Design of a New Indexing Organization for a Class-Aggregation Hierarchy in Object-Oriented Databases [1]

Chien-I Lee[†], Ye-In Chang[‡] and Wei-Pang Yang[♭]

[†]Institute of Information Education
National Tainan Teachers College
Tainan, Taiwan
Republic of China
{E-mail: leeci@ipx.ntntc.edu.tw}
{Tel: 886-6-2113111 (ext. 776)}
{Fax: 886-6-2244409}

[‡]Dept. of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China
{E-mail: changyi@math.nsysu.edu.tw}
{Tel: 886-7-5252000 (ext. 3819)}
{Fax: 886-7-5253809}

[♭]Dept. of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan
Republic of China
{E-mail: wpyang@cis.nctu.edu.tw}

[1]This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-84-2213-E-110-023.

Abstract

In an object-oriented databases, a class consists of a set of attributes, and the values of the attributes are objects that belong to other classes; that is, the definition of a class forms a class-aggregation hierarchy of classes. A branch of such a hierarchy is called a *path*. There have been several index organizations proposed to support object-oriented query languages, including $multiindex$, $join\ index$, $nested\ index$ and $path\ index$. In all the proposed index organizations, they are helpful only to a query which retrieves the objects of the root class of a given path by a predicate which specifies the value of the attribute at the end of the path. In this paper, we propose a new index organization for evaluating queries, called $full\ index$, where an index is allocated for each class and its attribute (or nested attribute) along the path. From the analysis results, we show that a $full\ index$ can support any type of queries along a given path with a lower retrieval cost than all the other index organizations. Moreover, to reduce the high update cost for a long given path, we will split the path into several subpaths and to allocate a separate index on each subpath. Given a path, the number of subpaths and the index organization of each subpath define an index configuration. Since a low retrieval cost and a low update cost are always a trade-off in all index organizations, we also propose cost formulas to determine the index configuration which can provide the best performance for various applications by taking into account various types of queries along a given path and a set of queries with more than one nested predicates along a given path.

(*Key Words:* Access methods, complex objects, index selection, object-oriented databases, query optimization.)

1 Introduction

The new generation of computer-based applications, such as computer-aided designed and manufacturing (CAD/CAM), multimedia databases (MMDB), and software development environments (SDEs), requires more powerful techniques to generate and manipulate large amounts of data. The traditional well-known record-based relational data model does not provide the possibility of directly modeling complex data. Moreover, many complex relationships among data, for example, instantiation, aggregation and generalization, can not be well defined in the relational data model. Furthermore, the relational data model does not provide mechanisms to associate data behavior with data definitions at schema level.

Object-oriented database management systems [1, 10, 12, 13, 19, 20, 22, 23, 24] represent one of the most promising directions in the database area towards meeting requirements posed by advanced applications. An object-oriented data model not only provides great expressive power to describe data and to define complex relationships among data, it but also provides mechanisms for behavioral abstraction. In an object-oriented data model [4, 6, 9], any real-world entity is represented by only one data modeling concept, the object. Each object is identified by a unique identifier (UID). The state of each object is defined at any point in time by the value of its attributes. The attributes can have an value both primitive objects (for example, strings, integers, or booleans) and non-primitive objects, which in turn, consist of a set of attributes. Objects with similar attributes are grouped into classes. A class C consists of a number of attributes, and the value of an attribute A of an object belonging to the class C is an object or a set of objects belonging to some other class C'. The class C' is called the *domain* of the attribute A of the class C and this association is called an *aggregation relationship* between class C and C'. The class C' in turn consists of a number of attributes, and their domains are other classes. In general speaking, a class is a hierarchy of classes of *aggregation relationships*, called *aggregation hierarchy*. A branch of such a hierarchy is called a *path*. An example of a class-aggregation hierarchy is shown in Figure 1. An example of a path against this class-aggregation hierarchy is *Student.study.taught-by.work-in.name*. There have been several index organizations [3, 5, 8, 17, 14, 16, 18, 21, 25] proposed to support object-oriented query languages, including *multiindex* [18], *join index* [21], *nested index* [3], *path index* [3] and *access support relation* [14].

Consider the following query: *Retrieve all the students who study some courses in the Department of Computer Science*, for the class-aggregation hierarchy as shown in Fig-

Figure 1: An example of a class-aggregation hierarchy.

ure 1 and the related instances shown in Figure 2, where 'study in the Department of Computer Science' is a nested predicate. (Note that nested predicates are often expressed using *path-expressions*; the above nested predicate can be expressed as *Student.study.taught-by.work-in.name* = 'Computer Science'.) Given a path = *Student.study.taught-by.work-in.name*, there are four indices in a *multiindex* set. The first index is on the subpath *Student.study* and contains the following pairs: (Course[i], {Student[o]}), (Course[j], {Student[p]}) and (Course[k], {Student[q]}). The second index is on the subpath *Course.taught-by* and contains the following pairs: (Teacher[i], {Course[k], Course[l]}) and (Teacher[j], {Course[i], Course[j]}). The third index is on the subpath *Teacher.work-in* and contains the following pairs: (Department[l], {Teacher[i]}) and (Department[m], {Teacher[j]}). The forth index is on the subpath *Department.name* and contains the following pairs: (Computer Science, {Department[l]}) and (Mathematics, {Department[m]}).

A *join index* [21] is similar to a *multiindex* except that a *join index* supports both forward and reverse traversal along the path; that is, there are two indices allocated between each class and its immediate attribute along the path. That is, for a given path, the number of indices for a *join index* is two times of the one for a *multiindex*.

For the same example shown above, a *nested index* will contain the following pairs: (Computer Science, {Student[q]}) and (Mathematics, {Student[o], Student[p]}), while a *path index* will contain the following pairs:

(Computer Science, {Student[q].Course[k].Teacher[i].Department[l]}) and
(Mathematics, {Student[o].Course[i].Teacher[j].Department[m],
Student[p].Course[j].Teacher[j].Department[m]}).

An *access support relation* [14] is an organization very similar to a path index except that an *access support relation* is possible to store incomplete path instantiations using null values in relations.

In general, a *multiindex* [18] is allocated on each class traversed by the path, which solves a nested predicate by scanning a number of indices equal to the path-length. Therefore, a *multiindex* has a high retrieval cost, but a low update cost. Since a *nested index*

Figure 2: Instances of classes in Figure 1.

only associates instances of the first class of the path with the values of the end of the path, a *nested index* has a lower retrieval cost for querying on the first class with the nested predicate on the last attribute of the path but has a high update cost for forward and backward object traversals to access the database itself. A *path index* [3] provides an association between an object at the end of the path and the instantiations ending with the object. Therefore, a *path index* can be used to evaluate nested predicates on all classes along the path; however, a *path index* has a high update cost.

