On the Signature Tree Construction and Analysis

Yangjun Chen and Yibin Chen

Abstract—Advanced database application areas, such as computer aided design, office automation, digital libraries, data-mining, as well as hypertext and multimedia systems, need to handle complex data structures with set-valued attributes, which can be represented as bit strings, called signatures. A set of signatures can be stored in a file, called a signature file. In this paper, we propose a new method to organize a signature file into a tree structure, called a signature tree, to speed up the signature file scanning and query evaluation. In addition, the average time complexity of searching a signature tree is analyzed and how to maintain a signature tree on disk is discussed. We also conducted experiments, which show that the approach of signature trees provides a promising index structure.

Index Terms—Signature files, bit-slice files, S-trees, signature trees, 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 [18], [30], and the signature file [13], [15], [21]. Full text searching imposes no space overhead, but requires high 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 [13], [15]. Nowadays, however, it is utilized in a wide range of applications, such as office filing [7], hypertext systems [16], and relational and object-oriented databases [4], [19], [24], [29], 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 and
- 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 matches 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 checked 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 constructing signature files have been proposed, such as sequential signature files, bit-slice files, S-trees, and their different variants. In this paper, we discuss a new mechanism to organize a signature file, called a *signature tree*, which improves the searching of signatures in a signature file by one order of magnitude or more. Especially, we have generalized the structure of signature trees to fit for different applications with different signature lengths.

The remainder of the paper is organized as follows: In Section 2, we show what is a signature file and review the relevant work. In Section 3, we discuss the signature trees and balanced signature trees, and analyze their time complexity. Section 4 is devoted to the maintenance of signature trees. In Section 5, we generalize the structure of signature trees, and in Section 6, we report the experiment results. Finally, Section 7 is a short conclusion.

2 Signature Files and Signature File Organization

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 it is 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 [13], [15]. Consequently, the signature file method involves high processing and I/O cost. This problem is mitigated by partitioning the signature file, by introducing auxiliary data structure, as well as by exploiting parallel computer architecture [9].

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

- *The authors are with the Department of Applied Computer Science, the University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9. E-mail: ychen2@uwinnipeg.ca, yibinchen@hotmail.com.*

Manuscript received 31 Aug. 2005; revised 09 Mar. 2006; accepted 22 Mar. 2006; published online 19 July 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0344-0805.

object:

John	12345678	professor

attribute signature:

John		010 000 100 110
12345678		100 010 010 100
professor	∨	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.

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" [13], [15]. 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 bit set to "1". As an example, assume that we have an attribute value "professor." Its signature can be constructed as follows: In terms of [6], the letter triplets in a word (or an attribute value) are the best choice for information carrying text segment in the construction of the signature for that word. Then, we will 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 pth bit in the string will be set to 1. For example, assume that we have $hash(\text{pro}) = 2$, $hash(\text{rof}) = 4$, $hash(\text{ofe}) = 8$, and $hash(\text{fes}) = 9$. Then, we will establish a bit string: 010 100 011 000 for "professor" as its word signature (see [12] 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."

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 does not match the query (i.e., $s \wedge s_q \neq s_q$). 3) The signature comparison indicates a match but the object in fact does not 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 does not 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 [6]:

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

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.)

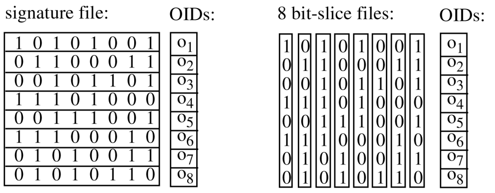
In a signature file, a set of signatures is sequentially stored, which is easy to implement and requires low storage space and low update cost. However, when a query is given, a full scan of the signature file is required. Therefore, it is generally slow in retrieval. Fig. 2a is a quite simple signature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

In this signature file, each signature is associated with an object identifier. If we have more than one same signatures in a file, we keep only one of them and associate it with more than one object identifiers.

2.2 Bit-Slice Files and Compressed Bit-Slice Files

A signature file can be stored in a column-wise manner. That is, the signatures in the file are *vertically* stored in a set of files [19]. Concretely, if the length of the signatures is m, then all the signatures will be stored in m files, in each of which one bit per signature for all the signatures is stored as shown in Fig. 2b.

With such a data structure, the signatures are checked slice-by-slice (rather than signature-by-signature) to find matching signatures. To demonstrate the retrieval process, consider a query signature $s_q = 10110000$. First, we check the first bit-slice file shown in Fig. 2b and find that only three positions: first, fourth, and sixth positions match the first bit in s_q. Then, we check the second bit-slice file. This time, however, only those three positions in the second file will be checked. Since the second bit in s_q is 0, no positions will be filtered. (Recall that the signature matching is an inexact matching. For a bit set to 0 in s_q, the corresponding bit in a signature in the signature file can be 1 or 0.) Next, we check the third bit-slice file against the third bit in s_q. Since all the three positions in it are set to 1, the same positions in a next bit-slice file, i.e., in the fourth bit-slice file will be checked against fourth bit in s_q. Since none of the three positions in the fourth bit-slice file matches this bit in the query, the search stops and reports a nil.



(a) signature file:

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

(b) 8 bit-slice files:

								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 and bit-slice signature files.

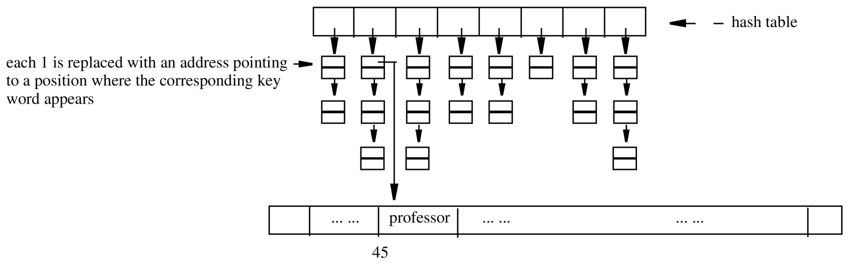

hash table
each 1 is replaced with an address pointing to a position where the corresponding key word appears
professor
45

Fig. 3. Illustration of compressed bit-slice file.

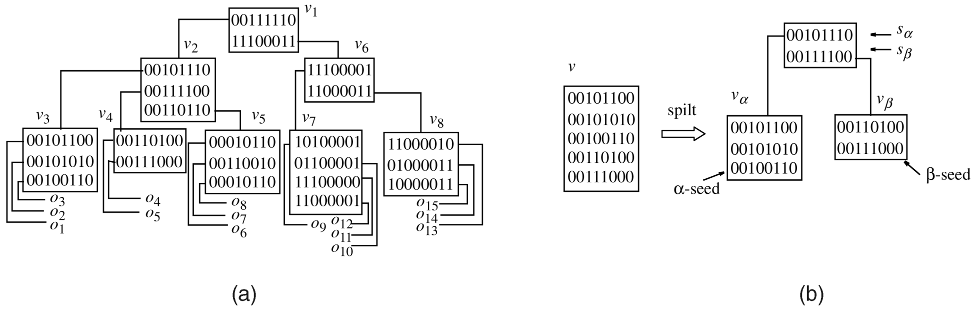


Fig. 4. Illustration for S-tree and node splitting.

From this process, we can see that only part of the m bit-slice files have to be scanned. Therefore, the search cost is lower than that of a sequential file. However, update cost becomes larger. For example, an insertion of a new signature requires about m disk accesses, one for each bit-slice file.

A bit-slice file can also be compressed. According to [14], [22], if a proper hash function is chosen, which maps each keyword to a signature with only one bit set to 1 (i.e., $k = 1$), a signature file will contain much less 1s than 0s. In this case, a signature file can be considered as a sparse matrix, and can be effectively compressed. A simple way to compress a bit-slice file is to replace each 1 with an address to a text position where the corresponding keyword can be found. All the addresses in a slice are then stored in a collection of buckets that are linked together. For example, the bit-slice file shown in Fig. 2b can be compressed as shown in Fig. 3.

In Fig. 3, an array (called a hash table in [14]) is used, in which each entry is a pointer to the head of a linked list of buckets. Each of them contains several addresses. Assume that we have a word "professor" that hashes to a signature with the second bit set to 1 (i.e., hash("professor") = 2), and that it appears in the document starting at the 45th byte of the text file. Then, searching the second linked bucket list, we will find the position where the word "professor" appears.

The space saved by this method can be estimated as follows: Let l be the size of an address. Then, the size of all the linked bucket lists is on $O(n \cdot D \cdot l)$. The size of the corresponding bit-slice file is $O(n \cdot m + n \cdot l)$. So, if $D < \frac{m}{l} - 1$, some space can be saved. However, such a space optimization is achieved at cost of query evaluation time since the probability of false drops will be definitely increased in the case of sparse signature files. In terms of [6], when a signature file is half-populated with 1s and half-populated with 0s, we have the lowest false drop probability. Obviously, the precondition for compressing a bit-slice file is just opposite to that.

2.3 S-Trees and Multilevel Signature Files

Similar to a B$^+$-tree, an S-tree is a height balanced multiway tree [11]. Each internal node corresponds to a page, which contains a set of signatures and each leaf node contains a set of entries of the form $< s, oid >$, where oid is an object identifier and s is the signature of the corresponding object. Let v be the parent node of v'. Then, there exists a signature in v, whose value is obtained by superimposing all the signatures in v'. See Fig. 4 for illustration.

To retrieve a query signature $s_q = 00110000$, we search the S-tree top-down. However, more than one path may be visited. For example, the first signature in the root v_1 shown in Fig. 4a leads us to its child node v_2 because the third and fourth bits are set to 1. In v_2, the second and third signatures match s_q. Then, we go to the leaf node v_4 and v_5. In v_4, we find two matching candidates o_4 and o_5, and in v_5, we have only one o_7.

The construction of an S-tree is an insertion-splitting process. At the very beginning, the S-tree contains only an empty leaf node and signatures in a file are inserted into it one by one. When a leaf node v becomes full, it will be split into two nodes and at the same time, a parent node v_{parent} will be generated if it does not exist. In addition, two new signatures will be put in v_{parent}. Assume that the capacity of v is K (i.e., v can accommodate K signatures.) Then, when we try to insert the $(K + 1)$th signature into v, it has to be split into two nodes v_α and v_β. To do this, we will pick a signature in v, which has the heaviest signature weight (i.e., with the most 1s) in v. It is called the α-seed and will be put in v_α. Then, we select a second signature, which has the maximum number of 1s in those positions where α has 0. That is, the signature provides the maximal weight increase to α. This signature is called the β-seed and put in v_β. Any of the rest $K-1$ signatures is assigned to v_α or v_β, depending on whether it is closer to v_α or v_β. The two new signatures (denoted s_α and s_β) to be put into the parent node are obtained by superimposing the signatures in v_α and v_β, respectively. See Fig. 4b for illustration.

