1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

THE STUDY OF INDEXING TECHNIQUES ON OBJECT ORIENTED DATABASES

Yin-Fu Huang and Jau-Min Chen

Institute of Electronics and Information Engineering
National Yunlin University of Science and Technology
Touliu, Yunlin, Taiwan 640, R.O.C.
Email: huangyf@el.yuntech.edu.tw

ABSTRACT

The object-oriented database (**OODB**) has been becoming more important in the recent years. It can deal with a large amount of complex objects and relationships that relational database systems can not handle well. However the retrieval and update performance of an OODB depends on indexing techniques. In this paper, we study the indexing techniques on object-oriented databases, based on the inheritance hierarchy and aggregation hierarchy. Given the access probability and the size of each class, we propose a cost function to evaluate the gain of building an index on the inheritance hierarchy. For the aggregation hierarchy, we use a path-catenation technique to evaluate how to build index files on classes. Through the experiments, we found our methods have better retrieval performance than most ones proposed before.

1. INTRODUCTION

The object-oriented database (**OODB**) has been becoming more important in the recent years. This is because the relational database (**RDB**) did not work well in these areas, such as computer-aided design and manufacturing (**CAD/CAM**), computer-integrated manufacturing (**CIM**), computer-aided software engineering (**CASE**), geographic information systems (**GIS**) etc. In general, there are a large amount of complex objects and relationships between them in these applications. An OODB is equipped with the techniques to express complex objects and relationships between them. To improve the performance of an OODB, many technologies have been proposed since then. For examples, if several related objects are often accessed together, we should place them in the neighbor place to reduce the I/O access. Indexing is one of common techniques to speed up the query on the database. There are two types of object indexing in an OODB; that is **class indexing** and **nested object indexing**. The both corresponding hierarchies are inheritance one and aggregation one, respectively. Class indexing is used when a user query references a single class or multiple classes rooted from some specified class in the inheritance hierarchy. Besides nested object indexing is used when a user query references a specific class via a reference path defined in the aggregation hierarchy.

In the inheritance hierarchy, Class Hierarchy Tree (**CH-tree**) is a well-known indexing technique. CH-tree based on **B^+ tree** concepts builds a single B^+ tree on the collection of all classes in the inheritance hierarchy. Another indexing technique called **H-tree** is to build a B^+ tree on per class in the inheritance hierarchy, and these indexes are nested with a superclass-subclass relationship. An L pointer is used for nesting and expresses the inheritance relationships between classes. In addition, **hcC-tree** and **CG-tree** are two similar indexing techniques. hcC-tree builds a class chain and a hierarchy chain. The class-chain is built according to each class and used to improve the single query that performs worst on CH-tree. The hierarchy-chain is like CH-tree. CG-tree is an index file grouped by set. It has a special directory page used to improve the range query. Finally, **Class-Division (CD) method** is a practical alternative to CH-tree. It is to find what classes should be combined in an index file (like CH-tree structure). CD method includes an index file built on the root of the inheritance hierarchy (i.e. CH-tree) and other index files for the other classes. Point queries involving more than one class can be handled by CH-tree, whereas the other index files can be used for querying extents or speeding-up the range query.

In the aggregation hierarchy, there are **multi-index**, **nested index**, **path index**, and **nested-inherited index** (**NIX**) methods to speed up the search [5]. Multi-index records all paths from one class to the other classes. Nested index records the first instance of all paths. Path index records all instances of all paths. They have more overheads for update operations. Nested-inherited index is an index structure integrating the aggregation and inheritance relationships. It combines the attribute indexes and auxiliary indexes, but also has more overheads for update operations.

Since the instances and access probability of each class in an OODB are not the same, we propose a cost **function** for the inheritance hierarchy and a path catenation method for the aggregation hierarchy, and then build the index files to improve the performance of the OODB. In the remainder of the paper, it is organized as follows. In Section 2, we review the related researches. The proposed concepts and algorithms are described in

Section 3. In Section 4, a few experiments are done to show the results of our methods and other methods. Finally Section 5 makes a conclusion.

2. PREVIOUS WORKS

Till now, several indexing techniques on object-oriented databases have been proposed. They can be classified into two types of index structures, i.e. inheritance hierarchy and aggregation hierarchy. We will investigate them in the following sections. Then an alternative skill of CH-tree called CD-method is also discussed.