In all the above proposed index organizations, they will be helpful only to a query which retrieves the objects of the root class of a given path by a predicate which specifies the value of the attribute at the end of the path. Consider another query: *Retrieve all the courses taught by those teachers who are in the Department*[*l*]. The path-expression for this query is *Course.taught-by.work-in* = 'Department[l]'. To answer this question, we have to scan the second index and the third index in the *multiindex* described above. The *nested index* described above will not be helpful to answer the query, while the *path index* described above will be only able to answer partial result ({Course[k]}) by scanning all the *path index*. (Note that the answer of the query should be {Course[k], Course[l]}.) To reduce the high retrieval cost in a *multiindex* and to overcome the problem that some queries in a *nested index* and a *path index* cannot be answered, in this paper, we propose a new index organization for evaluating queries, called *full index*, where an index is allocated for each class and each of its immediate and nested attributes along the path. For the same example shown above, a *full index* will contain 10 indices as shown in Table 1.

Class	*Attribute*	*Contents*
Student	study	(Course[i], {Student[o]}), (Course[j], {Student[p]}) and (Course[k], {Student[q]})
Student	taught-by	(Teacher[i], {Student[q]}) and (Teacher[j], {Student[o], Student[p]})
Student	work-in	(Department[l], {Student[q]}) and (Department[m], {Student[o], Student[p]})
Student	name	(Computer Science, {Student[q]}) and (Mathematics, {Student[o], Student[p]})
Course	taught-by	(Teacher[i], {Course[k], Course[l]}) and (Teacher[j], {Course[i], Course[j]})
Course	work-in	(Department[l], {Course[k], Course[l]}) and (Department[m], {Course[i], Course[j]})
Course	name	(Computer Science, {Course[k], Course[l]}) and (Mathematics, {Course[i], Course[j]})
Teacher	work-in	(Department[l], {Teacher[i]}) and (Department[m], {Teacher[j]})
Teacher	name	(Computer Science, {Teacher[i]}) and (Mathematics, {Teacher[j]})
Department	name	(Computer Science, {Department[l]}) and (Mathematics, {Department[m]})

Table 1: An example of a full index

A comparison of a *multiindex*, a *nested index*, a *path index* and a *full index* is shown in Figure 3. From the analysis results, we show that a *full index* can support any type of queries along a given path against a class-aggregation hierarchy with a lower retrieval cost than all the other index organizations. Therefore, our full index is suitable for queries against a given path, where queries on subpaths might not be *predictable*. To reduce the high update cost for a long given path, we can split the path into several subpaths and allocate a separate index on each subpath [7, 14, 15]. Given a path, the number of subpaths and the index organization of each subpath define an index configuration. But the increase of the number of indices for subpaths will also increase the retrieval cost for scanning a number of indices, which results in a high retrieval cost. Since a low retrieval cost and a low update cost are always a trade-off in all index organizations, we can establish a cost formula to look for a compromise between these two requirements. In [7], they have proposed cost formulas to evaluate the costs of various index configurations. However, they do not support the *partial instantiations* (defined in Section 2) and do not consider more than one nested predicates along the path. In this paper, we also propose cost formulas to determine the index configuration which can provide the best performance for various applications by taking into account various types of queries along a given path and a set of queries with more than one nested predicates along a given path.

The rest of this paper is organized as follows. In Section 2, we define different types of queries along a given path. In Section 3, we introduce the proposed *full index* and related index operations. In Section 4, we present the cost model for a *full index* and show some analysis results of a *full index* compared with those of a *multiindex*, a *path index* and

Figure 3: A comparison: (a) a *multiindex*; (b) a *nested index*; (c) a *path index*; (d) a *full index*.

a *nested index*. In Section 5, we present cost formulas which determine an optimal index configuration. Finally, Section 6 concludes this paper.

2 Query Types in a Class-Aggregation Hierarchy

An attribute of any class on a class-aggregation hierarchy is logically an attribute of the root of the hierarchy; that is, the attribute is a nested attribute of the root class. A predicate on a nested attribute is called *nested predicate*. A *path* is defined a $C(1).A(1).A(2).....A(n)$, where $C(i)$ is a class in a class-aggregation hierarchy and $A(i)$ is an attribute of class $C(i)$, $1 \leq i \leq n$. The *path-length* indicates the number of classes along a path, that is, the value is n. An *instantiation* of a path is defined as a sequence of $(n + 1)$ objects as $O(1).O(2).....O(n + 1)$, where $O(i)$ is an instance of class $C(i)$, $1 \leq i \leq (n + 1)$. A *patial instantiation* of a path is defined as a sequence of objects as $O(i).O(i + 1).....O(j)$, where $O(i)$ is an instance of class $C(i)$, $1 \leq i \leq j \leq (n + 1)$. Object-oriented query languages allow objects to be restricted by predicates on both nested and non-nested attributes of objects. In the following, we define types of queries along a given path against a class-aggregation hierarchy.

Definition 1 *Given a path which is defined as C(1).A(1).A(2)...A(n) (n ≥ 1) and an aggregation hierarchy H, a query of a simple type is expressed using path-expression as C(i).A(i).....A(j) = 'O(j)', where 1 ≤ i ≤ j ≤ n.*

That is, a query of a simple type retrieves the objects of a class along a given path by a predicate on its nested (or non-nested) attribute, where the specified class need not to be the root of the path and the specified attribute need not to be the end of the path. The following query **Q1** shows an example of a query of a simple type against the class-hierarchy shown in Figure 1.

Q1: *Retrieve all the students who study some courses in the Department of Computer Science.*

Q1 contains the nested predicate 'study in the Department of Computer Science'. Nested predicates are often expressed using *path-expressions*. For example, the above nested predicate can be expressed as *Student.study.taught-by.work-in.name* = 'Computer Science'.

Definition 2 *Given a path which is defined as C(1).A(1).A(2).....A(n) (n ≥ 1) and an aggregation hierarchy H, a query of a k-degree complex type is expressed using path-expressions as C(i).A(i).....A(j$_1$) = 'O(j$_1$)', C(i).A(i).....A(j$_2$) = 'O(j$_2$)', ..., C(i).A(i).....A(j$_k$) = 'O(j$_k$)', where 1 ≤ m ≤ k, 1 ≤ i ≤ j$_m$ ≤ n.*

That is, a query of a k-degree complex type retrieves the objects of a class along a given path by k predicates on its k nested (or non-nested) attributes along the given path. The following query **Q2** shows an example of a query of a 2-degree complex type against the class-hierarchy shown in Figure 1, where the path-expressions are *Student.study.taught-by* = 'Teacher[i]' and *Student.study* = 'Course[i]'.