The advantage of this method is that the scanning of a whole signature file is replaced by searching several paths in S-tree. However, the space overhead is almost doubled. Furthermore, due to superimposing, the nodes near the root tend to have heavy weights and thus have low selectivity. This is improved by Tousidou et al. [27], [28]. They elaborate the selection of α-seeds and β-seeds so that their distance is increased. However, this kind of improvement is achieved at cost of time, i.e., by checking more signatures, which makes the insertion of a signature into a S-tree extremely inefficient.

In [28], two algorithms were discussed. One needs $O(l^2)$ time to determine a α-seed and a β-seed, referred to as the *quadratic algorithm*, where l is the number of the signatures in a node. The other one needs $O(l^3)$ time, referred to as the *cubic algorithm*.

In general, to increase the selectivity of a signature in an internal node, longer signatures should be used, or the page for a node should not be fully populated. Both of them require more space. The multilevel signature file method discussed in [23] follows the same principle as the S-tree. The difference between them is in the way that signatures at a higher level are constructed. In the multilevel signature file method, a signature at a higher level is a superimposed code generated *directly* from a group of p text blocks, instead of superimposing p signatures at the lower level. In other words, a signature at a higher level can be considered as being generated from a bigger block containing more words. Assume that any signature at the lowest level (leaf level) is created from a block of size D. Then, any signature at the second lowest level is generated from a block of size pD. For a better understanding, consider the S-tree shown in Fig. 4a once again. Corresponding to this S-tree, we may have a multilevel signature file as shown in Fig. 5.

As in a S-tree, any signature at the lowest level corresponds to an object (or a text block). But, any signature at a higher level is constructed in a different way. We pay attention to the level L_1 shown in Fig. 5, at which each signature is not generated by superimposing some signatures at the level L_0, but created directly from the text blocks. For example, the first signature at L_1 is generated by taking o_1, o_2, and o_3 as a single block. So, it will have a different length than the signatures in L_0. Obviously, such a data structure needs more space than S-trees. However, the filtering power of any signature at a higher level is effectively increased. It is because the ratio of 1s over the signature length at a higher level is kept unchanged. Recall that in an S-tree, a signature in an internal node may be populated with more 1s than in a leaf node, decreasing its selectivity significantly.

Another interesting method is the so-called signature graphs proposed in [5]. It works in a similar way to the *signature trees* to be discussed in this paper. The problem of this method is, however, there is no way to guarantee that

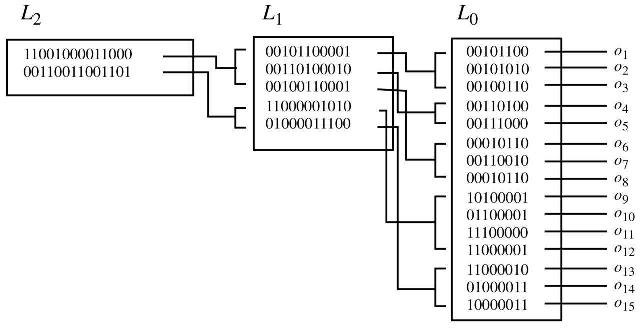


Fig. 5. Illustration for multilevel signature files.

the paths visited are of the same length, or say, the graph is "balanced." In the worst case, such a graph degrades to a signature file. In the following, we discuss the method of signature trees in great detail, by means of which all the drawbacks of S-trees as well as multilevel signature files can be removed.

3 Signature Trees

A signature tree works for a signature file like a *trie* [20], [25] for a text. But in a signature tree, each path is a *signature identifier* (defined below) which is not a continuous piece of bits, quite different from a trie, in which the bits (or characters) labeling a path are consecutive. In fact, the signature identifiers can be considered as a generalization of the concept of position identifiers [2], [11], extended to handle inexact matchings. As mentioned above, by the inexact matching, we ask for matches at the 1 bit positions of a query and indifferent at the 0 bit positions.

In comparison with the methods described in Section 2, the signature tree has the following advantages:

1. The slice checking in the bit-slice method is replaced with a single bit checking and less time is needed for both the insertion and deletion of signatures.
2. The checking of signatures in an internal node of an S-tree is changed to a binary tree searching and much less space is needed for the tree structure.

3.1 Definition of Signature Trees

Consider a signature s_i of length m. We denote it as

$$s_i = s_i[1]s_i[2]\ldots s_i[m],$$

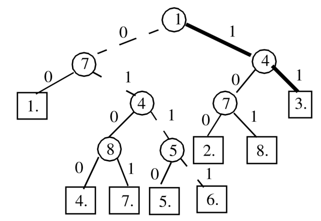
where each $s_i[j] \in \{0,1\} (j = 1, \ldots, m)$. We also use $s_i(j_1, \ldots, j_h)$ to denote a sequence of pairs with regard to $s_i : (j_1, s_i[j_1])(j_2, s_i[j_2]) \ldots (j_h, s_i[j_h])$, where $1 \le j_k \le m$ for $k \in \{1, \ldots, h\}$.

Definition 1 (*signature identifier*). *Let $S = s_1.s_2 \ldots .s_n$ denote a signature file. Consider $s_i (1 \le i \le n)$. If there exists a sequence: $j_1, \ldots, j_h$ such that for any $k \ne i (1 \le k \le n)$, we have $s_i(j_1, \ldots, j_h) \ne s_k(j_1, \ldots, j_h)$, then we say $s_i(j_1, \ldots, j_h)$ identifies the signature s_i or say $s_i(j_1, \ldots, j_h)$ is an identifier of s_i with regard to S.*

For example, in Fig. 4a,

s_1.	011 001 000 101
s_2.	111 011 001 111
s_3.	111 101 010 111
s_4.	011 001 101 111
s_5.	011 101 110 101
s_6.	011 111 110 101
s_7.	011 001 111 111
s_8.	111 011 111 111

(a)



(b)

Fig. 6. A signature files and a signature tree.

$$s_6(1,7,4,5) = (1,0)(7,1)(4,1)(5,1)$$

is an identifier of s_6 since for any $i \neq 6$, we have $s_i(1,7,4,5) \neq s_6(1,7,4,5)$. (For instance,

$$s_1(1,7,4,5) = (1,0)(7,0)(4,0)(5,0) \neq s_6(1,7,4,5),$$
$$s_2(1,7,4,5) = (1,1)(7,0)(4,0)(5,1) \neq s_6(1,7,4,5),$$

and so on. Similarly, $s_1(1,7) = (1,0)(7,0)$ is an identifier for s_1 since any $i \neq 1$ we have $s_i(1,7) \neq s_1(1,7)$.)

In the following, we'll see that in a signature tree, each path corresponds to a signature identifier.

Definition 2 (*signature tree*). *A signature tree 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 binary tree T such that*

1. *For each internal node of T, the left edge leaving it is always labeled with 0 and the right edge is always labeled with 1.*
2. *T has n leaves labeled $1, 2, \ldots, n$, used as pointers to n different positions of $s_1, s_2, \ldots$ and s_n in S. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.*
3. *Each internal node v is associated with a number, denoted by $sk(v)$, to tell which bit will be checked.*
4. *Let $i_1, \ldots, i_h$ be the numbers associated with 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 $p_1, \ldots, p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1) \ldots (j_h, p_h)$ makes up a signature identifier for $s_i, s_i(j_1, \ldots, j_h)$.*

Example 1. In Fig. 6b, we show a signature tree for the signature file shown in Fig. 6a. In this signature tree, each edge is labeled with 0 or 1 and each leaf node is a pointer to a signature in the signature file. In addition, each internal node v is marked with an integer $sk(v)$ to tell which bit will be checked. Consider the path going through the nodes marked 1, 7, and 4. If this path is searched for locating some signature s, then three bits of $s : s[1]$, $s[7]$, and $s[4]$ must be checked. If $s[4] = 1$, the search will go to the right child of the node marked "4." This child node is marked with 5 and then the fifth bit of $s : s[5]$ will be checked. Also, see the path consisting of the dashed edges in Fig. 6b, which corresponds to the identifier of $s_6 : s_6(1,7,4,5) = (1,0)(7,1)(4,1)(5,1)$. Similarly, the identifier of s_3 is $s_3(1,4) = (1,1)(4,1)$ (see the path consisting of thick edges).

In the following sections, we discuss how to construct a signature tree for a signature file and how a signature tree is searched.

3.2 A Simple Way for Constructing Signature Trees

Below, we give an algorithm to construct a signature tree for a signature file, which needs only $O(n \cdot m)$ time, where n represents the number of the signatures in the signature file and m is the length of a signature.

At the very beginning, the tree contains an initial node: a node containing a pointer to the first signature.

