VLDB Journal, 3, 355-399 (1994), Kyu-Young Whang, Editor

Index Configuration in Object-Oriented Databases

Elisa Bertino

Received December, 1992; revised version received August, 1993; accepted December, 1993.

Abstract. In relational databases, an attribute of a relation can have only a single primitive value, making it cumbersome to model complex objects. The object-oriented paradigm removes this difficulty by introducing the notion of nested objects, which allows the value of an object attribute to be another object or a set of other objects. This means that a class consists of a set of attributes, and the values of the attributes are objects that belong to other classes; that is, the definition of a class forms a hierarchy of classes. All attributes of the nested classes are nested attributes of the root of the hierarchy. A branch of such hierarchy is called a *path*. In this article, we address the problem of index configuration for a given path. We first summarize some basic concepts, and introduce the concept of index configuration for a path. Then we present cost formulas to evaluate the costs of the various configurations. Finally, we present the algorithm that determines the optimal configuration, and show its correctness.

Key Words. Index selection, physical database design, query optimization.

1. Introduction

The growing need for data management facilities to handle objects more complex than tuples of relations (e.g., CAD/DAM, software engineering, and office automation) has resulted in the development of object-oriented database systems (OODBMSs; Skarra et al., 1986; Banerjee et al., 1987; Fishman et al., 1987; Andrews and Harris, 1987; Bjornerstedt and Hulten, 1989; Breitl et al., 1989; Deux et al., 1990). Because of the increased complexity of the data model to be supported, OODBMSs have had to address new issues and requirements in the design and analysis of suitable access mechanisms. To be viable, the object-oriented approach to data management must be supported by an architecture that directly implements the basic concepts of the object-oriented paradigm.

Elisa Bertino, Sc.Dr., is Full Professor of Computer Science at the Dipartimento di Scienze dell'Informazione, Universita' degli Studi di Milano, Via Comelico 39, 20135 Milano, Italy.

This paradigm is based on a number of fundamental concepts (Bertino and Martino, 1993; Kim, 1990). Any real-world entity is represented by only one data modeling concept: the object. Each object is identified by a *unique identifier* (UID). The state of each object is defined at any point in time by the value of its *attributes* (also called instance variables). The attributes can have as value both *primitive* (or atomic) objects (e.g., strings, integers, or booleans) and *non-primitive* objects, which, in turn, consist of a set of attributes. (Note that when the value of an attribute A of an object is a non-primitive object O, the UID of O is stored in A.)

Objects with similar attributes and behavior are grouped into classes. A class specifies a set of attributes that define object structure, and a set of methods that define object behavior. An attribute definition consists of a name and a domain. The domain can be any class, including a primitive class. The fact that a class C' is the domain of an attribute of a class C establishes an association, most often called an *aggregation relationship*, between C and C'. Since C' in turn has aggregation relationships with the class domains of its attributes and so on, the definition of class C results in a directed graph of classes rooted at C, called an *aggregation hierarchy*. We refer to attributes of class C' in the aggregation hierarchy as *nested attributes* of C. An example of an aggregation hierarchy is shown in Figure 1. In the figure, an arc connects an attribute A of class C to class C' if C' is the domain of A. The * denotes a multivalued attribute. Furthermore, classes are organized into inheritance hierarchies. A *subclass* inherits attributes and methods from its *superclass* and, in addition to these, may have specific attributes and methods.

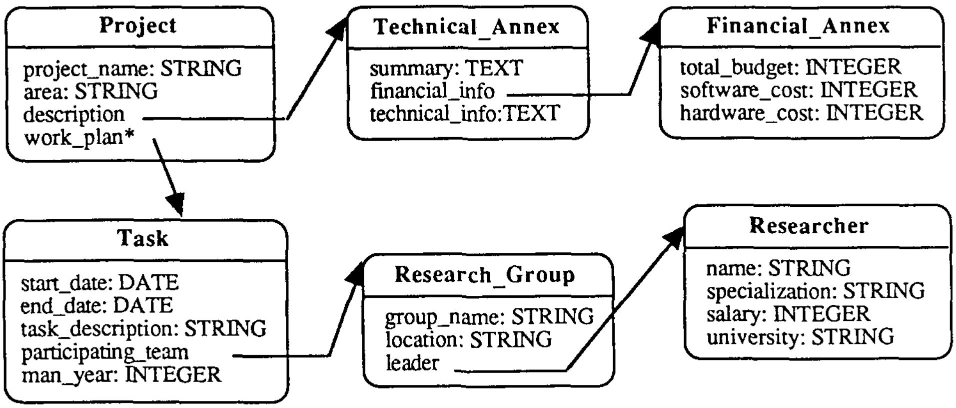
Object-oriented programming languages imply navigational access to objects. However, this capability alone is not adequate for applications that must deal with a large number of objects. Therefore, advanced OODBMSs provide associative query capabilities (Banerjee et al., 1988) in addition to navigational access to objects. Because of the nested object structures, object-oriented query languages such as the one described by Banerjee et al. (1988) allow restrictions on objects based on predicates on both nested and non-nested attributes of classes. The following is an example of a query against the aggregation hierarchy of Figure 1:

```
Retrieve all projects in the database area with a total
                  budget higher than $50,000.
```

In this query there is a predicate against the attribute "area" and a predicate against the nested attribute "total_budget" of class Project. We refer to a predicate defined on a nested attribute as a *nested predicate*. To support query predicates on class-nested attributes, object-oriented query languages (Bertino et al., 1992) usually provide some forms of *path-expression*. A path-expression specifies an implicit join between an object O and an object referenced by O.[1] Therefore, in object-oriented query languages it is useful to distinguish between the *implicit join*, deriving from

1. An object O references an object O', if O contains the UID of O' as value of some of its attributes.

Figure 1. Aggregation hierarchy example



the hierarchical nesting of objects, and the *explicit join*, similar to the relational join where two objects are explicitly compared on the values of their attributes. Note that some query languages only support implicit joins, based on the argument that in relational systems joins are mostly used to recompose entities that were decomposed for normalization (Breitl et al., 1989), and to support relationships among entities. In object-oriented data models there is no need to normalize objects, because these models directly support complex objects. Moreover, relationships among entities are supported through object references; thus, the same function of joins used in the relational model to support relationships is provided more naturally by path-expressions. Therefore, it appears that in OODBMSs there is no strong need for explicit joins, especially if path-expressions are provided. An example of path-expression (or simply path) is `Project.description.financial_info.total_budget` denoting the nested attribute "total_budget" of class Project. The evaluation of a query with nested predicates may cause the traversal of objects along aggregation hierarchies (Bertino, 1990; Jenq et al., 1989; Kim et al., 1988).

To expedite the evaluation of queries, relational database systems typically provide a secondary index using some variation of the B-tree structure (Bayer and McCreight, 1972; Comer, 1979), or some hashing technique. An index is maintained on an attribute or a combination of attributes of a relation. Since object-oriented databases require an attribute to be generalized to a nested attribute, secondary indexing must be also generalized to indexing of a nested attribute. It is important to note that in OODBMSs, as we discussed earlier, joins are very often implicit joins along aggregation hierarchies. This implies that most join operations are already predefined by the conceptual database schema. Moreover, joins are in most cases equality joins based on object-identifiers; that is, they are *identity equality* joins. Thus, it is possible to define specialized access techniques supporting fast traversal of aggregation hierarchies.

Maier and Stein (1986) and Kim (1990) provided preliminary discussions of the notion of secondary indexing on a sequence of nested attributes (or *path*). The concepts of nested index and path index were proposed by Bertino and Kim (1989) as access mechanisms to provide efficient support for queries on nested attributes and to evaluate their performance. Here, we address the problem of defining the optimal index configuration for a given sequence of nested attributes. Therefore, the contributions of this article with respect to the article by Bertino and Kim (1989) are:

- The definition of index configurations for a sequence of nested attributes.
- The definition of an algorithm that determines the optimal configuration.

We compare our approach with related work in the following section, after we introduce the basic concepts of our approach that are relevant for the understanding of the comparison.

The remainder of this article is organized as follows. In Section 2 we survey the concepts of nested index and path index (Bertino and Kim, 1989). We also introduce the concept of index configuration, and briefly describe index structures and operations. In Section 3 we present cost formulas that are used by the subsequent algorithm in Section 4.

