# On the Signature Trees and Balanced Signature Trees

Yangjun Chen

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

## 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.*

**Key Words:** 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 [14, 24] 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, 23], 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.

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

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 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 constructing signature files have been proposed, such as sequential signature files, bit-slice files, S-trees, and its 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.

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. Section 4 is devoted to the maintenance of signature trees. In Section 5, we report the experiment results. Finally, Section 6 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 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 the signature file, by introducing auxiliary data structure, as well

as by exploiting parallel computer architecture [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 $k$ 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 [4], 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 $p$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".

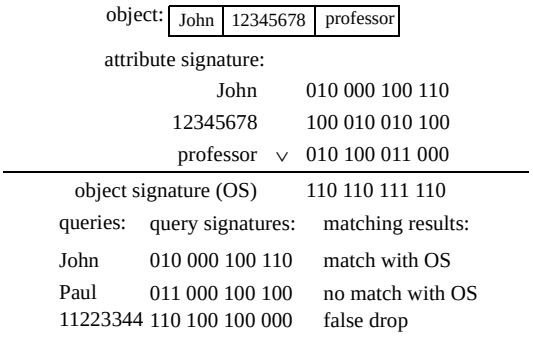



Fig. 1. Signature generation and comparison

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(a) is a quite simple signature file.

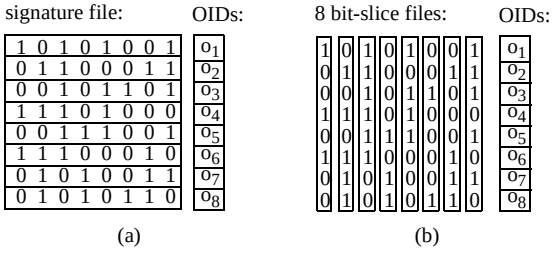



Fig. 2. Illustration of sequential and bit-slice signature files

### 2.2 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 [15]. 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. 2(b).

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. 2(b) and find that only three positions: 1st, 4th and 6th 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 2nd 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 3rd 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 4th 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*.

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 set signature requires about *m* disk accesses, one for each bit-slice file.

### 2.3 S-trees

Similar to a $B^+$-tree, an S-tree is a height balanced multiway tree [9]. 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 the object is accessed by the *oid* and *s* is its signature. Let *v* be the father node of *v*'. Then, there exists a signature in *v*, whose value is obtained by superimposing all the signatures in *v*'. See Fig. 3 for illustration.

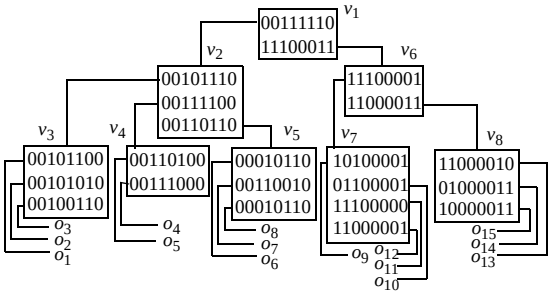



Fig. 3. Illustration for S-tree and node splitting

To retrieve a query signature $s_q$ = 00110000, we search the S-tree top-down. However, more than one paths may be visited. For example, the first signature in the root $v_1$ shown in Fig. 3(a) 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 $\{o_4, o_5\}$, 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* become 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 $v_\alpha$ and $v_\beta$. 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 $v_\alpha$. 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 $v_\beta$. Any of the rest *K* - 1 signatures is assigned to $v_\alpha$ or $v_\beta$, depending on whether it is closer to $v_\alpha$ or $v_\beta$. The two new signatures (denoted $s_\alpha$ and $s_\beta$) to be put into the parent node are obtained by superimposing the signatures in $v_\alpha$ and $v_\beta$, respectively. See Fig. 3(b) 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* [21, 22]. They elaborate the selection of the α-seed and the β-seed 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 [22], two algorithms were discussed. One needs $O(l^2)$ time to determine α-seed and 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.

In the following, we discuss a quite different method, called signature trees, by means of which all the drawbacks of S-trees can be removed.

## 3. Signature trees

A signature tree works for a signature file is just like a *trie* [16, 19] for a text. But in a signature tree, each path is a *signature identifier* 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, 5] 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 $s_i[j] \in \{0, 1\}$ ($j = 1, ..., m$). We also use $s_i(j_1, ..., j_h)$ to denote a sequence of pairs w.r.t. $s_i$: $(j_1, s_i[j_1])(j_2, s_i[j_2]) \ldots (j_h, s_i[j_h])$, where $1 \le j_k \le m$ for $k \in \{1, ..., h\}$.

