Signature File Hierarchies and Signature Graphs: a New Index Method for Object-Oriented Databases

Yangjun Chen* and Yibin Chen
Dept. of Business Computing
University of Winnipeg, Manitoba, Canada R3B 2E9

ABSTRACT

In this paper, we propose a new index structure for object-oriented databases. The main idea of this is a graph structure, called a signature graph, which is constructed over a signature file generated for a class and improves the search of a signature file dramatically. In addition, the signature files (accordingly, the signature graphs) can be organized into a hierarchy according to the nested structure (called the aggregation hierarchy) of classes in an object-oriented database, which leads to another significant improvements.

Categories & Subject Descriptors: H.2.4

General Terms: System

Key Words: object-oriented databases, index structure, signature files, signature graphs

1. INTRODUCTION

In the past two decades, object-oriented database systems (OODBS) have attracted a significant amount of attention in academic and industrial communities [4, 10]. Several experimental and commercial systems such as GemStone [15], Orion [11] and O2 [2] have been developed. The powerful modeling capability is a major advantage of OODBS over relational databases. However, much work still need to be done on query processing, optimization, and indexing techniques in order to improve the performance.

In this paper, we propose a new index structure that can be used to improve query evaluation significantly. First, we organize a set of sequential signature files into a hierarchical structure (in which each node is a signature file) to reduce the search space during a query evaluation. Second, we store a single signature file itself as a graph, the so-called *signature graph*, to expedite the scanning of a single signature file. When a signature file is large by itself, the amount of time saved is significant using this approach. A closely related work is the S-tree proposed in [6]. It is in fact a R-tree built over a signature file. Thus, it can be used to speed up the location of a signature in a signature file just like a R-tree for keys in a relational database. The methods proposed in [16, 17] are in fact the improved S-trees, suited for set-valued attributes. However, in the signature graph 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.

The rest part of the paper is organized as follows. In Section 2, we discuss signature files and signature graphs. In Section 3, we show how to construct a signature file hierarchy for an object-oriented database and how to use such an index structure to speed up query evaluation. Finally, Section 4 is a short conclusion.

SAC '04, March 14-17, 2004, Nicosia, Cyprus.

2. SIGNATURE FILES AND SIGNATURE GRAPHS

In this section, we discuss the concept of signature files and signature graphs. We first discuss what is a signature file in 2.1. Then, in 2.2, we show the structure of a signature graph, its construction and how it can be searched.

2.1 Signature Files

Signature files are based on the inexact filter. They provide a quick test, which discards many of the nonqualifying elements. But the qualifying elements definitely pass the test although some elements which actually do not satisfy the search requirement may also pass it accidentally. Such elements are called "false hits" or "false drops" [8, 9]. 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 [7] 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 file, called a *signature file*. Fig. 1 depicts the signature generation and comparison process of an object having three attribute values: "John", "12345678", and "professor".

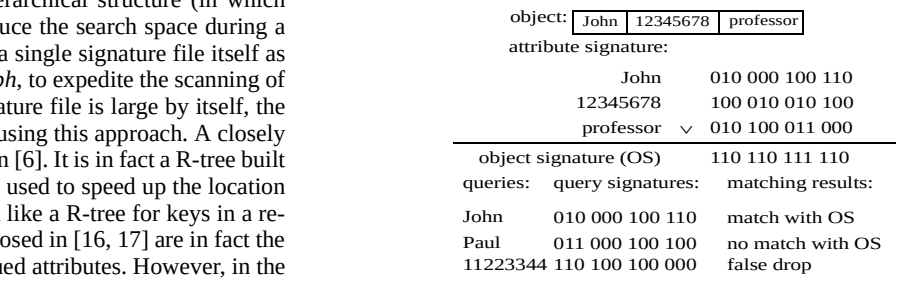

Fig. 1. Signature generation and comparison

When a query arrives, the object signatures are scanned and many nonqualifying objects are discarded. The rest are either checked (so that the "false drops" are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for attribute values. The query signature is then compared to every object signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 1: (1) the object matches the query; that is, for every bit set in s_q, the corresponding bit in the object signature s is also set (i.e., $s \wedge s_q = s_q$) and the object contains really the query word; (2) the object doesn't match the query (i.e., $s \wedge s_q \neq s_q$); and (3) the signature comparison

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

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 non-qualifying objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object access is prevented. Signature files have a much lower storage overhead and a simple file structure than inverted indexes.

