Short Cuts for Traversals in Object-Oriented Database Systems

Wang-Chien Lee and Dik Lun Lee
Department of Computer and Information Science, Ohio State University
Columbus, Ohio 43210-1277, USA

{wlee, dlee}@cis.ohio-state.edu
FAX: 614-292-2911

Abstract

We introduce a new class of indexing organizations for nested query processing in object-oriented database systems. Our methods maintain direct links connecting objects in two separate classes for fast object traversals. We develop cost models for different implementations of the indexing organizations and compare their performance. The clustering index has been shown to be the best in terms of storage and retrieval costs. However, the non-clustering and semi-clustering indexes are also very efficient when compared to traditional object traversal mechanisms.

1 Introduction

Nested query processing in object-oriented database systems often requires *traversal* among objects. In this paper, we investigate the issues with object retrieval and review several solutions proposed in the literature. We also develop a new class of secondary organizations for nested query processing.

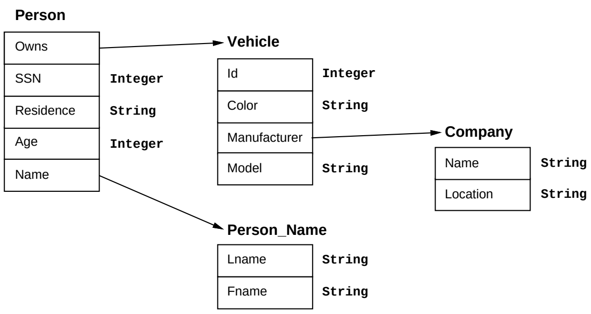

Figure 1: Aggregation hierarchy.

Figure 1 illustrates related concepts and terminologies for OODBSs within the context of this paper. The figure shows the class definitions for `Person`, `Vehicle`, `Person_Name`, and `Company`. These four classes form an *aggregation hierarchy*. The class `Person` has three *primitive attributes*, `SSN`, `Residence` and `Age`, and two *complex attributes*, `Owns` and `Name`. The domain classes of the attributes `Owns` and `Name` are `Vehicle` and `Person_Name`, respectively. The class `Vehicle` is defined by three primitive attributes, `Id`, `Color`, and `Model`, and a complex attribute `Manufacturer`, which has `Company` as its domain. `Company` and `Person_Name` each consists of two primitive attributes.

Every object in the database is identified by a unique *object identifier* (*OID*). By storing the OID of an object as an attribute value of another object, we establish an aggregation/association relationship between these two 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 as the *parent* object of O. Objects are nested according to the aggregation hierarchy.

Queries involving nested attributes or nested objects are called *nested queries*. In a nested query, the class from which objects are retrieved is called a *target class* and the classes with attributes involved in the query predicates are called *predicate classes*.

Nested queries may be classified by the relative positions of the target and predicate classes in the aggregation hierarchy:

- TP Queries: this type of queries have a set of predicates on (nested) attributes of the target class. For example,

  Q1: retrieve all persons who are 50 years old and own vehicles manufactured by Ford.

- PT Queries: this type of queries retrieves the nested objects of a given set of objects, which might be specified by predicates. For example,

  Q2: retrieve the manufacturers of the cars owned by persons at the age of 50.

- MX Queries: in this type of queries, the target class is between two predicate classes. For example,

  Q3: retrieve red cars made by GM and owned by persons at the age of 50.

The rest of the paper is organized as follows. In Section 2, we introduce the traversal approaches for nested object retrieval. Section 3 is a review of the indexing organizations proposed. Addressing the problems with existing indexing methods, we develop in Section 4 a new idea for improving the performance of query evaluation and a class of indexing organizations. In Section 5, we derive cost models for different organizations and compare their performance.

2 Traversals for Object Retrieval

Traversal is the most fundamental mechanism for nested object retrieval. There are two basic traversal methods:

- The *forward traversal* approach traverses the objects starting from an ancestor class to a nested class. Since the OID in a parent object leads directly to a child object, the child object can be accessed quickly.
- The *backward traversal* approach traverses up the aggregation hierarchy. Since a child object is not assumed to carry the OID of its parent object, we must compare the OID of the child object against the corresponding complex attribute in the parent class in order to find the parent object(s) of the child object. This is similar to a relational join when we have more than one child object to start with.