Then, we take a next signature and insert it into the tree. Let s be the next signature we wish to enter. We traverse the tree from the root. Let v be the node encountered and assume that v is an internal node with $sk(v) = i$. Then, $s[i]$ will be checked. If $s[i] = 0$, we go left. Otherwise, we go right. If v is a leaf node, we compare s with the signature s_0 pointed to by v. s cannot be the same as v since in S, there is no signature which is identical to anyone else. (If we have two identical signatures, one will be removed and the remaining one will be associated with two OIDs.) But, several bits of s can be determined, which agree with s_0. Assume that the first k bits of s agree with s_0; but, s differs from s_0 in the $(k+1)$th position, where s has the digit b and s_0 has $1 - b$. We construct a new node u with $sk(u) = k + 1$ and *replace* v with u. (Note that v will not be removed. By "replace," we mean that the position of v in the tree is occupied by u and v becomes one of u's children.) If $b = 1$, we make v and the pointer to s be the left and right child of u, respectively. If $b = 0$, we make v the right child of u and the pointer to s the left child of u. The following is the formal description of the algorithm.

```
Algorithm sig-tree-generation(file)
begin
 construct a root node $r$ with $sk(r) = 1$; /*where $r$
 corresponds to the first signature $s_1$ in the signature file*/
    for $j = 2$ to $n$ do
       call insert($s_j$);
end
Procedure insert($s$)
begin
    stack ← root;
    while stack not empty do
1      {$v$ ← pop(stack);
2      if $v$ is not a leaf then
3        {$i \leftarrow sk(v)$;
4          if $s[i] = 1$ then
              {let $a$ be the right child of $v$; push(stack, $a$);}
5          else {let $a$ be the left child of $v$; push(stack, $a$);}
6          }
7      else   (*$v$ is a leaf.*)
8      {compare $s$ with the signature $s_0$ pointed by $p(v)$;
9      assume that the first $k$ bit of $s$ agree with $s_0$;
10     but $s$ differs from $s_0$ in the $(k+1)$th position;
11     $w \leftarrow v$; replace $v$ with a new node $u$ with
          $sk(u) = k + 1$;
12     if $s[k+1] = 1$ then
          make $s$ and $w$ be respectively the right and left
          children of $u$
```

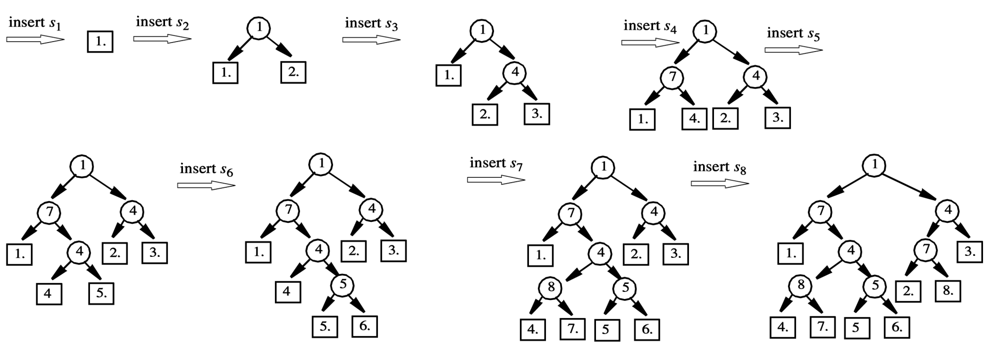


Fig. 7. Sample trace of signature tree generation.

```
13    else make $w$ and $s$ be the right and left children of $u$,
        respectively;}
14    }
end
```

In the procedure $insert()$, $stack$ is a stack structure used to control the tree traversal.

In Fig. 7, we trace the above algorithm against the signature file shown in Fig. 6a.

In the following, we prove the correctness of the Algorithm *sig-tree-generation()*. To this end, it should be specified that each path from the root to a leaf node in a signature tree corresponds to a signature identifier. We have the following proposition.

Proposition 1. *Let T be a signature tree for a signature file S. Let $P = v_1.e_1 \ldots v_{g-1}.e_{g-1}.v_g$ be a path in T from the root to a leaf node for some signature s in S, where $v_i (i = 1, \ldots, g)$ is a node and $e_i (i = 1, \ldots, g-1)$ is an edge from v_{i-1} to v_i. Then, we have $p(v_g) = s$. Denote $j_i = sk(v_i)(i = 1, \ldots, g-1)$. Then, $s(j_1, j_2, \ldots, j_{g-1}) = (j_1, b(e_1)) \ldots (j_{g-1}, b(e_{g-1}))$ makes up an identifier for s.*

Proof. Let $S = s_1.s_2....s_n$ be a signature file and T a signature tree for it. Let $P = v_1 e_1 \ldots v_{g-1} e_{g-1} v_g$ be a path from the root to a leaf node for s_i in T. Assume that there exists another signature s_t such that

$$s_t(j_1, j_2, \ldots, j_{g-1}) = s_i(j_1, j_2, \ldots, j_{g-1}),$$

where $j_i = sk(v_i)(i = 1, \ldots, g-1)$. Without loss of generality, assume that $t > i$. Then, at the moment when s_t is inserted into T, two new nodes v and v' will be inserted as shown in Figs. 8a or 8b (see lines 10-15 of the procedure $insert()$). Here, v' is a pointer to s_t and v is associated with a number indicating the position where $p(v_t)$ and $p(v')$ differ.

It shows that the path for s_i should be

$$v_1.e_1 \ldots v_{g-1}.e.ve'.v_g$$

or $v_1.e_1 \ldots v_{g-1}.e.ve''.v_g$, which contradicts the assumption. Therefore, there is not any other signature s_t with

$$s_t(j_1, j_2, \ldots, j_{n-1}) = (j_1, b(e_1)) \ldots (j_{n-1}, b(e_{n-1})).$$

So, $s_i(j_1, j_2, \ldots, j_{n-1})$ is an identifier of s_i. □

The analysis of the time complexity of the algorithm is relatively simple. From the procedure $insert()$, we see that there is only one loop to insert all signatures of a signature file into a tree. At each step within the loop, only one path is searched. Obviously, each path is bounded by m. Thus, we have the following proposition.

Proposition 2. *The time complexity of the algorithm* sig-tree-generation *is bounded by* $O(n \cdot m)$).

Proof. See the above analysis. □

3.3 Searching of Signature Trees

Now, we discuss how to search a signature tree to model the behavior of a signature file as a filter. 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:

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

In fact, this process just corresponds to the signature matching criterion, i.e., for a bit position i in s_q, if it is set to 1, the corresponding bit position in s must be set to 1; if it is set to 0, the corresponding bit position in s can be 1 or 0.

To implement this kind of inexact matching, we search the signature tree in a depth-first manner and maintain a stack structure $stack_p$ to control the tree traversal.

```
Algorithm signature-tree-search
input: a query signature $s_q$;
output: a set of signatures which survive the checking;
1. $R \leftarrow \emptyset$.
2. Push the root of the signature tree into $stack_p$.
3. If $stack_p$ is not empty, $v \leftarrow$ pop($stack_p$); else return($R$).
```

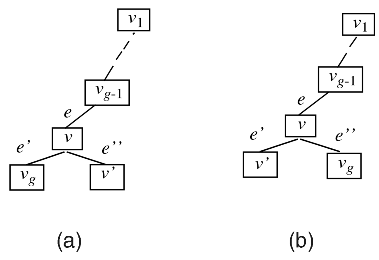


Fig. 8. Inserting a node v into T.

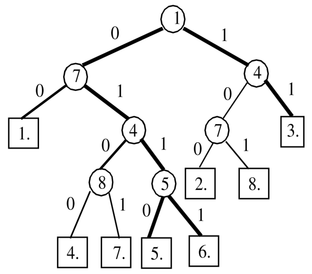
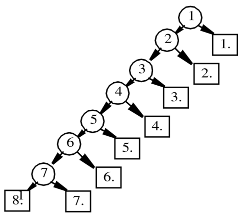


Fig. 9. Signature tree search.

s_1.	100 100 100 100
s_2.	010 010 010 010
s_3.	001 001 001 001
s_4.	000 110 010 010
s_5.	000 011 001 001
s_6.	000 001 100 100
s_7.	000 000 110 010
s_8.	000 000 010 110

(a)



(b)

Fig. 10. A skewed signature tree.

4. If v is not a leaf node, $i \leftarrow sk(v)$;
   If $s_q(i) = 0$, push c_r and c_l into $stack_p$; (where c_r and c_l are v's right and left child, respectively.) otherwise, push only c_r into $stack_p$.
5. Compare s_q with the signature pointed by $p(v)$.
   /*$p(v)$–pointer to the block signature*/
   If s_q matches, $R \leftarrow R \cup \{p(v)\}$.
6. Go to (3).

The following example helps for illustrating the main idea of the algorithm.

Example 2. Consider the signature file and the signature tree shown in Fig. 6 once again. Assume $s_q = 000\ 100\ 100\ 000$. Then, only part of the signature tree (marked with thick edges in Fig. 9) will be searched. On reaching a leaf node, the signature pointed to by the leaf node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching since only three signatures need to be checked while a signature file scanning will check eight signatures. For a balanced signature tree, the height of the tree is bounded by $O(\log_2 n)$, where n is the number of the leaf nodes. Then, the cost of searching a balanced signature tree will be $O(\lambda \cdot \log_2 n)$ on the average, where λ represents the number of paths traversed, which is equal to the number of signatures checked. Let t represent the number of bits which are set in s_q and checked during the search. Then, $\lambda = O(n/2^t)$. It is because each bit set to 1 (in s_q), which is checked during the search, will prevent half of a subtree from being visited. Compared to the time complexity of the signature file scanning $O(n)$, it is a major benefit. We will discuss this issue in Section 3.5 in great detail.

3.4 Balanced Signature Trees

A signature tree can be quite skewed as shown in Fig. 10.

But, the tree shown in Fig. 11 is completely balanced for the same signature file. However, the signature identifiers for the signatures are different from those shown in Fig. 10b.

The problem is how to control the process of constructing a signature tree in such a way that the generated signature tree is almost balanced.

In the following, we propose a weight-based method, which needs more time than the method discussed above, but always returns a balanced tree.

3.4.1 Weight-Based Method

An observation shows that a signature tree for a signature file corresponds to a recursive partitioning of the file. For instance, the left and right subtrees of the root divides the signature file into two parts: A and B. If $|A| = |B|$, we say the signature file is evenly divided. Again, each of A and B is further divided along the signature tree. If the partitioning at each level is even, then the corresponding signature tree must be well balanced and its height is on the order of $O(\log n)$. Based on the above observation, we propose a method to recursively divide a signature file in such a way that any time a subfile is partitioned as evenly as possible. The signature tree is created as a data structure "recording" this process.

Intuitively, a signature file $S = s_1.s_2 \ldots .s_n$ can be considered as a Boolean matrix. We use $S[i]$ to represent the ith column of S. 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 a 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 $g_1 = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ with each $s_{i_p}[j] = 0$ $(p = 1, \ldots, k)$ and $g_2 = \{s_{i_{k+1}}, s_{i_{k+2}}, \ldots, s_{i_N}\}$ with each $s_{i_q}[j] = 1$ $(q = k+1, \ldots, n)$; and generate a tree as shown in Fig. 12a. 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 shown in Fig. 12b. We repeat this process until the leaf nodes of a generated tree cannot be divided any more.

In Fig. 12a, $g_1 = \{s_1, s_3, s_5, s_6\}$ and $g_2 = \{s_2, s_4, s_7, s_8\}$; and, in Fig. 12b, $g_{11} = \{s_3, s_5\}$, $g_{12} = \{s_6, s_1\}$, $g_{21} = \{s_8, s_7\}$, and $g_{22} = \{s_4, s_2\}$.

