On the Selection of Optimal Index Configuration in OO Databases*

Sunil Choenni[1] Elisa Bertino[2] Henk M. Blanken[1] Thiel Chang[3]

[1]University of Twente
P.O. Box 217, 7500 AE Enschede,
The Netherlands

[2]University of Genova
Via L.B. Alberti 4, 16132 Genova,
Italy

[3]G.A.K.
P.O. Box 8300, 1005 CA Amsterdam,
The Netherlands

Abstract

An operation in object-oriented databases gives rise to the processing of a path. Several database operations may result into the same path. We address the problem of optimal index configuration for a single path. As it will be shown an optimal index configuration for a path can be achieved by splitting the path into subpaths and by indexing each subpath with the optimal index organization. We present an algorithm which is able to select an optimal index configuration for a given path. For the moment we consider a limited number of existing indexing techniques (simple index, inherited index, nested inherited index, multi-index, and multi-inherited index) but the principles of the algorithm will remain the same adding more indexing techniques.

1 Introduction

Object-oriented data models are based on some fundamental concepts which will be discussed briefly. A real world entity is represented by an object. Each object is uniquely identified by an object identifier (*oid*) which is supposed to be generated by the database system. An object has associated a state and a behaviour. The state of the object is defined by the value of its attributes. The value of an attribute can be either an atomic object, such as an integer, a string, etc or a non-atomic object; a non-atomic object in turn consists of a set of attributes. Therefore, objects can be built in terms of other objects giving rise to nested objects. The behaviour of an object is specified by a set of methods which operate on the object state.

Objects with the same attributes and behaviour are grouped in classes. The set of attributes in the class define the object structure. Each attribute has a value which falls in a pre-defined domain. The domain can be either an atomic class or a non-atomic class. In this way a relation between two classes C and C', in which C' is the domain of an attribute of class C, can be established which is called a part-of relationship between the two classes. Since the class C' has on its turn part-of relationships with the classes of its attributes and so on, the definition of a class C results in a tree with C as root. This is called an aggregation hierarchy. The attributes of class C' in the aggregation hierarchy will be called *nested* attributes of C. An example of an aggregation hierarchy is given in Figure 1. Multi-value attributes are marked by '+'. Table 1 introduces some abbreviations which will be used in the rest of the paper.

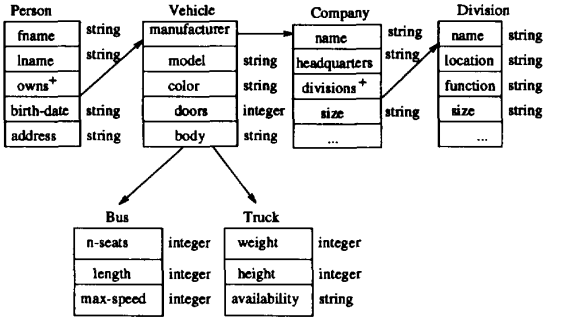

Figure 1: Example of an object-oriented logical schema

Classes, in an object-oriented database schema, are also organized in inheritance hierarchies. The inheritance hierarchy provides the possibility to define a class, called subclass, as a specialization of another existing class, called superclass. A subclass inherits the attributes and methods of its superclass and may have additional attributes and methods. In Figure 1 the classes Vehicle, Truck and Bus form an inheritance hierarchy.

Much research has been done and is still going on to develop well founded data models which incorporate the above mentioned concepts. To be viable, not only the concepts have to be supported by an architecture that directly implements them but querying and maintaining the database should require an acceptable amount of time. This may be established by two techniques which differ fundamentally. One way is to place the objects efficiently on disks such that fetching the objects requires a minimal number of page accesses. Clustering objects belonging to different classes and storing them on the same page is a topic belonging to this field.

*This research has been partially supported by the Dutch Organization for Scientific Research (NWO) while the first author was visiting the University of Genova for three months.

Per = Person	*Veh = Vehicle*
Div = Division	*Comp = Company*
man = manufacturer	*divs = divisions*

Table 1: Used abbreviations

Another technique to accelerate database operations is by constructing efficient access structures on a database given a certain physical implementation of the database. The allocation of indices on attributes belongs to this field. In this paper we study how to utilize this technique under certain assumptions. We assume that a page contains objects of only one class and only forward references between objects exists. Furthermore, we assume that attributes do not have NULL values. Figure 2 contains some instances of Figure 1.

A database operation in an object-oriented data model against a nested predicate leads to the processing of a path. Consider the following database operation against the aggregation hierarchy of Figure 1: 'Retrieve the persons who own a bus manufactured by the company Fiat'. To process this query we have to navigate through the classes *Per*, *Bus* and *Comp*, resulting in the path *Per.owns.man.name*. Evaluating this path in a naive way by taking an object of the class *Per*, checking which vehicle he owns and in the case he owns a bus checking if this is manufactured by the Fiat company may be very expensive. Therefore, several indexing techniques [2, 5, 6, 10, 11] and caching techniques [9] have been proposed to accelerate the processing of such paths.

In this paper, we address the following problem: 'given a number of operations on a database schema leading to the same path P and some other database characteristics (such as the number of objects in each class, page size, etc.), determine the optimal index configuration for the path; that is the one with the lowest costs in processing the path.' The idea of our approach is to split the path P into a proper set of subpaths and to allocate the most beneficial index on each subpath. The set of indexed subpaths will be called an *index configuration* of P.

We note, however, that the case considered in this paper, that is a single path, is quite significant. Indeed, a path corresponds to a nested predicate which is equivalent in a relational query language to a restriction on a relation attribute and several joins among different relations.

A comparable research has been reported in [3] which differs from this work on the following important point. The indexing techniques chosen in this paper deal with inheritance as well as with nested objects while the indexing techniques in [3] deal only with nested objects. Since, the foundations of optimization techniques for object-oriented databases is still in its infancy there is substantial need for such research.

The remainder of this paper is organized as follows: Section 2 summarizes the main principles of the indexing techniques considered in this paper. Section 3 is devoted to the mathematical models of the indexing techniques and the workload model. Section 4 is devoted to the notion of index configuration and the basics to develop an algorithm for the selection of optimal index configurations. In Section 5 the algorithm will be presented. Section 6 concludes the paper.

2 Indexing techniques

To make the paper self-contained we briefly discuss the main principles of the indexing techniques which we will consider. These are the simple index, multi-index, inherited index, multi-inherited index and nested inherited index. Before starting this discussion we recall some preliminary definitions in the next section which are adopted from [1, 10].

2.1 Preliminary definitions

As already noticed processing a database operation gives rise to the processing of a path of the form $P = C_1.A_1.A_2.....A_n$. The following gives a formal definition of a path.

Definition 2.1 Given an aggregation hierarchy H, a path $P = C_1.A_1.A_2.....A_n$ in which

- a class C_l appears at most once in the path
- C_1 is a class in H
- A_1 is an attribute of C_1
- A_l is an attribute of C_l in H, such that C_l is the domain of attribute A_{l-1} of class C_{l-1}, $1 < l \leq n$. □

Some notions related to definition 2.1 follow. The length of a path P will be denoted by $len(P)$ and is equal to the number of classes along the path. The classes along a path are denoted collectively by $class(P)$. The scope of a path P consists of all the classes in $class(P)$ and their subclasses and is denoted by $scope(P)$.

A subclass z of a class C_l is denoted as $C_{l,z}$. Note, the second subscript does not imply an order of the subclasses. It is only meant to distinguish the classes by labeling them. In the case a class does not have an inheritance hierarchy or when it is clear that we mean the root class at the highest level we omit the second subscript. The set of subclasses in an inheritance hierarchy rooted at class $C_{l,z}$ together with the root class is denoted as $C^*_{l,z}$. Furthermore the attribute A_n is called the *ending attribute* and the class C_1 is called the *starting class.*

Ex 2.1 Consider $P_e = Per.owns.man.name$ in the aggregation hierarchy of Figure 1. The classes involved with path P_e are Per, Veh and $Comp$. Therefore, $len(P_e) = 3$ and $class(P_e) = (Per, Veh, Comp)$. Since Vehicle has two subclasses Bus and Truck and the other classes in $class(P_e)$ do not have subclasses $scope(P_e) = (Per, Veh, Bus, Truck, Comp)$. □

2.2 Index organizations

We discuss the main principles of the index organization which will be considered in the rest of the paper.