To answer Q1 by forward traversal, we retrieve every object in the class `Person`, screen out the persons who are not 50 years old, and, for each qualified `Person` object, retrieve the `Vehicle` objects and their nested `Company` objects to check if the manufacturers' names are Ford. Using backward traversal, the objects in the class `Company` are retrieved to examine if their names are Ford. The OID's of the Ford companies are maintained in a set S. Then the vehicle objects in class `Vehicle` are examined to identify those vehicles made by the companies in S. The qualified vehicle objects are collected in a set S'. Finally, the `Person` objects are retrieved to see if they are 50 years old and own a vehicle in S'. The efficiency of these two methods depends on the selectivity of the two predicates and the number of objects connected directly or indirectly to the qualified objects in `Person` and `Company` after the predicates are tested.

Since query Q2 retrieves the nested complex attributes of a specific collection of objects, forward traversal is more efficient than backward traversal. In forward traversal, we first retrieve the objects in `Person` and check their ages. For those `Person` objects with `Age` 50, we traverse along the path `Person.Vehicle.Company` to retrieve the names of their car makers. However, the backward traversal approach is cumbersome for this query, since it requires the objects in `Company` to join to the objects in `Vehicle`, which in turn join to the `Person` objects before the predicate on `Age` can be evaluated. This is equivalent to performing a sequence of join *before* applying a selection operation.

From the discussion above, we observe that both forward and backward traversals spend a significant amount of time on accessing intermediate objects connecting two objects. In the next section, we review some of the indexing methods proposed in the literature for speeding up object traversal.

3 Indexing Methods for Object Retrieval

To avoid retrieving intermediate objects between the target and predicate classes, indexes are used to create implicit reverse links from a nested attribute to the target class.

Multi-index: The *multi-index* method [1,4] maintains multiple indexes, one for each link of the path from the indexed nested attribute to target class. An advantage of the multi-index method is that it reduces redundancy when paths with common subpaths are indexed. Also, the indexes are easy to maintain for update operations. However, an index is required for every link on the path and several index scans are necessary to answer a nested query.

Nested Index: Unlike the multi-index method, the nested index and the path index consist of only one index [1]. For a given path, the nested index maps the values of a specific nested attribute to the objects in the root class of the path. The nested index has a simple organization and fast object retrieval speed. However, maintenance of the index is complex, since it requires backward traversal on the index.

Path Index: The path index maps the values of the nested attributes to lists of objects; each of the objects in the list belongs to a class in the given path. The linkages of the objects constitute the physical paths in the database. The range of a path index mapping could be any of the classes on the path. Thus, the scope of a path index is broadly extended. However, extra time are needed for index scanning, because the OIDs of all of the objects on the path are stored in the leaf nodes of the index.

A common drawback of all of these indexing organizations is that the reverse links they create don't support PT queries. Even though forward traversal is in general faster than backward traversal, its speed is still not satisfactory. Therefore, an indexing organization supporting forward traversal is desirable.

4 Direct Links and Index Organizations

To reduce accesses to intermediate classes, direct links can be created for classes which are frequently accessed together. In this section, we formalize the notion of direct link.

4.1 Definition of Direct Links

Given a path P consisting of $C_1C_2...C_n$ in the aggregation hierarchy, the direct links DL on P is defined as follows.

$$\begin{aligned} DL = \; & \{(\theta_1, \theta_n) | OID(O_1) = \theta_1, OID(O_n) = \theta_n; \\ & O_1 \in C_1, O_n \in C_n; O_1 \rightsquigarrow O_n\}. \end{aligned}$$

$O_1 \rightsquigarrow O_n$ means that O_n is nested in O_1. Since C_1 is an ancestor class of C_n, we call C_1 an *a-class* and C_n a *c-class* of the direct links. Each pair of the OIDs in DL is a direct link. The direct links in DL may be sorted by a-class OIDs or by c-class OIDs. Suppose direct links are built between `Person` and `Company`. Since `Person` is an ancestor class of `Company`, `Person` is the a-class and

`Company` is the *c*-class of the direct links organization. To create the direct links, we first traverse the objects from `Person` to `Company` to collect the OIDs of the connected objects from these two classes. In the example below, ⟨`Person`[i], `Company`[j]⟩ represents a link between `Person`[i] and `Company`[j].