From the above analysis, we 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 [5]:

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

where 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.

2.2. Signature Graphs

To find a matching signature, a signature file has to be scanned. If it is large, the amount of time elapsed for searching such a file becomes significant. A first idea to improve this process 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 a different way and organize a signature file into a graph, called a *signature graph*, which will be discussed in this section in great detail.

Definition of signature graphs

A signature graph working for a signature file is just like a *trie* [12, 14] for a text. But in a signature graph, each path visited to find a signature that matches a query signature corresponds to a signature identifier, which is not a continuous piece of bits, and quite different from a trie in which each path corresponds to a continuous piece of bits.

Definition 1. (*signature graph*) A signature graph G for a signature file $S = s_1.s_2 \ldots .s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = m$ for $k = 1, \ldots, n$, is a graph $G = (V, E)$ such that

1. each node $v \in V$ is of the form (p, *skip*), where p is a pointer to a signature s in S, and *skip* is a non-negative integer i. If $i > 0$, it tells that the ith bit of s_q will be checked when searching. If $i = 0$, s will be compared with s_q.

2. Let $e = (u, v) \in E$. Then, e is labeled with 0 or 1 and $skip(u) > 0$. Let $skip(u) = i$. If e is labeled with 0 and $i > 0$, the ith bit of the signature pointed to by $p(v)$ is 0. If e is labeled with 1 and $i > 0$, the ith bit of the signature pointed to by $p(v)$ is 1. A node v with $skip(u) = 0$ does not have any children. □

In Fig. 2(b), we show a signature graph for the signature file shown in Fig. 2(a).

In Fig. 2, each p_i represents a pointer to a s_i ($i = 1, \ldots, 8$).

In the following, we first discuss how a signature graph is constructed. Then, we discuss how to use signature graphs to speed up the search of signature files.

Construction of Signature Graphs

Below we give an algorithm to construct a signature graph for a signature file, which needs $O(N \cdot m)$ time, where N represents the number of signatures in the signature file and m is the length of a signature.

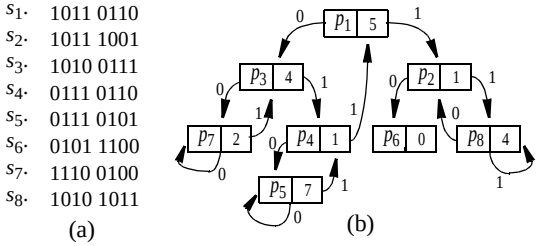

Fig. 2. Signature file and signature graph

At the very beginning, the tree contains an initial node: a node v with $p(v)$ pointing to the first signature and $skip(v) = 0$.

Then, we take the next signature to be inserted into the graph. Let s be the next signature we wish to enter. We traverse the graph from the root and each encountered node will be marked. Let v be a node encountered and assume that $skip(v) = i$. If v is not marked and $i > 0$, check $s[i]$ and mark v. If $s[i] = 0$, we go left. Otherwise, we go right. If $i = 0$ or v is marked, we compare s with the signature s' pointed to by $p(v)$. s' can not be the same as s since in S there is no signature which is identical to anyone else. (If there are two identical signatures s_1 and s_2, we remove s_2 and associate the oids of s_1 and s_2 with s_1.) But several bits of s can be determined, which agree with s'. Assume that the first k bits of s agree with s'; but s differs from s' in the $(k + 1)$th position, where s has the digit b and s' has $1 - b$. We construct a new node u with $skip(u) = k + 1$ and $p(u)$ pointing to s. Let $w_1 \rightarrow w_2 \ldots \rightarrow w_j \rightarrow v$ be the accessed path. Then, make u the left child of w_j if v is a left child of w_j; otherwise, make u the right child of w_j. If $b = 1$, we make v be the left child of u and the right pointer of u pointing to itself. If $b = 0$, we make v be the right child of u and the left pointer of u pointing to itself.