2. Index Organizations

In this section, we first briefly recall two index organizations that support nested predicates. Then we introduce the concept of index configuration and the associated operations that are novel with respect to the material presented in previous articles.

2.1 Preliminary Definitions

The remainder of this section is based on the following concepts (see Bertino and Kim, 1989, for a more precise definition):

Path: A branch in an aggregation hierarchy; it consists of a class C followed by a sequence of attribute names.

Path instantiation: A sequence of objects found by instantiating a path.

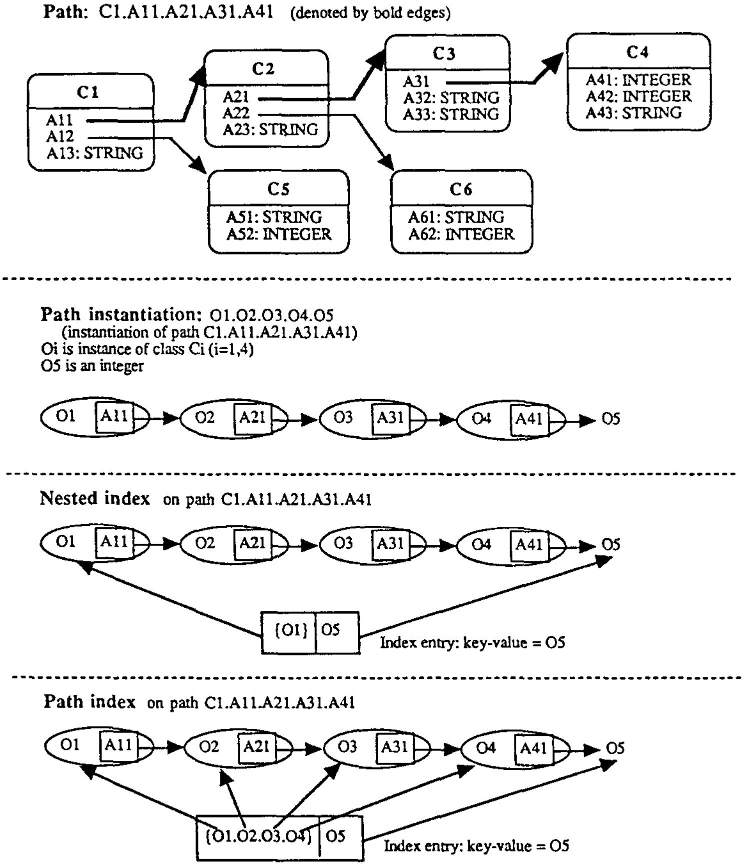
Nested index: An index establishing a direct connection between the object at the beginning of a path instantiation and the object at the end. The index is keyed on the objects at the end of path instantiations.

Path index: An index storing path instantiations (i.e., sequences of objects). The index is keyed on the objects at the end of path instantiations.

Figure 2 provides a graphic representation of those concepts.

The following are example paths for the aggregation hierarchy in Figure 1. In the examples, given a path P, len(P), class(P), and dom(P) denote, respectively, the length of P, the set of classes along P, and the domain of the last attribute in P.

Figure 2. Path, path instantiation, nested index, path index



P_1: `Task.participating_team.leader.name`
len(P_1)=3 class(P_1) = {Task, Research_Group, Researcher} dom(P_1) = STRING

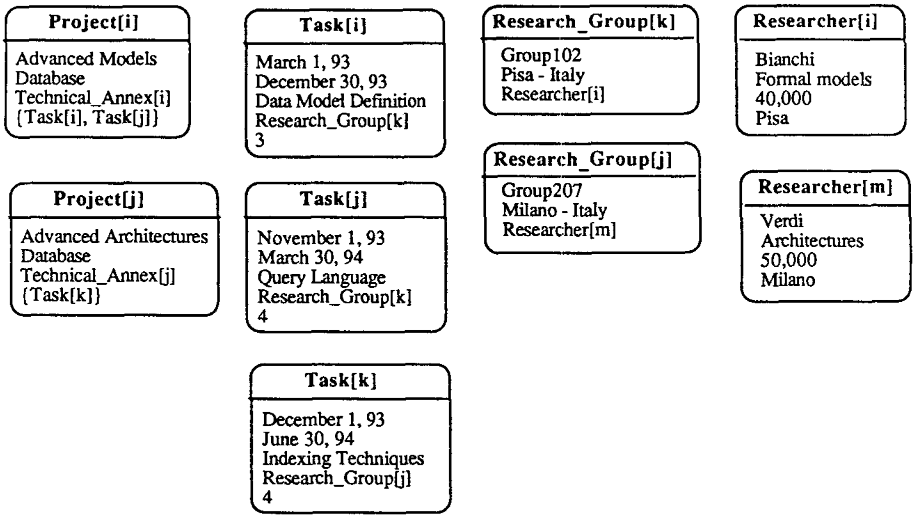
P_2: `Project.work_plan.participating_team.leader.name`
len(P_2)=4 class(P_2) = {Project, Task, Research_Group, Researcher} dom(P_2) = STRING

P_3: `Technical_Annex.financial_info.total_budget`
len(P_3)=2 class(P_3) = {Technical_Annex, Financial_Annex} dom(P_3) = INTEGER

P_4: `Project.description.financial_info`
len(P_4)=2 class(P_4) = {Project, Technical_Annex} dom(P_4) = Financial_Annex

The order of classes along a path is determined by the path definition itself. For example, in P_2, Project has position 1, Task has position 2, Research_Group has position 3, and Researcher has position 4.

Figure 3. Instances of classes of Figure 1



In the following, we assume that a UID for an object consists of the class identifier of the object's class, concatenated with the identifier of the object within its class. For example, Project[i] denotes the i-th instance of the class Project. Primitive objects (such as numbers, Booleans, characters, strings) are identified by their values. Note that an object O which is a component of another object O' has its own identifier, which does not contain the identifier of O'. This allows O to be a component of several different objects, and to be directly accessed without first accessing the object(s) of which it is a component.

The objects in Figure 3 are instances of some of the classes shown in Figure 1. As an example, a nested index on the path P_1 associates a distinct value of the name attribute with a list of object identifiers of class Task. For the objects shown in Figure 3, the nested index contains the following pairs:

(Bianchi, $\{$Task[i], Task[j]$\}$)
(Verdi, $\{$Task[k]$\}$).

The path-index on P_1 for the object in Figure 3 contains the following pairs:

(Bianchi, $\{$Task[i].Research_Group[k].Researcher[i],Task[j].
Research_Group[k].Researcher[i]$\}$)
(Verdi, $\{$Task[k].Research_Group[j].Researcher[m]$\}$).

Note that when $n=1$, the nested index and path index are identical and are the indexes used in most relational DBMSs. We refer to these indexes as *simple indexes*. Note also that a path index can be used to evaluate nested predicates on all classes along the path. In the current example, we could use the index to retrieve the researchers with a specified name, to retrieve the research groups whose leader has a specified name, or to retrieve the tasks with a participant team whose leader has a specified name.

Given a path $\mathcal{P}=C_1.A_1.A_2.\ldots.A_n$, and an object O_i of a class C_i in class($\mathcal{P}$), we use the term *forward traversal* to denote the access of objects $O_{i+1},\ldots,O_n$, such that O_{i+1} is referenced by O_i through $A_i,\ldots,$ O_n is referenced by O_{n-1} through A_{n-1}. Objects on a path may be traversed in a reverse direction of the path (i.e., $O_{i-1},\ldots,O_2,O_1$, such that O_{i-1} references object O_i through $A_{i-1},\ldots,$ and O_1 references O_2 through attribute A_1. It may not be profitable to support *backward traversal*, unless, given an object O, it is possible to directly determine the objects that reference O. For example, in GemStone reference from an object to another is unidirectional. In Orion (Kim et al., 1989) reverse references are supported for composite objects. Given an object O that has a reference to an object O', a reverse reference is a reference from O' to O. We make the assumption, as did Kim et al., that the reverse reference from an object O' to another object O is stored as a system-attribute in O'.

