Branch Index: An approach for Query Processing in OODB

Pichayotai Mahatthanapiwat and Wanchai Rivepiboon
Department of Computer Engineering
Chulalongkorn University
Bangkok Thailand 10330
p41pmh@hotmail.com,wanchai.r@chula.ac.th

Abstract

In this paper, we present an access method called branch index for query processing of the aggregation hierarchy as a tree in object-oriented databases. The algorithm of branch generation will be proposed to generate all branches for the tree aggregation of classes in the database. For each branch, the information of linking objects is stored so that class traversal methods can be eliminated. Using a set of attribute indexes and identity indexes for each branch, associative searching can be conveniently performed. We discuss the retrieval and update operation and then develop cost models in terms of storage overhead, retrieval cost and update cost. When compared with the path dictionary index for multiple paths, the result shows that our approach has less storage overhead and the retrieval cost is improving.

Key words: *access method, object-oriented database, aggregation hierarchy, query processing.*

1. Introduction

At present, object-oriented databases have been widely used in most engineering applications, such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM) and Geographical Information System (GIS). The complexity of data in these applications makes the conventional database, such as the relational database cumbersome to manage them. One of the benefits of the object-oriented database is from its data model [11]. In the object data model, the value of an attribute does not limit to a primitive value, such as integer, real or string, but the value of an attribute can be either a primitive value or a complex value. The complex value of an attribute is a unique Object Identifier (OID) of an object in a class. If a class C consists of an attribute A whose domain is a class C', the class C can reference the class C' from the attribute A. We call this relation of classes as an aggregation hierarchy. In the same way, the class C' consists of an attribute A' whose domain is a class C'' so that the class C' can link to the class C'' directly and the class C can link to the class C'' indirectly. If a class N is referenced by a class C either directly or indirectly and the class N does not reference any classes, the class N will be called a leaf class of the aggregation hierarchy. On the other hand, a class C will be called the root class of the aggregation hierarchy if it references other classes but it is not referenced by any classes. Any classes in the aggregation hierarchy that are between the root class and the leaf class will be called intermediate classes.

Class traversal methods for an aggregation hierarchy can be performed as forward traversal and reverse traversal. In the forward traversal approach, we start from one class and traverse to its child class by using the value of the complex attribute. On the other hand, the reverse traversal approach traverses up to the parent classes. Usually, the forward traversal approach can perform conveniently because of the inherent pointer of the complex attributes. However, the reverse traversal approach has more trouble unless reverse pointers are implemented between classes. When there is a query, the class that the predicate is involved is called the predicate class and the class of the target objects is called the target class.

If the predicate class and the target class are far away, i.e. there are several intermediate classes between the target class and the predicate class, cost of traversal will be high because of intermediate classes traversal. Therefore, much research has been performed to reduce cost of class traversal whereas the associative searching is also in consideration. The indexing techniques are considered to accelerate database operations by constructing efficient access structures on a database given a certain physical implementation of the database. Secondary index on an attribute or a combination of attributes is useful for evaluating queries on a nested class in an object-oriented database. A classic research on index [1] has been done on an aggregation hierarchy, for example, multi index, nested index, path index. Other [3], [13], [14], [15], [16] researches on the aggregation hierarchy attempted to improve the performance of searching by using the concept form [1]. Indexing techniques on both aggregation hierarchy and inheritance hierarchy are proposed by [4], [8], [9] and [12].

Most indexing techniques that are used for the aggregation hierarchy are proposed as a path scheme. However, for the application that a class schema is more complicated than a path, such as a tree, a new access method should be considered to cope with all classes in the aggregation hierarchy. An example of the aggregation hierarchy that forms a tree of linking classes is shown in Figure 1. It consists of eight classes, *Person, Vehicle, Company, Bank Engine, Course, University* and *Computer.*

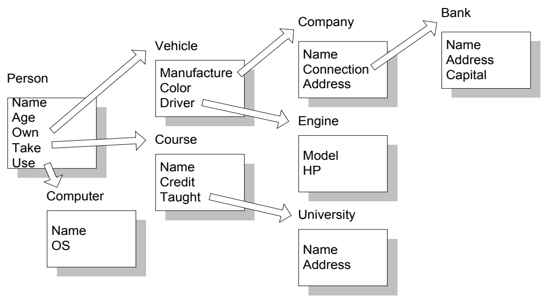


Figure 1. Aggregation hierarchy as a tree.

The *Person* class is the root class of the aggregation hierarchy, while *Bank*, *Engine*, *University* and *Computer* are leaf classes. The other classes; *Vehicle*, *Course* and *Company* are intermediate classes. We can create four possible paths from the root class to its leaf classes as follows.

Path 1: *Person→Vehicle→Company→Bank*
Path 2: *Person→Vehicle→Engine*
Path 3: *Person→Course→University*
Path 4: *Person→Computer*

When we specify an object of the *Person* class, using the forward traversal method can retrieve the corresponding objects of the nested classes for each path. It is also noticeable from Figure 1 that the join classes are the *Person* class and the *Vehicle* class. The Path 2 can be reduced to *Vehicle→Engine* because the corresponding objects of the Vehicle class from Path 1 are sufficient for further retrieval of objects form the classes of Path 2. We classify the queries by the following factors.

1. The class traversal methods from the predicate class to the target class.
   F(A,B) : Forward traversal from class *A* to Class *B*.
   R(A,B) : Reverse traversal from class *A* to class *B*.
2. The number of paths involved for the predicate class and the target class.
   SP: The predicate and the target class are on the same path.
   MP: The predicate class and the target class are on different paths.

In the following, we give examples of queries from the classification above. We denote *PC*, *TC* and *JC* for the predicate class, the target class and the join class respectively.

Q1: Retrieve persons who own cars made by the companies that connect to Bangkok Bank.
(R(PC,TC), SP)
Q2: Retrieve banks connected by the companies that manufacture cars for persons at the age of 40.
(F(PC,TC), SP)
Q3: Retrieve engines of the cars own by the persons who take course at Chulalongkorn University.
(R(PC,JC), F(JC,TC), MP)

Most indexing techniques can tackle the problem such as Q1 when the predicate is specified on the indexed attribute of the leaf class and the target class is the root class. A few techniques are proposed to eliminate the forward traversal between classes of the single path for the query Q2. Although applying the combination of various indexes can solve the query Q3, the joining between paths is still required and overhead occurred is considerable. The detail of overhead analysis will be discussed later.

The access methods of the aggregation hierarchy as a tree have been proposed recently. Direct Access to Terminal Virtual Path [23] is as follows. For each object in the root class *Person*, there will be corresponding objects in leaf classes *Bank, Engine, University* and *Computer*. Associated objects in leaf classes are stored together as if there were a path between them. This path is called Terminal Virtual Path (TVP). Therefore, the information in TVP consists of OIDs of the leaf classes that associate with the object in the root class. OID of the object in the root class is stored with the associated TVP as an entry in the linking file structure. Index can be created on simple attributes of the root class and map to the associated entries in the linking file. This access method shows that linking between objects in the root class and corresponding objects in the leaf classes stored in TVP can reduce cost of intermediate classes traversal. However, it is only suitable for the query that the predicate class and the target class are on leaf classes or the root class. Virtual Path Signature [24] is proposed to handle multi key indexing. For each aggregation of objects from the class schema from Figure 1, associated objects in leaf classes are stored together in a virtual path called Terminal Virtual Path (TVP) and associated objects in non-leaf classes are stored in a virtual path called Non-Terminal Virtual Path (NTVP). Signature is generated for objects in TVP and NTVP. The Virtual Path Signature shows significant improvement in retrieval when compared with Tree Signature [22], especially when the number of classes between the target class and the predicate class is high. However, Virtual Path Signature requires high storage overhead because of redundancy of objects due to reference sharing. Furthermore, its retrieval performance is lower when compared with the indexed attributes of the indexing techniques. Therefore a new approach should be proposed to tackle limitation mentioned above. It should have the characteristics as follows.

1. Its structure should be stored in the secondary storage other than OODB.
2. It should support traversal of classes in the aggregation hierarchy.
3. It should support associative searching.
4. It should support various kinds of queries for the aggregation hierarchy; i.e. the predicate class and the target class can be anywhere in the aggregation hierarchy.
5. Its cost model in terms of storage overhead and retrieval cost should be lower than other approaches when applied as multi paths, for example, path dictionary index [19].

The rest of the paper is organized as follows. Section 2 summarizes the related works and the path dictionary index that will be compared with our access method. Section 3 introduces the concept of the branch index. The implementation and database operation will be presented in Section 4 and Section 5 respectively. Cost models in terms of storage overhead, retrieval cost and update cost will be presented in Section 6. Finally, we conclude the paper in Section 7.

2. Related Works

In this section, we summarize some access methods, especially, the indexing techniques proposed for the aggregation hierarchy of object oriented databases.

Multi index [1] is created for two classes that linked by an inherent pointer of a complex attribute. For nth multi index, an index is created on a simple attribute of the class C_n and an indexed key is mapped to the associated OIDs of objects in the class C_n. For ith multi index, an index will be created on a complex attribute of a class C_i and the key will link to the associated OIDs of objects in the class C_i. If the predicate is specified on the indexed attribute of the class C_n and the target is the class C_i and there is a relation from C_i to C_n, $(n - I + 1)$ index lookups will be required. Therefore, this index is not appropriate for the query when the predicate class and the target class are far away. We can see that the multi index is only applicable for the query Q1 because it supports the reverse traversal from the nested class to the ancestor class. However, the multi index has flexibility for the update because it is easy to update linkage between indexed key and the associated OIDs.