Simple index (SIX) A simple index is an index on an attribute of a single class. With each value

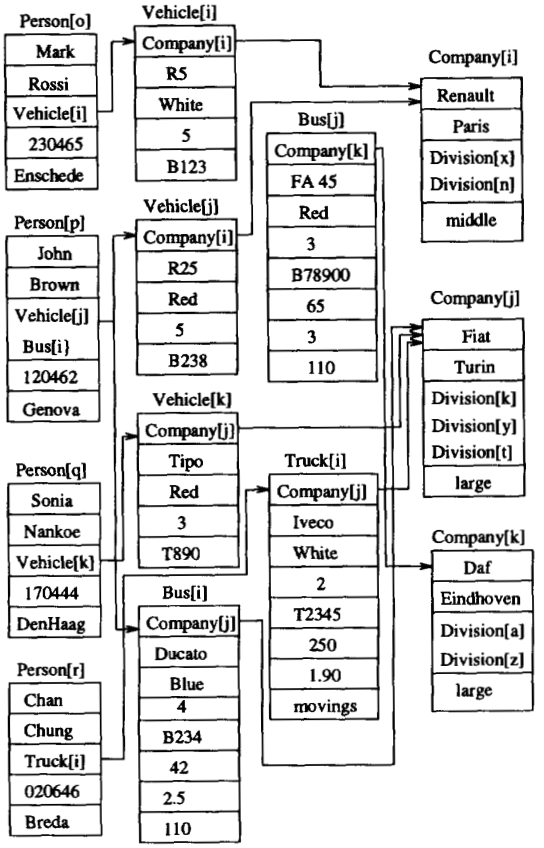

Figure 2: Some objects belonging to schema 1.

v of the indexed attribute the *oids* of the objects are associated which have v as value for the indexed attribute. Considering Figure 2, an index on the attribute *color* of the class Veh results into the pairs (White, {Vehicle[i]}) and (Red, {Vehicle[j], Vehicle[k]}). Note, Vehicle[i], Vehicle[j] and Vehicle[k] are *oids*.

Inherited index (IIX) An inherited index is an index on an attribute of all classes of a class inheritance-hierarchy rooted at a particular class. This organization was introduced in [10] and is also known as class hierarchy index. Allocating an inherited index on the attribute *color* of the class Veh considering Figure 2 again results into the pairs (White, {Vehicle[i], Truck[i]}), (Red,{Vehicle[j], Vehicle [k], Bus[j]}) and (Blue,{Bus[i]}).

Multi-index (MX) A multi-index allocates an index on each class in the scope of a path. The indexed attributes are the ones specified in the path. Suppose we have a path $P = C_1.A_1.A_2.....A_n$ and a class $C_l \in scope(P)$ such that A_l is an attribute of C_l, then an index on A_l associates with each value v of A_l the *oids* of the objects of C_l having v as value for attribute A_l. Considering the path $P_e = Per.owns.man.name$ and the figures 1 and 2 an MX on this path results into an index on *name* of the class *Comp*, an index on *man* of the classes Veh, Bus and $Truck$ and an index on the attribute *owns* of the class Per. Thus, we achieve the following pairs for the index on *name* (Fiat, {Company[j]}), (Renault, {Company[i]}) and (Daf, Company[k]}). An index on the attribute *man* of the classes Veh, Bus and $Truck$ results into entries into three indexes, namely (Company[i], {Vehicle[i], Vehicle[j]}) and (Company[j], {Vehicle[k]}) respectively (Company[j], {Bus[i]}) and (Company[k], {Bus[j]}) respectively (Company[j], {Truck[i]}. Finally an index on the attribute *owns* of the class Per looks as follows (Vehicle[i], {Person[o]}), (Vehicle[j], {Person[p]}), (Vehicle[k], {Person[q]}), (Truck[i], {Person[r]}) and (Bus[i], {Person[p]}).

Multi-inherited index (MIX) A MIX organization may be regarded as a combination of the last two index organizations. A multi-inherited index differs from a multi-index in the sense that a multi-inherited index allocates an index on all classes $\in class(P)$ while the multi-index allocates an index on all classes $\in scope(P)$. Furthermore, if a class $C_l \in class(P)$ forms an inheritance hierarchy then an index on attribute A_l associates with each value v of A_l the *oids* of the objects of C_l and of its subclasses which have v as value for A_l (implying an inherited index). Considering $P_e = Per.owns.man.name$ and the figures 1 and 2 again a multi-inherited index on the path results into an index on *name* of the class *Comp*, (Fiat, {Company[j]}), (Renault, {Company[i]}) and (Daf, Company[k]}, an index on *man* of the class Veh and its subclasses, (Company[i], {Vehicle[i], Vehicle [j]}), (Company[j], {Vehicle[k], Bus[i], Truck[i]}) and (Company[k], {Bus[j]} and an index on the attribute *owns* of the class Per, (Vehicle[i], {Person[o]}), (Vehicle[j], {Person[p]}), (Vehicle[k], {Person[q]}), (Truck[i], {Person[r]}) and (Bus[i], {Person[p]}).

Nested inherited-index (NIX) A nested inherited-index associates with each value v of the nested attribute A_n of a path P the *oids* of the objects of each class in $scope(P)$ having v as value for the nested attribute. An example of a NIX on path $P_e = Per.owns.man.name$ considering the figures 1 and 2 looks as follows: (Fiat, ({Company[j]}, {Vehicle[k]}, {Bus[i]}, {Truck[i]}, {Person[p], Person[q], Person[r]})), (Renault, ({Company[i]}, {Vehicle[i], Vehicle[j]}, {Person[o], Person[p]})) and (Daf, ({Company[k]}, {Bus[j]})). The relations between the objects are kept in an auxiliary index as will be discussed in section 3.

For the state of the art with regard to indexing techniques in object-oriented databases we refer to [4]. It should be clear that a SIX and an IIX can be regarded as special cases of an MX respectively a MIX. Therefore, we will not consider them in further discussions

explicitly.

3 Mathematical models

To be able to derive the costs in processing an operation with a certain indexing technique we have to know how this index is stored and how the operation will be processed using the index. In the following we restrict to operations consisting of equality predicates which have the form $A_n =' a'$, meaning that predicates are related to the *ending attribute* of a path. The extension to range predicates is straightforward. The cost models for the different indexing techniques will be derived in section 3.1. The model which will be used for the workload will be the subject of section 3.2.

3.1 Cost models

We assume that a page contains objects of only one class. Only the number of page accesses will be taken into account as cost factor. Furthermore, indices are supposed to be organized as B^+-trees and the leaf nodes are chained. The non-leaf node records contain pairs (attribute value, pointer). The pointer contains the physical address of the next-level index node. The leaf nodes contain the index records. The information stored in an index record differs per indexing technique. We will discuss briefly the index records of the several indexing techniques. Before going into details we present some frequently used symbols in Table 2.

Two pairs of functions will be used frequently. The first pair concerns the retrieval and maintenance costs of a specified *single* index record, denoted as CRL and CML respectively. In the following we assume that attribute A_l of class $C_{l,z}$ is the indexed attribute. The function CRL is defined as:

$$CRL(h_X, pr_X) = \begin{cases} h_X & ln_X \leq p \\ h_X - 1 + pr_X & \text{otherwise} \end{cases}$$

in which h_X is the height of the index with organization X, ln_X is the average length of an index record in an index with organization X and pr_X is the average number of pages involved in the retrieval of an index record with regard to index organization X. If an index record occupies more than one page, some organizations retrieve, depending on the database operation, only a fraction of the index record. This may happen for the NIX if the scope of a path and the cardinality of the classes are large and for IIX and MIX if the number of classes involved in the inheritance hierarchy and the cardinality of these classes are large. It should be clear that if the whole index record has to be retrieved $pr_X = \lceil ln_X/p \rceil$. A procedure to compute the height of an index can be found in [7]

The maintenance cost of a single index record can be expressed by:

$$CML(h_X, pm_X) = \begin{cases} h_X + 1 & ln_X \leq p \\ h_X - 1 + 2 * pm_X & \text{otherwise} \end{cases}$$

in which pm_X is the average number of pages involved in maintaining an index record with regard to an index organization X. The extra page in the above formula in the case $ln_X \leq p$ is required to rewrite the page. In the case a record occupies more than one page we assume that only the pages which should be modified are retrieved and updated.