The following is the formal description of the algorithm.

Algorithm *sig-graph-generation*(*file*)
begin
  construct a root node r with $skip(r) = 0$ and $p(v)$ pointing to s_1;
  for j = 2 to n **do**
    call insert(s_j);
end

Procedure *insert*(*s*)
begin
1 *stack* ← *root*;
2 **while** *stack* not empty **do**
3 {v ← pop(*stack*);
4 **if** v is not marked and $skip(r) \neq 0$ **then**
5 {i ← $skip(v)$; mark v;
6 **if** $s[i] = 1$ **then**
7 {let a be the right child of v; push(*stack*, a);}
8 **else** {let a be the left child of v; push(*stack*, a);}}
9 **else** (*v is marked or $skip(v) = 0$.*)
10 {compare s with the signature s' pointed to by $p(v)$;
11 assume that the first k bits of s agree with s';
12 but s differs from s' in the $(k + 1)$th position;
13 let $w_1 \rightarrow w_2 \ldots \rightarrow w_j \rightarrow v$ be the accessed path;
14 generate a new node u with $skip(u) = k + 1$ and $p(u)$ pointing to s;
15 make u be a child of w_j;
16 **if** $s[k + 1] = 1$ **then**

17 make v be the left child of u;
18 **else** make v be the right child of u;}
19 }
end

In the procedure *insert*, *stack* is a stack structure used to control the tree traversal.

Below we trace the above algorithm against the signature file shown in Fig. 2(a).

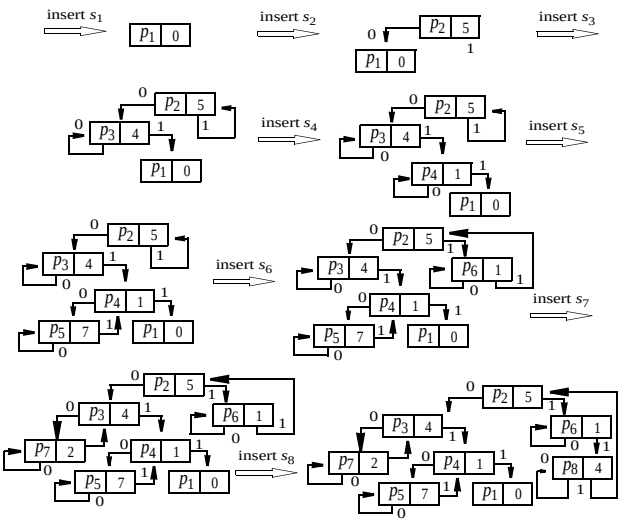

Fig. 3. Sample trace of signature graph generation

Searching of Signature Graphs

In terms of 2.1, the matching of signatures is a kind of 'inexact' matches. That is, for a signature s in S, any bit set to 1 in s_q, the corresponding bit in s is also set to 1, then we say, s matches s_q.

In the following, we first describe how to traverse a signature graph to find a signature in S which may be identical to s_q. Then, we present an algorithm which is able to find all the signatures that may match s_q.

To find a signature in S that may be identical to s_q, we do the following.

Algorithm *exact-matching*(G, s_q)

1. The search begins from the root.

2. Let v be the node encountered. Let $skip(v) = i$. If ith bit of s_q is 1, explore the right child of v; otherwise, explore the left child of v. v is marked.

3. The search ends up when a node v is encountered, which is marked or $skip(v) = 0$. In this case, compare s_q with the signature pointed to by $p(v)$.

In the following, we show the correctness of the Algorithm *exact-matching*( ). To do this, we borrow the concept of *signature identifiers* proposed in [19].