Nested index [1] uses the concept that is similar to that of the multi index. There is only one index to map from the indexed key of a simple attribute on the leaf class of the path to the associated OIDs of objects in the root class. Therefore, it is suitable for the query that the predicate is specified on the indexed attribute of the leaf class and the target is the root class. This index can also support the query such as Q1. However, it is not suitable to use nested index if the predicate class and the target class are anywhere in the path. Furthermore, the update requires the reverse traversal method from the updated object to its ancestor objects of the root class. Therefore, the reverse pointers should be implemented for all classes in the path to support the update operation.

Path index [1] extends the concept from the nested index. In this approach, an index is created on a simple attribute of the leaf class and this indexed key links to the associated paths. The information stored in a path is the linking of OIDs of objects from the classes on the path so that if the predicate is specified on an indexed attribute of the leaf class, the target class can be any classes on the path. We can see that the path index supports only the query Q1. Although the path index can support more queries than the nested index, it requires more storage overhead.

Direct link [17] is a structure stored in the secondary storage. The information of an entry in the structure maintains links connecting objects in the root class and associated objects in the leaf class. This access method is proposed in the condition that the predicate class and the target class are the root class or the leaf class. Therefore, the direct links between two classes provide short cuts for object traversals. For example, an object O_1 of the root class links to objects O_x, O_y, and O_z of the leaf class. OIDs of O_1 and its associated OIDs of O_x, O_y and O_z will be stored together as an entry in a file. This example shows the forward direct link from the root class to the leaf class. The reverse direct link can also be created by using the similar approach, i.e. storing links between objects of the leaf class to its associated objects of the root class. Furthermore, in order to facilitate associative searching of the direct links, indexes can be built to map attributes of either end classes to the direct link organization. Although the direct link is applicable for the query Q1, Q2 and Q3, it is only appropriate when the predicate class and the target class are on the root class or the leaf class.

Path dictionary [18], [19], [21] is proposed in the concept of grouping all objects in the path that link to the same object in the leaf class. Information stored in an entry of the path dictionary, which is represented as an s-expression is useful for the traversal of objects between classes in the path. The attribute index is created on top of the path dictionary to assist associative searching. When the predicate is specified on the indexed attribute, the target objects from the qualified s-expressions can be easily retrieved. When compared with the path index, path dictionary index shows the improving in both storage overhead and cost of performance. Furthermore, it can cover more queries in such a way that the predicate class and the target class can be any classes in the path. Therefore, it supports more queries than previous indexing techniques. Although the query Q3 can be solved with three path dictionary indexes, i.e. Path 3$\rightarrow$Path 1$\rightarrow$Path 2, the overhead of joining classes is still high because the information of joining classes must be stored for three path dictionaries.

Signature, as an alternative approach, can be used instead of the indexing techniques mentioned above. Signature is rooted from applications in text databases, which require an efficient search method. Signature techniques are the alternative approach for searching the target objects because we cannot always predict which key attribute will be used to access the database. The researches about the signature techniques, which are applied for object-oriented databases are presented in [2], [5], [6], [7], [10], [20], [22], [24]. However, if we know attributes that are frequently used in queries, the indexing techniques should be used instead of the signature techniques because the retrieval performance of searching tree of the indexed attribute is better than scanning signatures in the signature file.

3. Branch Index Organization

In this section, after defining several terms we will use in the paper, we will introduce the organization of the branch index.

A. Definitions

Definition 1:
For an aggregation hierarchy as a tree, if C_j is a non-leaf class or a leaf class and C_n is a leaf class and it is accessible from the class C_j, a relation from the class C_j to the class C_n will be called a *branch* in the aggregation hierarchy.
Example 1: Let us consider the aggregation hierarchy as a tree in Figure 1. The following are possible branches for the aggregation hierarchy.

B_1: Person$\rightarrow$Vehicle$\rightarrow$Company$\rightarrow$Bank
B_2: Course$\rightarrow$University
B_3: Engine
B_4: Computer

Note that each class in a branch cannot be a member of other branches. The other possible branches can be as follows.
B'_1: Company$\rightarrow$Bank
B'_2: Vehicle$\rightarrow$Engine
B'_3: Course$\rightarrow$University
B'_4: Person$\rightarrow$Computer

Definition 2:
The *branch length* indicates the number of classes in a branch. A branch will be called a *complete branch* if its branch length is greater than one. If there is only one class in a branch, it will be called an *incomplete branch*.
Example 2: We can see from Example 1 that B_1 and B_2 are complete branches because their branch length is 4 and 2 respectively. B_3 and B_4 have only one class, so they are incomplete branches.

Definition 3:
For an aggregation hierarchy as a tree, the longest branch in the aggregation hierarchy is called the *main branch*. If L is a set of leaf classes in the aggregation hierarchy, the main branch will start from the root class C_1 to a leaf class C_n; when C_n is a member in L.
Example 3: The main branch from Example 1 is B_1 because it is the longest branch and it starts from the root class *Person* to the leaf class *Bank*.

Definition 4:
For an aggregation hierarchy as a tree, if C_k is a class in the main branch B_i that references a class C_m, which does not on B_i, there will be a *child branch* of B_i starting from class C_m to its accessible leaf class in L. The class C_m will be called a *join class*. Therefore, a join class is a class of a branch that can link to its child branches.
Example 4: Let us consider the aggregation hierarchy as a tree in Figure 1 and the example branches in Example 1. The branch B_1 can link to the branch B_3 by the join class *Vehicle*. The branch B_2 and B_4 are linked to B_1 by the join class *Person*. Therefore, B_2, B_3 and B_4 are child branches of B_1.

Definition 5:
For a branch B_i of an aggregation hierarchy as a tree, if its child branch is an incomplete branch, this child branch will be called a *leaf branch* of B_i.
Example 5: We can see from Example 2 that the branch B_3 and B_4 are incomplete branches. Since B_3 and B_4 are child branches of the branch B_1, they will be leaf branches of B_1.

B. Algorithm of Branch Generation
The purpose of the algorithm is to generate the minimum number of complete branches. The smaller number of complete branches, the smaller joining between them.
Given an aggregation hierarchy as a tree and L is a set of leaf classes in the aggregation hierarchy. The procedure of the algorithm is as follows.

1. Find the main branch by considering a leaf class C_{ni} in L that causes the maximum number of classes between the root class C_1 and the leaf class C_{ni}.
2. If C_{n1} is the result of the leaf class in the main branch from 1, then the new set $L = L - \{C_{n1}\}$.
3. Repeat while L is not empty.
4. Consider a join class in an existing branch B_i and find the new longest branch from the child class of that join class of B_i. If a branch B_j is a child branch of B_i and C_{nj} is the ending class of the branch B_j
    - 4.1 If the branch length of B_j is greater than 1, then the new set $L = L - \{C_{nj}\}$.
    - 4.2 If the branch length of B_j is equal to 1, then the new branch is a leaf branch and will be combined to B_i. The new set $L = L - \{C_{nj}\}$.
5. Go to step 3.

An example of this algorithm is as follows:
In Figure 1, L = *{Bank, Engine, University, Computer}*. We can see that the longest branch is from the *Person* class to the *Bank* class. Therefore, the main branch will be generated and *Bank will be deleted* from L. The new set of L = *{Engine, University, Computer}*.

Since *Course* is the child class of the *Person* class in the main branch, the new branch will start from the *Course* class to the candidate leaf classes in L. So the new branch is generated from the *Course* class to the *University* class. For the remaining classes in L, we can see that the *Engine* class and the *Computer* class are direct child classes of the join class *Vehicle* and *Person* respectively. Since they are incomplete branches, *Engine* and *Computer* will be parts of the main branch. When set L is empty, the branch generation will be terminated.

C. Branch Index Organization
The architecture of the branch index is shown in Figure 2. The branch index is a separate structure from the object-oriented database and it is stored in the secondary storage. After using the algorithm of branch generation, the number of branches and the corresponding classes will be obtained. A set of attribute indexes and identity indexes is on top of the branch. The branch index consists of the following components.

Branch information
The information in the branch is OIDs linkage of objects for the classes in the branch. Therefore, the class traversal can be handled in the branch information instead of traversal in the database. In case of any child branches, the OIDs and pointers of the parent branch are also included as the information of the child branch. So, the traversal from the child branch to its parent branch can be easily managed.

Attribute Index
While the branch information can facilitate traversal among objects of classes in the branch, it does not support predicate evaluation that involves searching the object meeting the conditions specified on their attribute values. To facilitate the associative searching, attribute indexes should be used to map attribute values to OIDs in the branch information. For example, to tackle the query Q1 from Section 1, the attribute index should be created for the *Name* attribute of the *Bank* class. To find the target object of the *Person* class, we scan the attribute index of the *Name* attribute of the *Bank* class to obtain the qualified OIDs of the *Bank* class and the entry location of the branch information that store those OIDs. Then, we use the OIDs linkage in the branch information to retrieve the qualified OIDs from the target class.

Identity Index
Instead of creating the index by mapping the value of simple attributes to OIDs in the branch information, the identity index uses the values of complex attributes. Therefore, the branch information can be obtained with a given OIDs by using the identity index. Since identity search is important for retrieval and update, the identity index can reduce the cost for retrieval and update operations.