2.1 Inheritance Hierarchy Index

For a search predicate on an attribute of a specified class, we may be against the single class or classes in the inheritance hierarchy that is rooted on the specified class. Here five indexing techniques for the inheritance hierarchy are presented. The first one called the single-class index is maintained by indexing against an attribute of a single class. It is based on B^+-tree and implemented in the object-oriented database ORION [7]. The second one called the CH-tree index is a well-known indexing technique on the inheritance hierarchy [7]. It maintains only one index tree on an attribute for all classes in the inheritance hierarchy. It is also based on B^+-tree and the format of the non-leaf node is similar to the single-class index. The third one called the H-tree maintains one B^+-tree for each class, and these B^+-trees are nested according to their superclass-subclass relationships [8]. The H-tree has good performance to retrieve data on a signal class, but the storage overhead on L pointers and the checking of L pointers against queries should be improved. The fourth one called the hcC-tree (hierarchy class Chain) uses two types of chains such as hierarchy-chains and class-chains to store information [10]. Like the CH-tree, the hierarchy chain is well used to query the inheritance hierarchy, and the class-chain improves the CH-tree on its worst performance for a single class range query. But it must store duplicate oids on the two types of chains. The last one called the CG-tree is a set grouping index [6]. The CG-tree improves the range query like hcC-tree, but no duplicate oids are required to store.

2.2 Aggregation Hierarchy Index

In object-oriented databases, a search predicate for a query could be a path expression where an index structure associates the values of nested attributes with the objects in the leading classes of the path expression. Here four indexing techniques for the aggregation hierarchy are presented.

The first one is the multi-index that defines n indexes to retrieve the objects with the nested attribute A_n where the first index I_1 is defined on $C_1.A_1$, and the i-th index I_i is defined on $C_i.A_i$ ($1 \leq i \leq n$) [1] [4]. It has less update cost because only two simple indexes I_i and I_{i+1} are required to update if an attribute A_i is changed on the path. However the retrieval cost is enormous due to using multiple indexes.

The second one is the nested index that allocates an index for the specified nested attribute and the first class in the path. For a path $P=C_1.A_1.A_2...A_n$, the form of its nested index is (A_n, the list of C_1's oids). It can provide better performance than other indexing techniques, but when updating an object in the path, we must use forward and backward traversal to find the corresponding object; the backward traversal is very expensive because there are no inverse references between objects.

The third one is the path index that is based on a single index [3] [4]. Unlike the nested index, it does not need the backward traversal. For a key value, the path index stores all path instantiation, including complete (i.e. beginning at the first class) or partial (i.e. beginning at the second class or lower) paths. Thus it is suitable for retrieval queries because the leading class of the specified predicate could be any class in the aggregation hierarchy, but its update cost is very high.

The last one is the nested-inherited index (NIX) that can support inheritance hierarchy and aggregation hierarchy [2]. It has two organizations, i.e. primary index and auxiliary index based on B^+-tree. The primary index maintains the index on the nested attribute A_n. If j is one value of A_n, it stores the oids of all classes, which have the value j in A_n. The auxiliary index uses oids to be the key value, and stores the list of the oids of the parent class along the aggregation hierarchy. The entries of leaf nodes in the primary index contain pointers to the leaf nodes in the auxiliary index, and vice versa. The NIX has good retrieval performance, but has high update cost due to modifying the two indexes.

2.3 Class-Division Method (CD)

The Class-division method (CD) is properly viewed as an extension of CH-tree [9]. Like CH-tree, it contains an index built on root and includes all classes in the inheritance hierarchy. Besides there are some smaller indexes provided to query extents or speed-up range queries. The CD method uses Class-Division Algorithm to generate the smaller indexes that can be combined later for a query. Two heuristic procedures (i.e. Prune-Space and Rake-Contact) are used to combine the smaller indexes to retrieve the target objects of a class. As a result, the CD method avoids the overhead of CH-tree by skipping the unnecessary data when querying a class in the inheritance hierarchy.

3. OUR INDEXING METHODS

In this section, we propose the indexing methods for both inheritance hierarchy and aggregation hierarchy, based on considering the access probability and the size of each class.

3.1 Inheritance Hierarchy

CH-tree is widely used in practice and seems simpler (i.e. only B^+-tree used) than other structures. The major drawback of CH-tree is that it must filter a large amount of unnecessary data, especially in a query with the target class on the lower level in the inheritance hierarchy. The CD method mentioned in Section 2.3 is an alternative way to avoid filtering huge unnecessary data. Besides building an index on root, the CD method also builds a few small indexes using Class-Division Algorithm. However, the CD method does not consider the influence of the access probability and the size of each class. We expect that our method based on the two factors will improve the retrieval performance of an OODB. Besides the CD method can not deal with the multiple inheritance that can be handled in our method.

Here are the assumptions for the classes in the inheritance hierarchy [1].

