Enhanced Nested-Inherited Index for OODBMS

E.Bertino †, S. Salerno ‡, B.Shidlovsky ‡

†Dipartimento di Scienze dell'Informazione - Università di Milano
Via Comelico 39, 20135 Milano, Italy, bertino@disi.unige.it
‡DIIMA, Università di Salerno, 84084 Fisciano (SA), Italy
{salerno,boris}@ponza.dia.unisa.it

ABSTRACT

The *nested-inherited index* has been recently proposed as an access structure providing an integrated support for queries in object-oriented databases along both aggregation and inheritance hierarchies. It is very efficient for retrieval operations. However, its high update costs make this structure suitable only for hierarchies with a small number of classes. In this paper we propose an *enhanced nested-inherited index*, able to support update operations more efficiently, whereas supporting nested predicates as efficiently as the nested-inherited index. The new organization supports the construction of several index allocation strategies, from which the most efficient with respect to a given workload can be selected. The new and old indices are compared using an analytical cost model. Results of the analysis show that the enhanced nested-inherited index provides superior performance than the inherited-multiindex and nested-inherited index.

1 INTRODUCTION

The rich expressive power of the object-oriented data models and query languages requires suitable query optimization techniques and access structures. In particular, the features of an object-oriented data model that mostly impact query processing and access structures include: aggregation hierarchies, inheritance hierarchies, methods. Indeed, most query languages provide syntactical notations allowing navigation along aggregation hierarchies. Such navigations can be seen as *implicit joins* along aggregation hierarchies [Ber94]. Their optimization requires access structures similar to join indices [Val87]. Moreover, inheritance hierarchies imply that a query may apply to a class or to a hierarchy of classes. Therefore, access structures like class-hierarchy indices [KKD89] may be required. We refer the reader to [Ber93] for an overview of access structures for OODBMS.

An access structure, called *nested-inherited index*, has been recently proposed to precompute implicit joins along aggregation hierarchies and inheritance hierarchies [BeFo95]. It provides an integrated support for queries along both hierarchies and guarantees low retrieval cost. However, the index has high update costs, especially if allocated on long hierarchies (e.g. longer than 3).

In this paper, we propose an *enhanced nested-inherited index*, able to more efficiently support update operations, whereas supporting nested predicates as efficiently as the nested-inherited index. The indexing technique involves a new structure of leaf-node record and, unlike the nested-inherited index, proposes a number of distinct variants of index allocation for recording the precomputed join. The variant providing the minimal total cost for a given query workload is chosen as the best one.

The remainder of this paper is organized as follows. Section 2 summarizes definitions and notations. Section 3 describes the enhanced nested-inherited index and shows that the nested-inherited index is equivalent to one of the variants the enhanced nested-inherited index proposed. Moreover, section 3 describes a further organization, the inherited-multiindex, that is also compared with the enhanced nested-inherited index. Section 4 describes operations for the new index organization and section 5 reports the results of the comparison.

2 PRELIMINARY NOTATIONS

A *path* represents a branch in an aggregation hierarchy; more formally a path $\mathcal{P}$ is defined as $C_1.A_1.A_2.\dots A_n$ ($n \geq 1$) where C_1 is class in the database schema and the root of the path; A_1 is an attribute of class C_1 and 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}, $1 < i \leq n$; n is the path length. Then, some important notions are the following:

- nc_i denotes the number of classes in the inheritance hierarchy rooted at class $C_i, 1 \leq i \leq n$.
- given a class C on the path, the *position* of C, pos(C), is a pair (i, j), such that C belongs to the inheritance hierarchy rooted at class C_i and has position j among the subclasses in the hierarchy, $1 \leq i \leq n, 1 \leq j \leq$

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
CIKM '95, Baltimore MD USA
© 1995 ACM 0-89791-812-6/95/11..$3.50

nc_i; class C with position (i, j) is denoted as $C_{i,j}$; for the root of the hierarchy $C_i = C_{i,1}$. For simplicity, we will use a pair $(n+1, 1)$ to indicate the position of the attribute A_n and term C_{n+1} to denote attribute A_n itself.

- C_i^* denotes a set containing class C_i itself and all subclasses in the inheritance hierarchy rooted at class C_i: $C_i^* = \bigcup_{1 \leq j \leq nc_i} C_{i,j}$.

- class(C_i) $= \bigcup_{1 \leq j < i} C_j$,
  class($\mathcal{P}$) $=$ class(A_n) $= \bigcup_{1 \leq j \leq n} C_j$.

- scope(C_i) $= \bigcup_{1 \leq j < i} C_j^*$,
  scope ($\mathcal{P}$) $=$ scope(A_n) $= \bigcup_{1 \leq j \leq n} C_j^*$.

We distinguish between *instances* and *members* of a class. An object O is instance of a class C, $O \in C$, if C is the most specialized class associated with the object in a given inheritance hierarchy. An object O is member of a class C, $O \in C^*$, if it is an instance of C or of some subclass of C.

The following is an example path for the schema in Figure 1: $\mathcal{P}$=Line.flights.routing.arrival_airports.city, n=4;
Flight*={Flight, ExceptionalFlight, WeeklyFlight};
class($\mathcal{P}$)={Line, Flight, Sector, Airport};
scope($\mathcal{P}$)={Line, ItalianLine, Flight, ExceptionalFlight, WeeklyFlight, Sector, Airport};
pos(Line)=(1,1), pos(ItalianLine)=(1,2),
pos(Flight)=(2,1), pos(WeeklyFlight)=(2,2),
pos (Except.Flight)=(2,3), pos(Sector)=(3,1),
pos(Airport)=(4,1), pos (Airport.city)=(5,1).

