# Index Organizations for Object-Oriented Database Systems

Elisa Bertino and Paola Foscoli

***Abstract*—In this paper we present an indexing technique providing support for queries involving complex, nested objects and inheritance hierarchies. This technique is compared with two techniques obtained from more traditional organizations. The three techniques are evaluated using an analytical cost model. The discussion will be cast in the framework of object-oriented databases. However, results are applicable to data management systems characterized by features such as complex objects and inheritance hierarchies.**

***Index Terms*—Object-oriented databases, complex objects, inheritance, query languages, query processing, data structures.**

## I. INTRODUCTION

A number of different advanced data management systems have been developed in recent years. Most of those systems support data models semantically richer than the relational model. Examples are the object-oriented data model [3], knowledge-representation models [7], [11], logical-object models [6], [12], [21], and post-relational models [19]. Despite various differences those models share three basic characteristics whose advantages have been widely discussed in the literature. The first is the possibility of directly modeling complex, nested objects. The second is to organize classes (types) into inheritance hierarchies. The third is to support high-level declarative query languages. This is an important feature which, among others, has been retained from the relational model. Experience in the framework of relational DBMSs has shown that high-level declarative query languages substantially contribute to reduce application development times.

An important issue related to query languages concerns optimization techniques and access structures able to reduce query processing costs. This issue is a central one in data management architectures, since performance is still a key factor. In particular, while there has been a lot of research for new query languages, no comparable amount of research has been reported concerning indexing techniques. In this paper we make some contributions to the problem of defining indexing techniques for advanced data management systems. Our discussion will be in the framework of object-oriented database systems. The main novelty of the indexing technique presented here is that it provides an integrated support for queries involving both nested attributes of objects and inheritance hierarchies. In essence, it allows a query containing a predicate on a nested attribute and involving several classes in a given inheritance hierarchy to be solved with a single index lookup. This technique has been defined as a combination of ideas from class-hierarchy index [17], path-index [1], and join index [20] organizations.

Indexing techniques defined in the framework of object-oriented DBMSs can be classified as *structural* [18], [1], [2], [15], [20] and *behavioral* [5], [6], [16]. Structural indexing is based on object attributes. Structural indexing is very important because most object-oriented query languages allow query predicates to be issued against object attributes. Structural indexing techniques proposed so far can be classified into techniques providing support for nested predicates,[1] such as the ones presented in [1], [15], [18], and indexing techniques supporting queries issued against an inheritance hierarchy [17]. Most indexing techniques efficiently supporting the evaluation of nested predicates are based on pre-computing functional joins among objects [13]. Some of those techniques require a single-index lookup to evaluate a nested predicate [1], [15]. However, their update costs may be rather high. Others, such as the organization proposed in [18], require scanning a number of indices equal to the number of classes to be traversed to evaluate the nested predicate. However, their update costs are not very high. Indexing techniques for queries against inheritance hierarchies are based on allocating a single index on the entire class hierarchy rather than allocating an index for each class in the hierarchy. Behavioral indexing aims at providing efficient execution for queries containing method invocations. It is based on pre-computing or caching a method results and storing them into an index. The major problem of this approach is how to detect changes to objects that invalidate the results of a method. Some approaches can be found in [5], [14], [16].

The indexing technique presented in this paper is structural, in that indices are allocated on object attributes. The main novelty of this technique is that it provides an integrated support for queries along both aggregation hierarchies and inheritance hierarchies. A preliminary version of this technique was presented in [2]. However, the technique presented in [2] was defined only for the case of single-valued attributes. This restriction is removed in the technique presented here. Moreover, in [2] a preliminary cost model was presented only for the retrieval operation. Here, a more general model is presented that evaluates the costs of retrieval, delete, and insert opera-

Manuscript received May 27, 1992; revised January 21, 1993.
The authors are with Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy; e-mail bertino@disi.unige.it.
IEEECS Log Number K95022.

[1] A nested predicate is a predicate against a nested attributes of an object. A nested attribute of an object $O$ is an attribute of some object (directly or indirectly) referenced by $O$.

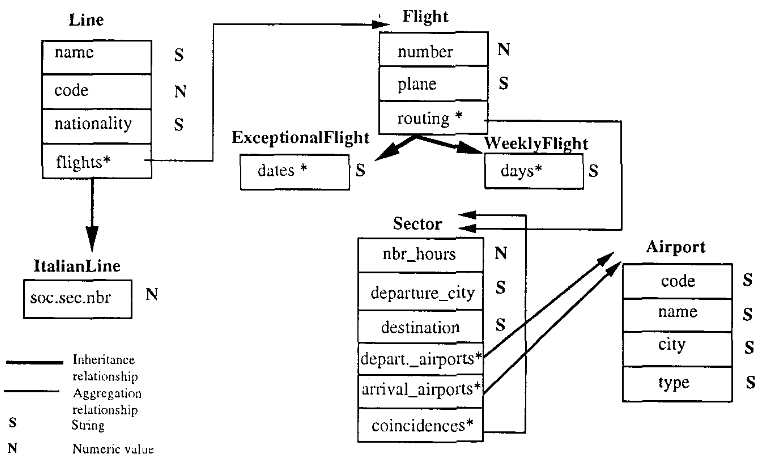


Fig. 1. Object-oriented database schema example.

tions, and releases many of the limiting assumptions of the model used in [2]. Finally, we note that even though the proposed indexing technique is structural, it could be combined with behavioral indexing techniques.

The remainder of this paper is organized as follows. Section II describes the technique introduced in this paper. In addition, two traditional techniques are presented that will be used as comparison. Section III presents cost models for the three indexing techniques. A comparison of those techniques is then presented in Section IV. Finally, Section V summarizes the major conclusions of the paper and outlines future work.

## II. Index Organizations

In this section we first present some preliminary definitions from [2]. Then we present the three indexing techniques and we describe how operations, such retrieval or delete, are performed on these techniques.

### A. Preliminary Definitions and Notations

$C^*$ denotes the set of classes in the inheritance hierarchy rooted at class $C$. As an example consider the object-oriented schema in Fig. 1:

Flight* = {Flight, ExceptionalFlight, WeeklyFlight}

A path represents a branch in an aggregation hierarchy; more formally a path $\mathcal{P}$ is defined as $C_1.A_1.A_2 \ldots .A_n$ ($n \geq 1$) where:

- $C_1$ is class in the database schema
- $A_1$ is an attribute of class $C_1$
- $A_i$ is an attribute of a class $C_i$ such that $C_i$ is the domain of the attribute $A_{i-1}$ of class $C_{i-1}$, $1 < i \leq n$.

Moreover

- len ($\mathcal{P}$) = $n$ denotes the length of the path
- class ($\mathcal{P}$) = $C_1 \cup \{C_i \mid C_i$ is domain of attribute $A_{i-1}$ of class $C_{i-1}$, $1 < i \leq n\}$
- scope ($\mathcal{P}$) = $\bigcup_{C_i \in class(\mathcal{P})} C_i^*$

  class $C_1$ is the root of the scope
- given a class $C$ in the scope of $\mathcal{P}$, the *position* (shortly pos) of $C$ is given by an integer $i$, such that $C$ belongs to the inheritance hierarchy rooted at class $C_i$, where $C_i \in$ class($\mathcal{P}$ ).

The scope of a path simply represents the set of all classes along the path and all their subclasses. The following is an example path for the schema in Fig. 1:

P = Line.flights.routing.arrival_airports.name len(P) = 4
class(P) = {Line, Flight, Sector, Airport}
scope(P) = {Line, ItalianLine, Flight, ExceptionalFlight, WeeklyFlight, Sector, Airport}
Line is the class root of the scope
pos(Line) = pos(ItalianLine) = 1
pos(Flight) = pos(WeeklyFlight) = pos(ExceptionalFlight) = 2
pos(Sector) = 3 pos(Airport) = 4

In the remainder of the discussion we will distinguish between the *instances* and the *members* of a class. An object is instance of a class $C$ if $C$ is the most specialized class associated with the object in a given inheritance hierarchy. An object is member of a class $C$ if it is an instance of $C$ or of some subclass of $C$. Therefore, the instances of a class $C$ are all objects that do not belong to any of the subclasses of $C$.

### B. Index Definitions

All indexing organizations we present here have the goal of supporting an efficient evaluation of nested predicates against a class, or set of classes. Two examples of such queries against the schema of Fig. 1 are the following:

*Retrieve all italian lines having a flight departing from the airport named Leonardo da Vinci* (Q1)

*Retrieve all lines (including the italian ones) having a flight departing from the airport named Leonardo da Vinci* (Q2).

NESTED-INHERITED INDEX (NIX). Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$, a nested-inherited index associates with a value $v$ of the attribute $A_n$ identifiers (OIDs) of instances of each class in the scope of $\mathcal{P}$ having $v$ as value of the (nested) attribute $A_n$. Fig. 3 provides an example of logical index entries for the hypothetical database represented in Fig. 2 whose schema is in Fig. 1.

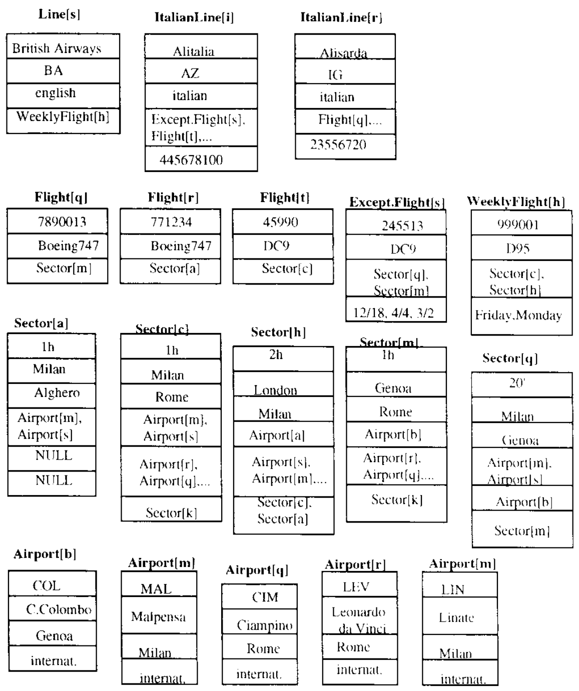


Fig. 2. A hypothetical database.

The nested-inherited index, as the nested index and the path index [1], and the access relation [15], supports fast retrieval operations. However, unlike those two organizations, the nested-inherited index does not require object traversals for update operations, because of some additional information that are stored in the index. The format of a non-leaf node has a structure similar to that of traditional indices based on B$^+$-tree. The record in a leaf node, called *primary record*, has a different structure. It contains the following information:

- record length
- key length
- key value
- class-directory
- for each class $C$ in the path scope, the number of elements of the list $L$, and the list $L$ which is structured as follows. Let $i$ be position of $C$ in the path scope; then

  —if $A_i$ is a single-valued attribute, or $i = n$, $L$ contains the OIDs of instances of class $C$ that hold the key value for the (nested) attribute $A_n$.