In the following we consider the values for pr_X and pm_X as input parameters. The value pm_X may differ depending on the fact whether an index should be maintained due to a deletion or an insertion. Whenever this is the case we use the notation pmd_X and pmi_X respectively to distinguish these situations. For more details with regard to these parameters, we refer to [7].

The second pair of functions concerns the maintenance and retrieval cost of a *set* of index records, denoted as CRT and CMT respectively. To retrieve t_k index records at level k of an index from n_k records stored on p_k pages the number of page accesses (npa) can be estimated by the formula of Yao [12]:

$$npa(t_k, n_k, p_k) = p_k(1 - \prod_{i=1}^{t_k} \frac{n_k(1 - 1/p_k) - i + 1}{n_k - i + 1})$$

So, in the case tx index records have to be retrieved from an index with height h_X, CRT is

$$CRT(h_X, tx, pr_X) = \begin{cases} \sum_{k=1}^{h_X} npa(t_k, n_k, p_k) & ln_X \leq p \\ \sum_{k=1}^{h_X - 1} npa(t_k, n_k, p_k) + tx * pr_X & \text{otherwise} \end{cases}$$

Note, $t_{h_X} = tx$ and $t_{k-1} = npa(t_k, n_k, p_k)$. The costs for maintaining tx index records in an index with height h_X is

$$CMT(h_X, tx, pm_X) = \begin{cases} \sum_{k=1}^{h_X} npa(t_k, n_k, p_k) + npa(t_{h_X}, n_{h_X}, p_{h_X}) & ln_X \leq p \\ \sum_{k=1}^{h_X - 1} npa(t_k, n_k, p_k) + 2tx * pm_X & \text{otherwise} \end{cases}$$

Note, a page will be rewritten if the maintenance of all index records on the page is completed. So, a page will be fetched only once.

MX As it has been shown an MX organization allocates an index to the corresponding attributes of each class in the scope of a path $P = C_1.A_1.A_2...A_n$. An index look up on an attribute A_l of class $C_{l,z}$ results into a set of *oids* belonging to the class $C_{l,z}$. An index look up on an attribute A_{l-1} for these *oids* in turn will result for each of them into a set of *oids* belonging to the classes rooted at $C_{l-1,1}$ and so on. So, the number of *oids* obtained by a look up of an indexed attribute A_l corresponding with the class $C_{l,z}$, ($noid_{l,z}$) for a given value v belonging to the ending attribute A_n is estimated by the formula:

$$noid_{l,z} = k_{l,z} \prod_{i=l+1}^{n} \sum_{j=1}^{nc_i} k_{i,j}$$

in which $k_{i,j}$ is the average number of objects in class $C_{i,j}$ having the same value for attribute A_i, $1 \leq i \leq n$, $1 \leq j \leq nc_i$.

$C_{l,z}$	=	subclass z which has as root the class C_l
$h^X_{l,z}$, $np^X_{l,z}$, $ln^X_{l,z}$	=	the height of an index allocated on attribute A_l of $C_{l,z}$ with organization X, # leaf pages to store the index respectively the average length of an index record
h_X, np_X, ln_X	=	shorter notations for $h^X_{l,z}$, $np^X_{l,z}$, and $ln^X_{l,z}$ in the case attribute A_l of $C_{l,z}$ is not relevant or it is clear which attribute is meant
$h_{l,z}$, $np_{l,z}$, $ln_{l,z}$	=	shorter notations for $h^X_{l,z}$, $np^X_{l,z}$, and $ln^X_{l,z}$ in the case the index organization is not relevant or it is clear which organization is meant
nc_l	=	# classes involved in an inheritance hierarchy rooted at class C_l
p	=	page size
$d_{l,z}$	=	# distinct values for attribute A_l of class $C_{l,z}$
$n_{l,z}$	=	# objects in class $C_{l,z}$
$nin_{l,z}$	=	average number of values held by a (multi-valued) attribute A_l of an object of class $C_{l,z}$
$k_{l,z}$	=	average number of objects in class $C_{l,z}$ with the same value for attribute A_l ($\frac{n_{l,z} * nin_{l,z}}{d_{l,z}}$)
$\overline{nin}_{l,z}$	=	average number of values held in the nested attribute A_n of an object of class $C_{l,z}$
pr_X	=	average number of pages involved in a retrieval of an index record with regard to an index organization X allocated on an attribute A_l
pm_X	=	average number of pages involved in maintaining an index record with regard to an index organization X allocated on an attribute A_l
pmd_X, pmi_X	=	average number of pages which should be rewritten to perform a deletion respectively an insertion in an index organization X allocated on attribute A_l
$par_{l,z}$	=	# parents of an object in class $C_{l,z}$ ($\sum_{i=1}^{nc_{l-1}} k_{l-1,i}$)
nar_{l+1}	=	number of auxiliary records involved in the distribution of $nin_{l,z}$ values over the class C_{l+1} and its subclasses in a NIX organization
$narp_l$	=	number of auxiliary records involved in the distribution of $par_{l,z}$ values over the class C_{l-1} and its subclasses in a NIX organization

Table 2: List of used symbols

To estimate the number of *oid*s obtained by a look up of an indexed attribute A_l corresponding with the class $C_{l,z}$ and all its subclasses, ($noid^*_{l,z}$) the following formula can be used: $noid^*_{l,z} = \sum_{C_{l,i} \in C^*_{l,z}} noid_{l,i}$.

The retrieval cost of a set of index records (due to a query) using a multi-index depends on the position of a class with respect to the ending attribute against which a predicate has to be evaluated. An evaluation against the ending attribute A_n with respect to a class $C_{l,z}$ requires $1 + \sum_{i=l+1}^{n} nc_i$ index looks up while an evaluation with respect to the classes $C^*_{l,z}$ requires $nc_{l,z} + \sum_{i=l+1}^{n} nc_i$ look ups ($nc_{l,z}$ is the number of subclasses rooted at class $C_{l,z}$). The number of records which has to be retrieved in the index corresponding with a class $C_{l,z}$ due to a query against the ending attribute is given by $noid^*_{l+1,1}$. Since we restrict our attention to equality predicates we define $noid^*_{n+1,1} = 1$. The retrieval cost of a set of index records with regard to a class $C_{l,z}$ using a multi-index, $CR_{MX}(C_{l,z})$ can be expressed by:

$$CR_{MX}(C_{l,z}) = CRT(h_{l,z}, noid^*_{l+1,1}, pr_{MX}) + \sum_{i=l+1}^{n-1} \sum_{j=1}^{nc_i} CRT(h_{i,j}, noid^*_{i+1,j}, pr_{MX}) + \sum_{j=1}^{nc_n} CRL(h_{n,j}, pr_{MX})$$

and the retrieving cost of a set of index records with respect to $C^*_{l,z}$ is:

$$CR_{MX}(C^*_{l,z}) = \sum_{C_{l,f} \in C^*_{l,z}} CRT(h_{l,f}, noid^*_{l+1,f}, pr_{MX}) + \sum_{i=l+1}^{n-1} \sum_{j=1}^{nc_i} CRT(h_{i,j}, noid^*_{i+1,j}, pr_{MX}) + \sum_{j=1}^{nc_n} CRL(h_{n,j}, pr_{MX})$$

in which $h_{i,j}$ is the height of the index on attribute A_i corresponding with the class $C_{i,j}$.

The maintenance of a specified index may entail the maintenance of many indices. A deletion of an index record (caused by an object deletion) corresponding to class C_l requires an adaptation of the index defined on class C_{l-1} and all its subclasses. As an example we consider the path *Per.owns.man.name* on which a multi-index is allocated and the figures 1 and 2[1]. Removing the object with *oid* Bus[i] means that the indexed attribute *man* of the class *Bus* with the value Company[j] (this is determined from the object Bus[i]) has to be accessed. Bus[i] has to be deleted from the index record. Then the indexed attribute *owns* on the class *Per* has to be accessed

[1]Recall the discussion about multi-index from section 2.2.

with value Bus[i]. This record has to be discarded from the index. So, in this case two index looks up are required since *Per* does not have subclasses. The cost involved in maintaining the index records due to a deletion of an object of class $C_{l,z}$ can be expressed in general by $CMT(h_{l,z}, nin_{l,z}, pm_{MX}) + \sum_{j=1}^{nc_{l-1}} CML(h_{l-1,j}, pm_{MX})$, in which $nin_{l,z}$ is the average number of values held by the attribute A_l of an object of class $C_{l,z}$.