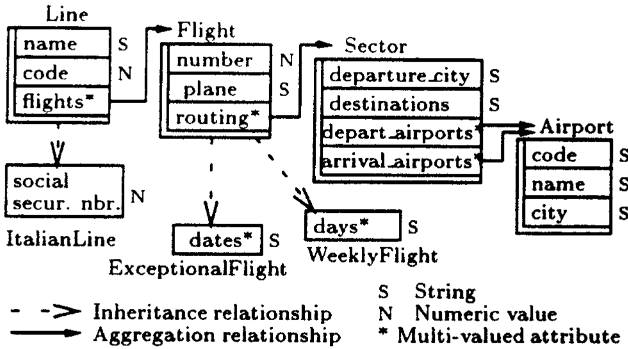

Figure 1: Example of database schema.

Figure 2 presents an example database for the schema in Figure 1.

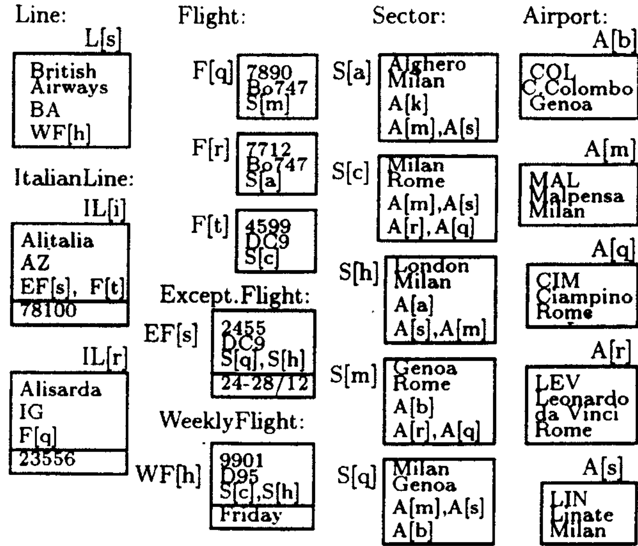

Figure 2. Database example.

3 INDEX DEFINITIONS

In this section we present the organization of the enhanced nested-inherited index, nested-inherited index and the inherited-multiindex.

3.1 ENHANCED NESTED-INHERITED INDEX (ENIX)

Let Ψ be a list of integers from the interval $[2 \ldots n]$ and list $\Delta = \Psi \bigcup \{n+1\}$. For $n = 4$, examples of Δ are $\{2, 4, 5\}$ and $\{5\}$. An integer in Δ indicates the position of class in the path where a special index supporting the precomputed joins is allocated on. Given a list Δ, two functions $prev(i)$ and $next(i)$ are defined for $i, 1 \leq i \leq n+1$ as follows: $next(i) = min\{j | j \in \Delta, j > i\}$, if $1 \leq i \leq n$ and $next(n+1) = n+1$; $prev(i) = max\{j | j \in \Delta, j < i\}$, if the set $\{j | j \in \Delta, j < i\}$ is not empty; otherwise, $prev(i) = 1$. For $\Delta = \{3, 5\}$, values of functions *next* and *prev* for $i = 1, \ldots, 5$ are given by $\{3, 3, 5, 5, 5\}$ and $\{1, 1, 1, 3, 3\}$, respectively.

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots .A_n$, an enhanced nested-inherited index is constructed with respect to a given list Δ. For any $i \in \Delta$, the index associates with O, member of C_i, all instances of a class in scope(C_i) referencing O. In particular, index associates with a value v of attribute A_n identifiers of all instances of a class in scope($\mathcal{P}$) having v as value of the nested attribute A_n.

The ENIX organization includes n indices for classes $C_1, C_2, \ldots, C_n$ and an attribute index for A_n. Every index is either a *far-look* or *one-step* type. A far-look index is allocated on class C_i if $i \in \Delta$. If $i \notin \Delta$, the index allocated on class C_i is of one-step type.

A **far-look index** allocated on class C_i contains as key values OIDs of members of C_i. The index is a B^+-tree whose leaves have a complex structure. The record in a leaf node corresponding to some member O of C_i^*, contains the following information:

- key value
- class-directory
- object-directory

- descendant-list

The *object-directory* contains subdirectories for all classes in class(C_i). In the subdirectory for $C_j \in$ class(C_i), entries for members of C_j^* referencing O are stored. An entry contains the OID of a member, its local identifier and list NCL of local identifiers of the member's children in the C_{j+1}^*. Local identifiers are assigned to all OIDs in the object-directory in order to minimize the leaf record size since the number of members of C_j referencing a given instance of class C_i is rather small as compared to the total number of members in C_j^*. While OID of an instance is unique within the entire database, its local identifier is unique only inside the subdirectory for C_j. A local identifier of some $O' \in C_{j,m}$ is simply a shortest sequence of bits allowing to distinguish it from other instances in the subdirectory. It is a concatenation of two sequences, the first one is the binary representation of m, the position of $C_{j,m}$ in the inheritance hierarchy rooted at C_j, and the second sequence is the position of O' among instances of $C_{j,m}$ inside the subdirectory. Clearly, an instance may have different local identifiers in different leaf records. Apart from the subdirectory for C_j, the local identifier of O' may appear in fathers' lists NCL in the subdirectory for C_{j-1}.