  —if $A_i$ is a multi-valued attribute, the elements of $L$ are pairs (OID, *numchild*) where OID identifies an object $O$ that holds the key value for the nested attribute $A_n$, and *numchild* is the number of children of $O$ that hold the key value for the nested attribute $A_n$.

The class-directory contains a number of entries equal to the number of classes having instances with the key value in the indexed attribute. For each such class $C_i$ an entry in the directory contains:

- the class identifier
- the offset in the primary record where the list $L$ of OIDs of $C_i$'s instances (or of pairs, if $A_i$ is multi-valued and $i < n$) is stored
- the pointer to an auxiliary record where the list of parents is stored for each instance of $C_i$. An auxiliary record is allocated for each class, except for the class root of the path and for its subclasses. An auxiliary record consists of a sequence of 4-tuples. A 4-tuple has the form:

  (OID, pointers to primary records, no-oids, $\{\mathrm{p}-\mathrm{oid}_1, \ldots, \mathrm{p}-\mathrm{oid}_n\}$).

  There are many 4-tuples as the number of instances of $C_i$. For an object $O$, instance of $C_i$, the tuple contains the identifier of $O$ (i.e., OID), the pointers to $k$ primary records, where $k$ is the number of key values in the nested attribute $A_n$ of $O$, the number of parent objects of $O$, the list of OIDs of the parents of $O$. In the 4-tuple definition above, $p - \mathrm{oid}_j$ denotes the $j$th parent of $O$.

Therefore the organization of a leaf-node record is similar to that of the class-hierarchy index defined in [17], in that there is a directory within the leaf-node record associating with each class the offset within the record where the OIDs of instances of the class are stored. The main difference, however, is that a nested-inherited index can be used for queries on all class hierarchies found along a given path. By contrast, a class-hierarchy index is allocated on a single class-hierarchy. Therefore, if a path has length $n$, the number of class-hierarchy indices allocated would be $n$.

Auxiliary records are stored in pages different than those storing the primary records. Given a primary record, there are several auxiliary records that are *connected* to it. On the auxiliary records a second B$^+$-tree is superimposed. The second B$^+$-tree indexes the 4-tuples based on the object-identifier that appears as first element of 4-tuples. Therefore, the index organization consists actually of two indices. The first, called *primary index*, is indexed on the values of the attribute $A_n$. It associates with a value $v$ of $A_n$, the set of OIDs of instances of all classes that have $v$ as value of the (nested) attribute $A_n$. The second index, called *auxiliary index*, has OIDs as indexing keys. It associates with the OID of an object $O$ the list of OIDs of the parents of $O$. The leaf-node records in the primary index contain pointers to the leaf-node records in the auxiliary index, and vice-versa. Therefore, the primary index is inverted with respect to the values of (nested) attribute $A_n$ and it is used for retrieval operations. The secondary index is inverted with respect to OIDs of instances of all classes (except for the class, root of the path, and for its subclasses) in the scope of a given

- **C.Colombo** {ItalianLine[i], ExceptionalFlight[s], Sector[q], Airport[b]}
- **Ciampino** {Line[s], ItalianLine[i], ItalianLine[r], Flight[t], Flight[q], ExceptionalFlight[s], WeeklyFlight[h], Sector[c], Sector[m], Airport[q]}
- **Leonardo da Vinci** {Line[s], ItalianLine[i], ItalianLine[r], Flight[t], Flight[q], ExceptionalFlight[s], WeeklyFlight[h], Sector[c], Sector[m], Airport[r]}
- **Linate** {Line[s], WeeklyFlight[h], Sector[h], Airport[s]}
- **Malpensa** {Line[s], WeeklyFlight[h], Sector[h], Airport[m]}

Fig. 3. Example of logical entries for a nested-inherited index allocated on path $P$ = Line.flights.routing.arrival_airports.name.

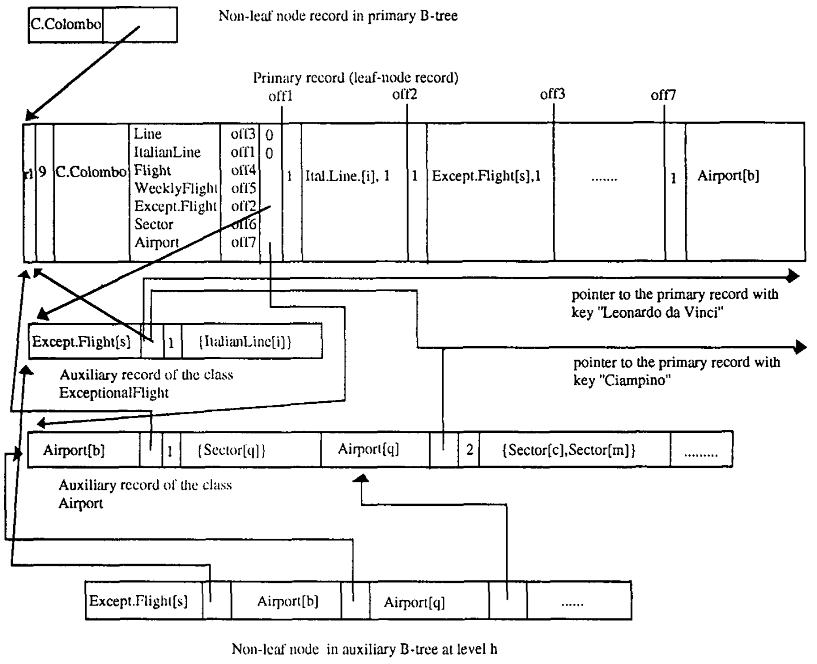


Fig. 4. Example of index content for a nested-inherited index.

path. Basically, the secondary index is used to determine all primary records where the OIDs of a given instance are stored in order to efficiently perform delete and insert operations. A similar organization is used for join indices [20]. As discussed in [20], two copies can be allocated of a join index. There is a copy being inverted on tuple identifiers for each one of the participating relations. The main difference between the NIX organization and the organization proposed in a previous paper [2] consists of the structure of the auxiliary $B^+$-tree. In the NIX organization the structure of the auxiliary $B^+$-tree has been extended to deal with multi-valued attributes that were not supported by the organization defined in [2]. Fig. 4 presents an example of index contents for a nested-inherited index allocated on path $P$ = Line.flights.routing.arrival_airports.name.

MULTIINDEX (MX). This organization consists of allocating an index on each class in the scope of the path. Therefore the number of allocated indices is equal to the number of classes in the scope of the path. In the example of path $P$ = Line.flights.routing.arrival_airports.name, seven indices would be allocated. Given a path $\mathcal{P} = C_1.A_1.A_2 \dots A_n$ and a class $C$ in the scope of $\mathcal{P}$, such that $C$ has position $i$ ($1 \leq i \leq n$), an index on $C$ will associate with the values of the attribute $A_i$ the OIDs of instances of class $C$. Since all attributes in the path (except $A_n$) have as domain non primitive classes, the key values are OIDs for all indices, except for the ones allocated on the last class in the path and its subclasses. An index whose key values are OIDs is called an *identity index*. The indices allocated for the last class and its subclasses may be identity indices or not depending on the domain of $A_n$.

INHERITED-MULTIINDEX (IMX). This organization represents an enhancement to the multiindex organization and consists of using a class-hierarchy index [17] for each inheritance hierarchy found along the path. Given a path $\mathcal{P} = C_1.A_1.A_2 \dots A_n$, there is an index on each class $C_i$ in class ($\mathcal{P}$ ). An index on $C_i$ associates with values of the attribute $A_i$ the

- Index for class Line on the attribute "flights"

  Entry: (**WeeklyFlight[h]**, {Line[s]})

- Index for class ItalianLine on the attribute "flights"

  Entries: (**ExceptionalFlight[s]**, {ItalianLine[i]}), (**Flight[q]**, {ItalianLine[r]}), (**Flight[t]**, {ItalianLine[i]})

- Index for class Flight on the attribute "routing"

  Entries: (**Sector[a]**, {Flight[r]}), (**Sector[c]**, {Flight[t] }), (**Sector[m]**, {Flight[q] })

- Index for class ExceptionalFlight on the attribute "routing"

  Entries: (**Sector[m]**, {ExceptionalFlight[s] }), (**Sector[q]**, {ExceptionalFlight[s] })

- Index for class WeeklyFlight on the attribute "routing"

  Entries: (**Sector[c]**, {WeeklyFlight[h]}), (**Sector[h]**, {WeeklyFlight[h],})

- Index for class Sector on the attribute "arrival_airports"

  Entries:(**Airport[b]**, {Sector[q]}), (**Airport[m]**, {Sector[h]}), (**Airport[q]**,{Sector[c], Sector[m]}), (**Airport[r]**, {Sector[c], Sector[m]}), (**Airport[s]**, {Sector[h]})

- Index for class Airport on the attribute "name"

  Entries: (**C.Colombo**, {Airport[b]}), (**Ciampino**, {Airport[q]}), (**Leonardo da Vinci**, {Airport[r]}), (**Linate**, {Airport[s] }), (**Malpensa**, {Airport[m]})

(a) Multiindex

- Index for class Line and its subclasses on the attribute "flights"

  Entries: (**WeeklyFlight[h]**, {Line[s]}), (**ExceptionalFlight[s]**, {ItalianLine[i]}), (**Flight[t]**, {ItalianLine[i]}), (**Flight[q]**, {ItalianLine[r]})

- Index for class Flight and its subclasses on the attribute "routing"

  Entries: (**Sector[a]**, {Flight[r]}), (**Sector[c]**, {Flight[t], WeeklyFlight[h]}), (**Sector[m]**, {Flight[q], ExceptionalFlight[s]}), (**Sector[q]**, {ExceptionalFlight[s]}), (**Sector[h]**, {WeeklyFlight[h]})

- Index for class Sector and its subclasses on the attribute "arrival_airports"

  Entries: (**Airport[b]**, {Sector[q]}), (**Airport[m]**, {Sector[h]}), (**Airport[q]**, {Sector[c], Sector[m]}), (**Airport[r]**, {Sector[c], Sector[m]}), (**Airport[s]**, {Sector[h]})

- Index for class Airport and its subclasses on the attribute "name"

  Entries: (**C.Colombo**, {Airport[b]}), (**Ciampino**, {Airport[q]}), (**Leonardo da Vinci**, {Airport[r]}), (**Linate**, {Airport[s]}), (**Malpensa**, {Airport[m]})

(b) Inherited-multiindex

Fig. 5. Example entries for the multiindex (a) and for the inherited-multiindex (b) organizations.

