JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 20, 275-304 (2004)

Building Signature Trees into OODBs

YANGJUN CHEN
Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, R3B 2E9, Canada

Although object-oriented database systems offer more powerful modeling capability than relational database systems, their performance suffers from the increased complexity in the data model. Recently, a lot of research has focused on mitigating this problem by building indexes over single classes, class hierarchies, or nested object hierarchies. In this paper, we propose a new indexing method. It is based on the technique that employs signature files, but differs from the existing methods in two aspects: (1) all the signature files are organized into a hierarchy to filter irrelevant data as early as possible; (2) a signature file itself is stored as a tree structure (called a *signature tree*) to speed up signature scanning. Together with the concept of query signature hierarchies, this technique reduces the search space dramatically and, therefore, improves significantly the time complexity of query evaluation.

Keywords: OODBs, indexes, signature files, signature trees, query evaluation

1. INTRODUCTION

In the past two decades, object-oriented database systems (OODBS) have attracted a significant amount of attention in academic and industrial communities [11, 27]. Several experimental and commercial systems, such as GemStone [36], Orion [27], and O2 [4], have been developed. Its powerful modeling capability is a major advantage of OODBS over relational databases. However, much work still needs to be done on query processing, optimization, and indexing techniques in order to improve performance.

In most OODBSs, secondary indexes on objects are supported to improve the retrieval of instances from a single class as well as from all the classes in a class hierarchy. A lot of indexing techniques have been proposed. For example, GemStone builds indexes along a path of object links [36]; class-hierarchy index and single-class index were investigated in the Orion project [28]. In addition, several B-tree like indexing methods have been published in the literature, such as CH-trees [29], H-trees [33], and hcC-trees [48], which are indexes over class hierarchies. Instead of focusing on the inheritance hierarchy of classes, other researchers have explored the aggregation hierarchy of classes (nested object hierarchies) and proposed various indexing structures on nested attributes [5, 7, 21, 30].

Received February 22, 2002; revised November 25, 2002; accepted February 21, 2003.
Communicated by Arbee L. P. Chen.

Quite different from the methods that employ tree-based indexes, signature file techniques have been extensively investigated with respect to relational databases and text retrieval, and recently extended to object-oriented databases [26, 32, 41], where each object is assigned a signature, and each class is assigned a signature file, which contains all the signatures of the objects belonging to it.

In this paper, we organize the signature files into a hierarchy to filter irrelevant data as early as possible. In addition, we introduce the concept of *signature trees*, by means of which signatures are not simply organized as a flat file, but stored as a tree structure. Then, the scanning of a signature file is changed to the searching of a binary tree, which reduces the time complexity by one order of magnitude or more.

The rest of this paper is organized as follows. In section 2, we survey related work. In section 3, we describe the basic features of queries used in an object-oriented database. Section 4 is devoted to introducing the concepts of signature files and signature file hierarchies. In section 5, we introduce signature trees and discuss how to use them to expedite query evaluation as well as how to maintain them. In section 6, we evaluate the performance achieved. Section 7 focuses on the maintenance of signature trees. Finally, section 8 is a short conclusion.

2. RELATED WORK

Index techniques have been extensively investigated in both the information retrieval and database research areas and many methods have been developed within the past three decades.

To index large files in information retrieval systems, different tree indexing approaches have been proposed, such as binary trees [31], suffix trees [15], position trees [3], and their variants built over flat files. By means of these mechanisms, the system performance can be improved significantly. The signature file scheme [13, 17, 20] is a method quite different from the tree indexing techniques, where each key word is assigned a signature, and a set of key words is assigned a "super" signature constructed by superimposing the signatures in that set. It works as an inexact filter. Another interesting approach is the inverted index, which is a set of postings lists [24], 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 [24]. Recently, Much work has been done on the encoding of postings lists in the context of document databases [37, 42]. When Golomb's encoding for the integers [22] is employed, the index size can be reduced to 14% of the indexed data with little or no loss of retrieval effectiveness [43]. Unfortunately, Golomb's encoding approach cannot be applied to an objected-oriented database since any postings list for it will be 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 [10, 25].

Some of the techniques mentioned above have been modified or extended to support query evaluation in databases. A notable example is B-tree [9] and its variants, such as B^+-tree, B*-tree [18] and fat-btree [41], which were developed based on the balancing

mechanism of binary trees with some special features added to ease tree balancing or to minimize accesses to data files. Such techniques have been further developed to speed up query evaluation in object-oriented databases [29, 33, 38]. In [29], an indexing structure, called CH-trees, was proposed, which is based on B^+-trees and essentially maintains a single index for all classes of a class (inheritance) hierarchy. It clusters the object oids of all classes in the class hierarchy for a given value of the indexed attribute. In [33], another indexing structure, called H-trees, was discussed. Its central idea is that one B^+-tree per class in a class hierarchy is maintained, but the indexes are nested according to their subclass-superclass relationship. Essentially, a H-tree clusters together the object oids of a single class with a given value of the indexed attribute. The above two tree-based indexing approaches were combined to create a new method, called hcC-trees, in [38], and it achieves has better performance. Recently, based on these techniques, several new methods have been proposed, such as χ-tree, discussed in [14], and the multikey type indexing method addressed in [35]. In addition to the indices over inheritance hierarchies, a lot of indexing schemes for nested attribute queries have been proposed in the literature [5, 7, 12, 21, 30]. Three index organizations for use in the evaluation of a query in an OODBS were introduced in [8]. As an extension of [8], the performance of path indexes for queries containing several predicates was evaluated in [7]. In [30], query processing in an OODBS was improved by maintaining separate structures to redundantly store objects which are frequently traversed by database queries. A hybrid indexing approach, called a generalized index, was proposed in [21] to support aggregation hierarchies of classes with complex and primitive objects. In [12], an optimal index configuration for a path was achieved by splitting the path into subpaths and optimally indexing each of them. Contrary to the single path indexing mentioned above, index interaction was considered in [12]. In the case of multi-indexes, the index with optimal behaviors will be chosen in terms of the objective functions developed in [12].

The signature file technique can be conveniently used to index nested object hierarchies. In [32], how to construct signature files for classes was discussed. In [42], an optimal method was proposed, by means of which signatures are distributed over a nested object hierarchy and the signature of a referred object is stored in the referring one. Then, such a signature can be used to check the predicate involving the corresponding object before it is visited. In this way, some evaluation time can be saved. In addition, signature files can also be utilized as a set access facility in OODBSs [26]. Moreover, according to the analysis of [26], the bit-sliced signature file (BSSF) achieves better performance than the sequential signature file approach (SSF) by almost 50% (of the time cost) in the best case. But the storage cost of BSSF is twice that of SSF, and the update cost of BSSF is triple that of SSF or more [26]. The basic advantage of sequential signature files lies in their efficiency in handling new insertions and queries on parts of words. When compared to tree-based indexing, however, sequential signature files suffer from two drawbacks: (1) they can not be used to evaluate range queries; and (2) for each query processed, the entire signature file needs to be scanned, which involves high processing and I/O costs.

In this paper, we try to mitigate the second problem to some extent. First, we organize the sequential signature files into a hierarchical structure which can be used to reduce the search space during a query evaluation. Second, we store a single signature

file as a tree, a so-called *signature tree*, to expedite the scanning of a single signature file. When a signature file itself is large, the amount of time saved using this approach is significant. A closely related work is the S-tree proposed in [16]. It is, in fact, a B-tree built over a signature file. Thus, it can be used to speed up the process of locating a signature in a signature file, just like a B-tree for primary keys in a relational database. However, in a 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.

In the following discussion, we use the term "signature file" to refer to a sequential signature file.

3. QUERIES IN OBJECT-ORIENTED DATABASES

In object-oriented database systems, an entity is represented as an object, which consists of methods and attributes. Methods are procedures and functions associated with an object defining actions taken by the object in response to messages received. Attributes represent the state of the object. Objects having the same set of attributes and methods are grouped into the same class. A class is either a primitive class or a complex class. Objects in the respective classes are called primitive objects and complex objects. A primitive class, such as an integer and string, is not further broken down into attributes or substructures. A complex class is defined by a set of attributes, which may be primitive, or complex with user-defined classes as their domains. Since a class *C* may have a complex attribute with domain *C'*, a relationship can be established between *C* and *C'*. The relationship is called the *aggregation* relationship. When arrows connecting classes are used to represent the aggregation relationship, an aggregation hierarchy (or, say, a nested object hierarchy) can be constructed to show the nested structure of the classes.

An example of a nested object hierarchy is extracted from [8] and shown in Fig. 1, where an attribute of any class can be viewed as a nested attribute of the root class.