As an example, consider path $P_1=$ `Task.participating_team.leader.name`, a forward traversal of P_1 starting from object Task[i] implies access to objects Research_Group[k] and Researcher[i]. Indeed, object Task[i] references object Research_Group[k] through attribute "participating_team." Object Research_Group[k], in turn, references Researcher[i] through attribute "leader." By contrast, a backward traversal with reverse references from object Researcher[i] implies access to object Research_Group[k].

Index Structure and Operations. The data structure that we use to model the nested index and path index organizations is a B$^+$-tree (Bayer and McCreight, 1972; Comer, 1979). The format of the non-leaf node is identical in these two organizations. A non-leaf node consists of f records, where a record is a triple (key-length, key, pointer). The pointer contains the physical address of the next-level index node.

The format of the leaf nodes differs in the two index organizations. In a nested index, a leaf-node record consists of the record-length, key-length, key-value, number of elements in the list of UIDs, and the list of UIDs. In a path index, the format of a leaf-node record consists of the record-length, key-length, key-value, the number of elements in the list of path instantiations, and the list of path instantiations. Each path instantiation is implemented as an array of dimension equal to the path length. Figure 4 shows the format of a leaf-node record. Figure 5 provides examples of leaf-node records for the objects in Figure 3 on the path P_1. In the remainder of this section we briefly describe the operations on the two index organizations.

Nested Index. Given a path $\mathcal{P}=C_1.A_1.A_2.\ldots.A_n$ and a nested index defined on this path, the evaluation of a predicate against the nested attribute A_n of class C_1 requires the lookup of a single index. Therefore, the cost of evaluating a nested

Figure 4. Leaf-node in a path index

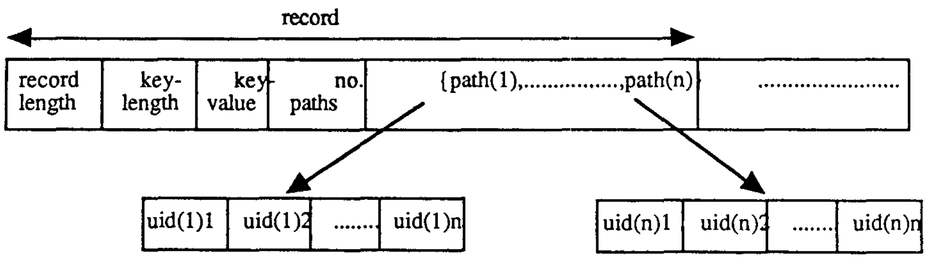
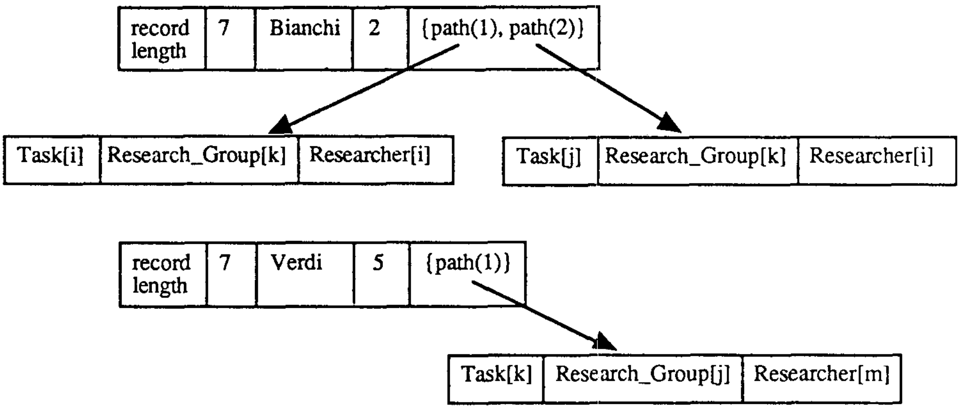


Figure 5. Leaf-node records in a path index



predicate is the same as if the attribute A_n were a direct attribute of class C_1.

Let us consider a path $\mathcal{P}=C_1.A_1.A_2.\ldots.A_n$ and an object O_i, $1 \leq i \leq n$ instance of a class C_i in class($\mathcal{P}$). Suppose that O_i has an object O_{i+1} as the value of attribute A_i and that O_i is updated to assign a new object O'_{i+1} to A_i. To update the index, two forward traversals must be executed to determine the value of nested attribute A_n with respect to O_{i+1} and O'_{i+1}. Then the path is reverse-traversed from object O_i to determine the UID(s) of object(s) of class C_1 that contains direct or indirect references to O_i. Finally, the data structure implementing the index is modified.

To update a nested index, in general, the path must be traversed. In particular, two forward traversals and one backward traversal are required. If O_i is the modified object, then the forward path traversal has length $l_f=n-i$, where n is the path length. The backward path traversal has length $l_b=i-2$ if $i>2$, $l_b=0$ if $i \leq 2$. If no reverse references are provided in objects, the nested index organization cannot be used.

The operation of insertion and deletion are similar to the update operation, except that only one forward traversal is executed.

Path Index. Given a path $\mathcal{P}=C_1.\ A_1.\ A_2.\ldots.\ A_n$ and a path index defined on it, the evaluation of a predicate against the nested attribute A_n of a class C_i, $1 \leq i \leq n$, requires a lookup of a single index. Once the set of path instantiations associated with the key value is determined, the i-th elements are extracted from the arrays representing these instantiations. However, a greater number of leaf-nodes may need to be accessed than with a corresponding nested index, because leaf-node records in a path index contain more information than those in a nested index. (For a comparison of the paths of lengths 2 and 3 see Bertino and Kim, 1989).

Again, let us assume that an object O_i, $1 \leq i \leq n$, which is an instance of class C_i in class($\mathcal{P}$), is modified by replacing object O_{i+1}, the value of A_i, with a new object O'_{i+1}. The effect of this update is that some instantiations have been modified and therefore the index must be updated. To update the index, two forward traversals must be performed, as in the case of the nested index. However, unlike the nested index organization, a path index does not require a backward traversal, since paths are stored in the leaf-node records. Therefore, this organization can be used even if backward references are not supported in objects. The update algorithm is described in detail elsewhere (Bertino and Kim, 1989).

2.2 Index Configuration

A path may be split into several subpaths, and for each subpath a different index organization may be used, or indexes may be used only on some subpaths. In this case we say that the path is supported by a *multi-index* organization. The motivation for splitting a path is mainly to reduce the update costs and, at the same time, provide efficient retrieval. Both nested indexes and path indexes have high update costs (Bertino and Kim, 1989), especially if allocated on long paths (i.e., with length greater than 3), and low retrieval cost. Conversely, the multi-index organization has low update cost and high retrieval cost. Therefore, the purpose of splitting a path into several subpaths is to provide intermediate configurations that allow the update costs to be reduced, while providing efficient retrieval. The algorithm described in Section 4 is used to determine the most efficient configuration. Let us consider the path P_2 = `Project.work_plan.participating_team.leader.name`. This path can be split in different ways, for example:

1. P_{2_1}=`Project.work_plan`
   P_{2_2}=`Task.participating_team`
   P_{2_3}=`Research_group.leader`
   P_{2_4}=`Leader.name`
2. P_{2_1}=`Project.work_plan.participating_team`
   P_{2_2}=`Research_group.leader`
   P_{2_3}=`Leader.name`

3. P_{2_1} =`Project.work_plan.participating_team`
   P_{2_2} =`Research_group.leader.name`

Given a path, the number of subpaths and the index organization of each subpath defines the *index configuration.* Which configuration is best depends on the access patterns and on data characteristics. An algorithm for configuration selection is presented in Section 4. The algorithm also determines whether a path must be split into several subpaths and for which subpaths an index must be allocated.

