Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases

Yangjun Chen and Karl Aberer

IPSI Institute, GMD GmbH, Dolivostr. 15,
64293 Darmstadt, Germany

Abstract. In this paper, a new indexing technique to support the query evaluation in document databases is proposed. The key idea of the method is the combination of the technique of pat-trees with signature files. While the signature files are built to expedite the traversal of object hierarchies, the pat-trees are constructed to speed up both the signature file searching and the text scanning. In this way, high performance can be achieved.

1 Introduction

We consider the combination of two different indexing techniques: signature files and pat-trees for optimizing query evaluation in document databases. Signature files can be feasibly organized into a hierarchical structure and therefore suitable for indexing documents stored structurally (in an object-oriented database). Concretely, it can be used to speed up the traversal along object hierarchies by filtering non-relevant objects as early as possible. The drawback of the signature file is that it is an inexact filter. A key word (appearing in the query) surviving the checking may be not in the text. Therefore, a scanning of the text has to be carried out to see whether the text really contains it. Furthermore, if many texts should be checked or a text is long, much time will be spent to do this task if no index is available. On the other hand, for a large document database, a signature file may be very large by itself and therefore the overhead caused by the sequential search of such files become significant. To make the matter worse, since the signature file works only as an inexact filter, it can not be sorted and thus the binary search can not be applied (see 5.3). To this end, we combine the technique of the pat-tree with the technique of the signature file to make both the searching of a signature file and the scanning of a text more efficient. As we can see later, the processes of the tree traversal and the signature file searching (also text scanning) supported by the pat-trees will be interleaved, leading to an efficient method.

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 a document database and discuss the approaches to query processing. Section 4 is devoted to hierarchies of representative words. In Section 5, we discuss different indexing techniques as well as their combination. In Section 6, we present our algorithms for evaluating queries with signature file hierarchies and pat-trees used. Finally, Section 7 is a short conclusion.

2 Related work

Index techniques have been extensively investigated in both information retrieval and database research area and a lot of 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 [Kn73], suffix trees [CR94] and their variants, position trees [AHU74, We73] built over flat files as well as pat-trees [Mo68] using individual bits of keys. Equipped with these mechanisms, the system performance can be improved by one order of magnitude or more. Signature file [CF84, DGL98, Fa92] is a method quite different from the tree indexing techniques, by which each key word is assigned a signature and a set of key words is assigned a "super" signature constructed by superimposing the signatures in the set. It works as an inexact filter. Another interesting approach is the inverted index that is a set of postings lists [HEBL92, WMB94], each of which maps one keyword to a list of links to the documents containing that keyword. Inverted indices can be implemented as sorted arrays, tries and various hashing structures [HFBL92]. The drawback of this approach is that much space is required for indices and not suitable for the implementation of a layered index structure as the signature file does (see 5.2).

Some of the techniques mentioned above have been modified or extended to support the query evaluation in databases. A notable example is B-tree [BU77] and its variants such as B$^+$-tree and B*-tree [EN89] which were developed based on the balancing mechanism of binary trees with some special features augmented to ease the tree balance or to minimize the accesses to data files. In addition, the signature files have been proved to be useful in object-oriented databases and many researches have been done to integrate this technique into the object-oriented databases to improve the response time of a query [YA94, LL92].

One may notice that there is no application of pat-trees or position trees in the database area. It is due to the essential distinction between the key structures of a relation and the key words in a text. In this paper, we explore a way to use the pat-tree in document databases indirectly - we build the pat-trees over signature files (therefore, it is a method of indexing over indexes.) Obviously, if an object-oriented database system uses signature files as the indexing mechanism, our method can also be used to improve the query evaluation. Additionally, in a document database, texts stored as attribute values may be long and it is necessary to index them if the space overhead for indexes remains low.