**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: $j_1, ..., j_h$ such that for any $k \ne i$ ($1 \le k \le n$) we have $s_i(j_1, ..., j_h) \ne s_k(j_1, ..., j_h)$, then we say $s_i(j_1, ..., j_h)$ identifies the signature $s_i$ or say $s_i(j_1, ..., j_h)$ is an identifier of $s_i$ w.r.t. $S$.

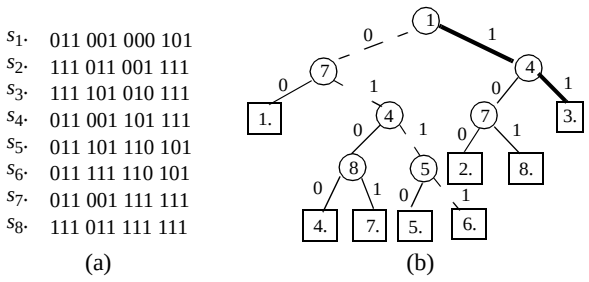
For example, in Fig. 4(a), $s_6(1, 7, 4, 5) = (1, 0)(7, 1)(4, 1)(5, 1)$ is an identifier of $s_6$ since for any $i \ne 6$ we have $s_i(1, 7, 4, 5) \ne s_6(1, 7, 4, 5)$. (For instance, $s_1(1, 7, 4, 5) = (1, 0)(7, 0)(4, 0)(5, 0) \ne s_6(1, 7, 4, 5)$, $s_2(1, 7, 4, 5) = (1, 1)(7, 0)(4, 0)(5, 1) \ne s_6(1, 7, 4, 5)$, and so on. Similarly, $s_1(1, 7) = (1, 0)(7, 0)$ is an identifier for $s_1$ since any $i \ne 1$ we have $s_i(1, 7) \ne s_1(1, 7)$.)

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 $s_i \ne s_j$ for $i \ne j$ and $|s_k| = m$ for $k = 1, ..., n$, 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, ..., $n$, used as pointers to $n$ different positions of $s_1$, $s_2$ ... and $s_n$ in $S$. Let $v$ be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.
3. Each internal node $v$ is associated with a number, denoted $sk(v)$, to tells which bit will be checked.
4. Let $i_1, ..., i_h$ 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 $i$th signature in $S$, *i.e.*, $p(v) = i$). Let $p_1, ..., p_h$ 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 $s_i$, $s_i(j_1, ..., j_h)$.

**Example 1.** In Fig. 4(b), we show a signature tree for the signature file shown in Fig. 4(a). 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 $s[4]$ will 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 5th bit of $s$: $s[5]$ will be checked. Also, see the path consisting of the dashed edges in Fig. 4(b), 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).




(a) (b)

Fig. 4. A signature file and a signature tree

In the following subsections, 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 $O(N \cdot min(m, \log N))$ time, where $N$ represents the number of the signatures in the signature file.

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 $v$. $s$ can not be the same as $v$ since in $S$ there is no signature which is identical to anyone else. (If we have two identical signa-

tures, 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**      {$i$ ← $sk(v)$;
**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**    $w$ ← $v$; 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 $u$
**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. 5, we trace the above algorithm against the signature file shown in Fig. 4(a).

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

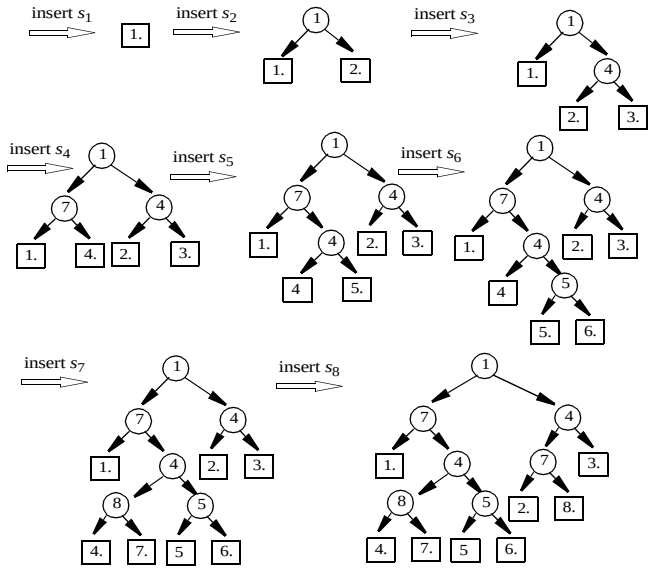



Fig. 5. Sample trace of signature tree generation

an identifier for $s$.

