View metadata, citation and similar papers at core.ac.uk brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

Layered Index Structures in Document Database Systems

Yangjun Chen and Karl Aberer

IPSI Institute, GMD GmbH
64293 Darmstadt, Germany
{yangjun, aberer}@darmstadt.gmd.de

Abstract In this paper, a signature file method for indexing document database systems is presented. For this purpose, the concept of *presentative word hierarchy* is introduced, based on which *signature file hierarchies* can be established. Together with the concept of *query signature hierarchy*, it improves significantly the retrieval efficiency of documents stored structurally in object oriented databases.

1 Introduction

With the advent of information highways and digital libraries, the issues of managing and accessing huge hypermedia document bases become important. An interesting way to do so is to integrate database technology into document management and bring the very nature of database systems into this area, such as query processing, efficient management of secondary storage, version and update control, etc. In order to optimize query evaluation in document databases, indexing mechanism should be supported to speed up the retrieval of documents structurally stored.

Most, if not all, indexing techniques in databases are based on tree or network structure. Key values in the index are linked with complex pointer in order to support flexible query styles. On the other hand, in a conventional document system, indexes are built over representative words assigned to documents rather than over key values since they are normally unavailable. Now we accommodate a set of documents in an object oriented database, trying to make explicit the logical structure of the information stored in documents and enable the usage of high level logical query languages. How can we establish indexing mechanism in such a database? More exactly, how the tree or network structure can be imposed on the indexing schema for documents?

To this end, we propose a signature file approach for indexing documents stored structurally in an object oriented database. The motivation is as follows:

1. Tight coupling of document retrieval and database technique should be explored to achieve high performance.
2. Signature files can be feasibly organized into a hierarchical structure and therefore suitable for indexing documents stored structurally. Furthermore, when we accommodate a document into an (aggregation) class hierarchy, all objects of a class is of a measurement of medium size, especially appropriate for constructing signature files (see [12]).
3. Signature files compare favorably to the competitors (e.g., full text scanning [20], inverted files [15], and pat trees [13, 16]) in several respects: lower space overhead, efficient searching and simple maintenance [12].

Signature file techniques have been extensively investigated in information retrieval [11, 12] and database research [17, 24]. However, to the best of our knowledge, it has not been extensively researched in the combination of documents and databases. To implement this integration, the concept of presentative word hierarchy is presented, based on which signature file hierarchies can be constructed for document elements as for objects in object oriented databases. Furthermore, in order to optimize the query evaluation in such an integrated system, we introduce another concept: *query signature hierarchy* which can be used to get rid of non-relevant objects as early as possible during a hierarchy traversal. This technique is quite different from that discussed in [24], in which the field replication technique is used to reduce the number of objects visited during a query evaluation. But these two methods can be combined to develop an efficient method for evaluating queries in object oriented databases (see 6.1).

The rest of this paper is organized as follows. In Section 2, we survey related work. In Section 3, we describe the storage strategy of documents in object oriented databases. In Section 4, we discuss briefly the query language. The indexing technique for a tight coupling of IRS and OODBMS is studied in Section 5. In Section 6, we present our algorithms for evaluating queries with the index hierarchy used and for maintaining signature file hierarchies. Section 7 is devoted to the performance evaluation. Finally, Section 8 is a short conclusion.

2 Related Work

Within the past two decades, a lot of work has been done on document management and document database systems. According to [21], three kinds of methods can be recognized: DBMS-oriented, IRS-oriented and integration approaches.

By the DBMS-oriented approach, the database system is used to manage the structure of documents while the texts are stored separately in an information retrieval system (IRS), as done in [18, 8, 10, 23]. In these systems, the query is expressed in terms of the database schemas constructed for representing document structures and then translated into another form executable in the IRS. Therefore, more powerful expressiveness beyond SQL is in general required in developing the corresponding query languages. However, due to the loose coupling, the indexing technique can not be fully used to optimize the query evaluation.

By the IRS-oriented approach, an IRS is extended to additionally obtain DBMS functionality. In [7], Croft *et al.* describe the integration between the inference net model retrieval system INQUERY and the database system Iris [14]. In this method, Iris is used to store elements of documents, and indexing and retrieval are done by the INQUERY. In [19], another combined DB/IRS-system is introduced, but the database functionality does not yet level up to the state-of-the-art database technology and the integration with structural search is not considered in detail.

By the integration approach, the documents are stored entirely or partially in the databases. In [3, 21, 5], the system HyperStorm is described in detail. In this system, documents are stored structurally or partially structurally in VODAK object oriented system [22] and indexes are established using inverted file technique or by coupling an IRS with the system through defining extra classes for it. In the method proposed in [1, 2], the text is partially loaded into the corresponding O_2 database [4] whenever a query is submitted to the system. But the full-text index is only built for the files storing SGML documents, which can be used to do some optimization for loading texts into databases.

