutsos, R. Lee, C. Plaisant and B. Shneiderman, Incorporating string search in hypertext system: User interface and signature file design issues, *HyperMedia*, 2(3), 1990, pp. 183-200.

[14] D. Harman, E. Fox, R. and Baeza-Yates, “Inverted Files,” in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 28-43.

[15] Y. Ishikawa, H. Kitagawa and N. Ohbo, Evaluation of signature files as set access facilities in OODBs, in *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C., May 1993, pp. 247-256.

[16] D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Pub. London, 1973.

[17] A.J. Kent, R. Sacks-Davis, and K. Ramamohanarao, “A signature file scheme based on multiple organizations for indexing very large text databases,” *J. Am. Soc. Inf. Sci.* 41, 7, 508-534.

[18] W. Lee and D.L. Lee, “Signature File Methods for Indexing Object-Oriented Database Systems,” P*roc. ICIC'92 - 2nd Int. Conf. on Data and Knowledge Engineering: Theory and Application*, Hongkong, Dec. 1992, pp. 616-622.

[19] Morrison, D.R., “PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric,” *Journal of Association for Computing Machinery*, Vol. 15, No. 4, Oct. 1968, pp. 514-534.

[20] E. Tousidou, A. Nanopoulos, Y. Manolopoulos, “Improved methods for signature-tree construction,”*Computer Journal*, 43(4):301-314, 2000.

[21] E. Tousidou, P. Bozanis, Y. Manolopoulos, “Signature-based structures for objects with set-values attributes,” *Infromation Systems*, 27(2):93-121, 2002.

[22] H.S. Yong, S. Lee and H.J. Kim, “Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases,” *Proc. of 10th Int. Conf. on Data Engineering*, Houston, Texas, Feb. 1994, pp. 518-525.

[23] J. Zobel, A. Moffat and K. Ramamohanarao, “Inverted Files Versus Signature Files for Text Indexing”, *ACM Transaction on Database Systems*, Vol. 23, No. 4, Dec. 1998, pp. 453-490.