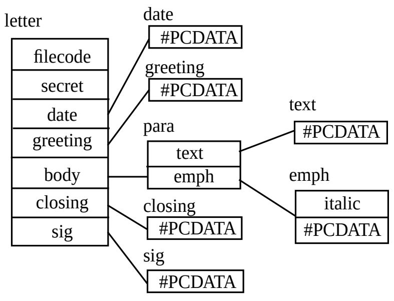
3 Queries in document databases

In document database systems, an element is represented as an object, which consists of methods and attributes. Methods are procedures and functions associated with an object defining the 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 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 aggregation relationship. Using arrows connecting classes to represent aggregation relationship, an aggregation hierarchy can be constructed to show the nested structure of the classes.

Fig. 1(a) shows a possible DTD for *letter* documents (SGML/XML documents). The class definition for the corresponding elements and the resulting aggregation hierarchy are shown in Fig. 1(b).

```
1. <!DOCTYPE        letter [
2. <!ELEMENT        letter    - - (date, greeting, body, closing, sig)>
3. <!ATTLIST        letter
      filecode     NUMBER        #REQUIRED
      secret      (yes | no)    "no">
4. <!ELEMENT        body        - - (para)+>
5. <!ELEMENT        (date, greeting, closing, sig) - - (#PCDATA)>
6. <!ELEMENT        para        - - (text | emph)*>
7. <!ELEMENT        (text | emph)    - - (#PCDATA)>
8.  <!ATTLIST       emph
      italic      (yes | no)          "yes">
9. <!ENTITY         salute              "Dear">
   ]>
```

(a)



(b)

Fig. 1. DTD and hierarchy structure

To show the application of our indexing method and to provide a background for discussion, here we consider a kind of simple queries: key word queries (but with the path concept involved) which are most frequently utilized in practice. In such queries, a search condition is expressed as conjunction of predicates of the form: <*path operator value*>. The *path* is of the form: $p_1.p_2$ ... $.p_n$, where each p_i (i = 1, 2, ..., n-1) represents a class name and p_n is an attribute name. In general, an *operator* represents a (set) relation operation {$\supseteq$, =} and a *value* is a set of individual (representative) words connected with "$\wedge$" or "$\vee$".

As an example, consider the query: retrieve all letters received in 1993, which contain strings "SGML" and "database", which can be expressed as follows:

select *
where *Letter.Date* $\supseteq$ "1993"
and *Letter.Body.Para.text* $\supseteq$ "SGML" $\wedge$ "database"

The search condition against the classes *Letter*, *Date* and *Para* consists of two predicates, one involving the nested attribute *Date* and the other involving the nested attribute *text* of *Para*.

A top-down approach will search all of the objects in class *Letter* and those whose *date* attribute contains "1993" will be singled out. Then, the system retrieves the *Body* objects referred by the *Letter* objects found in the previous scan and checks their *para* attribute, which leads to retrieving part of *Para* objects referred by the found *Body* objects. Finally, those *Para* objects containing both "SGML" and "database" are returned.

From the above description, we can see that two processes are involved to evaluate a query. They are

(1) traversing along object hierarchies and

(2) scanning texts to see whether the key words appearing in the query are contained.

What we want is to optimize these two processes by building indexes over both the object hierarchies and the texts stored as attribute values. For this purpose, we first introduce an important concept: *representative word hierarchy* in the next section, which is useful to specify the key idea of our method.

4 Representative word hierarchies

As is well-known, in a traditional document system, it is important to assign *representative words* to documents, capable of representing document contents and used to obtain access whenever documents are wanted. Then, signature files can be built over them and organized into a hierarchy, corresponding to the hierarchical structure of a document stored in databases.

- Representative words

Given a set of documents doc_i (i = 1, ..., n), we can identify a set of (representative) words W_i for each doc_i to discriminate it from others by computing *weight* for each word:

$$weight_{ik} = f_{ik} \cdot signal_k, \qquad (1)$$

where f_{ik} represents the frequency of word k appearing in doc_i and $signal_k$ is the signal value of word k, which can be computed as shown in [SM83]. For example, the representative words of a *letter* document may be a set (denoted W_{letter}) like {January, 27, 1993, Jean, SGML, databases, information, regards, Genise}, determined by applying the above formula to the actual document set. In some cases, we can use almost all words in the document only with high-frequency words (called *stop list*) removed. The words appearing in the stop list are poor discriminators and cannot possibly be used by themselves to identify document content.