The *class-directory* contains as many entries as the number of classes in scope (C_i). For a class $C_{j,m} \in$ scope(C_i), an entry in the directory contains the class identifier, the (primary) offset in the object-directory where the subdirectory for class C_j is stored, and the (secondary) offset inside the subdirectory where entries of class $C_{j,m}$ are allocated.

The *descendant-list* contains leaf record addresses of members of class $C_{next(i)}^*$ being referenced by O.

The attribute index (A_n-index) is always far-look index where any leaf record contains an attribute value as key value and descendant-list is omitted.

The **one-step index** is allocated on a class C_i such that $i \notin \Delta$. The index is also B$^+$-tree and contains OIDs of members of class C_i as search key. A leaf record for any $O \in C_i^*$ of the index contains the following information:

- key value
- parent-list
- descendant-list

The parent-list contains the pairs { *oid*, *&oid* }, where *oid* is the OID of a parent of O and *&oid* is an address of the leaf record of the parent in C_{i-1}-index. Like the far-look index,the descendant-list contains addresses of members of $C_{next(i)}^*$ being referenced by O. Note that the C_1-index is always one-step index and does not contain the parent-list.

For the path Line.flights.routing.arrival_airports.city, Figure 3 gives the ENIX with $\Delta = \{3, 5\}$. The two indices allocated on class Sector and attribute Airport.city are far-look indices, the other ones are of one-step type. In the index allocated on the class Sector, leaf record for S[h] contains the key value, class-directory of 5 entries, subdirectories for classes Line and Flight in the object-directory and descendant-list with the address of leaf record for "Milan" as a value of attribute Airport.city. In the subdirectory for Line in the record, the unique entry for IL[i] is contained, its local identifier 10 is obtained by concatenating 1, position of ItalianLine in the hierarchy rooted at Line, and 0, position of IL[i] in the subdirectory. Since EF[s] is the only child of IL[i] referencing S[h], the NCL list in the entry for IL[i] consists solely of the local identifier of EF[s]. As for the NCL list in the entry for EF[s], it is {0} to indicate the key value S[h].

3.2 NESTED-INHERITED INDEX (NIX)

Given a path $\mathcal{P} = C_1.A_1.A_2.\ldots.A_n$, the ENIX is constructed with a given list Δ. There are 2^{n-1} different Δs for the path and, therefore, there are 2^{n-1} distinct variants of the ENIX allocation. The ENIX with the fewest number of far-look indices appears when $\Delta = \{n+1\}$. In this case, only A_n-index is the far-look index while all the class indices are of one-step type. This variant of the ENIX is very similar to the nested-inherited index (NIX) introduced in [BeFo95]. The differences between them can be summarized as follows:

1. *Attribute index*: a leaf record of the NIX also has a class-directory and an object-directory with subdirectories. However, the structure of a subdirectory is simpler. An entry in the subdirectory also contains an OID of instance referencing the attribute value of A_n which is the record key. But, instead of local identifiers, it contains uniquely the number of children referencing the same attribute value.
2. *Class indices*: instead of n indices for classes C_i, $1 \leq i \leq n$, the NIX has the common index (auxiliary index) storing the records for instances of all classes found along the path. It allows to reduce the total number of indices from $n+1$ to 2, but often increases the B-tree height.

Tests show that these differences have not a crucial impact on the index performance, and we assume the NIX to be a special case of the ENIX with $\Delta = \{n+1\}$.

3.3 INHERITED-MULTIINDEX (MIX)

This organization consists of allocating an index on each inheritance hierarchy found along the path. Given a path $\mathcal{P} = C_1.A_1.A_2.\ldots.A_n$, 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 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 $\mathcal{P}$=Line.flights.routing.arrival_airports.city. 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, in the ENIX, class index allocated on C_i contains OIDs of members of C_i as key values while in MIX the index allocated on class C_i contains values of A_i as key values.

3.4 VISIBILITY GRAPH

In this subsection we introduce some graph notations for objects and nested references along a path to facilitate the description of operations for the ENIX organization.

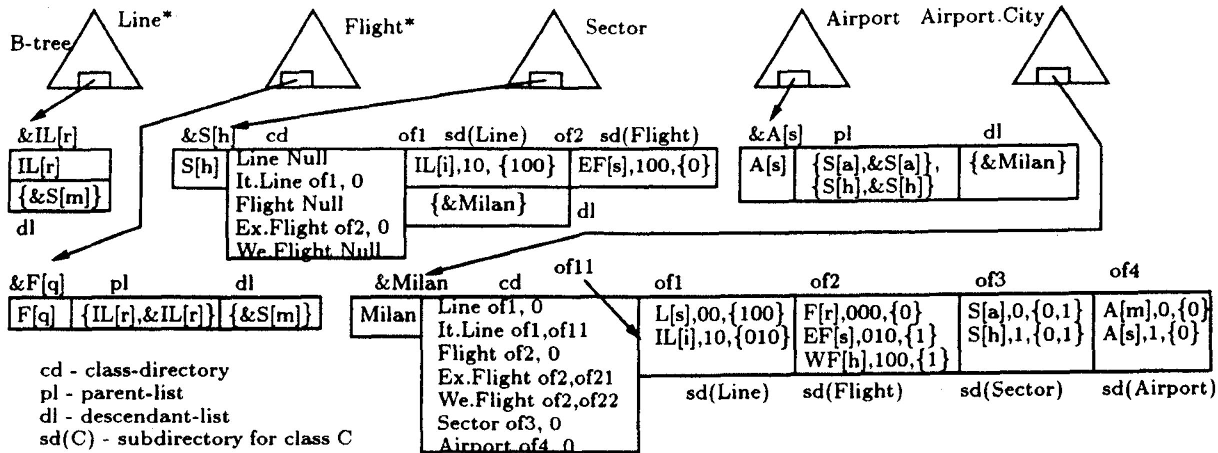

Figure 3. Enhanced nested-inherited index with $\Delta = \{3, 5\}$.