An insertion of an index record (caused by an object insertion) concerns always one class. Suppose an object Bus[k] which refers to Company[j] has to be added. The consequence for the multi-index is that the indexed attribute *man* on the class *Bus* with value Company[j] has to be accessed and Bus[k] has to be added to the index record. The cost involved in maintaining the index records due to an insertion of an object to a class $C_{l,z}$ can be expressed by $CMT(h_{l,z}, nin_{l,z}, pm_{MX})$.

The maintenance cost of a multi-index record with regard to a class $C_{l,z}$ due to a deletion and insertion, $CM_{MX}(C_{l,z})$ can be expressed by:

$$CM_{MX}(C_{l,z}) = CMT(h_{l,z}, nin_{l,z}, pm_{MX}) + flag * \sum_{j=1}^{nc_{l-1}} CML(h_{l-1,j}, pm_{MX})$$

in which $flag = 1$ if an object is deleted and $flag = 0$ otherwise.

An algorithm for insertion and deletion of indices with regard to a multi-index organization can be found in [5].

MIX The use and the organization of an multi-inherited index does not differ much from the multi-index. If a class has an inheritance hierarchy then an inherited index is allocated on the class otherwise a simple index is allocated. The retrieving cost of a set of index records using a multi-inherited index with respect to a class $C_{l,z}$ against the ending attribute A_n can be expressed by:

$$CR_{MIX}(C_{l,z}) = \sum_{i=l}^{n-1} CRT(h_i, noid^*_{i+1,1}, pr_{MIX}) + CRL(h_n, pr_{MIX})$$

The cost involved in maintaining the index records due to a deletion of an object of class $C_{l,z}$ can be expressed by $CMT(h_l, nin_{l,z}, pr_{MIX}) + CML(h_{l-1}, pr_{MIX})$. The maintenance cost due to an insertion of an object to class $C_{l,z}$ can be expressed as $CMT(h_l, nin_{l,z}, pm_{MIX})$. So, the maintenance cost of an inherited multi-index with regard to a class $C_{l,z}$, $CM_{MIX}(C_{l,z})$ is

$$CM_{MIX}(C_{l,z}) = CMT(h_l, nin_{l,z}, pm_{MIX}) + flag * CML(h_{l-1}, pm_{MIX})$$

in which *flag* is defined as above.

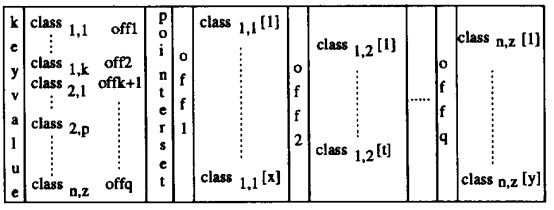

Figure 3: Format of a non-leaf primary index record

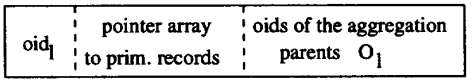

Figure 4: Format of a 3-tuple in a non-leaf auxiliary index record

NIX In this organization two kinds of indices can be distinguished, a primary and an auxiliary index. The primary index associates with each value v of the ending attribute A_n of a path P all the objects of the classes in the scope of P possessing this value. Therefore, the primary index is inverted with respect to the values of A_n and is used for queries. The auxiliary index associates with the *oid* of an object $O_{l,z}$ the list of *oids* of the (aggregation) parents of $O_{l,z}$. An auxiliary index aims to accelerate the maintenance of the primary index. Furthermore, it may accelerate the processing of some queries as illustrated in [7].

The architecture of a primary index record is given in Figure 3. The first field contains a value of the indexed attribute A_n (called key value) of a path $P = C_1.A_1.A_2...A_n$. In the second field all the classes of the scope of P are listed and the addresses of the classes in the primary index record (offset). When an index record occupies more than one page this can be useful otherwise it is redundant. If an attribute A_l is multi-valued the objects of the classes rooted at $C_{l,1}$ which hold the key value for A_n are stored as pairs (*oid*, numchild) in which *oid* identifies the object and numchild is the number of children which holds the key value for the nested attribute. If A_l is a single valued attribute it is sufficient to store the *oid* of the object which holds the key value for A_n since the value of numchild will be one. The value of numchild is used to determine whether an object should be deleted from the primary index or not. If the value of numchild is zero the object should be deleted.

Each class, except the first class and its subclasses (since they do not have parents), is connected with an auxiliary index record by the pointer set. An auxiliary index record consists of a sequence of 3-tuples. Figure 4 gives the format of a 3-tuple. For an object $O_{l,z}$ the 3-tuple contains the oid of the object in the first field. The pointer array connects the auxiliary record with the primary index records. The last field is a list of *oids* of the parents in a aggregation hierarchy of $O_{l,z}$. Note, for a class $C_{l,z}$ there are as many 3-tuples as the number of objects in the class. An example of a nested inherited index on the

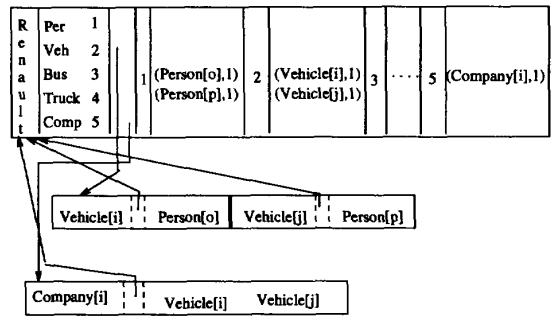

Figure 5: Example of a NIX on path P_e

path P_e = *Per.owns.man.name* with regard to the key value 'Renault' is given in Figure 5.

We assume that the average length of a primary index record is ln_{NIX} and the height of the index is h_{NIX}. Then the retrieval cost for a specific primary index record is expressed by $CRL(h_{NIX}, pr_{NIX})$. For an estimation of the size of ln_{NIX} we refer to [7].

An update of an object requires an adaptation of the primary index records containing the *oid* of the object as well as the auxiliary index records in which the object is involved. We consider the delete and insert operations. The following steps are involved in the *deletion* of an object $O_{l,z}$ according to an algorithm presented in [5].

1. The set of values SV of the attribute A_l for object $O_{l,z}$ is determined. If A_l is a single-valued attribute SV consists of a single value otherwise SV has $nin_{l,z}$ values[2].
2. The auxiliary index is accessed with the values of SV and the 3-tuples are modified by removing $O_{l,z}$ from the lists of parents in the 3-tuples.
   Then, the 3-tuple of $O_{l,z}$ is accessed (thus the auxiliary index record associated with class $C_{l,z}$). The parents of $O_{l,z}$ and $O_{l,z}$ it self are stored in a list called parentlist. The pointers to the primary index records are stored in S. Then the 3-tuple is removed.
3. As long as the parentlist is not empty the following actions are performed:
   (a) For each primary index record R whose address is in S do
      i. The primary index record R is accessed and the offsets of the *oid*s of the parentlist are determined.
      ii. For each *oid* $\in$ parentlist do
      If the *oid* identifies $O_{l,z}$ it is removed from R.
      If *oid* identifies a parent without having a field numchild then it is removed from R and inserted in an auxiliary list L together with the address of R.
      If an *oid* identifies a parent having a field numchild then the value of numchild is decremented. If numchild has reached the value zero, the pair (*oid*,numchild) is removed from R. If the object is not a member of the root of the path, it is inserted in the auxiliary list L together with the address R.
   (b) The auxiliary index is accessed with the *oid*s of L as search-keys and the list parentlist is emptied.
   (c) The 3-tuples of the *oid*s of L are accessed and the pointers of the primary records from which the *oid*s have been removed are deleted from the 3-tuples. Then the list of parents in the 3-tuples are determined and are concatenated in a new list parentlist. Then step 3 is executed again.

Since an object which has to be deleted entails an access to the object the cost of the first step can be neglected in estimating the cost to update an index due to a deletion.

The total number of 3-tuples that are accessed in step 2 is $nin_{l,z} + 1$. Therefore, the retrieval costs for these 3-tuples is $CRT(h_{AX}, nin_{l,z} + 1, 1)$, in which h_{AX} is the height of the B$^+$-tree associated with the auxiliary index. Because a 3-tuple in an auxiliary index will be much smaller than a page the third parameter of CRT will take value equal to 1. Furthermore, the number of 3-tuples at the leaf level of the auxiliary index is $\sum_{i=2}^{n} \sum_{j=1}^{nc_i} n_{i,j}$.