OIDs of instances of $C_i$ and of all its subclasses. The number of indices allocated is equal to the path length. As an example, consider the path $P$ = Line.flights.routing.arrival_airports.name. For this path there would be: a class-hierarchy index allocated on class Line indexing instances of class Line and ItalianLine; a class-hierarchy index allocated on class Flight indexing instances of classes Flight, ExceptionalFlight, and WeeklyFlight; a class-hierarchy index allocated on class Sector; a class-hierarchy index allocated on class Airport (note that the class-hierarchy indices on classes Sector and Airport reduce to single-class indices, since Sector and Airport have no subclasses). Fig. 5 provides examples of index entries for the multiindex and inherited-multiindex organizations.

## C. Operations

In this subsection, we describe retrieval, delete, and insert operations for the three organizations.

### *C.1. Retrieval*

#### *Nested-Inherited Index*

The nested-inherited index supports a fast evaluation of predicates on the indexed attribute for queries having as target any class, or class hierarchy, in the scope of the path ending with the indexed attribute. As an example, suppose that an index is allocated on the path $P$ = Line.flights.routing.arrival_airports.name and that a query is issued that retrieves all the instances of class ItalianLine having a flight that lands at the airport "Leonardo da Vinci." This query is processed by first executing a lookup on the primary index with key value equal to "Leonardo da Vinci." The primary record is then accessed. A lookup in the class directory is executed to determine the offset where the OIDs of ItalianLine are stored. Then these OIDs are fetched and returned as result of the query. They are {ItalianLine[i], ItalianLine[r]}. We now consider a query like the previous one but with a class hierarchy rooted at class Line as target class. The same previous steps are executed. The only difference is that the class-directory lookup is executed for the classes Line and ItalianLine and two different portions of the record are accessed, one for each offset obtained from the class-directory lookup. The results of the query are {ItalianLine[i], ItalianLine[r], Line[s]}.

*Multiindex*

Retrieval operations on the multiindex organization are very expensive and they may need the lookup of several indices. Let us consider again the query retrieving all instances of class ItalianLine having a flight that lands at the airport "Leonardo da Vinci." This query is executed by first scanning the index allocated on the class Airport. This scanning returns the OID Airport[r]. A lookup of such key value is executed on the index allocated on the class Sector obtaining the keys {Sector[c], Sector[m]}. Then a lookup is executed on each one of the indices allocated on the classes Flight, ExceptionalFlight and WeeklyFlight with search keys equal to the ones obtained previously. These three index lookups return the following set of OIDs: {WeeklyFlight[h], ExceptionalFlight[s], Flight[t], Flight[q]}. Finally, these OIDs are used to execute the lookup on the index allocated on class ItalianLine. Note that this query needs the lookup of six indices, whereas if the query were applied to the class hierarchy rooted at class Line, the total index lookups would have been seven.

*Inherited-Multiindex*

The inherited-multiindex organization is generally more efficient than the multiindex since it needs the lookup of a smaller number of indices. If we analyze again the previous query the scanned indices are those allocated on the classes Airport, Sector, Flight and Line (note that the index allocated on class Line also indexes instances of class ItalianLine). Therefore, only four index lookups must be executed. Note that if the query were applied to the class hierarchy rooted at class Line, the total index lookups would have been still four.

### *C.2. Delete*

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and a class $C$, belonging to the scope of $\mathcal{P}$ and having position $i$, suppose that an object $O$, instance of $C$, is deleted. The execution of such operation for the three indices is described in the following subsections.

*NestedInherited Index*

The overall effect of a delete operation on the index must be that the OID of the object $O$ is eliminated from every primary record containing it, while all instances referencing $O$ are eliminated from the same primary records only if they have no other children contained in such primary records. The tuples for instances referencing $O$ (except for the instances of class root of the path and of its subclasses) are updated, while the tuple of $O$ is deleted from the auxiliary record corresponding to the class $C$. Finally, $O$ must be eliminated from the parent list of the its children. The following steps are executed:

1) The set of values $SCH$ of the attribute $A_i$ of $O$ is determined. If $A_i$ is a single-valued attribute the set of values $SCH$ contains a single value.
2) The auxiliary index is accessed with the values of $SCH$ and $O$ as key values.
3) The tuples corresponding to the values of $SCH$, retrieved in the previous lookup, are modified by removing $O$ from the lists of parents in the tuples.
4) The tuple of $O$ is accessed; the OIDs of $O$'s parents and the set $S$ of pointers to the primary records are determined. Then the tuple is removed. The OIDs of $O$ and of its parents are stored in a temporary list *listparent.*
5) The following steps are executed as long as the list *listparent* is not empty:

   (a) For each primary record $P$ whose address is contained in $S$ the following steps are executed:

   - the primary record $P$ is accessed;
   - a lookup is executed on the class-directory in the primary record $P$ to determine the offsets where the OIDs in *listparent* are stored; if the external loop is just started the class of $O$ is also retrieved from the class-directory;
   - for each OID in *listparent* the following activity is executed:

     if OID identifies $O$ or OID is stored in the primary record without the field "numchild," OID is removed from the list of OIDs stored at the offset corresponding to the class $C$ determined at the previous step. Moreover, if OID identifies a parent of $O$ it is inserted in an auxiliary list $L$ with the address of the primary record $P$. In the other cases, instead, the value of the field "numchild" associated with the OID, stored at the offset determined previously, is decremented by one and if it becomes zero the pair (OID, numchild) is removed from the primary record. If OID is not a member of the class root of the path, it is inserted in an auxiliary list $L$ with the address of the primary record $P$.

   (b) The auxiliary index is accessed with the OIDs of $L$ as search keys and the list *listparent* is emptied.

   (c) The tuples of the OIDs of $L$ are accessed and the pointers to the primary records from which the OIDs have been removed are deleted from the tuples. Then the lists of parents in the tuples are determined and are concatenated in a new list *listparent.* Then step 5 is again executed.

Another strategy for retrieving the tuples consists of scanning the auxiliary records whose addresses are obtained from the class-directory stored in the primary record. This second method is valid when a record is stored in only one page or in a small number of pages.

As an example suppose that the object Sector[h] must be removed from the database and that a nested-inherited index is allocated on the path $P$ = Line.flights.routing.arrival_airports.name. First a lookup of the auxiliary B$^+$-tree is executed with the OIDs Sector[h], Airport[m] and Airport[s] as search keys.[2]

[2] Airport[m] and Airport[s] are the OIDs found as values of attribute 'arrival_airport' of object Sector[h]. Note that determining those OIDs does not require additional costs, since Sector[h] must be accessed to be marked as deleted.

The tuples corresponding to Airport[m] and Airport[s], returned by the index scan, are updated by deleting the OID Sector[h] from the list of parents of Airport[m] and of Airport[s]. Then the tuple associated with Sector[h] is accessed. From that tuple the parent list {WeeklyFlight[h]}, and the list of pointers to the primary records are extracted. The primary records associated with the tuple of Sector[h] are those with key value "Malpensa" and "Linate." Then the OIDs of Sector[h]'s parents (i.e., {WeeklyFlight[h]}), and Sector[h] are stored in the list *listparent*. The primary record with key value equal to "Malpensa" is accessed. The classes Flight and Sector are retrieved from the class-directory of the primary record and the offsets are determined of the primary record where the lists of pairs associated with such classes are stored. The pair (Sector[h], 1) is removed from the list corresponding to the class Sector. Since the pair (WeeklyFlight[h], 1) corresponding to the class WeeklyFlight has the value 1 in the field "numchild," it is also deleted from the list. The OID WeeklyFlight[h] is inserted into an auxiliary list $L$ together with the address of the primary record having key value "Malpensa." The same procedure is applied to the primary record with key value "Linate." Therefore only the pairs (WeeklyFlight[h], 1) and (Sector[h], 1) are deleted from the record with key value "Linate," and the address of such primary record is added to the OID WeeklyFlight[h] in the list $L$.

Now the OIDs in $L$, in this case only WeeklyFlight[h], are retrieved in the auxiliary $B^+$-tree and the list *listparent* is emptied. The tuple of WeeklyFlight[h] is accessed and the pointers to the primary records stored in $L$ are deleted from the list of pointers present in the tuple. Then the list of parents in the tuple is determined. Such list contains only the OID Line[s]. This OID is then removed from the primary records having key values "Malpensa" and "Linate" and therefore the delete operation completed.

*Multiindex and Inherited-Multiindex*

The steps of the algorithm for deleting an object are the following:

1) The set of values, $SCH$, of attribute $A_i$ of $O$ is determined. If $A_i$ is a single-valued attribute the set of values $SCH$ contains only one value.
2) The index allocated on the class $C$ is accessed with the values of $SCH$ as key values.
3) The leaf-node records corresponding to the values of $SCH$ are accessed and the OID of $O$ is removed from the list of OIDs present in such records.
4) The indices allocated on the classes of position $i - 1$ in the path are accessed with the OID of $O$ as key value. Note that in an inherited-multiindex organization there is only one index corresponding to the classes of position $i - 1$.
5) In every index the leaf page determined at the previous step is modified by removing the record corresponding to the OID of $O$ from the page. Note that such record does not exist, if $O$ is not referenced by any object of classes with position $i - 1$.

For example, suppose that Flight[r] is removed. As first step, the index allocated on the class Flight is scanned with key value equal to the OID Sector[a]. The OID Flight[r] is removed from the record determined at the previous step. Then in the multiindex organization the indices allocated on the classes Line and ItalianLine are accessed with Flight[r] as search key, while in the inherited-multiindex organization the index allocated on the hierarchy rooted at the class Line is scanned with search key Flight[r]. Since Flight[r] is pointed by no object, no updates must be performed on the index.

*C.3. Insert*

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and a class $C$, belonging to the scope of $\mathcal{P}$ and having position $i$, suppose that an object $O$, instance of class $C$, must be inserted. The execution of such operation for the three organizations is described in the following.

*Nested-Inherited Index*

The steps that are executed to insert an object in the database are the following:

1) The set of values $SCH$ of attribute $A_i$ of $O$ is determined. If $A_i$ is a single-valued attribute the set of values $SCH$ contains only one value.
2) The auxiliary index is searched using as key values the values in $SCH$.
3) The tuples corresponding to the values of in set $SCH$ are retrieved and modified by inserting $O$ in the list of parents present in the tuples and the set $S$ of pointers to the primary records are determined.
4) For each primary record $P$ whose address is contained in $S$ the following steps are executed:
   - the primary record $P$ is accessed;
   - a lookup is executed on the class-directory in the primary record $P$ to determine the offset and the address of the auxiliary record corresponding to class $C$;
   - the OID of $O$ is inserted in the list of OID (or of pairs if the attribute $A_i$ is multi-valued and if $i < n$); if the list consists pairs, the value of the field "numchild" is initialized adequately.
5) A new tuple with OID of $O$ as first element is inserted in the auxiliary record corresponding to the class $C$ whose address has been found at the step 4.

