ELSEVIER

Information Processing Letters 82 (2002) 213–221

Information Processing Letters

www.elsevier.com/locate/ipl

Signature files and signature trees

Yangjun Chen

Department of Business Computing, Winnipeg University, 515 Portage Avenue, Winnipeg, MB, Canada, R3B 2E9

Received 12 January 2001; received in revised form 15 March 2001

Abstract

The signature file method is a popular indexing technique used in information retrieval and databases. It excels in efficient index maintenance and lower space overhead. However, it suffers from inefficiency in query processing due to the fact that for each query processed the entire signature file needs to be scanned. In this paper, we introduce a tree structure, called a signature tree, established over a signature file, which can be used to expedite the signature file scanning by one order of magnitude or more. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Index; Signature file; Signature identifier; Signature tree; Information retrieval

1. Introduction

An important question in information retrieval is how to create a database index which can be searched efficiently for the data one seeks. Today, one or more of the following three techniques have been frequently used: full text searching, inversion and the signature file. Full text searching imposes no space overhead, but requires long response time. In contrast, inversion and the signature file work quickly, but need a large intermediary representation structure (index), which provides direct links to relevant data.

The inverted index excels in query processing efficiency. It is a set of postings lists [15], each of which maps one keyword to a list of links to the data entries containing that keyword. Inverted indices can be implemented as sorted arrays, tries, B-trees and various hashing structures, whereby each real text block address (or document identifier) is stored more than once. The scheme needs to frequently undergo re-organization under intensive information insertion/updating procedures. Recently, a lot of work has been done on the encoding of postings list in the context of document databases [19,23]. Using Golomb's encoding for the integers [13], the size of the inverted index can be reduced to 14% of the indexed data with little or no loss of retrieval effectiveness [23]. However, Golomb's encoding can not be utilized in some applications. For instance, in an object-oriented database system, if the inverted index is used, the postings list will be a a series of pairs of the form: (C, oid), where C represents a class name and *oid* represents an object identifier, not satisfying the encoding condition. Therefore, in the context of object-oriented databases, the inverted file will require much storage space for postings lists [3,14].

The signature file method was originally introduced as a text indexing methodology [10,12]. Nowadays, however, it is utilized in a wide range of applications, such as in office filing [6], hypertext systems [12], relational and object-oriented databases [5,16,18,21, 22], as well as in data mining [1]. Compared to the

E-mail address: ychen2@uwinnipeg.ca (Y. Chen).

0020-0190/01/$ – see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0020-0190(01)00266-6

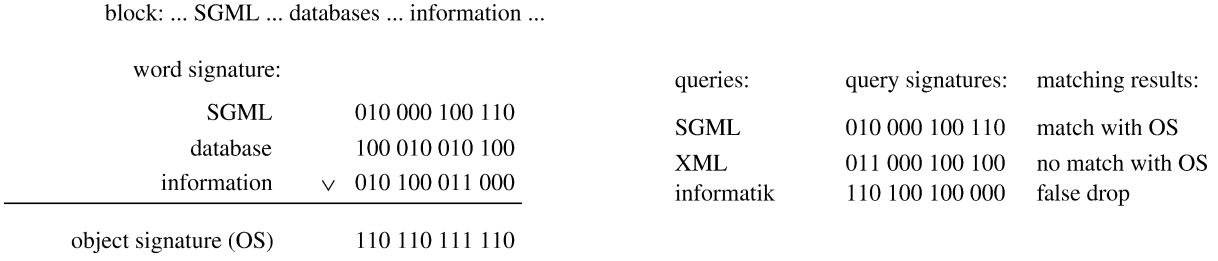

Fig. 1. Signature generation and comparison.

inverted index, the signature file is more efficient in handling new insertions and queries on parts of words. But the scheme introduces information loss. More specifically, its output usually involves a number of false drops, which may only be identified 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 [4,10,11]. Consequently, the signature file method involves high processing and I/O cost. This problem is mitigated by partitioning the signature file, as well as by exploiting parallel computer architecture [7,17,20].

During the creation of a signature file, each word is processed separately by a hashing function. The scheme sets a constant number (m) of 1s in the $[1..F]$ range. The resulting binary pattern is called the word signature. Each text is seen to consists of fixed size logical blocks and each block involves a constant number (D) of non-common, distinct words. The D word signatures of a block are superimposed (bit OR-ed) to produce a single F-bit pattern, which is the block signature stored as an entry in the signature file.