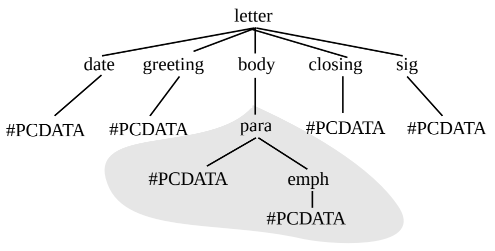
In contrast, in the work presented in this paper, to improve performance, we do a "tight" coupling" of IRS and databases by not only storing the entire documents, but also building indexes physically for them as in traditional databases. Due to the special characteristics of structured documents, the indexes constructed for them (in a database) differentiate a lot from those in a conventional database. On the other hand, they are also different from those in a tradition text retrieval system as the documents are stored structurally rather than in flat files.

3 Document Storage in Databases

In this section, we discuss briefly the mapping of an SGML document into an object oriented schema. To do this, we should first make clear what components an SGML document is composed of and how its structure is represented. For a simple illustration, see a possible DTD for *letter* documents shown in Fig. 1(a).

```
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            - - (#PCDATA | emph)*>
7. <!ELEMENT        emph            - - (#PCDATA)>
8.  <!ATTLIST       emph
        italic      (yes | no)          "yes">
9. <!ENTITY         salute              "Dear">
   ]>
```

(a)



(b)

Fig. 1. DTD and hierarchy structure

An SGML document is defined as having elements and attributes [9]. Elements are always marked up with tags; and an element may be associated with several attributes to identify domain-specific information. For example, element *letter* has two attributes: *filecode* and *secret* (see line 3 of Fig.1(a)). Further, we distinguish between primitive and complex elements. A primitive element contains only data of primitive types such as integer, string and '#PCDATA' (which is more or less comparable to string) while a complex element contains one or more sub-elements which are primitive or complex by themselves. For example, elements *date*, *greeting*, *closing* and *sig* are all primitive (see line 5). But element *letter* is a complex element (see line 2). It contains five sub-elements, of which *body* is itself complex (see line 4). In addition, attention should be paid to the line starting with '<!ENTITY' which introduces a "replacement text" (see line 9). That is, if a string of the form '&salute;' appears in any concrete document belonging to the DTD, this string will be substituted with "Dear" (see [9] for SGML entity definition.) The mapping strategy can be summarized as follows:

1. Corresponding to a primitive element, a primitive class will be generated, which has a primitive attribute for each attribute appearing in 'ATTLIST' and an extra primitive attribute (normally named *text*) for its content. A primitive attribute is an attribute whose value can not be further broken down into sub-values. For example, for the element *emph*, a primitive class of the following form will be generated:

   class *Emph* (*name*: <string>, *italic*: <boolean>, *text*: <#PCDATA>),
   where *text* is defined for the element content.

2. Corresponding to a complex element, a complex class will be created, which has a primitive attribute for each attribute appearing in 'ATTLIST' and a complex attribute for each sub-element. A complex attribute has user-defined classes as its domain. If a complex element contains also non-ATTLIST primitive data besides sub-elements, corresponding primitive attributes will be defined. For example, for the element *letter*, a complex class of the following form will be generated:

   class *Letter* (*name*: <string>, *filecode*: <string>, *secret*: <boolean>, *date*: <OID>, *greeting*: <OID>, *body*: <OID>, *closing*: <OID>, *sig*: <OID>).

3. Constraints may be associated with an attribute to specify its value limitation. For example, for the element *body* (see line 4), we will have a class of the form: (*name*: <string>, *para*: <OID>). A constraint should be associated with attribute *para*, declaring that the number of attribute values of a *Para* object must be equal to or larger than one to reflect the semantics of occurrence indicator "+". The same approach applies to the other SGML occurrence indicators "*" and "?" (see [9] for their definitions.) Furthermore, the default behavior of an attribute should also be declared, such as standard display, read and write methods, as well as default values.

By the above mapping strategy, three points should be noticed:

(i) The handling of SGML connector "|" is deliberately ignored. That is, we define an attribute for each sub-element appearing in the content model of an element, no matter whether they are connected with "|" or not. For example, the class generated for *para* will have two attributes for (#PCDATA | emph). The reason for this is that any concrete element of this type will contain either *#PCDATA* or *emph* and we can always store the corresponding data in one of them. Therefore, an extra data type: *union* as suggested in [1] for accommodating this kind of elements is not necessary.