*Proof.* Let $S = s_1.s_2 \ldots .s_n$ be a signature file and $T$ a signature tree for it. Let $P = v_1e_1 \ldots v_{g-1}e_{g-1}v_g$ 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 Fig. 6(a) or (b). (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.

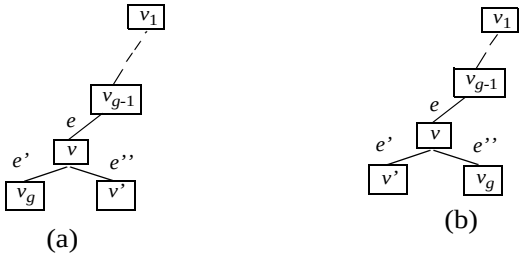



Fig. 6. Inserting a node $v$ into $T$

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, which needs at most O(*min*(*m*, log*N*)) time, where *m* is the length of a signature and *N* represents the number of signatures in a signature file. Thus, we have the following proposition.

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

*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 *i*th 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.

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. $R \leftarrow \varnothing$.
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*).
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. 4 once again.

Assume $s_q$ = 000 100 100 000. Then, only part of the signature tree (marked with thick edges in Fig. 7) 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 3 signatures need to be checked while a signature file scanning will check 8 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 $\lambda$ 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 the next subsection in more detail.

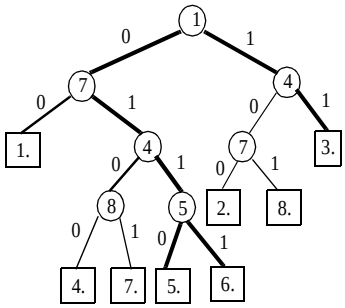


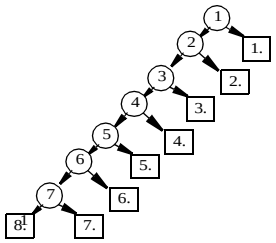

Fig. 7. Signature tree search

### 3.4. Balanced signature trees

A signature tree can be quite skewed as shown in Fig. 8.

$s_1$· 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. 8. A skewed signature tree

But the tree shown in Fig. 9 is completely balanced for the same signature file. However, the signature identifiers for the signatures are different from those shown in Fig. 8(b).

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

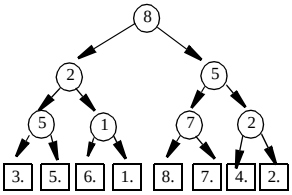
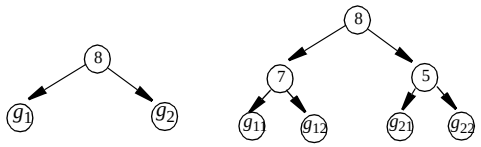



Fig. 9. A balanced signature tree

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

- *Weight-based method*

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 $i$th 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 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 $s_{i_p}[j] = 0$ ($p = 1, \ldots, k$) 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. 10(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 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. 10(b). We repeat this process until the leaf nodes of a generated tree cannot be divided any more.




(a) (b)

Fig. 10. Illustration of generation of balanced signature trees

In Fig. 10(a), $g_1 = \{s_1, s_3, s_5, s_6\}$ and $g_2 = \{s_2, s_4, s_7, s_8\}$; and in In Fig. 10(b), $g_{11} = \{s_3, s_5\}$, $g_{12} = \{s_6, s_1\}$, $g_{21} = \{s_8, s_7\}$, 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 $g_1 = \{ s_{i_1}, s_2, \ldots, s_{i_k} \}$ 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 $s_{i_q}[j] = 1$ ($q = k + 1, \ldots, N$)

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**

By applying this algorithm to the signature file shown in Fig. 8(a), 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 $O(N \cdot m \cdot \log N)$.

- *Analysis of balanced signature trees*

A signature tree is a binary tree generated by exploiting arbitrary bit difference. We claim that if a signature file involves equal number of 1s and 0s, or say, if the probability of appearances of 1s or 0s is 1/2, the signature tree over it is likely balanced. In the following, we give a probabilistic analysis to show that this property holds for most cases.