Below is a formal description of the above process.

```
**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 $|w(S[i]) - \frac{1}{2}N|$ is minimum;
    let $g_1 = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ with each $s_{i_p}[j] = 0$ $(p = 1, \ldots, k)$;
    let $g_2 = \{s_{i_{k+1}}, s_{i_{k+2}}, \ldots, s_{i_N}\}$ with each
        $s_{i_q}[j] = 1 (q = k+1, \ldots, N)$
```

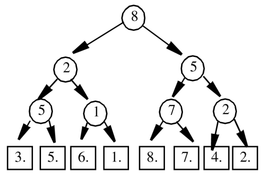


Fig. 11. A balanced signature tree.

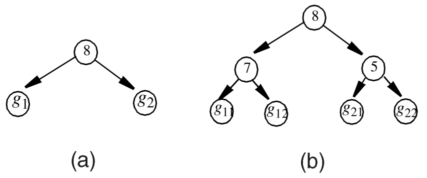


Fig. 12. Illustration of generation of balances signature trees.

```
    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
        $balanced\text{-}tree\text{-}generation(g_1)$;
    replace the node marked $g_2$ with
        $balanced\text{-}tree\text{-}generation(g_2)$;}
else return;
end
```

By applying this algorithm to the signature file shown in Fig. 10a, a balanced signature tree as shown in Fig. 9 will be created. Since $O(n \cdot m)$ time is needed to generate the nodes at each level of the tree, the time complexity of the whole process is on the order of $O(n \cdot m \cdot \log n)$.

3.5 Average Number of Checked Nodes

For a balanced binary tree of size n, it takes $O(\log n)$ time to search along a path. However, when exploring a balanced signature tree, more than one path is traversed. An interesting question is, how many nodes in the tree will be checked? In the following, we give a probabilistic analysis to answer this question.

Denote by t a signature tree, in which the edges are labeled with 0 or 1. Let $s = s[1]s[2]\ldots.s[m]$ be a query signature, where $s[i] \in \{0,1\}$. Then, 1 in s matches only 1 in t, while 0 in s matches both 0 and 1 in t. We use $c_s(t)$ to represent the cost of searching t against s. In addition, we use $s', s'', s''', \ldots$ to designate the patterns obtained by circularly shifting the bits of s to the left by 1, 2, 3, ... positions.

Obviously, if the first bit of s is 0, we have, for the expected cost of a random string s,

$$c_s(t) = 1 + c_{s'}(t_1) + c_{s'}(t_2), \tag{1}$$

where t_1, and t_2 represent the two subtrees of the root of t. See Fig. 13 for illustration.

It is because in this case, the search has to proceed in parallel along the two subtrees with s changing cyclically to s'. If the first bit in s is 1, we have

$$c_s(t) = 1 + c_{s'}(t_2) \tag{2}$$

since in this case, the search proceeds only in t_2.

Given n $(n \geq 2)$ random nodes in t, the probability that

$$|t_1| = p, |t_2| = n - p \tag{3}$$

can be estimated by the *Bernoulli* probabilities

$$\binom{n}{p}\left(\tfrac{1}{2}\right)^p\left(\tfrac{1}{2}\right)^{n-p} = \tfrac{1}{2^n}\binom{n}{p}. \tag{4}$$

Let $c_{s,n}$ denote the expected cost of searching a signature tree of size n against s. We have the following recurrences:

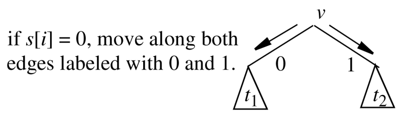


Fig. 13. Illustration for signature tree searching.

$$\text{if } s \text{ starts with } 0, \quad c_{s,n} = 1 + \tfrac{2}{2^n}\sum_p \binom{n}{p} c_{s',p}, \quad n \geq 2; \tag{5}$$

$$\text{if } s \text{ starts with } 1, \quad c_{s,n} = 1 + \tfrac{1}{2^n}\sum_p \binom{n}{p} c_{s',p}, \quad n \geq 2. \tag{6}$$

Let $\lambda_i = 1$ if ith bit in s is 1, and $\lambda_i = 2$ if ith bit in s is 0. The above recurrence can be rewritten as follows:

$$c_{s,n} = 1 + \tfrac{\lambda_1}{2^n}\sum_p \binom{n}{p} c_{s',p} - \delta_{n,0} - \delta_{n,1}, \tag{7}$$

where $\delta_{n,j} (j = 0, 1)$ is equal to 1 if $n = j$; otherwise, equal to 0.

Proposition 3. *The exponential generating function of the average cost*

$$C_s(z) = \sum_{n \geq 0} c_{s,n}\tfrac{z^n}{n!} \tag{8}$$

satisfies the relation

$$C_s(z) = \lambda^1 e^{z/2}\, C_{s'}\left(\frac{z}{2}\right) + e^z - 1 - z. \tag{9}$$

Proof. In terms of (7), $C_s(z)$ can be rewritten as follows:

$$\begin{aligned} C_s(z) &= \sum_{n\geq 0}\left(1 + \lambda_1\left(\tfrac{1}{2}\right)^n \sum_p \binom{n}{p} c_{s',p} - \delta_{n,0} - \delta_{n,1}\right)\tfrac{z^n}{n!} \\ &= \sum_{n\geq 0}\tfrac{z^n}{n!} + \sum_p \lambda_1\left(\tfrac{1}{2}\right)^n \sum_{n\geq 0}\binom{n}{p} c_{s',p}\tfrac{z^n}{n!} - \sum_{n\geq 0}\delta_{n,0}\tfrac{z_n}{n!} \\ &\quad - \sum_{n\geq 0}\delta_{n,1}\tfrac{z_n}{n!} \\ &= e^z + \lambda_1 \sum_p \tfrac{(z/2)^p}{p!}\sum_{n\geq 0} c_{s',p}\tfrac{(z/2)^{n-p}}{(n-p)!} - 1 - z \\ &= \lambda_1 e^{z/2} C_{s'}\left(\tfrac{z}{2}\right) + e^z - 1 - z. \end{aligned} \tag{10}$$

□

In the same way, we will get $C_{s'}(z), C_{s''}(z), \ldots$, and so on. Concretely, we will have the following equations:

$$\begin{aligned} C_s(z) &= \lambda_1 e^{z/2} C_{s'}\left(\tfrac{z}{2}\right) + e^z - 1 - z, \\ C_{s'}(z) &= \lambda_2 e^{z/2} C_{s''}\left(\tfrac{z}{2}\right) + e^z - 1 - z \\ &\ldots\;\ldots \\ C_{s^{(m-1)}}(z) &= \lambda_m e^{z/2} C_s\left(\tfrac{z}{2}\right) + e^z - 1 - z. \end{aligned} \tag{11}$$

These equations can be solved by successive transportation. For instance, when we transport the expression of $C_{s'}(z)$ given by the second equation in (11), we have

$$C_s(z) = a(z) + \lambda_1 e^{z/2} a\left(\tfrac{z}{2}\right) + \lambda_1\lambda_2 e^{z/2} e^{z/2^2} C_{s''}\left(\tfrac{z}{2^2}\right), \tag{12}$$

where $a(z) = e^z - 1 - z$.

In a next step, we transport $C_{s'''}$ into the equation given in (12). This kind of transformation continues until the relation is only on C_s itself. Then, we have

$$\begin{aligned} C_s(z) &= \lambda_1\lambda_2\ldots\lambda_m exp\big[z\big(1-\tfrac{1}{2^m}\big)\big]C_s\big(\tfrac{z}{2^m}\big) \\ &\quad + \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\Big[z\Big(1-\tfrac{1}{2_j}\Big)\Big]\big(exp\big(\tfrac{z}{2^j}\big)-1-\tfrac{z}{2^j}\big) \\ &= 2^{m-k} exp\big[z\big(1-\tfrac{1}{2^m}\big)\big]C_s\big(\tfrac{z}{2^m}\big) \\ &\quad + \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\Big[z\Big(1-\tfrac{1}{2_j}\Big)\Big]\big(exp\big(\tfrac{z}{2^j}\big)-1-\tfrac{z}{2^j}\big), \end{aligned} \tag{13}$$

where k is the number of 1s in s.

Let $\alpha = 2^{m-k}, \beta = 1-\frac{1}{2^m}, \lambda = \frac{1}{2^m}$ and

$$A(z) = \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\Big[z\Big(1-\tfrac{1}{2_j}\Big)\Big]\big(exp\big(\tfrac{z}{2^j}\big)-1-\tfrac{z}{2^j}\big).$$

We have

$$C_s(z) = \alpha e^{\beta z}C_s(\lambda z) + A(z). \tag{14}$$

This equation can be solved by iteration as discussed above:

$$\begin{aligned} C_s(z) &= \sum_{j=0}^{\infty}\alpha^j exp\Big(\beta\tfrac{1-\lambda^j}{1-\lambda}z\Big)A\big(\lambda^j z\big) \\ &= \sum_{j=0}^{\infty}2^{j(m-k)}\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h\big[exp(z) \\ &\quad - exp\big(z\big(1-\tfrac{1}{2^h2^{mj}}\big)\big)\quad\big(1+\tfrac{z}{2^h2^{mj}}\big)\big]. \end{aligned} \tag{15}$$

Using the *Taylor* formula to expand $exp(z)$ and $exp\big(z\big(1-\frac{1}{2^h2^{mj}}\big)\big)\big(1+\frac{1}{2^h2^{mj}}\big)$ in $C_s(z)$ given by the above sum, and then extracting the *Taylor* coefficients, we get

$$c_{s,n} = \sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h\sum_{j\geq 0}2^{j(m-k)}D_{jh}(n), \tag{16}$$

where $D_{00}(n) = 1$ and for $j > 0$ and $h > 0$,

$$D_{jh}(n) = 1-(1-2^{-mj-h})^n - n2^{-mj-h}(1-2^{-mj-h})^{n-1}. \tag{17}$$

To estimate $c_{s,n}$, we resort to the complex analysis, which shows that $c_{s,n} \sim n^{1-\frac{k}{m}}$. If $\frac{k}{m} = \frac{1}{2}$, we have

$$c_{s,n} = \mathrm{O}(n^{0.5}). \tag{18}$$

(See the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, for a detailed computation based on the contour integration of complex variabled functions.)

In terms of above analysis, we have the following proposition.

