Signature Files and Signature File Construction

Yanjun Chen
University of Winnipeg, Canada

Yong Shi
University of Winnipeg, Canada

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 four techniques have been frequently used: full text searching, B-trees, inversion, and the signature file. Full text searching imposes no space overhead but requires long response time. In contrast, B-trees, inversion, and the signature file work quickl, 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 different construction approaches of a signature file.

The signature technique cannot only be used in document databases but also in relational and object-oriented databases. In a document database, a set of semistructured (XML) documents is stored and the queries related to keywords are frequently evaluated. To speed up the evaluation of such queries, we can construct signatures for words and superimpose them to establish signatures for document blocks, which can be used to cut off nonrelevant documents as early as possible when evaluating a query. Especially, such a method can be extended to handle the so-called containment queries, for which not only the key words but also the hierarchical structure of a document has to be considered. We can also handle queries issued to a relational or an object-oriented database using the signature technique by establishing signatures for attribute values, tuples, as well as tables and classes.

BACKGROUND

The signature file method was originally introduced as a text indexing methodology (Faloutsos, 1985; Faloutsos, Lee, Plaisant & Shneiderman, 1990). Nowadays, however, it is utilized in a wide range of applications, such as in office filing (Christodoulakis, Theodoridou, Ho, Papa, & Pathria, 1986), hypertext systems (Faloutsos et al.), relational and object-oriented databases (Chang & Schek, 1989; Ishikawa, Kitagawa, & Ohbo, 1993; Lee & Lee, 1992; Sacks-Davis, Kent, Ramamohanarao, Thom, & Zobel, 1995; Yong, Lee, & Kim, 1994), as well as data mining (Andre-Joesson & Badal, 1997). It requires much smaller storage space than inverted files and can handle insertion and update operations in databases easily.

A typical query processing with the signature file is as follows: When a query is given, a query signature is formed from the query value. 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.

A variety of approaches for constructing signature files have been proposed, such as bit-slice files, S-trees, and signature trees. In the following, we overview all of them and discuss a new application of signatures for tree inclusion problem, which is important for containment query evaluation in document databases.

SIGNATURE FILES AND SIGNATURE FILE ORGANIZATION

Signature Files

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. 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 signa-

ture file needs to be searched (Faloutsos, 1985; Faloutsos, 1992). 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 (Ciaccia & Zezula, 1996).

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 consist of fixed-size logical blocks and each block involves a constant number (D) of noncommon, 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.

Figure 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 Figure 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 doesn't 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 doesn't 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 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 evaluating a query, a full scan of the signature file has to be performed. Therefore, it is generally slow in retrieval.

Figure 1(b) shows a simple signature file. To determine the length of signatures, we use the following formula (Faloutsos, 1985):

$$F \times \ln 2 = m \times D \quad (1)$$

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 (Ishikawa et al., 1993), i.e., in F files, in each of which one bit per signature for all the signatures is stored as shown in Figure 2.

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, consider the query signature s_q = 10110000. First, we check the first bit-slice file 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 will be checked. Since the second bit in s_q is 0, no positions will be filtered. Next, we check the third bit-slice file against the third bit in s_q. Because all the three positions are set to 1 in it, the same positions in the 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, the search stops and reports a *nil*.

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

S-Trees

Figure 1. Signature generation, comparison and signature file

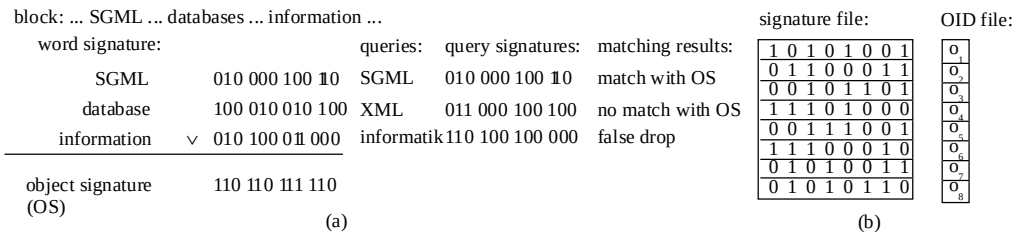
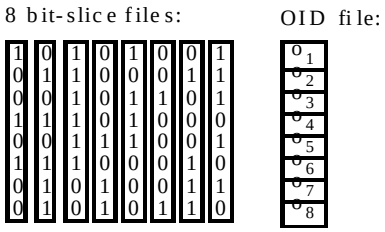


Figure 2. Illustration for bit-slice file



Similar to a B$^+$-tree, an S-tree is a height-balanced multiway tree (Deppisch, 1986). 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 *N* be the parent node of *N*'. Then, there exists a signature in *N* whose value is obtained by superimposing all the signatures in *N*'. See Figure 3 for illustration.