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

Definition 2 (*signature identifier*) Let $S = s_1.s_2 ... .s_n$ denote a signature file. Consider s_i ($1 \le i \le n$). If there exists a sequence: $j_1, ..., j_h$ such that for any $k \ne i$ ($1 \le k \le n$) we have $s_i(j_1, ..., j_h) \ne s_k(j_1, ..., j_h)$, then we say $s_i(j_1, ..., j_h)$ identifies the signature s_i or say $s_i(j_1, ..., j_h)$ is an identifier of s_i w.r.t. S. □

As an example, consider the signature file shown in Fig. 2(a), in which $s_6(1, 5) = (1, 0)(5, 1)$ is an identifier of s_6 since for any $i \ne 6$ we have $s_i(1, 5) \ne s_6(1, 5)$. (For instance, $s_1(1, 5) = (1, 1)(5, 0) \ne s_6(1, 5)$, $s_2(1, 5) = (1, 1)(5, 1) \ne s_6(1, 5)$, and so on. Similarly, $s_5(5, 4, 1, 7) = (5, 0)(4, 1)1, 0)(7, 0)$ is an identifier for s_1 since for any $i \ne 5$ we have $s_i(5, 4, 1, 7) \ne s_1(5, 4, 1, 7)$.

Let $v_1 \rightarrow ... v_{k-1} \rightarrow v_k$ be the path explored. Let $skip(v_i) = j_i$ ($i = 1, ..., k$). Let s the signature pointed to by $p(v_k)$. Denote l_i the label for $v_{i-1} \rightarrow v_i$. Then, we have

$$s(j_2, ..., j_k) = s_q(j_2, ..., j_k) = (j_1, l_1)(j_2, j_2) ... (j_{k-1}, l_{k-1}).$$

But we don't have any other signature such that

$$s'(j_2, ..., j_k) = (j_1, l_1)(j_2, j_2) ... (j_{k-1}, l_{k-1}).$$

Now we discuss how to search a signature graph to model the behavior of a signature file as a filter and to get all the signatures that may match s_q.

Denote $s_q(i)$ the ith position of s_q. During the traversal of a signature graph, the inexact matching can be done as follows:

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

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

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

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

To implement this inexact matching strategy, we search the signature graph in a depth-first manner and maintain a stack structure $stack_p$ to control the graph traversal.

Algorithm *signature-graph-search*

input: a query signature s_q;

output: set of signatures which survive the checking;

1.$Set \leftarrow \varnothing$.

2.Push the root of the signature tree into $stack_p$.

3.If $stack_p$ is not empty, $v \leftarrow$ pop($stack_p$); else return(Set).

4.If v is not a marked node and $skip(v) \ne 0$, $i \leftarrow skip(v)$;

mark v;

If $s_q(i) = 0$, push c_r and c_l into $stack_p$; (where c_r and c_l are v's right and left child, respectively.) otherwise, push only c_r into $stack_p$.

5.Compare s_q with the signature pointed by $p(v)$.

(*$p(v)$ - pointer to a signature*)

If s_q matches, $Set \leftarrow Set \cup \{p(v)\}$.

6.Go to (3).

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

Example 1. Consider the signature file shown in Fig. 2(a) and the signature graph generated in Fig. 3 once again.

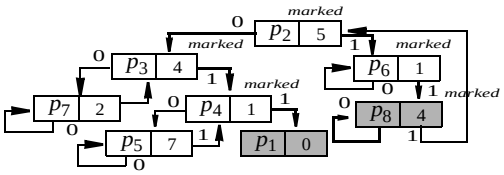

Fig. 4. Signature graph search

Assume s_q = 1011011. Then, only part of the signature tree (marked with thick edges in Fig. 4) will be searched. On reaching a node *v* that is marked or *skip*(*v*) = 0, the signature pointed to by this node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching since only 2 signatures (marked grey) need to be checked while a signature file scanning will check 8 signatures.

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 and checked during the searching, since each bit set in s_q will prohibit half of a subtree from being visited. Compared to the time complexity of the signature file scanning O(*N*), it is a major benefit.

3. BUILDING INDEXES FOR OODBS

In this section, we discuss how to use the technique discussed above to speed up query evaluation in an object-oriented environment.

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 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'*, an 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 (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 [3] and shown in Fig. 5, where an attribute of any class can be viewed as a nested attribute of the root class.