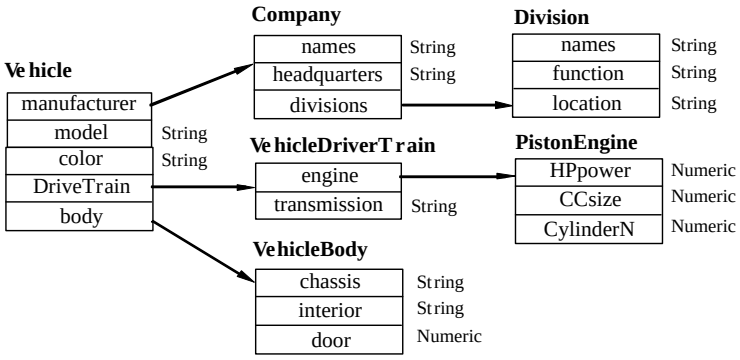


Fig. 1. An example of a nested object hierarchy.

As pointed out in [32], an important element of OODBS is the concept that the value of an attribute of an object can be an object or a set of objects. If an object *O* is referenced as an attribute of object *O*', then *O* is said to be nested in *O*', and *O*' is referred to as the parent object of *O*.

In object-oriented databases, the search condition in a query is expressed as a boolean combination of predicates of the form <attribute operator value>. The attribute may be a nested attribute of the target class. For example, the query "retrieve all red vehicles manufactured by a company with a division located in Ann Arbor" can be expressed as:

```
select   vehicle
where    Vehicle.color = "red"
and      Vehicle.company.Division.location = "Ann Arbor"
```

The search condition against the class Vehicle consists of two predicates, one involving the attribute 'Color' and the other involving the nested attribute 'location'.

Without indexing structures, the above query can be evaluated in a top-down manner as follows. First, the system has to retrieve all of the objects in the class Vehicle and single out those that are red in color. Then, the system retrieves the company objects referenced by the red vehicles and checks the locations of the divisions of the manufacturers. Finally, those red vehicles made by a company that has a division located in "Ann Arbor" are returned.

In this paper, we introduce a new indexing method to speed up this process. This method is based on the technique of signature files and can be summarized as follows:

(i) All signature files are organized into a hierarchical structure to facilitate the implementation of a step-by-step filtering strategy.
(ii) Each signature file is stored as a tree structure to speed up signature file scanning.

In the following, we first discuss signature file hierarchies in section 4. We then discuss signature trees in section 5.

4. SIGNATURES

In this section, we discuss the concept of signature file hierarchies as well as its application to efficient query evaluation. We first discuss how to construct a signature file hierarchy in 4.1. Then, in 4.2, we introduce the concept of query signature hierarchies. Together with signature file hierarchies, this concept enables us to reduce the search space to be explored during a query evaluation.

4.1 Signature File Hierarchy

Signature files are based on the inexact filter. They provide a quick test, which discards many nonqualifying elements. Besides qualifying elements that definitely pass the test, some elements which actually do not satisfy the search requirement may also

pass it accidentally. Such elements are called “false hits” or “false drops” [19, 20]. In an object-oriented database, 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, stored in the signature file (see [17] to know how to construct a signature for an attribute value). An object signature is formed by superimposing its attribute values. (By “superimposing,” we mean a bit-wise OR operation.) Object signatures of a class will be stored sequentially in another signature file.

In the following, we first show how to establish a signature for an object. Then, the construction of signature files for classes will be discussed.

[13] showed that letter triples are the best choice for information carrying text segments in the construction of word signatures. Given an attribute value, for example, “professor”, we will decompose it into a series of triplets: “pro,” “rof,” “ofe,” “fes,” “ess,” and “sor.” Then, using a hash function h, we will map a triplet to an integer k indicating that the kth bit in the string will be set to 1. For instance, assume that we have h(pro) = 2, h(rof) = 4, h(ofe) = 8, and h(fes) = 9. Then, we will establish a bit string: 010 100 011 000 for “professor” as its word signature (see [17] for a detailed discussion). To establish a bit string for an object, we superimpose the signatures of all its attribute values together. Fig. 2 depicts the signature generation and comparison process of an object having three attribute values: “John,” “12345678,” and “professor.”

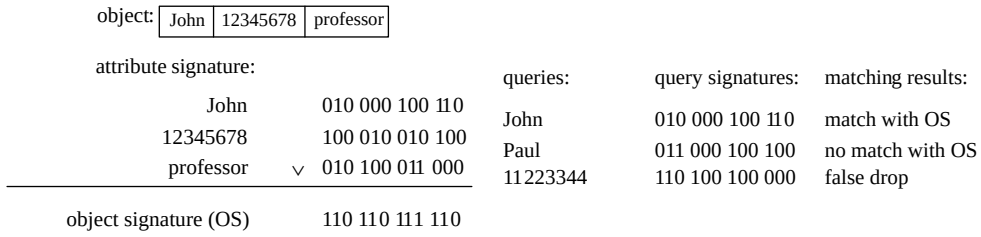


Fig. 2. Signature generation and a comparison.

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 is done 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. 2: (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 really contains 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 does not match the search criteria (false drop). In order to eliminate false drops, the object must be examined after the object signature signifies a successful match.

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 accesses are

prevented. Signature files have a much lower storage overhead and a simpler file structure than inverted indexes.

In terms of an aggregation hierarchy, a signature file hierarchy can be constructed as follows:

(i) The signature of an object is generated by superimposing the signatures of all its primitive and complex attributes.
(ii) The signature of a primitive attribute is obtained by hashing on the attribute values; the signature of a complex attribute is the signature of the object it references.
(iii) Let C be a class, and let o_1, ..., o_l be its objects; there exists a signature file S such that each o_i (i = 1, ..., l) has an entry <*osig*, *oid*> in S.
(iv) Let S_i and S_j be two signature files associated with classes C_i and C_j, respectively. If there exists an arrow from C_i to C_j, then there is implicitly an arrow from S_i to S_j.

To illustrate the above process, consider the class "Division" in the class hierarchy shown in Fig. 1, which contains no complex attributes. The signature of an object o of this class can be constructed as shown in Fig. 3 (a), where each $s(o, x)$ stands for the signature produced for the attribute value x of o and $s(o)$ for the signature of o. (See [17] for a detailed discussion on the generation of a signature for an attribute value.) For a class containing complex attributes, the signature of its objects can be generated in the same way as for a class containing only primitive attributes. The only difference is that the signature of a complex attribute is the signature of the object it references. See Fig. 3 (b) for an illustration. In Fig. 3 (b), *o*' stands for an object of class "Company", and object *o* (of class "Division") is the attribute value of "division" of *o*'.

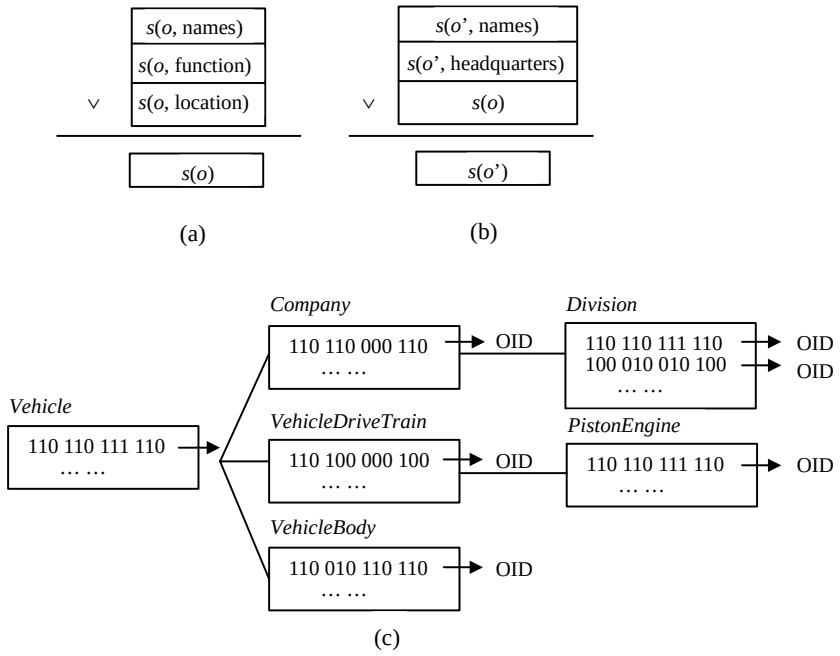


Fig. 3. Signature and signature file hierarchy.

In Fig. 3 (c), we show a signature file hierarchy which may be constructed for a database with the schema shown in Fig. 1.

From the above analysis, we can see that each class will be associated with a signature file with each signature for an object, which is constructed by superimposing the signatures for its attribute values. Then, each object can be considered as a block. To determine the size of a signature file, we use the following formula [13]:

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

where N is the number of signatures in a signature file, m is the signature length, h is the number of bits set to 1 in a signature, and D is the average size of a block.