To retrieve a query signature, we search the S-tree top-down. However, more than one path may be visited. The first signature in the root N_1 leads us to its child node N_2 because the third and fourth bits are set to 1. In N_2, the s_q. Then, we go to the leaf node N_4 and N_5. In N_4, we find two matching candidates $\{o_4, o_5\}$, and in N_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 *N* becomes full, it will be split into two nodes, and at the same time a parent node N_{parent} will be generated if it does not exist. In addition, two new signatures will put in N_{parent}. Assume that the capacity of *N* is *K* (*i.e.*, *N* can accommodate *K* signatures.) Then, when we try to insert the (*K* + 1)th signature into *N*, it has to be split into two nodes N_α and N_β. To do this, we will pick a signature in *N* which has the heaviest signature weight (*i.e.*, with the most 1s) in *N*. It is called the α-seed and will be put in N_α. Then, we select a second signature which has the maximum number of 1s in those positions whereα has 0. That is, the signature provides the maximal weight increase to α. This signature is called the β-seed and put in N_β. Any of the rest *K* - 1 signatures are assigned to N_α or N_β, depending on whether it is closer to N_α or N_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 N_α and N_β, respectively. See Figure 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 an 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, Bozanis, and Manolopoulos (2002). 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 the cost of time, i.e., by checking more signatures, which makes the insertion of a signature into a S-tree extremely inefficient.

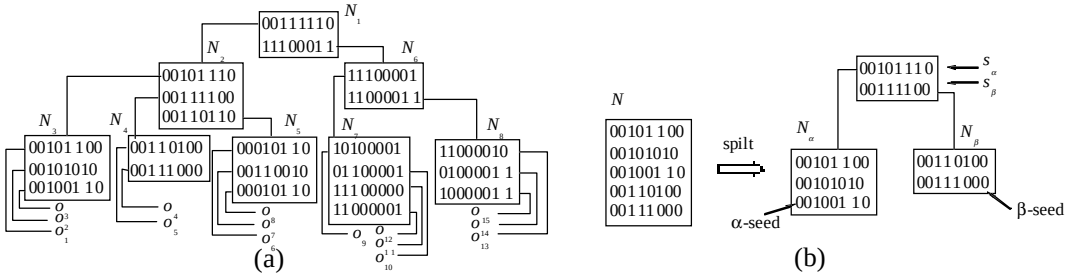
In the following, we discuss a quite different method, called signature trees, by means of which all the drawbacks of S-trees are removed.

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 $1 \le j_k \le F$ for $k \in \{1, \ldots, h\}$.

Definition 1 (*signature identifier*). Let $S = s_1.s_2 \ldots .s_n$ denote a signature file. Consider $s_i (1 \le i \le n)$. 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.

Figure 3. Illustration for S-tree and node splitting



Definition 2 (*signature tree*). A signature tree for a signature file $S = s_1.s_2 \ldots .s_n$, where $s_i^1 \neq s_j$ 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, ..., n, used as pointers to n different positions of s_1, s_2 ... and 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 Figure 4(b), we show a signature tree for the signature file shown in Figure 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 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 fifth bit of s: $s[5]$ will be checked.
  See the path consisting of the dashed edges in Figure 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 the thick edges in Figure 4(b).
  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 defined 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 visited.

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

- **Example 2.** Consider the signature file and the signature tree shown in Figure 4(b) once again. Assume s_q = 000 100 100 000. We search the signature tree top-down. First, we check the root r. Since $sk(r) = 1$, we check the first bit in s_q. It is 0. Then, both the child nodes of r will be explored. (See the thick edges in Figure 5.) When we visit the left child node v of r, the seventh bit in s_q will be checked since $sk(v) = 7$. It is equal to 1. Then, only the right child node of v will be checked. We repeat this process until all the possible leaf nodes are visited. 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

Figure 4. Signature tree

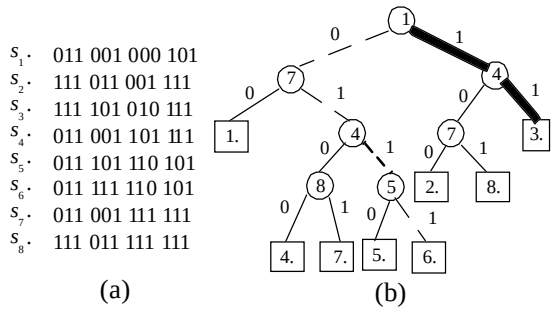
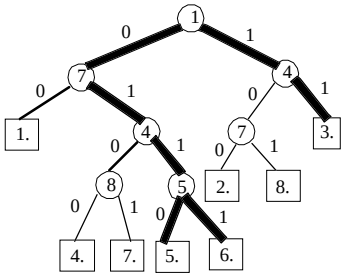


Figure 5. Signature tree search



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.