As pointed out in [13], an important element common to OODBS is the view 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'*, *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 consisting of two predicates, one involving the attribute 'Color' and the other involving the nested attribute 'location'.

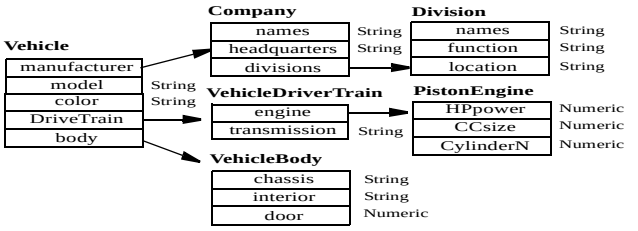

Fig. 5. An example of nested object hierarchy

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 class Vehicle and single out those with red color. Then, the system retrieves the company objects referenced by the red vehicles and checks the manufacturer's divisions' location. Finally, those red vehicles made by a company that has a division located in "Ann Arbor" are returned.

A simple indexing strategy is to construct a signature graph for class Vehicle, by means of which the target objects can be quickly located. Then, the rest part of the database will be searched using referencing links among the objects of different classes. This process can be further improved if a signature file (or signature graph) is established for every class and all the signature files are organized into a hierarchy as shown in Fig. 6.

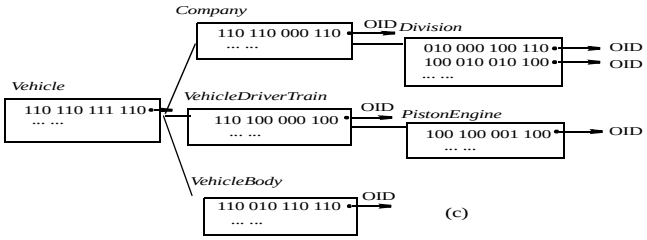

Fig. 6. A signature file hierarchy

Such a hierarchical structure enables us to get rid of non-relevant data as soon as possible by using the so-called *query signature tree* as shown in Fig. 7(b).

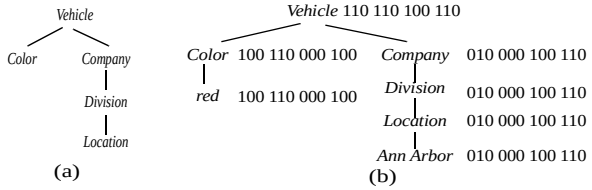

Fig. 7. Query tree and query signature tree

Given a query tree as shown in Fig. 7(a), the signature of a node in a query signature tree can be constructed by superimposing the signatures of its children.

In the following, we give an algorithm for evaluating queries with the above index structures employed. The main idea of it is to use the query signature tree to reduce the search space. For this purpose, two stack structures are needed to control the 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 object identifiers belonging to the same class reached during the class hierarchy traversal.

Algorithm *top-down-hierarchy-retrieval*;
input: an object query *Q*;
output: a set of OIDs whose attribute values 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 OIDs 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 match 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 classes referenced by C; then partition the OID set of the objects referred 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 each of the child nodes of s_q into $stack_q$.
6.Go to (3).
7.For each leaf object, check false drops.

By this strategy, the optimization is achieved by executing step (4). In this step, some objects are filtered using the corresponding signatures in the query signature tree. In step (5), the referred object sets and the signatures of the child nodes of the query signature tree will be put in $stack_c$ and $stack_q$, respectively. (We note that in $stack_c$ each element is a set of object identifiers while in $stack_q$ each element is just a single signature.) In step (7), the checking of false drops will be performed.

Example 2 Continue with our running example. Assume that part of the signature file hierarchy constructed for a database with the schema shown in Fig. 5 is of the form as shown in Fig. 6.

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 Algorithm *top-down-hierarchy-retrieval* will further check the signatures referenced by them in the signature file for *Company* (called *C*-file for short). Assume that the first signature in *C*-file is referenced by the first signature in *V*-file while the second in *C*-file is referenced by the second in *V*-file. We 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 referred by it will not be checked further (see the part marked grey in Fig. 8 for illustration.) It is efficient in comparison with the algorithm *top-down-retrieval* since by it the checking against all *Division* object signatures will be performed.