Person[1]	Company[4]
Person[3]	Company[2]
Person[5]	Company[2]
Person[8]	Company[2]
Person[2]	Company[7]
Person[6]	Company[7]

The direct links provide short cuts between the `Person` and `Company` classes. However, nested queries usually are coupled with predicates for associative search. Take the query "retrieve persons who have GM cars" as an example, we have to search the `Company` class based on "`Name`= GM". Indexing and signature file methods [2,3] are two of the most important techniques for associative search. We combine indexing techniques with direct links to build new indexing organizations.

An index may be created on one of the attributes from the *a*-class or *c*-class. The attribute is called the *key attribute* of the index. The *a*-class and the *c*-class of the direct links are the *range classes* of the index.

4.2 Forward and Backward Indexes

The *forward index* maps from an attribute in the *a*-class to a list of direct links. Given a list of direct links, DL, and an attribute $\mathcal{A}$ in the *a*-class of DL, a forward index may be defined as follows.

$$FI = \{(\kappa, \theta_a, \theta_c) | (\theta_a, \theta_c) \in DL; \mathcal{A}(\theta_a) = \kappa\}.$$

The following is an example of the forward index for the path `Person.Vehicle.Company`; `Person.Age` is the key attribute.

24	Person[1]	Company[4]
33	Person[3]	Company[2]
42	Person[5]	Company[2]
50	Person[8]	Company[3]
50	Person[2]	Company[7]
60	Person[6]	Company[7]

As its name implies, the forward index may be used for forward traversal. Therefore, PT queries benefit the most from this organization. We use Q2 as an example to illustrate the query processing with the forward index. We first scan the index to locate the direct links corresponding to "`Person.Age`=50". Through the direct links collected, the OIDs of the `Company` objects are returned. Similarly, a query such as "retrieve persons at the age of 50" may be directly answered from the index without accessing to the objects in the database.

On the other hand, a backward index is created based on one of the attributes in the *c*-class. Given a list of direct links, DL, and an attribute $\mathcal{A}$ of the *c*-class of DL, a backward index may be defined as follows:

$$BI = \{(\kappa, \theta_c, \theta_a) | (\theta_a, \theta_c) \in DL; \mathcal{A}(\theta_c) = \kappa\}.$$

The following is an example of the backward index for the path `Person.Vehicle.Company`:

BMW	Company[4]	Person[1]
GM	Company[2]	Person[3]
GM	Company[2]	Person[5]
GM	Company[3]	Person[8]
Ford	Company[7]	Person[2]
Ford	Company[7]	Person[6]

The backward index supports backward traversal. It may be used to answer TP queries. For instance, to evaluate the query "retrieve persons who have cars made by Ford", we scan the backward index organization to locate the direct links corresponding to "`Company.Name`= Ford" and return the `Person` OIDs in the qualified links.

4.3 Clustering and Non-clustering Indexes

One of our requirements in designing the structure of direct links is to support multiple access methods. In other words, we must be able to couple the direct links structure with arbitrary indexes built on the attributes of the *a*-class or *c*-class of the direct links organization. Thus, an index organization will consist of two levels: the upper level is the attribute indexes, which are tree-structured associative searching mechanisms (e.g., B^+-trees), and the lower level is the direct links file, which is accessed through the attribute indexes.

To provide fast access to the direct links, the direct links file may be clustered on a given attribute. However, only one attribute can be selected for clustering the direct links file. We call the attribute a *clustering attribute*. The index organization based on a clustering attribute is called a *clustering index*.

For every cluster in the direct links file, the direct links are sorted by the OIDs in the class containing the clustering attribute. Thus, non-clustering indexes may be further divided into two categories:

- *Semi-clustering index*: the indexed attribute is in the same class with the clustering attribute.
- *Non-clustering index*: the indexed attribute is not in the same class as the clustering attribute.

Take the direct links for `Person.Vehicle.Company` as an example, if the direct links file is clustered on `Company.Name`, then `Company.Location` is a semi-clustering attribute and `Person.Age` is a non-clustering attribute.

After searching a clustering attribute index, the direct links corresponding to the searched attribute value can be sequentially read from the direct links file. For non-clustering attributes, the corresponding direct links are distributed so that random reads from the direct links file are necessary. However, semi-clustering indexes may have better performance than non-clustering indexes, because the corresponding direct links are more likely to be stored together.

4.4 Structures of Attribute Indexes

The clustering, semi-clustering and non-clustering indexes have similar data structures. We choose the B$^+$-tree for the implementation of the attribute indexes of these index organizations.