Proposition 4. *Let T be a signature tree over a signature file $S = s_1.s_2\ldots.s_n$ with $|s_i| = k$ $(i = 1,\ldots,n)$. Let s_q be a query signature of length m with k bits set to 1. Then, the average nodes checked during the searching of T against s_q is on the order of $\mathrm{O}(n^{1-\frac{k}{m}})$.*

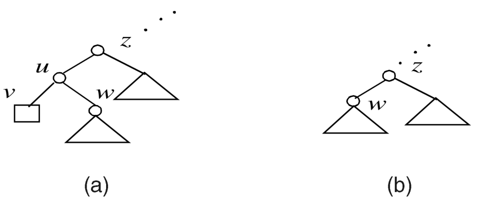


Fig. 14. Illustration for deleting a signature.

Proof. See the above analysis. □

4 Signature Tree Maintenance

In this section, we address how to maintain a signature tree. First, we discuss the case that a signature tree can entirely fit in main memory in Section 4.1. Then, we discuss the case that a signature tree cannot be put in main memory once for all in Section 4.2.

4.1 Maintenance of Internal Signature Trees

An internal signature tree refers to a tree that can fit entirely in main memory. In this case, insertion and deletion of a signature into a tree can be done quite easily as discussed below. When a signature s is added to a signature file, the corresponding signature tree can be changed by simply running the procedure $insert()$ with s as the input (see Section 3.2). When a signature is removed from the signature file, we need to change the corresponding signature tree as follows:

1. Let z, u, v, and w be the nodes as shown in Fig. 14a and assume that v is a pointer to the signature to be removed.
2. Remove u and v. Set the left pointer of z to w. (If u is the right child of z, set the right pointer of z to w.)

The resulting signature tree is as shown in Fig. 14b.

From the above analysis, we see that the maintenance of an internal signature tree is an easy task. However, after several insertions and deletions, a signature tree may become unbalanced. For this reason, we will maintain a variable to record the difference between the lengths of the longest and the shortest paths. If the value of the variable is above a given threshold, the signature tree should be reconstructed by running *balanced-tree-generation( ).*

4.2 Maintenance of External Signature Trees

In a database, files are normally very large. Therefore, we have to consider the situation where a signature tree cannot fit entirely in main memory. We call such a tree an external signature tree (or an external structure for the signature tree). In this case, a signature tree is stored in a series of pages organized into a tree structure as shown in Fig. 15, in which each node corresponds to a page containing a binary tree.

Formally, an external structure ET for a signature tree T is defined as follows (to avoid any confusion, we will, in the following, refer to the nodes in ET as the page nodes while the nodes in T as the binary nodes or simply the nodes.):

1. Each internal page node n of ET is of the form: $b_n(r_n, a_{n1},\ldots,a_{ni_n})$, where b_n represents a subtree of T, r_n is its root, and $a_{n1},\ldots,a_{ni_n}$ are its leaf nodes. Each

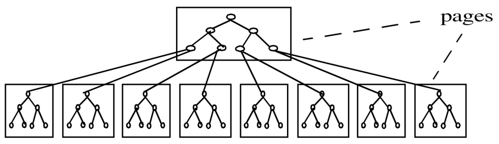


Fig. 15. An external signature tree.

internal node u of b_n is of the form: $< v(u), l(u), r(u) >$, where $v(u)$, $l(u)$, and $r(u)$ are the value, left link, and right link of u, respectively. Each leaf node a_{ni_j} of b_n is of the form: $< v(a_{ni_j}), lp(a_{ni_j}), rp(a_{ni_j}) >$, where $v(a_{ni_j})$ represents the value of a_{ni_j}, and $lp(a_{ni_j})$ and $rp(a_{ni_j})$ are two pointers to two pages containing the left and right subtrees of a_{ni_j}, respectively.

2. Let m be a child page node of n. Then, m is of the form: $b_m(r_m, a_{m1}, \ldots, a_{mi_m})$, where b_m represents a binary tree, r_m is its root and $a_{m1}, \ldots, a_{mi_m}$ are its leaf nodes. If m is an internal page node, $a_{m1}, \ldots, a_{mi_m}$ will have the same structure as $a_{n1}, \ldots, a_{mi_n}$ described in (1). If m is a leaf node, each $a_{mi_l} = p(s)$, the position of some signature s in the signature file.
3. The size $|b|$ of the binary tree b (the number of nodes in b) within an internal page node of ET satisfies

$$|b| \leq 2^k,$$

where k is an integer.
4. The root page of ET contains at least a binary node and the left and right links associated with it.

If $2^{k-1} \leq |b| \leq 2^k$ holds for each node in ET, it is said to be balanced; otherwise, it is unbalanced. However, according to the analysis of Section 3.4, an external signature tree is normally balanced, i.e., $2^{k-1} \leq |b| \leq 2^k$ holds for almost every page node in ET.

As with a B$^+$-tree, insertion and deletion of page nodes always begins from a leaf node. To maintain the tree balance, internal page nodes may split or be merged during the process. In the following section, we discuss these issues in great detail.

4.2.1 Insertion of Binary Nodes

Let s be a signature newly inserted into a signature file S. Accordingly, a node a_s will be inserted into the signature tree T for S as a leaf node. In effect, it will be inserted into a leaf page node m of the external structure ET of T. It can be done by taking the binary tree within that page into main memory and then inserting the node into the tree as discussed in Section 4.1. If for the binary tree b in m we have $|b|2^k$, the following node-splitting will be conducted:

1. Let $b_m(r_m, a_{m1}, \ldots, a_{mi_m})$ be the binary tree within m. Let r_{m1} and r_{m2} are the left and right child node of r_m, respectively. Assume that

$$b_{m1}(r_{m1}, a_{m1}, \ldots, a_{mi_j})(i_j < i_m)$$

is the subtree rooted at r_{m1} and

$$b_{m2}(r_{m1}, a_{mi_{j+1}}, \ldots, a_{mi_m})$$

is rooted at r_{m2}. We allocate a new page m' and put $b_{m2}(r_{m1}, a_{mi_{j+1}}, \ldots, a_{mi_m})$ into m'. Afterwards, promote r_m into the parent page node n of m and remove $b_{m2}(r_{m1}, a_{mi_{j+1}}, \ldots, a_{mi_m})$ from m.
2. If the size of the binary tree within n becomes larger than 2^k, split n as above. The node-splitting repeats along the path bottom-up until no splitting is needed.

4.2.2 Deletion of Binary Nodes

When a node is removed from a signature tree, it is always removed from the leaf level as discussed in Section 4.1. Let a be a leaf node to be removed from a signature tree T. In effect, it will be removed from a leaf page node m of the external structure ET for T. Let b be the binary tree within m. If the size of b becomes smaller than 2^{k-1}, we may merge it with its left or right sibling as follows:

1. Let m' be the left (right) sibling of m. Let

$$b_m(r_m, a_{m1}, \ldots, a_{mi_m})$$

and $b_{m'}(r_{m'}, a_{m'1}, \ldots, a_{m'i_{m'}})$ be two binary trees in m and m', respectively. If the size of $b_{m'}$ is smaller than 2^{k-1}, move $b_{m'}$ into m and afterwards eliminate m'. Let n be the parent page node of m and r is the parent node of r_m and $r_{m'}$. Move r into m and afterwards remove r from n.
2. If the size of the binary tree within n becomes smaller than 2^{k-1}, merge it with its left or right sibling if possible. This process repeats along the path bottom-up until the root of ET is reached or no merging operation can be done.

Note that it is not possible to redistribute the binary trees of m and any of its left and right siblings due to the properties of signature trees, which may leave an external signature tree unbalanced. According to the analysis of Section 3.4, however, it is seldom. If it is the case, i.e., if the difference between the lengths of the longest and the shortest paths is above a given threshold, we will use *balanced-tree-generation( )* to reconstruct the whole signature tree.

5 On the General Signature Trees

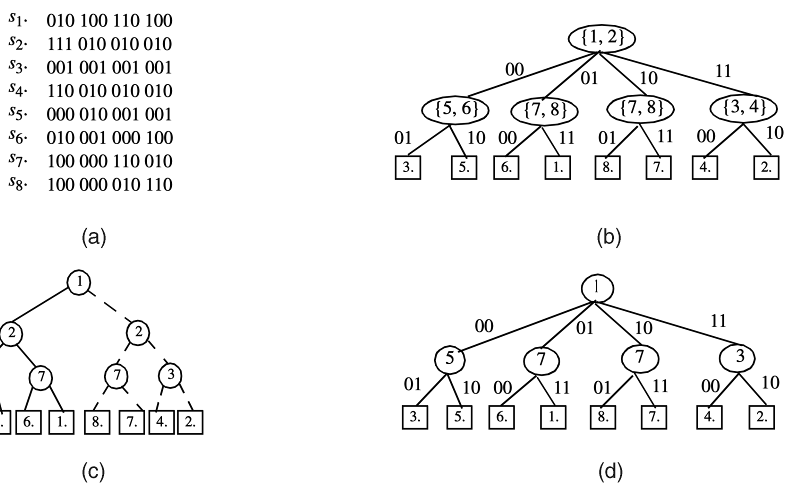
In this section, we extend the above signature tree structure by assigning each internal node v a sequence: $i_1, i_2, \ldots, i_l$ for some l to tell that the i_1th, i_2th, $\ldots$, and i_lth bits in s_q will be checked when v is encountered during the searching of a signature tree against s_q. In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

5.1 Definition of General Signature Trees

Assume that $S = s_1.s_2 \ldots .s_n$ is 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 3 (*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 binary 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, $\ldots$, and i_lth bits in the query signature will be checked when v is encountered.*



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. 16. Illustrations for the construction of general signature trees.

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 $b_1 b_2 \dots b_l$, denoted $label(e)$.*
3. *$T(l)$ has n leaves labeled $1, 2, \dots, n$, used as pointers to n different positions of $s_1, s_2 \dots$ and s_n in S. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.*
4. *Let $v_1, \dots, v_h$ 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 $\{i_1^j, i_2^j, \dots, i_l^j\}$ be the sequence associated with $v_j (1 \le j \le h-1)$. Let $e_1, \dots, e_{h-1}$ be the edges on the path and let $b_1^j, b_2^j, \dots, b_l^j$ be the bit string labeling $e_j (1 \le j \le h-1)$. Then,*

$$(i_1^1, b_1^1) \dots (i_l^1, b_l^1) \dots (i_1^{h-1}, b_1^{h-1}) \dots (i_l^{h-1}, b_l^{h-1})$$

makes up a signature identifier for

$$s_i, s_i(i_1^1, \dots, i_l^1, \dots, i_1^{h-1}, \dots, i_l^{h-1}).$$

Example 3. In Fig. 16b, we show a general signature tree with $l = 2$, generated for the signature file shown in Fig. 16a. It is easy to see that this general signature tree contains less nodes than the signature tree shown in Fig. 16c, which is generated for the same file.