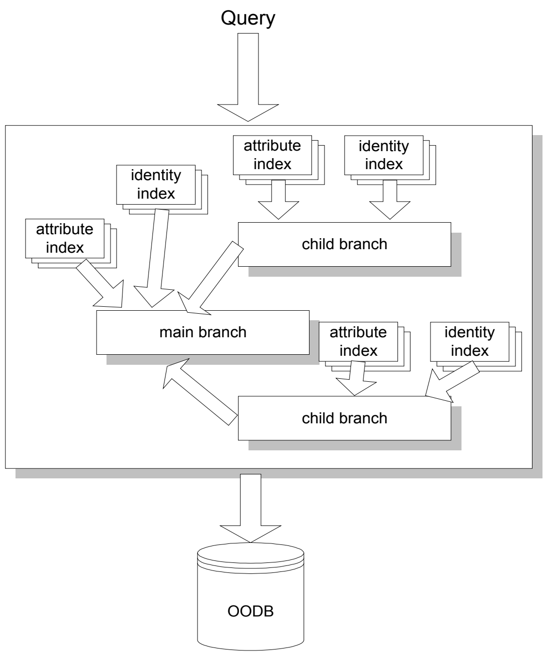


Figure 2. Branch index.

D. Details of the branch information

For a complete branch B_i of the aggregation hierarchy as a tree, there is a relation from the starting class to the ending class of this branch. When objects are instantiated, in logical view, the objects of the starting class point to their child objects until the objects of the ending class. We represent linking of objects with linking of their OIDs or OIDs linkage. Therefore, it is much faster to traverse by using OIDs linkage in a branch than objects in the database.

The necessary information that should be kept in a complete branch consists of the following:

- OIDs of objects of the classes in the branch and the pointers to their child objects.
- OIDs of the parent objects for the branch, in case it is not the main branch, and the corresponding pointers to the parent branch.
- OIDs of the leaf branch.

When several objects in one class reference the same object of the child class, it is called the reference sharing. Therefore, OIDs linkage should be kept to save the storage in case of the reference sharing. Figure 3 shows an example of object instantiation and the reference sharing by using the information from Figure 1.

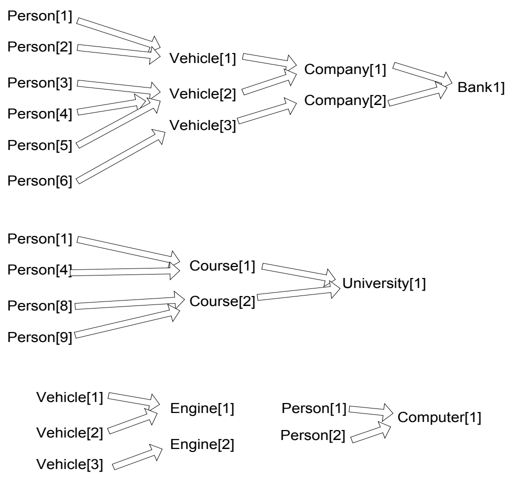


Figure 3. An example of object instantiation.

From Figure 3, the ith object of the *Person* class will be denoted as *Person[i]*. OIDs of objects of other classes will use the same notation. We can see that *Person[1]* and *Person[2]* reference the same object *Vehicle[1]*.

From the algorithm of branch generation presented earlier, we obtain two complete branches of the result as follows.

- The main branch that starts from the *Person* class to *Vehicle*, *Company* and *Bank*. Since the *Computer* class and the *Engine* class are leaf branches of the main branch, they are also part of the main branch.
- The child branch that starts from the *Course* class to the *University* class.

To cope with the reference sharing, we use the concept that is similar to that of the path dictionary. All ancestor objects of an object in the ending class of the branch will be kept as an entry of the branch information. For example, for the object *Bank[1]* of the ending class, the entry of the main branch consists of *Person[1]* to *Person[6]*, *Vehicle[1]* to *Vehicle[3]*, *Company[1]* to *Company[2]* and *Bank[1]*. All linkages between objects are also kept, for example, pointer between *Person[1]* and *Vehicle[1]*, pointer between *Person[2]* and *Vehicle[1]* and so on. Since the leaf branch *Engine* and *Computer* are parts of the main branch, their objects will correspond to the main branch. Therefore, *Engine[1]* will be linked from *Vehicle[1]* and *Vehicle[2]*; *Engine[2]* will be linked from *Vehicle[3]*. Finally, *Computer[1]* will be linked from *Person[1]* and *Person[2]*.

4. Implementation

The structure of an entry of a branch is shown in Figure 4.

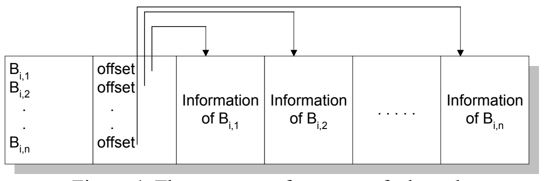


Figure 4. The structure of an entry of a branch.

From Figure 4, $B_{i,1}$ denotes the starting class of ith branch while $B_{i,n}$ denotes the ending class of ith branch. We assume that there are *n* classes that have relation in ith branch. The relation is in the form that $B_{i,1}$ references $B_{i,2}$ and $B_{i,2}$ references $B_{i,3}$,…, $B_{i,n-1}$ references $B_{i,n}$. The offset for each class points to the location of 1st OID of the object in that class, for example, the offset of $B_{i,1}$ locates the address of 1st OID of object of the starting class. The example of an entry of the main branch is shown in Figure 5.

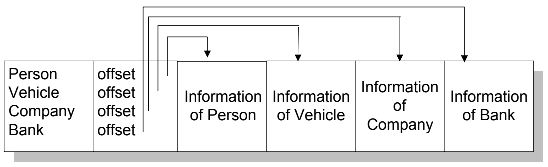
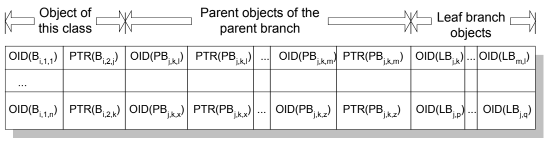


Figure 5. An example of the structure of an entry of the main branch.

There are four classes of the main branch and their associated information for each class. The offset will point to the first entry of the information for that class. Therefore, given a specified class, we can determine the information comfortably. The detail implementation of information for each class of a branch is shown in Figure 6.


Object of this class
Parent objects of the parent branch
Leaf branch objects

$OID(B_{i,1,1})$	$PTR(B_{i,2,j})$	$OID(PB_{j,k,l})$	$PTR(PB_{j,k,l})$	...	$OID(PB_{j,k,m})$	$PTR(PB_{j,k,m})$	$OID(LB_{j,k})$	...	$OID(LB_{m,l})$
...									
$OID(B_{i,1,n})$	$PTR(B_{i,2,k})$	$OID(PB_{j,k,x})$	$PTR(PB_{j,k,x})$	...	$OID(PB_{j,k,z})$	$PTR(PB_{j,k,z})$	$OID(LB_{j,p})$	...	$OID(LB_{j,q})$

Figure 6. The structure of an information of a class in an entry of a branch.

From Figure 6, $B_{i,1,1}$ and $B_{i,1,n}$ denotes the 1st object and the nth object of the starting class of the ith branch respectively. The *n* objects that belong to the starting class of the ith branch may point directly or indirectly to the same object of the ending class of the ith branch. Each object of the starting class is implemented as a record that consists of members as follows.

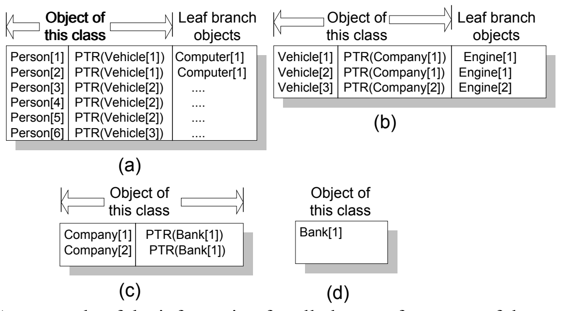
1. OID of the object itself.
2. Pointer to OID's child object.
3. Multiple pairs of OID's parent object and its pointer to the parent branch (except the main branch).
4. Multiple OIDs of leaf branch objects for the starting class in case that the starting class has leaf branches.

In general, for the second class to the class before the ending class of a branch, a record for each object of the class consists of members as follows.

1. OID of the object itself.
2. Pointer to OID's child object.

3. Multiple OIDs of leaf branch objects for the class in case it has leaf branches.

Finally, for the ending class, the information will be only OID of the ending class. The number of OIDs for the ending class will be only one for each entry of the branch. Figure 7 shows an example of the information of the class *Person*, *Vehicle*, *Company* and *Bank* for an entry of the main branch.



(a)

Object of this class		Leaf branch objects
Person[1]	PTR(Vehicle[1])	Computer[1]
Person[2]	PTR(Vehicle[1])	Computer[1]
Person[3]	PTR(Vehicle[2])	....
Person[4]	PTR(Vehicle[2])	....
Person[5]	PTR(Vehicle[2])	....
Person[6]	PTR(Vehicle[3])	....

(b)

Object of this class		Leaf branch objects
Vehicle[1]	PTR(Company[1])	Engine[1]
Vehicle[2]	PTR(Company[1])	Engine[1]
Vehicle[3]	PTR(Company[2])	Engine[2]

(c)

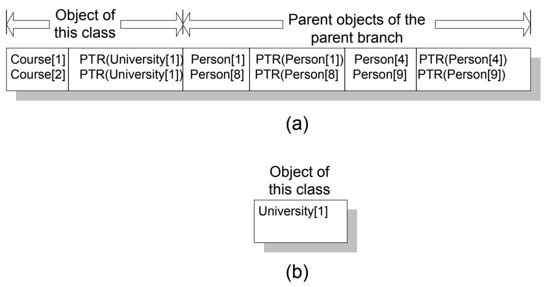
Object of this class	
Company[1]	PTR(Bank[1])
Company[2]	PTR(Bank[1])

(d)

Object of this class
Bank[1]

Figure 7. An example of the information for all classes of an entry of the main branch.