Fig. 1 depicts the signature generation and comparison process of a block containing three words (then $D = 3$), say "SGML", "database", and "information". Each signature is of length $F = 12$, in which $m = 4$ bits are set to 1. When a query arrives, the block signatures are scanned and many nonqualifying blocks are discarded. The rest are either checked (so that the "false drops" are discarded; see below) 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 word signatures. The query signature is then compared to every block signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 1: (1) the block matches the query; that is, for every bit set in s_q, the corresponding bit in the block signature s is also set (i.e., $s \wedge s_q = s_q$) and the block contains really the query word; (2) the block does not match the query (i.e., $s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the block in fact does not match the search criteria (false drop). In order to eliminate false drops, the block must be examined after the block signature signifies a successful match.

In this paper, we propose a method to speed up the (sequential) signature file scanning by establishing a tree structure, called *signature tree*, for it just like a position tree for a text [2]. But by the construction of a position tree, a *position identifier* is a continuous piece of character sequence, while by the construction of a signature tree a *signature identifier* is not a continuous piece of bit string.

A closely related work is the S-tree proposed in [8]. It is in fact an B-tree built over a signature file. Thus, it can be used to speed up the location of a signature in a signature file just like an B-tree for keys in a relational database. However, in the signature tree each path corresponds to a signature identifier which can be used to identify uniquely the corresponding signature in a signature file. It helps to find the set of signatures matching a query signature quickly.

Signature files can also be utilized as set access facility in OODBSs [16]. Especially, according to the analysis of [16], the bit-sliced signature file (BSSF) achieves a higher performance than the sequential signature file (SSF) by almost 50% (of time cost) in the best case. But the storage cost of BSSF doubles that of SSF and the update cost of BSSF triples that of SSF or more [16]. Later on, we will see that the

signature tree has a much better time complexity and less update costs than BSSF but with almost the same storage cost.

2. Signature trees

A first idea to improve the performance is to sort the signature file and then employ a binary searching. Unfortunately, this does not work due to the fact that a signature file is only an inexact filter. The following example helps for illustration.

Consider a sorted signature file containing only three signatures:

010 000 100 110
010 100 011 000
100 010 010 100

Assume that the query signature s_q is equal to 000 010 010 100. It matches 100 010 010 100. However, if we use a binary search, 100 010 010 100 can not be found.

For this reason, we try another method and construct a signature tree to avoid scanning a signature file completely.

2.1. Definition of signature trees

Consider a signature s_i of length F. We denote it as $s_i = s_i[1]s_i[2]\ldots s_i[F]$, where each $s_i[j] \in \{0,1\}(j = 1,\ldots,F)$. We also use $s_i(j_1,\ldots j_h)$ 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 $1 \leqslant j_k \leqslant F$ 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 $(1 \leqslant i \leqslant n)$. If there exists a sequence $j_1,\ldots,j_h$ such that for any $k \neq i$ $(1 \leqslant k \leqslant n)$ we have $s_i(j_1,\ldots,j_h) \neq s_k(j_1,\ldots,j_h)$, 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.

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 $|s_k| = F$ 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 $s_1, s_2, \ldots, s_n$ in S (signature file). For a leaf node u, $p(u)$ represents the pointer to the corresponding signature in S.
(3) Each internal node v is associated with a number, denoted $sk(v)$ which is the bit offset of a given bit position in the block signature pattern. That bit position will be checked when v is encountered.
(4) Let $j_1,\ldots,j_h$ be the numbers associated with the nodes on a path from the root to a leaf node labeled i (then, this leaf node is a pointer to the ith signature in S). 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. 2(b), we show a signature tree for the signature file shown in Fig. 2(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 is associated with a positive integer (which is used to tell how many bits to skip when searching). 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 have been checked at that moment. 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.

See the path consisting of the dashed edges in Fig. 2(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

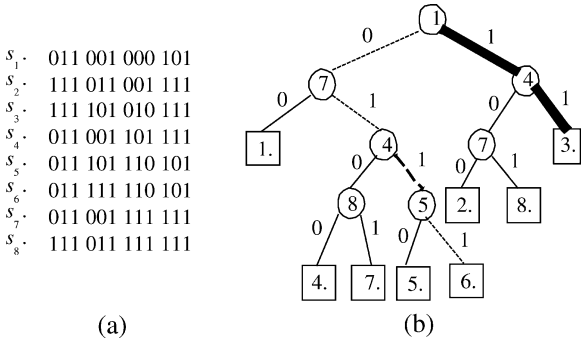

Fig. 2. Signature tree.

identifier of s_3 is $s_3(1,4) = (1,1)(4,1)$ (see the path consisting of the thick edges in Fig. 2(b)).