- Representative word hierarchy

As mentioned above, the representative word hierarchy is built in terms of the storage structure of the documents, which can be illustrated as shown in Fig. 2.

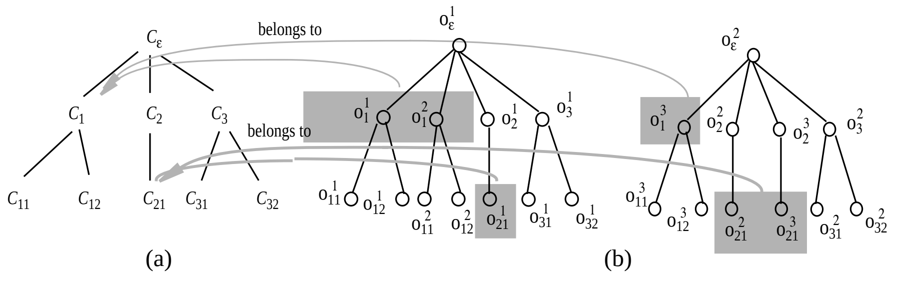


Fig. 2. Class hierarchy and object hierarchy

For exposition, we assume that the database is of the schema as shown in Fig. 2(a), containing two documents which are organized into two object hierarchies as shown in Fig. 2(b). In the figure, the objects are subscripted in such a way that the objects with the same subscript belong to the same class. The superscripts are used to number the objects of a class. For example, o_1^1, o_1^2 and o_1^3 belong to C_1 and superscribed with 1, 2, and 3, respectively.

Complying with such an storage structure of documents, the representative word hierarchy can be constructed in a recursive way as follows.

(i) First, each o_ε^i is associated with a set of representative words (representing doc_i) determined using formula (1).

(ii) Let o be an object and W be the set of representative words associated with o. Let o_1, ..., o_n be the sub-objects of o, which are partitioned into several groups g_1, ..., g_m ($m \leq n$) such that each group g_j belongs to the same class. Then, W is partitioned into W_1, ..., W_m by regarding each g_j as a single document with the following two rules observed:

(1) If $w \in W$ and w appears in g_j, but $w \notin W_j$, then add w to W_j: $W_j \leftarrow W_j \cup \{w\}$.

(2) If $w \notin W$, but $w \in W_j$, then delete w from W_j: $W_j \leftarrow W_j - \{w\}$.

(iii) For each g_j, its W_j is further partitioned into W_{j1}, ..., W_{jk} such that each $o_{jl} \in g_j$ is associated with W_{jl}, which is made in the same way as step (ii).

(iv) For each o_{jl} and its W_{jl}, do step (ii).

For example, W_{letter} may be partitioned into {January, 27, 1993}, {Jean}, {SGML, databases, information}, {regards}, {Genise} for those texts accommodated at classes *Date*, *Greeting*, *Body*, *Closing* and *Sig*, respectively (see Fig. 3).

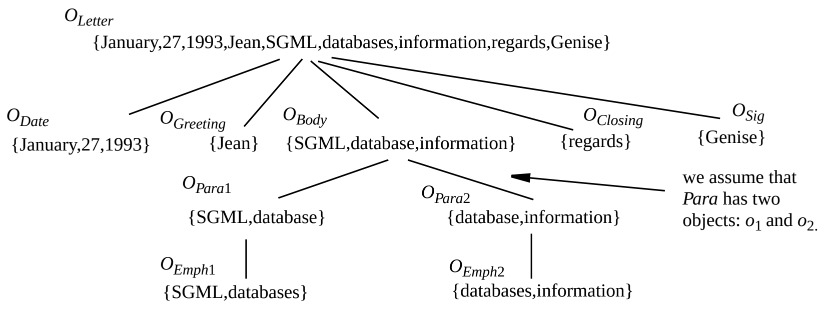


Fig. 3. Representative word hierarchy

For each W, its signature can be calculated as discussed in 5.2, based on which we construct signature files for each class (subclass) by collecting the relevant signatures together.