(ii) We can suppress the structure of a DTD by collapsing part of subtrees. We distinguish between two kinds of structure suppressions: *subtree* and *part-subtree compression*. By the subtree compression, we create only one class for a subtree. For example, instead of generating a class for every element in the subtree rooted at *para* (marked grey in Fig. 1(b)), we define only one class for *para* with an attribute *text* (besides the attributes for those in 'ATTLIST'); the entire subtree will be treated as a flat text and stored as the attribute value of *text*. By the part-subtree compression, we define a primitive attribute for a subtree in the corresponding parent class and no classes are generated for the elements in the subtree at all, including its root. For example, we can define a primitive attribute, in the class for *letter*, for the subtree rooted at *date* element (circled with a brocken line in Fig. 1(b)). This may reduce the number of objects and thus decrease space costs and improve query evaluation, depending on different applications.

(iii) This mapping approach is also suitable for HyTime [9] since the architecture forms of it (such as *clink*, *ilink*, *treeloc*, *dataloc* and so on) are defined in an SGML manner although they play a different roll of semantics. That is, they work as mata-declaration instead of declaration. But at the syntactic level, they are similar to element types of SGML. Therefore, the corresponding classes can be defined for them without difficulty. Of course, special treatment is needed in that case, which is, however, beyond the scope of this paper.

From the above discussion, we see that in a document database, 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 (short) primitive, (long) text, or complex with user-defined classes as their domains. The relationship between a class *C* and some complex attribute domain *C'* of it is called *aggregation relationship*. Using arrows connecting classes to represent this relationship, an aggregation hierarchy can be constructed to show the nested structure of the classes.

4 Expressing Queries

In a traditional document retrieval system, the query processing is mainly based on the tree pattern matching. That is, a structured document can be viewed as a labeled tree. The height of a document tree is equal to the number of the different element types appearing in its DDT plus one. The root represents the top level element in the DDT, its children represent the second level, and so on. The leaves represent the individual (representative) words. The interior nodes are labeled by the level names while the leaves are labeled by the words. To retrieve documents, a partial pattern of the desired document trees are specified. All documents satisfying the pattern are returned.

If T_d and T_p are trees, an *embedding* of T_p in T_d is an injective function *f* from the nodes of T_p to the nodes of T_d. An embedding *f* preserves a binary property *E* between nodes, if for any pair of nodes n_1 and n_2 of T_p, we have $E(n_1, n_2)$ holds in T_p if and only if $E(f(n_1), f(n_2))$ holds in T_d. A typical property is the parent-child relationship. Another example is the labeling preservation. If we number the children of a node from left to right, we can also check the preservation of left-to-right ordering. This concept can be extended to a set of properties. Let *S* be a set of properties to be preserved. An *S-embedding* is an embedding that preserves the properties of *S*. Given a pattern tree T_p, a target tree T_d, and a set *S* of properties, T_p is included in T_d if and only if there exists an *S-embedding* of T_p in T_d.

To implement such a retrieval model, we define the concept of search conditions below.

Definition 1 (*Search condition*) A search conditions in a query is expressed as conjunction of predicates of the form: <*path operator value*>. The *path* is of the form: $p_1.p_2 \ldots .p_n$, where each p_i ($i = 1, 2, \ldots, n\text{-}1$) represents a class name and p_n is an attribute name. In general, an *operator* represents a (set) relation operation $\{\subseteq, \supseteq, =, <, \leq, >, \geq, \neq\}$ and a *value* is a set of individual (representative) words, a boolean expression over individual (representative) words, or another path.

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

select *
from *Letter*, *Letter.Date*, *Letter.Body.Para*
where *Letter.Date.text* $\supseteq$ {"1993"}
    and *Letter.Body.Para.text* $\supseteq$ {"SGML", "database"}

The search condition against the classes *Letter*, *Date* and *Para* consists of two predicates, one involving the nested attribute *text* of *Date* and the other involving the nested attribute *text* of *Para*. (Note that in the above query, if we apply the second kind of structure suppression to the subtree rooted at *date*, the first predicate can be changed into a simpler form: *Letter.date* $\supseteq$ {"1993"}.)

We can extend the above relation set to include more complicated conceptual relations such as *synonym*, *near synonym*, *taxonomy* and so on [6]. We can also use *path variable* to express an unknown path. Further, the length of paths represented by a path variable can be specified using an expression like $P(i)$, where *P* is a variable and *i* is an integer to indicate the length of paths to be evaluated. But for the purpose of this paper, we will not go any deep in these directions so that we can concentrate ourselves on the organization of signature file hierarchies. In addition, we are particularly interested in a kind of restricted search conditions, called *simple search condition* which can be used to optimize query evaluation in our indexing framework discussed in the next section. A simple search condition is a conjunction of predicates of the form: <*path operator value*>. The *path* is of the form: $p_1.p_2 \ldots .p_n$, where each p_i ($i = 1, 2, \ldots, n\text{-}1$) represents a class name and p_n is an attribute name. The *operator* represents a (set) relation operation $\{\supseteq, =\}$ and the *value* is of the form $\{w_1, \ldots, w_m\}$ or $w_1 \vee \ldots \vee w_m$. Such a predicate is known as *simple predicate*.

Obviously, the search condition of the above query is a simple search condition.