Definition 8. Given a path $\mathcal{P}=C_1.A_1.A_2.\ldots.A_n$, $(n \geq 1)$ an *index configuration* for $\mathcal{P}$ of degree k, denoted as $\chi(\mathcal{P})$, is defined as a sequence of pairs, $\{T_1, T_2, \ldots, T_k\}$, $(k \leq n)$. T_i $(1 \leq i \leq k)$ has the form $<S_i, IT_i>$ where

$S_i=C_j.\ A_j.\ A_{j+1}.\ldots.\ A_{j+l_i}$ $(j \geq i$ and $l_i \geq 0)$ is a *subpath definition*; the subpath length is $l_i + 1$; C_j is in class($\mathcal{P}$) and is called the *starting class* of the subpath and is denoted by SC_i; A_{j+l_i} is called *ending attribute* of the subpath and is denoted by EA_i;

IT_i indicates whether an index is allocated on the subpath, and the type of index allocated on the subpath; it can assume one of the following values: *NX, PX, I,* θ, where *NX* denotes a nested index, *PX* a path index, and *I* a simple index (i.e., a nested index defined on a subpath of length 1). The symbol θ denotes that no index is allocated on the subpath.

The sequence $\Sigma = \{S_1, S_2, \ldots, S_k\}$ is called the *subpath specification.* A configuration having degree greater than one is called a *split configuration.* This means that under the configuration the path has been split in at least two subpaths.

Given a path $\mathcal{P}=C_1.\ A_1.\ A_2.\ldots.A_n$, $(n \geq 1)$ a configuration $\chi(\mathcal{P})$ of degree k must satisfy the following constraints:

1. The ending attribute of S_k must be A_n.
2. The starting class of subpath S_i $(1 < i \leq k)$ must be the domain of the ending attribute of path S_{i-1}.

Note that these constraints require that subpaths be non-overlapping. This is to avoid overlapping indexes. Overlapping indexes cause higher update costs, since an update to an object implies the modification of several indexes. Moreover, retrieval becomes inefficient if there are several overlapping indexes on the same path.

Definition 9. Given a path $\mathcal{P}=C_1.\ A_1.\ A_2.\ldots.\ A_n$ $(n \geq 1)$, an index configuration for $\mathcal{P}, \chi(\mathcal{P})$, and a class C_i in class(S_j), C_i is *indexed* by $\chi(\mathcal{P})$ if one of the following conditions is satisfied:

a path index is allocated on S_j;

a nested index is allocated on S_j and C_i is the starting class of S_j.

As an example, consider the path P_3 = `Technical_Annex.financial_info.total_budget`, involving two classes. The possible configurations are as follows:

1. $\{<$ Technical_Annex.financial_info.total_budget, $NX >\}$
   In this case P_3 is not split and a nested index is used. Class Technical_Annex is indexed in this configuration, while class Financial_Annex is not indexed.
2. $\{<$ Technical_Annex.financial_info.total_budget, $PX >\}$
   In this case P_3 is not split and a path index is used. Both classes are indexed in this configuration.
3. $\{<$Technical_Annex.financial_info.total_budget, $\theta >\}$
   In this case no index is allocated on the path. Neither class is indexed in this configuration.
4. $\{<$Technical_Annex.financial_info, $I>$, $<$Financial_Annex.total_budget, $I>\}$
   In this configuration P_3 is split into two subpaths of length 1. A simple index is allocated on each subpath. Both classes are indexed in this configuration.
5. $\{<$Technical_Annex.financial_info, $\theta >$, $<$Financial_Annex.total_budget, $I>\}$
   In this configuration P_3 is split into two subpaths of length 1. An index is allocated only on the second subpath. Only Financial_Annex is indexed in this configuration.
6. $\{<$Technical_Annex.financial_info, $I>$, $<$Financial_Annex.total_budget, $\theta >\}$
   This configuration is similar to the previous one, except that the index is allocated only on the first subpath. Only Technical_Annex is indexed in this configuration.

In configurations 1, 2, and 3 the path is not split. Therefore, the subpath specification coincides with P_3. On the other hand, configurations 4, 5, and 6 are split configurations and the subpath specification is $\Sigma = \{$Technical_Annex.financial_info, Financial_Annex.total_budget$\}$.

Finally, a configuration where an index is defined on each subpath is *completely indexed*; otherwise it is said to be *partially indexed*. For example, Configuration 4 above is completely indexed, while Configuration 6 is partially indexed.

Note that given a configuration such as

$$\chi_1(P_2) = \{< \text{Project.work\_plan}, \theta >, < \text{Task.participating\_team}, \theta >, \\ < \text{Research\_Group.leader.name}, PX > \}$$

is equivalent with respect to the index allocation to the configuration

$$\chi_2(P_2) = \{< \text{Project.work\_plan.participating\_team}, \theta >, \\ < \text{Research\_Group.leader.name}, PX >\}.$$

We call a configuration like $\chi_2(P_2)$, where there are no two consecutive subpaths on which no index is allocated, a *non-trivial* configuration.

2.2.1 Retrieval Operations. Given a path $\mathcal{P}=C_1.\ A_1.\ A_2.\ldots.\ A_n$ and a configuration $\chi(\mathcal{P})$ of degree $k \geq 2$, the evaluation of a predicate against the nested attribute A_n with respect to class C_i may require the lookup of several indexes.

Let S_h be the subpath to which the class C_i belongs, then the evaluation of the predicate is executed as follows:

If $h < k$ (i.e., C_i does not belong to the last subpath) then

If an index is allocated on S_k, an index lookup is performed on this index to determine the instances of SC_k (starting class of S_k) that satisfy the given predicate (using whatever index organization is defined for S_k).

If no index is allocated on S_k, the instances of SC_k satisfying the given predicate are determined by accessing the instances themselves. The strategy for evaluating the predicate is determined by some query optimization algorithm (possible execution strategies have been proposed by Bertino, 1993). Let us indicate the qualifying set of SC_k instances as UID_{k-1}.

Then the instances of SC_{k-1} are determined such that their nested attribute EA_{k-1} assumes values in the set UID_{k-1}. This activity is executed by using an index, if an index is allocated on subpath S_{k-1}, or accessing the instances, according to some query execution strategy.

This process is repeated until the subpath S_h is reached. If an index is allocated on S_h and C_i is indexed by $\chi(\mathcal{P})$, an index lookup is then performed to determine the instances of class C_i such that the nested instance attribute EA_h assumes values in the set UID_h. Otherwise such instances are determined by accessing the instances themselves.

If $h{=}k$ (i.e., C_i belongs to the last subpath) then:

If an index is allocated on S_k and C_i is indexed by $\chi(\mathcal{P})$, only one index lookup is executed to determine the instances of C_i that satisfy the given predicate.

If no index is allocated or C_i is not indexed by $\chi(\mathcal{P})$, the instances verifying the predicate are determined by accessing the instances themselves, according to some query execution strategy.

Example 1. Consider the path P_2= `Project.work_plan.participating_team.leader.name` and the configuration

$$\{<\text{Project.work\_plan.participating\_team}, PX>,\ <\text{Research\_Group.leader.name}, NX>\}.$$

In this configuration the path has been split into two subpaths, such that a path index is allocated on the first subpath, while a nested index is allocated on the second path. The configuration is completely indexed since there is an index on each subpath. The two indexes will have the following entries for the objects in Figure 3:

Path index on `Project.work_plan.participating_team`

(Research_Group[k], {Project[i].Task[i], Project[i].Task[j]})

(Research_Group[j], {Project[j].Task[k]})

Nested index on `Research_Group.leader.name`
(Bianchi, {Research_Group[k]})
(Verdi, {Research_Group[j]})

Note that only classes Project, Task, and Research_Group are indexed by this configuration.

Suppose that we wish to retrieve all instances of class Project such that the nested attribute "name" is equal to a given value (e.g., "Bianchi"). In this case, a lookup on the nested index defined on the second subpath (i.e., `Research_Group.leader.name`) is performed. This lookup returns a set of UIDs of instances of class Task such that their nested attribute "name" verifies the given predicate. This set is {Research_Group[k]}.