If an instance O of class $C_{i,k}$ has O' as a value of nested attribute $A_{j-1}, i \leq j \leq n+1$, where $O' \in C_{j,m}$ ($O' \in A_n$), than O' is called a **descendant** of O in class $C_{j,m}$ (attribute A_n) and O is called an **ancestor** of O' in class $C_{i,k}$. Also, we will say that O is *visible* for O'.

For any $O \in C_{i,k}$, let V_j be the set of the members of $C_j \in \text{class}(C_{i,k})$ visible for O. Also, let E_j be the set of directed edges corresponding to the nested references $(O', O''), O' \in V_j$ and $O'' \in V_{j+1}$. A graph $G = (V, E)$ constructed with the set of nodes $V = \bigcup_{j=1}^{i-1} V_j \cup \{O\}$ and the set of edges $E = \bigcup_{j=1}^{i-1} E_j$ is called a **visibility graph** for O. Given the path $\mathcal{P}$=Line.flights.routing.arrival_airports.city, Figure 4.a provides the visibility graph for value "Milan" of attribute Airport.city. In the figure, objects of the same inheritance hierarchy are circled.

The visibility graph for $O \in C_{i,k}$ ($O \in A_n$) is a rooted directed acyclic graph (DAG). It has i levels and edges between adjacent levels are only allowed in the graph. Level $j = 1, \ldots, i$ contains nodes for members of class C_j. Level i corresponds to the root of the DAG and contains solely the OID(attribute value) of O .

Clearly, a leaf record of O, instance of class $C_{i,k}$, in the far-look index allocated on the class C_i has been defined so that the class and object-directories in the record are an implementation of the visibility graph for O compressed as much as possible. An entry in a subdirectory of the object-directory corresponds to some node of the visibility graph together with edges going out of the node. Also, level i is omitted since OID of O is hold as a key value.

In the next sections we will use some basic manipulations with a visibility graph stored in a leaf record. Here we define them using the graph notation and give examples:

- **removal of a node from the visibility graph for O**: the node is removed from the graph as well as all nodes that become invisible for O. All the edges adjacent to a removing node are removed, too. Figure 4.b gives a result of the removal of S[h] from the visibility graph for "Milan".
- **removal of an edge from the visibility graph for O**: the edge is removed from the graph; if any node becomes invisible for O, the node and all adjacent edges are removed, too. Figure 4.c gives a result of the removal of the reference (WF[h], S[h]) from the visibility graph for "Milan".
- **reconstruction of the visibility graph for O$\in$ $C_{i,j}$**: Figure 4.d gives the visibility graph for A[s] reconstructed based on data stored in the leaf record for A[s] and visibility graphs for S[a] and S[h], ancestors of A[s] in class Sector.
- **addition of an edge e to the visibility graph for O**: the edge is added to the graph; if any new node becomes visible for O, that is, there is a chain of edges connecting the node with O, the node and all edges of the chain will be in the resulting graph. Figure 4.e gives the visibility graph for "Milan" after addition of the nested reference (F[t], S[a]).

4 OPERATIONS

In this section, we describe the following five operations for the ENIX organizations: *retrieval, object delete, nested reference delete, object insert* and *nested reference insert.*

4.1 RETRIEVAL

The ENIX supports a fast evaluation of predicates on the attribute A_n for queries having as target any class, or class hierarchy, in the scope of the A_n. As an example, suppose that an index is allocated on the path $\mathcal{P}$ =Line.flights.routing.arrival_airports.city and that a query is issued to retrieve all the instances of class ItalianLine landing at the airports of Milan. The processing of the query begins with a lookup on the A_n-index with key value equal to "Milan". The leaf record corresponding to the key value is accessed and the offset of the subdirectory for Line is extracted from the class-directory. Then,

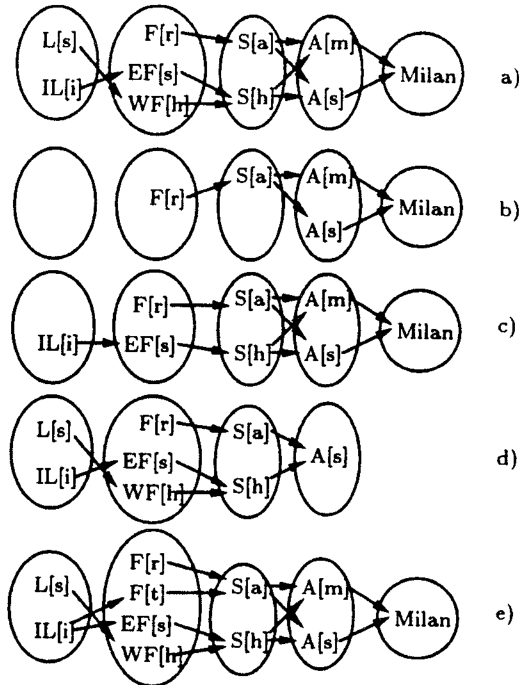

Figure 4. Visibility graph for value "Milan" of the attribute Airport.city.

all OIDs of ItalianLine in the subdirectory for Line are returned as the result, that is, {IL[i]}.