The tree structure of the search conditions and the objects storing documents suggest that answering a query on a class involving paths of length more than one requires traversal of a subgraph rooted at the target class. Two approaches to evaluate a query with paths are conceivable: *top-down* and *bottom-up* evaluations. For illustration, let's consider the above example again. To answer the query in a top-down manner, all of the objects in class *Letter* have to be searched 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. In the bottom-up approach, all objects in class *Para* are retrieved to examine if they contain both "SGML" and "databases". Without backward reference support, the OIDs of retrieved *Para* objects have to be kept in a set and search the corresponding objects in class *Body*, whose OIDs should also be stored in another set. Then, the objects in class *Letter* will be retrieved to examine those, whose *date* attribute contains "1993", having the stored *Body* OIDs as attribute values. As a result, the returned objects are letters received in 1993 and at the same time containing strings "SGML" and "databases".

A lot of factors affect the performance of the top-down and the bottom-up approaches: the search conditions specified on the classes and their selectivities, the sizes of the classes along the paths from the target class to the nested attributes, and whether backward references are supported in the system, as well as whether indexes are available for the classes involved in the query. If no indexes and backward references are available, the bottom-up approach will have to scan through all objects located in the classes on the paths. The top-down approach is in general more efficient because only forward references are needed, but the actual cost will depend on the other factors listed above.

Most of the index schemas in the literature are space consuming. Therefore, the number of attributes (classes) to be indexed must be kept to a minimum. Consequently, only queries involving some indexed attributes and classes benefit from these indexing mechanism. To avoid this inefficiency, the system has to build and maintain indexes on every attributes (classes), which is however infeasible due to storage limitation. In this paper, we present a different secondary mechanism, namely, signature file hierarchies, to improve the general performance of document database query evaluation at affordable storage overhead.

5 Representative Words and Signature File Techniques

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. In the following, we discuss the concept of representative words in 5.1. In 5.2, the signature file technique will be described.

5.1 Representative Word Hierarchy

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:

$$weight_{ik} = f_{ik} \cdot signal_k,$$

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 [20]. For example, the presentative 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 dicriminators and cannot possibly be used by themselves to identify document content.

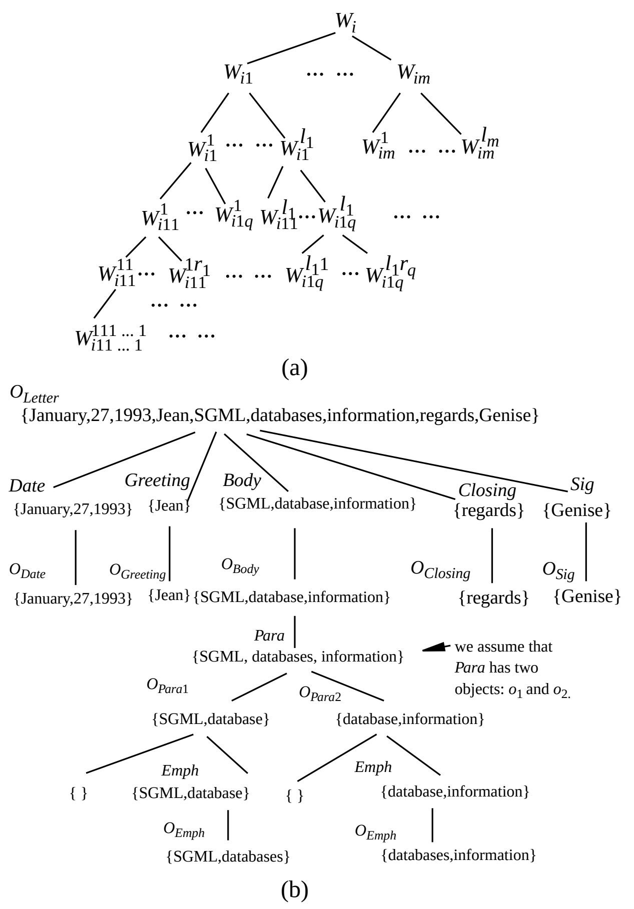
Assume that the class C created for all doc$_i$ (i = 1, ..., n) has m subclasses C_1, ..., C_m in the aggregation hierarchy. Let o be an object of C, in which doc$_i$ is accommodated. Let o_j^k (k = 1, ..., l_j) be the objects of C_j referred by o. Then we can partition W_i into m subsets W_{i1}, ..., W_{im} as follows. We regard each part doc$_{ij}$ of doc$_i$ accommodated in $o_j^1 \cup o_j^2 \ \ldots \cup o_j^{l_j}$ (j = 1, ..., m) as a single document and identify a set of representative words W_{ij} for each doc$_{ij}$ to distinguish it from other doc$_{ip}$'s ($p \neq j$) using the above formula for computing word weights in a similar way, but with the following two rules observed:

(1) If $w \in W_i$ and w appears in $o_j^1 \cup o_j^2 \ \ldots \cup o_j^{l_j}$, but $w \notin W_{ij}$, then add w to W_{ij}: $W_{ij} \leftarrow W_{ij} \cup \{w\}$.