4.2 Reducing Searching Space Using Signature Files

To cut off irrelevant data as early as possible, we use the top-down approach, by means of which all of objects are retrieved along the path from the target class to its nested attributes specified in the search condition of the query. Then, the value of the nested attribute is checked to decide if it is a desired object or not. With the signature file, the query is evaluated as follows. A query signature s_q for the query Q is generated. s_q is compared with every signature stored in the signature file associated with the target class. If a signature matches s_q, the path is traverse to verify the nested attribute. In the following, a trivial algorithm for top-down retrieval is described. A refined version of it will be discussed later in detail.

Algorithm *top-down-retrieval*;
Input: an object query Q;
Output: a set of OIDs whose texts satisfy the query.
1. Compute the query signature s_q for the query Q.
2. For every entry $<osig_i, oid_i>$ of the signature file associated with the target class, compare s_q with $osig_i$. If $osig_i$ matches s_q, then put oid_i in a temporal set S.
3. For each object in S, traverse the path from the object to the nested attributes specified in Q to eliminate false drops.

Example 1 Consider the query given in section 3. Assume that the signatures for "red" and "Ann Arbor" are s_{red} = 100 110 000 100 and $s_{Ann\ Arbor}$ = 010 000 100 110, respectively. First, we construct $s_q = s_{red} \vee s_{Ann\ Arbor}$ = 100 110 000 100 $\vee$ 010 000 100 110 = 110 110 100 110. It matches the entry in the signature file of *Vehicle* shown in Fig. 3(c). Then, the corresponding OID is put in S. In step (3), we first check the 'color' attribute of OID. If it matches s_{red}, we traverse the paths from this *Vehicle* object to those *Division* objects reachable over some *Company* objects. Such objects of *Division* are checked to eliminate false drops. The signature of the *Division* object shown in Fig. 3(c) matches $s_{Ann\ Arbor}$. Then, the attribute value of "Location" of this object is checked to see whether it really matches "Ann Arbor".

From this example, we can see that the signature files are used only to locate the relevant objects of the target class. The optimization possibility provided by the signature

file hierarchy is not employed at all. It is not efficient because all the subtrees rooted at the relevant objects of the target class have to be searched exhaustively. To overcome this drawback, we have developed another strategy, by means of which attention is paid to the query structure to make the signature file hierarchy useful. To this end, we define the following two concepts.

Definition 1 (*Query tree*) Let $pred_1 \wedge pred_2 \ldots \wedge pred_k$ be the search condition in query Q, where each $pred_i$ is a predicate of the form: <attribute operator value>. Then, all the paths appearing in the search condition constitute a query tree, denoted Q_t. (See Fig. 4 (a) for illustration.)

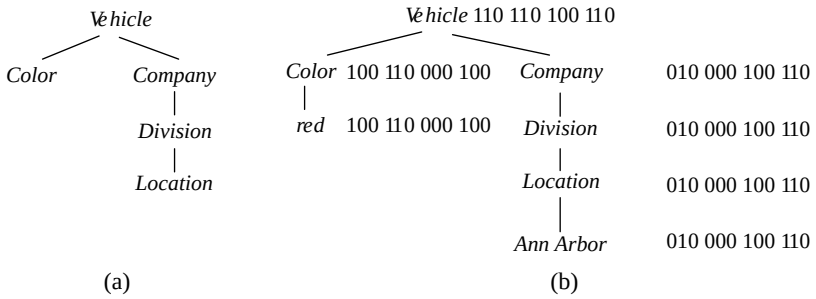


Fig. 4. Query tree and query signature tree.

Definition 2 (*Query signature tree*) Let $p_1.p_2 \ldots .p_n$ be a path in a query tree Q_t (from the root to some leaf). Let <$p_i \ldots .p_n$ *operator value*> be a predicate appearing in the search condition of Q. Then p_n's signature is s_{value}. The signature of a non-leaf node in Q_t can be obtained by superimposing the signatures of its child nodes. The query signature tree is denoted $Q_{(s,t)}$.

For example, for our exemplar query, the query tree and the query signature tree are as shown in Figs. 4 (a) and (b), respectively.

In addition, for the purpose of optimization, each entry in a signature file should be a triple of the form: <*osig, oid, list*>, where *list* is a list containing all the addresses of the object signatures referenced by *oid*.

In the following, we give an algorithm for evaluating queries with the query structure considered. The main idea here is to use the query signature tree to reduce the search space. For this purpose, two stack structures are needed to control depth-first traversal of tree structures: $stack_q$ for $Q_{(s,t)}$ and $stack_c$ for the class hierarchy. In $stack_q$, each element is a signature, while in $stack_c$, each element is a set of objects belonging to the same class reached during class hierarchy traversal.

Algorithm *top-down-hierarchy-retrieval*;
Input: an object query Q;
Output: a set of OIDs whose texts satisfy the query.
1. Compute the query signature hierarchy $Q_{(s,t)}$ for the query Q.

2. Push the root signature of $Q_{(s,t)}$ into $stack_q$; push the set of object OID of the target class into $stack_c$.
3. If $stack_q$ is not empty, $s_q \leftarrow$ pop$stack_q$; else go to (7).
4. $S \leftarrow$ pop $stack_c$; for each $oid_i \in S$, if its signature $osig_i$ does not compare s_q, remove it from S; put S in S_{result}.
5. Let C be the class to which the objects of S belong; let C_1, ..., C_k be the subclasses of C; then partition the OID set of the objects referenced by the objects of S into S_1, ..., S_k such that S_i belongs to C_i; push S_1, ..., S_k into $stack_c$; push the child nodes of s_q into $stack_q$.
6. Go to (3).
7. For each leaf object, check false drops.

By means of this strategy, optimization is achieved by executing step (4). In this step, some objects are filtered using the corresponding signature in the query signature tree. In step (5), the referenced objects and the signatures of the child nodes of the query signature tree are put in $stack_c$ and $stack_q$, respectively. In step (7), the checking of false drops is performed.

Example 2 We will Continue with our running example. Assume that part of the signature file hierarchy constructed for a database with the schema shown in Fig. 1 is of the form shown in the upper part of Fig. 5.

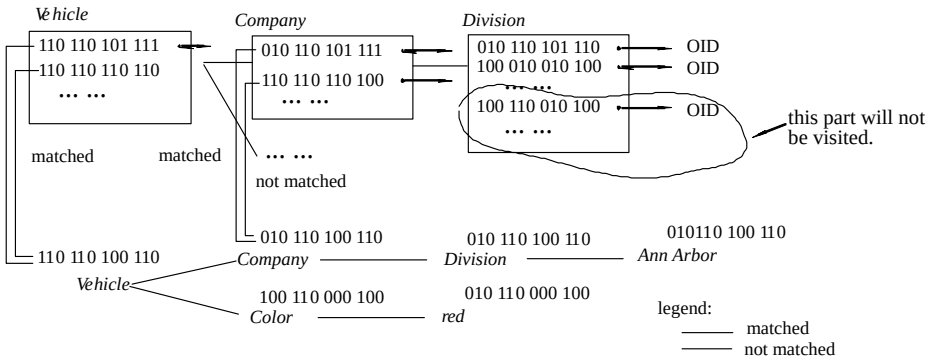


Fig. 5. Illustration of query evaluation.

Since both the top two signatures in the signature file for *Vehicle* (called *V*-file for short) match the corresponding signature in the query signature tree, the signatures referenced by them in the signature file for *Company* (called *C*-file for short) are further checked. Assume that the first signature in *C*-file is referenced by the first signature in V-file, while the second one in *C*-file is referenced by the second one in *V*-file. We can see that the second signature in *C*-file does not match the corresponding signature in the query signature tree. Thus, all those *Division* object signatures referenced by it will not be checked further (see the grey part of Fig. 5 for an illustration.) This is optimal

compared to "*top-down-retrieval*" since by means of "*top-down-retrieval*", checking against all *Division* object signatures has to be performed.

5. SIGNATURE TREES

Although query signature hierarchies can be used to reduce the search space for top-down evaluation of queries, the signature file for the target class has to be scanned completely. If it is large, the amount of time used to search such a file becomes significant. Furthermore, if a path in a query contains variables, the method presented in the previous section does not help a lot. As an example, consider a query containing a search condition like Vehicle.*x*.location = "Ann Arbor" or Vehicle.*x* = "Ann Arbor", where *x* is variable representing a sub-path. Using a top-down method, all the paths whose signatures "match" the signature of "Ann Arbor" will be checked. Obviously, this is not efficient. If backward references are supported, we may try a bottom-up method. In this case, all the signature files for leaf classes will be searched.

From the above discussion, we can see that efficient scanning of signature files should always be supported. One idea for improving the performance is to sort the signature file and then employ binary searching. Unfortunately, this does not work due to the fact that a signature file is only an inexact filter. The following example helps to illustrate this.