Then a lookup on the path index defined on the first subpath is performed to retrieve the instances of the class Project having the attribute "participating_team" (ending attribute of the first subpath) values in the set of UIDs returned by the previous index lookup. In this case, this set contains only one UID (i.e. Research_Group[k]). The lookup of the path index for this UID returns as result {Project[i]}. □

Example 2. We now consider a partially indexed configuration, the path P_2 and the configuration

$$\{< \text{Project.work\_plan.participating\_team}, \theta >, \\ < \text{Research\_Group.leader.name}, NX >\}.$$

In this configuration, a nested index has been allocated on the second subpath (as in Example 1), while no index has been allocated on the first subpath. Suppose that, as in Example 1, we wish to determine all projects having a task whose leader is a researcher named Bianchi. This query is executed as follows. First an index lookup on the nested index is executed. The set {Research_Group[k]} is returned. Then instances of class Project are determined as having the nested attribute "participating_team" values in the set returned by the index lookup. Note that, depending on the query execution strategy, we may need to access instances of classes Task and Project, since there is no index allocated on the subpath containing these classes. □

2.2.2 Update Operations. We now consider the update operation. Consider a class C_i ($1 \leq i \leq n$) which is modified. Let S_h be the subpath to which C_i belongs. The update is executed depending on the index type, as discussed previously. Note, however, that the forward traversals need only to determine the old and new values of EA_h (subpath ending attribute) for the modified objects. Therefore, no access must be executed to instances of classes that belong to subpaths different from S_h. Similarly, if the index type is nested, the backward path traversal must be executed only up to the class following the starting class of the path.

Example 3. Consider the path P_2 = `Project.work_plan.participating_team.leader.name` and the configuration of Example 1:

{<Project.work_plan.participating_team, *PX*>,
<Research_Group.leader.name, *NX*>}.

Suppose that an update is performed that replaces the leader of Research_Group[j] with the new researcher Researcher[p]. In this configuration, the updates are on the second subpath and therefore only instances of classes in this subpath must be traversed. To determine the updates to be performed on the index, the following steps must be executed:

Researcher[m] is accessed and its value for attribute "name" is determined. The attribute value is "Verdi."

Researcher[p] is accessed and its value for attribute "name" is determined. Suppose the attribute value is "Smith."

The following updates are executed to the nested index:

Research_Group[j] is eliminated from the set of instances associated with the key value "Verdi."

Research_Group[j] is added to the set of instances associated with the key value "Smith."

Note that no backward traversal is needed in this case. Indeed, the modified class is the class at the beginning of the subpath. Note also that the classes in the other subpath do not need to be accessed. □

Finally, consider the configuration of Example 2, where no index is allocated on the first subpath. In this case, any update concerning instances of the classes in the second subpath is executed as in Example 3. If, however, updates occur on the first subpath, the update operation has no additional costs due to the index updates.

2.3 Related Work on Indexing

The problem of efficiently supporting hierarchical data has been widely investigated in the framework of CODASYL database systems. A basic feature of the CODASYL data model is the one of *set*, which allows a record type called *owner* to be connected to another record type called *member*. To efficiently support navigation from occurrences of a record owner to occurrences of a record member, different implementations of the set have been proposed (the most common order being based on pointer arrays or on list structures). However, a basic difference is that the CODASYL data model does not support a high level declarative query language as in the case of object-oriented data models. For example, to provide the equivalent of a nested predicate (e.g., `Task.participating_team.leader.name = Bianchi`), a program must be used in CODASYL. Therefore, most techniques proposed for supporting sets are really oriented toward a step-by-step navigation. For example, a pointer array associated with a given record occurrence only stores the pointers of the child record occurrences, but not those of the other descendants. To determine the descendants, a child occurrence first must be fetched, then the addresses of the record occurrences following in the hierarchies are determined from the pointer array of the child occurrence, and so on. Some extensions of these techniques are provided by the IMS system for a direct support of hierarchies of more than one set by flattening each hierarchy occurrence into a two-level hierarchy (Batory, 1985). However, those techniques can be applied only to trees, while in our case general graphs must be handled. Moreover, we note that all information needed to implement sets (such as pointer arrays) is stored with the record occurrences themselves (for example, a pointer array is stored with the occurrence of the parent record). By contrast, the purpose of the indexing techniques we present in this article is to provide generalized data organizations that allow nested predicates to be efficiently solved without having to access the objects themselves. Therefore, some data that facilitate an efficient query processing are stored in separate structures. This makes the data structures quite small and therefore more efficient. However, it is worthwhile noting that OODBMSs support two modes of object access: (i) high-level queries (like relational systems); (ii) step-by-step navigation (like CODASYL system). Very often the two access modes are used in a complementary way. A query is used to select a set of objects. The retrieved objects and their components are then accessed by using navigational capabilities. The problem we are addressing concerns the efficient support for high-level queries with nested predicates. CODASYL techniques could be used and/or extended for efficiently supporting navigation among objects. However, note that techniques used for supporting efficient step-by-step navigation could be used, in certain cases, as alternatives to indices to support queries. Therefore, the overall problem of physical database design for object-oriented databases is very complex because both associative accesses through queries and step-by-step navigation must be taken into account.

An indexing technique for complex objects has been proposed (Valduriez et al., 1986), based on the notion of *join index*, which was originally proposed for the relational model (Valduriez, 1987). A join index on two relations R and S is a file of pairs, where each pair contains the identifier (*surrogate*) of a tuple of R and the identifier of a tuple of S, such that the two tuples verify a given join predicate. In the implementation proposed by Valduriez (1987), two copies of the file can be allocated. One copy is clustered with respect to relation R and the other with respect to relation S. However, for limited access patterns (e.g., if always given a tuple of R, the matching tuples of S must be determined), a single copy is sufficient. Both copies are implemented as a B^+-tree. A join index in an object-oriented database can be used to support the implicit join between the instances of a class C and the instances of a class which is the domain of an attribute of C. Therefore, a sequence of join indexes could be used to support a nested predicate. We note that in this case a join index provides the same function as a nested index allocated on a path of length 1 (i.e., a simple index). Therefore, the sequence of join indexes is equivalent to a configuration where a given path has been split into several subpaths all of length 1. Therefore, our approach concerning path configurations is more general since it provides the equivalent of a sequence of join indexes as a particular case. Moreover, note that the path index that we consider here constitutes a generalization of the join index, since it allows an arbitrary number of classes connected through aggregated relationships to be related (rather than relating two classes).

A recent work (Kemper and Moerkotte, 1990) proposes a technique, called the *access support relation*, which is equivalent to the path index. The authors suggested that a path be split into several subpaths and different access support relations be allocated on each subpath, which is similar to our notion of configuration (except that they proposed a single indexing technique, while we consider two different indexing techniques). However, they proposed no algorithm for determining the optimal way to split a path. The definition of such an algorithm is our main goal in this article.

3. Cost Functions

In this section we present basic cost functions of index access and maintenance for the nested index and path index. Using these costs we derive the cost functions for a generic configuration. We present the workload model used in the selection algorithm. In defining the cost functions we use the parameters listed in Table 1. The parameters k_i ($1 \leq i \leq n$) model the *degree of reference sharing*. Two instances of a class C_i share a reference if they reference the same object O as value of attribute A_i. For example, objects Task[i] and Task[j] share a reference along the path P_1, since both objects reference Research_Group[k] through attribute "participating_team." None of the parameters used in the cost functions (except the physical page size) are input parameters to the configuration algorithm (Section 4). The parameters in Table 1 are derived from the input parameters of the algorithms. The input parameters are listed in Table 3.

3.1 Basic Cost Functions

We define the access cost functions for two types of index access: *single-key* and *key-set*. In the first type, a single key value is provided as input to the index lookup. In the second type, a set of key values randomly selected among the index keys is provided as input to the index lookup. We also define index maintenance costs and briefly discuss the access costs in cases where a predicate must be evaluated directly on the instances. In defining the cost functions we make the following assumptions:

1. The values of attributes are uniformly distributed among instances of the class defining the attributes.

Table 1. Cost function parameters

- h B^+-tree height
- X record size in a leaf-node
- D number of distinct key values in the index
- in_l $(1 \leq l \leq h)$ number of index nodes at level l of the index
- np number of pages occupied by a record when record size is larger than page size;
  $np = \lceil X/P \rceil$
- k_i average number of instances of class C_i assuming the same value for attribute A_i $(1 \leq i \leq n)$
- P physical page size

2. All attributes are single-valued.[2]
3. All key values have the same length.
4. Each instance of a class C_i is referenced by instances of class C_{i-1}, $1 < i \leq n$. Without this assumption, we would have to introduce additional parameters to take into account object reference topologies.

Access cost. The single-key access cost is denoted by I_{single} and is formulated as follows:

- $I_{single} = h$ if $X \leq P$
- $I_{single} = h - 1 + np$ if $X > P$.