Q2: *Retrieve all the students whom are taught by Teacher[i] and who study in Course[i].*

Definition 3 *Given a path which is defined as C(1).A(1).A(2).....A(n) (n ≥ 1) and an aggregation hierarchy H, a general set of queries is expressed using path-expressions as C(i$_1$).A(i$_1$).....A(j$_1$) = 'O(j$_1$)', C(i$_2$).A(i$_2$).....A(j$_2$) = 'O(j$_2$)', ..., C(i$_k$).A(i$_k$).....A(j$_k$) = 'O(j$_k$)', where 1 ≤ m ≤ k, 1 ≤ i$_m$ ≤ j$_m$ ≤ n.*

That is, there are more than one queries along a given path. The following example **Q3** shows two queries in a general set against the class-hierarchy shown in Figure 1, where their path-expressions are *Student.study* = 'Course[i]' and *Course.taught-by.work-in.name* = 'Mathematics', respectively.

Q3: *Retrieve all the students who study in Course[i], and retrieval all the courses taught by those teachers who are in the Department of Mathematics.*

3 A Full Index

In this section, we first give the formal definition of a *full index*. Then, we describe four operations on a *full index*, which are *retrieval*, *update*, *insertion* and *deletion*.

3.1 Organization

Definition 4 *Given a path P which is defined as C(1).A(1).A(2).....A(n) ($n \geq 1$) and an aggregation hierarchy H, a full index(FX) on P is defined as a set of indices, which are FX_1^1, FX_1^2, ..., FX_1^n, FX_2^2, ..., FX_n^n, where FX_i^j is an index on class C(i) and attribute A(j), $1 \leq i \leq j \leq n$.*

For example, let a path = *Student.study.taught-by.work-in.name* of H be shown in Figure 1 and the instances of classes in H be shown in Figure 2, a *full index* consisting of ten indices is described as follows.

The first index FX_1^1 on class *Student* and attribute *study* contains the following pairs:

(Course[i], {Student[o]}),
(Course[j], {Student[p]}) and
(Course[k], {Student[q]}).

The second index FX_1^2 on class *Student* and attribute *taught-by* contains the following pairs:

(Teacher[i], {Student[q]}) and
(Teacher[j], {Student[o], Student[p]}).

The third index FX_1^3 on class *Student* and attribute *work-in* contains the following pairs:

(Department[l], {Student[q]}) and
(Department[m], {Student[o], Student[p]}).

The forth index FX_1^4 on class *Student* and attribute *name* of class *Department* contains the following pairs:

(Computer Science, {Student[q]}) and
(Mathematics, {Student[o], Student[p]}).

The fifth index FX_2^2 on class *Course* and attribute *taught-by* contains the following pairs:

(Teacher[i], {Course[k], Course[l]}) and
(Teacher[j], {Course[i], Course[j]}).

The sixth index FX_2^3 on class *Course* and attribute *work-in* contains the following pairs:

(Department[l], {Course[k], Course[l]}) and
(Department[m], {Course[i], Course[j]}).

The seventh index FX_2^4 on class *Course* and attribute *name* of class *Department* contains the following pairs:

(Computer Science, {Course[k], Course[l]}) and
(Mathematics, {Course[i], Course[j]}).

The eighth index FX_3^3 on class *Teacher* and attribute *work-in* contains the following pairs:

(Department[l], {Teacher[i]}) and
(Department[m], {Teacher[j]}).

The ninth index FX_3^4 on class *Teacher* and attribute *name* of class *Department* contains the following pairs:

(Computer Science, {Teacher[i]}) and
(Mathematics, {Teacher[j]}).

The tenth index FX_4^4 on class *Department* and attribute *name* contains the following pairs:

(Computer Science, {Department[l]}) and
(Mathematics, {Department[m]}).

3.2 Operations

A *full index* supports a fast retrieval of all types of queries defined in Section 2. When an evaluation of a nested predicate against the nested attribute $A(j)$ of class $C(i)$, it requires a lookup of a single index FX_i^j, where $1 \leq i \leq j \leq n$. Suppose that an instance $O(i)$ of class $C(i)$ along the path has an object $O(i+1)$ as the value of the attribute $A(i)$. Now, $O(i)$ is updated to a new object $O'(i+1)$. An update to the *full index* proceeds as follows.

First, we take a lookup of index FX_i^i and replace $O(i+1)$ of $O(i)$ with a new object $O'(i+1)$. Second, we update the index FX_m^k by using the index FX_m^{k-1} and FX_k^k, where $1 \leq m < i$ and $i \leq k \leq n$. Third, we update the index FX_i^m by using the index FX_i^{m-1} and FX_m^m, where $i < m \leq n$.

As the example shown in Figures 1 and 2, suppose the object in attribute *taught-by* of $Course[i]$ is updated from $Teacher[j]$ to $Teacher[i]$. Then, a series of operations of update on the *full index* are performed as follows.

First, we take a lookup of the index FX_2^2 and replace $Teacher[j]$ of $Course[i]$ with $Teacher[i]$; that is, FX_2^2 contains the following pairs:

(Teacher[i], {Course[i], Course[k], Course[l]}) and
(Teacher[j], {Course[j]}).

Second, we update the index FX_1^2 by using FX_1^1 and FX_2^2; that is, FX_1^2 contains the following pairs:

(Teacher[i], {Student[o], Student[q]}) and
(Teacher[j], {Student[p]}).

Then, we update the index FX_1^3 by using FX_1^2 and FX_3^3; that is, FX_1^3 contains the following pairs:

(Department[l], {Student[o], Student[q]}) and
(Department[m], {Student[p]}).

Moreover, we update the index FX_1^4 by using FX_1^3 and FX_4^4; that is, FX_1^4 contains the following pairs:

(Computer Science, {Student[o], Student[q]}) and
(Mathematics, {Student[p]}).

Third, we update the index FX_2^3 by using FX_2^2 and FX_3^3; that is, FX_2^3 contains the following pairs:

(Department[l], {Course[i], Course[k], Course[l]}) and
(Department[m], {Course[j]}).

Then, we update the index FX_2^4 by using FX_2^3 and FX_4^4; that is, FX_2^4 contains the following pairs:

(Computer Science, {Course[i], Course[k], Course[l]}) and
(Mathematics, {Course[j]}).

Insertion and deletion operations are similar to the update operation. We perform an insertion/deletion operation of an object on index FX_i^i, instead of the replacement operation in the first step of the update to the index.

4 Performance Analysis

In this section, we will describe the cost model and analyze some performance results of a *full index*. Moreover, a comparison of performance of these related indexing schemes will also be presented.