As an example suppose that the object Flight[m] that points to the object Sector[q] is inserted. Then the OID Sector[q] is retrieved from the auxiliary $B^+$-tree and the tuple of Sector[q] is updated inserting the OID Flight[m] in the parents list of Sector[q]. From the tuple, the addresses of the primary records storing the OID Sector[q] are determined. In this case there is only one record associated with Sector[q] having "C.Colombo" as key value. This primary record is accessed and a lookup of its class-directory is executed with Flight as key value. In this way the offset and the address of the auxiliary record associated with class Flight is determined. Using that offset, the list of pairs of Flight is accessed and the pair (Flight[m],1) is inserted into this list. Then the auxiliary record associated with the class Flight is updated inserting a new tuple with Flight[m] as first element and the empty list as parent list.

*Multiindex and inherited-multiindex*

The insert operation is executed as follows:

1) The set of values *SCH* of attribute $A_i$ of $O$ is determined. If $A_i$ is a single-valued attribute the set of values *SCH* contains only one value.
2) The index allocated on the class $C$ is searched using the values in *SCH* as key values.
3) The leaf-node records corresponding to the values of *SCH* are accessed and the OID of $O$ is inserted in the list of OIDs present in such records.

For example, suppose that the object, identified by Flight[z] and referencing Sector[h], is to be inserted. The OID Sector[h] is retrieved from the index allocated on the class Flight. Once the leaf-node record corresponding to such search key is determined, Flight[z] is inserted in the list of OIDs in the record.

## III. Cost Models

In this section we first introduce parameters and assumptions of the model. Then we present cost formulations for retrieval, delete, and insert operations.

### A. Parameters

*Logical data parameters*

The parameters, describing the classes in a path $\mathcal{P} = C_1.A_1....A_n$, are listed in the following. There are a large number of parameters. This is due to the need to precisely characterize the *topology* of references among objects.[3]

- $nc_i$—Number of classes in the inheritance hierarchy rooted at class $C_i$, $1 \le i \le n$.
- $D_{i,j}$—Number of distinct values for attribute $A_i$ of class $C_{i,j}$, $1 \le i \le n$, $1 \le j \le nc_i$.
- $D^t_{i,j}$—Average number of instances of class $C_{i+1,t}$ that are referenced as values of attribute $A_i$ of instances of class $C_{i,j}$, $1 \le i < n$, $1 \le j \le nc_i$ and $1 \le t \le nc_{i+1}$. We make the assumption that $D^t_{i,j} = D_{i,j}/nc_{i+1}$.
- $D_i$—Number of distinct values for attribute $A_i$ for all instances in the inheritance hierarchy rooted at class $C_{i,1}$; $D_i = \sum_{j=1}^{nc_i} D_{i,j}$.
- $N_{i,j}$—Cardinality of class $C_{i,j}$, $1 \le i \le n$, $1 \le j \le nc_i$.
- $Nh_i$—Number of members of class $C_{i,1}$, $1 \le i \le n$; $Nh_i = \sum_{j=1}^{nc_i} N_{i,j}$.
- $fan_{i,j}$—Average number of elements contained in the attribute $A_i$ for instances of class $C_{i,j}$, when $A_i$ is multivalued, $1 \le i \le n$ and $1 \le j \le nc_i$.
- $fan_i$—Average number of elements contained in the attribute $A_i$ for members of class $C_{i,1}$, when $A_i$ is multivalued, $1 \le i \le n$. The difference of this parameter with the previous is that this parameter is obtained as the average evaluated on all members of a class hierarchy, while in the previous the average is for each class.
- $d_{i,j}$—number of instances of class $C_{i,j}$, having a value different than NULL for attribute $A_i$, $1 \le i \le n$ and $1 \le j \le nc_i$.
- $d_i$—number of members of class $C_{i,1}$, having a value different than NULL for attribute $A_i$, $1 \le i \le n$; $d_i = \sum_{j=1}^{nc_i} d_{i,j}$
- $k_{i,j}$—Average number of instances of class $C_{i,j}$, having the same value for attribute $A_i$, $1 \le i \le n$ and $1 \le j \le nc_i$; $k_{i,j} = \lceil (d_{i,j} * fan_{i,j}) / D_{i,j} \rceil$.
- $kh_i$—Average number of members of class $C_{i,1}$, having the same value for attribute $A_i$, $1 \le i \le n$.

*System parameters*

- OIDL—Object identifier length.
- $P$—Page size.

*Index parameters*

- $f$—Average fanout of a non-leaf node.
- $h_A$, $h_P$, $h_{i,j}$, $h_i$—Internal height, respectively, of the auxiliary index and of the primary index in the NIX organization, of the index allocated on the class $C_{i,j}$ in the MIX organization, and of the index allocated on the class hierarchy rooted at $C_{i,1}$ in the IMX organization. The internal height represents the height of the index minus one. In other words the internal height represents the number of levels of the index excluding the leaf-node level.
- $kl$—Key length.
- $pp$—Pointer length.
- $of$—Offset length.
- $XNI$—Average length of a leaf-node record of the primary B$^+$-tree in the NIX organization.
- $XAUX_{i,j}$—Average length of an auxiliary record associated with class $C_{i,j}$ in the NIX organization.
- $XIM_i$—Average length for a leaf-node record for the $i$th index ($1 \le i \le n$) in the IMX organization.
- $XM_{i,j}$—Average length of a leaf-node record for the index on class $C_{i,j}$ ($1 \le i \le n$, $1 \le j \le nc_i$) in the MX organization.
- $LP_P$, $LP_A$—Number of leaf-node pages of the primary B$^+$-tree and the auxiliary B$^+$-tree of the NIX organization, respectively.
- $np$—Number of pages occupied by a record when the record size is greater than the page size; $np = \lceil recordsize / P \rceil$.
- $n_l$—Number of nodes at level $l$ in a B$^+$-tree.

*Assumptions*

Most of the assumptions made are commonly found in analytical models for database access structures. In particular, we make the assumption that key values are uniformly distributed among instances of the same class. Moreover, the cardinality of instances of a class is not correlated with cardinalities of instances of other classes belonging to the same inheritance hierarchy.

*Derived Parameters*

These parameters are derived from the input ones and are particularly important since they precisely characterize the

[3] In the remainder of the discussion, given an inheritance hierarchy having position $i$ in a path, the notation $C_{i,1}$ denotes the class root of the inheritance hierarchy.

dimension of records for the NIX organization. Because of space restriction, we cannot present here the derivations for these parameters (they can be found in [4]).

- $\overline{k}_{i,j}$—Average number of instances of class $C_{i,j}$ having the same value for the nested attribute $A_n$ ($1 \le i \le n$, $1 \le j \le nc_i$).
- $\overline{kh}_i$—Average number of members of class $C_{i,1}$ having the same value for the nested attribute $A_n$ ($1 \le i \le n$).
- $\overline{k'}_{i,j}$—Average number of instances of class $C_{i,j}$ having the same value for the nested attribute $A_n$ and having a fixed object $O$ as value of the attribute $A_t$ ($1 \le i \le n$, $1 \le j \le nc_i$, and $i \le t \le n$.)
- $\overline{fan}_{i,j}$—Average number of values held in the nested attribute $A_n$ of an instance of the class $C_{i,j}$ ($1 \le i \le n$, $1 \le j \le nc_i$).
- $\overline{fan}^{t}_{i,j}$—Average number of objects held as value of the nested attribute $A_t$ of an instance of the class $C_{i,j}$ ($1 \le i \le n$, $1 \le j \le nc_i$, $i \le t \le n$).
- $RefVal_{A_n}(i, j, t)$—Average number of instances of class $C_{i,j}$ containing at least a value among a set of $S$ given values for the nested attribute $A_n$, where $S$ is a subset of the definition domain of the attribute $A_n$ of cardinality $t$ ($1 \le i \le n$, $1 \le j \le nc_i$, $t \ge 0$).
- $Ref_{Out\_k}(i, j, y, m)$—Average number of instances of class $C_{i,j}$ containing, as nested value, a value among a set of $S$ values and not containing, as nested value, a given instance $O$ member of the class $C_{k,1}$, where $S$ is a subset of $m$ elements of the definition domain of the attribute $A_y$ ($1 \le i \le n$, $1 \le j \le nc_i$, $i \le y \le n$, $i \le k \le y$, $m \ge 0$).
- $Defh_{A_n}(i)$—Average number of members of class $C_{i,1}$ having at least a value, different from NULL, for the nested attribute $A_n$($1 \le i \le n$).

Note that in [1] the problem of finding the average number of instances of a class $C_{i,j}$ having the same value for the nested attribute $A_n$ had been solved under a large number of restrictions, such as only single-valued attributes and without considering the case of NULL values. Those restrictions have been removed in the model used here.

### B. Retrieval Cost Model

#### *B.1. NIX Organization*

The average size of a primary (leaf-node) record is given by the following expression

$$XNI = OIDL * \left[\sum_{i=1}^{n}\sum_{j=1}^{nc_i}\overline{k}_{i,j}\right] + kl + (OIDL + of + pp) * \left(\sum_{i=1}^{n} nc_i\right) \tag{1}$$

Therefore, the average number of pages required to store a leaf-node record is given by $np = \lceil XNI / P \rceil$.

Let $c$ be the number of classes target of the query; $c$ represents the number of classes whose instances must be retrieved from the index lookup. The average number of pages of a leaf-node index record storing OIDs of the instances of the same class is formulated as follows

$$NPC = \left\lceil np \Big/ \sum_{i=1}^{n} nc_i \right\rceil . \tag{2}$$

Therefore, the average number of pages accessed for an index lookup for a nested predicate being evaluated for a number $c$ of classes is:

$$C(retrieve) = \begin{cases} h_P + 1 & \text{if } XNI \le P \\ h_P + c * NPC + (np - c * NPC)/np & \text{if } XNI > P \end{cases}$$

where $h_{P+1}$ is the height of the index, $c*NPC$ represents the number of pages that are surely accessed, and $(np - c*NPC)/np$ is the probability of accessing another page in the case in which the class-directory is not stored in one of the $c*NPC$ pages. If the target of the query is a single class, we have that $c = 1$. When the target of the query is the set of all classes in the inheritance hierarchy rooted at class $C_{i,1}$, we have that $c = nc_i$.