The number of auxiliary records involved with the $nin_{l,z}$ values, nar_{l+1} depends on the distribution of these values among the class C_{l+1} and its subclasses. In the case C_{l+1} does not have any subclasses it should be clear that $nar_{l+1} = 1$. Let us assume that the distribution of the $nin_{l,z}$ values over nc_{l+1} subclasses is $(nin_{l+1,1}, nin_{l+1,2}, ..., nin_{l+1,nc_{l+1}})$ such that $\sum_{i=1}^{nc_{l+1}} nin_{l+1,i} = nin_{l,z}$. Then, $nar_{l+1} = \sum_{i=1}^{nc_{l+1}} abs(nin_{l,i})$, in which $abs(x) = 1$ if $x > 0$ otherwise it is 0. These records and the auxiliary record corresponding to class $C_{l,z}$ should be written back after they are modified. Let ln_{AX} be the average length of an auxiliary record, p_{lax} be the number of leaf pages of the auxiliary index and pm_{AX} the average number of pages involved in rewriting an auxiliary record. Then, the cost involved in rewriting the modified auxiliary index records is

$$CRR(nar_{l+1} + 1) = \begin{cases} npa(nar_{l+1} + 1, n_{ax}, p_{lax}) & \text{if } ln_{AX} \leq p \\ (nar_{l+1} + 1)pm_{AX} & \text{otherwise} \end{cases}$$

in which $n_{ax} = \sum_{j=2}^{n} nc_j$.

So, the cost of step 2 can be expressed by $CSD2 = CRT(h_{AX}, nin_{l,z} + 1, 1) + CRR(nar_{l+1} + 1)$.

Let $\overline{nin_{l,z}}$ denote the average number of values held in the nested attribute A_n of an object of the class $C_{l,z}$. Then the cost involved in fetching the primary records

[2]Note, the $nin_{l,z}$ values are *oid*s belonging to the objects in the hierarchy rooted at class C_{l+1}.

can be expressed by $CRT(h_{NIX}, \overline{nin_{l,z}}, prd_{NIX})$, in which prd_{NIX} is the average number of relevant pages which should be retrieved. According action (a)ii these pages are modified. So, these pages should be rewritten which implies that pmd_{NIX} is equal to prd_{NIX}. Therefore, the cost of action 3(a) is $CS3a = CMT(h_{NIX}, \overline{nin_{l,z}}, pmd_{NIX})$.

The number of parents *oid* in list L for object $O_{l,z}$ is equal to $par_{l,z} = \sum_{i=1}^{nc_{l-1}} k_{l-1,i}$ and the number of parents for the $par_{l,z}$ objects can be expressed by $par_{l-1,z} = par_{l,z} \sum_{i=1}^{nc_{l-2}} k_{l-2,i}$ and so on. Assuming that the distribution of $par_{i,j}$ *oids* over class C_{i-1} and its subclasses is $(par_{i-1,1}, par_{i-1,2} ... par_{i-1,nc_{i-1}})$, the number of auxiliary records involved with $par_{i,j}$ is given by $narp_{i-1} = \sum_{j=1}^{nc_{i-1}} abs(par_{i-1,j})$. Therefore, the cost involved with maintaining the auxiliary records in steps 3b and 3c can be expressed by:

$$CU3bc = \sum_{i=2}^{l-1} CRR(narp_i)$$

The cost to retrieve the $\sum_{i=2}^{l} par_{i,z}$ *oids* can be done in two ways. First, the leaf nodes of the auxiliary index can be scanned for retrieving the values since the remaining nodes are in main memory already. The cost involved with this retrieval action is $SA_1 = npa(\sum_{i=2}^{l} par_{i,z}, \sum_{i=1}^{n} \sum_{j=1}^{nc_i} n_{i,j}, p_{lax})$

A second technique to find the *oids* is via the primary record since the address of the auxiliary record of each class is stored in the primary record. Then, the *oids* have to be retrieved from the auxiliary records. The cost involved in retrieving the auxiliary records is

$$SA_2 = \begin{cases} npa(\sum_{i=2}^{l-1} narp_i, n_{ax}, p_{lax}) & \text{if } ln_{AX} \leq p \\ \lceil \frac{ln_{AX}}{p} \rceil \sum_{i=2}^{l-1} narp_i & \text{otherwise} \end{cases}$$

The total cost of step 3 is $CSD3 = CS3a + CU3bc + min(SA_1, SA_2)$.

Therefore the maintenance cost due to a deletion can be expressed as $CSD2 + CSD3$.

An *insertion* of an object $O_{l,z}$ entails the following steps to maintain the index in a NIX organization.

1. The set of values SV of the attribute A_l for object $O_{l,z}$ is determined. If A_l is a single-valued attribute SV consists of a single value otherwise SV has $nin_{l,z}$ values.
2. The auxiliary index is accessed with the values of SV and the 3-tuples are modified by inserting $O_{l,z}$ in the lists of parents and the set of pointers S to the primary records are determined.
3. For each primary record R whose address is in S do
   (a) The primary record R is accessed, the offsets and the address of the auxiliary record corresponding to class $C_{l,z}$ is determined.
   (b) The *oid* of $O_{l,z}$ is inserted in the list of *oids*; if the list consists pairs, the value of the field numchild is initialized adequately.
4. A new 3-tuple with the *oid* of $O_{l,z}$ is inserted in the auxiliary index record corresponding to class $C_{l,z}$.

The first two steps are quite similar with the corresponding steps in the case of a deletion. The only difference is that we do not access the auxiliary record corresponding to class $C_{l,z}$ now. Therefore, the cost of step 2 is $CSI2 = CRT(h_{AX}, nin_{l,z}, 1) + CRR(nar_{l+1})$. The cost of step 3 is $CSI3 = CMT(h_{NIX}, \overline{nin_{l,z}}, pmi_{NIX})$. Note, that the average number of pages which should be retrieved and rewritten to maintain a primary index due to an insertion differs from the number of pages due to a deletion [7].

The last step entails the modification of the auxiliary record associated with $C_{l,z}$. So, the number of auxiliary record which should be modified becomes $nar_{l+1} + 1$. Taking this step together with step 2 results into a cost $CSI24 = CRT(h_{AX}, nin_{l,z}, 1) + CRR(nar_{l+1} + 1)$. Therefore, the maintenance cost due to an insertion can be given by $CSI24 + CSI3$. So, $CM_{NIX}(C_{l,z})$ may be expressed by

$$CM_{NIX}(C_{l,z}) = \begin{cases} CSD2 + CSD3 & \text{in the case of a deletion} \\ CSI24 + CSI3 & \text{in the case of an insertion} \end{cases}$$

3.2 Workload model

Each operation implies the processing of a path and several operations may result in the same path. The load on a path is distributed over the involved classes; that is the frequency of queries against the ending attribute with respect to a class, the frequencies of insertions and deletions on each class. In practice database administrators may predict the distribution very well. In the following, the load distribution on a path $P = C_1.A_1.A_2...A_n$ is denoted by $LD_{A_n}(scope(P)) = \{(\alpha_{1,1}, \beta_{1,1}, \gamma_{1,1}), ..., (\alpha_{1,nc_1}, \beta_{1,nc_1}, \gamma_{1,nc_1}), ..., (\alpha_{n,1}, \beta_{n,1}, \gamma_{n,1}), ..., (\alpha_{n,nc_n}, \beta_{n,nc_n}, \gamma_{n,nc_n})\}$. Each triplet $(\alpha_{l,z}, \beta_{l,z}, \gamma_{l,z})$, $1 \leq l \leq n$, $1 \leq z \leq nc_l$ represents the frequency of queries against the ending attribute A_n with respect to $C_{l,z}$, the frequency of insertions and deletions on $C_{l,z}$ respectively.

The workload on a subpath $S_k = C_k.A_k.A_{k+1}...A_m$ of a path P depends on whether the starting class of the subpath is equal to the starting class of P or not. In the case that the starting class of S_k is equal to the starting class of its superpath P the load distribution against A_m will remain the same as against A_n for the classes involved in $scope(S_k)$.

If the starting class of S_k is not equal to the starting class of its superpath the load on the subpath becomes $LD_{A_m}(scope(S_k)) = \{(\alpha_{k,1} + \sum_{i=1}^{k-1} \sum_{j=1}^{nc_i} \alpha_{i,j}, \beta_{k,1}, \gamma_{k,1}), (\alpha_{k,2}, \beta_{k,2}, \gamma_{k,2}), ... (\alpha_{m,nc_m}, \beta_{m,nc_m}, \gamma_{m,nc_m})\}$ since the processing of queries with regard to a class $\in scope(C_1.A_1...A_{k-1})$ against A_n entails a processing of S_k as well.