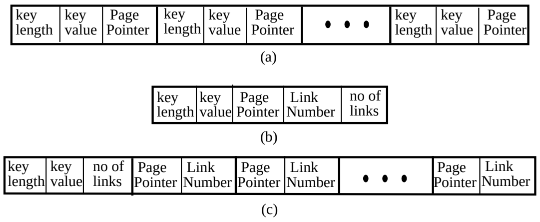

Figure 2: Structures of attribute indexes

Figure 2 illustrates the data structures used for the attribute indexes. Figure 2(a) shows a nonleaf node page for the clustering and non-clustering attribute indexes. Figure 2(b) is a leaf node record for a clustering index, and Figure 2(c) is a leaf node record for a non-clustering index.

The leaf node record of a clustering index is smaller than that of a non-clustering index. A leaf record keeps the location, specified as a page pointer and a link number, and the number of direct links corresponding to a specific key value. Since the direct links file is clustered by its key values, the collection of direct links may be sequentially read.

For non-clustering indexes, the leaf node records have to keep track of the locations of all corresponding direct links. Due to the way direct links are clustered, a semi-clustering index is likely to have some direct links sequentially stored in the direct links file. To simplify our design, however, we use the same leaf-record structure for both semi-clustering indexes and non-clustering indexes.

5 Analytical Cost Model

In this section, we build a forward index for the path $C_1C_2...C_n$ using different implementations, namely the clustering, semi-clustering and non-clustering indexes. We develop analytical cost models for each implementation and compare their performance in terms of storage overhead and retrieval cost.

5.1 The Parameters of the Cost Model

Given a path, $C_1C_2...C_n$, the parameters of the cost models for the three index implementations are listed below.

Table 1: Parameters of the cost models.

P	=	4096	ll	=	2
$OIDL$	=	8	kl	=	1
pp	=	4	kv	=	8
cl	=	4	f	=	218

C_a: a-class for the direct links, $C_a = C_1$.
C_c: c-class for the direct links, $C_c = C_n$.
N_i: the number of objects in class $C_i, 1 \leq i \leq n$.
S_i: the average size of an object in class C_i.
A_i: the complex attribute of C_i used to connect C_i to C_{i+1}, $1 \leq i \leq n-1$.
D_i: the number of distinct values for complex attribute A_i.
Br_i: the average number of branches from C_i to C_{i+1}. ($Br_i =$ (number of OIDs in$A_i)/N_i$.)
Sr_i: the ratio of shared references from C_i to C_{i+1}. ($Sr_i =$ (number of OIDs in$A_i)/N_{i+1}$.)
$\mathcal{A}$: the attribute of C_a selected for indexing.
U: the number of distinct values in $\mathcal{A}$.
q: the ratio of shared attribute values between objects in class C_a and values for attribute $\mathcal{A}$. ($q = N_a/U$.)
$OIDL$: the length of an object identifier.
P: the page size.
pp: the length of a page pointer.
ll: the length of a link number.
f: average fanout from a nonleaf node in the attribute index.
kv: average length of a key value in $\mathcal{A}$.
kl: the length of the key length field in the attribute index.
cl: the length of the counter fields (i.e., no. of links).
h: the number of levels of non-leaf nodes in the attribute index.

Performance is measured by the number of I/O accesses. we use *page* to estimate the storage overhead and the performance cost. All lengths and sizes used above are in *bytes*. Table 1 lists the parameter values used in the analysis.

To simplify the cost models, we assume that:

1. There are no partial instantiation, which implies $D_i = N_{i+1}$.

2. All key values have the same length.

3. Attribute values are uniformly distributed among the objects of the class containing the attribute.

5.2 Storage Overhead

Direct Links: In the direct links file, each direct link consists of two OIDs. Therefore, the size of a direct link is $(2 \cdot OIDL)$ bytes. The number of direct links in the file can be determined by:

$$NDL = N_1 \cdot Br_1 Br_2 ... Br_{i-1} = Sr_1 Sr_2 ... Sr_{i-1} \cdot N_n.$$

The total number of pages needed for the direct links file is:

$$SDL = \lceil NDL \cdot (2 \cdot OIDL)/P \rceil.$$

Nonleaf Nodes: All three implementations have the same structure for nonleaf node records. The number of nonleaf pages can be derived from:

$$NLP_i = \lceil LO_i/f \rceil + \lceil \lceil LO_i/f \rceil / f \rceil + ... + 1,$$

where $LO_i = min(U, LP_i)$ and LP_i is the number of leaf node pages for implementation i.