5 Pat trees and signature files

Now we discuss our indexing technique. The main idea of it can be summarized as follows:

(i) construct the signature file hierarchies to support the traversal of the object hierarchies;

(ii) build the pat trees over the texts to expedite the text scanning; and

(iii) build the pat tree over the signature files to avoid the sequential search of them.

In the following, we first discuss the technique of the pat trees in 5.1. Then, in 5.2, the signature file hierarchies are addressed. We motivate the combination of the pat trees and signature files in 5.3. (The discussion on the application of such a combination is shifted to Section 6.)

5.1 Pat-tree built over representative words

Pat-tree is a digital (binary) tree, by which the key words (or representative words) is represented as a sequence of digits (in our case, the ASCII codes of characters are used.) During a traversal of a pat-tree, the individual bits of the key words are used to decide on the branching.

Given a text containing key words kw_1, kw_2, ..., kw_n. We define the corresponding key strings as the strings, each starting from the place where the corresponding key word first appears to the end of the text. To make each key string not be a prefix of another, we add a special symbol, say $, to the end of the text, which appears nowhere else. If a key word appears several times in the text, only the first appearance is used. The following example helps for illustration.

	1	30	51	57	65
text:	This paragraph describes the	technique concerning	SGML,	XML and	databases.$

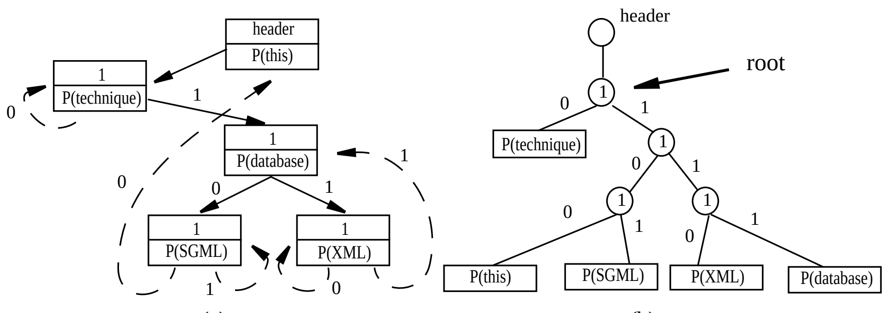
Each position in the text indicates a *suffix* or *semi-infinite string* (sistring), which goes to the end of the text. There are about 70 characters in the example; therefore, there are about 70 semi-infinite strings. We look at only semi-infinite strings that start at the beginning of key words. There are 5 such strings (“This ....”, “technique ...”, etc.) Here we assume that “this”, “technique”, “SGML”, “XML” and database” are five key words. The numbers above the text show positions of starting characters of these sistrings in the text. Concretely, we have the key strings as shown in Fig. 4(a).

key-string1:	This paragraph ... .$	key-string1 = 10011 ... ...
key-string2:	technique ... .$	key-string2 = 01000 ... ...
key-string1:	SGML ... .$	key-string3 = 10101 ... ...
key-string1:	XML ... .$	key-string4 = 11100 ... ...
key-string1:	database ... .$	key-string5 = 11001 ... ...
	(a)	(b)

Fig. 4. Key strings and arrays of bits

Note that we take the text as an array of bits and assume that they are of the form as shown in Fig. 4(b).

Using *patricia* algorithm [Mo68], a graph shown in Fig. 5(a) can be constructed. (Due to space limitation, a complete description of this algorithm can not be given here.)



(a) (b)

Fig. 5. Pat-tree

It consists of a header and n - 1 = 5 - 1 = 4 nodes. Each node contains six fields:

- Pointer to the text. In Fig. 5(a), P(x) (where x is a word) shown within each node is a pointer to the text, e.g., P(SGML) is the number 51, the starting place of key-string3 in the text.