It should be noticed that we consider only queries against the ending attribute A_n in this workload model.

4 Index configurations

Since each indexing technique has its own advantages and disadvantages it may be sensible to split a path into subpaths and to make a trade-off between the available indexing techniques, on basis of the operations defined on each subpath, in indexing each subpath. This brings us to the subject of index configuration. Some examples of index configurations are followed by the formal definition of index configuration.

Definition 4.1 Given path $P = C_1.A_1.A_2...A_n$ ($n \geq 1$), an index configuration for P of degree m, $IC_m(P)$ is defined as a sequence of pairs $\{T_1, T_2, ..., T_m\}$ $m \leq n$. A pair T_i has the form (S_i, X_i) in which

- $S_i = C_j.A_j.A_{j+1}.....A_{j+l_i}$ ($j \geq i$, $l_i \geq 0$) is a subpath with length $l_i + 1$. The class $C_j \in \textit{class}(P)$ is the starting class of the subpath and the attribute A_{j+l_i} is the ending attribute of the subpath.
- X_i is the index allocated to subpath S_i. □

Ex 4.1 Consider $P_e = \textit{Per.owns.man.name}$. Examples of index configurations of degrees 1, 2 and 3 are $IC_1(P_e) = \{(\textit{Per.owns.man.name}, \text{MIX})\}$, $IC_2(P_e) = \{(\textit{Per.owns}, \text{SIX}), (\textit{Veh.man.name}, \text{NIX})\}$ respectively $IC_3(P_e) = \{(\textit{Per.owns}, \text{SIX}), (\textit{Veh.div}, \text{IIX}), (\textit{Comp.name}, \text{SIX})\}$. □

The processing cost of a query using indices is determined by the retrieval costs of the index records required for locating the objects on disk which satisfy to the query and the costs to retrieve these objects. In the following we concentrate on the retrieval cost of the indices required for locating the objects on disk. We call this the *searching* cost.

The following proposition determines the searching costs involved in processing queries with regard to the starting class of a path P having an index configuration of degree $m > 1$. It says that the searching cost of a query against the ending attribute with regard to the starting class of P is equal to the sum of the searching cost on each subpath. The proof of the proposition and its corollary can be found in [7].

Proposition 4.1 Let $P = C_1.A_1.A_2...A_n$, ($n \geq 1$), be a path with configuration $IC_m(P) = \{(S_1, X_1), (S_2, X_2), ..., (S_m, X_m)\}$, $1 < m \leq n$, $X_i \in \{MX, MIX, NIX\}$ and $C(i)$ is the starting class of S_i. Then, the processing cost of a query with respect to class $C(1)$, $CR_{IC_m(P)}(C(1)) = \sum_{i=1}^{m} CR_{X_i}(C(i))$, in which $CR_{X_i}(C(i))$ represents the searching cost on subpath S_i of the objects of class $C(i)$ which satisfy to the predicate against the ending attribute A_n

Corollary The processing cost of a query with respect to a class $C_{l,z}$ belonging to a subpath S_g such that $(S_g, X_g) \in IC_m(P)$ and $m > 1$ can be expressed as $CR_{IC_m(P)}(C_{l,z}) = CR_{X_g}(C_{l,z}) + \sum_{i=g+1}^{m} CR_{X_i}(C(i))$.

The maintenance cost of an index record, due to a deletion, allocated on a class depends on the indices allocated on the previous classes as has been shown in the previous section. We study the consequence of this dependency in the context of index configurations.

Consider 2 subpaths S_i and S_j of a path P such that $S_i = C_k.A_k.A_{k+1}...A_t$ and $S_j = C_{t+1}.A_{t+1}.A_{t+2}...A_w$ on which the indices $X_i, X_j \in \{MX, MIX, NIX\}$ are allocated respectively.

Then, independently of X_i the attribute A_t will be inverted. Since the domain of A_t are the *oid*s of the objects belonging to class $C^*_{t+1,1}$ these *oid*s will be the key values in the index record. So, index records will have the form $\{(o^{nr}_{t+1,x}, ...), (o^{nrl}_{t+1,x}, ...), ...\}$, in which $o^{nr}_{t+1,x}$ and $o^{nrl}_{t+1,x}$ are *oids* of objects in class $C_{t+1,x}$.

A deletion of an object of class $C_{t+1,q}$, let say $o^{nrl}_{t+1,q}$ implies that the index on S_j should be updated. Section 3 is devoted how this can be done for the several index organizations. Furthermore, the index on S_i should be updated also. This means that the record with the key value $o^{nrl}_{t+1,q}$ should be deleted. The cost of this action depends on X_i and may be expressed by:[3]

$$CMD_{X_i}(A_t) =$$

$$\begin{cases} \sum_{i=1}^{nc_t} CML(h^{SIX}_{t,i}, \lceil \frac{ln^{SIX}_{t,i}}{p} \rceil) & \text{if } X_i = MX \\ CML(h^{IIX}_t, \lceil \frac{ln^{IIX}_t}{p} \rceil) & \text{if } X_i = MIX \\ CML(h^{NIX}_t, \lceil \frac{ln^{NIX}_t}{p} \rceil) + \textit{delpoint} & \text{if } X_i = NIX \end{cases}$$

in which *delpoint* is defined as $\textit{delpoint} = 2 * npa(\sum_{l=k+1}^{t} \sum_{z=1}^{nc_l} \overline{nin^t_{l,z}}, \sum_{l=k+1}^{t} \sum_{z=1}^{nc_l} n_{l,z}, p_{lax})$, p_{lax} is the number of leaf pages of the auxiliary index in a NIX organization and $\overline{nin^t_{l,z}}$ is the average number of values held in the nested attribute A_t of an object of class $C_{l,z}$.

Note, *delpoint* express the cost to delete the pointers in the auxiliary index records which point to the primary record with key value $o^{nrl}_{t+1,q}$ in the case $X_i = NIX$.

In definition 4.2 (see below) we assign the cost $CMD_{X_i}(A_t)$ to the maintenance cost of the index on A_t and not to the maintenance cost of the index on A_{t+1}. In this way we can estimate the cost due to maintenance on a subpath independently of each other. This is justified from the fact that a deletion of an object belonging to the starting class of a subpath S_j will lead to the deletion of exactly *one* index record in the index in which the ending attribute of the subpath which precedes S_j is involved.

As has been shown in the previous section the insertion of an index record in an index is independent of other indices. This is the case for all index types.

[3] In below mentioned formula the second parameter of CML takes as value the number of pages that an index record occupies since all these pages should be deleted.

The following definition computes the maintenance cost of an index in an index configuration with degree $m > 1$.

Definition 4.2 Let $P = C_1.A_1.A_2...A_n$, $(n \geq 1)$, be a path with a configuration $IC_m(P) = \{(S_1, X_1), (S_2, X_2), ..., (S_m, X_m)\}$, $1 < m \leq n$, $X_i \in \{MX, MIX, NIX\}$. The maintenance cost, due to insertions and deletions on a class $C_{l,x}$, of an index X_i allocated on a subpath S_i having A_t as ending attribute is given by:

$$CM'_{X_i}(C_{l,x}) = \begin{cases} CM_{X_i}(C_{l,x}) + flag * CMD_{X_i}(A_t) & \text{if } A_l = A_t \neq A_n \\ CM_{X_i}(C_{l,x}) & \text{otherwise} \end{cases}$$

in which $flag = 1$ if an object is deleted and $flag = 0$ if an object is inserted.

The processing cost of a (sub)path is defined as the sum of the cost to maintain the indices on the (sub)path and the searching costs on the subpath of those objects which satisfy to the queries. Proposition 4.2 determines the processing cost of a path P with has an index configuration of degree $m > 1$. The proof can be found in [7].

Proposition 4.2 The processing cost of a path $P = C_1.A_1.A_2...A_n$ with a configuration $IC_m(P) = \{(S_1, X_1), (S_2, X_2), ..., (S_m, X_m)\}$, $1 < m \leq n$, $X_i \in \{MX, MIX, NIX\}$, is the sum of the processing cost of each subpath.