Clustering Index: The length of a leaf node record for the clustering index implementation is:

$$XP_{ci} = kl + kv + pp + ll + cl.$$

Therefore, the number of leaf node pages needed is:

$$LP_{ci} = \lceil U / \lfloor P / XP_{ci} \rfloor \rceil.$$

The storage overhead for implementing the forward index as a clustering index is:

$$SCI = LP_{ci} + NLP_{ci} + SDL.$$

Non-clustering Indexes: The average number of direct links associated with an object in C_a is:

$$BR = Br_1 Br_2 ... Br_{n-1}.$$

The average number of objects in C_a corresponding to a key value in attribute $\mathcal{A}$ is q. Consequently, the average length of a leaf node record for a non-clustering index is:

$$XP_{ni} = kl + kv + cl + q \cdot BR \cdot (pp + ll).$$

Therefore, the number of leaf node pages needed is:

$$LP_{ni} = \begin{cases} \lceil U / \lfloor P / XP_{ni} \rfloor \rceil, & \text{if} \quad XP_{ni} \leq P \\ U \lceil XP_{ni} / P \rceil, & \text{if} \quad XP_{ni} > P. \end{cases}$$

The storage overhead for implementing the forward index as a non-clustering index is:

$$SNI = LP_{ni} + NLP_{ni} + SDL.$$

Comparison

We use the above formulae to compare the storage overheads of a forward index when implemented as a clustering or non-clustering index. In our comparison, we use a path of four class, $C_1 C_2 C_3 C_4$. An attribute of C_a with sharing ratio $q = 20$ is chosen as the key attribute of the forward index. We also fix the number of objects in C_a to 100000 and the average size of an object to 100 bytes.

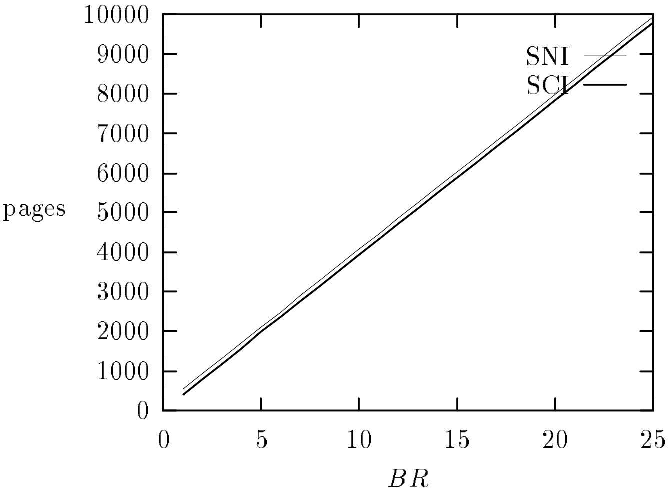

Figure 3: Storage Overhead

BR is the average number of direct links corresponding to an object in C_a. It is critical to the storage overhead of the direct links file and the non-clustering implementation. Therefore, we vary BR to observe its impact on storage requirements.

From Figure 3, we observe that the storage overhead for both implementations are proportional to BR. Since there are little difference between the storage requirements of the clustering and non-clustering methods, we conclude that most of the storage overhead is attributable to the direct links file. Fortunately, a single direct links file is shared among many indexes. Thus, the amortized overhead for each index is small.

5.3 Retrieval Cost

Since we use a forward index for comparison, we will only consider PT queries. Backward index which supports TP queries should have comparable cost and performance as the forward index. We estimate the cost for evaluating PT queries with different implementations of the forward index. The predicate class is C_a and the target class is C_c.

Clustering Index: The query is answered by first searching through the nonleaf and leaf nodes of the index, and then sequentially reading the direct links from the direct links file. The number of pages accessed for retrieval is:

$$RCI = h + 1 + \lceil q \cdot BR \cdot 2 \cdot OIDL / P \rceil.$$

Non-clustering Index: Queries can be answered with non-clustering indexes in the same way as with clustering indexes. However, the number of page accesses to the leaf nodes depends on the number of links corresponding to the key value. Further, since the direct links are not clustered, sequential read can't be used to fetch direct links.