4.2 OBJECT DELETE

Given a path $\mathcal{P} = C_1.A_1.A_2....A_n$ and a class $C \in$ scope($\mathcal{P}$) having position $i, 1 \leq i \leq n+1$, suppose that an object O, instance of C, is deleted. In ENIX, the effect of an object delete operation must be that the identifier of the object $O \in C$ is removed from every leaf record in C_j-index, $j > i, j \in \Delta$ containing it. In the C_j-index, $prev(i) \leq j \leq i-1$, leaf record of an ancestor of O is updated if deleting O results in decreasing the descendant-list. Finally, leaf record of O is deleted from the C_i-index while O must be eliminated from the parent-list of its children in C_{i+1}. The following steps are executed (numbers in the brackets indicate subscripts of indices being updated):

1. $[i+1]$ If $i \leq n$ and $i+1 \notin \Delta$, the set of values SV of the attribute A_i of O is determined; then the C_{i+1}-index is accessed with the values of SV as key values and corresponding leaf records are modified by removing OID of O and its address from the parent-list of the records. Othewise, this step is omitted.

2. $[i]$ The C_i-index is searched with the value of O as key value and the leaf record of O is accessed. If $i \in \Delta$, for every object $O_j \in C_j^*, prev(i) \leq j \leq i-1$ in the object-directory of O, an entry $(O_j, \{O\})$ is inserted into list TDL_j. The record of O is removed and $k := i$. Go to step 4.
Otherwise, if $i \notin \Delta$, the set SP of addresses of the leaf records in $C_{next(i)}$-index are determined from the descendant-list of O. The leaf record of O is removed and $k := next(i)$.

3. $[next(i)]$ For a leaf record of $O' \in C_{next(i)}$ whose address is in the SP, object O is removed from the object-directory of O'. If during the removal, an entry of $O_j \in C_j^*, prev(i) \leq j \leq i-1$ is cancelled from the subdirectory for C_j, then O' is added to a list TDL_j. If no entry for O_j is in TDL_j, a new entry $(O_j, \{O'\})$ is inserted into the list. Otherwise, O' is added to the sublist of the entry for O_j.
After steps 2-3, the list $TDL_j, prev(i) \leq j \leq i-1$, contains entries for members of C_j. An entry for $O_j \in C_j^*$ has a form $(O_j, sublist)$ where $sublist$ consists of members of $C_{next(j)}$ that becomes invisible for O_j after deleting the object O.

4. $[prev(i) \ldots i-1]$ Every C_j-index such that the list TDL_j is not empty, is updated. The C_j-index is scanned with values of TDL_j. A leaf record of $O_j \in C_j^*$ is accessed only if record $(O_j, sublist)$ appears in TDL_j. The leaf record of O_j is updated by deleting all OIDs of $sublist$ from the descendant-list of the record.

5. $[j \geq next(k), j \in \Delta]$ The C_j-index, $j \geq next(k), j \in \Delta$ is updated in the same way as $C_{next(i)}$-index (step 3). The addresses of leaf records in C_j-index are determined by merging descendant-lists of leaf records updated in $C_{prev(j)}$-index. Each such leaf record is updated by removing OID of O from the object-directory.

As an example, suppose that the object A[s] must be removed from the database and that the ENIX with $\Delta = \{3, 5\}$ is allocated on the path $\mathcal{P}$=Line.flights.routing.arrival_airports.city. As Airport is the last class on the path, the first step of the algorithm is omitted. A lookup on the Airport-index is executed with the OID A[s] as search key. The corresponding leaf record is accessed and addresses of descendants are determined. List SP is {&Milan}.

The leaf record with key value equal to "Milan" is accessed in the index allocated on Airport.city and updated by removing the instance A[s] from the object-directory. As a result, entry for A[s] is eliminated from the subdirectory for Airport and the local identifier 1 of A[s] is removed from the list NCL in the subdirectory for Sector in both records of S[a] and S[h]. Since no entry has been deleted in subdirectories for Line, Flight and Sector, all the TDL lists are empty. Also, there is no more far-look index to update and algorithm stops.

4.3 NESTED REFERENCE DELETE

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and a class $C \in$ scope($\mathcal{P}$) having position $i, i \leq n$, suppose that nested reference from object O, instance of C, to instance O' of A_i, is deleted. The effect of a reference delete operation in the ENIX must be to remove the reference from every leaf record in C_j-index, $j > i, j \in \Delta$ containing it. In C_j-index,

$prev(i) \leq j \leq i$, leaf record of an instance referencing O is accessed for update only if the size of its descendant-list decreases due the reference deletion. Finally, O must be eliminated from the parent-list of the O' in C_{i+1}-index. The following steps are executed:

1. $[i+1]$ The C_{i+1}-index is accessed with the value of O' as key value. If $i+1 \notin \Delta$, the leaf record of O' is modified by removing OID of O and its address from the parent-list of the record. The set SP of addresses of the leaf records in the $C_{next(i+1)}$-index is determined from the descendant-list and $k := next(i+1)$. If $i+1 \in \Delta$, $SP = \{\&O'\}$ and $k := i+1$.