1. All key values have the same length.
2. The key values for all classes are totally inclusive.
3. The cardinality of a class in the inheritance hierarchy is independent of the cardinality of any of its super-classes or subclasses.
4. Each class has its own access probability and instantiations.

3.1.1 Basic Concepts and Algorithms

For an inheritance hierarchy where the access probability and the size of each class is given, we try to build index files such that the total file number and the accessed block number when querying a class in the inheritance hierarchy, are minimized as possible as we can. In general, a query with a higher-probability target class should have the less accessed block if we have a good design for the index files. Then the retrieval performance of database systems will be improved.

Like CD method, we also use small indexes. Each small index is applied to one class and its subordinate classes (i.e. subclasses). Which small indexes are built is depended on the access probabilities and the sizes of the class and its subclasses. Here we have a cost function to determine the gain (or weight) to build a small index on a class.

$$G(i) = P_i - \sum_{\substack{C_j \in subclass(C_i) \\ and C_j \notin subclass(C_k)}} p_j * \frac{S_i - S_j}{S_i}$$

Within the cost function, $G(i)$ is the gain to build a small index on class i, P_i is the access probability of class i in a query, class k is one on which an index has been built, and S_i is the sum of all the instances in the sub-tree rooted at class i, excluding the subclasses of class k.

On the right hand side of the formula, P_i (or P_i*1) implies the "clean" gain when a small index is built on class i, and no unnecessary data are filtered. However it is impossible unless class i is a leaf node in the inheritance hierarchy. Thus we must consider how many data should be filtered, if an index is built on class i and the subclass of class i is the target class in a query. Here the number of filtered data is expressed as a ratio in the formula. The classes that are subclasses of class i and not subclasses of some class k on which an index has been built, should be considered and summed up as a negative item against the "clean" gain.

In general, we think that the more index files built, the better performance on retrieving target instances. However it is not true since more index files might be accessed when we query a target class, not to mention the storage cost and update cost. Therefore we can not build the index files as many as we can. Here we sort the gains of all classes $G(i)$ and build the index files class-by-class beginning from the class with the largest gain. If the gain value of a class is positive and none of its ancestors have been built, we think building an index file on the class would be beneficial for the system. The building is kept on going till the root class is reached or the gain of the current class is negative. In our method, an index file is always built on the root class like CH-tree.

The algorithm to build small index files is shown as follows:

```
procedure Build_Index_File
/* Input: the inheritance hierarchy where its classes are
          denoted by $C_1, C_2, ..., C_n$ ($C_1$ is a root), the
          probability of each class, and the size of each
          class
   Output: the classes with index files built */
    for ($i$ := 1; $i \le n$; $i$++) do
        evaluate $G(i)$;
    end for;
    sort $G(i)$;
    choose the class with the largest gain;
    while (the current class is not a root and $G(i) > 0$)
        if none of its ancestors have been built  then
            build an index file on class $i$;
            adjust the remaining gain values and
            resort them;
        end if;
        choose the next class;
    end while;
    build an index file on the root class;
end procedure Build_Index_File;
```

3.1.2 Examples

Here we take an example to explain our method. As shown in Figure 1, we have an inheritance hierarchy with nine classes and the links from the lower-level classes to the upper-level classes indicate the inheritance relationships. Around each node, we label its access probability and instance number.

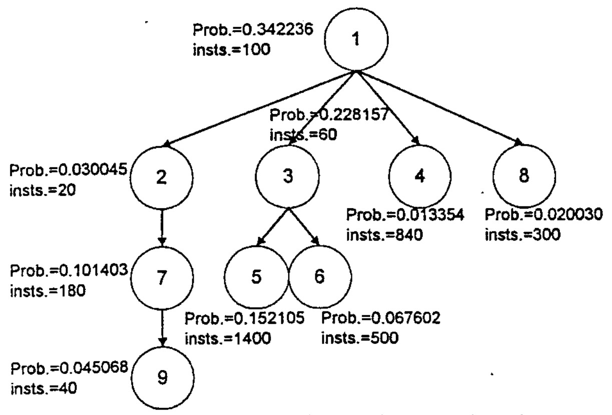

Figure 1 The inheritance hierarchy rooted at class 1

After the evaluation on *G(i)*, we sort them as shown in Table I. In the first iteration, an index file is built on Class 5 since none of its ancestors have been built. After building the index file, we adjust the remaining gain values (i.e. Class 3 and 1) and resort them as shown in Table II. In the second iteration, an index file is built on Class 3. At this iteration, only the gain of Class 1 is adjusted and resorted as shown in Table III. In the third iteration, we have Class 1 with the largest gain value, but it is also a root class. Thus we build an index file on Class 1 and terminate the algorithm. In summary, we have built three index files on Class 5, 3, and 1.