#### *B.2. MX and IMX Organizations*

Initially, we determine the value of the parameter $VisBtree(k, n, h)$ representing the cost of the batched scanning of a B$^+$-tree of height $h$ with a total number of $n$ keys to retrieve a number $k$ of keys. Suppose that the leaf-node records may be stored in only one page. Then we have that

$$\begin{aligned} VisBtree(k, n, h) &= \sum_{l=1}^{h}\sum_{i=1}^{n_l}\left(1 - \frac{C(n - n/n_l, k)}{C(n, k)}\right) \\ &= \sum_{l=1}^{h} n_l * \left[1 - \prod_{i=1}^{k}\frac{n - (n/n_l) - i + 1}{n - i + 1}\right] \end{aligned} \tag{3}$$

Suppose that the class target of the query is $C_{i,j}$. Under the MX organization the number of pages $A_{n,t}$ with $1 \le t \le nc_n$ to be accessed from the indices allocated on the classes with position $n$ is

$$A_{n,t} = \begin{cases} h_{n,t} + 1 & \text{if } XM_{n,t} \le P \\ h_{n,t} + \lceil XM_{n,t}/P \rceil & \text{if } XM_{n,t} > P \end{cases}$$

We now determine the number of pages of the indices allocated on the classes of position $k$ with $i \le k < n$. The number of OIDs, $NOID_k$, found in an index of position $k + 1$ with $i \le k < n$ is determined as follows:

$$\mathrm{NOID}_k = \begin{cases} kh_{k+1} & \text{if } k = n - 1 \\ \overline{kh}_{k+1} & \text{if } 1 \le k < n - 1 \end{cases}$$

Finally the cost of scanning an index allocated on the class $C_{k,t}$ for $i \le k < n$ and $1 \le t \le nc_k$ is

$$A_{k,t} = VisBtree(NOID_k, D_{k,t}, h_{k,t} + 1)$$

The cost of evaluating a query with target class $C_{i,j}$ under the MX organization is

$$C(retrieve)_{i,j} = A_{i,j} + \sum_{l=i+1}^{n} \sum_{s=1}^{nc_l} A_{l,s}$$

If the query has an inheritance hierarchy as target, the cost is

$$C(retrieve)_i = \sum_{l=i}^{n} \sum_{s=1}^{nc_l} A_{l,s}$$

The cost of retrieval under the IMX organization is determined through the same previous formulas where $D_k$ substitutes $D_{k,t}$ and the total cost is

$$C(retrieve)_i = \sum_{l=i}^{n} A_l$$

## C. Delete Cost Model

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and an object $O_{i,j}$, $1 \le i \le n$, $1 \le j \le nc_i$, an index on the path may have to be updated when $O_{i,j}$ is deleted from the database. In this subsection we will derive the cost formulas for delete operations.

### *C.1. NIX Organization*

To evaluate the delete cost model of the NIX organization, we consider three different cases:

1) the object $O_{i,j}$ has a null value for the attribute $A_i$;
2) all the objects referenced by $O_{i,j}$ have a null value for the nested attribute $A_n$;
3) the object $O_{i,j}$ has values different than null for the nested attribute $A_n$.

In the first case, since no primary record is accessed and only one auxiliary record is updated, the delete cost is given by the following expression:

$$C_{indef} = h_A + 2$$

In the second case, the number of auxiliary records that are updated is obtained as follows:

$$DNR_A(i, j) = 1 + NRS_A(i, j)$$

where $NRS_A(i, j)$ represents the average number of the auxiliary records corresponding to the classes of position $i + 1$ in the path that are updated. $NRS_A(i, j)$ is formulated as follows:

$$NRS_A(i, j) = \sum_{t=1}^{nc_{i+1}} \left( 1 - \frac{C\left(D_{i,j} - D^t_{i,j},\ fan_{i,j}\right)}{C\left(D_{i,j},\ fan_{i,j}\right)} \right)$$
$$= \sum_{t=1}^{nc_{i+1}} \left[ 1 - \prod_{r=1}^{fan_{i,j}} \frac{D_{i,j} - D^t_{i,j} - r + 1}{D_{i,j} - r + 1} \right]$$

The number of pages storing a number $DNR_A(i, j)$ of records is:

$$DNP_A(i, j) = \begin{cases} H\left(DNR_A(i, j),\ LP_A,\ \sum_{k=2}^{n} Nh_k\right) & \text{if } \overline{XAUX} \le P \\ DNR_A(i, j) & \text{otherwise} \end{cases}$$

where $H$ is the function of Yao [22]; $\overline{XAUX}$ represents the size of an auxiliary record and it is expressed as:

$$\overline{XAUX} = \left\lceil \frac{\sum_{i=2}^{n} \sum_{j=1}^{nc_i} XAUX_{i,j}}{\sum_{i=2}^{n} nc_i} \right\rceil.$$

$LP_A$, that is the number of leaf pages of the auxiliary B$^+$-tree, is formulated as follows:

$$LP_A = \left\lceil \frac{\sum_{i=2}^{n} \sum_{j=1}^{nc_i} XAUX_{i,j}}{P} \right\rceil.$$

The parameter $XAUX_{i,j}$, appearing in the previous formulas, represents the size of an auxiliary record associated with the class $C_{i,j}$; it is evaluated as

$$XAUX_{i,j} = N_{i,j} * \left[ OIDL * (1 + kh_{i-1}) + pp * \overline{fan}_{i,j} \right]$$

where the second term of the product is the average dimension of a tuple. Since the cost of retrieving the $fan_{i,j} + 1$ keys in the auxiliary B$^+$-tree is given by the parameter *VisBtree*, we obtain that the total cost in the second case is

$$C_{f\_indef}(i, j) = 2 * DNP_A(i, j) + VisBtree\left( fan_{i,j} + 1,\ \sum_{i=2}^{n} \sum_{j=1}^{nc_i} N_{i,j},\ h_A \right)$$

We now analyze the third case that may arise. The derivation of the number of updated leaf pages of the primary B$^+$-tree is similar to the derivation obtained for the retrieval cost:

$$DNP_P(i, j) = \begin{cases} H\left(\overline{fan}_{i,j},\ LP_P,\ D_n\right) & \text{if } XNI \le P \\ \overline{fan}_{i,j} * \left[ \left(1 + \sum_{s=2}^{i-1} nc_s\right) * NPC + \right. \\ \left. \left( np - \left(1 + \sum_{s=2}^{i-1} nc_s\right) * NPC \right) \Big/ np \right] & \text{if } XNI > P \end{cases}$$

where

$$LP_P = \left\lceil (D_n * XNI) / P \right\rceil$$

and *NPC* has been defined in (2).

The cost of retrieving the $v * fan_{i,j} + 1$ keys in the auxiliary B$^+$-tree is evaluated by the parameter *VisBtree* where $v$ is a flag having value 1 if $i < n$, and 0 otherwise. The number of auxiliary records to be accessed is given by

$$DNR_A(i, j) = 1 + NRS_A(i, j)$$
$$+ \sum_{s=2}^{i-1} \sum_{t=1}^{nc_s} \left( 1 - val(s, t) * \frac{C\left(Ref_{out\_i}\left(s, t, n, \overline{fan}_{i,j}\right),\ \overline{k}^i_{s,t}\right)}{C\left(RefVal_{A_n}\left(s, t, \overline{fan}_{i,j}\right),\ \overline{k}^i_{s,t}\right)} \right)$$
$$= 1 + NRS_A(i, j)$$
$$+ \sum_{s=2}^{i-1} \sum_{t=1}^{nc_s} \left[ 1 - val(s, t) * \prod_{r=1}^{\overline{k}^i_{s,t}} \frac{Ref_{out\_i}\left(s, t, n, \overline{fan}_{i,j}\right) - r + 1}{RefVal_{A_n}\left(s, t, \overline{fan}_{i,j}\right) - r + 1} \right]$$

where the parameter $val(s, t)$ is a flag having value 1 if $\overline{fan}^i_{s,t} > 1$, and 0 otherwise. The third term of the sum represents the average number of the auxiliary records that are updated and corresponds to the classes of position lower than $i$ in the path. The parameter *val*(*s*, *t*) is a flag having value 1 if $\overline{fan}^{i}_{s,t} > 1$ and 0 otherwise.

If the auxiliary records storing the tuples of the ancestors of $O_{i,j}$ are retrieved and updated through the class-directory, all pages storing such records must be accessed. The number of pages is

$$DNP_A(i, j) = \begin{cases} H\left(DNR_A(i, j), LP_A, \sum_{k=2}^{n} Nh_k\right) & \text{if } \overline{XAUX} \le P \\ DNR_A(i, j) * \lceil \overline{XAUX}/P \rceil & \text{otherwise} \end{cases}$$

If, instead, the tuples of the ancestors of $O_{i,j}$ are retrieved through a scanning of the auxiliary B$^+$-tree, it is not necessary to access all pages storing every auxiliary record. Therefore, the number of pages of the auxiliary B$^+$-tree to be accessed is

$$DNP1_A(i, j) = \sum_{t=2}^{i-1} VisBtree\left(numtuple(t), \sum_{i=2}^{n}\sum_{j=1}^{nc_i} N_{i,j}, h_A + 1\right)$$

where *numtuple*(*t*), representing the number of tuples associated with the ancestors of $O_{i,j}$ of the classes of position *t* in the path, is formulated as follows:

$$numtuple(t) = \sum_{s=1}^{nc_i} \bar{k}^{i}_{t,s} * \left(1 - val(t, s) * \frac{C\left(Ref_{Out\_i}\left(t, s, n, \overline{fan}_{i,j}\right), \bar{k}^{i}_{t,s}\right)}{C\left(RefVal_{A_n}\left(t, s, \overline{fan}_{i,j}\right), \bar{k}^{i}_{t,s}\right)}\right)$$

where *val*(*t*,*s*) has value 1 if $\overline{fan}^{i}_{s,t} > 1$, and 0 otherwise. Therefore the cost is given by:

$$DNP_{opt}(i, j) = min\left(DNP_A(i, j), DNP1_A(i, j)\right)$$

The total cost for the third case is

$$C_{def}(i, j) = VisBtree\left(v * fan_{i,j} + 1, \sum_{i=2}^{n}\sum_{j=1}^{nc_i} N_{i,j}, h_A + 1\right)$$
$$+ 2 * DNP_P(i, j) + DNP_{opt}(i, j) + H\left(DNR_A(i, j), LP_A, \sum_{k=2}^{n} Nh_k\right)$$

By merging the three cases we obtain

$$C(delete)_{i,j} = C_{indef}(i, j) * P_1 + C_{f\_indef}(i, j) * P_2 + C_{def}(i, j) * (1 - P_1 - P_2)$$

where $P_1$ and $P_2$ represent the probability respectively that the first and the second case occur. They are evaluated as follows:

$$P_1 = \left(N_{i,j} - d_{i,j}\right) / N_{i,j} \qquad (4)$$

$$P_2 = \begin{cases} \dfrac{C\left(D_{i,j} - D_{i,j}/Nh_{i+1} * Defh_{A_n}(i+1), fan_{i,j}\right)}{C\left(D_{i,j}, fan_{i,j}\right)} & \text{if } i < n \\ 0 & \text{if } i = n \end{cases} \qquad (5)$$