From Figure 7, there is no information of the parent objects and the parent pointers because it is the ancestor branch of all branches. However, for the other branches, we have to keep OID's parent objects and their pointers as mentioned earlier. An example of the information for the child branch is shown in Figure 8.



(a)

Object of this class		Parent objects of the parent branch			
Course[1]	PTR(University[1])	Person[1]	PTR(Person[1])	Person[4]	PTR(Person[4])
Course[2]	PTR(University[1])	Person[8]	PTR(Person[8]	Person[9]	PTR(Person[9])

(b)

Object of this class
University[1]

Figure 8. An example of the information for all classes of an entry of the child branch.

The information of the *Course* class and the *University* class is shown in Figure 8(a) and Figure 8(b) respectively. Notice that this child branch has no leaf branch for all classes of the branch. However, since it is the child branch of the main branch, the OIDs of the parent objects and pointers to the main branch have to be stored.

At present time, the price of the media storage is decreasing and the capacity of the storage is increasing. Therefore, the storage overhead of an access method is not significant when compared with the retrieval performance. The branch information will be created for every branch generated in case that the predicate class and the target class can be any classes in the aggregation hierarchy. It will be stored sequentially on the secondary storage. However, if we know exactly where the predicate class and the target class are, we can create the branch information for the branches involved. An entry of the branch information is not allowed to cross page boundaries unless its size is greater than the page size. Free space directory is required for each page to inform the free space left. If there is not space enough left for an entry of the branch, the new page will be allocated. Therefore, the free space directory will be stored before the branch information.

The data structure that we will use to model the various indexes is based on tree-structures, such as B^+-trees. The format of a non-leaf node for the identity index is similar to that of the attribute index. Figure 9(a) and Figure 9(b) shows the format of a non-leaf node for the identity index and the attribute index respectively.

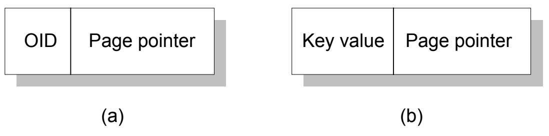


Figure 9. Non-leaf node record of the identity index and the attribute index.

The format of a non-leaf node record of the identity index consists of OID and page pointer. The page pointer contains the address of the next level non-leaf page of the OID or the address of the leaf page of the OID. The format of a non-leaf node record of the attribute index is similar to that of the identity index. Key value is used for the attribute index instead of OID used for the identity index.

The format of a leaf node record of the identity index and the attribute index is shown in Figure 10.

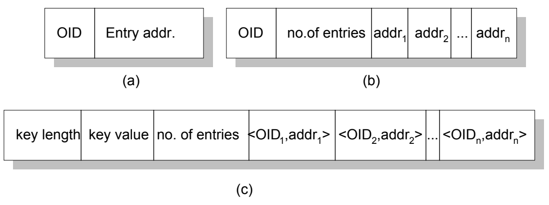


Figure 10. Leaf node record of the identity index and the attribute index.

The identity index is created for all objects for each class of a branch. Therefore, if there are *n* classes involved in a branch, there will be *n* identity indexes for each class in the branch. A leaf node record for the identity index of any classes is shown in Figure 10(a). However, for the class that is a leaf branch, the identity index is shown in Figure 10(b). The leaf node for the attribute index is shown in Figure 10(c) of the corresponding OIDs and addresses for the indexed attribute.

5. Retrieval and Update Operation

In this section, we discuss the retrieval and update operation on the branch index.

5.1 Retrieval Operations

We use the aggregation hierarchy as a tree shown in Figure 1 to discuss the retrieval operation. As mentioned in Section 3, there will be two branches generated when we use the algorithm of branch generation. Therefore, we will use B_1 and B_2 to represent the main branch and the child branch respectively. We can classify a query that involves the predicate class and the target class as follows.

- The predicate class and the target class are on the same branch.
- The predicate class and the target class are on different branches.

A. The predicate class and the target class are on the same branch

In this case, the predicate class and the target class can be any classes on the branch. For example, to find the owner of the car manufactured by the company that connected to Bangkok bank. We can see that the predicate class is the *Bank* class and the target class is the Person class of branch B_1. In this case, if an index is created on the *Name* attribute of the *Bank* class, we can determine the qualified entries of the branch that associate with the indexed key. Since the information of an entry of a branch consists of OIDs of every class in the branch, OIDs of the objects in the *Person* class can be easily retrieved. Furthermore, if the target objects are on any classes of the branch, the OIDs of the objects for those classes can also be easily retrieved. When mention about the leaf branch, it is also a part of a complete branch. The query that the predicate class or the target class are on the leaf branch is similar to that of discussion above, for example, to find the manufacturer of the car that own by the person who use the computer with OS UNIX. We can see that the leaf branch, in this case, is the *Computer* class that is a part of the main branch B_1. The predicate class is the *Computer* class and the target class is the *Company* class of branch B_1. If an index is created on the *Name* attribute of the *Computer* class, we can use this index to find the qualified entries of the branch B_1 and access the qualified OIDs from the *Company* class. We can conclude that if the predicate class and the target class are on the same branch and the predicate is specified on the indexed attribute, we can access the qualified OIDs of the target class by scanning the indexed attribute. Several attribute indexes can be created with low storage overhead because the overhead is only for the non-leaf node records and leaf node records of the attribute indexes.

B. The predicate class and the target class are on different branches

In this case, the predicate class and the target class occurs on different branches, for example, a predicate is on a class of branch B_1 and the target is on a class of branch B_2. From Figure 1, the query "to find the university of the person who own the car manufactured by the company that connect to Bangkok bank" is an example above. We can see that the predicate occurs on the *Bank* class of the main branch B_1 and the target class is the *University* class of the child branch B_2. The main branch B_1 connects to its child branch B_2 by the *Person* class. If an index is created on the *Name* attribute of the *Bank* class, the OIDs of the *Person* class from the qualified entries of the main branch will be obtained. Then, we can use the forward traversal technique from each qualified OIDs of the *Person* class to the *Course* class and the *University* class of branch B_2 and access the qualified objects from the *University* class. Also, an alternative approach is to scan the identity index of the *Person* class for the qualified objects on the branch B_2 to access the target objects of the *University* class from the qualified entries of branch B_2. On the contrary, the query "to find the bank that is connected by the manufacturer of the car own by the person who take course at Chulalongkorn University" is somewhat different. Although the predicate class and the target class are on different branches, in this case, we can not use the join class for the reverse traversal. If an index is created on the *Name* attribute of the *University* class of branch B_2, we can scan this index to obtain the qualified entries of the branch B_2. Because the branch B_2 is the child branch of B_1 and the information of branch

B_2 consists of OIDs and the addresses of the parent branch B_1, we can use these information to determine the qualified entries of branch B_1 and retrieve the OIDs from the *Bank* class. Therefore, we can conclude that the traversal from the child branch to its parent branch can be achieved easily by using the information stored in the child branch. However, we do not store information from the parent branch to its child branch because we can use the forward traversal method from the join objects to the target objects directly.

5.2 Update Operations

We use Figure 1 and Figure 3 to discuss the update operations. We consider only update the complex attribute because it reflects the information stored in the branch. We can classify the update operations as follows.

- Update the reference between the parent object and the child object on the same branch.
- Update the reference between the parent object and the child object on different branches.

A. Update the reference on the same branch.

In this case, the parent object and its child object are on the classes of the same branch. We assume that an object *O* of class *C* changes the reference from an object *O′* of class *C′* to an object *O″* of class *C′*. We have to search the identity index of class *C′* to find the entries that associate with object *O′* and *O″*. Furthermore, we assume that E_1 and E_2 are the corresponding entries for *O′* and *O″* respectively. If E_1 is equal to E_2, we will change the pointer that *O* points from *O′* to *O″*. However, when E_1 is not equal to E_2, we have to delete OIDs and the pointers of class *C* and its ancestor classes that associate with object *O′* in E_1 and add these information in the entry E_2 that associates with object *O″*. Also, the information stored in the child branch for the moved class has to be updated. Meanwhile, the associated identity indexes have to be updated. For example, if *Vehicle[1]* that references *Company[1]* changes to *company[5]*, we firstly search the entry of a branch for *Company[1]* and *Company[5]*. If the entries are different, we have to delete *Vehicle[1]* and the pointer to *Vehicle[1]* from the entry of *Company[1]* and add this information in the entry of *Company[5]*. Furthermore, we have to move *Person[1]*, *Person[2]* and corresponding pointers from the entry of *Company[1]* to the entry of *Company[5]*. All associated leaf branch objects for *Vehicle[1], Person[1]* and *Person[2]* have to be moved. Therefore, *Engine[1]* and *Computer[1]* will be moved to the entry of *Company[5]*. The child branch B_2 is affected when the entry of its parent branch is updated. Therefore, the information that associates with *Person[1]* in *Course[1]* will also be updated. Finally, the identity index for *Person[1], Person[2], Vehicle[1], Engine[1] and Computer[1]* have to be updated. Additionally, if the attribute index is created for the classes involved for the moving, the attribute index will also be updated, for example, if an index is created for the *Name* attribute of the *Computer* class, this attribute index has to be updated by removing the associated addresses with corresponding to *Company[1]* from the leaf node record and insert the address of the entry that corresponding to *Company[5]* to that leaf node record.