4.1 Cost Model

In this paper, we use a cost model which is similar to the one proposed in [3, 7], in which the data structure to model indices is based on a *B-tree* [2, 11]. Similar to their model [3, 7], we assume that the values of attributes are uniformly distributed among instances of the class and all key values have the same length. However, in [3, 7], since a *nested index* and a *path index* can not support any query for partial instantiations, they need to make one more assumption: no partial instantiations; that is, each instance of a class $C(i)$ is referenced by instances of class $C(i$ - $1)$, $1 < i \leq n$. In this paper, we release this assumption to support a query of any type for partial instantiations in a *full index* by taking into account of 'NULL' value of instances. Given a path $C(1).A(1).....A(n)$, the parameters that we consider in the cost model are grouped as follows.

Parameter	**Description**
$DV(i)$	Number of distinct values held in attributes $A(i)$ including the NULL value, $1 \leq i \leq n$.
$m(i)$	Average number of values for a set for attribute $A(i)$, $1 \leq i \leq n$.
$D(i)$	Number of distinct sets for attribute $A(i)$, $1 \leq i \leq n$; that is, $D(i) = \binom{DV(i)}{m(i)}$.
$N(i)$	Cardinality of class $C(i)$ including the NULL value, $1 \leq i \leq n$.
$K(i)$	Average number of instances of class $C(i)$ with the same set of values for attribute $A(i)$; that is, $K(i) = \lceil \frac{N(i)}{D(i)} \rceil$.
$UIDL$	Length of the object-identifier in bytes.
PS	Page size in bytes.
d	Order of a nonleaf node.
f	Average fanout from a nonleaf node.

pp	Length of page pointer.
kl	Average length of a key value for the indexed attribute.
ol	Length of a header in an index record.
DS	Length of the directory at the beginning of the record, when the record size is greater than the page size.

Moreover, to compare these indexing schemes on the same ground, we use the same assumptions and parameters as those in [3], except that we consider the case that an attribute $A(i)$ has a set of values, instead of a single value, and the average number of values in a set is $m(i)$. Therefore, we use $DV(i)$ to denote number of distinct values for $A(i)$, and $D(i)$ $(= \binom{DV(i)}{m(i)})$ to denote the number of distinct sets. In this case, when $A(i)$ has a single value (i.e., $m(i) = 1$), $DV(i) = D(i)$. That is, we consider a more general case in $A(i)$. It is straightforward to extend the cost models [3] of multiindex, path index, and nested index to consider the case that an attribute has a set of values, instead of a single value. In this paper, we do have used these extended cost models of those indexing schemes in our performance comparison described in section 4.5. Nore that, here, to simplify our presentation for the performance analysis, we have omitted some parameters used in [3], since they can be easily derived from the other parameters. For the values of those parameters which are not critical in the comparison, we have kept them as constants; those parameters are also kept constant in [3].

4.2 Retrieval Cost

Let $K(i, j)$ be the average number of instances of class $C(i)$ having the same set of values held in the nested attribute $A(j)$, where $1 \leq i \leq j \leq n$; that is, $K(i, j) = \prod_{r=i}^{j} K(r)$. XF_i^j is denoted as the average length of a leaf-node index record for the index FX_i^j in the $full$ $index$ and

$$\begin{aligned} XF_i^j &= K(i, j)m(i)UIDL + kl + ol, && XF_i^j \leq PS, \\ XF_i^j &= K(i, j)m(i)UIDL + kl + ol + DS, && XF_i^j > PS, \\ &\text{where } DS = \lceil \tfrac{K(i,j)m(i)UIDL+kl+ol}{PS} \rceil (UIDL + pp). \end{aligned}$$

The number of leaf pages LP_i^j for the index FX_i^j is

$$\begin{aligned} LP_i^j &= \lceil \tfrac{D(j)}{\lfloor \frac{PS}{XF_i^j} \rfloor} \rceil, && XF_i^j \leq PS, \\ LP_i^j &= D(j) \lceil \tfrac{XF_i^j}{PS} \rceil, && XF_i^j > PS. \end{aligned}$$

The number of index pages RC_i^j accessed in the index FX_i^j for a nested predicate on class $C(i)$ with a nested attribute $A(j)$ is

$$RC_i^j = h(i, j) + 1, \qquad XF_i^j \leq PS,$$

where $h(i, j)(= \lceil \log_f D(j) \rceil)$ is the number of nonleaf nodes that must be accessed in the index FX_i^j. When the record size is larger than the page size, i.e., $XF_i^j > PS$, np is the number of leaf pages needed to store the record, i.e., $np = \lceil \frac{XF_i^j}{PS} \rceil$. Therefore,

$$RC_i^j = h(i, j) + np, \qquad XF_i^j > PS.$$

4.3 Maintenance Cost

The index maintenance cost deriving from update, deletion or insertion operations for an instance of a class $C(i)$ is denoted by U, D and I, respectively. To simplify the analysis, we consider only the costs of leaf-page modification and exclude the costs of index page splits. The cost CBM_i^i of an update on the index FX_i^i is the sum of the cost of removing the UID of object $O(i)$ from the record associated with its attribute $O(i + 1)$ and the cost of adding it to the new value $O'(i + 1)$; that is,

$$CBM_i^i = CO(1 + pl),$$

where CO denotes the cost of finding the leaf node containing the key value and the cost of reading and writing the leaf node, and pl is the probability that the old and new values are on different leaf node.

When a leaf page is modified, one page access is needed to read the leaf page containing the update record, and another one page access is needed to write this page; in addition, $h(i, i)$ pages are accessed to determine the leaf node containing the record to be updated. Therefore,

$$CO = h(i, i) + 2, \qquad XF_i^i \leq PS.$$

When the record size is larger than the page size and np is the number of leaf pages needed to store the record, i.e., $np = \lceil \frac{XF_i^i}{PS} \rceil$, there are $h(i, i) + 1$ of pages which must be accessed to find the header of the record with the old value. From the header of the record, it is possible to determine the page from which a UID must be deleted or to which a UID must be added. If this page is different from the page containing the header of the record, a further page access must be performed. This probability is given by $\frac{np-1}{np}$. Therefore,

$$CO = h(i, i) + 2 + \tfrac{np-1}{np}, \qquad XF_i^i > PS.$$

The probability that the current and new values are on different leaf nodes is

$$pl = 1, \qquad XF_i^i > PS,$$
$$pl = 1 - \frac{\lfloor \frac{PS}{XF_i^i} \rfloor - 1}{D(i) - 1}, \qquad XF_i^i \leq PS.$$

Moreover, an update operation on the index FX_i^i will cause other associated indices to be updated as stated in Subsection 3.2. When an index FX_m^k is updated ($1 \leq m \leq i$, $i < k \leq n$), the total cost for this update consists of the update cost CBM_m^k on the index FX_m^k and the retrieval cost $2RC_k^k$. (Note that one cost RC_k^k is for finding $O(k+1)$ to determine which an object $O(k+1)$ for object $O(k)$ to be updated, and the other one cost RC_k^k is for finding $O'(k+1)$ to determine which a new object $O'(k+1)$ to be updated to.)