In the next section, we discuss how to construct such a signature tree for a signature file in great detail.

2.2. Construction of signature trees

Below we give an algorithm to construct a signature tree for a signature file, which needs only $O(N)$ time, where N represents the number of 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 the next signature to be inserted 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 by v. s can not be the same as v since in S there is no signature which is identical to any other. 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 occupied by u.v will become one of u's children.) If $b = 1$, we make v and the pointer to s be the left and right children of u, respectively. If $b = 0$, we make v and the pointer to s be, respectively, the right and left children 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
11          position;
          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 s and w 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. 3, we trace the above algorithm against the signature file shown in Fig. 2(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, i.e., $p(v_g) = s$. Denote $j_i = sk(v_i)$ $(i = 1, \ldots, g-1)$. Then, $s(j_1, j_2, \ldots, j_{g-1}) = (j_1, b(e_1)) \ldots (j_{g-1}, b(e_{g-1}))$ constitutes 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 $P = v_1 e_1 \ldots v_{g-1} e_{g-1} v_g$ 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 $j_i =$

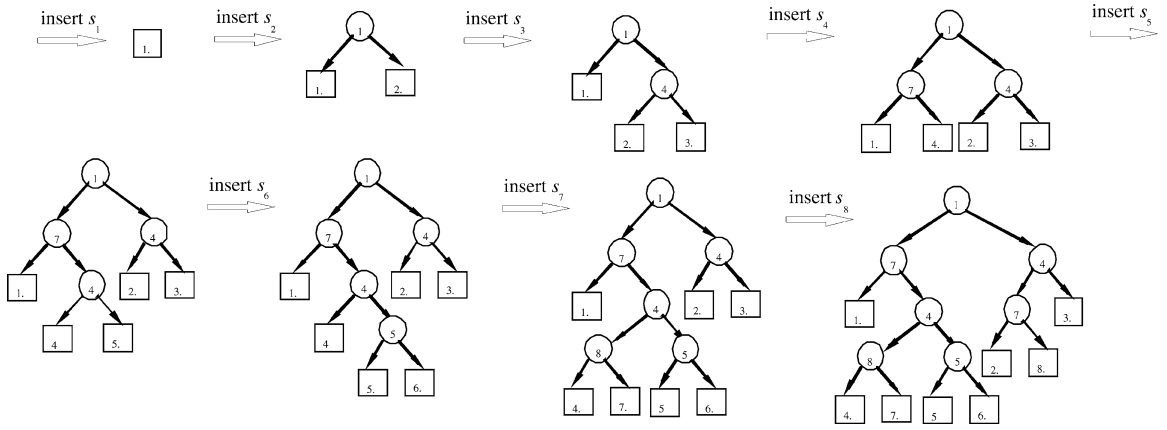

Fig. 3. Sample trace of signature tree generation.

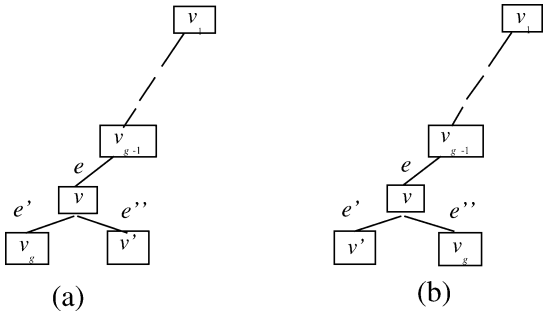

Fig. 4. Inserting a node v into T.

$sk(v_i)$ $(i = 1, \ldots, g-1)$. 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. 4(a) or 4(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 $p(v')$ differs.

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 $O(F)$ time. Thus, we have the following proposition.