B. Update the reference on different branches.

We assume that an object *O* in a class *C* of branch B_1 changes the reference from an object *O′* in a class *C′* of branch B_2 to an object *O″* in the class *C′* of branch B_2. We have to search the identity index of the class *C′* of the branch B_2 to find the entries that associate with the object *O′* and *O″* and assume that they are E_1 and E_2 respectively. We will only perform with the branch B_2 by deleting OID of *O* and the pointer of *O* from the entry of *O′* and insert them to the entry of *O″*. For example, *Person[1]* of branch B_1 that previously references *Course[1]* of branch B_2 is updated to reference *Course[2]*. Therefore, we will delete *Person[1]* and its pointer from *Course[1]* and insert them to the entry of *Course[2]*. Finally, we update the identity index of *Person[1]* and the attribute index involved on the branch B_2.

6. Cost Model

In this section, we will formulate the cost model in terms of storage overhead, retrieval cost and update cost. Then, we will compare them with the cost model of the path dictionary index. We will give the parameters that used in the analysis and adopt some common parameters from [19] to facilitate our comparison.

Parameters :

$N_{i,j}$	: The number of objects in class j of branch i or path dictionary i.
$A_{i,j}$	: The complex attribute of class j on branch i or path dictionary i.
$D_{i,j}$	: Distinct value of complex attribute $A_{i,j}$.
$UIDL$	: The length of Object Identifier.
P	: Page size.
pp	: The size of page pointer.
f	: Average fan out from a non-leaf node.

kl	: Average length of a key value in attribute index.
$OFFL$	: The length of offset field in the path dictionary.
SL	: The length of start field in the path dictionary and branch information.
FSL	: The length of free space in path dictionary and branch information.
EL	: The length of EOS in path dictionary.
PL	: The length of pointer in branch information.
$SA_{i,j,k}$	: Simple attribute k of class j on branch i or path dictionary i.
$U_{i,j,k}$	: The number of distinct values for simple attribute $SA_{i,j,k}$.
$q_{i,j,k}$	: The ratio of shared attribute value $= N_{i,j} / U_{i,j,k}$.
Pk_i	: Reference sharing of the parent class of class i.
$k_{i,j}$	: Reference sharing of class j on branch i or path dictionary i.
$nlb_{i,j}$	: The number of leaf branch of class j on branch i.

We have adopted values for some parameters as in [19]. The chosen values of the path dictionary index and branch index are listed in Table 1.

$UIDL$	= 8	$OFFL$	= 2
P	= 4096	SL	= 2
pp	= 4	FSL	= 2
f	= 218	EL	= 4
kl	= 8	PL	= 2

Table 1. Parameters of cost models.

Performance is measured by the number of I/O accesses. We use a page to estimate the storage overhead and the cost of performance because it is the basic unit for data transfer between the main storage and the secondary storage. All lengths and sizes above are in bytes. To directly adopt the formulae from [19], we follow their assumptions.

Assumptions :

1. There are no partial instantiation. This implies that $D_{i,j} = N_{i,j+1}$.
2. All key values have the same length.
3. Attribute values are uniformly distributed among the objects of the class defining the attribute.
4. All attributes are single-valued.

6.1 Storage Overhead

In this subsection, we will adopt storage overhead of the path dictionary index developed by [19] and formulate our storage overhead for the branch index. We will use the information of Figure 11 in our analysis. All branches in Figure 11(a) are generated by using the algorithm of branch generation for the aggregation hierarchy of Figure 1, while all path dictionaries in Figure 11(b) are generated to mimic the branches in Figure 11(a).

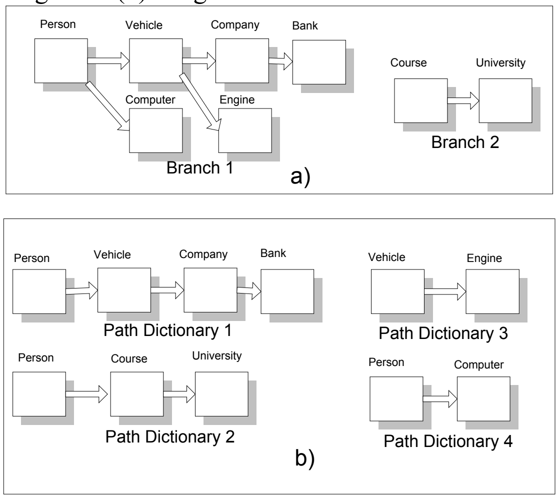


Figure 11. Branch and path dictionary generated from the aggregation hierarchy.

6.1.1 Path Dictionary Index

Path dictionary

For the path dictionary i, the average number of objects in an s-expression is:

$$NOBJ = 1 + \sum_{l=1}^{n-1} \prod_{j=l}^{n-1} k_{i,j}.$$

when there are n classes in the path dictionary i.

The average size of an s-expression is:

$$SS = SL * (n-1) + (UIDL + OFFL) * NOBJ \ + EL.$$

The number of pages needed for all of the s-expressions on the path is:

$$SSP = \begin{cases} \lceil N_{i,n} / \lfloor P / SS \rfloor \rceil & \text{if } SS \le P, \\ N_{i,n} * \lceil SS / P \rceil & \text{if } SS > P. \end{cases}$$

The number of pages needed for the free space directory is:

$$FSD = \lceil SSP * (pp + FSL) / P \rceil.$$

The total number of objects in this path dictionary is:

$$TOBJ = NOBJ * N_{i,n}.$$

identity index

The number of leaf pages needed for path dictionary i is:

$$LP_{iden} = \lceil TOBJ / \lfloor P / (UIDL + pp) \rfloor \rceil.$$

The number of non-leaf pages is:

$$NLP_{iden} = \lceil LP_{iden} / f \rceil + \lceil \lceil LP_{iden} / f \rceil / f \rceil + \ldots \ + \ldots + x.$$

where $x < f$. If $x \neq 1, NLP_{iden}$ is increased by 1 for the root node.
Therefore, the total number of identity index is:

$$IIP = LP_{iden} + NLP_{iden}.$$

attribute index

The average number of pages needed for a leaf node record is:

$$XP_{SA_{i,j,k}} = kl + q_{i,j,k} * (UIDL + pp).$$

The number of leaf node pages is:

$$LP_{SA_{i,j,k}} = \begin{cases} \lceil U_{i,j,k} / \lfloor P / XP_{SA_{i,j,k}} \rfloor \rceil & \text{if } XP_{SA_{i,j,k}} \le P. \\ U_{i,j,k} * \lceil XP_{SA_{i,j,k}} / P \rceil & \text{if } XP_{SA_{i,j,k}} > P. \end{cases}$$

The number of non-leaf pages is:

$$NLP_{SA_{i,j,k}} = \lceil LO_{SA_{i,j,k}} / f \rceil + \lceil \lceil LO_{SA_{i,j,k}} / f \rceil / f \rceil + \ldots + x.$$

Where $LO_{SA_{i,j,k}} = \min(U_{i,j,k}, LP_{SA_{i,j,k}})$ and

$x<f$. If $x \neq 1, NLP_{SA_{i,j,k}}$ is increased by 1 for the root node.
Therefore, the total number of pages for attribute index is:

$$AIP_{SA_{i,j,k}} = LP_{SA_{i,j,k}} + NLP_{SA_{i,j,k}}.$$

In case of m attribute indexes:

$$AIP = AIP_{index1} + AIP_{index2} + \ldots + AIP_{indexm}.$$

Therefore, the storage cost for path dictionary i is:

$$SC_{PDI} = FSD_i + SSP_i + IIP_i + AIP_i.$$

6.1.2 Branch Index
Branch Information
For an aggregation hierarchy as a tree and after applying the algorithm of branch generation, we assume that m branches are generated.
We also assume that there are n classes in a branch B_i. These classes are related as in the form $C_1C_2C_3\ldots C_n$. The size that associates with one object of the class C_1 of B_i consists of the following:
- OID of this object for class C_1 and its pointer to the child object.
- OIDs and pointers of the parent objects for the object in the first class of branch B_i.
- OIDs of leaf branch objects for class C_1 of branch B_i.

$$= (UIDL + PL) + Pk_i * (UIDL + PL) + (nlb_{i,1}) * UIDL.$$
$$= (Pk_i + 1) * (UIDL + PL) + (nlb_{i,1}) * UIDL.$$
$$= (Pk_i + nlb_{i,1} + 1) * UIDL + (Pk_i + 1) * PL.$$

The size of an entry for one object in a class C_j of branch B_i; when $2 \leq j \leq n-1$; consists of the following:
- OID of this object for class C_j and its pointer to the child object.
- OIDs of leaf branch objects for class C_j of branch B_i.

$$= UIDL + PL + (nlb_{i,j}) * UIDL.$$

The size of an entry for one object in the class C_n of branch B_i:

$$= UIDL.$$

The size of an entry for every object in class C_1 of branch B_i:

$$SE(B_{i,1}) = (\coprod_{j=1}^{n-1} k_{i,j}) * [(Pk_i + nlb_{i,1} + 1) * UIDL \quad + (Pki + 1) * PL] + SL.$$

The size of an entry for every object in class C_j of branch B_i, $2 \leq j \leq n-1$:

$$SE(B_{i,j}) = (\coprod_{l=j}^{n-1} k_{i,l}) * [(nlb_{i,j} + 1) * UIDL + PL] + SL.$$

The size of an entry for every object in class C_n of branch B_i:

$$SE(B_{i,n}) = UIDL + SL.$$

The total size of an entry of branch B_i:

$$SE(B_i) = SE(B_{i,1}) + \sum_{j=2}^{n-1} SE(B_{i,j}) + SE(B_{i,n}).$$

In case of the main branch B_1.