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 000010010100. 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 over a signature file like a suffix tree for a text.

How to construct such a tree is discussed in detail in the following subsections.

5.1 Definition of Signature Trees

A signature tree works for a signature file just as a *trie* [31] does for a text. But in a signature tree, each path is a signature identifier, which is not a continuous piece of bits, so it is quite different from a trie, in which each path corresponds to a continuous piece of bits.

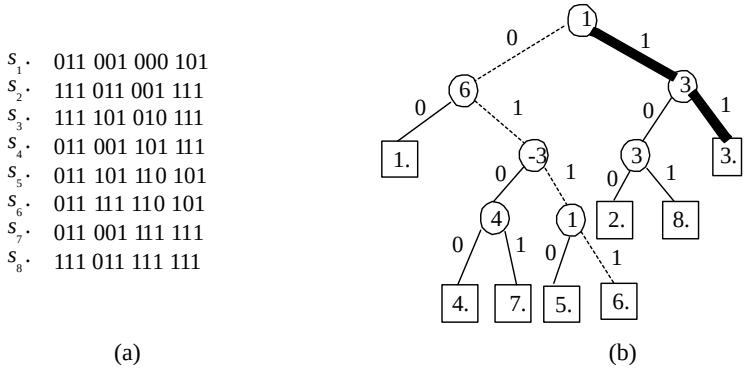
Consider a signature s_i of length m. We denote it as $s_i = s_i[1]s_i[2] \ldots s_i[m]$, where each $s_i[j] \in \{0, 1\}$ ($j = 1, \ldots, m$). We also use $s_i(j_1, \ldots, j_h)$ to denote a sequence of pairs w.r.t. s_i: $(j_1, s_i[j_1])(j_2, s_i[j_2]) \ldots (j_h, s_i[j_h])$, where $1 \le j_k \le m$ for $k \in \{1, \ldots, h\}$.

Definition 3 (*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 $k \neq \bar{i}$ ($1 \le k \le n$) we have

$s_i(j_1, ..., j_h) \neq \bar{s}_k(j_1, ..., j_h)$, then we say that $s_i(j_1, ..., j_h)$ identifies the signature s_i, or we say that $s_i(j_1, ..., j_h)$ is an identifier of s_i w.r.t. S.

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

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



(a) (b)

Fig. 6. Signature tree.

Definition 4 (*signature tree*) A signature tree for a signature file $S = s_1.s_2 ... .s_n$, where $s_i \neq s_j$ for $i \neq j$ 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 0, and the right edge is always labeled 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.
3. Each internal node is associated with a number which tells how many bits to skip when searching.
4. Let $i_1, ..., i_h$ be the numbers associated with the nodes on a path from the root to a leaf labeled i (then, this leaf node is a pointer to the ith signature in S). Let $p_1, …, p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1) ... (j_h, p_h)$ makes up a signature identifier for s_i, $s_i(j_1, ..., j_h)$, where $j_k = \sum_{l=1}^{k} i_l (k = 1, \ldots, h)$.

Example 3 In Fig. 6 (b), we show a signature tree for the signature file shown in Fig. 6 (a). In this signature tree, each edge is labeled 0 or 1 and each leaf node is a pointer to a signature in the signature file. In addition, each internal node is marked with an integer (which is not necessarily positive) used to calculate how many bits to skip when searching. Consider the path going through the nodes marked 1, 6 and − 3. If this path is

searched to locate some signature *s*, then three bits of *s*: *s*[1], *s*[7] (7 = 1 + 6) and *s*[4] (4 = 1 + 6 – 3) are checked at that moment. If *s*[4] = 1, the search goes to the right child of the node marked "– 3." This child node is marked 1, and then the 5th bit of *s*: *s*[5] (5 = 1 + 6 – 3 + 1) is checked.

See the path consisting of dashed lines in Fig. 6 (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 bold lines).

In the next subsection, we discuss how to construct a signature tree for a signature file.

5.2 Construction of Signature Trees

In order to construct a signature tree for a signature file, we need another concept that is the so-called *signature graph*, which is a directed graph and can be generated using an algorithm presented below. This graph can then be transformed into a signature tree in a simple step. To ease explanation, we first introduce some terminology from graph theory.

During a depth-first traversal of a directed graph, we distinguish four types of edges [39]:

(i) *tree edges*: An edge e: $v \rightarrow u$ is a tree edge if u is reached from v when it is scanned, and if u has not been visited before (then at this moment, *defnumber*(u) = 0 if we initially assign 0 to *defnumber*(u) for each node u.)
(ii) *forward edges*: An edge e: $v \rightarrow u$ is a forward edge if when it is scanned for the first time, *defdefnumber*(u) > *defnumber*(v) > 0.
(iii) *back edge*: An edge e: $v \rightarrow u$ is a back edge if when it is scanned for the first time, *defnumber*(v) > *defnumber*(u) > 0, and at the same time, u is an "ancestor" of v.
(iv) *cross edges*: An edge e: $v \rightarrow u$ is a cross edge if when it is scanned for the first time, *defnumber*(v) > *defnumber* > 0, but u is not an "ancestor" of v.

For convenience, we regard an edge of the form: $v \rightarrow v$ as a back edge.
Using the above terms, we define a signature graph as follows.

Definition 5 A signature graph associated with a signature file containing n signatures is a directed graph containing only tree and back edges and at the same time, satisfying the following properties:

1. The graph consists of a header and $n - 1$ nodes.
2. Each node U contains six fields:
   – Pointer to a signature in the signature file, denoted P(U).
   – LLINK(U) and RLINK(U): pointers within the graph. (LLINK is always labeled 0, and RLINK is always labeled 1.)
   – LTAG(U) and RTAG(U): one-bit fields which tell whether or not LLINK and RLINK, respectively, are pointers to sons (tree edges) or to ancestors (back edges) of the node. (For example, when LTAG(U) = 1, this indicates that

LLINK(U) is a pointer to an ancestor of U. When LTAG(U) = 0, this indicates that LLINK(U) is a pointer to a son of U.)

– SKIP(U): a number which tells how many bits to skip when searching.

3. The head has only three fields: Pointer to a signature, LLINK and LTAG.

To construct a signature graph for a signature file, we use two procedures: one for graph search and one for node insertion, which together are, in fact, a modified version of the Patricia algorithm [34]. Using the Patricia algorithm, a tree structure is constructed to index a text (which is represented as a bit string) and each path (from the root to a leaf) corresponds to a prefix of a sistring (see [34] for a definition of the sistring). But in a signature tree, each path corresponds to a signature identifier. Note that a prefix of a sistring is always a continuous piece of a bit sequence, while a signature identifier is not.

At the very beginning, the graph contains an initial node: Header with P(Header) pointing to the first signature in the signature file, LLINK(Header) = Header and LTAG(Header) = 1. (Note that the other three fields for Header are not defined.)

Then, we take the next signature to be inserted into the graph. Let *s* be the next signature we wish to insert. We first execute the procedure *search* for graph search. It must terminate unsuccessfully since no signature is the same as any other one in a signature file. But several bits of *s* can be determined, which agree with some other signature s_0 already inserted into the graph. (See step (6) of *search*.) Assume that *k* bits of *s* agree with s_0, but that *s* differs from s_0 in the $(k + 1)$th position, where *s* has the digit *b* and s_0 has $1 - b$. Now we repeat the procedure *search* with *s* replaced with those *k* bits. This time, the search is successful, and the insertion point can be determined. In a next step, the procedure *insertion* is performed to insert *s* into the graph.

The procedure below consists of six steps.

Procedure *search*

(1) U ← Header; $j \leftarrow 0$; $n \leftarrow$ number of bits of *s*.
(2) Q ← U; U ← LLINK(Q); If LTAG(Q) = 1, go to step (6).
(3) $j \leftarrow j$ + SKIP(U); If $j > n$, go to step (6). (*SKIP(U) is calculated by Procedure *insertion*.*)
(4) If the *j*th bit of *s* is 0, go to step (2); otherwise go to step (5).
(5) Q ← U; U ← RLINK(Q); If RTAG(Q) = 0, go to step (3).
(6) Compare *s* with the signature pointed to by P(U) in the signature file. If they are equal (up to *n* bits, the length of *s*), the algorithm terminates successfully; otherwise, it terminates unsuccessfully. (*Through the comparison, *k* can be determined.*)

The following procedure takes five arguments as the inputs: U, Q, *b*, *j*, and *k*, which are produced during the execution of Procedure *search*.

U is the node encountered; U is determined by line (2) or (5) in Procedure *search*;
Q is U's parent;
b is the first bit of *s*, which differs from the signature pointed to by P(U);

j is a number indicating that if LTAG(Q) = 1 or RTAG(Q) = 1, then the *j*th bit of *s* is checked when Q is visited (see lines (2) and (5)); otherwise, the *j*th bit of *s* is checked when U is visited (see line (3)); and

k is a number indicating how many first bits of *s* match U, which is determined by line (6).