Therefore, the total update cost U_i is

$$U_i = CBM_i^i + \sum_{m=1}^{i-1} \sum_{k=i}^{n} (CBM_m^k + 2RC_k^k) + \sum_{m=i+1}^{n} (CBM_i^m + 2RC_m^m).$$

Since there is only a deletion of an old value or an insertion of a new value on the index FX_i^i, pl is 0. Therefore, the cost of deletion D_i and the cost of insertion I_i are given by

$$D_i = I_i = CO + \sum_{m=1}^{i-1} \sum_{k=i}^{n} (CBM_m^k + 2RC_k^k) + \sum_{m=i+1}^{n} (CBM_i^m + 2RC_m^m).$$

4.4 Storage Cost

The number of nonleaf pages NLP_i^j for the index FX_i^j is

$$NLP_i^j = \lceil \frac{LO}{f} \rceil + \lceil \frac{\lceil \frac{LO}{f} \rceil}{f} \rceil + ... + \lceil X \rceil,$$

where $LO = \min(D(j), LP_i^j)$ and each term is successively divided by f until the last term X is less than f.

Then, the total storage cost SC for a *full index* is

$$SC = \sum_{i=1}^{n} \sum_{j=i}^{n} (LP_i^j + NLP_i^j).$$

4.5 A Comparison

In this subsection, we will show a number of interesting results of a *full index* (denoted as FX) on the basis of the analysis cost model described in the above subsections and compare the performance of the *full index* with those of a *multiindex* (denoted as MX), a *nested index* (denoted as NX) and a *path index* (denoted as PX) [3].) (Note that since

a *join index* is similar to a *multiindex* and an *access support relation* is similar to a *path index*, we omit the performances for a *join index* and an *access support relation*.) By using different values of parameters, we simulate some interesting situations for different application requirements. However, there are some parameters kept constant in all the simulations, which are $N(1) = 200{,}000$, $UIDL = 8$, $kl = 2$, $ol = 6$, $pp = 4$, $f = 218$, $d = 146$ and $PS = 4096$. The values of these parameters are the same as those in [3].

Figure 4 shows the retrieval costs for queries of a simply type with a path-expression $C(1).A(1)....A(3) = `O(3)'$, by using a *multiindex*, a *nested index*, a *path index* and a *full index*, respectively, where $n = 3$, $m(1) = m(2) = m(3) = 1$, $K(2) = K(3) = 10$, and $K(1)$ is varied from 1 to 50. Figure 5 shows the retrieval costs for queries of a simply type with a path-expression $C(1).A(1)....A(3) = `O(3)'$, by using a *multiindex*, a *nested index*, a *path index* and a *full index*, respectively, where $n = 3$, $K(1) = K(2) = K(3) = 10$, $m(2) = m(3) = 1$, and $m(1)$ is varied from 1 to 5. From these two figures, we observe that the retrieval costs for these four index are increased with the values of $K(1)$ or $m(1)$. The reason is that as $K(3)$ or $m(3)$ are increased, the size of a leaf-node index record is increased, which may result in an increase of the number of leaf pages for the index record. Consequently, the retrieval costs are increased. Moreover, since the *multiindex* requires scanning three indices to access the desired objects, the *multiindex* has the highest retrieval cost. The *full index* has a lower retrieval cost than the *multiindex* and the *path index*. The reason is that the *full index* requires only one lookup in the index FX_1^3, but the *multiindex* requires one lookup for each index and the *path index* has to take a lookup for a larger size of index than the *full index*. In this case, the *nested index* is the same as the index FX_1^3 of the *full index*. Therefore, the *nested index* has the same retrieval cost as the *full index*.

Figure 6 shows the retrieval costs for queries of a simply type with a path-expression $C(1).A(1)....A(n) = `O(n)'$, by using a *full index*, a *multiindex*, a *nested index* and a *path index*, respectively, where $m(y) = 1$, $K(y) = 2$, $1 \leq y \leq n$ and n is varied from 2 to 10. For the same reasons in Figure 4, the *multiindex* has the highest retrieval cost, and the *full index* and the *nested index* have the lowest retrieval cost. Moreover, Since a *multiindex* is allocated on each class traversed by the path, which is solving a nested predicate by scanning a number of indices equal to the path-length, the retrieval cost of the *multiindex* is increased as n is increased. A *path index* provides an association between an object at the end of the path and the instantiations ending with the object. Therefore, the index size of a *path index* is increased as n is increased, which results in an increase of the retrieval cost. Since the index FX_1^n of a *full index* and a *nested index* only associate

Figure 4: The retrieval cost under different values of $K(1)$.

Figure 5: The retrieval cost under different values of $m(1)$.

Figure 6: The retrieval cost under different values of the path-length (n).

instances of the first class of the path with the values of the end of the path, a *full index* and a *nested index* have lower retrieval costs and these costs are not affected by n.

Consider queries of a simply type with path-expressions $C(1).A(1)....A(j) = `O(j)'$, where $1 \leq j \leq n$, $m(y) = 1$, $K(y) = 2$, $1 \leq y \leq n$ and $n$ is varied from 2 to 10. Suppose a *multiindex*, a *nested index*, a *path index* and a *full index* have been allocated for queries of a simple type with a path-expression $C(1).A(1)....A(n) = `O(n)'$, respectively. Since a *path index* and a *nested index* can not support any query for partial instantiations as stated before, a lookup in real databases is required. Therefore, the retrieval costs for a *nested index* and a *path index* will be very high. In Figure 7, we compare the average retrieval cost based on a *multiindex* and a *full index* for all the queries with path-expressions $C(1).A(1)....A(j) = `O(j)'$, where $1 \leq j \leq n$, which have the same probability. From this figure, we can find that a *full index* can provide a constant retrieval cost for the reason of only one lookup on an index for each query. While the retrieval cost for a *multiindex* is increased as n is increased for the reason of requiring scanning a number of indices for each query.

Figure 8 shows the average retrieval cost for queries in a general set with path-expressions $C(i).A(i)....A(j) = `O(j)'$, where $1 \leq i \leq n$, $i \leq j \leq n$ and n is varied from 2 to 10. For the same reasons in Figure 7, we can find that a *full index* can provide a constant retrieval cost, while the retrieval cost for a *multiindex* is increased as n is increased.