Table I The gain order after sorting

Class	5	3	6	7	9
Gain	0.152105	0.134342	0.067602	0.064529	0.045068
Class	8	4	2	1	
Gain	0.020030	0.013354	-0.015962	-0.099690	

Table II The gain order after building an index on Class 5

Class	5*	3	6	7	9
Gain	0.152105	0.220914	0.067602	0.064529	0.045068
Class	8	4	2	1	
Gain	0.020030	0.013354	-0.015962	-0.060425	

Table III The gain order after building an index on Class 3

Class	5*	3*	1	6	7
Gain	0.152105	0.220914	0.165138	0.067602	0.064529
Class	9	8	4	2	
Gain	0.045068	0.020030	0.013354	-0.015962	

3.2 Aggregation Hierarchy

As mentioned in Section 2.2, the path index is a popular index structure and is suitable for retrieval queries because the leading class of the specified predicate could be any class in the aggregation hierarchy. However, it always has a lot of redundant space because many object identifiers are recorded for a long path. Besides, it does not consider the influence of the access probability of each class. Here we reduce the length of the object identifier list and get better retrieval performance by splitting a long path into several sub-paths according to the access probability of each class.

Here are the assumptions for the classes in the aggregation hierarchy.

1. All key values have the same length.
2. All attributes are single-valued.
3. There are no partial instantiations.
4. The key values of the instances of a class are uniform-distributed.
5. Each class has its own access probability.

3.2.1 Basic Concepts and Algorithms

Basically our method is a variation of the path index. Instead of building a long path, we build several sub-paths to reduce the redundant space. Besides our method can handle the aggregation hierarchy with the multiple references on a class, where other methods can not be applied. Like the path index, given a nested attribute, users can access the target class following the reference path in the aggregation hierarchy. As shown in Figure 2, when users want to access Class **Vehicle** with a nested attribute **Country.name,** two reference paths such as **Vehicle.Assuror.City. Country.name** and **Vehicle.Manufacturer.Division.City.Country.name** should be built in advance. However these two paths have the common sub-path **City.Country.name.** In our method, the common sub-path is not necessarily built twice. Here we first find how many path catenations of a path could be generated, and then apply a cost function to evaluate which path catenation is the best for the retrieval performance.

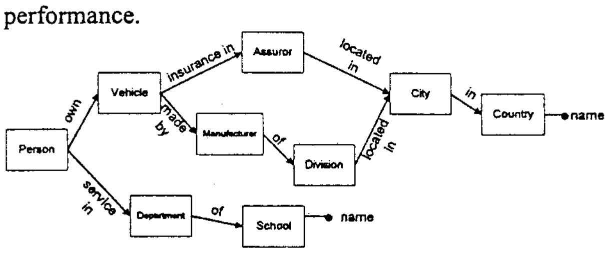

Figure 2 The aggregation hierarchy

In our method, we build an optimal path catenation for each path based on the access probabilities. First, for each target class C_i, we find all paths from class C_i to nested attribute A_n and put them into the set **possible_path**. Second, to reduce the redundant space, the paths that are sub-paths of another paths are removed from **possible_path**. Third, for each path P_i in **possible_path**, we include the classes in the set **split_node**, which are 1) the target class nodes on the path P_i, 2) the class nodes with in-degree > 1 or out-degree >1 on the path P_i, and 3) C_{n+1} (C_{n+1} is an alias of A_n). The set **split_node** is used to find all catenations of sub-paths on the path P_i, which will be put into the set **catenation**. For each path catenation of a path P_i, we will calculate its retrieval cost according to the cost function and the access probabilities of all target classes on the path P_i. Finally we find a path catenation with the least retrieval cost for the path P_i.

The cost function used in our algorithm was defined by Bertino and Kim [1], which is the number of index pages accessed. It is expressed as follows:

$$A = h + 1 \quad if \quad XP \le P$$
$$A = h + \lceil XP / P \rceil \quad if \quad XP > P$$

where h is the number of nonleaf nodes accessed (i.e. the height of a B$^+$ tree), P is the page size, and XP is the average length of a leaf-node index record for a path index. As for XP, it can be calculated as follows:

$$XP = PN * UIDL * n + kl + ol \quad if \quad XP \le P$$

$$XP = PN * UIDL * n + kl + ol + DS \quad and$$

$$DS = \lceil [PN * UIDL * n + kl + ol] / P \rceil * (UIDL * n + pp) \quad if \quad XP > P$$

PN is the average number of path instantiations ending with the same value for the nested attribute A_n.

$$PN = \prod_{i=1}^{n} k(i)$$

$UIDL$ is the length of the object-identifier.
kl is the average length of a key value for the nested attribute.
ol is the sum of sizes of the key-length field, the record-length field, and the number oids field.
DS is the length of the directory at the beginning of the record.
pp is the length of a page pointer.