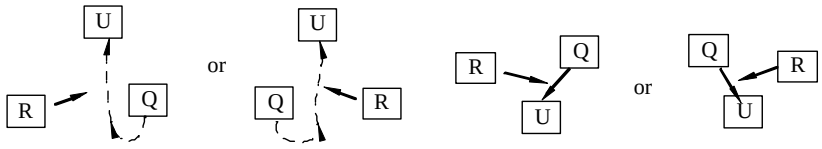
In the following procedure, R represents a new node to be inserted into the graph.

Procedure *insertion*

```
(1)  R ← address of a structure created for the new element;
(2)  P(R) ← position of the new sistring;
(3)  if LLINK(Q) = U then
           {LLINK(Q) ← R; t ← LTAG(Q); LTAG(Q) ← 0;}
         else (*move right; RLINK(Q) = U*)
           {RLINK(Q) ← R; t ← RTAG(Q); RTAG ← 0;}
(4)  if b = 0 then
           {LTAG(R) ← 1; LLINK(R) ← R; RTAG(R) ← t; RLINK(R) ← U;}
         else (*b = 1*)
           {RTAG(R) ← 1; RLINK(R) ← R; LTAG(R) ← t; LLINK(R) ← U;}
(5)  if t = 1 then
           {SKIP(R) ← 1 + k – j;}
       else (*t = 0*)
         {SKIP(R) ← 1 + k – j + SKIP(U); SKIP(U) ← j – k + 1;}
```

In the above procedure, attention should be paid to the calculation of SKIP(R).

If t = 1, then SKIP(R) ← 1 + *k* – *j*, which corresponds to the insertion shown in Fig. 7 (a). This situation happens when Procedure *search* stops after the execution of line (2) and then goes to line (6). If t = 0, then SKIP(R) ← 1 + *k* – *j* + SKIP(U), and at the same time, SKIP(U) is changed: SKIP(U) ← *j* – *k* + 1, which corresponds to the insertion shown in Fig. 7 (b). This situation happens when Procedure *search* stops after the execution of line (3) and then goes to line (6).



(a) (b)

Fig. 7. Illustration of the insertion of a node R.

In addition, we note line (4) in Procedure *insertion*. If *b* = 0, then the corresponding position in the signature pointed to by U is 1. Thus, R should be inserted in such a way that U becomes the right child node of R. Similarly, if *b* = 1, U should become the left child node of R. In Fig. 7, a dashed arrow represents a back edge, and a solid arrow represents a tree edge.

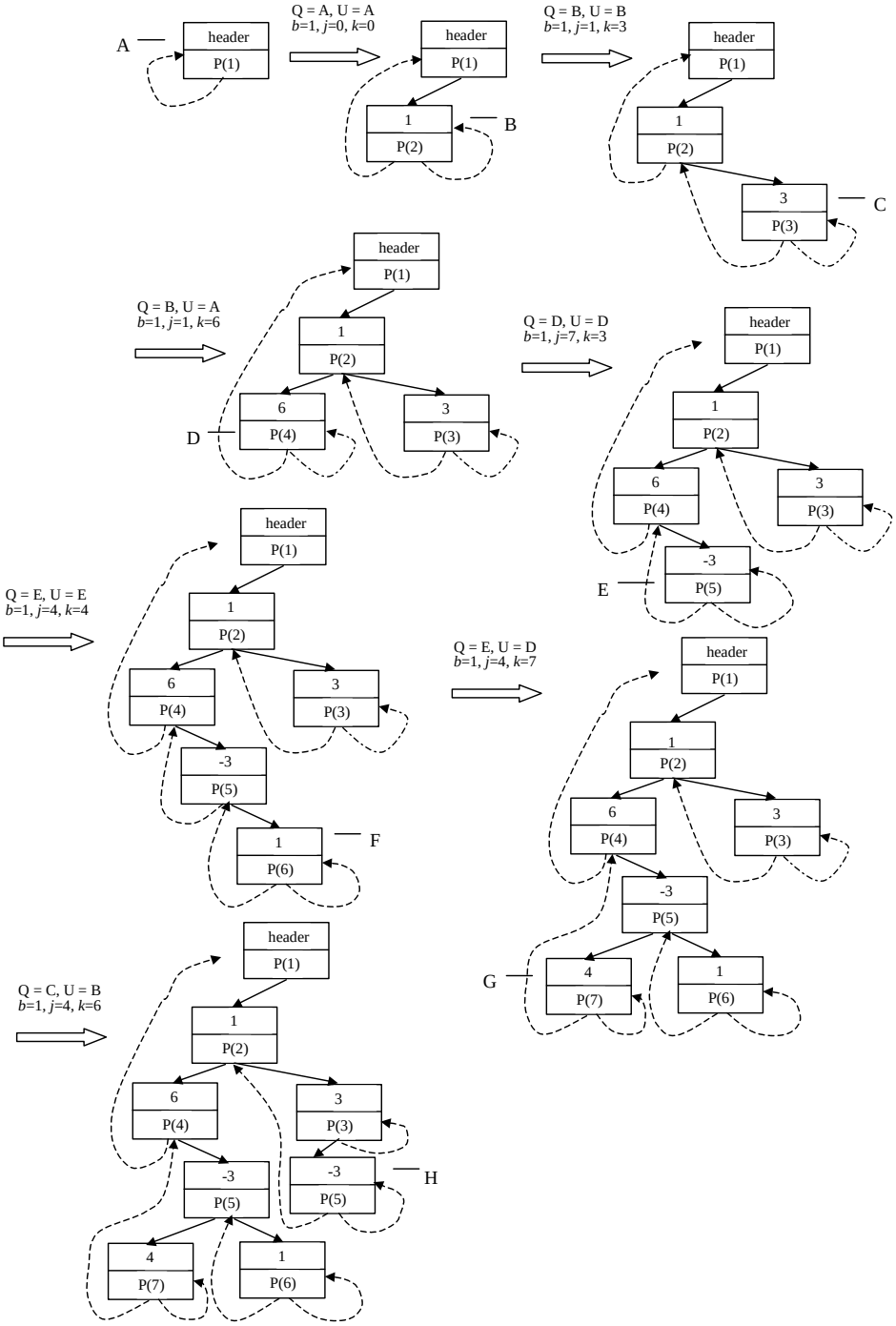


Fig. 8. Sample trace of signature-graph generation.

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

In Fig. 8, P(*i*) represents a pointer to the *i*th signature in the signature file, and *s* = *j* indicates that the *j*th signature is being considered. The nodes are also labelled A, B, ..., H as they are inserted into the graph.

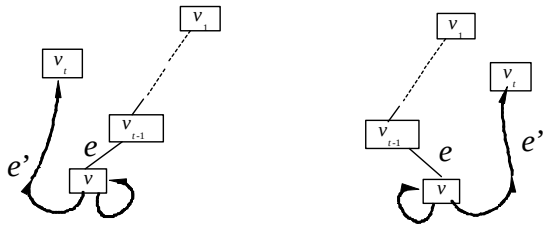
Next we give an important property of a signature graph, which exhibits its main usage.

First, we notice that in a signature graph, each node *v* is associated with a number to show how many bits to skip and with a pointer to a signature in the signature file, denoted $k(v)$ and $p(v)$, respectively. In addition, each edge *e* (except the edge extending from the header to its left child) is labelled with a value (0 or 1), denoted $b(e)$. The edge from the header to its left child is established at the very beginning - before Procedure *search* is executed and no value (0 or 1) is associated with it. We call the left child of the head the *root* of the signature graph.

Definition 6 (*identifying path*) Let $P = v_1.e_1 \ldots v_{t-1}.e_{t-1}.v_t$ be a path in a signature graph, where v_i $(1 \le i \le t)$ represents a node on the path and e_j $(1 \le j \le t-1)$ represents an edge from v_{j-1} to v_j. If v_1 is the root and t_{n-1} is the only back edge appearing on *P*, then *P* is called an *identifying path*.

Proposition 1 Let $P = v_1.e_1 \ldots v_{t-1}.e_{t-1}.v_t$ be an identifying path for some signature *s*, *i.e*, $p(v_t) = s$. Denote $j_l = \sum_{i=1}^{l} k(v_i)$ $(l = 1, \ldots, t-1)$. Then, $s(j_1, j_2, \ldots, j_{t-1}) = (j_1, b(e_1)) \ldots (j_{t-1}, b(e_{t-1}))$ constitutes an identifier for *s*.