5 The selection of index configurations

The algorithm takes as input a path $P = C_1.A_1.A_2...A_n$ with the load distribution on the classes. The body of the algorithm mainly consists of 3 procedures, *Cost_Matrix*, *Min_Cost*, and *Opt_Ind_Con*. Each of these procedures will be discussed in sequence. The output of the algorithm will be the optimal index configuration for the path P and the cost to process the workload on the path.

The algorithm starts with *Cost_Matrix* which computes the processing costs for all possible subpaths with each index organization and represents it in a matrix. The size of the matrix depends on the length of the path for which an index configuration should be determined. A path with length n can be split into $\frac{1}{2}n*(n+1)$ subpaths. These paths will be numbered as $S_1, S_2, ..., S_{\frac{1}{2}n*(n+1)}$, which determine the rows in the matrix. The factor $\frac{1}{2}n*(n+1)$ follows from the fact that there are n subpaths of length 1, $n-1$ paths of length 2, $n-2$ paths of length 3 and so on. The number of columns is determined by the number of index types which will be considered. For the moment we consider the three index organizations MX, MIX and NIX. So, the size of the matrix will be 3 to $\frac{1}{2}n*(n+1)$. The value of an element $a_{i,j}$ in the matrix represents the costs in processing a subpath S_i with index X_j. A hypothetical matrix is given in Figure 6 for a path $P_{ex} = C_1.A_1.A_2.A_3.A_4$. Note, in the case a path has length one and it does not have subclasses the organizations for MX, MIX and NIX are almost equivalent.

	MX	MIX	NIX
$C_1.A_1$	3	4	6
$C_2.A_2$	4	4	4
$C_3.A_3$	2	3	4
$C_4.A_4$	4	5	5
$C_1.A_1.A_2$	7	6	10
$C_2.A_2.A_3$	6	5	10
$C_3.A_3.A_4$	7	14	6
$C_1.A_1.A_2.A_3$	9	8	18
$C_2.A_2.A_3.A_4$	11	12	5
$C_1.A_1.A_2.A_3.A_4$	14	13	9

Figure 6: Possible subpaths of path P_{ex} with their processing costs using an index $X_i \in \{MX, MIX, NIX\}$.

Therefore, the processing costs will be almost same. The cost of a MX organization for a path of length n is the sum of the cost of a SIX organization for each class separately.

Then, the procedure *Min_Cost* determines the minimal costs in each row which indicates the best indexing technique for a subpath. In Figure 6 the minimal costs in each row are underlined.

Then, the procedure *Opt_Ind_Con* determines the optimal index configuration for a path. The idea of this procedure is based on the consideration of all possible ways to recombine the original path from the subpaths; that is to choose a number of subpaths such that the concatenation of the subpaths will result into the original path and each class belongs to exactly one subpath. The processing cost of the original path will be the sum of the underlined costs of each subpath. A possible way to recombine P_{ex} is by concatenating subpath $C_1.A_1.A_2$ and $C_3.A_3.A_4$ allocated by an MIX respectively a NIX. The processing cost of path P_{ex} will be 12 with this configuration. It is clear that we may compute the processing cost of all possible recombinations and then choose the recombination which holds the minimal costs. However this method will only be feasible when the length of a path for which an index configuration should be determined will be small, since the number of possible recombinations for a path with length n is 2^{n-1}. This can be verified easily. Starting with the fixed subpath $C_1.A_1...A_{n-1}$ there is only one way to get the original path with length n that is by concatenating it with the subpath $C_n.A_n$. Starting with the subpath $C_1.A_1.A_2...A_{n-2}$ there are 2 possible ways to obtain the original path, the concatenation with the subpath $C_{n-1}.A_{n-1}.A_n$ and with the sequence of subpaths $C_{n-1}.A_{n-1}$, $C_n.A_n$. The number of recombinations starting with a fixed path $C_1.A_1.A_2...A_k$ is given by $2^{n-(k+1)}$, since these are the number of ways to order the $n-k$ classes.

Theoretically the complexity involved in determining the optimal index configuration in this way will be $O(2^{n-1})$. Because in practice a path has rarely a length greater than 7 the complexity is determined by the expression $3 * O(\frac{1}{2}n*(n+1))$ which is the size of the matrix.

Although the complexity of the algorithm seems not

to be a problem we still want to implement the algorithm as efficient as possible. We apply the idea of branch and bound in the recombination process. Despite the fact that branch and bound does not guarantee that it will be able to reduce the complexity in all cases it has proved to be useful in practice in many fields.

To explain the procedure *Opt_Ind_Con* we will distinguish subpaths by two subscripts (until now one subscript was sufficient), the first subscript indicates the starting class of the subpath the second subscript indicates the ending attribute of a subpath.

The procedure *Opt_Ind_Con* starts by considering the index configuration $IC_1(P)$. Note, there is just one index configuration with degree 1. We read the minimal processing cost (PC) from the *Cost_Matrix* for $IC_1(P)$. This cost will be assigned to the variable PC_{min}.

Then, P will be split into two subpaths $S_{1,n-1} = C_1.A_1.A_2...A_{n-1}$ and $S_{n,n} = C_n.A_n$. We read the minimal processing cost of $S_{1,n-1}$ ($PC_{S_{1,n-1}}$) from the *Cost_Matrix*. If $PC_{S_{1,n-1}} \geq PC_{min}$ then the index configuration including $S_{1,n-1}$ will be not considered any longer since the processing cost of this index configuration will be higher than the processing cost of $IC_1(P)$. Otherwise, we compute the processing cost of the candidate configuration $\{(S_{1,n-1}, X_{1,n-1}), (S_{n,n}, X_{n,n})\}$, in which $X_{i,j}$ is the best index organization for $S_{i,j}$, by adding $PC_{S_{1,n-1}}$ and $PC_{S_{n,n}}$ resulting into PC_{can}. If $PC_{can} < PC_{min}$ then PC_{min} will get the value of PC_{can} and the belonging index configuration is the best one until now, denoted as IC^{best}.

Then, the path will be split into $S_{1,n-2} = C_1.A_1.A_2...A_{n-2}$ and $S_{n-1,n} = C_{n-1}.A_{n-1}.A_n$. If $PC_{S_{1,n-2}} \geq PC_{min}$ then the index configurations including $S_{1,n-2}$ will be not investigated further. Otherwise, the processing cost of the configuration $\{(S_{1,n-2}, X_{1,n-2}), (S_{n-1,n}, X_{n-1,n})\}$ will be computed and in the case this cost is lower than PC_{min}, PC_{min} will get this value and this configuration will be the best one until now.

Then subpath $S_{n-1,n}$ will be investigated in the same manner. Subpath $S_{n-1,n}$ will be split into the subpaths $S_{n-1,n-1} = C_{n-1}.A_{n-1}$ and $S_{n,n} = C_n.A_n$. If $PC(S_{1,n-2}) + PC(S_{n-1,n-1}) \geq PC_{min}$ then index configurations containing the subpaths $S_{1,n-2}$ and $S_{n-1,n}$ will not be considered further. Otherwise, the processing cost of the configuration consisting of the subpaths $S_{1,n-2}$, $S_{n-1,n-1}$ and $S_{n,n}$ will be investigated for becoming the best one.

This process will be repeated until we are not able to split the path P into exactly two subpaths. Note, the last two subpaths in which P will be split by the algorithm are $S_{1,1} = C_1.A_1$ and $S_{2,n} = C_2.A_2.A_3...A_n$. The same process will be applied on each subpath of a subpath.

Applying this procedure on P_{ex} with the *Cost_Matrix* of Figure 6 results in the following steps. We start with the index configuration $\{P, NIX\}$ with processing cost 9. Then $PC_{min} = 9$ and $IC^{best} = \{P, NIX\}$. Then the path will be split into $S_{1,3} = C_1.A_1.A_2.A_3$ and $S_{4,4} = C_4.A_4$. Since $PC(S_{1,3}) = 8 < 9$ we compute the processing cost of the configuration $\{(S_{1,3}, MIX), (S_{4,4}, MX)\}$. This results into 8+4=12. Thus the configuration $\{P, NIX\}$ is still the best one.