Proposition 2. *The time complexity of the algorithm* sig-tree-generation *is bounded by* $O(N)$, *where* N *represents the number of signatures in a signature file.*

Proof. See the above analysis. □

Finally, we note that the above technique can also be used for a more general case that a signature file contains signature duplicates. In this case, a leaf node may be a set of pointers to different locations with the identical signature. We change the lines 8–13 of procedure *insert*( ) to construct such a leaf node as follows:

```
{compare s with the signature s0 pointed by
    a pointer in v;
 if s and s0 are identical then
    put the address of s in v
 else {assume that the first k bit of s agree
          with s0;
       but s differs from s0 in the (k + 1)th position;
       w ← v; replace v with a new node u
          with sk(u) = k + 1;
       if s[k + 1] = 1 then
          make s and w be, respectively, the right and
          left children of u
       else make s and w be the right and
          left children of u, respectively;}
}
```

3. Searching and maintenance of signature trees

In this section, we discuss the searching and maintenance of signature trees.

3.1. Searching a signature tree

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 defined 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 visited.

In fact, this definition just corresponds to the signature matching criterion.

To implement this inexact matching strategy, 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: set of signatures which survive the checking;
1. $S \leftarrow \emptyset$.
2. Push the root of the signature tree into $stack_p$.
3. If $stack_p$ is not empty, $v \leftarrow \text{pop}(stack_p)$; else return($S$).
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, $S \leftarrow S \cup \{p(v)\}$.
6. Go to (3).

The following example helps to illustrate the main idea of the algorithm.

Example 2. Consider the signature file and the signature tree shown in Fig. 2 once again.

Assume s_q = 000 100 100 000. Then, only part of the signature tree (marked with thick edges in Fig. 5) will be searched. On reaching a leaf node, the signature pointed by the leaf node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching. For this example, only 42 bits are checked (6 bits during the tree search and 36 bits during the signature checking). But by the scanning of the signature file, 96 bits will be checked. In general, if a signature file contains N signatures, the method discussed above requires only $O(N/2^l)$ comparisons in the worst case, where l represents the number of bits set in s_q, since each bit set in s_q will prohibit 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 section in more detail.

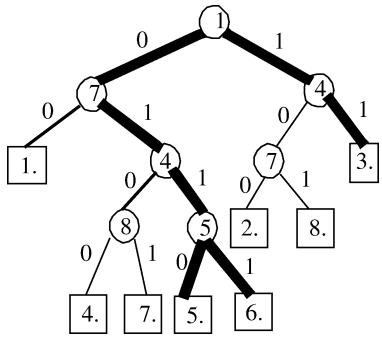

Fig. 5. Signature tree search.

3.2. Maintenance of a signature tree

When a signature s is added to a signature file, the corresponding signature tree can be changed easily by running the algorithm *insert*( ) once with s as the input (see Section 2.2).

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

(i) Let z, u, v, and w be the nodes as shown in Fig. 6(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. 6(b).

From the above analysis, we see that the maintenance of a signature tree is an easy task.

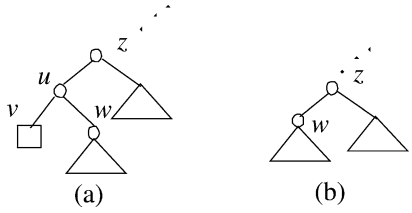

Fig. 6. Illustration for deleting a signature.

4. Computational complexity

4.1. Time complexity

To analyze the performance of the signature tree, we consider four parameters: N — the number of signatures in a signature file, F — the signature length, m — the number of bits set to 1 in a signature, and D — the size of a block. When the average signature is half-populated with 1s, the false drop probability and storage overhead trade-off combination is optimized [4]. In such a setting, the two parameters N and F satisfy the following inequality.

$$F \leqslant \binom{F}{F/2}. \tag{1}$$

We have the above inequality based on a simple observation that if $N > \binom{F}{F/2}$ there must exist two signatures having the same binary strings. In this case, one of them will be removed from the signature file.

In terms of *Stirling* formula, $F! \sim \sqrt{2\pi F}\left(\frac{F}{\mathrm{e}}\right)^F$, we have

$$\binom{F}{F/2} \sim \sqrt{\frac{2}{\pi F}} \cdot 2^F. \tag{2}$$

Then, we have

$$N \leqslant \sqrt{\frac{2}{\pi F}} \cdot 2^F. \tag{3}$$

From this, we have $\log_2 N \leqslant \frac{1}{2} - \frac{1}{2}\log_2 \pi - \frac{1}{2}\log_2 F + F$.