Proof: Let $S = s_1.s_2 \ldots .s_n$ be a signature file, and let *G* be a signature graph for it. Let $P = v_1e_1 \ldots v_{t-1}e_{t-1}v_t$ be an identifying path for s_i in *G*. Assume that there exists another signature s_g such that $s_g(j_1, j_2, \ldots, j_{t-1}) = s_i(j_1, j_2, \ldots, j_{t-1})$, where $j_l = \sum_{i=1}^{l} k(v_i)$ $(l = 1, \ldots, t-1)$. Without loss of generality, assume that $g > i$. Then, at the moment when s_g is inserted into *G*, a new node *v* will be inserted as shown in Fig. 9 (a) or (b).



(a) (b)

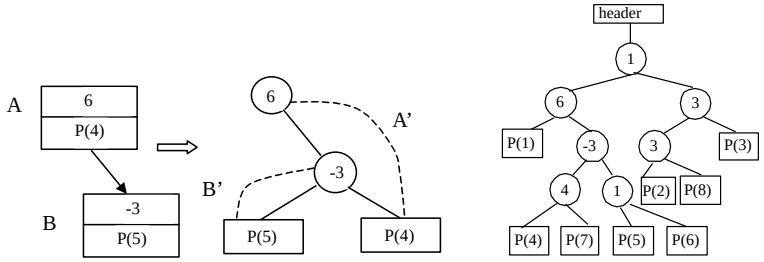
Fig. 9. Inserting a node *v* into *G*.

Fig. 9 shows that the identifying path for s_i should be $v_1.e_1$ ... $v_{t-1}.e.ve'.v_t$, which contradicts the assumption. Therefore, there is no other signature s_g with $s_g(j_1, j_2, ..., j_{t-1}) = (j_1, b(e_1))$ ...$(j_{t-1}, b(e_{t-1}))$, so $s_i(j_1, j_2, ..., j_{t-1})$ is an identifier of s_i. ❑

A signature graph can always be transformed into a signature tree by splitting each node into two nodes. That is, each pointer to the text is separated from the corresponding node. There is an arc from a node v to a separated pointer node u (corresponding to a pointer to the text) if there is a back edge from v to a node containing u in the original graph. See Fig. 10 (a) for an illustration.

In Fig. 10 (a), node A is split into two nodes connected by a dashed line A', and node B is split into two nodes connected by a dashed line B'. The signature tree shown in Fig. 10 (b) is produced from the signature graph shown in Fig. 8.

Based on the above analysis, we have the following proposition.



(a) (b)

Fig. 10. Node splitting.

Proposition 2 Each signature graph can be transformed into a signature tree by splitting each node into two nodes as described above.

Proof: Let G be a signature graph, and let T be another graph obtained by splitting each node of G as described above. Then, each pointer (in G) to a signature (in the corresponding signature file) becomes a leaf node in T; and each path in T from the root (corresponding to the root node in G) to a leaf is just an identifying path in G. Note that the transformation will not change the binary property of G (i.e., each internal node in T will have two child nodes) or the edge label. Based on Definition 4, T is a signature tree. ❑

5.3 Searching Signature Trees

Next, we discuss how to search a signature tree to model the behaviour of a signature file as a filter. Let s_q be the node encountered during a traversal of the query signature hierarchy $Q_{(s,h)}$. The i-th position of s_q is denoted by $s_q(i)$. During the traversal of a signature tree, inexact matching is defined as follows:

(i) Let v be the node encountered, and let $s_q(i)$ 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 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 node in $Q_{(s,h)}$;
Output: a set of object OIDS whose signatures survive the checking process;
1. Let s_q be the node encountered during a traversal of the query signature hierarchy $Q_{(s,h)}$. The i-th position of s_q is denoted by $s_q(i)$. $S \leftarrow \varnothing$.
2. Push the root of the signature tree into $stack_p$.
3. If $stack_p$ is not empty, then $v \leftarrow$ pop $stack_p$; else return(S).
4. If v is not a leaf node, then $i \leftarrow$ skip(v);
   If $s_q(i) = 0$, then push c_r and c_l into $stack_p$ (where c_r and c_l are v's right and left children, respectively); otherwise, push only c_r into $stack_p$.
5. Compare s_q with the signature pointed to by $p(v)$.
   If s_q matches, then $S \leftarrow S \cup \{\text{OID}\}$, where OID is the object identifier associated with the signature pointed to by $p(v)$.

The following example illustrates the main idea of the algorithm.

Example 3 Consider the signature file and the signature tree shown in Fig. 6 once again.

Assume that s_q = 000 100 100 000. Then, only part of the signature tree (indicated by bold lines in Fig. 11) will be searched. When a leaf node is reached, the signature pointed to by the leaf node will be checked against s_q. Obviously, this process is much more efficient than sequential searching. In this example, only 42 bits are checked (6 bits during tree search and 36 bits during signature checking). But when the signature file is scanned, 96 bits are 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 keep half of a subtree from being visited. Compared with the time complexity of signature file scanning $O(N)$, this is a major benefit. We will discuss this issue in the next section in more detail.

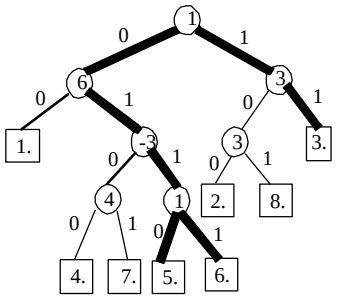


Fig. 11. Signature tree search.

6. PERFORMANCE

An indexing technique is always associated with some trade-offs. This is also true for the method presented in this paper. In this section, we first show the time savings obtained using the signature file hierarchy in 6.1. Then, in 6.2, we demonstrate the benefits obtained by using a signature tree. In 6.3, we analyze the extra space overhead of a signature tree.

6.1 Performance of Signature File Hierarchies

For the purpose of comparison, we consider a linear setting as in [42]. That is, query evaluation is performed along a class path as shown in Fig. 12.

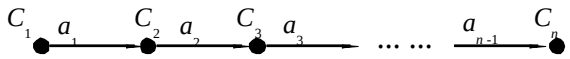


Fig. 12. Class path.

In this figure, each class C_i has a complex attribute a_i whose domain is C_{i+1}. In addition, we assume that a predicate p_i is defined over C_i and is executed to evaluate the query. In the following, the parameters and assumptions used for the performance evaluation are given:

P_i: Probability that an object in class C_i satisfies p_i.
v_i: Number of objects in class C_i visited by a nested loop (brute force) top-down strategy to process the given query.
P_f: False drop probability of signatures.
P_q: Probability that an object satisfies checking against the corresponding signature in the query signature hierarchy.
d: Average out-degree of objects (i.e., the average number of objects referenced by an object).
N_i: Number of objects in class C_i.

In the following, we use these parameters to estimate the numbers of objects visited when evaluating a query using three different methods: top-down-retrieval (TDR), the method proposed in [42] (referred to as *Yong's method* later) and top-down-hierarchy-retrieval (TDHR).

Top-down-retrieval (TDR)

With this method, the number of the visited objects in class C_i for evaluating a query can be computed using the following formula:

$$v_i = d \cdot v_{i-1} \cdot P_{i-1} \qquad (2 \le i \le n)$$
$$= d \cdot v_1 \cdot P_1 \cdot P_{i-1}$$

Therefore, the total number of the visited objects is equal to the following sum:

$$v = \sum_{i=1}^{n} v_i = v_1 \cdot (1 + d \cdot \sum_{i=2}^{n} \prod_{j=1}^{i} P_{j-1}).$$

Yong's method

Yong's method was proposed in [42]. With this method, the signature of a referenced object is stored in the referring one. Then, predicate checking can be performed against their signatures before they are accessed. In this way, a lot of I/O operations can be saved. According to this property, we analyze its performance as follows.