*C.2. MX and IMX Organizations*

To remove an object $O_{i,j}$, instance of a class $C_{i,j}$ $1 \le i \le n$, $1 \le j \le nc_i$, from an index, a number $fan_{i,j}$ of OIDs identifying the objects pointed by $O_{i,j}$ are retrieved from the index allocated on the class $C_{i,j}$. The cost of this visit is

$$A_{i,j} = VisBtree\left(fan_{i,j}, D_{i,j}, h_{i,j} + 1\right).$$

Then, under the MX organization, the OID of $O_{i,j}$ is retrieved from the indices allocated on the classes of position $i - 1$ in the path. The cost for a scanning of an index allocated on class $C_{i-1,t}$ for $1 \le t \le nc_{i-1}$ is

$$A_{i-1,t} = \begin{cases} h_{i-1,t} + 1 & \text{if } XM_{i-1,t} \le P \\ h_{i-1,t} + \lceil XM_{i-1,t}/P \rceil & \text{otherwise} \end{cases}$$

Since the probability that $O_{i,j}$ has value NULL in the attribute $A_i$ is $d_{i,j}/N_{i,j}$, we obtain that

$$C(delete)_{i,j} = \left(A_{i,j} + 1\right) * d_{i,j}/N_{i,j} + \sum_{k=1}^{nc_{i-1}} \left(A_{i-1,k} + 1\right).$$

The formulas to determine the cost of a delete operation for the IMX organization are similar to those for the MX organization. The total cost for the IMX organization is formulated as follows

$$C(delete)_i = \left(A_i + 1\right) * d_i/N_i + \left(A_{i-1} + 1\right)$$

## D. Insert Cost Model

In this subsection, we formulate the cost model for the three organizations in the case in which the index allocated on the path $\mathcal{P} = C_1.A_1.A_2...A_n$ must be updated to insert a new object $O_{i,j}$, instance of a class $C_{i,j}$ $1 \le i \le n$, $1 \le j \le nc_i$ in the database.

*D.1. NIX Organization*

To derive the cost formulas of an insert operation, we consider the same three cases described in the previous subsection. The first two cases are dealt in the same way as the delete operation and therefore we analyze only the third case. The number of leaf pages of the primary B$^+$-tree to be accessed is:

$$INP_P(i, j) = \begin{cases} H\left(\overline{fan}_{i,j}, LP_P, D_n\right) & \text{if } XNI \le P \\ \overline{fan}_{i,j} * \lceil NPC + (np - NPC)/np \rceil & \text{otherwise} \end{cases}$$

where the parameter *NPC* has been defined in (2). The retrieval cost for the auxiliary B$^+$-tree is given by the parameter $VisBtree(fan_{i,j}, \sum_{i=2}^{n}\sum_{j=1}^{nc_i} N_{i,j}, h_A)$ defined by expression (3). The total number of auxiliary records to be accessed is

$$INR_A(i, j) = NRS_A(i, j) + 1$$

since only the auxiliary records containing at least a child of $O_{i,j}$ or object $O_{i,j}$ itself are updated. The number of pages, $INP_A$, storing such records is given by the following expression:

$$INP_A(i, j) = \begin{cases} H\left(INR_A(i, j), LP_A, \sum_{k=2}^{n} nc_k\right) & \text{if } \overline{XAUX} \le P \\ INR_A(i, j) & \text{otherwise} \end{cases}$$

Therefore in the third case, the insert cost is given by

$$C_{def}(i, j) = VisBtree\left(fan_{i,j}, \sum_{i=2}^{n}\sum_{j=1}^{nc_i} N_{i,j}, h_A + 1\right)$$
$$+ 2 * INP_P(i, j) + 2 * INP_A(i, j).$$

Merging the three cases, we obtain

$$C(insert)_{i,j} = C_{indef} * P_1 + C_{f\_indef}(i, j) * P_2 + C_{def}(i, j) * (1 - P_1 - P_2)$$

where $P_1$ and $P_2$ are defined, respectively, by expressions (4) and (5).

### *D.2. MX and IMX Organizations*

In these organizations, the costs are due to the scanning of a single index. Therefore for the MIX organization we obtain

$$C(insert)_{i,j} = \left[VisBtree\left(fan_{i,j},\ D_{i,j},\ h_{i,j} + 1\right) + 1\right] * d_{i,j} / N_{i,j}$$

while for the IMX organization the cost is

$$C(insert)_i = \left[VisBtree\left(fan_i,\ D_i,\ h_i + 1\right) + 1\right] * d_i / N_i$$

Note that if the object that must be inserted in the database contains the value NULL in the property $A_i$ the index is not updated.

## IV. Comparison

To evaluate the performances of the three index organizations, we have executed a large number of simulation experiments on the basis of the mathematical cost model. Initially through a series of preliminary experiments, we have determined the parameters of the model that have a considerable impact on the costs. The parameters that characterize the topology of the database are those that can affect in varying degrees the size and performance of an index. In particular, the parameter $fan_{i,j}$, representing the average number of objects pointed by an instance of a class $C_{i,j}$, has a relevant impact on costs. Many other parameters are dependent on that parameter: for example the parameter $k_{i,j}$ representing the average number of instances of the class $C_{i,j}$ having the same value for attribute $A_i$.

The values of some of the cost model parameters are fixed, as illustrated in Fig. 6, according to common implementations of the B$^+$-tree index. Other parameters, such as *n*, *D*, *N*, *d*, and *fan* for all or for some classes in the path, are given as input while the remaining parameters are derived through mathematical formulas.

| | | | | | |
|---|---|---|---|---|---|
| OIDL=8 | kl=2 | of=2 | pp=4 | P=4096 | f=218 |

Fig. 6. Table 1.

In this section we report only some of the experiments done and, through the results obtained for different values of the most influential parameters, we compare the performances of the three organizations.

### A. Storage Cost

In all experiments performed, we have obtained that the traditional indices have the lowest storage cost. In particular the IMX organization has the best costs when in the path there are inheritance hierarchies while the NIX organization has the highest cost. We note, however, that storage costs may not be crucial, since large capacity storage devices are today widely available. Therefore, it may be preferable to privilege organizations providing good performance, even if they have large storage requirements.

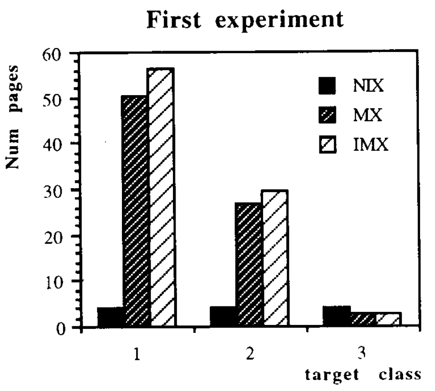
### B. Retrieval Cost

From the simulation experiments we have obtained as result that the NIX organization offers better performance than traditional indices in nearly all cases. To illustrate this conclusion, we consider some of the most meaningful simulation experiments. In the graphs we report here, the parameters *d*, *D*, and *N* are given value 250,000 for all classes. Moreover, the reported experiments deal with paths without inheritance hierarchies. We show three groups of experiments in which the target class of the query changes. Note in the first graph of Fig. 7 that the MX and IMX organizations have low costs if the target class has position 3 in the path while they have high costs in the other cases. If the retrieval operation is on the last class in the path the cost of traditional indices is slightly lower than the NIX organization. We now explain the reason of this behavior. The value of the parameter $fan_{3,1}$, on which the parameter $k_{3,1}$ depends, determines the number of search keys in the index allocated on the class $C_{2,1}$. If the retrieval operation is performed on the first class of the path the costs for the IMX and MX organizations are highest than the previous case since in addition to the lookup of the index allocated on the class $C_{2,1}$ a search of the index on the class $C_{1,1}$ must be executed. In the second visit the number of search keys is 20 since $fan_{2,1}$ = 1 and $fan_{3,1}$ = 20. The retrieval costs of for the NIX organization are constant and do not depend on the position of the target class.

Now if we consider $fan_{1,1} = fan_{3,1}$ = 1, and $fan_{2,1}$ = 20 and we maintain the other parameter values equal to those of the previous experiment, the MX and IMX organizations have high costs only when the target class of the query has position 1 in the path. Indeed the high value of the parameter $fan_{2,1}$ only affects the cost of the lookup of the index allocated on the class $C_{1,1}$ in which the number of search keys is given by $fan_{2,1}$. We now consider the case in which $fan_{2,1} = fan_{3,1}$ = 1, and $fan_{1,1}$ = 20. In this case the high value of the parameter $fan_{1,1}$ does not influence the cost of the organizations.

The second graph in Fig. 7 represents the case in which the target class of the query is $C_{1,1}$. The value *pos*(*class*) = 1 corresponds to the case in which $fan_{1,1}$ = 20 and $fan_{2,1} = fan_{3,1}$ = 1, *pos*(*class*) = 2 corresponds to the case in which $fan_{2,1}$ = 20 and $fan_{1,1} = fan_{3,1}$ = 1 while *pos*(*class*)=3 corresponds to the case in which $fan_{3,1}$ = 20 and $fan_{1,1} = fan_{2,1}$ = 1. In conclusion, the MX and IMX organizations have the highest costs when the classes positioned at the end of the path are characterized by a high fan, while if some high fans are positioned at the beginning of the path the retrieval costs are lower.

If we consider range value queries, i.e., queries in which the predicate varies in a range of values, the trend of the costs is the same as for the previous experiments but the values of the costs in the worst cases for the MX and IMX organizations are highest. An example of costs is presented in Fig. 8 for a range



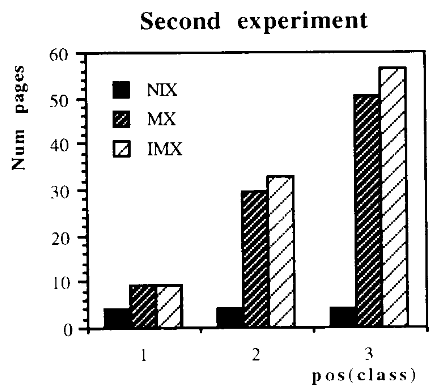


Fig. 7. First exp.: Retrieval costs obtained by varying the position of the target class with $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and $fan_{1,1} = fan_{2,1} = 1$, $fan_{3,1} = 20$; Second exp.: Retrieval costs obtained by varying the position of the class with the highest *fan* (*fan* = 20) while the value of *fan* for the other classes is 1, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ $C_{1,1}$ is the target class.

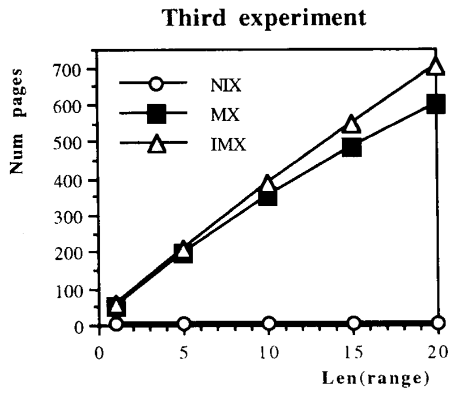


Fig. 8. Range value retrieval costs obtained by varying the length of the range with $C_{1,1}$ as target class and $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and $fan_{1,1} = fan_{2,1} = 1, fan_{3,1} = 20$.