The size of an entry for every object in class C_j of branch B_1, $1 \leq j \leq n-1$:

$$SE(B_{1,j}) = (\prod_{l=j}^{n-1} k_{1,l}) * [(nlb_{1,j} + 1) * UIDL + PL] + SL.$$

The size of an entry for every object in class C_n of branch B_1:

$$SE(B_{1,n}) = UIDL + SL.$$

The total size of an entry of branch B_1:

$$SE(B_1) = \sum_{j=1}^{n-1} SE(B_{1,j}) + SE(B_{1,n}).$$

If BP_i is the number of pages used for every entry in the branch B_i, then

$$BP_i = \begin{cases} \lceil N_{i,n} / \lfloor P / SE(B_i) \rfloor \rceil, \text{ if } SE(B_i) \le P \\ N_{i,n} * \lceil SE(B_i) / P \rceil. \text{ if } SE(B_i) > P \end{cases}$$

The number of pages for the free space directory of branch B_i:

$$FSD_i = \lceil BP_i * (pp + FSL) / P \rceil.$$

The total size for all branch.

$$TBP = \sum_{i=1}^{m} (BP_i + FSD_i)$$

Identity Index

The identity index will be created for every object for each class of a branch. The average length of a leaf node index record for the identity index of class C_j.

$$XI = \begin{cases} UIDL + PP, & \text{if } C_j \text{ is not a leaf branch} \\ UIDL + Pk_j * PP. & \text{if } C_j \text{ is a leaf branch} \end{cases}$$

The number of leaf pages for the identity index of class C_j on branch B_i.

$$LP_{iden,i,j} = \lceil N_{i,j} / \lfloor P / XI \rfloor \rceil.$$

The number of non-leaf pages for the identity index of class C_j on branch B_i.

$$NLP_{iden,i,j} = \lceil LP_{iden,i,j} / f \rceil + \lceil \lceil LP_{iden,i,j} / f \rceil + f \rceil + \ldots + x.$$

If $x < f$ and $x \neq 1$, we will add 1 in $NLP_{iden,i,j}$ for the root node. Therefore, the number of pages for the identity index of class C_j on branch B_i:

$$IIP_{i,j} = LP_{iden,i,j} + NLP_{iden,i,j}.$$

If there are n classes on the branch B_i, the number of pages for the identity index of branch B_i:

$$IIP_i = \sum_{j=1}^{n} IIP_{i,j}.$$

The number of of pages for the identity index of every branch is:

$$TIIP = \sum_{i=1}^{m} IIP_i.$$

Attribute Index

When creating the attribute index on a primitive value of a class *j* of branch B_i, $SA_{i,j,k}$ represents a primitive value *k* of the class *j* of branch B_i and an index is created on $SA_{i,j,k}$. The average length of a leaf node index record for the attribute index is:

$$XP_{SA_{i,j,k}} = kl + q_{i,j,k} * (UIDL + PP).$$

The number of leaf pages of the attribute index on branch B_i is:

$$LP_{SA_{i,j,k}} = \begin{cases} \lceil U_{i,j,k} / \lfloor P / XP_{SA_{i,j,k}} \rfloor \rceil, \text{if } XP_{SA_{i,j,k}} \le P \\ U_{i,j,k} * \lceil XP_{SA_{i,j,k}} / P \rceil. \text{ if } XP_{SA_{i,j,k}} > P \end{cases}$$

The number of non leaf pages of the attribute index on branch B_i is:

$$NLP_{SA_{i,j,k}} = \lceil LO_{SA_{i,j,k}} / f \rceil + \lceil \lceil LO_{SA_{i,j,k}} / f \rceil / f \rceil + \ldots + x.$$

when $LO_{SA_{i,j,k}} = \min(U_{i,j,k}, LP_{SA_{i,j,k}})$ and $x < f$. If $x \neq 1$ add 1 to $NLP_{SA_{i,j,k}}$ for the root node.
Therefore, the number of pages for index on $SA_{i,j,k}$ is:

$$AIP_{SA_{i,j,k}} = LP_{SA_{i,j,k}} + NLP_{SA_{i,j,k}}.$$

Actually, we can create many attribute indexes. If there are *n* indexes on the branch B_i, the number of pages for these indexes is:

$$TAIP_i = \sum_{j=1}^{n} AIP_{i,\,indexj}$$

when $AIP_{i,\,indexj}$ is the *j*th index of branch B_i.
The number of pages for the attribute index of every branch is:

$$TAIP = \sum_{i=1}^{m} TAIP_i.$$

Finally, the storage cost is:

$$SC_{BI} = TBP + TIIP + TAIP.$$

6.1.3 Comparison

Using the formulae developed above, we compare the storage cost between the path dictionary index and branch index. We use the aggregation hierarchy as a tree mentioned earlier and the result of branches and paths are shown in Figure 11. The storage costs are calculated for all branches and paths and their associated indexes. Attribute indexes are created for all leaf classes. Also, we fix the cardinality of the root class to 200,000.

We assume that all reference sharing of all classes and the shared key values are set to the same value, which we represent it as *K*. We observe the impact of *K* to the storage overhead and cost of performance. The value of *K* is varied from 2 to 10 as shown from the x-axis of Figure 12. It is shown apparently that the storage cost of the branch index is less than that of the path dictionary index in all ranges of *K*. The storage cost of the path dictionary index is higher because we have to store all information for every path.

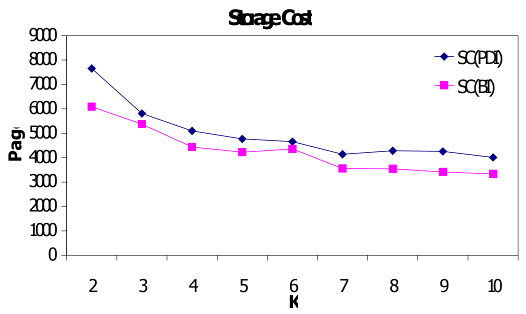


Figure 12. Storage cost of the path dictionary index and branch index.

6.2 Retrieval Cost

To simplify our analysis, we assume that there is only one predicate attribute in the queries and the predicate is specified on the indexed attribute.
We will classify cost formulae as in the discussion in Section 5.1. Furthermore, we choose the identity index to scan for the required entries instead of the forward traversal technique.

6.2.1 The predicate class and the target class are on the same branch

In this case, the predicate class and the target class are on the same branch. Therefore, it is convenient to perform the class traversal in the branch when using the branch index. However, it is possible that, in the considered branch, the predicate class and the target class are on different path dictionaries.

6.2.1.1 Path Dictionary Index

The retrieval cost of path dictionary index consists of the following:
- Cost of attribute index scanning.
- Cost of accessing the target objects for the qualified s-expressions when the target class is in the same s-expression as the predicate class, otherwise access the qualified join objects to traverse to the target objects in the other path dictionary.

Case 1: the predicate class and the target class are on the same path dictionary

$$RC_{PDI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SS / P \rceil.$$

when h_{attr} is the height of the attribute index -1; for the predicate class. XP_{attr} is the leaf node record of the attribute index. $N_{P/Q}$ is the number of the qualified s-expressions for the predicate P of query Q. SS is an s-expression of the path dictionary.

Case 2: the predicate class and the target class are on different path dictionaries

We can formulate the retrieval cost according to the location of the target class and join class.

- *The target class is an ancestor class of the join class*

$$RC_{PDI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SS_p / P \rceil + N_j * [(h_{iden} + 1) + (SS_t / P)].$$

when SS_p is an s-expression of path dictionary for the predicate class, SS_t is an s-expression of path dictionary for the target class. N_j is the number of qualified join objects of the join class. h_{iden} is the height of the identity index -1; of the join class.
- *The target class is a descendant class of the join class*

We can use the same formula above. Furthermore, we can use the forward traversal from the objects of the join class to the target objects of the target class. However, if the distance between the join class and the target class is high, we should use the identity index of the join class to retrieve the qualified s-expressions to access the target objects.

6.2.1.2 Branch Index

The retrieval cost of the branch index consists of the following:
- Cost of the attribute index scanning.
- Cost of the accessing the target objects from the target class for the qualified entries.

$$RC_{BI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{Bi} / P \rceil.$$

when h_{attr} is the height of the attribute index-1, XP_{attr} is the length of a leaf node index record, $N_{P/Q}$ is the number of the qualified entries of a branch B_i for the predicate P of query Q.

6.2.1.3 Comparison

We use the same parameters and assumptions as we use in evaluating the storage cost. In this subsection, we assume that the predicate is specified on the indexed attribute of the *Computer* class and the target is the *Bank* class. For the branch index, we can scan the attribute index to obtain the qualified entries of the branch for that predicate and locate the target objects from those entries of the branch. Although the query in this case requires only one branch for the branch index, it requires two path dictionaries to solve the query. The first path dictionary is from the *Computer* class to the *Person* class. After obtaining the qualified objects from the class *Person*, we use them to scan the identity index of the path dictionary from the *Person* class to the *Bank* class to obtain the qualified s-expressions and retrieve the qualified objects from the *Bank* class. We can see that the retrieval cost, in this case, of the branch index is much lower than that of the path dictionary index because it is unnecessary to traverse between paths. However, when the predicate class and the target class are on the same path dictionary, the retrieval cost of the branch index and the path dictionary is so close.