Let $T_n$ be a family of signature trees built from $n$ signatures. Each signature is considered as a random bit string containing 0s and 1s. We assume that the probability of appearances of 0 and 1 in a string is equal to $p$ and $q = 1 - p$, respectively. The occurrence of these two values in a bit string is independent of each other.

To study the average length of paths from the root to a leaf, we check the *external path length* $L_n$ - the sum of the lengths of all paths from the root to all leaf nodes of a signature tree in $T_n$. Note that in a signature tree, the $n$ signatures are split randomly into the left subtree and the right subtree of the root. Let $X$ denote the number of signatures in the left subtree. Then, for $X = k$, we have the following recurrence:

$$L_n = \begin{cases} n + L_k + L_{n-k}, & \text{for } k \neq 0, n \\ \text{undefined}, & \text{for } k = 0, k = n \end{cases}$$

where $L_k$ and $L_{n-k}$ represent the external path length in the left and right subtrees of the root, respectively. Note that a signature tree is never degenerate (i.e., $k = 0$ or $k = n$). So one-way branching on internal nodes never happens. The above formula is a little bit different from the formula established for the external path length of a binary tree [16]:

$$B_n = n + B_k + B_{n-k}, \qquad \text{for all } k = 0, 1, 2, \ldots, n,$$

where $B_k$ represents the sum of the lengths of all paths from the root to all leaf nodes of a binary tree having $k$ leaf nodes.

According to [16], the expectation of $B_n$ is

$$\boldsymbol{E}B_0 = \boldsymbol{E}B_1 = 0,$$

$$\boldsymbol{E}B_n = n + \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} (B_k + B_{n-k}), \quad n > 1.$$

When $p = q = 0.5$, we have

$$\boldsymbol{E}B_n = \sum_{k=2}^{n} (-1)^k \binom{n}{k} \frac{k}{1-2^{1-k}}.$$

For large $n$, the following holds:

$$\boldsymbol{E}B_n = n\log_2 n + n[\frac{\gamma}{L} + \frac{1}{2} + \delta_1(\log_2 n)\,] - \frac{1}{2}L + \delta_2(\log_2 n)$$

where $L = \log_e 2$, $\gamma = 0.577...$ is the *Euler* constant, $\delta_1(x)$ and $\delta_2(x)$ are two periodic functions with small amplitude and mean zero (see [16] for a detailed discussion).

In a similar way to [16], we can obtain the following formulae:

$$\boldsymbol{E}L_0 = \boldsymbol{E}L_1 = 0,$$

$$\boldsymbol{E}L_n = n(1 - p^n - q^n) + \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} (B_k + B_{n-k}),\ n > 1.$$

When $p = q = 0.5$, we have

$$\boldsymbol{E}L_n = \sum_{k=2}^{n} (-1)^k \binom{n}{k} \frac{k2^{1-k}}{1-2^{1-k}} = \boldsymbol{E}B_n - n + \delta_{n,1},$$

where $\delta_{n,1}$ represents the *Kronecker delta function* (see [20]), which is 1 if $n = 1$, 0 otherwise.

From the above analysis, we can see that for large $n$ we have the following:

$$\boldsymbol{E}L_n = O(n\log_2 n).$$

This shows that the average value of the external path length is asymptotically equal to $\log_2 n$, which implies that a signature tree is normally balanced. In fact, in term of the algorithm for generating signatures for words [10], the generation of signatures is a random process. Therefore, in a large signature file, we expect that 0's and 1's are approximately equally distributed and the corresponding signature tree is almost balanced.

## 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 4.1. Then, we discuss the case that a signature tree cannot entirely fit in main memory in 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.

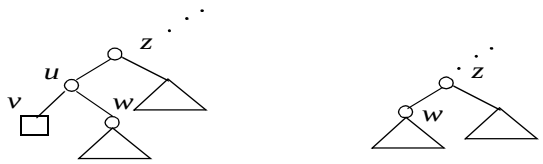
When a signature *s* is added to a signature file, the corresponding signature tree can be changed by simply running the procedure *insert*() with *s* as the input (see 3.2).

When a signature is removed from the signature file, we need to change the corresponding signature tree as follows:

(i) Let *z*, *u*, *v*, and *w* are the nodes as shown in Fig. 11(a) and assume that *v* is a pointer to the signature to be removed.