- LLINK and RLINK: pointers within the graph. (LLINK is always labeled with 0 and RLINK is always labeled with 1.)
- LTAG and RTAG: one-bit fields which tell whether or not LLINK and RLINK, respectively, are pointers to sons or to ancestors of the node. The dotted arcs in Fig. 4(a) correspond to pointers whose TAG bit is 1.
- SKIP: a number which tells how many bits to skip when searching, as explained below. The SKIP fields are shown as numbers within each node of Fig. 5(a).

The graph shown in Fig. 5(a) can be represented as a tree by splitting each node into two ones as shown in Fig. 5(b). 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 an ancestor link (dotted arc) from *v* to a node containing *u* in the original graph.

A search in Pat-tree is carried out as follows. Suppose we are looking up the word SGML (assume that its bit pattern is 10101 11100 11000 10001). We start by looking at the SKIP field of the root node (see Fig. 5(b)), which tells us to examine the first bit of the argument (the bit pattern of SGML). It is 1, so we move to the right. The SKIP field in the next node tells us to look at the 1 + 1 = 2nd bit of the argument. It is 0, so we move to the left. The SKIP field of the next node tells us to look at the 2 + 1 = 3rd bit, which is 1; now we reach a leaf node which refer us to the text at position P(SGML). The search path we have taken would occur for any argument whose bit pattern is 010x ... x (where “x” represents “don’t care), and we must check to see if it matches the unique key which begins with that pattern. If it matches, we know that the text contains SGML. Otherwise, SGML does not appears in the text since the path leading to the leaf node with P(SGML) is unique and nowhere else through a path matches the pattern 010x ... x.

From Fig. 5(b) we see that the size of a pat-tree is very small. Its space complexity is bounded by O(num_w), where num_w is the number of the representative words of a text. If the text is long, it is worth while building a pat-tree to make the text scanning quickly.

5.2 Signature File Hierarchy

Signature files are based on the inexact filter. They provide a quick test, which discards many of the nonqualifying elements. But the qualifying elements definitely pass the test although some elements which actually do not satisfy the search requirement may also pass it accidentally. Such elements are called “false hits” or “false drops”. In a document database, an element is stored as an object and represented by a set of representative words assigned to the text stored in it. The signature of a representative word is a hash-coded bit string of length *k* with *m* bit set to “1”, stored in the “signature file” (see [Fa85]). An object signature is formed by superimposing the signatures of its representative words. Object signatures of a class will be stored sequentially in another signature file. Fig. 6 depicts the signature generation and comparison process of an object having a text attribute value which is represented by three words, say “SGML”, “database”, and “information”.

text: ... SGML ... databases ... information ...

representative word signature:

SGML		010 000 100 110
database		100 010 010 100
information	∨	010 100 011 000
object signature (OS)		110 110 111 110

queries:	query signatures:	matchin results:
SGML	010 000 100 110	match with OS
XML	011 000 100 100	no match with OS
informatik	110 100 100 000	false drop

Fig. 6. Signature generation and comparison

When a query arrives, the object signatures are scanned and many nonqualifying objects are discarded. The rest are either checked (so that the “false drops” are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for representative words. The query signature is then compared to every object signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 6: (1) the object matches the query; that is, for every bit set in s_q, the corresponding bit in the object signature s is also set (i.e., $s \wedge s_q = s_q$) and the object contains really the query word; (2) the object doesn’t match the query (i.e., $s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the object in fact doesn’t match the search criteria (false drop). In order to eliminate false drops, the object must be examined after the object signature signifies a successful match.

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 the representative word hierarchy, a signature file hierarchy can be constructed as follows:

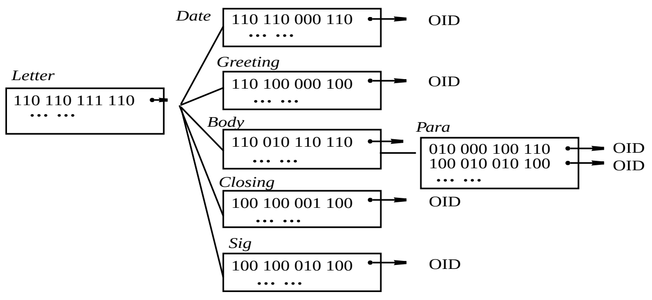
(i) For every representative word *w*, assign it a signature *wsig*. (How to construct a signature for a representative word can be found in [DGL98].)