The algorithm to find a path catenation with the least retrieval cost for the path from class C_i to nested attribute A_n is shown as follows:

```
procedure Best_Catenation
/* Input: the aggregation hierarchy where its classes are
          denoted by $C_1$, $C_2$, ..., $C_n$ ($C_1$ is a root), nested
          attribute $A_n$, and the probability $Prob_i$ of each
          class
   Output: catenations of sub-paths with the least retrieval
          cost */
    for each class $C_i$ with probability $\ne 0$ do
        possible_path $\leftarrow$ all paths from class $C_i$ to
                                   nested attribute $A_n$;
    end for;
    remove the paths that are sub-paths of another paths
    from possible_path;
    for each path $P_i$ in possible_path do
        split_node $\leftarrow$ all class nodes with probability
                         $\ne 0$ on the path $P_i$
               $\cup$ all class nodes with in-degree > 1 or
               out-degree >1 on the path $P_I$
               $\cup$ $C_{n+1}$ ($C_{n+1}$ is an alias of $A_n$);
        catenation $\leftarrow$ all catenations of sub-paths
                         based on split_node on the
                         path $P_i$;
        for each catenation $Cat_j$ in catenation do
            retrieval_cost$_j$ = 0;
            for each class $C_i$ with probability $\ne 0$ do
            if $C_i$ on this path then
                cost $\leftarrow$ evaluate cost according to
                          the cost function
                          + no. of accessed index files
                          * average seek time;
                retrieval_cost$_j$ = retrieval_cost$_j$
                                  + cost * Prob$_i$;
            end if;
            end for;
        end for;
        find a path catenation with the least retrieval cost
        for the path $P_i$;
    end for;
end procedure Best_Catenation;
```

3.2.2 Examples

Here we take an example to explain our method. As shown in Figure 3, we have an aggregation hierarchy with eight classes and a nested attribute. Around each target node, we label its access probability. The label on each arc indicates the number of instantiations in the predecessor class for one instantiation in the successor class. For example, the number of instantiations in class 1 is 50*100*50*5 for one value of the nested attribute. Thus the instantiations in each class can be calculated using the formula PN. Besides we list the parameter values used in the cost function, as shown in Table IV.

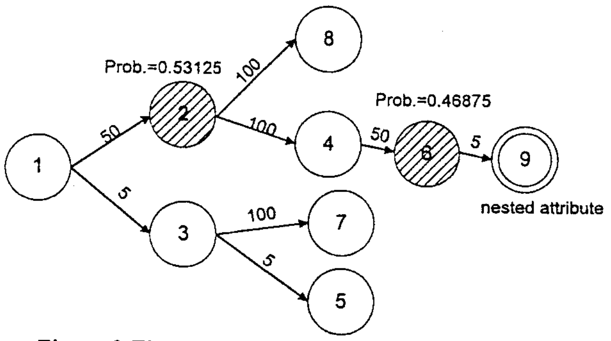

Figure 3 The aggregation hierarchy rooted at class 1

Table IV The parameter values

$UIDL$=8	kl=8	ol=6	pp=4	P=4096

For the target classes C_2 and C_6, we have the paths $C_2C_4C_6A$ and C_6A. Since C_6A is a sub-path of $C_2C_4C_6A$, the path C_6A is removed. Next we try to find the best path catenation for the path $C_2C_4C_6A$. After finding the split nodes C_2, C_6, and A (or C_9), we have two path catenations; that is ($C_2C_4C_6$, C_6C_9) and $C_2C_4C_6C_9$. For the path catenation ($C_2C_4C_6$, C_6C_9), we calculate the retrieval cost on the target class C_2 based on two path index files $C_2C_4C_6$ and C_6C_9. The retrieval cost is 33.6 ms. Next we calculate the retrieval cost on the target class C_6 based on the path index file C_6C_9. Totally we have the retrieval cost 39.85 ms for the path catenation ($C_2C_4C_6$, C_6C_9). For another path catenation $C_2C_4C_6C_9$, we also calculate the total retrieval cost 59.56 ms based on the path index file $C_2C_4C_6C_9$. The retrieval cost of each path catenation is shown in Table V. Finally we find that the path catenation ($C_2C_4C_6$, C_6C_9) has the least retrieval cost; thus two path index files $C_2C_4C_6$ and C_6C_9 is built for the path $C_2C_4C_6C_9$.