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

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. 1(b)).

Further, we partition W_{ij} into $W_{ij}^1, \ldots, W_{ij}^{l_j}$ by taking each part of the document accommodated in an object o_j^k as a single document and using the above formula again to compute weights. Similarly, rules (1) and (2) should be followed. Now we consider W_{ij}^k (k = 1, ..., l_j). For each of them, the entire process stated above can be applied to do a deeper partition. For illustration, assuming C_j has subclasses C_{j1}, ..., C_{jq}, then we can further partition W_{ij}^k into $W_{ij1}^k, \ldots, W_{ijq}^k$. If o_{js}^{kt} ($s \in \{1, ..., q\}$, t = 1, ..., r_s) are the objects of C_{js} referred by some o_j^k, then W_{ijs}^k can be further partitioned into $W_{ijs}^{k1}, \ldots, W_{ijs}^{kr_s}$. We illustrate this process as shown in Fig. 2(a). Fig. 2(b) is a possible hierarchy for our running example.



W_i
W_{i1} ... ... W_{im}
W_{i1}^1 ... ... $W_{i1}^{l_1}$
W_{im}^1 ... ... $W_{im}^{l_m}$
W_{i11}^1 ... W_{i1q}^1 $W_{i11}^{l_1}$...$W_{i1q}^{l_1}$ ... ...
W_{i11}^{11}... $W_{i11}^{1r_1}$ ... ... $W_{i1q}^{l_11}$ ... $W_{i1q}^{l_1r_q}$
... ...
$W_{i11...1}^{111...1}$ ... ...
(a)
O_{Letter}
{January,27,1993,Jean,SGML,databases,information,regards,Genise}
Date
Greeting
Body
Closing
Sig
{January,27,1993}
{Jean}
{SGML,database,information}
{regards}
{Genise}
O_{Date}
$O_{Greeting}$
O_{Body}
$O_{Closing}$
O_{Sig}
Para
{SGML, databases, information}
we assume that Para has two objects: o_1 and o_2.
O_{Para1}
O_{Para2}
{SGML,database}
{database,information}
{ }
Emph
{SGML,database}
{database,information}
O_{Emph}
{SGML,databases}
{databases,information}
(b)

Fig. 2. 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.

5.2 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 [11]). 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. 3 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	$\vee$	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. 3. 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. 3: (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 simple file structure than inverted indexes. They can be particularly good for medium size data sets [12]. Notice that, in a document database, all the objects of a class is just of such a measurement of size.

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

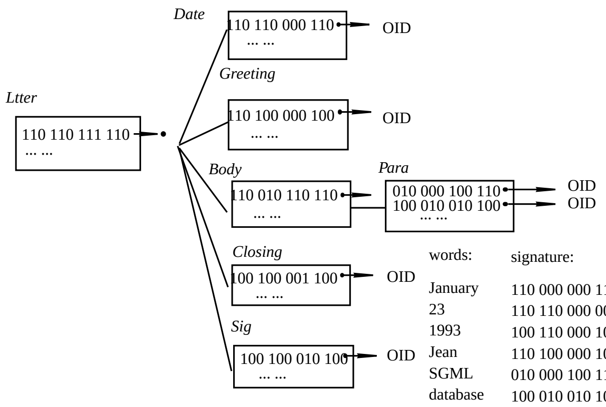
(i) For every representative word w, there exists a pair <*wsig*, *pos*>, where *wsig* is the signature of w and *pos* is w's position within the text stored in the corresponding object. Such a pair may be maintained in the corresponding object if the number of representative words is rather small in comparison with the text stored in it; otherwise, it is not worth while maintaining such entries.

(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. 4, which is constructed in terms of the representative word hierarchy shown in Fig. 2(b).



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. 4. Signature file hierarchy

Note that such a hierarchy can be refined by further partitioning each signature file S into several segments Seg_1, ..., Seg_m such that each Seg_i contains only the signatures of all those objects referred by the same object belonging to the parent class. In this way, the scanning within a signature file can be done more quickly.

6 Document Retrieval and Signature Maintenance

In this section, we first discuss the document retrieval based on the signature file hierarchies described in Section 5 and introduce a new concept: *query signature hierarchy* to optimize the query evaluation. Then, the algorithm for maintaining signature file hierarchies will be outlined.