Let v_i' be the number of objects in class C_i visited by Yong's method. Due to the earlier checks done for the referenced objects, for each class C_i, v_i' can be computed as follows:

$$\begin{aligned} v_i' &= v_i - (1 - P_i) \cdot v_i + (1 - P_i)P_f \cdot v_i \\ &= v_i \cdot (P_i + (1 - P_i)P_f), \end{aligned}$$

where $(P_i + (1 - P_i)P_f)$ is the probability that an object is not removed by checking against the signatures of the referenced objects. Thus, the total number of the objects accessed by Yong's method is

$$v' = \sum_{i=1}^{n} v_i' = v_1 \cdot (1 + d \cdot \sum_{i=2}^{n} (P_i + (1 - P_i) P_f) \prod_{j=1}^{i} P_{j-1}).$$

Top-down-hierarchy-retrieval (TDHR)

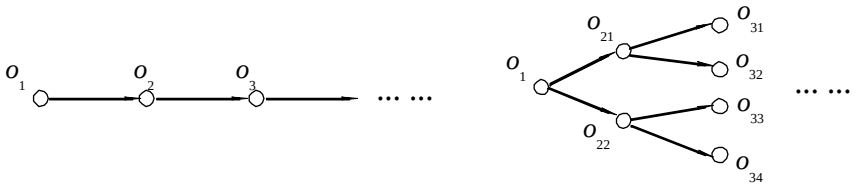
Top-down-hierarchy-retrieval has a stronger filtering ability than Yong's method. This is because in each check against a node in a query signature hierarchy, not only is the predicate related to the current node involved, but also some other predicates whose impacts are propagated up several paths to that node. We represent such impacts by means of a probability P_q. Accordingly, we give the following analysis. Let v_i" be the number of objects in class C_i visited by TDHR. Then, we have

$$v_i'' = v_i \cdot (P_i + (1 - P_i)P_f) \cdot P_q$$

Thus, the total number of visited objects is

$$v'' = \sum_{i=1}^{n} v_i'' = v_1 \cdot P_q \cdot (1 + d \cdot \sum_{i=2}^{n} (P_i + (1 - P_i) P_f) \prod_{j=1}^{i} P_{j-1}).$$

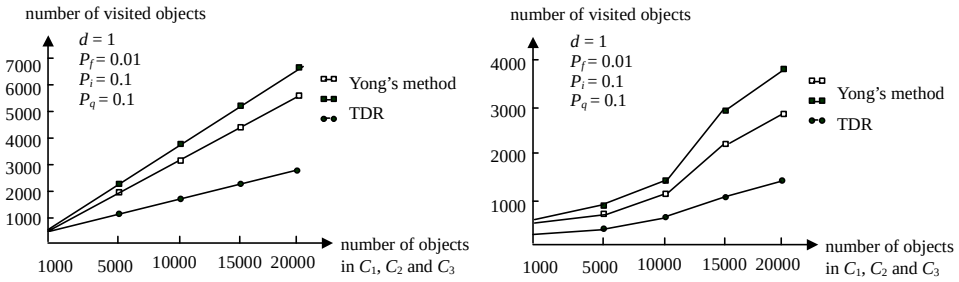
We compare the above three methods above using two simple abstract data sets. For the first data set, each object has only one sub-object (i.e, $d = 1$; see Fig. 13 (a)). For the second, each object refers two sub-objects (i.e, $d = 2$; see Fig. 13 (b)).



(a) (b)

Fig. 13. Two data sets.

Figs. 14 (a) and (b) show the comparison results for the two data sets distributed in three classes C_1, C_2 and C_3, respectively. In the performance analysis, we assumed that P_i = 0.1 (i = 1, 2, 3), P_f = 0.01, and P_q = 0.5. Contrary to [42], the visited objects of class C_1 were counted since by using the query signature hierarchy, a lot of objects of the target class can also be removed by checking the corresponding signature file, leading to a drastic reduction in the total number of accessed objects.



(a) (b)

Fig. 14. Comparison results.

The figure shows that we can achieve high performance by means of "*top-down-hierarchy-retrieval*." From an abstract point of view, the query signature hierarchy is a "global" filter, while the replication technique developed in Yong's method can be thought of as a "local" one. Both reduce the number of objects accessed.

6.2 Performance of Signature Trees

To analyze the performance of the signature tree, we consider four parameters: N, the number of signatures in a signature file; m, the signature length; h, the number of bits set to 1 in a signature; and D, the size of a block. When the signature is on average half-populated with 1s and half-populated with 0s, the false drop probability is minimized [13]. In such a setting, the two parameters N and m satisfy the following inequality:

$$N \le \binom{m}{m/2}.$$

We have the above inequality based on a simple observation that if $N > \binom{m}{m/2}$, then there must exist two signatures having the same binary strings. In this case, one of them will be removed from the signature file. In the following, we concentrate on the estimation of $N/2^l$ - the number of signatures to be checked using a signature tree.

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

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

Then, we have

$$N \le \sqrt{\frac{2}{\pi m}} \cdot 2^m$$

From this, we have $\log_2 N \le \frac{1}{2} - \frac{1}{2} \cdot \log_2 \pi - \frac{1}{2} \cdot \log_2 m + m.$

Thus, m satisfies the following inequality:

$$\log_2 N - \frac{1}{2} - \frac{1}{2} \cdot \log_2 \pi - \frac{1}{2} \cdot \log_2 m \le m$$

According to [13], in the case that the average block signature involves an equal number of 1s and 0s, the three design parameters h, m, and D satisfy the relationship below:

$$m \times \ln 2 = h \times D$$

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

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

$$N / 2^l \sim N / 2^h = N / 2^{\frac{m \ln 2}{D}}.$$

$$N / 2^{\frac{m \ln 2}{D}} \le N / 2^{(\ln N - \frac{1}{2} + \frac{1}{2}\log \pi + \frac{1}{2}\log m)\frac{\ln 2}{D}} = N / (N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \sqrt{m})^{\frac{\ln 2}{D}}.$$

Finally, we have the inequality

$$N / 2^{\frac{m \ln 2}{D}} \le N / (N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \sqrt{m})^{\frac{\ln 2}{D}}$$
$$\le N / (N \cdot \frac{1}{\sqrt{2}} \cdot \sqrt{\pi} \cdot \sqrt{\log N - \frac{1}{2} + \log \sqrt{\pi} + \log \sqrt{m}})^{\frac{\ln 2}{D}}$$
$$\sim (\sqrt{\frac{2}{\pi}})^{\frac{\ln 2}{D}} \cdot N / (N \log N)^{\frac{\ln 2}{D}}$$

Fig. 15 shows the calculation results obtained using the above formula.

In Fig. 15, the number of signatures checked is computed in terms of N - the size of a signature file. From this, we can see that the performance of signature tree searching

worsens as the size of the block increases. This is because, given a fixed signature length, a larger block requires fewer bits in a signature set to 1, which weakens the filtering power of signature trees when it comes to single term query processing.

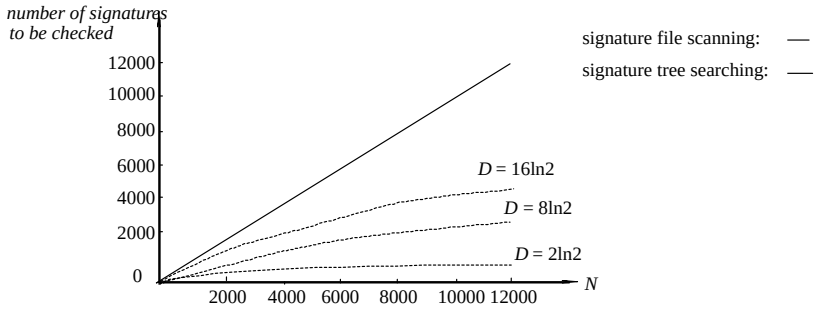


Fig. 15. Time complexities of signature file scanning and signature tree searching.

Normally, l should be considered as a variable parameter. If $l > h$, the number of signatures to be checked will be reduced. If $l < h$, more signatures will be checked.

6.3 Extra Space Overhead of a Signature Tree

Note that a signature tree is a binary tree. Thus, a signature tree can be stored as a set of triples of the form: <v, lp, rp>, 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 m, and that the number of signatures in a file is N. (The size of the signature file is, therefore, $N \times m$ bits.) Then, for each v, we need $log_2 m$ bits, and for each lp (rp), we need $log_2 N$ bits. Accordingly, for all the internal nodes of a signature tree, we need space for $N \times \log_2 m + 2N \times \log_2 N$ bits. To mitigate this problem to some extent, we use the following relative address encoding:

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

$$N \times \log_2 m + 2\sum_{i=0}^{k} 2^i (i+1),$$

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

7. SIGNATURE TREE MAINTENANCE

In this section, we address how to maintain a signature tree. First, in 7.1, we discuss the case where a signature tree can entirely fit in main memory. Then, in 7.2, we discuss the case where a signature tree cannot entirely fit in main memory.

7.1 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 and from a tree, respectively, 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 algorithm *insert*() once with *s* as the input (see 5.2).

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. 16 (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*.)

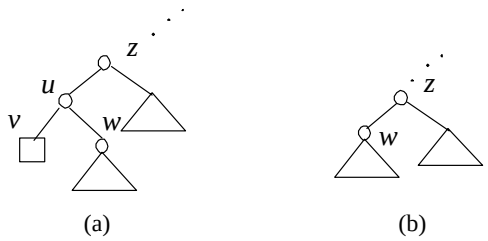


Fig. 16. Illustration of deleting a signature.

The resulting signature tree is as shown in Fig. 16 (b).

From the above analysis, we can see that maintaining an internal signature tree is an easy task.

7.2 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 a signature tree). In this case, a signature tree is stored in a series of pages organized into a tree structure as shown in Fig. 17, in which each node corresponds to a page containing a binary tree.