The key-set access cost is denoted by $I_{set}(s)$ where s is the cardinality of the key-set. We evaluate the cost of a search for a number s of keys by using the formulation proposed by Lang et al. (1989):

- $I_{set}(s) = \sum_{l=1}^{h} H(s, in_l, D)$ if $X \leq P$
- $I_{set}(s) = (\sum_{l=1}^{h} H(s, in_l, D)) + s \times (np - 1)$ if $X > P$.

2. We introduce this assumption mainly for simplifying the presentation of the cost formulas. The organizations can easily support multi-valued attributes. In particular, while the path index does not need any extension, the nested index must be extended by including a counter for each UID in the set associated with a key value, indicating the number of different path instantiations starting with the same object and ending with the key value.

where H is the formula defined by Yao (1977). This formula determines the number of pages hit when accessing a number k of records randomly selected from a file containing n records grouped into m pages:

$$H(k,m,n) = m \times \left(1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}\right).$$

Bertino and Kim (1989) defined how values for h, X, and in_l are derived for both the nested index and path index, given parameters defining the data logical characteristics.

Maintenance cost. The index maintenance cost deriving from update, delete, or create operations for an instance of a class C_i (where the subscript i indicates the position of the class along the path) is denoted by U, D, and I, respectively. In computing this cost we exclude the costs of updating, deleting, or creating the instance itself, since these costs are common to all organizations, and focus on the additional costs of index modification. To further simplify the analysis, we consider only the costs of leaf-page modification and exclude the costs of index page splits (cf. Schkolnick and Tiberio, 1985).

In defining the cost functions we will make use of the following additional variables:

CFT cost of a forward traversal (in IO operations)
CBT cost of a backward traversal (in IO operations)
CBM average cost of the B^+-tree modification (in IO operations).

The cost functions for modification operations on a class C_i are as follows:

Nested Index $U = 2 \times CFT + CBT + 2 \times CBM$
$D = I = CFT + CBT + CBM$
Path Index
$U = 2 \times CFT + 2 \times CBM$
$D = I = CFT + CBM.$

CFT is evaluated by observing that the number of objects that must accessed is $n - i$. For each object, first an access must be executed to determine the physical address of the object (since references are logical), and then a second access to fetch the object itself. Therefore:

$CFT = 2 \times (n - i)$.
CBT is evaluated by the following expression:[3]
$CBT = 2 \times (\sum_{j=2}^{i-1} (\prod_{l=j}^{i-1} k_l))$ if $i > 2$
$CBT = 0$ otherwise.

3. The *CBT* cost is evaluated under the assumption that reverse references are used. The *CBT* cost in the case of no reverse references is not of interest for the present discussion and therefore is not reported.

In the previous expression the quantity in parentheses determines the number of objects to be accessed. For each of those objects, two IO operations are performed.

We note that costs of both forward and backward traversal are dependent on i (i.e., on the position of the modified class along the path). Since the purpose of forward traversal is to determine the value of the nested attribute at the end of the path for the modified instance, the cost of the forward traversal depends on the position of the class in the path. If the class is very close to the beginning of the path, the cost of forward traversal is very high, being proportional to the difference between the path length and the class position in the path. Similarly, the cost of backward traversal is directly proportional to the position of the class along the path.

CBM is formulated by the following expressions:

$CBM = h + 1$ if $X \leq P$

A number h of IO operations is executed to retrieve the leaf page containing the record to be modified; an additional IO operation is then executed to rewrite the modified page.

$CBM = h + 1 + (np - 1)/np$ if $X > P$

If the record size is larger than the page size, then a number h of IO are executed to access the leaf page that contains the initial part of the record. From the initial part of the record, it is possible to determine the page from which the UID must be deleted or to which the UID must be added. If this page is different from the page containing the initial part of the record, then a further access must be performed. The probability of a further page access is given by $(np - 1)/np$.

If no index is defined on a subpath, the cost maintenance for the subpath is zero.

Instance Access Cost. As we have seen in the previous section, when no index is defined on a subpath, a given predicate must be evaluated by accessing the instances themselves. The cost depends on the query execution strategy used (Bertino and Martino, 1993). However, it should be noted that the cost functions used are orthogonal to the algorithm presented in Section 4. For example, it is possible to use the cost estimates provided by a query optimizer (Finkelstein et al., 1988). In the following:

$A_{set}(C, A, U)$ denotes the cost of determining which instances of the class C assume values for the (nested) attribute A in a set of UIDs of cardinality U. As an example, given the set $U_c = \{$Task[i], Task[j]$\}$, A_{set}(Project, work_plan, card(U_c)) denotes the cost of determining which instances of the class Project have in their work plans Task[i] and/or Task[j].

$A_{single}(C, A, pred)$ denotes the cost of determining which instances of the class C have the (nested) attribute A that verifies the predicate *pred*. As an example, consider the predicate `name = Bianchi`. $A_{single}($ `Research_Group, leader.name, name=Bianchi`) denotes the cost of determining which instances of the class Research_Group are headed by a researcher named Bianchi.

3.2 Configuration Cost Functions

Now we present access and maintenance costs for index configurations of degree $k \geq 2$ (i.e., configurations consisting of at least two subpaths). When the configurations consist of only one subpath, then the costs are the ones presented in the previous subsection.

Access Costs. The access costs are provided only for the case of single-key predicates. The access costs for range-key or set-key predicates can be easily derived from the cost of single-key predicate.

Given a path $\mathcal{P}=C_1.\ A_1.\ A_2 \ldots A_n$ and a non-trivial configuration $\chi(\mathcal{P})$ of degree k, the cost of evaluating a single-key predicate *pred* on attribute A_n with respect to a class C_i $(1 \leq i \leq n)$ is denoted as $cost\_a[C_i,\ pred]$. and is obtained as follows. In the following cost expressions, $NUID_j$ denotes the cardinality of the set of UIDs obtained by the lookup on the j+1-th subpath (this set of UIDs is obtained by accessing the instances themselves or by scanning an index depending on the configuration of subpath S_{j+1}).

Let S_h be the subpath to which C_i belongs, then[4]
for h=k (i.e., S_h is the last subpath)

$$cost\_a[C_i,\ pred] = \begin{cases} I_{single}(S_k) & \text{if an index is allocated on} S_k \\ A_{single}(C_i, EA_k, pred) & \text{otherwise} \end{cases}$$

for h<k (i.e., S_h is not the last subpath)

$cost\_a[C_i, pred] = cost\_a[SC_k, pred] + [\sum_{j=h+1}^{k-1} Q_{set}(SC_j, NUID_j)] + Q_{set}(C_i, NUID_h)$.

The expression for function Q_{set} is provided in Table 2.

In the previous expression $I_{single}(S_k)$ denotes the access cost to the index allocated on S_k (last subpath) for evaluating *pred*. Since we made the assumption that *pred* is a single-key predicate, the access cost is the cost of an index lookup when a single key value is provided as input. This cost depends on the index organization defined on S_k and it is obtained by applying the basic cost functions for the single-key case defined in the previous subsection.

Similarly, $I_{set}(S_j,\ NUID_j)$ denotes the access cost to the index defined on the j-th subpath when a set of key-values is provided as input to the index lookup. The access cost is obtained by applying the cost functions defined for the key-set case for the two index organizations.

Example 4. Consider a path P_2= `Project.work_plan.participant_team.leader name` and the configuration of Example 2 in Section 2:

{<Project.work_plan.participating_team,θ >,
<Research_Group.leader.name,*NX*>}.

---

4. Recall that SC_j (EA_j), for $1 \leq j \leq k$ is the starting class (ending attribute) of subpath S_j.

Table 2. Functions Q_{single} and Q_{set}

- for $h+1 \leq j < k$

$$Q_{set}(SC_j, NUID_j) = \begin{cases} I_{set}(S_j, NUID_h) & \text{if an index is allocated on} S_j \\ A_{set}(SC_j, EA_j, NUID_j) & \text{otherwise} \end{cases}$$

-

$$Q_{set}(C_i, NUID_h) = \begin{cases} I_{set}(S_h, NUID_h) & \text{if an index is allocated on} S_h \\ A_{set}(C_i, EA_h, NUID_h) & \text{otherwise} \end{cases}$$

Suppose that we wish to determine all tasks with a participating team headed by a researcher named Bianchi. The cost of this query under the given configuration is given by *cost_a*[Task, name="Bianchi"]=*cost_a* [Research_Group, name="Bianchi"] + Q_{set}(Task, $NUID_1$)= $I_{single}(S_2)$ + A_{set}(Task, participating_team, $NUID_1$).