Table V Retrieval cost for each path catenation

Path catenation	Target class	Retrieval cost	Total cost
($C_2C_4C_6$, C_6C_9)	C_2	33.66 ms	39.85 ms
	C_6	6.19 ms	
$C_2C_4C_6C_9$	C_2	31.64 ms	59.56 ms
	C_6	27.92 ms	

4. EXPERIMENTS

In this section, we present the experimental results of performance in an inheritance hierarchy and an aggregation hierarchy. The probability and instantiations of each class are considered in our model to reflect the real usage on an OODB. Here we compare the performance of CH-tree, CD method, and our method in the inheritance hierarchy, and the performance of multi-index, nested index, path index, and our method in the aggregation hierarchy.

4.1 Environment of Experiments

Different from the experiments in the previous works, we use disk access time as an evaluated parameter in our experiments. This is because using the number of accessed pages as an evaluated parameter is unfair if these pages are read from different index files. The seek time between these different files should not be ignored, especially because the access time is usually dominated by the seek time. All these experiments are undertaken on a PC Pentium 200 Hz where a hard disk "Seagate ST32122A" is attached. The parameters about the hard disk are external transfer rate 16.6 MB/sec and average seek time 12.5 ms. Based on the parameters, we calculate it takes 0.2353 ms to read a 4K page.

Two different database sizes are given in the experiments for the inheritance hierarchy, and the common ratios of class sizes are SR=1.7 and SR=2.7, respectively. The size of the maximal class is 201 times than that of the minimal one in the smaller database, and the maximal one is 20589 times than the minimal one in the larger database. We assign the size and probability to each class according to SR and PR in Table VI. Ten inheritance hierarchies, each with 10 classes, are generated randomly in both databases. The parameters used in the inheritance hierarchy are shown in Table VI.

In the experiments for the aggregation hierarchy, we have two different class numbers in the hierarchy: one is with 10 classes and another is with 15 classes. The target class numbers are 3 and 4, respectively. The number of instantiations in the predecessor class for one instantiation in the successor class (i.e. K(*i*)) is assigned randomly, as shown in Table VII. Ten aggregation hierarchies are also generated randomly in both experiments. The parameters used in the aggregation hierarchy are shown in Table VII.

Table VI The parameters used in the inheritance hierarchy

Parameters	Definitions and assigned values
N	total number of objects - 85,965 (small) or 3,633,195 (large)
D	total number of distinct key values - 300
C	the length of the control field in the key record of a leaf node - 16
RP	the common ratio of the geometric series in probability - 1.5
RS	the common ratio of the geometric series in size - 1.7 or 2.7
cid	the length of class id - 4
offset	the length of offset - 2
d	the order of an internal node - 146
f	the average fan-out of an internal node on the B^+tree - 218
UIDL	the length of oid - 8
P	the page size - 4096
L	the average length of a non-leaf node index record - 14

Table VII The parameters used in the aggregation hierarchy

Parameters	Definitions and assigned values
K(*i*)	the average number of objects of class *i*, assuming the same value for attribute A*i* - 1 or 5 or 50 or 100 or 1000
pp	the length of a page pointer - 4
kl	the average length of a key value for the nested attribute - 8
ol	the sum of sizes of the key-length field, the record-length field, and the number oids field - 6
d	the order of an internal node - 146
f	the average fan-out of an internal node on the B^+tree - 218
UIDL	the length of oid - 8
P	the page size - 4096
L	the average length of a non-leaf node index record - 14

4.2 Inheritance Hierarchy

The retrieval performance of three indexing techniques is shown in Figure 4(a) and Figure 4(b). When the database size is small, CH-tree and our method have better performance than the CD method, because the latter one opens more index files and requires more seek time than the former ones. When the database size is large, CH-tree is the worst among three indexing techniques, because it reads all instantiations on the inheritance hierarchy and has a lot of filter overheads. Our method still performs better than the CD method, because the number of its opened files is far more than ours.