Thus, F satisfies the following inequality:

$$\log_2 N - \frac{1}{2} + \frac{1}{2}\log_2 \pi + \frac{1}{2}\log_2 F \leqslant F. \tag{4}$$

According to [4,9], in the case that the average block signature involves an equal number of 1s and 0s, the three design parameters m, F, and D satisfy the relationship below:

$$F \times \ln 2 = m \times D. \tag{5}$$

In addition, on average l (the number of bits set to 1 in a query signature) is equal to m.

From the above, we derive the time complexity of the signature tree searching as follows:

$$N/2^l \sim N/2^m = N/2^{(F\ln 2)/D}. \tag{6}$$

In terms of (2) and (6), we have

$$\begin{aligned} N/2^{(F\ln 2)/D} &\leqslant N/2^{((\log N - \frac{1}{2} + \frac{1}{2}\log\pi + \frac{1}{2}\log F)\ln 2)/D} \\ &= N \Big/ \left(N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \sqrt{F}\right)^{(\ln 2)/D}. \end{aligned} \tag{7}$$

Finally, we have the inequality

$$\begin{aligned} N/2^{(F\ln 2)/D} &\leqslant N \Big/ \left(N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \sqrt{F}\right)^{(\ln 2)/D} \\ &\leqslant N \Big/ \left(N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \left(\log N - \frac{1}{2} + \log\sqrt{\pi} + \log\sqrt{F}\right)^{1/2}\right)^{(\ln 2)/D} \\ &\sim \left(\sqrt{\frac{2}{\pi}}\right)^{(\ln 2)/D} \cdot N/\left(N\sqrt{\log N}\right)^{(\ln 2)/D}. \end{aligned} \tag{8}$$

Fig. 7 shows the calculation related to the above formula.

In Fig. 7, the signatures checked are computed in terms of N — the size of a signature file. From this, we can see that the performance of the signature tree searching degrades as the size of a block increases. It is because given a fixed signature length a larger block requires that fewer bits in a term signature are set to 1, which weakens the filtering power of signature trees when it comes to single term query processing.

The above result also shows that the signature tree outperforms the bit-sliced signature file (BSSF). In terms of the analysis of [16], BSSF improves the signature file scanning by almost 50%. But the signature tree can be 10 times better than the scanning of a signature file.

When compared with the S-tree [8], we note that given the size N of a signature file and the length F of the signatures in it the S-tree's time complexity decreases linearly as the query weight increases (see the experimental results of [8]). But the time complexity of the signature tree reduces exponentially with the query weight increments according to $O(N/2^l)$, the

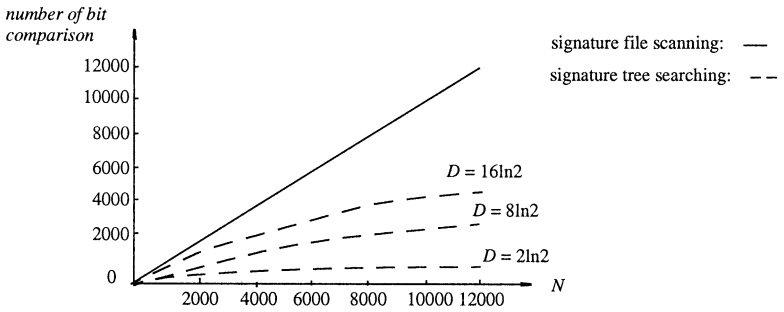

Fig. 7. Time complexities of signature files scanning and signature tree searching.

number of comparisons to be conducted during a signature tree traversal, where l represents the number of bits set to 1 in the query signature.

We also notice that the uncompressed inverted index structure is very space-consuming. It requires 50 to 300% of the space required for the data [14]. Therefore, in the case that the integer encoding can not be used for inversion, the signature tree is a promising choice.

4.2. Extra space overhead of a signature tree

Note that the signature tree is a binary tree. Thus, a signature tree can be stored as a set of triples of the form: $\langle v, lp, rp\rangle$, where v represents the number associated with a node, lp represents the pointer to the left subtree and rp represents the pointer to the right subtree.

Assume that the length of a signature is F and the number of signatures in a file is N. (The size of the signature file is therefore $N \times F$ bits.) Then, for each v we need $\log_2 F$ bits and for each lp (rp) we need $\log_2 N$ bits. Accordingly, for all the internal nodes of a signature tree, we need $N \times \log_2 F + 2N \times \log_2 N$ bits space. To mitigate this problem to some extent, we use the following relative address encoding:

(1) The triples for a signature tree are stored in the breadth-first order.
(2) lp and rp are relative addresses, i.e., the absolute address of node v' (denoted $add(v')$) pointed by lp (or rp) is equal to $add(v') = add(v) + lp$ (or $add(v) + rp$).