2. $[k]$ For a leaf record of $O'' \in C_k^*$ whose address is in SP, reference (O, O') is removed from the object-directory of O''. If during the removal a node for $O_j \in C_j, prev(i) \leq j \leq i$ is cancelled from the subdirectory for C_j, then O' is added to a list TDL_j. If no entry for O_j is in TDL_j, a new entry $(O_j, \{O'\})$ is inserted into the list. Otherwise, O' is added to a sublist of the entry for O_j.
   After this step, list $TDL_j, prev(i) \leq j \leq i$ contains entries for members of C_j. An entry for $O_j \in C_j^*$ has a form $(O_j, sublist)$ where *sublist* consists of members of $C_{next(j)}$ that becomes invisible for O_j after deleting the object O.

3. $[prev(i) \ldots i]$ Every C_j-index such that the list TDL_j is not empty, is updated. The C_j-index is accessed with values of TDL_j. A leaf record for $O_j \in C_j^*$ is accessed only if record $(O_j, sublist)$ appears in TDL_j. The leaf record of O_j is updated by deleting all OIDs present in *sublist* from descendant-list of the record.

4. $[j \geq next(k), j \in \Delta]$ The C_j-index, $j \geq next(k), j \in \Delta$ is updated in the same way as C_k-index (step 2). The addresses of leaf records in C_j-index are determined by merging descendant-lists of leaf records updated in $C_{prev(j)}$-index. Each such leaf record is updated by subtracting the reference (O, O') from the object-directory.

As an example, suppose that the nested reference between WF[h] and S[h] must be removed. Since Sector-index is of far-look type, $k := 3$ and $SP = \{\&S[h]\}$. A lookup on Sector-index is executed with S[h] as search key. The leaf record of S[h] is updated by removing the reference (WF[h],S[h]) from the object-directory. As a result, entries for WF[h] and L[s] are removed from subdirectories for Flight and Line, respectively, and lists TDL_1 and TDL_2 are created. The former contains an entry (L[s], {&S[h]}), the later contains (WF[h], {&S[h]}).

At step 3, Line-index and Flight-index are retrieved and address &S[h] is deleted from descendant-lists of entries for L[s] and WF[h]. Finally, since the descendant-list for S[h] is {&Milan}, the leaf record of "Milan" is accessed in Airport.city-index and the reference (WF[h], S[h]) is removed from the object-directory resulting in removing the entries for L[s] and WF[h].

4.4 OBJECT INSERT

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and a class $C \in$ scope($\mathcal{P}$) having position i, suppose that an object O, instance of class C, must be inserted. Since the insertion of O in ENIX changes no visibility graph, the C_i-index is accessed using the OID of O as key value and a new leaf record with OID of O is inserted.

4.5 NESTED REFERENCE INSERT

Given a path $\mathcal{P} = C_1.A_1.A_2 \ldots A_n$ and a class $C \in$ scope ($\mathcal{P}$) having position $i, i \leq n$, suppose that nested reference from object O, instance of C, to instance O' of A_i, is inserted. The effect of a reference insert operation must be to add the reference to every leaf record in C_j-index, $j > i, j \in \Delta$ which is a descendant of O'. In C_j-index, $prev(i) \leq j \leq i-1$, leaf record of an instance referencing O is accessed for update only if the insertion of the reference (O, O') makes visible new descendants in the class $C_{next(i)}$. Finally, O must be inserted in the parent-list of the O' in C_{i+1}-index. The following steps are executed:

1. $[i, i+1]$ The C_i-index and C_{i+1}-index are searched with values of O and O', respectively. Leaf records of O and O' are accessed. If $i+1 \notin \Delta$, parent-list of O' is modified by inserting OID of O and its address. Then descendant-list of O' is copied to a list SP and parent-list of O is copied to a list $Par(i)$. If $i+1 \in \Delta$, the list SP contains the only OID of O'. The list SP is added to the descendant-list of O. If the size of the descendant-list increases, the newly added elements remain in the list SP, the others are deleted.

2. $[prev(i) \ldots i-1]$ Then, C_m-indices, $m = i-1, \ldots, prev(i)$ are consequently updated. Whenever any leaf record of C_m-index with address from list $Par(m+1)$ is accessed, its descendant-list is updated inserting the list SP and parent-list of the leaf record is extracted and inserted in the list $Par(m)$. Concurrently, visibility graph for O is reconstructed.

3. $[j > i, j \in \Delta]$ The leaf records of C_j-index are accessed with addresses taken from descendant lists of $C_{prev(j)}$-index. The leaf records are updated by adding the reference (O, O') using the visibility graph for O reconstructed at the previous step.

For example, suppose that reference (F[t], S[a]) must be inserted. The leaf records of F[t] and S[a] are retrieved from the indices allocated on the classes Flight and Sector, respectively. Since Sector-index is of far-look type, the list SP is {S[a]} and address &S[a] is inserted in the descendant-list of F[t]. The leaf record of IL[i], parent of F[t], is accessed in Line-index and updated by inserting &S[a] in the descendant-list. Then, the reference (F[t], S[a]) is added to the object-directory of S[a]. As a result, the reference (IL[i],F[t]) appears in the graph visibility for S[a]. "Milan" is the only descendant of S[a] in Airport.city and the reference is added to the object-directory of "Milan" resulting in appearance of the reference (IL[i],F[t]) in it as well.