6.1 Retrieval

As we mentioned before, there are two methods to evaluate a query: top-down and bottom-up approaches. The top-down approach is to retrieve all of the objects 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 with s_q, traverse the path to verify the nested attribute. In the following, a trivial algorithm for top-down retrieval is first described. Then, a refined version of it is discussed 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 4. First, we construct $s_q = s_{1993} \vee s_{SGML} \vee s_{database}$ = 100 110 000 100 $\vee$ 010 000 100 110 $\vee$ 100 010 010 100 = 110 110 010 110. It matches the entry in the signature file of *Letter* shown in Fig. 4. Then, the corresponding OID will be put in S. In step (3), the paths from *Letter* to *Date* and from *Letter* to *Para* will be traversed and the objects of *Date* and *Para* will be checked to eliminate false drops. The object signature of *Date* shown in Fig. 4 matches s_{1993}. Then, the text stored in this object will be scanned to see whether it contains really this word. If the entry <s_{1993}, pos_{1993}> is stored, the corresponding word can be directly located and thus the check can be performed immediately. Otherwise, the text has to be scanned in a normal way. The same process applies to the objects of *Para*.

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 develop another strategy, by 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 2 (*Query hierarchy*) Let $pred_1 \wedge pred_2 \wedge \ldots \wedge pred_k$ be the simple part of the search condition in query Q, where each $pred_i$ is a simple predicate. Then, all the paths appearing in the simple part of the search condition constitutes a (partial) query hierarchy, denoted Q_h.

Definition 3 (*Query signature hierarchy*) Let $p_1.p_2 \ldots .p_n$ be a path in a (partial) query hierarchy Q_h (from the root to some leaf). Let <$p_i \ldots .p_n$ *operator value*> be a simple predicate appearing in the simple part of the search condition of Q. If *value* is of the form: $\{w_1, \ldots, w_m\}$, then p_n's signature is equal to $s_{w_1} \vee \ldots \vee s_{w_m}$. If *value* is of the form: $w_1 \vee \ldots \vee w_m$, then p_n's signature is equal to $s_{w_1} \wedge \ldots \wedge s_{w_m}$. The signature of a non-leaf node in Q_h can be obtained by superimposing the signatures of its child nodes. The query signature hierarchy is denoted $Q_{(s,h)}$.

For example, for our exemplar query, the query hierarchy and the query signature hierarchy are as shown in Fig. 5(a) and (b), respectively.

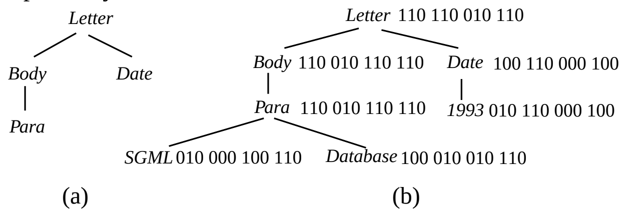


Fig. 5. Query hierarchy and query signature hierarchy

In the following, we give our algorithm for retrieving documents with the signature file hierarchy used. The main idea of it is to use the query signature hierarchy to reduce the search space. For this purpose, two stack structures are needed to control the depth-first traversal of hierarchies: $stack_q$ for $Q_{(s,h)}$ and $stack_d$ for the document hierarchy. In $stack_q$, each element is a signature while in $stack_d$ each element is a set of objects belonging to the same class reached during this document 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 $Q_{(s,h)}$ for the query $Q$.
   2. Push the root signature of $Q_{(s,h)}$ into $stack_q$; push the set of object OID of the target class into $stack_d$.
   3. If $stack_q$ is not empty, $s_q \leftarrow$ pop $stack_q$; else go to (7).
   4. $S \leftarrow$ pop $stack_d$; 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 $C_1, \ldots, C_k$ be the subclasses of $C$; then partition the OID set of the objects referred by the objects of $S$ into $S_1, \ldots, S_k$ such that $S_i$ belongs to $C_i$; push $S_1, \ldots, S_k$ into $stack_d$; push 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 signature in the query signature hierarchy. In step (5), the referred objects and the signatures of the child nodes of the query signature hierarchy will be put in $stack_q$ and $stack_d$, respectively.

Example 2 Continue with our running example. But we assume that a letter element may contain more than one bodies to show why "*top-down-hierarchy-retrieval*" works efficiently. A possible signature file hierarchy may be of the form as shown in Fig. 6.

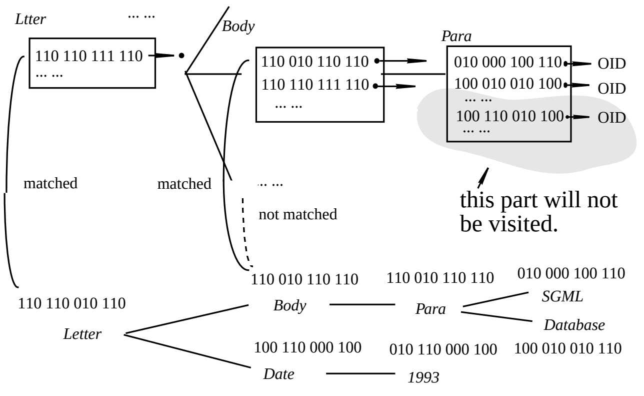


Fig. 6. Illustration of query evaluation

Using "*top-down-hierarchy-retrieval*" to evaluate the query, the second *Body* object will be filtered since its signature does not match the signature associated with *Body* in the query signature hierarchy shown in Fig. 5(b) (as indicated by the dashed line in Fig. 6.) Thus, all those *Para* objects referred by it will not be checked further (see the part marked grey in Fig. 6.) It is optimal compared to "*top-down-retrieval*" since by "*top-down-retrieval*" the checking against all *Para* objects have to be performed.