Due to limitation of space, we don't discuss here the maintenance of signature trees. An interested reader is referred to Chen (2004) for detailed description.

Integrating Signatures Into Tree Inclusion

In this section, we discuss how to integrate signatures into the tree inclusion problem, which is important to containment queries in document databases. As pointed out in Kilpelainen and Mannila (1995), the evaluation of a containment query is in essence to check whether a query tree is concluded in a document tree.

Let T be an ordered, rooted tree with root v and children $v_1, v_2, ..., v_i$. The *postorder* traversal of $T(v)$ (the tree rooted at v) is obtained by visiting $T(v_k)$, $1 \le k \le i$ in order, recursively, and then visiting the v. The *postorder number*, post(v), of a node $v \in V(T)$ is the number of nodes preceding v in the postorder traversal of T. We define an ordering of the nodes of T given by $v \pi v'$ iff post(v) < post(v'). Also, $v \tilde{} v'$ iff v p v' or $v = v'$. Furthermore, we extend this ordering with two special nodes $\wedge \pi v \pi$ T. The *left relatives*, lr(v), of a node $v \in V(T)$ are the set of nodes that are to the left of v and similarly the *right relatives*, rr(v), are the set of nodes that are to the right of v.

Definition 3. Let S and T be rooted labeled trees. We defined an ordered embedding (f, S, T) as an injective function f: $V(S)$ $V(T)$ such that for all nodes v, $u \in V(S)$,

1. label(v) = label($f(v)$); (label preservation condition)
2. v is an ancestor of u iff $f(v)$ is an ancestor of $f(u)$; (ancestor condition)
3. v is to the left of u iff $f(v)$ is to the left of $f(u)$; (sibling condition) □

Figure 6 shows an example of an ordered inclusion.

In Figure 6(a), we show that the tree on the left can be included in the tree on the right by deleting the nodes labeled: d, e, and b. Figure 6(b) shows a possible embedding. An embedding is *root preserving* if f(root(S)) = root(T). Figure 6(b) shows also an example of the root preserving embedding.

A lot of algorithms have been developed to check tree inclusion, such as those reported in Alonso and Schott (1993), Kilpelainen and Mannila (1995), Richter (1997), and Chen (1998). All the methods focus, however, on the bottom-up strategies to get optimal computational complexities, which are not suitable for the database environment since the algorithms proposed assume that both the target tree (or, say, the document tree) and the pattern tree (or, say, the query tree) can be accommodated completely in main memory. Recently, a top-down algorithm was proposed (Chen & Chen, 2004) which has the same time complexity as the best bottom-up algorithm but needs no extra space. More importantly, it works well in a database environment for the reason that it checks a target tree in a top-down fashion, and each time only part of the tree is manipulated. Furthermore, it can be combined with *signatures* to speed up query evaluation.

Figure 6. Illustration of tree inclusion

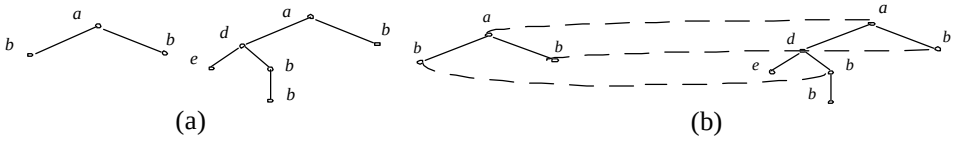
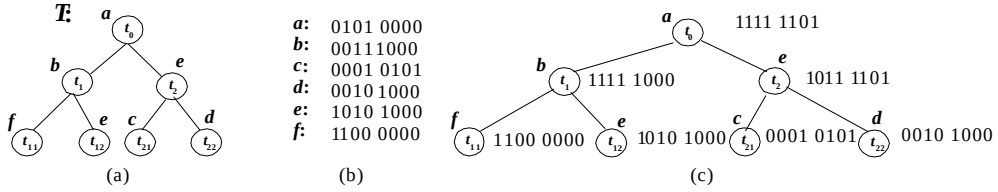


Figure 7. Node signatures



The top-down algorithm can be outlined as follows. A detailed description can be found in Chen and Chen (2004).