The number of page accesses for randomly accessing k records from a file of n records which are divided into m pages is [5]:

$$H(k,m,n) = m \cdot \left[1 - \prod_{i=1}^{k} \frac{n-(n/m)-i+1}{n-i+1}\right].$$

The average number of links accessed from the direct links file is $q \cdot BR$. The number of direct links in the file is NDL and the number of pages for the file is SDL. Applying the above formula, we obtain the number of page accesses to the direct links file:

$$RDL_{ni} = H(q \cdot BR, SDL, NDL).$$

As a result, the total number of page access for retrieval using a non-clustering index is:

$$RNI = h + \lceil XP_{ni}/P \rceil + RDL_{ni}.$$

where XP_{ni} is the size of a leaf node record in a non-clustering index.

Semi-clustering Index: The semi-clustering index has a lower number of page accesses than non-clustering index, because the links in each cluster is sorted by the OIDs in C_a. The number of direct links accessed is $q \cdot BR/K$, where $1 \leq K \leq BR$. K is the average number of consecutive direct links with the same OIDs in C_a. Each random access to the direct links file will read $(K \cdot 2 \cdot OIDL)$ bytes sequentially. Thus, the number of page accesses to the direct links file is:

$$RDL_{si} = H(\frac{q \cdot BR}{K}, SDL, NDL) \cdot \lceil (K \cdot 2 \cdot OIDL)/P \rceil.$$

Thus, the total number of page accesses for retrieval using the semi-clustering index is:

$$RSI = h + \lceil XP_{si}/P \rceil + RDL_{si}$$

where $XP_{si} (= XP_{ni})$ is the size of a leaf node record in the semi-clustering index.

Traditional Forward Traversal: To answer a PT query with forward traversal, the number of pages accesses is:

$$RFT = \lceil N_a S_a/P \rceil + q(Br_1 + Br_1 Br_2 + ... + Br_1 Br_2 ... Br_{n-1})$$

Comparison

In the comparison, we fix K to 2 when we estimate the retrieval cost of the semi-clustering method. From Figure 4, the clustering index method has the best retrieval performance, which is consistent with our expectation. The retrieval cost of semi-clustering index is between that of the clustering and non-clustering indexes. If we increase the value of K for the semi-clustering index, its retrieval cost will shift toward the performance of the clustering index method.

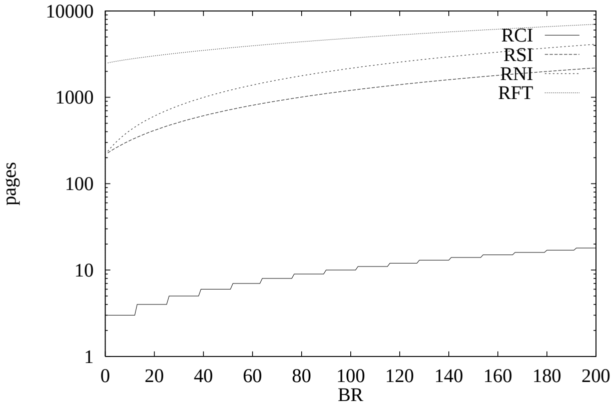

Figure 4: Retrieval Cost

6 Conclusion

In this paper, we examine the problems with object traversal and review some of the indexing organizations which provide short cuts for traversal. Also, we argue that direct links are effective in providing short cuts for object traversal. Based on direct links, a new class of indexing mechanisms, forward index and backward index, for both forward and backward traversals is proposed. Three ways of realizing direct links, namely, clustering index, semi-clustering index and non-clustering index, are evaluated. Our analysis shows that the clustering index implementation has the best performance in storage overhead and retrieval speed. Semi-clustering index has the same storage overhead as the non-clustering index, but its retrieval speed is faster. Although non-clustering index is not as good as the other two implementations, it is still much better than the traditional traversal approaches.

Reference

[1] E. Bertino & W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, June 1989, 196–214.

[2] Y. Ishikawa, H. Kitagawa & N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings of the 1993 SIGMOD Conference*, Washington, DC, June 1993, 247–256.

[3] W.-C. Lee & D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," *Proceedings of the 2nd International Computer Science Conference*, Hong Kong, Dec. 1992, 616–622.

[4] D. Maier & J. Stein, "Indexing in an object-oriented DBMS," *Proceedings of International Workshop on Object-Oriented Database Systems*, 1986, 171–182.

[5] S.B. Yao, "Approximating block accesses in database organizations," *Communications of ACM*, Vol. 20, No. 4, Apr. 1977, 260–261.