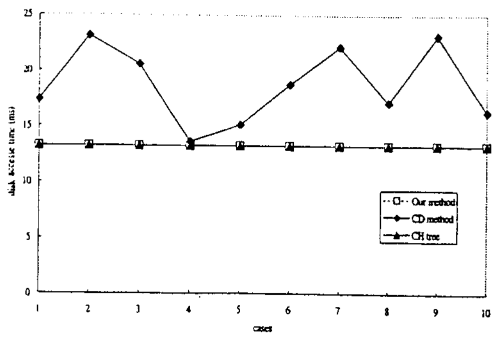

Figure 4(a) Retrieval cost in the inheritance hierarchy with RS=1.7

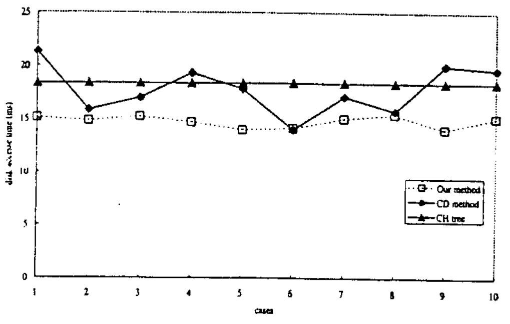

Figure 4(b) Retrieval cost in the inheritance hierarchy with RS=2.7

On the other hand, we also compare the update performance of three indexing techniques, as shown in Figure 5(a) and Figure 5(b). When the database size is small, CH-tree is the best, because it accesses only one index file (i.e. CH-tree structure). Besides our method still performs better than the CD method, because of more seek time required for the CD method. When the database size is large, the same results occur for three indexing techniques.

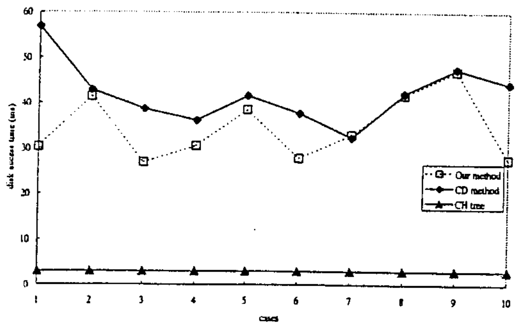

Figure 5(a) Update cost in the inheritance hierarchy with RS=1.7

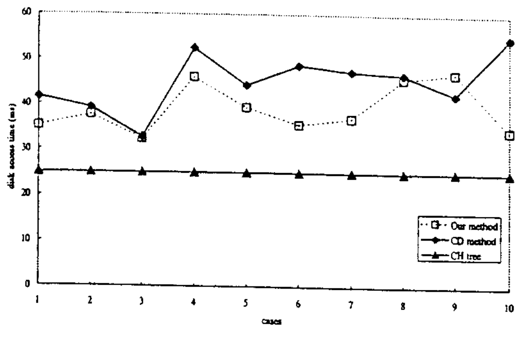

Figure 5(b) Update cost in the inheritance hierarchy with RS=2.7

4.3 Aggregation Hierarchy

The retrieval performance of four indexing techniques is shown in Figure 6(a) and Figure 6(b). The multi-index has the worst retrieval performance because of using multiple index files. The nested index has the best retrieval performance because only the target oids for a key value need to be stored. However since each nested index is only for one class, users can not query other classes using the same nested index file. As for our method, owing to considering the access probabilities of target classes, it is at least better than the path index, and the difference is not small (see the vertical scale in Figure 6). Especially when the reference path is longer, the gain from our method usually increases more.

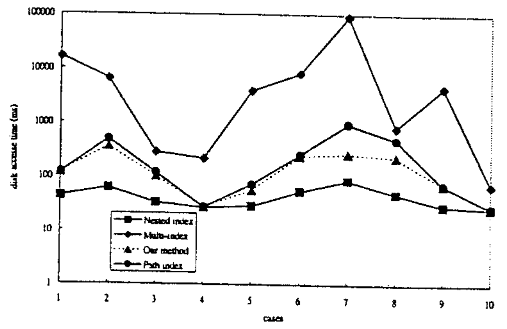

Figure 6(a) Retrieval cost in the aggregation hierarchy with 10 classes

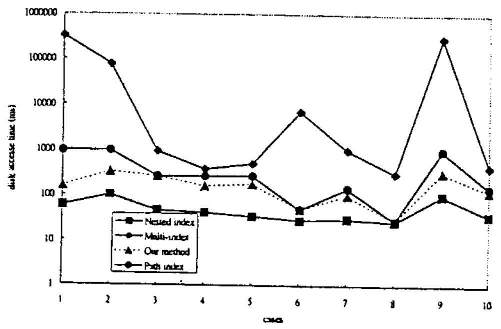

Figure 6(b) Retrieval cost in the aggregation hierarchy with 15 classes

On the other hand, we also compare the update performance of four indexing techniques, as shown in Figure 7(a) and Figure 7(b). The ranks of four indexing techniques are exactly reversed in Figure 7, compared with Figure 6. The nested index has the worst update cost, because the path length is always larger than two and it should care about the reverse traversal operation. In general, the reference path should be forward traversed twice and reversely traversed once when updating a nested index. The multi-index has the best update cost, because only one sub-path is required to update. Besides our method is almost equal to the path index. Even if ours performs worse than the path index, the difference is very little (see the vertical scale in Figure 7). In fact the results of the path index are the special ones of our method.