The cost of the query in the example is explained by observing that the query is executed by first determining the instances of class Research_Group headed by a researcher named Bianchi, and then by determining the instances of class Task having as participating team one of the qualified instances of class Research_Group (cf. Example 2 in Section 2). The first step is executed by scanning an index, since an index is allocated on the subpath Research_Group.leader.name. The second step is executed by accessing the instances of class Task, since no index is allocated on the subpath Project.work_plan.participating_team. □

$NUID_j$ ($j<k$) for a configuration of degree k is evaluated as follows:

$$NUID_j = \prod_{i=ncl}^{n} k_i \times f(pred)$$

where ncl is such that C_{ncl} is the starting class of the $j+1$-th subpath; $f(pred)$ is a factor depending on the type of predicate *pred* (it is derived by using standard formulas for predicate selectivities estimation). In particular, when the predicate *pred* contains the operator $=$, $f(pred) = 1$. In the example in Figure 6, we consider a configuration that consists of four subpaths and a predicate *pred* containing the $=$ operator. We present the values of *NUID* for subpaths S_1, S_2, and S_3.

We note that the cost of the j-th index lookup, when $j<k$, is dependent only on the cardinality of the set $NUID_j$ and on the index organization for the subpath S_j. However, the cost is independent from the organization of all the other subpaths. When no index is allocated on a subpath S_j, the cost of determining which instances of a class in S_j assume a value in $NUID_j$ for the nested attribute at the end of the subpath is independent from the index organization of the other subpaths.

Figure 6. Example of evaluation of $NUID_i$ for configuration

---

$P=C_1.\ A_1.\ A_2.\ A_3.\ A_4.\ A_5.\ A_6.\ A_7$, len($P$)=7
$\chi(P)=\{<C_1.\ A_1.\ A_2,\ NX>, <C_3.\ A_3,\ I>, <C_4.\ A_4.\ A_5,\ PX>, <C_6.\ A_6.\ A_7,\ NX>\}$
$k_1=2,\ k_2=2,\ k_3=1,\ k_4=2,\ k_5=2,\ k_6=2,\ k_7=3$
$NUID_1=k_3 \times k_4 \times k_5 \times k_6 \times k_7=24$ $NUID_2=k_4 \times k_5 \times k_6 \times k_7=24$ $NUID_3=k_6 \times k_7=6$

When $h=k$, the cost of the j-th index loop is only dependent from the index organization defined on S_k, and from the data logical characteristics. The cost is similar when no index is allocated on a subpath.

Therefore these cost functions verify a property that is similar to the *separability* property defined by Whang et al. (1984). This is important because we can choose the optimal index organization for each subpath independently from the index organizations chosen for the other subpaths.

Maintenance Cost. The maintenance costs are defined as the basic costs. The only difference is that the forward traversal and the backward traversal (for the nested index organization) are limited to classes in the subpath to which the modified class belongs.

More formally, let C_i be the modified class, S_h be the subpath to which C_i belongs, e be such that A_e is the ending attribute of S_h, h_s be such that C_{h_s} is the starting class of S_h (cf. Section 2). Then:

$CFT = 2 \times (e - \mathrm{i})$
$CBT = 2 \times (\sum_{j=h_s+1}^{i-1} (\prod_{l=j}^{i-1} k_l))$ if $i - h_s > 1$
$CBT = 0$ otherwise.

Because a modification operation on a class C_i involves accessing only object instances of the classes in the subpath to which C_i belongs, the cost functions for update, delete, and create operations have the separability property. That is, the costs are only the functions of the index organization defined for the subpath and it is independent from the organizations of the other subpaths.

In the following we will denote the cost of an update on a class C_i as *cost_u*$[C_i]$. The cost of a delete or create operation will be denoted as *cost_d_i*$[C_i]$, since cost functions are equal for the delete and create operations. Note that *cost_u*$[C_i]$=*cost_d_i*$[C_i]$=0 if no index is allocated on the subpath.

3.3 Workload Model

To determine the optimal index configuration for a given path, the expected workload for classes along the path must be specified.

Given a path $\mathcal{P}=C_1.A_1.A_2....A_n$, the workload in our case is characterized by a set of triplets

$$W(\mathcal{P}) = \{(\alpha_i,\ \nu_i,\ \delta_i),\ i = 1,2,\ldots,n\},$$

where:

α_i is the frequency of evaluation of a single-key predicate on attribute A_n with respect to class C_i;

ν_i is the frequency of updates executed on attribute A_i of class C_i;

δ_i is the frequency of instance deletions and generations for class C_i.

All frequencies are expressed as real numbers in the interval [0,1]. We assume that the frequencies are provided as input to the algorithm. Very often, these frequencies are provided by the physical database designer (Finkelstein et al., 1988) or can be obtained by monitoring the system (Yu et al., 1985).

Given a workload specification, the optimal index configuration must minimize the following cost expression:

$$\sum_{i=1}^{n} \alpha_i \times cost\_a[C_i,pred] + \nu_i \times cost\_u[C_i] + \delta_i \times cost\_d\_i[C_i].$$

3.3.1 Subpath Workload. Given a configuration $\chi(\mathcal{P})$ and a workload specification, it is important to determine the workload on each subpath. This makes it possible to determine the best index organization for each subpath. For a given S_i, the subpath workload of S_i determines the frequencies of retrieval and modification operations that are executed on each class in class(S_i).

Given a subpath S_i, the workload specification of S_i is defined as a set of triplets

$$DW(S_i) = \{\ (d\alpha_j,\ d\nu_j,\ d\delta_j), j = i_s, i_s + 1,\ldots, i_s + l_i - 1\}.$$

where l_i denotes the length of S_i (cf. Section 2) and i_s denotes the subscript of the starting class of S_i. Therefore, the derived workload for a given subpath contains a number of triplets equal to the number of classes along the subpath; $d\alpha_j$, $d\nu_j$, and $d\delta_j$ are derived as follows:

$d\alpha_{i_s} = [\sum_{j=1}^{i-1} \sum_{h=0}^{l_j-1} \alpha_{(j_s+h)}] + \alpha_{i_s}$

$d\alpha_j = \alpha_j,\ j = i_s+ 1, \ldots, i_s+ l_i- 1.$

$d\nu_j = \nu_j,\ j = i_s, i_s+ 1, \ldots, i_s+ l_i- 1.$

$d\delta_j = \delta_j,\ j = i_s, i_s+ 1, \ldots, i_s+ l_i- 1.$

The workload for a subpath S_i is derived by observing that the index defined on the subpath is accessed for retrieval each time a predicate on any class along S_i must be evaluated and also each time a predicate on any class along any subpath S_j preceding S_i (i.e., $j < i$) must be evaluated. By contrast, the workload for the modification operations depends only on the frequencies of these operations for the classes along S_i. In particular, the instances of SC_i (starting class of S_i) will have to be retrieved when:

Figure 7. Example of derived workload

$P{=}C_1.\,A_1.\,A_2.\,A_3.\,A_4.\,A_5$, len($P$)=5 $\chi(P)=$
$\{<C_1.\,A_1.\,A_2, NX>, <C_3.\,A_3, I>, <C_4.\,A_4.\,A_5, PX>\}$
$\alpha_1{=}0.20$ $\nu_1{=}0.01$ $\delta_1{=}0.01$ $\alpha_2{=}0.00$ $\nu_2{=}0.01$ $\delta_2{=}0.01$ $\alpha_3{=}0.20$ $\nu_3{=}0.10$ $\delta_3{=}0.02$
$\alpha_4{=}0.20$ $\nu_4{=}0.05$ $\delta_4{=}0.02$
$\alpha_5{=}0.10$ $\nu_5{=}0.05$ $\delta_5{=}0.02$
$d\alpha_1{=}0.20$ $d\nu_1{=}0.01$ $d\delta_1{=}0.01$ $d\alpha_2{=}0.00$ $d\nu_2{=}0.01$ $d\delta_2{=}0.01$
$d\alpha_3{=}0.40$ $d\nu_3{=}0.10$ $d\delta_3{=}0.02$ $d\alpha_4{=}0.60$ $d\nu_4{=}0.05$ $d\delta_4{=}0.02$
$d\alpha_5{=}0.10$ $d\nu_5{=}0.05$ $d\delta_5{=}0.02$

1. a predicate must be evaluated on SC_i or
2. a predicate must be evaluated on any class preceding SC_i in $\mathcal{P}$.