In this way, we need only 2 bits for the addresses of the nodes at the first level, 2^2 bits for the second level, 2^3 bits for the third level, and so on. The space overhead can then be reduced to

$$N \times \log_2 F + 2\sum_{i=0}^{k} 2^i \cdot (i+1), \tag{9}$$

where $2^k = N$. It is almost half of the size of the corresponding signature file.

5. Conclusion

In this paper, a new concept of signature identifiers has been introduced, which can be used to differentiate signatures in a signature file from each other. Based on this concept, a tree structure, called a signature tree, is proposed in which each path from the root to a leaf node corresponds to a signature identifier. Then, the scanning of a signature file can be replaced by the traversal of a signature tree, which improves the query processing efficiency significantly.

References

[1] H. Andre-Joesson, D. Badal, Using signature files for querying time-series data, in: Proc. 1st European Symp. on Principles of Data Mining and Knowledge Discovery, 1997.
[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, London, 1974.
[3] A. Cardenas, Analysis and performance of inverted data base structures, Comm. ACM 18 (5) (1975) 253–263.
[4] S. Christodoulakis, C. Faloutsos, Design consideration for a message file server, IEEE Trans. Software Engrg. 10 (2) (1984) 201–210.

[5] W.W. Chang, H.J. Schek, A signature access method for the STARBURST database system, in: Proc. 19th VLDB Conf., 1989, pp. 145–153.
[6] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, 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) 345–386.
[7] P. Ciaccia, P. Zezula, Declustering of key-based partitioned signature files, ACM Trans. Database Systems 21 (3) (1996) 295–338.
[8] U. Deppisch, S-tree: A dynamic balanced signature index for office retrieval, in: ACM SIGIR Conf., September 1986, pp. 77–87.
[9] D. Dervos, Y. Manolopulos, P. Linardis, Comparison of signature file models with superimposed coding, Inform. Process. Lett. 65 (1998) 101–106.
[10] C. Faloutsos, Access methods for text, ACM Computing Surveys 17 (1) (1985) 49–74.
[11] C. Faloutsos, Signature files, in: W.B. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures & Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 44–65.
[12] C. Faloutsos, R. Lee, C. Plaisant, B. Shneiderman, Incorporating string search in hypertext system: User interface and signature file design issues, HyperMedia 2 (3) (1990) 183–200.
[13] S.W. Golomb, Run-length encoding, IEEE Trans. Inform. Theory 12 (3) (1966) 399–401.
[14] R. Haskin, Special purpose processors for text retrieval, Database Engrg. 4 (1) (1981) 16–29.
[15] D. Harman, E. Fox, R. Baeza-Yates, Inverted files, in: W.B. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures & Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 28–43.
[16] Y. Ishikawa, H. Kitagawa, N. Ohbo, Evaluation of signature files as set access facilities in OODBs, in: Proc. of ACM SIGMOD Internat. Conf. on Management of Data, Washington, DC, May, 1993, pp. 247–256.
[17] D.L. Lee, Massive parallelism on the hybrid text-retrieval machine, Inform. Process. Management 31 (6) (1992) 281–289.
[18] W. Lee, D.L. Lee, Signature file methods for indexing object-oriented database systems, in: Proc. ICIC'92 — 2nd Internat. Conf. on Data and Knowledge Engineering: Theory and Application, Hongkong, December 1992, pp. 616–622.
[19] A. Moffat, J. Zobel, Self-indexing inverted files for fast text retrieval, ACM Trans. Inform. Syst. 14 (4) (1996) 349–379.
[20] C. Stanfill, B. Kahle, Parallel free-text search on connection machine system, Comm. ACM 29 (12) (1986) 1229–1239.
[21] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, J. Zobel, Atlas: A nested relational database system for text application, IEEE Trans. Knowledge Data Engrg. 7 (3) (1995) 454–470.
[22] H.S. Yong, S. Lee, H.J. Kim, Applying signatures for forward traversal query processing in object-oriented databases, in: Proc. of 10th Internat. Conf. on Data Engineering, Houston, TX, February, 1994, pp. 518–525.
[23] J. Zobel, A. Moffat, K. Ramamohanarao, Inverted files versus signature files for text indexing, ACM Trans. Database Syst. 23 (4) (December 1998) 453–490.