Figure 9 shows the average update costs for a path $= C(1).A(1)....A(n)$, by using a *full index*, a *multiindex*, a *nested index* and a *path index*, respectively, where $m(y) = 1$, $K(y) = 4$, $1 \leq y \leq n$ and n is varied from 2 to 10. A *multiindex* has the lowest average update cost for the reason of updating a single index for each update operation.

Figure 7: The average retrieval cost for queries of a simply type.

Figure 8: The average retrieval cost for queries in a general set.

Figure 9: The average update cost.

As stated in [3], for an update on an object $O(i)$, a *path index* requires the cost for a *forward traversal* and the cost of a *B-tree* update, and a *nested index* requires the costs for a *forward* and a *backward traversals* and the cost of a *B-tree* update. (Note that a *forward traversal* is defined as the accesses of objects $O(i + 1)$,..., $O(n)$ such that $O(i + 1)$ is referenced by object $O(i)$ through attribute $A(i)$, ..., and $O(n)$ is referenced by object $O(n - 1)$ through attribute $A(n - 1)$. On the other hand, a *backward traversal* is defined as the accesses of objects $O(i - 1)$,..., $O(2)$ such that $O(i - 1)$ is referenced by object $O(i)$ through attribute $A(i - 1)$, ..., and $O(1)$ is referenced by object $O(2)$ through attribute $A(1)$.) The cost for a *forward traversal* is in proportion to n, and the cost for a *backward traversal* is in proportion to 2^n [3]. Therefore, a *nested index* has a higher average update cost than a *path index*, and a *path index* has a higher average update cost than a *multiindex*. (Note that as stated in [3], the update cost of the traversal operations for a nested index is also related to the size of the real database involved in the path, while the update cost of our full index is independent of the size of the real database involved in the path.) Since in a *full index*, an update operation in an index may cause some other update operations in other indices, the average update cost for a *full index* is higher than the one for a *multiindex*. Moreover, the number of updated indices for an update in a *full index* is in proportion to n^2 and the cost for these updated indices is higher than the cost for a *forward traversal* in a *path index*. Therefore, a *full index* has a higher average update cost than a *path index*. When n is small, since the cost for a *backward traversal* in a *nested index* is lower than the cost for the updated indices in a *full index*, a *nested index* has a lower average update cost than a *full index*. On the other hand, when $n \geq 5$, a *full index* has a lower average update cost than a *nested index*.

Figure 10: The storage cost for $K(2) = K(3) = 10$ under different value of $K(1)$.

Figure 10 shows the storage costs for a *full index*, a *multiindex*, a *nested index* and a *path index*, where $n = 3$, $m(1) = m(2) = m(3) = 1$, $K(2) = K(3) = 10$, and $K(1)$ is varied from 1 to 50. Obviously, the *full index* can reduce the retrieval cost at the cost of increasing the storage cost; therefore, the *full index* has a higher storage cost than all the others. When $n = 3$, there are six indices in the *full index*, but there are three indices in the *multiindex* and there is only one index in the *nested index* and the *path index*. Since a *path index* records all the instantiations ending with the object of end of the path and some instantiations may share some references, a *path index* has a certain degree of redundancy and has a higher storage cost than a *multiindex*. Moreover, when $K(i)$ is high enough, the *path index* may need a higher storage cost than the *full index*. Figure 11 shows the storage costs for a *full index* and a *path index*, where $n = 3$, $m(1) = m(2) = m(3) = 1$, $K(2) = K(3) = 1$, and $K(1)$ is varied from 50 to 80. From this figure, we can find that when $K(1) \geq 57$, the *path index* needs a higher storage cost than the *full index*.

5 Optimal Index Configuration

To reduce the high update cost for a long given path, we can split the path into several subpaths and allocates a separate index on each subpath [7, 14]. Given a path, the number of subpaths and the index organization of each subpath define an index configuration. But the increase of the number of indices for subpaths will also increase the retrieval cost for scanning a number of indices, which results in a high retrieval cost. Since a low retrieval cost and a low update cost are always a trade-off in all index organizations, we can establish a cost formula to look for a compromise between these two requirements. In [7], they have

Figure 11: The storage cost for $K(2) = K(3) = 1$ under different value of $K(1)$.

proposed cost formulas to evaluate the costs of various index configurations. However, they do not support the *partial instantiations* and do not consider more than one nested predicates along the path. In this section, we will propose cost formulas to determine the index configuration which can provide the best performance for various applications by taking into account various types of queries along a given path and a set of queries with more than one nested predicates along a given path.

For example, suppose that we have a path = C(1).A(1)....A(3), where path-length = 3. This path can be split in several different ways. All the possible split ways s_i $(1 \leq i \leq 4)$ are grouped as follows, where P_i^j denotes the jth subpath in the ith split way, and on the subpath P_i^j, $P_i^j\_c$ and $P_i^j\_a$ denote the starting class and the ending attribute, respectively:

Split Way	**Subpaths**	**Boundary**
s_1 :	$P_1^1 = C(1).A(1)$	$P_1^1\_c = 1 : P_1^1\_a = 1$
	$P_1^2 = C(2).A(2)$	$P_1^2\_c = 2 : P_1^2\_a = 2$
	$P_1^3 = C(3).A(3)$	$P_1^3\_c = 3 : P_1^3\_a = 3$
s_2 :	$P_2^1 = C(1).A(1).A(2)$	$P_2^1\_c = 1 : P_2^1\_a = 2$
	$P_2^2 = C(3).A(3)$	$P_2^2\_c = 3 : P_2^2\_a = 3$
s_3 :	$P_3^1 = C(1).A(1)$	$P_3^1\_c = 1 : P_3^1\_a = 1$
	$P_3^2 = C(2).A(2).A(3)$	$P_3^2\_c = 2 : P_3^2\_a = 3$
s_4 :	$P_4^1 = C(1).A(1).A(2).A(3)$	$P_4^1\_c = 1 : P_4^1\_a = 3$

Let us consider the split way s_3, we can allocate an index organization such as a $multiindex(MX)$, a $nested\ index(NX)$, a $path\ index(PX)$ and a $full\ index(FX)$, on each subpath. For example, $\{FX$ -> P_3^1, NX -> $P_3^2\}$ denotes that a $full\ index$ is allocated on subpath P_3^1 and a $nested\ index$ is allocated on subpath P_3^2, respectively.