Algorithm *top-down-tree-inclusion*

1. Let r_1 and r_2 be the roots of T and S, respectively.
2. Let T_1, ..., T_n be the subtrees of r_1, and S_1, ..., S_m be the subtrees of r_2.
3. If r_1 doesn't match r_2, try to find an i ($1 \le i \le n$) such that T_i includes the whole S.
4. If r_1 matches r_2, try to find i_1, ..., i_k such that T_{i_1} contains S_1, ..., S_{j_1}, T_{i_2} contains S_{j_1+1}, ..., S_{j_2}, and T_{i_k} contains $S_{j_{k-1}+1}$, ..., S_{j_k}, where $S_{j_k} = S_m$.

To speed up this process, we assign each label a signature, and construct the signatures for non-leaf nodes as follows:

Definition 4. Let v be a node in a tree T. If v is a leaf node, its signature $s_{v\ 1}$, …, s_n be its children, then $s_v = s \vee S_{v_1} \vee ... \vee S_{v_n}$, where s represents the signature for the label associated with v, and S_{v_1}, ... , and S_{v_n} are the signatures of v_1, ..., v_n, respectively.

Example 3. Consider the tree shown in Figure 7(a). If the signatures assigned to the labels are those shown in Figure 7(b), rach node in the tree will have a signature as shown in Figure 7(c).

Then, each time we check a node u in S against a node v in T, we will first check their signatures. If they don't match, the subtree rooted at v will be cut off and not be searched any more, reducing the time overhead greatly.

Here, an important problem is how to determine the length of signatures. Due to the superimposing of signatures along the tree paths, Equation (1) (shown in the section Signature Files) is not useful any more since it was established only for the simple structure of sequential signature files. However, if the length of signatures is not properly determined, as in an S-tree, the signatures near the root will be very heavy and the selectivity will be reduced dramatically. For this reason, we make the following analysis and develop a new way to estimate the signature length in such a way that the above problem can be removed.

Consider two signatures s_1 and s_2. Assume that both of them are of length F and with m_1 and m_2 bits set to 1, respectively. Now, let $s = s_1 \vee s_2$. Obviously, s will possibly contain more 1s. To keep the ratio of 1s in s not increased, s should be set longer. The question is: How long should s be? Let l be the number of 1s in s and denote $\delta = l - m'$, where $m' = max(m_1, m_2)$. Then, $F + c\delta$ should be a reasonable length for s, where c is a constant and should be tuned for different applications. The value of δ can be estimated as follows.

Let λ be a random variable representing the number of positions, in which both s_1 and s_2 have 1s. Then, the mathematical expectation of λ can be calculated as below:

$$E\mathrm{l} = 1\times p(\mathrm{l} = 1) + 2\times p(\mathrm{l} = 2) + ... + m''\times p(\mathrm{l} = m'') \quad (2)$$

where $m'' = min(m_1, m_2)$ and $p(\lambda = i)$ represents the probability of l equal to i. To calculate this probability, we use the following formula:

$$p(\lambda = i) = \frac{\binom{F - m_2}{m_1 - i}\binom{m_2}{i}}{\binom{F}{m_1}} \quad (3)$$

Note that $l = m_1 + m_2 - \lambda$. Then, we have $\delta = l - m = m_1 + m_2 - \lambda - max(m_1, m_2)$.

Using the above formulas, we can determine the length of signatures for a tree as follows. First, we calculate the average number of key words in all the leaf nodes, which is used as the value of D to determine the initial values of F and m using Equation (1). Then, we compute the lengths of signatures for the internal nodes in a bottom-up way. That is, we first calculate the lengths for all those nodes, each of which is a parent of some leaf nodes. Then, we compute the lengths for the nodes at a higher lever. This process is repeated until the length of the signature for the root is computed, which will be used as the length of all the signatures to be generated.

FUTURE TREND

As discussed above, the signature file is a useful technique for text indexing and query evaluation in databases. It can also be utilized for some problems in the computational graph theory, i.e., for the tree inclusion problem. As future research, we will concentrate on an interesting issue: how to reduce the size of a signature file. This may be done by elaborating Equation (1),

shown in the section Signature Files, which is only an empirical formula. What we want is to find a mathematical method to determine, for a given set of key words which are distributed in a collection of blocks, the minimum length of the signatures and the best choice of the number of bits that are set to 1.