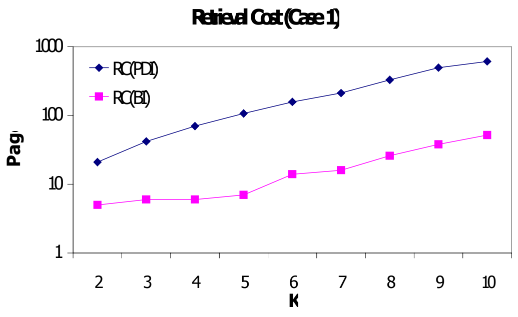


Figure 13. Retrieval cost of the path dictionary index and branch index when the predicate class and the target class are on the same branch.

6.2.2 The predicate and target class on different branches

In this case, the predicate class and the target class are on different branches for the branch index. For the path dictionary index, the predicate class and the target class are apparently on different path dictionaries.

6.2.2.1 Path Dictionary Index

The retrieval cost of the path dictionary index is the same as in Section 6.2.1.1 of Case 2. However, the additional cost may be occurred when the predicate class and the target class on the path dictionaries are far away. The general formula for the retrieval cost when the predicate class is on a path dictionary *j* and the target class is on a path dictionary *k* is:

$$RC_{PDI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SS_p / P \rceil + \sum_{l=j}^{k-1} [N_{jl,\,l+1} * (h_{idenl} + 1) + N_{jl,\,l+1} * \lceil SS_{l+1} / P \rceil].$$

when $N_{jl,\,l+1}$ are the qualified objects of the join class that link between the path dictionary *l* and the path dictionary *l+1* and there are several path dictionaries between the path *j* and path *k*.

6.2.2.2 Branch Index

The retrieval cost of the branch index can be classified on the location of the predicate class and the target class.

- *The predicate class is on an ancestor branch of the target class*

There is no information of the child branch stored in the parent branch. Therefore, after scanning the attribute index and obtain the qualified join objects from the join class, we have to use the identity index of the join class to retrieve the qualified entries of the child branch. The general formula for the branch index when the predicate is on a branch *j* and the target class is on a branch *k* is:

$$RC_{BI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{Bi} / P \rceil + \sum_{l=j}^{k-1} [N_{jl,\,l+1} * (h_{idenl} + 1) + N_{jl,\,l+1} * \lceil SE_{Bl+1} / P \rceil].$$

when $N_{jl,\,l+1}$ are the qualified objects of the join class that link between the branch *l* and the branch *l+1* and there are several branches between the branch *j* and branch *k*.

- *The target class is on an ancestor branch of the predicate class*

Because the information of the child branch can link directly to its parent branch, the retrieval cost in this case is:

$$RC_{BI} = h_{attr} + \lceil XP_{attr} / P \rceil + N_{P/Q} * \lceil SE_{Bi} / P \rceil + \sum_{l=j}^{k-1} N_{jl,\,l+1} * \lceil SE_{Bl+1} / P \rceil.$$

6.2.2.3 Comparison

In this subsection, we classify the location of the predicate class and the target class on different branches. In Figure 14, we assume that the predicate is specified on the indexed attribute of the *Engine* class of the main branch and the target is the *University* class, the child branch of the main branch. We also notice the impact of *K* for the retrieval cost for both path dictionary index and branch index. We can see that the retrieval cost for both access methods in this case are nearly the same. The retrieval cost of the path dictionary index is higher because it must traverse more than two path dictionaries from the *Engine* class to the *University* class.

However, when the predicate is specified on the indexed attribute of the *University* class and the target is the *Engine* class as shown in Figure 15, the retrieval cost of the branch index is much lower than that of the path dictionary index. That is because the child branch of the branch index stores the information of its parent objects and the associated entries so that the retrieval of the entries of the parent branch can be easily performed.

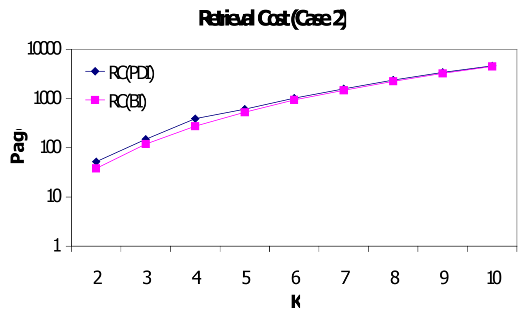


Figure 14. Retrieval cost of the path dictionary index and branch index when the predicate class is on the parent branch of the target class.

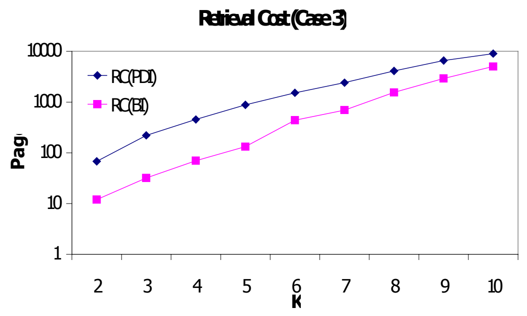


Figure 15. Retrieval cost of the path dictionary index and branch index when the predicate class is on the child branch of the target class.

6.3 Update Cost

We will formulate the update cost as discussed in Section 5.2. To simplify the analysis, we do not include the cost due to page overflow caused by update operation. When a complex attribute of one object is updated, the possible result is as follows.

- Update the reference on the same branch.
- Update the reference on different branches.

6.3.1 Update the reference on the same branch

In this case, we consider the update of the reference on the same branch of the branch index.

6.3.1.1 Path Dictionary Index

When a complex attribute of one object is updated, Two different cases are categorized as follows.

A. The class of the updated object or its ancestor classes have no attribute index

In this case, the update will be performed to the reference between objects. We can use the identity index of the old and new child objects to retrieve the qualified s-expressions and then update the information in the s-expressions. Finally, update of the identity index for the updated object and its ancestor objects have to be performed. We assume that the updated object is on the m^{th} class of the path dictionary.

$$UC_{PDI} = 2*(h_{iden}+1+2*\lceil SS/P\rceil)+(\sum_{l=1}^{m-1}\prod_{j=l}^{m-1}k_{i,j}+1)*(h_{iden}+2).$$

when h_{iden} is the height of the identity index - 1.

B. The class of the updated object or its ancestor classes have an attribute index

In addition to all terms in previously cost model, cost for update attribute index should be considered.

$$UC_{PDI} = 2 * (h_{iden} + 1 + 2 * \lceil SS / P \rceil) + (\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1) * (h_{iden} + 2) + (h_{attr} + 2 * \lceil XP_{attr} / P \rceil).$$

when h_{iden} is the height of the identity index -1and h_{attr} is the height of the attribute index -1.

6.3.1.2 Branch Index

The update of the branch index is more complicated than that of the path dictionary index because more information is stored in the entry of the branch. Four different cases are categorized as follows.

A. The class of the updated object or its ancestor classes have no attribute index, no leaf branch and no child branch

In this case, the formula is similar to that of the path dictionary index in Section 6.3.1.1 case A. Also, We assume that the updated object is on the m^{th} class of branch *i*.

$$UC_{BI} = 2 * (h_{iden} + 1 + 2 * \lceil SE_{Bi} / P \rceil) + (\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1) * (h_{iden} + 2).$$

when h_{iden} is the height of the identity index - 1.

B. The class of the updated object or its ancestor classes have an attribute index but no leaf branch and no child branch

$$UC_{BI} = 2 * (h_{iden} + 1 + 2 * \lceil SE_{Bi} / P \rceil) + (\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1) * (h_{iden} + 2) + (h_{attr} + 2\lceil XP_{attr} / P \rceil).$$

C. The class of the updated object or its ancestor classes have an attribute index and leaf branches but no child branch.

The number of objects for the leaf branches from objects of the first class to the updated objects is:

$$NLO = [\sum_{j=1}^{m-1} (nlb_{i,j} * \prod_{l=j}^{m-1} k_{i,l}) + nlb_{i,m}].$$

Therefore, the update cost is:

$$UC_{BI} = 2 * (h_{iden} + 1 + 2 * \lceil SE_{Bi} / P \rceil) + (\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 + NLO) * (h_{iden} + 2) +$$
$$(h_{attr} + 2 * \lceil XP_{attr} / P \rceil).$$

D. The class of the updated object or its ancestor classes have an attribute index, leaf branches and child branches.

We will use some parameters defined earlier. So the update cost is:

$$UC_{BI} = 2 * (h_{iden} + 1 + 2 * \lceil SE_{Bi} / P \rceil) + (\sum_{l=1}^{m-1} \prod_{j=l}^{m-1} k_{i,j} + 1 + NLO) * (h_{iden} + 2) +$$
$$(h_{attr} + 2 * \lceil XP_{attr} / P \rceil) + \sum_{l=1}^{m} \prod_{j=1}^{ncb_{i,l}} (2 * \lceil SCB_{l,j} / P \rceil).$$

when $ncb_{i,l}$ is the number of child branches of a branch *i* of class *l* and $SCB_{l,j}$ is the entry size of a child branch *j* of class *l*.