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. 16b can be stored as shown in Fig. 16d, 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:

1. Let v be the node encountered. Assume that the sequence associated with it is $i_1, i_2, \dots, i_l$ for some l. Then, $s_q[i_1], \dots, s_q[i_l]$ will be checked.
2. Let e be an edge outgoing from v and labeled with a bit string $b_1 b_2 \dots b_l$. Then, if $b_1 b_2 \dots b_l$ matches $s_q[i_1], \dots, s_q[i_l]$, 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 4. Consider the signature file shown in Fig. 16a once again. Assume $s_q = 100\,110\,010\,000$. Then, only part of the general signature tree (marked with thick edges in Fig. 17a) will be searched. On reaching a leaf node, the signature pointed to by the leaf node will be checked against s_q.

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. 17b.)

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.

5.2 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 $i_1\text{th}, i_2\text{th}, \dots,$ and $i_l\text{th}$ columns to divide the whole S into $j (\le 2^l)$ groups: $g_1 = \{s_1^1, s_2^1, \dots, s_{i_1}^1\}, \dots, g_j = \{s_1^j, s_2^j, \dots, s_{i_j}^j\}$ such that:

1. In each $g_k (1 \le k \le j)$, for any two signatures s_a^k and s_b^k, we have $s_a^k[i_1] = s_b^k[i_1], \dots,$ 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 $i_z \in \ i_1, i_2, \dots, i_l\}$ such that for any $s_1 \in g_x$ and $s_2 \in g_y$, we have $s_1[i_z] \ne s_2[i_z]$.
3. $\max\{|g_1|, \dots, |g_j|\} - \min\{|g_1|, \dots, |g_j|\}$ 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 $\{i_1, i_2, \dots, i_l\}$ and j leaf nodes with each labeled with a g_k. For instance, for the signature file shown in Fig. 16a, we can generate a tree as

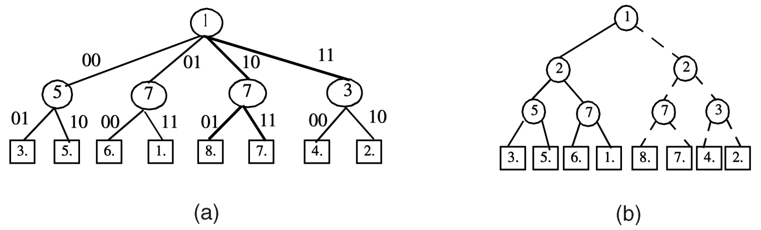


Fig. 17. Illustration for searching general signature trees and signature trees.

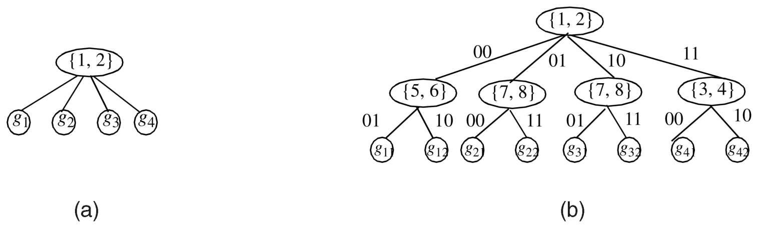


Fig. 18. Illustration of generation of general signature trees.

shown in Fig. 18a. 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 the next step, we consider each $g_k (k = 1, \ldots, j)$ as a single signature file with $i_1\text{th}, i_2\text{th}, \ldots,$ and $i_l\text{th}$ columns removed, and perform the same operations as above. Assume that $T_{g_k} (k = 1, \ldots, j)$ 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. 16a, a tree as shown in Fig. 18b 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 $g_{42} = \{s_2\}$.