Then P is split into $S_{1,2} = C_1.A_1.A_2$ and $S_{3,4} = C_3.A_3.A_4$. Since $PC(S_{1,2}) = 6 < 9$ the subpath $S_{3,4}$ is eligible for further investigation. First we compute the processing cost of the configuration $\{(S_{1,2}, MIX), (S_{3,4}, NIX)\}$, which results into $6+6 = 12$. Since the processing cost of the configuration $\{P, NIX\}$ is smaller this is still the best one. Second, subpath $S_{3,4}$ is split into $S_{3,3} = C_3.A_3$ and $S_{4,4} = C_4.A_4$. Since, $PC(S_{1,2}) + PC(S_{3,3}) = 6 + 2 = 8 < 9$ we compute the cost of the configuration containing $S_{1,2}$, $S_{3,3}$ and $S_{4,4}$ resulting $6 + 2 + 4 = 12$. Thus configuration $\{P, NIX\}$ is still the best one.

Finally P is split into $S_{1,1} = C_1.A_1$ and $S_{2,4} = C_2.A_2.A_3.A_4$. Since $PC(S_{1,1}) = 3 < 9$ we consider the configurations including $S_{1,1}$. First, we compute the processing cost of the configuration consisting of $S_{1,1}$ and $S_{2,4}$. Since, the cost of this configuration is $3 + 5 = 8$ this will become the best configuration and $PC_{min} = 8$. Then $S_{2,4}$ is split into $S_{2,3} = C_2.A_2.A_3$ and $S_{4,4} = C_4.A_4$. Since, $PC(S_{1,1}) + PC(S_{2,3}) = 3 + 5 = 8$ is equal to PC_{min} the index configurations containing $S_{1,1}$ and $S_{2,3}$ will not be considered further. Then $S_{2,4}$ is split into $S_{2,2} = C_2.A_2$ and $S_{3,4} = C_3.A_3.A_4$. Since $PC(S_{1,1}) + PC(S_{2,2}) = 3 + 4 < 8$ we compute first the processing cost of the configuration consisting of the subpaths $S_{1,1}$ $S_{2,2}$ and $S_{3,4}$. Since, this results into $3 + 4 + 6 = 13$, this is worse than the best index configuration found until now. Then $S_{3,4}$ is split into $S_{3,3} = C_3.A_3$ and $S_{4,4} = C_4.A_4$ Since $PC(S_{1,1}) + PC(S_{2,2}) + PC(S_{3,3}) = 9 > 8$ the configuration containing $S_{1,1}$, $S_{2,2}$ and $S_{3,3}$ is not a candidate for the optimal configuration.

Thus the optimal configuration for P_{ex} results $\{(C_1.A_1, MX), (C_2.A_2.A_3.A_4, NIX)\}$ with processing cost 8.

The following experiment presented as Example 5.1 supports the idea of splitting a path into subpaths and allocating the best index on each subpath.

Ex 5.1 Consider P_{exa} = *Per.owns.man.divs.name* and the database schema of Figure 1. The chosen database characteristics with regard to the schema of Figure 1 are given by Figure 7. We compute the processing cost of all possible subpaths of P_{exa} according to the formulae presented in section 3 and 4. In the case a subpath has length one and the subpath does not have subclasses we organize the NIX and the IIX on this class such that they are equivalent with the SIX organization on the class. In the case a subpath with length one has subclasses the NIX is organized such that it is equivalent with the IIX on the class. The procedure *Min_Cost* underlines the minimal cost in each row. The results are presented in Figure 8.

Procedure *Opt_Ind_Con* results into the optimal configuration $\{(Per.owns.man, NIX),$ $(Comp.div.name, MX)\}$ with a processing cost 16.03.

Class	$n_{i,j}$	$d_{i,j}$	$nin_{i,j}$	$LD_{name}(P_{exa})$
Per	200.000	20.000	1	(0.3, 0.1, 0.1)
Veh	10.000	5.000	3	(0.3, 0.0, 0.05)
Bus	5.000	2.500	2	(0.05, 0.05, 0.1)
Truck	5.000	2.500	2	(0.0, 0.1, 0.0)
Comp	1.000	1.000	4	(0.1, 0.1, 0.1)
Div	1.000	1.000	1	(0.2, 0.2, 0.1)

Figure 7: Database and workload characteristics belonging to the logical schema of Figure 1

	MX	*MIX*	*NIX*
Per.owns	34.95	34.95	34.95
Veh.man	10.79	4.40	4.40
Comp.div	2.58	2.58	2.58
Div.name	2.50	2.50	2.50
Per.owns.man	45.75	39.35	10.95
Veh.man.div	13.68	7.12	13.68
Comp.div.name	5.08	5.08	5.09
Per.owns.man.div	48.31	41.95	26.30
Veh.man.div.name	15.86	9.50	13.76
Per.owns.man.div.name	50.81	44.46	42.84

Figure 8: Cost matrix belonging to the workload and database characteristics of Figure 7.

The procedure found the optimal configuration by exploring 4 index configurations instead of exploring all the 8 index configurations. □

Two major conclusions can be drawn from the experiment described in example 5.1. First, the idea of optimal index configuration decreases the processing cost of a path by a factor 2.7. Without index configurations the whole path would be indexed by one index type. In this case a NIX would be allocated on P_{exa} requiring a processing cost of 42.84. Second, the technique of branch and bound reduced the number of evaluations of index configuration considerably in this experiment.

6 Conclusions and further research

The main concepts of object-oriented data models are nested objects and inheritance. To be viable, these models should be supported by an architecture that directly implements the concepts in an efficient way. Several indexing techniques are available for this purpose. Since a database operation leads to the processing of a path, such a path is often indexed to accelerate the processing. As it has been shown that indexing a whole path by a single index may become inefficient. Therefore, we propose an algorithm which is able to split a path in subpaths and allocating an index on each subpath such that the processing of the path can be done with minimal cost in terms of page accesses. For the moment we consider five indexing techniques. As has been shown the five indexing techniques could be reduced to three, multi-index, multi-inherited index and nested inherited index.

The presented algorithm is able to select the *optimal* index configuration for a given path with a *feasible* complexity in practice. Since, the foundations of optimization techniques for object-oriented databases is still in its infancy there is a substantial need for such an algorithm and research activities in this direction. Despite the fact that foundations of optimization techniques are closely related to database languages there has been a lot of research devoted to database languages while no comparable amount of research has been reported concerning foundations of optimization techniques.

A topic for further research is the extension of the algorithm such that it may generate index configurations for n paths. In this case we have to investigate how to deal with related paths. Note a path may be a subpath of another path or paths may overlap each other. Furthermore, we will incorporate in the algorithm the possibility that no index will be allocated on a subpath. The incorporation of path and nested indices [6, 2] can be done straightforward since we may verify easily that the maintenance and retrieval costs on a subpath indexed by these types can be estimated independently of other subpaths.

References

[1] Bertino, E., Kim, W., Indexing Techniques for Queries on Nested Objects, in IEEE TKDE, June 1989, pp. 226-244.

[2] Bertino, E., Optimization of Queries using Nested Indices, in Proc. EDBT 1990, pp. 44-59.

[3] Bertino, E., On Index Configurations in Object-Oriented Databases, submitted for publication in Int. J. VLDB.

[4] Bertino, E., A Survey of Indexing Techniques for Object-Oriented Databases, to appear in Query processing for Advanced Database applications, Freytag, J., Vossen, G., Maier, D., (eds), Morgan Kaufman (1993).

[5] Bertino, E., Foscoli, P., Index Organizations for Object-Oriented Database Systems, to appear in IEEE TKDE.

[6] Bertino, E., Guglielmina, C., Optimization of Object-Oriented Queries using Path Indices, Proc. RIDE TQP 92, pp. 140-149.

[7] Choenni, R., Bertino, E., Blanken, H.M., Chang, S.C., On the Selection of Optimal Index Configuration in OO Databases (extended version of this paper), to appear as tecnical report.

[8] Finkelstein, S., Schkolnick, M., Tiberio, P., Physical Database Design for Relational Databases, in ACM TODS, Vol.13, No. 1, pp. 91-128 (1988).

[9] Kato, K., Masuda, T., Persistent Caching: An Implementation Technique for Complex Objects with Object Identity, in IEEE TSE, Vol 18, No. 7, pp. 631-645 (1992).

[10] Kim, W., Kim K.C., Dale, A., Indexing Techniques for Object-Oriented Databases, in Object-Oriented Concepts, Databases and Applications, Kim, W., Lochovsky, F., (eds), Addison-Wesley, pp. 371-394 (1989).

[11] Valduriez, P., Join Indices, in ACM TODS, Vol. 12, No. 2, pp. 218-246 (1987).

[12] Yao, S.B., Approximating Block Accesses in Database Organizations, in Comm. of ACM, Vol. 20, No. 4, pp. 260-261 (1977).