(ii) Let *o* be an object and $W = \{w_1, ..., w_k\}$ be the set of representative words associated with it. There exists an entry <*osig*, *oid*>, where *osig* is the signature of *o* and *oid* is the object identifier of *o*. *osig* is obtained by superimposing the signatures of w_i ($i = 1, ..., k$).

(iii) Let *C* be a class and $o_1, ..., o_l$ 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.

As an example, see the signature file hierarchy shown in Fig. 7, which is constructed in terms of the representative word hierarchy shown in Fig. 3.



words:	signature:
January	110 000 000 110
23	110 110 000 000
1993	100 110 000 100
Jean	110 100 000 100
SGML	010 000 100 110
database	100 010 010 100
information	010 100 011 000
regards	100 100 001 100
Genise	100 100 010 100

Fig. 7. Signature file hierarchy

5.3 Combining pat trees and signature files

In a large document database, a signature file itself may be long. Therefore, the time elapsed for searching such a file becomes significant. A first idea to improve the performance is to sort the signature file and then employ a binary searching. Unfortunately, this does not work due to the fact that a signature file is only an inexact filter. The following example helps for illustration.

Consider a sorted signature file containing only three signatures:

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

Assume that the query signature s_q is equal to 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 pat-tree over a signature file by considering each signature as a word code; but use a different matching strategy. That is, for the pat-tree built for a text, the exact matching is used (i.e., 1 matches 1 and 0 matches 0) while for the pat-tree built for a signature file the inexact matching will be utilized to work as an inexact filter. Concretely, it works as follow. Let $s_q(i)$ be a bit checked when a node v is met during a traversal of the pat-tree. If $s_q(i) = 0$, then the entire subtree rooted at v will be further searched. If $s_q(i) = 1$, we move to the right child of v. That is, only the subtree rooted at the right child of v will be further traversed.

How to control this process is discussed in detail in the following section.

6 Retrieval

In this section, we first briefly sketch how to use the signature files to cut branches. Then, we discuss how to use the pat-tree to speed up this process in great detail. During the traversal of a pat-tree, an inexact matching is carried out, which is quiet different from the method shown in 5.1.

The signature file can be utilized to expedite the query evaluation by constructing a query signature tree for the submitted (simple) query, in which each node is a signature. To evaluate the query, the corresponding signature file hierarchy S_f will be searched against the query signature tree Q_T and at each step, a node in Q_T is checked against the corresponding node (a signature file) in S_f to discard the non-relevant branches. (See [CA98] for a detailed discussion of this process.)

Obviously, many signature files will be searched during a query evaluation. However, since the signature file works as an inexact filter, we can not expedite its search by sorting it and then using a binary search. For this reason, we build a pat-tree over each signature file. To work as an inexact filter, we use a different matching strategy. Let s_q be the node encountered during a traversal of the query signature tree Q_T. The i-th position of s_q is denoted as $s_q(i)$. During the traversal of a pat-tree, the inexact matching is defined as follows:

(i) Let b be the node encountered and $s_q(i)$ be the position to be checked.

(ii) If $s_q(i) = 1$, we move to the right child of b.

(iii) If $s_q(i) = 0$, both the right and left child of b will be visited.

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

To implement this inexact matching strategy during a traversal of a pat-tree, we search the pat-tree in the depth-first manner and maintain a stack structure $stack_p$ during the process.