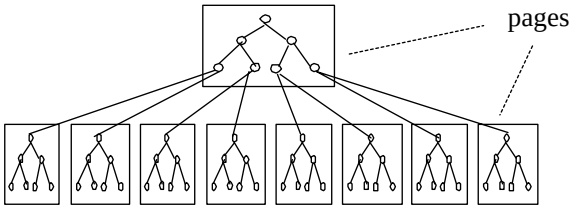


Fig. 17. An external signature tree.

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 page nodes and to the nodes in *T* as binary nodes or, simply, nodes.)

1. Each internal page node *n* of *ET* is of the form: $b_n(r_n, a_{n1}, ..., a_{ni_n})$, 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* be 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, then 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 [17], the generation of word signatures is a random process. Therefore, in a large signature file, we expect approximately equal numbers of 1s and 0s, which hints that a signature tree for a large signature file is approximately 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 tree balance, internal page nodes may split or merge 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*. This can be done by moving the binary tree within that page into main memory and then inserting the node into the tree as

discussed in 7.1. If for the binary tree b in m, we have $|b| > 2^k$, the following node-splitting process 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} be the left and right child nodes 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 that $b_{m2}(r_{m1}, a_{mi_{j+1}}, ..., a_{mi_m})$ 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, we promote r_m to 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, we split n as described above. The node-splitting is repeated 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 7.1. 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}, then we move $b_{m'}$ into m and afterwards eliminate m'. Let n be the parent page node of m, and let r be the parent node of r_m and $r_{m'}$. We 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}, then we merge it with its left or right sibling if possible. This process is repeated 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 or any of its left and right siblings due to the properties of a signature tree, which may leave an external signature tree unbalanced. According to the analysis of 5.4, however, this is not a typical case.

Finally, we point out that for an application in which the signature files do not frequently change, the internal page nodes of an ET can be implemented as a heap structure. In this way, a lot of space can be saved.

8. CONCLUSIONS

In this paper, a new indexing technique has been proposed. The main idea of this approach is to combine of signature file hierarchies and signature trees. To optimize the traversal of object hierarchies, we build signature file hierarchies to cut off irrelevant branches as early as possible. However, since the signature file works only as an inexact filter, it can not be sorted and binary search, thus, can not be utilized to speed up signature file scanning. To this end, we construct a signature tree over each signature file

which appears as a node in the signature file hierarchy. In this way, sequential searching can be avoided, which leads to a reduction of the time needed to search a signature file by one order of magnitude or more.

REFERENCES

1. S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon, "Querying documents in object databases," *International Journal on Digital Libraries*, Vol. 1, 1997, pp. 5-19.
2. S. Abiteboul, S. Cluet, and T. Milo, "Querying and updating the file," in *Proceedings of the 9th International Conference on Very Large Data Bases* (*VLDB*), 1993, pp. 386-397.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.
4. F. Bancihon, S. Cluet, and C. Delobel, "A query language for O_2," *Building an Object-Oriented Database System: The Story of O_2*, 1992, pp. 234-255.
5. E. Bertino, "Optimization of queries using nested indices," in *Proceedings of International Conference on Extending Database Technology*, 1990, pp. 44-59.
6. F. Bancihon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System: The Story of O_2*, 1992.
7. E. Bertino and C. Guglielmani, "Optimization of object-oriented queries using path indices," in *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992, pp. 140-149.
8. E. Bertino and W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, 1989, pp. 196-214.
9. R. Bayer and K. Unterrauer, "Prefix B-tree," *ACM Transactions on Database Systems*, Vol. 2, 1998, pp. 11-26.
10. A. Cardenas, "Analysis and performance of inverted data base structures," *Communications of ACM*, Vol. 18, 1987, pp. 253-263.
11. R. G. G. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley Publishing Company, Inc., 1991.
12. S. Choenni, E. Bertino, H. M. Blanken, and T. Chang, "On the selection of optimal index configuration in OO databases," in *Proceedings of 10th International Conference on Data Engineering*, 1994, pp. 526-537.
13. S. Christodoulakis and C. Faloutsos, "Design consideration for a message file server," *IEEE Transactions on Software Engineering*, Vol. 10, 1984, pp. 201-210.
14. C. Y. Chan, C. H. Goh, and B. C. Ooi, "Indexing OODB instances based on access proximity," in *Proceedings of 13th International Conference on Data Engineering*, 1997, pp. 14-21.
15. M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, 1994.
16. U. Deppisch, "S-tree: A dynamic balanced signature index for office retrieval," *ACM SIGIR Conference*, 1986, pp. 77-87.
17. D. Dervos, Y. Manolopoulos, and P. Linardis, "Comparison of signature file models with superimposed coding," *Journal of Information Processing Letters*, Vol. 65, 1998, pp. 101-106.

18. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Benjamin Cumming, California, 1989.
19. C. Faloutsos, "Access methods for text," *ACM Computing Surveys*, Vol. 17, 1985, pp. 49-74.
20. C. Faloutsos, "Signature files," *Information Retrieval: Data Structures & Algorithms*, W. B. Frakes and R. Baeza-Yates, ed., Prentice Hall, New Jersey, 1992, pp. 44-65.
21. F. Fotouhi, T. G. Lee, and W. I. Grosky, "The generalized index model for object-oriented database systems," in *10th Annual International Phonix Conference on Computers and Communication*, 1991, pp. 302-308.
22. S. W. Golomb, "Run-length encoding," *IEEE Transactions on Information Theory*, Vol. 12, 1996, pp. 399- 401.
23. *OpenODB Reference Manual B3185A*, Hewlett-Packard, 1992.
24. D. Harman, E. Fox, and R. Baeza-Yates, "Inverted files," *Information Retrieval: Data Structures & Algorithms*, 1992, pp. 28-43.
25. R. Haskin, "Special purpose processors for text retrieval," *Database Engineering*, Vol. 4, 1981, pp. 16-29.
26. Y. Ishikawa, H. Kitagawa, and N. Ohbo, "Evaluation of signature files as set access facilities in OODBs," in *Procreedings of ACM SIGMOD International Conference on Management of Data*, 1993, pp. 247-256.
27. W. Kim, *Introduction to Object-Oriented Databases*, The MIT Press, 1990.
28. W. Kim, "A model of queries for object-oriented databases," in *Proceedings of International Conference on Very Large Data Base*, 1989, pp. 423-432.
29. W. Kim, K. C. Kim, and A. Dale, *Indexing Techniques for Object Oriented Databases*, Addison Wesley, 1989, pp. 371-394.
30. A. Kemper and G. Moerkotte, "Access support relations: an indexing method for object bases," *Information Systems*, Vol. 17, 1992, pp. 117-145.
31. D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Publishing, London, 1973.
32. W. Lee and D. L. Lee, "Signature file methods for indexing object-oriented database systems," in *Proceedings of 2nd International Conference on Data and Knowledge Engineering: Theory and Application*, 1992, pp. 616-622.
33. C. C. Low, B. C. Ooi, and H. Lu, "H-trees: a dynamic associative search index for OODB," in *Proceedings of 1992 ACM SIGMOD Conference on the Management of Data*, 1992, pp. 134-143.
34. D. R. Morrison, "PATRICIA – practical algorithm to retrieve information coded in alphanumeric," *Journal of ACM*, Vol. 15, 1968, pp. 514-534.
35. T. A. Mueck and M. L. Polaschek, "The multikey type index for persistent object sets," in *Proceedings of 13th International Conference on Data Engineering*, 1997, pp. 22- 31.
36. D. Maier and J. Stein, "Indexing in an object-oriented DBMS," in *Proceedings of International Workshop on OODB Systems*, 1986, pp. 171-182.
37. A. Moffat and J. Zobel, "Self-indexing inverted files for fast text retrieval," *ACM Transaction on Information Systems*, Vol. 14, 1996, pp. 349-379.

38. B. Sreenath and S. Seshadri, "The hcC-tree: an efficient index structure for object oriented database," in *Proceedings of International Conference on Very Large Database*, 1994, pp. 203-213.
39. R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, Vol. 1, 1972, pp. 146-150.
40. I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes*, Van Nostrand Reinhold, 1994.
41. H. Yokota, Y. Kanemasa, and J. Miyazaki, "Fat-Btree: an update-conscious parallel directory structure," in *Proceedings of 15th International Conference on Data Engineering*, 1999, pp. 448-457.
42. H. S. Yong, S. Lee, and H. J. Kim, "Applying signatures for forward traversal query processing in object-oriented databases," in *Proceedings of 10th International Conference on Data Engineering*, 1994, pp. 518-525.
43. J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems*, Vol. 23, 1998, pp. 453-490.

Yangjun Chen received his B.S. degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and his diploma and Ph.D. degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a research assistant professor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a senior engineer at the German National Research Center of Information Technology (GMD) for more than two years. After a short staying at Alberta University, he joined the Department of Business Computing at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, constraint satisfaction problem, graph theory, and combinatorics. He has more than 80 publications in these areas.