5 COMPARISON RESULTS

In this section, we analyze the performance of the ENIX organization, comparing it with the NIX and MIX. We first summarize the cost model and parameters. Then, we present the results of several tests.

Cost model

The cost model analytically evaluates the number of I/Os for all operations described in the previous section. It takes into account a large number of parameters, describing the topology of classes in a path $\mathcal{P}$. The cost model for the MIX and NIX organizations has been developed in [BeFo95] and the cost model for ENIX is based on the same assumptions and considerations [BSS95]. Most of the assumptions made in the cost model 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 as well as values of attribute A_i are uniformly distributed among the instances of the inheritance hierarchy rooted at the class C_{i+1}.

The selection of the best ENIX

As we have seen in the subsection 3.2, there are 2^{n-1} distinct variants of ENIX allocation for a given path of length n. Therefore, unlike NIX, the ENIX organization becomes sensible to the operation workload. The workload is a set of frequencies of all operations in the path. The selection of the best ENIX means choosing such variant of ENIX that provides the minimal total cost with respect to the given workload. Here we are faced with a complexity problem. Indeed, the problem of selecting the best variant from the 2^{n-1} possible variants has an exponential complexity. Therefore, an exhaustive enumeration of all ENIX variants can be done for a path whose length is not greater than 15-20. In real databases a path longer than 10 is unlikely to be frequent and such enumeration is quite sufficient for the practical goals. In the general case, branch-and-bound technique is applied that considerably reduces the number of variants to evaluate [BB90].

Tests

We tested the three index organizations, namely, MIX, NIX and ENIX, for a wide range of database parameters. The NIX is considered as a special case of the ENIX when the list Δ contains the only element $n+1$.

The costs of the following operations in the path of length 4 are being evaluated. The query is issued with a class C_i, $1 \leq i \leq 4$ as a target class. The update operation may be issued with an instance of any class in the path or value of the attribute A_4. A reference can be inserted/deleted between objects of two adjacent classes or class C_4 and attribute A_4. Since the three index organizations cope with inheritance hierarchies in the same way, we vary in the tests only the parameters that are intrinsic for an entire inheritance hierarchy rooted at class C_i, $1 \leq i \leq 4$. The parameters varied in the experiments are as follows:

1. D_i; number of members of class C_i, $i = 1, \dots, 4$.

2. D_5; number of values of attribute A_4.

3. fan_i; average number of children for members of class C_i, $i = 1, \dots, 4$.

4. $prob$; probability of a query.

The total cost is calculated as $prob * Retrieval + (1 - prob) * Update$, where $Retrieval$ and $Update$ are the sums of retrieval and update costs, respectively. In the first test, we assume a naive workload where query and update operations in the path are equally likely ($prob = 0.5$). The reason is to determine how each operation affects the total cost for all of the tested index configurations. Several tests has been performed and only some of them are reported here.

Test 1. $D_1 = 100.000, D_2 = 150.000, D_3 = 200.000,$
$D_4 = 250.000, D_5 = 2.500;$
$fan_1 = 5, fan_2 = 3, fan_3 = 2, fan_4 = 1.$

Costs for all index organizations are collected in Table 1.

Index/ Storage	Operation	C_1	C_2	C_3	C_4	A_4	Total cost
NIX	RE	9	7	5	4		
	OD	179.	43	22	9	1441	
	RD	164	34	16	10		
	OI	4	4	4	4	4	
96MB	RI	168	40	30	25		1114
ENIX with {2,4,5}	RE	9	7	5	4		
	OD	186	51	17	13	107	
	RD	162	39	18	10		
	OI	4	4	4	4	4	
112MB	RI	162	42	20	10		442
MIX	RE	543	266	108	3		
	OD	15	14	11	8	4	
	RD	4	4	4	4		
	OI	0	4	4	4	4	
51MB	RI	4	4	4	4		509

RE Retrieval; OD Object Delete; RD Reference Delete; OI Object Insert; RI Reference Insert

Table.1. Operation costs in test 1.

The ENIX with $\{2,4,5\}$ provides the best total cost and the MIX performs on average better that NIX. Both NIX and ENIX have low retrieval cost for any class in the path. Instead, the MIX requires too much time to retrieve the members of class C_1, C_2 and C_3. By comparing the ENIX and NIX, one can discover that the most impressive difference appears in the delete cost of attribute value of A_4. The cost hits hardly the total cost of the NIX whereas it is quite moderate for the ENIX.

Test 2. $D_1 = 250.000, D_2 = 200.000, D_3 = 150.000,$
$D_4 = 100.000, D_5 = 10.000;$
$fan_1 = 1, fan_2 = 2, fan_3 = 3, fan_4 = 3.$

Costs for all index organizations are collected in Table 2.

In this test, the naive workload has been substituted with a more realistic one. It assumes that the frequency of any update operation is four times less than that of a retrieval operation($prob = 0.8$). Under these conditions, the total cost for the NIX is essentially lower that for MIX but ENIX with $\{2,3,5\}$ has the lowest cost.