This method and the method proposed in [24] (known as Yong's method hereafter) can be combined to develop a more efficient strategy. By Yong's method, the signatures of the referred objects are stored in the referring ones. Then, the predicate checking can be performed against their signatures before they are accessed. In this way, a lot of I/O can be saved.

We illustrate the evaluation processes of the two methods as shown in Fig. 7.

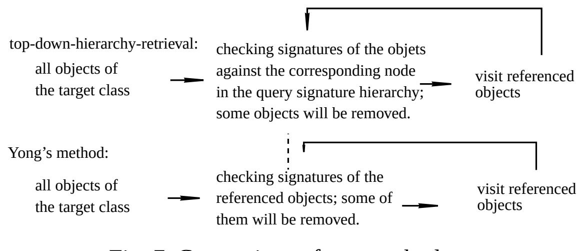

top-down-hierarchy-retrieval:
all objects of the target class
checking signatures of the objets against the corresponding node in the query signature hierarchy; some objects will be removed.
visit referenced objects
Yong's method:
all objects of the target class
checking signatures of the referenced objects; some of them will be removed.
visit referenced objects

Fig. 7. Comparison of two methods

Form this figure, we can see that the elimination of the non-relevant objects by our method happens just one step earlier than Yong's method. That is, the first checking by our method is done for the objects of the target class, whereas the first checking by Yong's method is made for the objects referred by the target class through storing redundantly the signatures of the referred objects. Then, in a next step, by the both methods, the referred objects will be visited. Afterwards, the second elimination happens. (We note that the referred objects in the previous step become now the referring ones.) For this time, some of the referring objects will be removed by our method while by Yong's method some of the referred objects will be discarded. Each of the two processes repeats iteratively.

Clearly, these two methods can be integrated harmoniously. That is, in each step we check not only the referring objects against the query signature hierarchy but also the referred objects if their signatures are available. This can be done by changing step (5) of "*top-down-hierarchy-retrieval*" a bit:

5. Let C be the class, to which the objects of S belong; let $C_1, ..., C_k$ be the subclasses of C; check the predicates using signatures of the referred objects; let S_o be the set of the OIDs of those objects who satisfy the checking; then partition S_o into $S_1, ..., S_k$ such that S_i belongs to C_i; push $S_1, ..., S_k$ into $stack_d$; push the child nodes of s_q into $stack_q$.

6.2 Signature Maintenance

Whenever the text of an object is modified, its signature needs possibly to be changed accordingly. To do this, we have to redetermine the representative words for the object, which can be done by regarding the corresponding document as a new one and invoking the entire process for calculating representative words as discussed in Section 5. Obviously, it is not efficient. Another approach is to do the modification "locally". We scan the new text of the object to see whether any element of the representative word set W is lost from the text due to the modification and any new word w which can be used as representative word occurs. (This can be done with the help of human beings.) In the former case, the lost representative word will be removed from W. In the latter case, w will be added to W. Then, the new object signature will be calculated and propagated to all of its ancestor objects. The following algorithm works in a two-step fashion and updates signatures in terms of the above local modification strategy.

Algorithm *signature-maintenance*;
step 1 - *signature-modification*: (input: the OID of an object o; output: o's new signature)

1. Use the OID to retrieve o.
2. Check the text of o to find whether any representative word is lost from the text and any new word which can be used as representative word occurs.
3. If the cases mentioned in (2) happen, change the representative word set of the object and construct a new object signature in terms of it.

step 2 - *signature-propagation*: (input: the OID of an object o which has been updated; output: changing the signatures of o's ancestor objects)

1. Use the old signature of o to retrieve o's parent objects o_p and change o_p's signature.
2. Recursively apply *signature-propagation* to o_p.

For example, if the text stored in one of *Para* objects o is changed, the set of its representative words may also be changed. In this case, its signature should be recomputed, which leads to the update of some object signatures of class *Body* if they refer o through aggregation arrows. Then, some *Letter* object signatures have to be changed due to the newly updated object signatures of class *Body.*

7. Performance

For the comparison purpose, we consider a linear setting as done in [24]. That is, the query evaluation will be performed along a class path as shown in Fig. 8.

$$\begin{tikzcd} C_1 \arrow[r, "a_1"] & C_2 \arrow[r, "a_2"] & C_3 \arrow[r, "a_3"] & \cdots\cdots & {} \arrow[r, "a_{n-1}"] & C_n \end{tikzcd}$$

Fig. 8. Class path

In the 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 will be executed for evaluating the query. The following is the parameters and assumptions used for the performance evaluation.