(ii) 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. 11(b).




(a) (b)

Fig. 11. Illustration for deleting a signature

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. 12, in which each node corresponds to a page containing a binary tree.

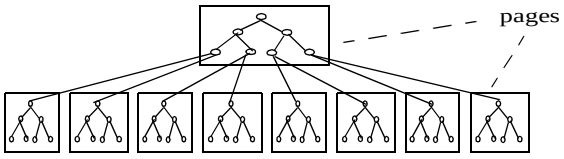



Fig. 12. An external signature tree

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: $b_n(r_n, a_{n1}, ..., a_{ni})$, where $b_n$ represents a subtree of *T*, $r_n$ is its root and $a_{n1}, ..., a_{ni_n}$ are its leaf nodes. Each 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 $m$ is a child page node of $n$. Then, $m$ is of the form: $b_m(r_m, a_{m1}, ..., a_{mi_m})$, where $b_m$ represents a binary tree, $r_m$ is its root and $a_{m1}, ..., a_{mi_m}$ are its leaf nodes. If $m$ is an internal page node, $a_{m1}, ..., a_{mi_m}$ will have the same structure as $a_{n1}, ..., a_{ni_n}$ described in (1). If $m$ is a leaf node, each $a_{mi_j} = p(s)$, 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
$|b| \le 2^k$,
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 $2^{k-1} \le |b| \le 2^k$ holds for each node in $ET$, it is said to be balanced; otherwise, it is unbalanced. However, according to the analysis of 3.4, an external signature tree is normally balanced, i.e., $2^{k-1} \le |b| \le 2^k$ holds for almost every page node in $ET$.

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

- *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 4.1. If for the binary tree $b$ in $m$ we have $|b| > 2^k$, the following node-splitting will be conducted.

1. Let $b_m(r_m, a_{m1}, ..., a_{mi_m})$ 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 $b_{m1}(r_{m1}, a_{m1}, ..., a_{mi_j})$ ($i_j < i_m$) is the subtree rooted at $r_{m1}$ and $b_{m2}(r_{m1}, a_{mi_{j+1}}, ..., a_{mi_m})$ is rooted at $r_{m2}$. We allocate a new page $m$' and put $b_{m2}(r_{m1}, a_{mi_{j+1}}, ..., a_{mi_m})$ into $m$'. Afterwards, promote $r_m$ into the parent page node $n$ of $m$ and remove $b_{m2}(r_{m1}, a_{mi_{j+1}}, ..., a_{mi_m})$ 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.

- *Deletion of binary nodes*

When a node is removed from a signature tree, it is always removed from the leaf level as discussed in 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 $b_m(r_m, a_{m1}, ..., a_{mi_m})$ and $b_{m'}(r_{m'}, a_{m'1}, ..., a_{m'i_{m'}})$ 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 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. 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.

We have tested four methods: SSF [11, 12], BSSF [15], S-trees [9], improved S-trees (which uses the cubic algorithm to find α-seed and β-seed) [22], and the signature trees discussed in this paper; and applied them 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.

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. 13(a). A bit-slice file is stored as

shown in Fig. 2(b).

Tabel 1:

| 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 |

In addition, an S-tree is stored as shown in Fig. 3(a) and a signature tree is stored as discussed in 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. 13(b) for illustration.)

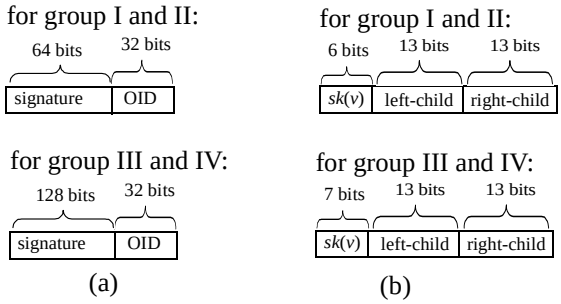



Fig. 13. Illustration for storage structures

Fig. 14 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.