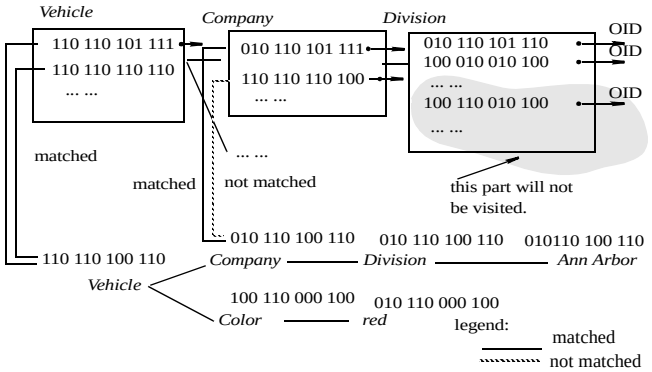

Fig. 8. Illustration of query evaluation

Now we consider another query:

select vehicle
where Vehicle.color = "red"
and Vehicle.*x* = "Ann Arbor",

where *x* represents a path.

For such a query, a signature file hierarchy is not so useful since every possible path starting from class Vehicle has to be considered and the query signature tree constructed for it is not a powerful filter. In this case, the bottom-up strategy should be used to start the search from the leaf nodes in the corresponding class hierarchy. If each leaf node is associated with a signature graph, a lot of futile search can be avoided.

4. CONCLUSION

In this paper, a new index structure for OODBs is proposed. The main idea of this approach is a graph structure, called a signature graph, which is constructed over a signature file of a class to locate possible matching signatures quickly. Together with the concept of signature file hierarchies, this approach improves the efficiency of query evaluation in an object-oriented database significantly.

REFERENCES

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

[2] F. Bancihon, S. Cluet and C. Delobel, A query language for O_2, in Francois Bancihol, Claude Delobel and Paris Kanellakis editors, *Building an Object-Oriented Database System - The Story of O_2*, Margan Kaufmann, 1992, pp. 234-255.

[3] E. Bertino and W. Kim, Indexing Techniques for Queries on Nested Objects, IEEE Transaction on Knowledge and Data Engineering, 1(2):196-214, JUne 1989.

[4] R.G.G. Cattell, *Object Data Management: object-oriented and extended relational database systems*, Addison-Wesley Publishing Company, INC., 1991.

[5] S. Christodoulakis and C. Faloutsos, "Design consideration for a message file server," *IEEE Trans. Software Engineering*, 10(2) (1984) 201-210.

[6] U. Deppisch, S-tree: A Dynamic Balanced Signature Index for Office Retrieval, ACM SIGIR Conf., Sept. 1986, pp. 77-87.

[7] D. Dervos, Y. Manolopulos and P. Linardis, "Comparison of signature file models with superimposed coding," *J. of Information Processing Letters* 65 (1998) 101 - 106.

[8] C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys*, 17(1), 1985, pp. 49-74.

[9] 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.

[10] W. Kim, *Introduction to Object-oriented Databases, The MIT Press, Cambrige, Massachusetts*, 1990.

[11] W. Kim, A Model of Queries for Object-oriented databases, in *Proc. of Int. Conf. on Very Large Data Base*, 1989, pp. 423-432.

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

[13] W. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems, " P*roc. ICIC'92 - 2nd Int. Conf. on Data and Knowledge Engineering: Theory and ApApplication*, Hongkong, Dec. 1992, pp. 616-622.

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

[15] D. Maier and J. Stein, Indexing in an Object-oriented DBMS, in *Proc. Int. Workshop on OODB Systems*, 1986, pp. 171-182.

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

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

[18] E. Ukkonen, "Constructing suffix trees on-line in linear time," *Information Processing 92*, Vol. 1 (ed. J. van Leeuwen), 1992, pp. 484-492.

[19] Y. Chen, Signature Files and Signature Trees, *Information Processing Letters* 82(2002) 231-221, Elsevier Science B.V.