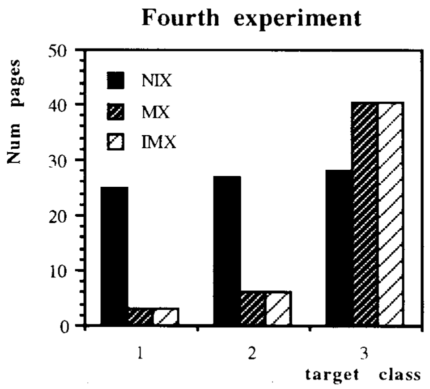


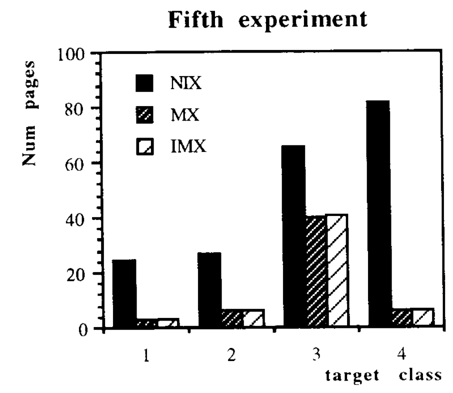
Fig. 9. Delete costs obtained by varying the position of the target class with $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and $fan_{1,1} = fan_{2,1} = 1, fan_{3,1} = 20$.

varying between 1 and 20.

### *B.1. Conclusions on Retrieval Cost*

From the experiments that have been performed, many of which are not reported here, we can conclude that the NIX organization offers in general good performance unlike the traditional organizations. The MX and IMX organizations are advantageous only when most retrieval operations have as target the last classes of the path. However, the purpose of allocating the organization presented in this paper is to reduce the evaluation costs of nested predicates. Therefore, the interesting cases are those when the target of the query is one of the classes at the beginning of the path. If the database has small dimension or the fans of the classes have low values (1 or 2), the traditional indices have costs that does not differ much from the NIX organization. Also when the fans of all the classes but the first class of the path are low, the costs of the MX and IMX organizations are acceptable. We note how the position of the target class of the query is important. In fact, if the target class is the last of the path, the costs for the traditional organizations are low while, if the position of the such class is 1, the costs of the MX and IMX grow exponentially for varying values of the *fan* parameter and for different positions of the class with the highest fan. The costs of the NIX organization do not depend on the presence of inheritance hierarchies, while this factor influences the performance of the MX organization. Its costs, indeed, depend on the number of classes in the path, unlike those of the IMX organization that only depend on the length of the path. Another factor influencing the costs of the MX and IMX organizations is the length of the path mainly when the target classes are positioned in thebeginning of the path.

These evaluations are valid also for range value queries. In this case, however, in addition to the factors previously discussed, the number of key-values included in the range must be considered. The larger that number is, the higher the costs for the traditional organizations are. Also the cost of the NIX organization grows for increasing dimensions of the range but to a lower degree than the MX and IMX organizations.



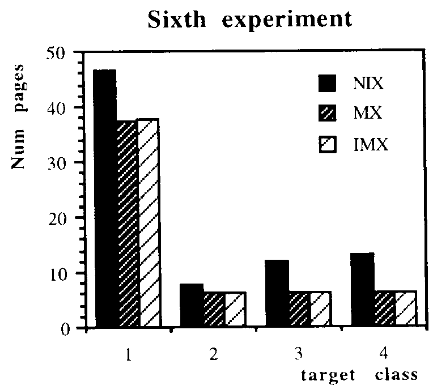


Fig. 10. Fifth exp.: Delete costs obtained by varying the position of the target class with $n = 4$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \le i \le 4$ and $fan_{1,1} = fan_{2,1} = fan_{4,1} = 1$, $fan_{3,1} = 20$. Sixth exp.: Delete costs obtained by varying the position of the target class with $n = 4$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \le i \le 4$ and $fan_{2,1} = fan_{3,1} = fan_{4,1} = 1$, $fan_{1,1} = 20$.

## C. Delete Costs

A delete operation, unlike a retrieval operation, is generally more expensive in a NIX organization than in the traditional organizations. Also in the experiments for the evaluation of the delete costs the relevance of the *fan* parameter has emerged.

As an example, we show the results of an experiment in which the path has length 3 and contains no inheritance hierarchy. Moreover $N = D = d = 250{,}000$ for each class in the path. Suppose that the attributes $A_1$ and $A_2$ are single-valued and that $fan_{3,1} = 20$. If an instance of class $C_{3,1}$ is deleted, the NIX organization offers the best performance. Indeed the costs of the MX and IMX organizations are high since $fan_{3,1}$ represents the number of keys to be searched in the index allocated on class $C_{3,1}$. If the delete operation is performed on class $C_{2,1}$ or $C_{1,1}$, the costs of the NIX organization are slightly lower than those of the previous case, while the costs of the MX and IMX organizations are the lowest. Therefore the high value of $fan_{3,1}$ impacts the costs of the NIX organization, while for the MX and IMX organizations it influences the costs only when the delete is executed on the classes that are characterized by the high fan. The results of the fourth experiment are illustrated in Fig. 9.

Now we consider a path of length 4 and we fix $N = D = d = 250{,}000$ for each class of the path. The fans of all classes are 1 but *fan*3,1 that has value 20. In this case the highest costs for the three organizations are obtained in the case in which the delete is from class $C_{3,1}$, confirming the results of the previous experiments (Fig. 10, fifth experiment). For the traditional organizations the costs are highest when the delete is executed on the class with a high fan. Now we consider a variation of the previous experiment, in which we fix $fan_{2,1} = fan_{3,1} = fan_{4,1} = 1$ and $fan_{1,1} = 20$. We obtain good results also for the NIX organization which has low costs in all cases but when an instance of the class $C_{1,1}$ is deleted. The results of this last experiment are illustrated in Fig. 10 (sixth experiment).

Now we consider again a path of length 3 and suppose that no indexed attribute is single-valued. Moreover, $fan_{1,1} = fan_{2,1} = 2$, while the value of $fan_{3,1}$ varies from 1 to 20 and $N$, $D$, and $d$ have the same values as in the previous experiments. We obtain a trend of the costs like to that one discussed previously. The results are presented in Fig. 11.

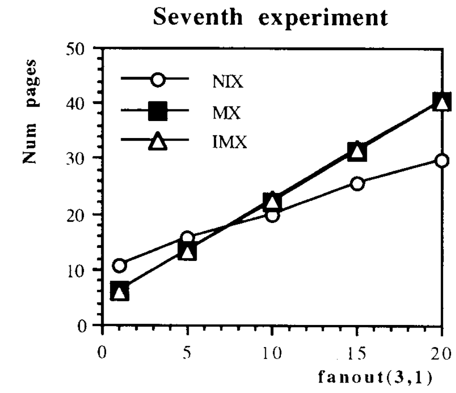


Fig. 11. Delete costs obtained by varying the value of $fan_{3,1}$ from 1 to 20 with $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$, for $1 \le i \le 3$ and $fan_{1,1} = fan_{2,1} = 2$, and with $C_{3,1}$ as target class.

### C.1. Conclusions on the Delete Costs

The NIX organization has costs that depend on the value of the parameter *fan* more heavily than the traditional organizations. While the deletion of an instance from a class is expensive for the MX and IMX organizations only if such class is characterized by a high fan, this operation is expensive for the NIX organization independently on the fan of the class from which the instance is removed. There are some cases, however, in which the costs for the NIX organization are only slightly higher or even lower than those of traditional organizations. For example, if only the first class of the path has a high fan and all other classes have single-valued indexed attributes the costs for the NIX organization are acceptable, in particular if the delete does not have as target the first class of the path. An optimal case for the NIX is when all indexed attributes are single-valued. The trend of the costs of the NIX and of the MX and IMX just described is independent from the presence in the path of inheritance hierarchies. We have experienced that the cost of the NIX organization is in inverse ratio with the parameter $D$. Indeed, the NIX organization depends more on the parameter $k$ which is function of parameters $D$ and *fan*, while the MX and IMX organizations only depend on the parameter fan, and consequently on parameter $k$ but not on $D$. Indeed the cost of the retrieving the parent's tuples from the auxiliary records through a lookup of the auxiliary $B^+$-tree is influenced by the parameter $k$ representing the number of search keys. In conclusion the NIX organization offers good performance when all classes in the path have single-valued indexed attributes or when only the first class of the path has a multi-valued attribute in the path and all other classes have single-valued indexed attributes in the path, or when the delete operation has as target the class of the path that has a high fan.

## D. Insert Costs

The results obtained for the three organizations with regard to the insert costs are similar to those described in the previous subsection. As an example we show the results of an experiment in which $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and $fan_{1,1} = fan_{2,1} = 1$ while $fan_{3,1}$ is varied from 1 to 20. If we insert an object in the class $C_{3,1}$ we obtain costs for the three organizations that are lower than the delete costs, but that have a trend similar to the delete costs. The graphs reporting the results of this experiment is presented in Fig. 12.

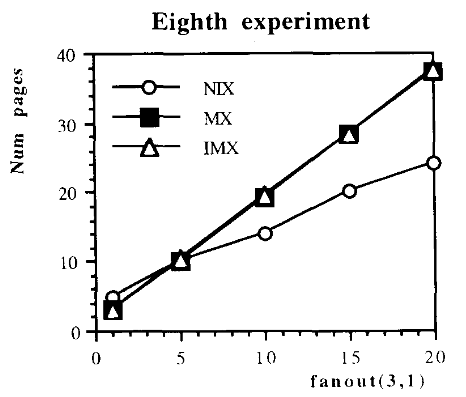


Fig. 12. Insert costs obtained by varying the value of $fan_{3,1}$ from 1 to 20 with $n = 3$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and $fan_{1,1} = fan_{2,1} = 1$, and with $C_{3,1}$ as target class.

The main difference between delete costs and insert costs is for the NIX organization. In this case the *fans* of the class into which an object is inserted and of the classes immediately following such class in the path heavily impact the costs. Therefore if an insert operation is performed on a class $C$ with low fan and the classes with position greater than $C$ have low fans, the performance of the NIX organization is very good. As an example, consider the following experiment. We have fixed $n = 4$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 3$ and we have varied the position of the class with high fan (*fan* = 20), while the fans of all other classes are 1. Therefore we analyze the four cases in which an instance is inserted in the first or in the second or in the third or in the fourth class of the path. This experiment further illustrates our conclusions. The trend of the cost depends on the fact that when an object $O$ is inserted in a class $C$ the children of $O$ must be retrieved from the auxiliary $B^+$-tree and the number of search keys is given by the fans of the classes immediately following $C$ in the path. However, there is no need of updating the auxiliary records of the classes preceding $C$ in the path and for this reason the fans of classes with position lower than $C$ have no weight. The results of this experiments are illustrated in Fig. 13.