In general, given a path $C(1).A(1)....A(n)$, the set of split ways S is $\{s_1, s_2, ..., s_r\}$, where r denotes the number of split ways of the path, and each split way s_i $(1 \leq i \leq r)$ is $\{P_i^1, ..., P_i^g\}$, where g denotes the number of subpaths of s_i. The organization sets for s_i is $ITS_i = \{IT_i^1$ -> $P_i^1, ..., IT_i^g$ -> $P_i^g\}$, where $IT_i^k \in \{MX, NX, PX, FX\}$, $1 \leq i \leq r$ and $1 \leq k \leq g$.

5.1 Access Cost

Given a path $C(1).A(1)....A(n)$, a split way s_t and its index organization set ITS_t $(1 \leq t \leq r)$, the retrieval cost for a query with a path-expression $C(i).A(i)....A(j) =$ '$O(j+1)$' $(1 \leq i \leq j \leq n)$ is denoted as $Cost\_A_t(i,j)$ and is obtained as follows.

$$\begin{aligned} Cost\_A_t(i, j) = {} & RC_t(i, P_t^l\_c) + \\ & + RC_t(P_t^l\_c, P_t^l\_a) \\ & + RC_t(P_t^{l+1}\_c, P_t^{l+1}\_a) + ... \\ & + RC_t(P_t^m\_c, P_t^m\_a) \\ & + RC_t(P_t^m\_a, j), \end{aligned}$$

where $P_t^{l-1}\_c < i \leq P_t^l\_c$, $P_t^m\_a \leq j < P_t^{m+1}\_a$, $1 \leq l \leq m \leq g$ and g is the number of subpaths of s_t. Moreover, $RC_t(a, b) = RC_a^b$, when $IT_t^y =$ 'FX', $P_t^y\_c \leq a \leq b \leq P_t^y\_a$ and $1 \leq y \leq g$; that is, a $full\ index$ is allocated on subpath P_t^y. If IT_t^y is 'MX', 'NX' or 'PX', $RC_t(a, b)$ can be obtained as described in [3].

5.2 Maintenance Cost

Given a path $C(1).A(1)....A(n)$, a split way s_t and its index organization set ITS_t $(1 \leq t \leq r)$, the update cost for $O(i+1)$ of $O(i)$ $(1 \leq i \leq n)$ is denoted as $Cost\_U_t(i)$ and is obtained as follows.

$$Cost\_U_t(i) = U_t^l(i), \qquad P_t^l\_c \leq i \leq P_t^l\_a.$$

Moreover, $U_t^l(i) = U_i$, when $IT_t^l =$ 'FX' and $1 \leq l \leq g$; that is, a $full\ index$ is allocated on subpath P_t^l. If IT_t^l is 'MX', 'NX' or 'PX', $U_t^l(i)$ can be obtained as described in [3]. As similar to the update cost, the deletion cost $Cost\_D_t(i)$ and the insertion cost $Cost\_I_t(i)$ can be easily obtained.

5.3 Cost Formulas

The total cost of an index configuration with a split way s_t and its a corresponding organization set IT_t for a query with a path-expression $C(i).A(i)....A(j)$ = '$O(j+1)$' ($1 \leq i \leq j \leq n$) is denoted as

$$\alpha Cost\_A_t(i,j) + \sum_{m=1}^{n} \beta_m Cost\_U_t(m) + \sum_{m=1}^{n} \gamma_m Cost\_D_t(m) + \sum_{m=1}^{n} \delta_m Cost\_I_t(m),$$
$$\text{where } \alpha + \sum_{m=1}^{n} \beta_m + \sum_{m=1}^{n} \gamma_m + \sum_{m=1}^{n} \delta_m = 1.$$

Moreover, the total cost of an index configuration with a split way s_t and its a corresponding organization set IT_t for a set queries with path-expressions $C(i_1).A(i_1)....A(j_1)$ = '$O(j_1+1)$', $C(i_2).A(i_2)....A(j_2)$ = '$O(j_2+1)$', ..., $C(i_k).A(i_k)....A(j_k)$ = '$O(j_k+1)$' ($k \geq 1$, $1 \leq q \leq k$ and $1 \leq i_q \leq j_q \leq n$), is denoted as

$$\sum_{q=1}^{k} \alpha_q Cost\_A_t(i_q, j_q) + \sum_{m=1}^{n} \beta_m Cost\_U_t(m) + \sum_{m=1}^{n} \gamma_m Cost\_D_t(m) + \sum_{m=1}^{n} \delta_m Cost\_I_t(m),$$
$$\text{where } \sum_{q=1}^{k} \alpha_q + \sum_{m=1}^{n} \beta_m + \sum_{m=1}^{n} \gamma_m + \sum_{m=1}^{n} \delta_m = 1.$$

Therefore, given a path $C(1).A(1)....A(n)$ and a set of queries with path-expressions $C(i_1).A(i_1)....A(j_1)$ = '$O(j_1+1)$', $C(i_2).A(i_2).....A(j_2)$ = '$O(j_2+1)$', ..., $C(i_k).A(i_k).....A(j_k)$ = '$O(j_k+1)$' ($k \geq 1$, $1 \leq q \leq k$ and $1 \leq i_q \leq j_q \leq n$), the optimal index configuration can be obtained by trying all possible split ways combined with all possible index organization sets and then finding one which can provide the minimum cost.

5.4 Simulation Results

In this subsection, we do several simulations to find an optimal index configuration for queries along a given path against a class-aggregation hierarchy. In all these simulations, we assume that $n = 8$, $N(1) = 200{,}000$, $m(y) = 1$, $K(y) = 2$, ($1 \leq y \leq n$), $UIDL = 8$, $kl = 2$, $ol = 6$, $pp = 4$, $f = 218$, $d = 146$ and $PS = 4096$. Since the cost for an insertion(or a deletion) is in proportion to the cost for an update, we only consider the retrieval and update costs by letting the probabilities of insertion and deletion be 0 in the following simulations. The retrieval probability (denoted as R) is varied form 1 to 0, at the same time, the update probability (denoted as U) is varied from 0 to 1.

Figure 12 shows the optimal index configurations for a query of a simply type with a path-expression $C(1).A(1).....A(8)$ = '$O(8)$'. Figure 13 shows the optimal index configurations for queries of a 2-degree complex type with the same probability to be per-

formed, which path-expressions are $C(1).A(1).....A(8)$ = '$O(8)$' and $C(1).A(1).....A(4)$ = '$O(4)$', respectively. Figure 14 shows the optimal index configurations for queries of a 8-degree complex type with the same probability to be performed, which path-expressions are $C(1).A(1).....A(j)$ = '$O(j)$', $1 \leq j \leq 8$. Figure 15 shows the optimal index configurations for four queries in a general set with the same probability to be performed, which path-expressions are $C(1).A(1).....A(8)$ = '$O(8)$', $C(2).A(2).....A(5)$ = '$O(5)$', $C(3).A(3).....A(7)$ = '$O(7)$' and $C(4).A(4).....A(8)$ = '$O(8)$', respectively. Figure 16 shows the optimal index configurations for queries in a general set with the same probability to be performed, which path-expressions are $C(i).A(i).....A(j)$ = '$O(j)$', $1 \leq i \leq j \leq 8$.