Algorithm *pat-tree-search*

input: a node in Q_T;

output: set of object OIDS whose signatures survive the checking;

```
1. Let $s_q$ be the node encountered during a traversal of the query signature hierarchy
   $Q_T$. The $i$-th position of $s_q$ is denoted as $s_q(i)$. $S \leftarrow \emptyset$.
2. Push the root of the pat-tree into $stack_p$.
3. If $stack_p$ is not empty, $b \leftarrow$ pop $stack_p$; else return($S$).
4. If $b$ is not a leaf node, $i \leftarrow$ skip($b$);
   If $s_q(i) = 0$, push $c_r$ and $c_l$ into $stack_p$; (where $c_r$ and $c_l$ are $b$'s right and left child,
   respectively.) otherwise, push only $c_r$ into $stack_p$.
5. Compare $s_q$ with the substring beginning from position pointed by $b$.
   If $s_q$ matches, $S \leftarrow S \cup$ {OID}, where OID is the object identifier associated with
   the substring (signature).
```

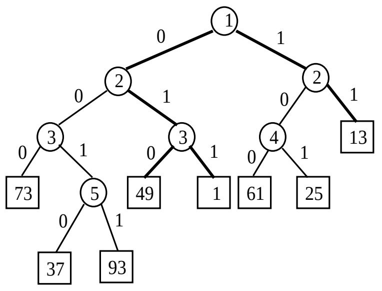
The following example helps for illustrating the main idea of the algorithm.

Example 3 Consider the signature file shown in Fig. 8(a). The pat-tree built over it is shown in Fig. 8(b).

Assume s_q = 010 000 000 000. Then, only part of the pat-tree (marked with thick edges) will be searched. On reaching a leaf node, the bit substring from the position pointed by the leaf node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching. If the signature file contains N signatures, this method 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.

```
011 001 000 101
110 010 001 011
100 100 010 111
001 000 101 110
010 001 011 100
100 010 111 000
000 101 110 000
001 011 100 000
```

(a)



(b)

Fig. 8. Pat-tree for signature file

7 Conclusion

In this paper, a new indexing technique has been proposed. The main idea of this approach consists in the combination of pat-trees and signature files. To optimize the traversal of object hierarchies, we build signature file hierarchies to cut off non-relevant branches as early as possible. However, since the signature file works only as an inexact filter, it can not be sorted and thus the binary search can not be utilized to improve the efficiency. To this end, we construct a pat-tree over each signature file which appears as a node in the signature file hierarchy. In this way, the sequential search can be avoided. In addition, we may build a pat-tree over a text stored as an attribute value if it is very long. At last, we notice that a pat-tree itself is small and therefore no much space overhead is assumed. On the other hand, since each sequential search (of a signature file or a text) is replaced with a small tree search along one or several paths (see Subsection 5.1 and Section 6), the time complexities of both the traversal of a signature file hierarchy and the text search can be reduced by one order of magnitude or more.

References

ACCM96 S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte and J. Simeon, “Querying documents in object databases,” *Int. J. on Digital Libraries*, Vol. 1, No. 1, Jan. 1997, pp. 5-19.

ACM93 S. Abiteboul, S. Cluet and T. Milo, “Querying and Uodating the File,” *Proc. of the 9th VLDB Conference*, Dublin, Ireland, 1993, pp. 386-397.

AHU74 Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Com., London, 1969.

BA94 K. Böhm and K. Aberer, “Storing HyTime Documents in an Obeject-Oriented Database,” *Proc. of 3th Int. Conf. on Information and Knowledge Management*, Gaithersburg, Maryland, ACM, Nov. 1994, pp. 26-33.

BDK92 F. Bancihon, C. Delobel and P. Kanellakis, “*Building an Object-oriented Database System: The Story of O_2*,” San Mateo, California, Morgan Kaufman, 1992.

BANY97 K. Böhm, K. Aberer, E.J. Neuhold and X. Yang, “Structured Document Storage and Refined Declarative and NAvigational Access Mechanism in HyperStorm,” *Int. J of VLDB*, 1997.

BU77 R. Bayer and K. Unterrauer, “Prefix B-tree,” ACM Transaction on Database Systems, 2(1), 11-26.

CA98 Y. Chen, K. Aberer, Layered Index Structures in Document Database Systems, *Proc.* 7th *Int. Conference on Information and Knowledge Management (CIKM)*, Bethesda, MD, USA: ACM, 1998, pp. 406-413.

CF84 S. Christodoulakis and C. Faloutsos, “Design consideration for a message file server,” *IEEE Trans. Software Engineering*, 10(2) (1984) 201-210.