We repeat this process 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 $i_1\text{th}, i_2\text{th}, \ldots,$ and $i_l\text{th}$ columns to divide the
    whole $S$ into $j (\leq 2^l)$ groups:
    $g_1 = \{s_1^1, s_2^1, \ldots, s_{i_1}^1\}, \ldots, g_j = \{s_1^j, s_2^j, \ldots, s_{i_j}^j\}$ as described
    above; generate a tree containing a root $r$ and $j$ child
    nodes marked with $g_1, \ldots, g_j$, respectively;
    $c(r) \leftarrow \{i_1, i_2, \ldots, i_l\}$;
    for ($i = 1$ to $j$) do
        {replace the node marked $g_i$ with
          $general\text{-}tree\text{-}generation(g_i, l)$;}
**else** return;
**end**
```

By applying this algorithm with $l = 2$ to the signature file shown in Fig. 16a, a general signature tree as shown in Fig. 16b will be created. Since $\mathrm{O}(\binom{m-l\cdot i}{l} \cdot l \cdot n)$ 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

$$\sum_{i=1}^{\lceil(\log n)/l\rceil} \binom{m - l \cdot i}{l} \cdot l \cdot n.$$

In the above discussion, a very important issue has not yet been addressed. That is, for a file with the signature length m, 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 $v_1, v_2, \ldots,$ and v_k be the child nodes of a node v in T, and $e_1 = (v, v_1), e_2 = (v, v_2), \ldots,$ 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 $c(v) = \{i_1, i_2, \ldots, i_l\}$ and $s_q[i_1] \ldots s_q[il] = b_1 \ldots b_l$. Then, any entry $A[j]$ with j equal to the value of a bit string $b'_1 \ldots b'_l$ should be explored if for any i with $b_i = 1$, we have $b'_i = 1$. It is easy to show that the average number of entries in A, which may be explored, is

$$\tfrac{1}{2^l}\left(2^l + \binom{l}{1} 2^{l-1} + \binom{l}{2} 2^{l-2} + \cdots + 1\right) = \left(\tfrac{3}{2}\right)^l. \qquad (19)$$

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

However, if k is large, we cannot store the children of a node in an array as shown 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 $\mathrm{O}(2^{l-1})$ time is needed. So, in this case, the average number of the nodes to be checked is estimated by $\mathrm{O}\big(\frac{(3/2)^{\lceil \log_2 n \rceil} - 1}{(3/2)^l - 1}\big)$.

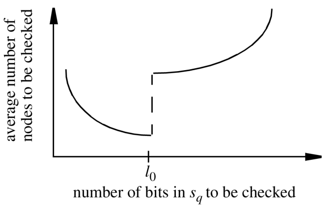


Fig. 19. Average number of the nodes to be checked.

Assume that, when $k \leq 2^{l_0}$ 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 nodes to be checked when searching a general signature tree is of the pattern shown in Fig. 19.

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.

6 Experiment Results

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.

The tests are organized into two groups. In the first group, we test seven methods: sequential signature files (SSF for short) [13], [15], bit-slice files (BSSF for short) [19], compressed bit-slice files (CBSSF for short) [14], S-trees [11], improved S-trees (which uses the cubic algorithm to find α-seed and β-seed) [28], (MLSF for short) [23], and the signature trees (ST for short) discussed in this paper. In the second group, we compare the signature tree approach with different versions of general signature trees.

6.1 Experiment I

In this experiment, we apply the seven methods mentioned above 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 and the time required to satisfy a query. For each query, an average of 20 measurements was taken.

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

For SSF, S-trees, and the signature tree method, an entry in a signature file contains two fields: a signature and an object identifier as shown in Fig. 20a. A bit-slice file is stored as shown in Fig. 2b and a compressed bit-slice file is stored as shown in Fig. 3. In addition, an S-tree is stored as shown in Fig. 4a and a multilevel signature file is stored as shown in Fig. 5. A signature tree is stored as discussed in Section 4.2. Each entry in an internal node of an S-tree also contains two fields: a signature and a pointer to a child page while the 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. 20b for illustration.)

TABLE 1
Considered Parameters and the Tested Values for Each Parameter

parameters \ data	groupI	groupII	groupIII	groupIV
number of signatures (×1024)	50	100	50	100
signature size/weight (in bits)	64/32	64/16	128/64	128/32
page size (in KB)	1	2	1	2

Fig. 21 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 Fig. 21, we can see that the signature tree structure outperforms all the other three strategies. First, we discuss why the signature tree is better than the S-tree. In this test, the size of each page is 1 K. So, each page can accommodate 10 signatures (and the corresponding OIDs) and all the signatures are stored in $5 \times 1,024 = 5,120$ pages. However, for the S-tree, to increase the filter ability of the signatures in an internal node, a page should not be fully populated since in this case, a signature in an internal node will be with too many 1s, degrading the performance dramatically. We have tested different population ratios from 30 to 80 percent of the page capacity and report the average test results over 30 percent, 40 percent, 50 percent, 60 percent, and 70 percent of the page capacity since when each page is 80 percent populated, the performance is much worse than when each page is 70 percent populated. If each page is 70 percent populated, then all the signatures need 7,312 pages to store instead of 5,120 pages. The number of the internal page nodes is about 824 and the outdegree of an internal page node cannot be larger than 7. However, although the number of the internal nodes of the signature tree is about $\frac{1}{2}(50 \times 1,024) = 25,100$, each node only occupies 30 bits and each page can accommodate 32 nodes, i.e., a subtree of height 5. Then, the number of the internal page nodes is about 785. More importantly, the outdegree of an internal page node can be up to 16. So, the height of the signature tree should be lower than that of the S-tree. Another reason why the signature tree outperforms the S-tree is that each

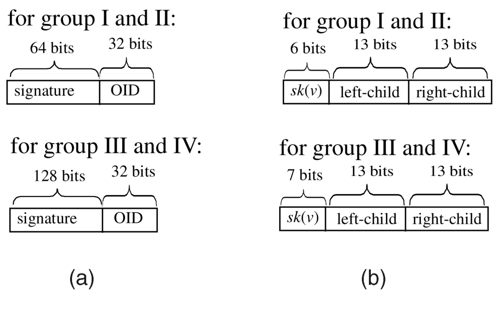
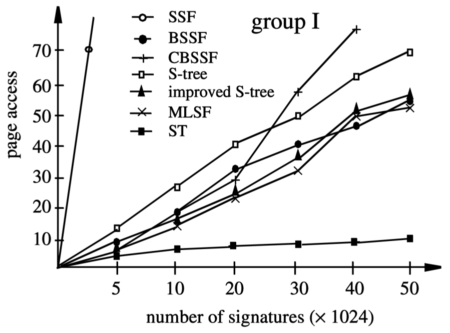


Fig. 20. Illustration for storage structures.



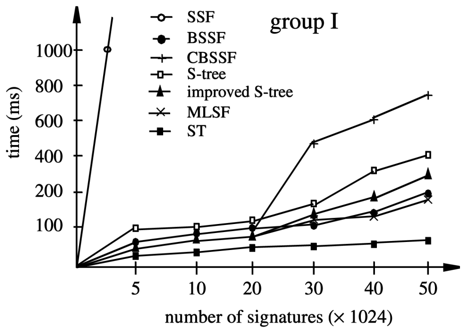
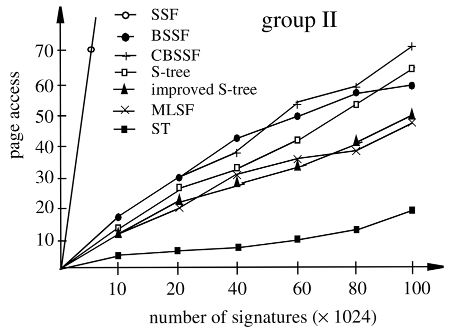


Fig. 21. Test results of group 1.



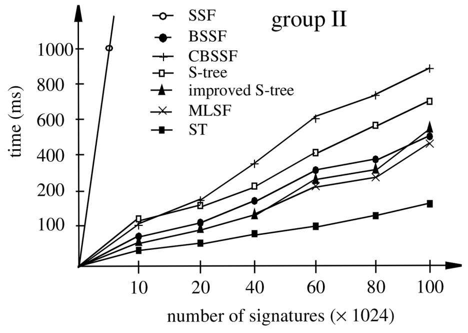


Fig. 22. Test results of group 11.

internal page node of the signature tree is a tree itself (a nonlinear structure) while each internal node of the S-tree is a set of signatures (a linear structure). Normally, a nonlinear structure should be stronger as a filter than a linear structure.

Using the bit-slice file strategy, all the signatures are stored in 64 files with each containing $50 \times 1,024$ bits. So, the size of each file is 50 pages. For evaluating a query s_q, we will check these files one by one. But, for a bit set to 0 in s_q, the corresponding file need not be checked. In addition, the result of checking a bit $s_q[i]$ against the ith file can be used to reduce the number of the pages to be checked when examining the $s[i+1]$ against the $(i+1)$th file. This is the main reason why the bit-slice file is better than the S-tree in some cases, especially when the query weight is low. However, as a filter, the bit-slice file is not so efficient as the signature tree since each time when checking a page for an internal node, the signature tree can examine up to 5 bits, which may need more than one page access by means of the bit slice file.

In addition, we notice that the compressed bit-slice file is much worse than the bit-slice file. It is because in this test, the signature file is not a sparse one and the size of all linked bucket lists is actually much bigger than that of all bit-slice files.

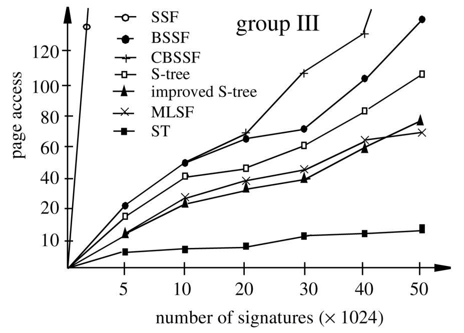
The multilevel signature file is slightly better than the S-tree. Theoretically, the multilevel signature file needs more space than the S-tree. But, in practice, each node in the S-tree is at most 70 percent populated. So, the space occupied by the multilevel signature file is just a little bit more than that of the S-tree. On the other hand, however, a signature at a higher level in the multilevel signature file is normally a more powerful filter than a signature in an internal node in the S-tree.

Fig. 22 shows the test results for group II. From this, we can see that when the weight of signatures in a signature file is low, the performance of the S-tree becomes better. It is because in this case, the signatures in the internal nodes of a S-tree will be less heavily populated. In contrast, the performance of the bit-slice file degrades because the more 1s a bit-slice file has, the more chance a bit in a next bit-slice file will be checked. However, the compressed bit-slice file is better since less 1s are in the file, leading to less addresses stored in the linked bucket lists. Also, the signature tree becomes worse because in this case, we have much more 0s than 1s and the tree cannot be well balanced.

Fig. 23 and 24 show the test results for groups III and IV, respectively. In these two cases, since the signatures are much longer, the number of the internal nodes of a S-tree are greatly increased since each internal node needs more space for the signatures stored, which are used as filters. However, the size of an internal node of a signature tree is only one bit augmented. So, the number of page accesses is almost not changed. The BSSF becomes worse because for longer signatures, more bit-slice files need to be checked.

In the above tests, the impact of compressed bit-slice files cannot be observed since each signature file generated is not a sparse matrix. To see in what cases the compression of bit-slice signature files is beneficial, we did an extra test, by which the number of 1s in signatures is limited. The test result is shown in Fig. 25. For the test, the signature file contains $50 \times 1,024$ signatures.

From this, we can see that, when a signature file contains less 1s, the compression is quite effective. However, as a signature file is populated with more 1s, the benefit



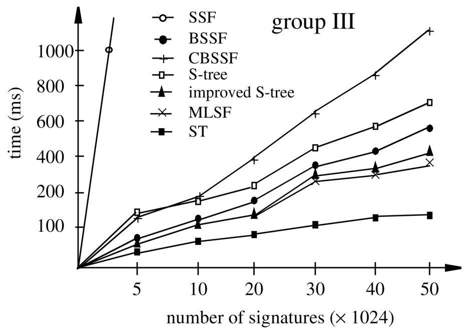
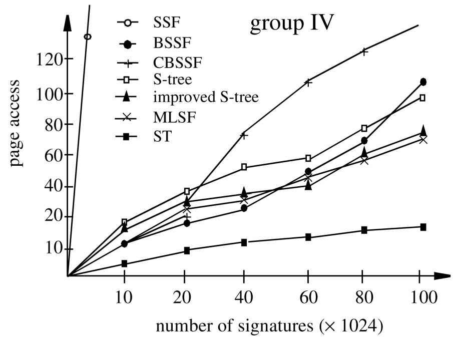


Fig. 23. Test results of group III.



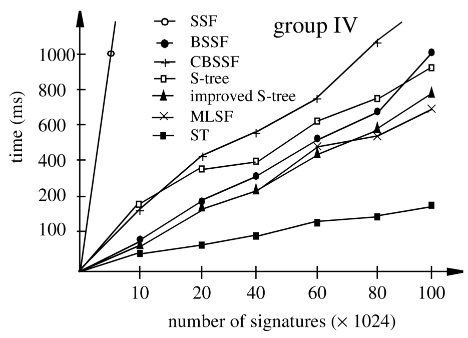


Fig. 24. Test results of group IV.

diminishes. Especially, when a signature file is half-populated with 1s, the compressed bit-slice file is worse than the bit-slice file without compression.