CONCLUSION

In this article, four methods for constructing signature files are described. They are the sequential signature file, the bit-slice signature file, the S-tree, and the signature tree. Among these methods, the signature file has the simplest structure and is easy to maintain, but it is slow for information retrieval. In contrast, the bit-sliced file and the S-tree are efficient for searching but need more time for maintenance. In addition, an S-tree needs much more space than a sequential signature file or a bit-slice file. The last method, i.e., the signature tree structure, improves the S-tree by using less space for storage and less time for searching. Finally, as an important application, the signatures can be integrated into the top-down tree inclusion strategy to speed up the evaluation of containment queries. This can also be considered as a quite different way to organize a signature file.

REFERENCES

Alonso, L., & Schott, R. (1993). On the tree inclusion problem. *Proceedings of Mathematical Foundations of Computer Science*, (pp. 211-221).

Andre-Joesson, H., & Badal, D. (1997). Using signature files for querying time-series data. *Proceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery*.

Chang, W. W., & Schek, H. J. (1989). A signature access method for the STARBURST database system. *Proceedings of the 19th VLDB Conference*, (pp. 145-153).

Chen, W. (1998). More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, *26*, 370-385.

Chen, Y. (2004). Building signature trees into OODBs. *Journal of Information Science and Engineering, 20*, 275-304.

Chen, Y., & Chen, Y. B. (2004). An efficient top-down algorithm for tree inclusion. *Proceedings of the 18th International Conference Symposium on High Performance Computing System and Application*,.

Christodoulakis, S., Theodoridou, M., Ho, F., Papa, M., & Pathria, A. (1986). Multimedia document presentation, information extraction and document formation in MINOS—A model and a system. *ACM Transactions On Office Information Systems, 4*(4), 345-386.

Ciaccia, P., & Zezula, P. (1996), Declustering of key-based partitioned signature files. *ACM Transactions on Database Systems*, *21*(3), 295-338.

Deppisch, U. (1986). S-tree: A dynamic balanced signature index for office retrieval. *ACM SIGIR Conference*, (pp. 77-87).

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, *17*(1), 49-74.

Faloutsos, C. (1992). Signature files. In W. B. Frakes & R. Baeza-Yates (Eds.), *Information retrieval: Data structures & algorithms* (pp. 44-65). NJ: Prentice Hall.

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

Ishikawa, Y., Kitagawa, H., & Ohbo, N. (1993). Evaluation of signature files as set access facilities in OODBs. *Proceedings of ACM SIGMOD International Conference on Management of Data*, (pp. 247- 256).

Kilpelainen, P., & Mannila, H. (1995). Ordered and unordered tree inclusion. *SIAM Journal of Computing, 24*, 340-356.

Lee, W., & Lee, D. L. (1992). Signature file methods for indexing object-oriented database systems. *Proceedings of ICIC'92—2nd International Conference on Data and Knowledge Engineering: Theory and Application*, (pp. 616-622).

Richter, T. (1997). A new algorithm for the ordered tree inclusion problem. In *Lecture Notes of Computer Science: Vol. 1264. Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, CPM* (pp. 150-166).

Sacks-Davis, R., Kent, A., Ramamohanarao, K., Thom, J., & Zobel, J. (1995). Atlas: A nested relational database system for text application. *IEEE Transactions on Knowledge and Data Engineering*, *7*(3), 454-470.

Tousidou, E., Bozanis, P., & Manolopoulos, Y. (2002). Signature-based structures for objects with set-values attributes. *Information Systems*, *27*(2), 93-121.

Yong, H. S., Lee, S., & Kim, H. J. (1994). Applying signatures for forward traversal query processing in

object-oriented databases. *Proceedings of 10th International Conference on Data Engineering*, (pp. 518-525).

KEY TERMS

Bit-Slice Signature File: A bit-slice file is a file in which one bit per signature for all the signatures is stored. For a set of signatures of length *F*, *F* bit-slice files will be generated.

Sequential Signature File: A signature file is a set of signatures stored in a file in a sequential way.

Signature Identifier: A signature identifier for a signature in a signature file is a positioned bit string which can be used to identify it from others.

Signature Tree: A signature tree is an index structure in which each path represents a signature identifier for the signature pointed to by the corresponding leaf node.

Signatures: A bit string generated for a key word by using a hash function.

S-tree: An S-tree is a height-balanced multiway tree. 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.

Tree Inclusion: Let *T* and *S* be ordered, labeled trees. *S* is said to be included in *T* if there is a sequence of delete operations performed on *T* which make *T* isomorphic to *S*.