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 the checking against the corresponding signature in the query signature hierarchy.
d:	average out-degree of objects (i.e., the average number of referred objects of an object).
N_i:	Number of objects in class C_i.

Using these parameters, the numbers of objects visited for evaluating a query using different methods can be estimated as below.

nested loop top down retrieval (NLTR)

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

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

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

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 referred objects, for each class C_i, v_i' can be computed as follows:

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

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

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

combining Yong's method with top-down-hierarchy-retrieval (THR)

By combining Yong's method with *top-down-hierarchy-retrieval*, more benefits can be obtained. Let v_i" be the number of objects in class C_i visited by THR. 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

$$v'' = \sum_{=1}^{n} v_i'' =$$

$v_1 \cdot P_q \cdot (1 + d \cdot \qquad\qquad\qquad\qquad\qquad)$.

We compare the three methods above using two simple abstract data sets. By the first data set, each object has only one sub-object (i.e, d =1; see Fig. 9(a)). By the second, each object refers two sub-objects (i.e, d =2; see Fig. 9(b)).

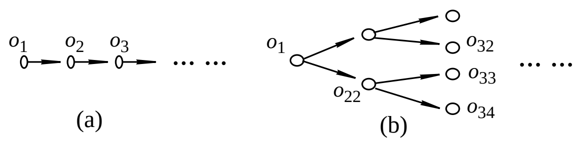


Fig. 9. Two data sets

Fig. 10(a) and (b) show the comparison results for the two data sets distributed in three classes C_1, C_2 and C_3, respectively. By the performance analysis, we assume that P_i = 0.1 (i = 1, 2, 3), P_f = 0.01 and P_q = 0.5. Contrary to [24], the visited objects of class C_1 is counted since 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 of accessed objects in total.

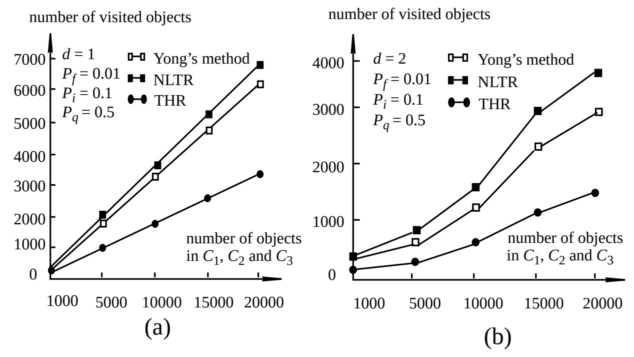


Fig. 10. Comparison results

The figure shows that we can achieve high performance by combining Yong's method and "*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 make fewer objects accessed.

8. Conclusion

In this paper, a signature file method is developed to index the documents stored in object oriented databases. First, the concept of presentative word hierarchy has been introduced, which can be constructed based on the canonical information theory [20]. In terms of a presentative word hierarchy, a signature file hierarchy can be established by hashing each representative word into a bit string and then superimposing them into object signatures. By means of organizing the signatures of the objects belong to a class into a signature file, we obtain a signature file hierarchy, corresponding to the aggregation class hierarchy for the documents stored. Such a hierarchy is useful to speed up the document retrieval by filtering as many non-relevant objects as possible if the query signature hierarchy is available. In addition, the method proposed in [24] can be integrated into our strategy to get a more efficient algorithm for the query evaluation in object oriented databases.

Reference

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

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

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

[5] 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.

[6] D.A. Cruse, "Lexical Semantics," Cambridge University Press, 1986.

[7] 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.

[8] 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.

[9] S.J. DeRose and D.D. Durand, "*Making Hypermedia Work: A User's Guide to HyTime*," Kluwer Academic Publishers, London, 1994.

[10] S.C. Deerwester, K. Waclena and M. Lamar, "A Textual Object Management System," *Proc. of 15th Ann. Int. SIGIR*, Denmark, 1992.

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

[12] 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.

[13] 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.

[14] Hewlett-Packard, *OpenODB Reference Manual B3185A*, 1992.

[15] 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.

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

[17] 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 Application*, Hongkong, Dec. 1992, pp. 616-622.

[18] I.A. Macleod, "Storage and Retrieval of Structured Documents," *J. of Information Processing & Management*, Vol. 26, No. 2, 1990, pp. 197-208.

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

[20] G. Salton and M.J. McGill, "*Introduction to Modern Information Retrieval*," McGray-Hill Int. Book Com., Hamburg, 1983.

[21] 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.

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

[23] 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.

[24] 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.