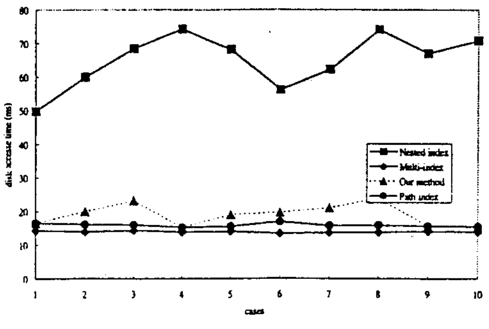

Figure 7(a) Update cost in the aggregation hierarchy with 10 classes

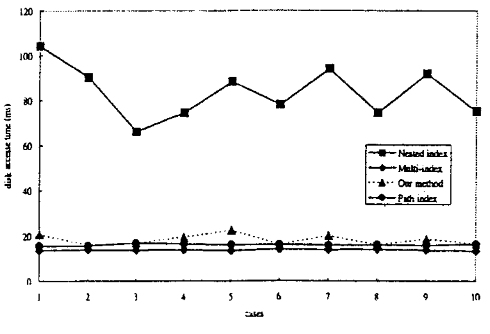

Figure 7(b) Update cost in the aggregation hierarchy with 15 classes

5. CONCLUSIONS

The object-oriented database system (OODB) is more popular than before. It can deal with complex objects and is used in many applications where relational database systems can not be applied. The performance of an OODB is strongly dependent on the indexing techniques. In this paper, we study the indexing techniques based on the inheritance hierarchy and aggregation hierarchy in the OODB. Here we consider the access probabilities and the number of instantiations of the classes in the OODB and expect our methods to have better retrieval performance than other ones.

For the inheritance hierarchy, we provide a cost function to evaluate the gain of building an index file on the sub-inheritance hierarchy rooted at one class. We use the CH-tree structure and improve it by avoiding extra checking when the target class in a query is not the root class. We also compare our method with the CD method. In despite of the database size, ours has better retrieval and update performance than the CD's. Finally the CD method can not deal with the multiple inheritance that can be handled in our method.

For the aggregation hierarchy, we use the path-catenation technique to evaluate how to build index files on the classes where users are interested. After the evaluation, the path-catenation with the least retrieval cost is found, and the built index shows better retrieval performance than the path index in the experiments. At the same time, it has not-bad update performance. Besides our method can handle the aggregation hierarchy with the multiple references on a class, where other methods can not be applied.

REFERENCES

[1] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects," IEEE Trans. on Knowledge and Data Engineering, Vol. 1, No. 2, Oct. 1989, pp. 196-214.

[2] E. Bertino, "An Indexing Technique for Object-oriented Databases," Proc. IEEE International Conference on Data Engineering, pp. 160-170, 1991.

[3] E. Bertino and C. Guglielmina, "Optimization of Object-oriented Queries Using Path Indices," IEEE Computer, 1992, pp. 140-149.

[4] E. Bertino and L. Martino, *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley Publishing Company, Inc., 1993.

[5] E. Bertino and P. Foscoli, "Index Organizations for Object-oriented Database Systems," IEEE Trans. on Knowledge and Data Engineering, Vol. 7, No. 2, April 1995, pp. 193-209.

[6] C. Kilger and G. Moerkotte, "Indexing Multiple Sets," Proc. International Conference on VLDB, pp. 180-191, 1994.

[7] W. Kim, K. C. Kim, and A. Dale, "Indexing Techniques for Object-oriented Databases," in *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, Inc., 1989, pp. 371-394.

[8] C. C. Low, B. C. Ooi, and H. Lu, "H-trees: a Dynamic Associative Search Index for OODB," Proc. ACM SIGMOD International Conference on Management of Data, pp. 134-143, 1992.

[9] S. Ramaswamy and P. C. Kanellakis, "OODB Indexing by Class-division," Proc. ACM SIGMOD International Conference on Management of Data, pp. 139-150, 1995.

[10] B. Sreenath and S. Seshadri, "The hcC-tree: an Efficient Index Structure for Object Oriented Databases," Proc. International Conference on VLDB, pp. 203-213, 1994.