The main conclusions from the experiment results can be summarized as follows:

Index/ Storage	Operation	C_1	C_2	C_3	C_4	A_4	Total cost
NIX	RE	4	4	4	4		
	OD	21.	24	21	6	249	
	RD	17	17	12	7		
	OI	4	4	4	4	4	
50MB	RI	21	21	17	14		106.
ENIX with {2,3,5}	RE	4	4	4	4		
	OD	23	29	14.	11	33	
	RD	24.	23	15.	7		
	OI	4	4	4.	4	4	
56MB	RI	24	26	16	7		67
MIX	RE	382	211	68	3		
	OD	4.	12	15	14	4	
	RD	4	4	4	4		
	OI	0	4	4	4	4	
20MB	RI	4	4	4	4		552

RE Retrieval OI Object Insert
OD Object Delete RI Reference Insert
RD Reference Delete

Table.2. Operation costs in test 2.

1. The MIX organization has always the best performance for update operations while NIX and ENIX guarantee the fast evaluation of queries. In database applications characterized by queries predominance the advantages of NIX and ENIX over MIX become essential.

2. The ENIX has a better performance than NIX for the object delete operations. The cost of the query is the same for both indices. Also, the object insert is fast for both indexing techniques since the operation changes no visibility graph. As a result, the ENIX provides a lower total cost than the NIX does. In all the tests conducted in a wide range of parameters, the best ENIX never reached its minimum with $\Delta = \{n+1\}$ if only workload frequencies of update operations are different from 0.

3. The major disadvantage of both NIX and ENIX organizations is the storage increase. In most tests, NIX organization requires 2-3 times more storage that MIX, and the ENIX providing the best total cost usually occupies 10%-30% more storage that NIX. Not every database application may allow extra memory overhead and two solutions can be here proposed. The first consists simply in selecting those index allocations which safisty some storage limitation. The second modifies the total cost function to be minimized by adding a storage cost properly weighted.

6 CONCLUSION

Experiments have shown that though the NIX is one of the possible ENIX variants, the best ENIX never reaches the minimum total cost at the NIX if only update operations have some positive weight in the workload. Here we try to explain this fact. The nested-inherited index was introduced to maintain the basic set of precomputed joins between the attribute A_n and all classes $C_i, 1 \leq i \leq n$ in the path. The index provides an integrated support for queries along both aggregation and inheritance hierarchies and has low retrieval cost. However, the cost of an update operation is high. Indeed, when an object of a class in the path is deleted/inserted, the indices supporting the basic joins should be updated. The easiest way would be to make use of the precomputed joins between the class the instance being updated belongs to, and other classes in the path. But the NIX does not support such joins and it has to reconstruct the appropriate fragment of the joins. The time needed for this reconstruction results in the high operation cost.

Thus, the maintenance of the basic set of precomputed joins alone is not the optimal strategy. The ENIX overcomes this problem by adding some precomputed joins between the classes in the path to the basic set of joins the NIX maintains. Moreover, the ENIX proposes the new structure of the leaf-node record permitting to store in a compressed mode a visibility graph and efficiently use it when a fragment of a join not present in the index should be reconstructed.

The set of joins that should be added to the basic set is determined from the logical database parameters and frequencies of the operations in the path. The set of additional joins minimizing the total cost in a given workload together with the basic set composes the best enhanced nested-inherited index.

The work reported in this paper can be extended by adding the proposed index organization to the repertoire of index organizations used in our previous work on index allocation [Ber94]. When no indexing technique provides a satisfactory cost of operations, the path can be split into several subpaths and possibly different indexing techniques are used on each subpaths.

REFERENCES

[Ber93] E Bertino A Survey of Indexing Techniques for Object-Oriented Databases *Proc of Dagsthul Seminar on Query Processing in Object-Oriented, Complex- Object and Nested Relational Databases*, C Freytag, D Maier, and G. Vossen, eds , Morgan- Kaufmann, 1993

[Ber94] E Bertino On Indexing Configuration in Object-Oriented Databases *VLBD Journal*,v 3, No.3 (1994)

[BeFo95] E Bertino, P Foscoli Index Organizations for Object-Oriented Database Systems *IEEE Trans. on Knowledge and Data Eng.*, v.7, No 2 (1995)

[BSS95] E. Bertino, S.Salerno, B.Shidlovsky. Enhanced Nested-Inherited Index for OODBMS *Extended version*, (1995), in ftp://eiler.crmpa.unisa.it/pub/boris/cikm95e.ps.Z

[BB90] G. Brassard, P Bratley *Algorithmics. Theory and Practice*. Prentice-Hall, 1990

[Gal92] L. J. Gallagher. Object SQL Language Extensions for Object Data Management In *Proc. CIKM*, Baltimore, Maryland, November 1992.

[Gra93] G. Graefe Query Evaluation Techniques for Large Databases *ACM Computing Surveys*, v.25, No 2 (1993).

[KKD89] W. Kim, K.C Kim, A Dale Indexing Techniques for Object-Oriented Databases. *Object-Oriented Concepts, Databases, and Applications*, W.Kim and F.Lochovsky, eds., Addison- Wesley, 1989

[MeTs90] K. Mehlhorn, A Tsakalidis Data Structures In J Van Leeuwen, editor, *Handbook of Theoretical Computer Science, v. A. Algorithms and Complexity*, Elsevier,(1990)

[Val87] P Valduriez Join Indices *ACM Trans. on Database Systems*, v.12, No.2 (1987), pp.218-246

[Yao77] S.B Yao Approximating Block Access in Database Organizations, *ACM Comm.*, v.20, No 4 (1977),pp. 260-261