Fig. 26 shows the impact of page sizes. For this test, the number of signatures is $50 \times 1,024$.

From Fig. 26, we can see that the page access of both BSSF and the improved S-tree reduces almost linearly as the size of pages increases. However, the page access of the signature tree decreases a little bit faster. It is because the larger a page is, the less space in it is wasted. This can also be seen from Fig. 15. Smaller page sizes make more space in a page not used. Finally, the weight of a query signature (i.e., the percentage of 1-bits in a query signature) affects BSSF, the S-tree, and signature trees greatly. Fig. 27 shows the number of page access when the three methods are used to search a signature file containing $50 \times 1,024$ signatures to locate query signatures with different weights.

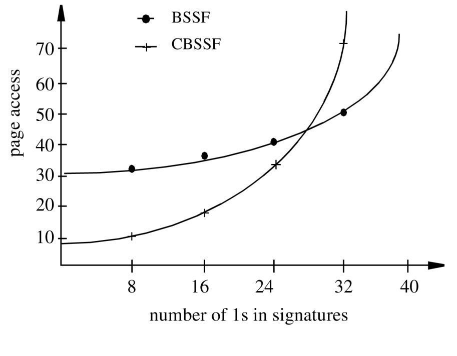


Fig. 25. Comparison of bit-slice and compressed bit-slice signature files.

From this, we can see that as the weight of a query signature increases, the searching time of both the signature tree method and the S-tree reduces. It is because each bit set to 1 in the query signature may cut off a subtree. For BSSF, however, each bit set to 1 in the query signature entails more access to bits in bit-slice files. The weight of a query signature has almost no impact on SSF.

6.2 Experiment II

We have also experimentally compared 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 is 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 comparison purpose, a general signature trees is also stored page-wise as illustrated in Fig. 28.

The considered parameters and the tested values for each parameter are given in Table 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. 29a. 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,

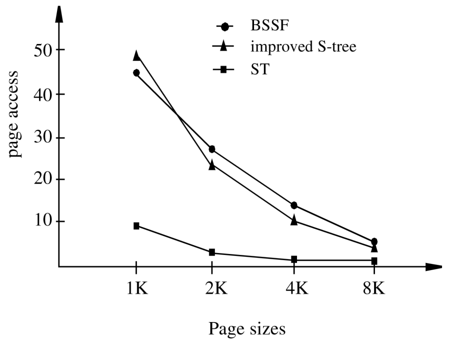


Fig. 26. Impact of page sizes.

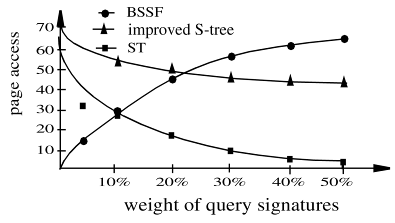


Fig. 27. Impcat of weight of query signatures.

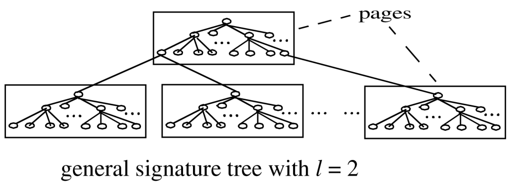


Fig. 28. Illustration for tree storage.

TABLE 2
Considered Parameters and the Tested Values for Each Parameter

parameters \ data	groupV	groupVI	groupVII	groupVIII
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

respectively. (See Fig. 29b 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 four pointers to its child nodes. (See Fig. 29c for illustration.)

Fig. 30 shows the test results for group V. 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. 31.

In Fig. 32 and Fig. 33, we show the results of Group VII and Group VIII, respectively. These results also confirm the above analysis.

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. 34 shows the number of page access when the three methods are used to search a signature file containing $100 \times 1,024$ 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.

7 Conclusion

In this paper, a new method to organize signature files has been proposed. The main idea of this approach is the concept of signature identifiers, which can be used to distinguish signatures in a file from each other. Based on this concept, we transform a signature file into a balanced signature tree, in which each edge is labeled with 0 or 1, and each node is associated with a number, indicating which bit in a query signature to check. In this way, the searching of a signature file is replaced by a binary tree searching. In addition, a general signature tree structure is discussed. In such a general structure, each node is associated with a sequence of integers showing what bits in a query signature to check. This can be quite useful in the cases where the signatures are very long. Next, a probabilistic analysis of

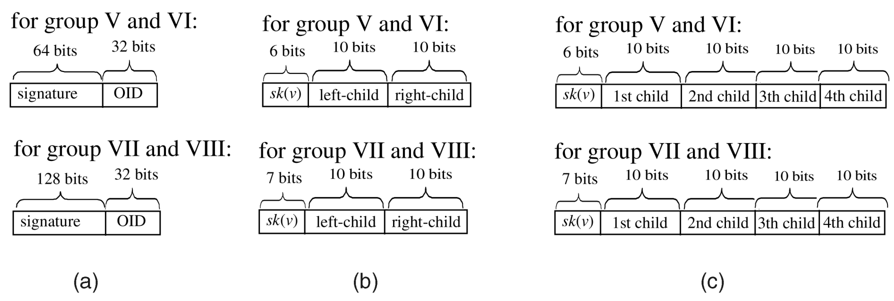


Fig. 29. Illustration for storing signature file entries and internal nodes in signature trees.

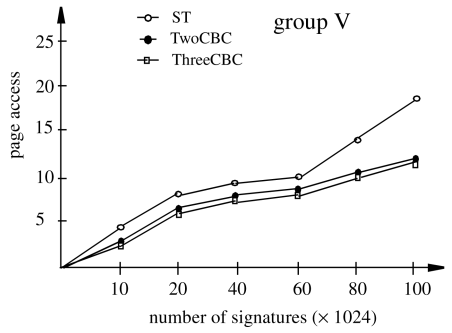


Fig. 30. Test results of group I.

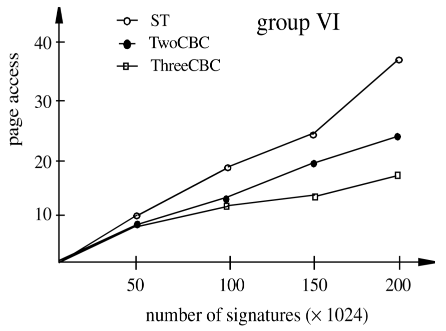


Fig. 31. Test results of group II.

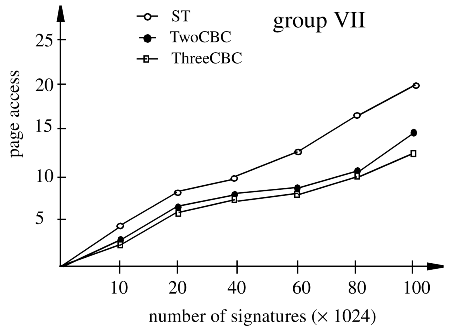


Fig. 32. Test results of group III.

signature tree searching is conducted, which shows a sublinear time complexity. Especially, if a query signature is half-populated, the cost of searching a signature tree is bounded by $O(n^{0.5})$. In order to show the efficiency of the signature tree method, a series of experimental tests were performed to compare it with the most important existing methods, such as sequential signature files, bit-slice signature files, compressed bit-slice signature files, S-trees, and multilevel signature files. All the conducted tests show that the signature tree method significantly outperforms all such methods. Finally, the maintenance of a signature tree is discussed, which shows that it can be done in a way similar to B^+-trees.

Acknowledgments

The authors are grateful to the anonymous referees for their invaluable comments to improve the manuscript. The manuscript is an extended version of the article "On the Signature Trees and Balanced Signature Trees," published in the *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pp. 742-753, 2005. The authors are supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

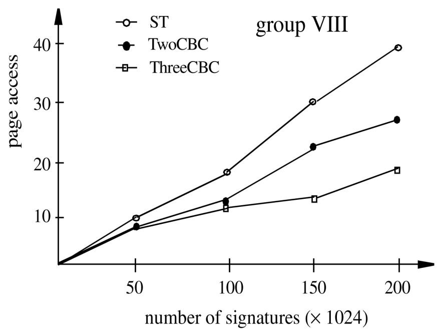


Fig. 33. Test results of group IV.

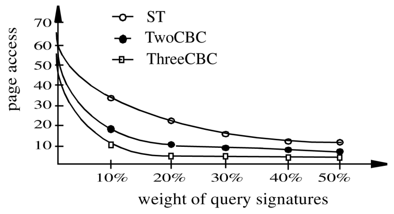


Fig. 34. Test results.

References

[1] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon, "Querying Documents in Object Databases," *Int'l J. Digital Libraries,* vol. 1, no. 1, pp. 5-19, Jan. 1997.
[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* London: Addison-Wesley Publishing Company, 1974.
[3] R. Bayer and K. Unterrauer, "Prefix B-Tree," *ACM Trans. Database Systems,* vol. 2, no. 1, pp. 11-26, 1977.
[4] W.W. Chang and H.J. Schek, "A Signature Access Method for the STARBURST Database System," *Proc. 19th Very Large Data Bases Conf.,* pp. 145-153, 1989.
[5] Y. Chen and Y.B. Chen, "Signature File Hierarchies and Signature Graphs: A New Indexing Method for Object-Oriented Datbases," *Proc. ACM Symp. Applied Computing (SAC '04),* pp. 724-728, 2004.
[6] S. Christodoulakis and C. Faloutsos, "Design Consideration for a Message File Server," *IEEE Trans. Software Eng.,* vol. 10, no. 2, pp. 201-210, 1984.
[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 Information Systems,* vol. 4, no. 4, pp. 345-386, 1986.
[8] R.V. Churchill, *Operational Mathematics.* New York: McGraw-Hill Book Company, 1958.
[9] P. Ciaccia and P. Zezula, "Declustering of Key-Based Partitioned Signature Files," *ACM Trans. Database Systems,* vol. 21, no. 3, pp. 295-338, 1996.
[10] M. Crochemore and W. Rytter, *Text Algorithms.* New York: Oxford Univ. Press, 1994.
[11] U. Deppisch, "S-Tree: A Dynamic Balanced Signature Index for Office Retrieval," *Proc. ACM SIGIR Conf.,* pp. 77-87, Sept. 1986.
[12] D. Dervos, Y. Manolopulos, and P. Linardis, "Comparison of Signature File Models with Superimposed Coding," *J. Information Processing Letters 65,* pp. 101-106, 1998.
[13] C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys,* vol. 17, no. 1, pp. 49-74, 1985.
[14] C. Faloutsos and R. Chan, "Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison," *Proc. 14th Int'l Conf. Very Large Data Bases,* pp. 280-293, Aug. 1988.

[15] C. Faloutsos, "Signature Files," *Information Retrieval: Data Structures & Algorithms,* W.B. Frakes and R. Baeza-Yates, eds., pp. 44-65, New Jersey: Prentice Hall, 1992.

[16] C. Faloutsos, R. Lee, C. Plaisant, and B. Shneiderman, "Incorporating String Search in Hypertext System: User Interface and Signature File Design Issues," *HyperMedia,* vol. 2, no. 3, pp. 183-200, 1990.

[17] P. Flajolet and C. Puech, "Partial Match Retrieval of Multidimentional Data," *J. ACM,* vol. 33, no. 2, pp. 371-407, Apr. 1986.

[18] D. Harman, E. Fox, and R. Baeza-Yates, "Inverted Files," *Information Retrieval: Data Structures & Algorithms,* W.B. Frakes and R. Baeza-Yates, eds., pp. 28-43, New Jersey: Prentice Hall, 1992.

[19] Y. Ishikawa, H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 247-256, May 1993.

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

[21] 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. Information Science,* vol. 41, no. 7, pp. 508-534, 1990.

[22] S. Kocberber and F. Can, "Compressed Multi-Framed Signature Files: An Index Structure for Fast Information Retrieval," *Proc. ACM Symp. Applied Computing (SAC '99),* pp. 221-226, 1999.

[23] D.L. Lun, Y.M. Kim, and G. Patel, "Efficient Signature File Methods for Text Retrieval," *IEEE Trans. Knowledge and Data Eng.,* vol. 7, no. 3, June 1995.

[24] W. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," *Proc. ICIC '92—Second Int'l Conf. Data and Knowledge Eng.: Theory and Application,* pp. 616-622, Dec. 1992.

[25] D.R. Morrison, "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric," *J. Assoc. for Computing Machinery,* vol. 15, no. 4, pp. 514-534, Oct. 1968.

[26] J. Riordan, *Comninatorial Identities.* New York: Wiley, 1968.

[27] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos, "Improved Methods for Signature-Tree Construction," *Computer J.,* vol. 43, no. 4, pp. 301-314, 2000.

[28] E. Tousidou, P. Bozanis, and Y. Manolopoulos, "Signature-Based Structures for Objects with Set-Values Attributes," *Infromation Systems,* vol. 27, no. 2, pp. 93-121, 2002.

[29] H.S. Yong, S. Lee, and H.J. Kim, "Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases," *Proc. 10th Int'l Conf. Data Eng.,* pp. 518-525, Feb. 1994.

[30] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted Files Versus Signature Files for Text Indexing," *ACM Trans. Database Systems,* vol. 23, no. 4, pp. 453-490, Dec. 1998.

Yangjun Chen received the BS degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and the Diploma and PhD degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a postdoctor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a senior engineer at the German National Research Center of Information Technology (GMD) for more than two years. After a short stay at Alberta University, he joined the Department of Applied Computer Science at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, constraint satisfaction problem, graph theory, and combinatorics. He has about 100 publications in these areas.

Yibin Chen is currently an undergraduate student studying in the Department of Electrical and Computer Engineering, University of Waterloo, Canada.