Cr86 D.A. Cruse, *Lexical Semantics*, Cambridge University Press, 1986.

CR94 Crochemore, M. and Rytter, W., *Text Algorithms*. Oxford University Press, New York, 1994.

CST92 W.B. Croft, L.A. Smith and H.R. Turtle, “A Loosely Coupled Integration of a Text Retrieval System and an Object Oriented Database,” *Proc. of 15th Ann. Int. SIGIR*, Denmark, June 1992.

DaD88 C. Damier and B. Defude, “The Document Management Component of a Multimedia Data Model,” *Proc. of 11th Int. Conf. on Research&Development in Information Retrieval*, Grenoble, France, 1988, pp. 451-464.

DD94 S.J. DeRose and D.D. Durand, “*Making Hypermedia Work: A User’s Guide to HyTime*,” Kluwer Academic Publishers, London, 1994.

DGL98 A. Dessmark, O. Garrido and A. Lingas, “Comparison of signature file models with superimposed coding,” *J. of Information Processing Letter* 65 (1998) 101 - 106.

DWL92 S.C. Deerwester, K. Waclena and M. Lamar, “A Textual Object Management System,” *Proc. of 15th Ann. Int. SIGIR*, Denmark, 1992.

EN89 R. Elmasri and S. B. Navathe, *Fundamantals of Database Systems*, Benjamin Cumming, California, 1989.

Fa85 C. Faloutsos, “Access Methods for Text,” *ACM Computing Surveys*, 17(1), 1985, pp. 49-74.

Fa92 C. Faloutsos, “Signature Files,” in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 44-65.

GBS92 G.H. Gonnet, R.A. Baeza-Yates, “New Indices for Text: Pat Trees and Pat Arrays,” in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 66-82.

Hew92 Hewlett-Packard, *OpenODB Reference Manual B3185A*, 1992.

HFBL92 D. Harman, E. Fox, R. and Baeza-Yates, “Inverted Files,” in: *Information Retrieval: Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 28-43.

Kn73 D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Pub. London, 1973.

LL92 W. Lee and D.L. Lee, “Signature File Methods for Indexing Object-Oriented Database Systems,” *Proc. ICIC'92 - 2nd Int. Conf. on Data and Knowledge Engineering: Theory and Application*, Hongkong, Dec. 1992, pp. 616-622.

Ma90 I.A. Macleod, “Storage and Retrieval of Structured Documents,” *J. of Information Processing & Management*, Vol. 26, No. 2, 1990, pp. 197-208.

[Mo68] Morrison, D.R., PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of Association for Computing Machinary*, Vol. 15, No. 4, Oct. 1968, pp. 514-534.

Sch93 P. Schäuble, “SPIDER: A MultiMedia Forum - An Interactive Online Journal,” *Proc. of Conf. on Electronic Publishing*, John Wiley & Sons, Ltd, 1994, pp. 413-422.

SM83 G. Salton and M.J. McGill, “*Introduction to Modern Information Retrieval*,” McGray-Hill Int. Book Com., Hamburg, 1983.

VAB96 M. Volz, K. Aberer and K. Böhm, “Applying a Flexible OODBMS-IRS_Coupling to Structured Document Handling,” *Proc. of 12th Int. Conf. on Data Engineering*, New Orleans, 1996, pp. 10-19.

VML95 VODAK V 4.0 User Manual. *Technical Report 910*, GMD-IPSI, St. Augustin, April 1995.

WMB94 I.H. Witten, A. Moffat and T.C. Bell, *Managing Gigabytes*, Van Nostrand Reinhold, 1992.

YA94 T.W. Yan and J. Annevelink, “Integrating a Structural-Text Retrieval System with an Object-Oriented Database System,” *Proc. of 20th VLDB Conf.*, Santiago, Chile, 1994, pp. 740-749.

YLK94 H.S. Yong, S. Lee and H.J. Kim, “Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases,” *Proc. of 10th Int. Conf. on Data Engineering*, Houston, Texas, Feb. 1994, pp. 518-525.