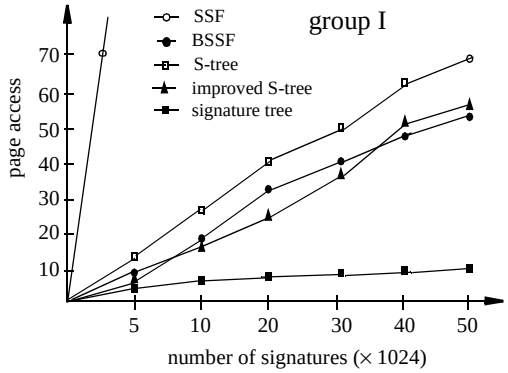



Fig. 15. Test results of group II

From Fig. 14, 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 signature (and the corresponding OIDs) and all the signatures are stored in $5 \times 1024 = 5120$ 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% of the page capacity and report the average test results over 30%, 40%, 50%, 60% and 70% of the page capacity since when each page is 80% populated, the performance is much worse than when each page is 70% populated. If each page is 70% populated, then all the signatures need 7312 pages to store instead of 5120 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 $\frac{1}{2}(50 \times 1024) = 25100$, 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 internal page node of the signature tree is a tree itself (a non-linear structure) while each internal node of the S-tree is a set of signatures (a linear structure). Normally, a non-linear 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 $50 \times 1024$ 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 needn't be checked. In addition, the result of checking a bit $s_q[i]$ against the $i$th 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 accesses by means of the bit slice file.

Fig. 15 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. 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. 16 and 17 show the test results for group 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. .

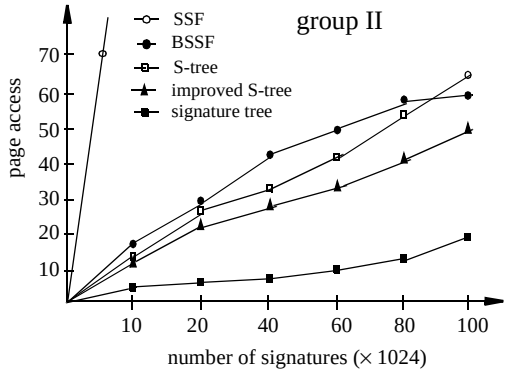



Fig. 14. Test results of group I

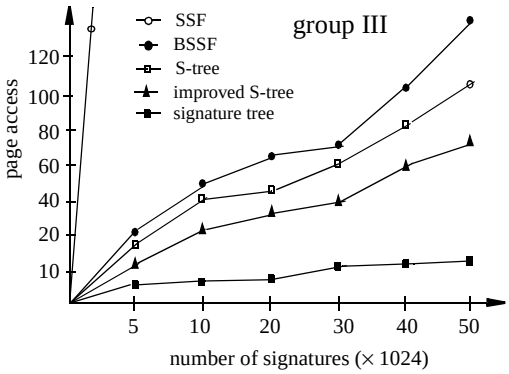



Fig. 16. Test results of group III

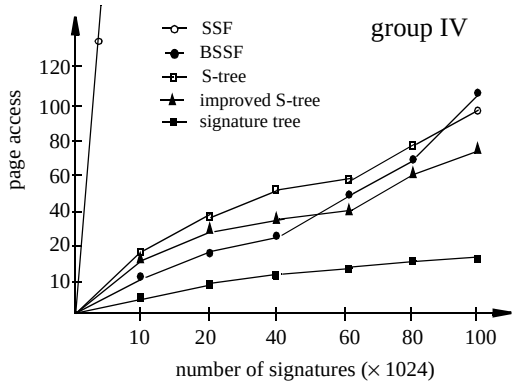



Fig. 17. Test results of group IV

In addition, the weight of a query signature (*i.e.*, the percentage of 1-bits in a query signature) affects BSSF, the S-tree, and signature trees greatly. Fig. 18 shows the number of page access when the three methods are used to search a signature file containing $50 \times 1024$ 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 tree method and the S-tee 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.

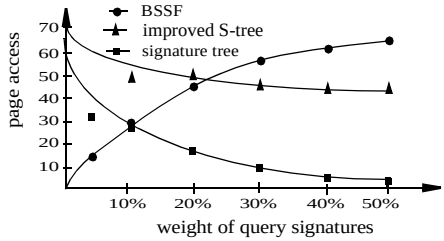



Fig. 18. Test results

## 6. Conclusion

In this paper, a new indexing technique has been proposed. The main idea of this approach is to organize a signature file into a balanced signature tree. In this way, the searching of a signature file can be done in a binary searching manner. In addition, the maintenance of a signature tree is discussed and an experimental test is demonstrated, which shows that using the signature tree structure, the query evaluation can be sped-up significantly.

## 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] Crochemore, M. and Rytter, W., *Text Algorithms*. Oxford University Press, New York, 1994.

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

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

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

[23] 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.

[24] 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.