6.3.1.3 Comparison

We use the same parameters and assumptions as used in the storage cost and retrieval cost. In this subsection, we assume that one object of the *Person* class changes its reference to another object of class *Vehicle*. We can see that the update cost of the branch index is more than that of the path dictionary index because the information stored in an entry of branch index is more than that stored in s-expression of path dictionary index. However, the small increased update cost of the branch index is likely acceptable when we consider the gain from the retrieval.

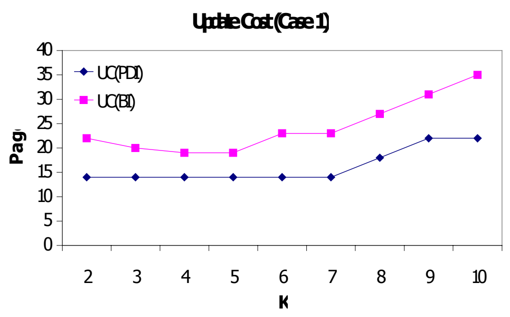


Figure 16. Update cost of the path dictionary index and branch index when the predicate class and the target class are on the same branch.

6.3.2 Update the reference to the different branch

6.3.2.1 Path Dictionary Index

It makes no different for the path dictionary index. Therefore, the update cost of the path dictionary is the same as in Section 6.3.1.1.

6.3.2.2 Branch Index

The update of the branch index is performed only on the object of the first class of the child branch. The parent objects and associated pointers will be updated for the branch information of the child branch. Therefore, the update cost consists of the following:

- The scanning of the identity index for the old and new OID of object of the first class of the child branch.
- The update of the qualified entries of the child branch.
- The update of the identity index of the parent object.

$$UC_{BI} = 2 * (h_{iden1} + 1 + 2 * \lceil SE_{Bj} / P \rceil) + (h_{iden2} + 2).$$

when SE_{Bj} is an entry size of a branch *j*; the child branch of a branch *i*.

h_{iden1} is the height of the identity index - 1; of the first class of the branch *j*

h_{iden2} is the height of the identity index - 1; of the parent class of the branch *j*

6.3.2.3 Comparison

In this subsection, we assume that an object in the *Person* class changes its reference to another object of the *Course* class. We can see that the associated entries of the child branch are updated. Also, the identity index of the updated object is updated. Therefore, the update cost for the branch index in this case is similar to that of the path dictionary index.

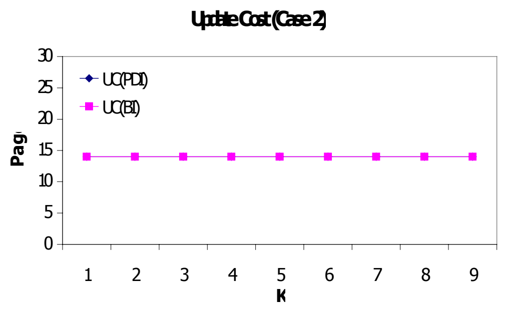


Figure 17. Update cost of the path dictionary index and branch index when the predicate class and the target class are on different branches.

7. Conclusion

The aggregation hierarchy as a tree is considered as a more complicated form than a path. Therefore, the efficient access method to handle the query on the aggregation hierarchy should be developed. We introduce the new access method called the branch index to evaluate all kinds of query on an aggregation hierarchy as a tree and then compared it with the path dictionary index method for the path scheme. We varied the reference sharing of classes and shared key values to observe the performance and the storage overhead of the path dictionary index and branch index.

We create path dictionaries to mimic our branch for comparison. The attribute indexes are created on all leaf classes of the aggregation hierarchy. The identity indexes are also created for the objects of every class on a branch of the branch index and on a path of the path dictionary index for complex attribute searching. The result is that the storage overhead of the branch index is much lower than cost of the path dictionary index because we can store the information of the leaf branch as part of the complete branch so that the redundant path is eliminated. However, the entry size of a branch may be bigger than the s-expression of a compared path dictionary because more information, such as the leaf branch and the associated parent branch, is stored in a branch. Therefore, the update cost of the branch index is a little higher than the update cost of the path dictionary index. However, we gain the benefit from the organization of the branch index so that its retrieval cost is much lower than path dictionary index cost. We can conclude that the branch index is more appropriate for the aggregation hierarchy as a tree than the path dictionary index, especially when the retrieval operation is high. The branch index can be reduced to the form of the path dictionary index when there is only one path of the aggregation hierarchy and then all cost will be the same. Therefore, the branch index is more general than the path dictionary index.

Throughout this paper, we have some assumptions that limit the general case for the access method. The multi value attribute and complex query are a challenge one for the next research of this area. Furthermore, the most complicated form of the aggregation hierarchy as a graph should be in consideration in the future.

References

[1] E. Bertino and W. Kim, "Indexing Technique for Queries on Nested Objects," *IEEE Trans. on Knowledge and Data Eng*., vol. 1, pp. 196-214, 1989.

[2] H.–S. Young, S. Lee, and H.–J. Kim, "Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases*," Proc. 10th Int'l Conf. on Data Eng*., pp. 518-525, 1994.

[3] K.A. Hua and C. Tripathy, "Object Skeleton: An Efficient navigation Structure for Object-Oriented Database System," *Proc. 10th Int'l Conf. on Data Eng*., pp. 508-517, 1994.

[4] E. Bertino and P. Foscoli, "Index Organizations for Object-Oriented Database System," *IEEE Trans. on Knowledge and Data Eng*., vol. 7, pp. 193-209, 1995.

[5] Y.–H. Chen and A.J.T. Chang, "Object Signatures for Supporting Efficient Navigation in Object-Oriented Databases," *Proc. 8th Int'l Workshop on Database and Expert System Application*, pp. 502-507, 1997.

[6] H. Shin and J. Chang, "A New Signature Scheme for Query Processing in Object-Oriented Database," *Proc. 20th Int'l Conf. on Computer Software and Applications*", pp. 400-405, 1996.

[7] D.L. Lee and W.-C. Lee, "Signature Path Dictionary for Nested Object Query Processing," *Proc. 15th Int'l Conf. on Computers and Communications*, pp. 275-281, 1996.

[8] E. Gudes, "A Uniform Indexing Scheme for Object-Oriented Databases*," Proc. 12th Int'l Conf. on Data Eng*., pp. 238-246, 1996.

[9] S. Choenni, E. Bertino, H.M. Blahken and T. Chang, "On the Selection of Optimal Index Configuration in OO Databases," *Proc. 10th Int'l Conf. on Data Eng*., pp. 526-537, 1994.

[10] S.Y. Sung and J. Fu, "Access Methods on Aggregation of Object-Oriented Database," *Proc. Int'l Conf. on Systems, Man and Cybernetics*, vol. 2, pp. 977-982, 1996.

[11] W. Kim, "Object-Oriented Databases: Definition and Research Directions," *IEEE Trans. on Knowledge and Data Eng*.", vol. 2, No.3, pp. 327-341, 1990.

[12] F. Fotouhi, T.-G. Lee and W.I. Grosky, "The Generalized Index Model for Object-Oriented Database Systems," *Proc. 10th Phoenix Conf. on Computer and Communication*, pp. 302-308, 1991.

[13] E. Bertino and C. Guglielmina, "Optimization of Object-Oriented Queries Using Path Indices," *Proc. 2nd Int'l Workshop Research Issues on Data Eng*., pp. 140-149, 1992.

[14] B. Shidlovsky and E. Bertino, "A Graph-Theoretic to Indexing in Object-Oriented Databases*," Proc. 12th Int'l Conf. on Data Eng*., pp. 230-237, 1996.

[15] W.-S. Cho, S.-S. Lee and Y.-I. Yoon, "A Join Algorithm Utilizing Multiple Path Indexes in Object-Oriented Database Systems," *Proc. 2nd Int'l Conf. on Eng. of Complex Systems*, pp. 376-382, 1996.

[16] S.K. Seo and Y.J. Lee, "Optimal Configuration of Nested Attribute Indexes in Object-Oriented Databases," *Proc. 20th EUROMICRO Conf. on System Architecture and Integration*, pp. 379-386, 1994.

[17] W.-C. Lee and D.L. Lee, "Short Cuts for Traversals in Object-Oriented Database Systems," *Proc. Int'l Computer Symposium*, pp. 1172-1177, 1994.

[18] W.-C. Lee and D.L. Lee, "Combining indexing Technique with Path Dictionary for Nested Object Queries*," Proc. 4th Int'l Conf. on Database Systems for Advanced Applications*, pp. 107-114, 1995.

[19] W.-C. Lee and D.L. Lee, "Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases," *IEEE Trans. on Knowledge and Data Eng*., vol. 10, No.3, pp. 371-388, 1998.

[20] Y. Ishikawa and H. Kitagawa, "Analysis of Indexing Schemes to Support Set Retrieval of Nested Objects," *Proc. Int'l Symposium on Advanced Database Technologies and Their Integration*, 1994.

[21] D.L. Lee and W.-C. Lee, "Using path Information for a Query Processing in Object-Oriented Database Systems," *Proc. Conf. on Information and Knowledge Management*, pp. 64-71, 1994.

[22] W.-C. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," *Proc. 2nd Int'l Computer Science Conf.*, pp. 616-622, 1992.

[23] P. Mahatthanapiwat and W. Rivepiboon, "Direct Access to Terminal Virtual Path in OODB," *Proc. National Computer Science and Engineering*, 1999.

[24] P. Mahatthanapiwat and W. Rivepiboon, "Virtual Path Signature: An approach for Flexible Searching in OODB, " *Proc. Int'l Conf. on Intelligent Technology*, pp. 335-340, 2000.