### *D.1. Conclusions on Insert Costs*

The insert costs for the three organizations are lower than those of delete operations, since the number of updates to the index structures is smaller. The trend of the costs of the indices is as discussed in the previous subsection. There is only one difference: while a delete operation has costs for the NIX organization that are independent from the fan of the class from which an object is deleted, for an insert operation this factor is important. The NIX organization offers good performance if the class into which an object is inserted and the immediately following classes in the path have low fan. In particular the performance of the NIX organization for the insert operation is as good as the other organizations when the classes at the beginning of the path have low fans.

## V. Conclusions and Future Work

In this paper we have presented an index organization for object-oriented databases, able to support queries involving several classes along aggregation and inheritance hierarchies. This organization has been compared with other two organizations: the first based on allocating an index on each class found in the scope of a path; the second consists of allocating a class-hierarchy index [17] on each inheritance hierarchy found in the indexed path. The performance of those organizations has been evaluated for the retrieval, delete, and insert operations. A large number of parameters have been taking into account. In particular, several parameters have been introduced to provide an accurate model of topologies among object references. The model releases a number of limiting assumptions made by previous related works. The major conclusions can be summarized as follows:

- The NIX organization offers the best retrieval performance in most cases. It is outperformed by the other organizations only when queries are mainly on the last class of the indexed path. Therefore, the NIX organization should be mainly used when the number of nested predicates in queries is high.
- The costs of the various organizations for modification operations (delete, insert) are highly dependent on the object references topologies and on the class target of the

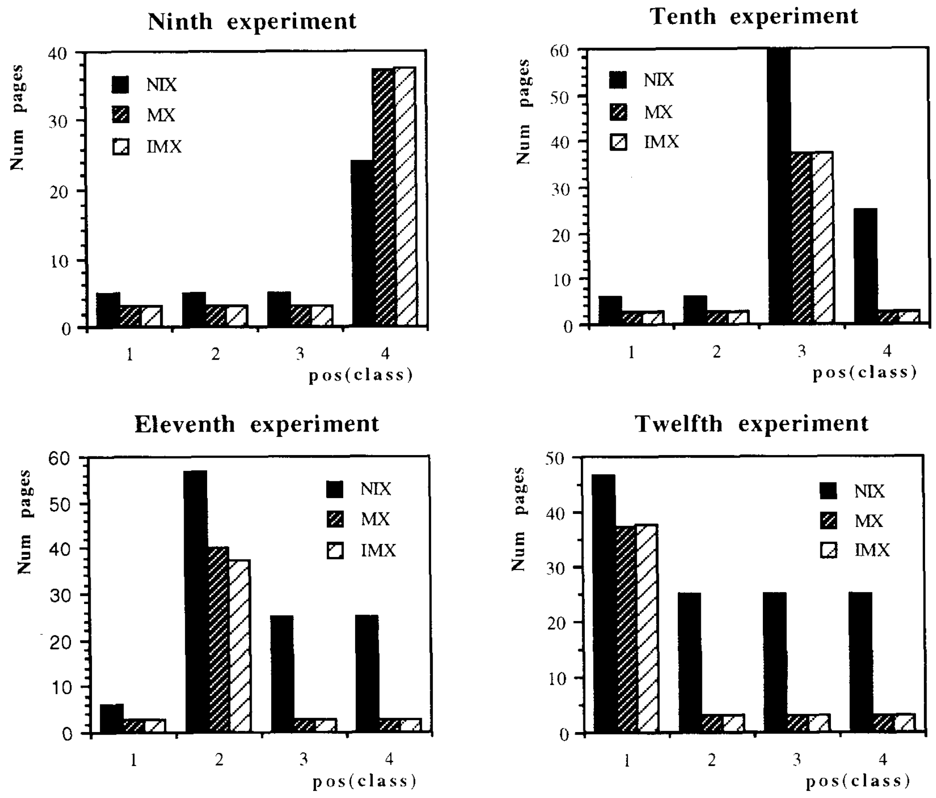


Fig. 13. Costs of inserting an object, respectively, in class $C_{4,1}$, $C_{3,1}$, $C_{2,1}$, and $C_{1,1}$ (from left to right, and from top to bottom) obtained by varying the position of the class with high fan ($fan = 20$), while the fans of the other classes are 1 with $n = 4$, $N_{i,1} = D_{i,1} = d_{i,1} = 250{,}000$ for $1 \leq i \leq 4$.

operations. In some situations the NIX has better performance than the other organizations. These situations are characterized by a high frequency of modification operations on classes with the highest fan. Another optimal situation for the NIX organization is when all attributes along the indexed path are single-valued; or when only the first class in the path has multi-valued attributes, while all the other classes have single-valued attributes. In other situations, the NIX organization has a cost which is not very high with respect to the costs of the other organizations.

The work reported in this paper is being extended along several directions. First, an extensive comparison is being carried out between the nested-inherited index, proposed in the present paper, and the path index. The path index [1], [9] is an organization defined for efficiently supporting fast traversal of aggregation hierarchies. The main idea of the path index is to pre-compute and store as arrays all instantiations along an indexed path. In this respect the path index and the nested-inherited index are similar, since both are based on pre-computing the sequences of implicit joins found along a path in an aggregation hierarchy. However, unlike the nested-inherited index, in case of updates the path index requires accesses to objects, other than the object being modified, to determine the updates to be performed on the leaf-node records of the index. An extensive comparison between the two organizations is being carried out.

Second, the usage of the proposed indexing techniques is being investigated in the framework of complex queries, containing several nested predicates. Third, the problem of index configuration is being addressed. In particular, a path can be split into several subpaths and possibly different index organizations used on each subpath. Finally, additional indexing techniques are being investigated.

## REFERENCES

[1] E. Bertino and W. Kim, "Indexing techniques for queries on nested objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 1, no. 2, pp. 196-214, 1989.

[2] E. Bertino, "An indexing technique for object-oriented databases," *Proc. Seventh IEEE Int'l Conf. Data Eng.*, Kobe, Japan, Apr. 1991.

[3] E. Bertino and L. Martino, "Object-oriented database management systems: Concepts and issues," *Computer*, vol. 24, no. 4, pp. 33-47, 1991.

[4] E. Bertino and P. Foscoli, "On modeling cost functions for complex objects," submitted for publication, Jan. 1992, available from the authors.

[5] E. Bertino and A. Quarati, "An approach to support method invocations in object-oriented queries," *Proc. Int'l Workshop Research Issues in Transactions and Query Processing (RIDE-TQP)*, Phoenix, Ariz., Feb. 1992.

[6] E. Bertino and E. Montesi, "Towards a logical-object oriented programming language for databases," *Proc. Third Int'l Conf. Extending Database Technology (EDBT)*, Vienna, Mar. 1992.

[7] E. Bertino, S. Bottarelli, M. Damiani, M. Migliorati, and P. Randi, "The ADKMS knowledge acquisition system," *Proc. Second Far-East Workshop Future Database Systems*, Apr. 1992.

[8] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella, "Object-oriented query languages: The notion and the issues," *IEEE Trans. Knowledge and Data Eng.*, vol. 4, no. 3, pp. 223-237, 1992.

[9] E. Bertino and C. Guglielmina, "Path-Index: An approach to the efficient execution of object-oriented queries," *Data and Knowledge Eng.*, North-Holland, 1993.

[10] E. Bertino, "A survey of indexing techniques for object-oriented databases," *Proc. Dagsthul Seminar Query Processing in Object-Oriented, Complex-Object and Nested Relational Databases*, C. Freytag, D. Maier, and G. Vossen, eds., Morgan-Kaufmann, 1993.

[11] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick, "CLASSIC: A structural data model for objects," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, Portland, Ore., June 1989.

[12] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari, "Integrating object-oriented data modeling with a rule-based programming paradigm," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, Atlantic City, N.J., May 1990.

[13] M. Carey and D. DeWitt, "An overview of the EXODUS project," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, Chicago, Ill., June 1988.

[14] A. Jhingran, "Precomputation in a complex object environment," *Proc. Seventh IEEE Int'l Conf. Data Eng.*, Kobe, Japan, Apr. 1991.

[15] A. Kemper and G. Moerkotte, "Access support in object bases," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, Atlantic City, N.J., May 1990.

[16] A. Kemper, C. Kilger, and G. Moerkotte, "Function materialization in object bases," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, Denver, Colo., May 1991.

[17] W. Kim, K.C. Kim, and A. Dale, "Indexing techniques for object-oriented databases," *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[18] D. Maier and J. Stein, "Indexing in an object-oriented DBMS," *Proc. IEEE Workshop Object-Oriented DBMS*, Asilomar, Calif., Sept. 1986.

[19] M. Stonebraker, L. Rowe, and M. Hirohama, "The implementation of Postgres," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, pp. 125-142, 1990.

[20] P. Valduriez, "Join indices," *ACM Trans. Database Systems*, vol. 12, no. 2, pp. 218-246, 1987.

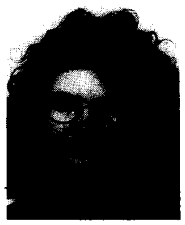
[21] C. Zaniolo, "Object identity and inheritance in deductive databases—An evolutionary approch," *Proc. First Int'l Conf. Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, Dec. 1989.

[22] S.B. Yao, "Approximating block accesses in database organizations," *ACM Comm.*, vol. 20, no. 4 , pp. 260-261, 1977.

**Elisa Bertino** is professor of computer science at the University of Milan, where she heads the Database Systems Group. She was previously a member of the faculty in the Department of Computer and Information Science of the University of Genova, Italy. Until 1990 she was a researcher for the Italian National Research Council in Pisa, Italy, where she headed the Object-Oriented Systems Group. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation in Austin, Texas, and at George Mason University.

Bertino has participated in several projects sponsored by the Italian National Research Council and the European Economic Communities. She served as general co-chair of the IEEE International Workshop on Research Issues on Data Engineering—Interoperability in Multidatabase Systems (RIDE-IMS), 1993, and vice program chair for the 1993 IEEE Data Engineering Conference. She is a co-author of the forthcoming book, *Object-Oriented Database Systems—Concepts and Architectures* (Addison-Wesley).

Bertino's research interests include object-oriented databases, distributed databases, multimedia databases, interoperability of heterogeneous systems, and integration of artificial and database techniques.



**Paola Foscoli** received the master's degree in computer science from the University of Genova, Italy, in October 1992. Since graduation she has been working as a research consultant for the Computer and Information Science Department of the University of Genova.

Foscoli's research interests include indexing techniques and query processing strategies for object-oriented databases.