From these figures, in general, we can find that a *full index* on the path $C(1).A(1).....A(8)$ is an optimal index configuration when the retrieval probability is high. The reason is that a *full index* can support any type of queries along a given path with a lower retrieval cost than all the other index organization, and the update probability is so small such that the update cost is neglectable; therefore, the path does not have to be split into several subpaths for reducing update cost. As the retrieval probability is decreased (that is, the update probability is increased), the update cost is not neglectable, which results in a need of a split on the path for reducing the update cost. And for each split subpath, a *full index*, a *nested index* or a *path index* is allocated according to the query status on this subpath. Since a *nested index* and a *path index* cannot support the *partial instantiations*, a *full index* is always a good choice to be allocated on each subpath as shown on Figures 14 and 16. When the update proability is high, a *multiindex* on the path $C(1).A(1).....A(8)$ is a good choice for the reasons of requiring the lower update cost than the others and the neglectable retrieval cost.

6 Conclusion

In this paper, we have proposed a new index organization for evaluating queries, called *full index*, where an index is allocated for each class and its attribute (or nested attribute) along the path. From the analysis results, we have found that a *full index* can support any type of queries along a path against a class-aggregation hierarchy with a lower retrieval cost than all the other index organizations (a *multiindex*, a *nested index* and a *path index*). Moreover, to reduce the high update cost for a long given path, we have split the path into several subpaths and allocated a separate index on each subpath. Since a low retrieval cost and a low update cost are always a trade-off in all index organizations, we have established a cost formula to look for a compromise between these two requirements. In [7], they have

Figure 12: The optimal index configurations for a query of a simple type.

Figure 13: The optimal index configurations for queries of a 2-degree complex type.

Figure 14: The optimal index configurations for queries of a 8-degree complex type.

Figure 15: The optimal index configurations for four queries in a general set.

Figure 16: The optimal index configuration for queries in a general set.

proposed cost formulas to evaluate the costs of various index configurations. However, they do not support the *partial instantiations* and do not consider more than one nested predicates along the path. In this paper, we have proposed cost formulas to determine the best index configuration to provide the best performance for various applications by taking into account various types of queries along a given path and a set of queries with more than one nested predicates along a given path. From the simulation results, in general, a *full index* is an optimal index configuration against a path when the retrieval probability is high. How to provide an efficient index organization for queries along more than one paths against a class-aggregation hierarchy is the future research direction.

References

[1] S. Abiteboul and R. Hull, "IFO: A Formal Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 12, No. 4, 1987, pp. 525-565.

[2] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Information*, Vol. 1, No. 3, 1972, pp. 173-189.

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

[4] E. Bertino and L. Martino, "Object-Oriented Database Management Systems: Concepts and Issues," *Computer*, Vol. 24, No. 4, 1991, pp. 33-47.

[5] E. Bertino, "An Indexing Technique for Object-Oriented Databases," in *Proceedings of IEEE International Conference on Data Engineering*, 1991, pp. 160-170.

[6] E. Bertino and L. Martino, "Object-Oriented Database Systems: Concepts and Architectures," Addison-Wesley Publishing Inc., New York, 1993.

[7] E. Bertino, "Index Configuration in Object-Oriented Databases," *The VLDB Journal*, Vol. 3, No. 3, 1994, pp. 355-399.

[8] E. Bertino and P. Foscoli, "Index Organizations for Object-Oriented Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 2, 1995, pp. 193-209.

[9] R.G.G. Cattell, "Object Data Management: Object-Oriented and Extended Relational Database systems," Addison-Wesley Publishing Inc., New York, 1994.

[10] S. Christodoulakis, J. Vanderbroek, J. Li, T. Li, S. Wan, Y. Wang, M. Papa and E. Bertino, "Development of a Multimedia Information System for an Office Environment," in *Proceedings of the Tenth International Conference on Very Large Data Bases*, 1984, pp. 261-271.

[11] D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11, No. 2, 1979, pp. 121-137.

[12] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima and Y. Yamane, "The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System," *ACM Transactions on Database System*, Vol. 18, No. 1, 1993, pp. 1-50.

[13] A. Karmouch, L. Orozco-Barbosa, N. D. Georganas and M. Goldberg, "A Multimedia Medical Communications System," *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 3, 1990, pp.325-339.

[14] A. Kemper and G. Moerkotte, "Access Support in Object Bases," in *Proceedings of the ACM SIGMOD*, 1990, pp. 364-374.

[15] A. Kemper and G. Moerkotte, "Access Support Relations: an Indexing Method for Object Bases," *Information Systems*, Vol. 17, No. 2, 1992, pp.117-145.

[16] W. Kim and F. Lochovsky, "Indexing Techniques for Object-Oriented Databases," in W. Kim and F. Lochovsky (ed.), *Object-oriented concepts, databases, and applications*, Addison-Wesley Publishing Inc., New York, 1989.

[17] C. C. Low, B. C. Ooi, and H. Lu, "H-Trees: a Dynamic Associative Search Index for OODB," *ACM SIGMOD*, 1992, pp. 134-143.

[18] D. Maier and J. Stein, "Indexing in an Object-Oriented Database," in *Proceedings of IEEE Workshop on Object-Oriented DBMSs*, 1986, pp. 171-182.

[19] C. Meghini, F. Rabitti and C. Thanos, "Conceptual Modeling of Multimedia Documents," *IEEE Computer*, Oct. 1991, pp. 23-30.

[20] E. Oomoto and K. Tanaka, "OVID: Design and Implementation of a Video-Object Database System," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 4, 1993, pp. 629-643.

[21] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems*, Vol. 12, No. 2, 1987, pp. 218-246.

[22] D. Woelk, W. Kim and W. Luther. "An Object-Oriented Approach to Multimedia Databases," in *Proceedings of ACM SIGMOD*, 1986, pp. 311-325.

[23] D. Woelk, "Multimedia Information Management in an Object-Oriented Database System," in *Proceedings of the 13th VLDB Conference*, 1987, pp. 319-329.

[24] A. Yoshitaka, S. Kishida, M. Hirakawa and T. Ichikawa, "Knowledge-Assisted Content-Based Retrieval for Multimedia Databases," *IEEE Multimedia*, Winter 1994, pp. 12-21.

[25] Z. Xie and J. Han, "Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases," in *Proceedings of the 20th VLDB Conference*, 1994, pp. 522-533.