As an example, consider the configuration $\{<$ Project.work_plan.participating_team, $\theta >, <$ Research_Group.leader.name, $NX>\}$.

Under this configuration, the instances of class Research_Group must be retrieved each time a query is issued on class Research_Group, and also when queries are issued on classes Project and Task, since these are classes preceding Research_Group in the path. Indeed, the qualified set of UIDs of instances of class Research_Group are used to determine the qualifying instances of class Project and Task.

Therefore, the relative derived retrieval frequency for the starting class of each path includes also the retrieval due to queries on preceding classes. As in the example in Figure 7, we consider an index configuration, consisting of three subpaths and a workload, and we derive *DW.*

Note that the cost of the j-th index lookup, when $j<k$, is dependent on the cardinality $NUID_j$ and on the index organization for the subpath S_j. However, the cost is independent from the index organization of all the other subpaths.

Given a configuration $\chi(\mathcal{P})$ of degree k, and a subpath S_i, $(1 \leq i < k)$ we define the overall cost for S_i as

$$cost(S_i) = \sum_{h=0}^{l_i-1} d\alpha_{(i_s+h)} \times Q_{set}(C_{(i_s+h)}, NUID_i) +$$

$$[\sum_{h=0}^{l_i-1} d\nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}] + d\delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}]].$$

Substituting expressions for $d\alpha_i$, $d\nu_i$, and $d\delta_i$, and recalling the configuration cost functions presented in the previous subsection, we obtain:

$$cost(S_i) = Q_{set}(SC_i, NUID_i) \times \sum_{j=1}^{i-1} \sum_{h=0}^{l_j-1} \alpha_{(j_s+h)} + \sum_{h=0}^{l_i-1} \alpha_{(i_s+h)} \times$$

$Q_{set}(C_{(i_s+h)}, NUID_i) + \nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}] + \delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}]$.[5]

The overall cost for S_k (last subpath) is defined as follows:

$$cost(S_k) = \sum_{h=0}^{l_k-1} d\alpha_{(k_s+h)} \times cost\_a[C_{(k_s+h)}, pred] +$$

$$d\nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}] + d\delta_{(k_s+h)} \times cost\_d\_i[C_{(k_s+h)}].$$

As in the previous case, we obtain the following expression for $cost(S_k)$:

$$cost(S_k) = cost\_a[SC_k, pred] \times \sum_{j=1}^{k-1} \sum_{h=0}^{l_j-1} \alpha_{(j_s+h)} + \sum_{h=0}^{l_k-1} \alpha \times$$

$$cost\_a[C_{(k_s+h)}), pred] + \nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}] + \delta_{(k_s+h)} \times cost\_d\_i[C_{(i_s+h)}].$$

Because of the separability property (cf. Subsection 3.2), $cost(S_i)$ $(1 \le i \le k)$ is independent from the $cost(S_j)$ $(1 \le j \le k$ and $i \ne j)$.

Finally we define the overall cost for $\chi(\mathcal{P})$ as

$$ov\_cost[\chi(\mathcal{P})] = \sum_{i=1}^{k} cost(S_i).$$

The following proposition holds.

Proposition 1. Given a path $\mathcal{P}$=$C_1. A_1. A_2, \ldots, A_n$, a workload $W(\mathcal{P})$, and a configuration $\chi(\mathcal{P})$ of degree k

$$\textstyle\sum_{i=1}^{n} \alpha_i \times cost\_a[C_i, pred] + \nu_i \times cost\_u[C_i] + \delta_i \times cost\_d\_i[C_i] = \quad (1)$$

$$\textstyle\sum_{i=1}^{k} cost(S_i). \quad (2)$$

The proof of the proposition is given in Appendix A. Therefore, the problem of finding a configuration that minimizes expression (1) can be restated as the problem of finding a configuration that minimizes expression (2).

4. Selection Algorithm

The selection algorithm receives as input a set of parameters defining:

path definition, $\mathcal{P}$=$C_1. A_1. A_2. \ldots A_n$ $(n>1)$;

5. Recall that C_{i_s}=SC_i since i_s denotes the subscript of the starting class of path S_i.

Table 3. Data parameters

- D_i number of distinct values for attribute A_i, $1 \leq i \leq n$.
- N_i cardinality of class C_i, $1 \leq i \leq n$ assuming the same value for attribute A_i ($1 \leq i \leq n$).
- $PC(C_i)$, $1 \leq i \leq n$, number of disk pages containing instances of class C_i.
- r_i ($2 \leq i \leq n$) a binary variable assuming value equal to 1 if instances of class C_i have reverse references to instances of class C_{i-1} in the path; equal to 0 otherwise.
- *kl* average length of a key value for the indexed attribute, (i.e. A_n).
- $r[dom(P)]$ a binary variable assuming value equal to 1 if the instances of the class $dom(P)$ have reverse references to the instances of class C_n; equal to 0 otherwise. This variable assumes always value 0 if $dom(P)$ is a primitive class (i.e., string, number, character, etc.).
- *UIDL* length of the object-identifier.

data characteristics (listed in Table 3);[6]
workload specification (as defined in the previous section).

The algorithm is organized in n steps. In the first step all subpaths of length 1 are considered and for each of them the costs are evaluated. In the first step there is only one choice to be made since for the case of subpath length equal to 1 the nested index and path index are identical. Therefore, the only choice is whether to allocate an index. At the second step all subpaths of length 2 are considered. In this case there are six possible choices for each subpath. A nested or path index can be used, or no index allocated, or the subpath can be further split into two subpaths of length 1, such that an index is allocated on both subpaths, or only on one (Subsection 2.2). In particular, a nested index is taken into consideration only if there are backward references from the second class in the subpath to the first class in the subpath and there is no retrieval from the second class of the subpath. The path index is always taken into consideration. Then the costs for all possible subpath configurations are evaluated and the configuration with the minimum cost

6. Parameters in Table 3 are used to derive all parameters (except the page size) used in the cost formulations, such as the index height listed in Table 1. In this article we do not discuss how parameters in Table 1 are derived from parameters in Table 3, since this derivation is presented in a previous article (Bertino and Kim, 1989). However, this derivation is quite trivial; it mainly concerns the determination of index characteristics, such as height or leaf-node sizes, from input parameters such as the number of instances per class and the number of distinct values for attributes.

is chosen. Note that the cost of the configuration where the subpath is split into two subpaths of length 1 is obtained as the sum of the costs of each of these subpaths. These costs have already been evaluated at the previous step.

At the k-th step, all paths of lengths k are considered. For each subpath the algorithm considers k+2 choices. The first choice is represented by the nested index. This organization is taken into consideration if there are reverse references among the classes in the subpaths, and if the frequency of retrieval is zero for all the classes in the subpath except the starting class. The second choice is represented by a path index, which is always taken into consideration. The third choice is represented by not allocating any index on the path. The remaining $k - 1$ choices are obtained by considering all possible configurations obtained by further splitting the subpath into two subpaths, that is, considering all the split configurations. Given a subpath $S_j = C_j.\ A_j.\ A_{j+1}.\ A_{j+2}.\ldots.\ A_j{+}k{-}1$ (of length k) the configurations that are considered are the following:

$$(1)\overbrace{C_j.A_j.A_{j+1}.\ldots.A_{j+k-2}},\ \overbrace{C_{j+k-1}.A_{j+k-1}}$$

$$(2)\overbrace{C_j.A_j.A_{j+1}.\ldots.A_{j+k-3}},\ \overbrace{C_{j+k-2}.A_{j+k-2}.A_{j+k-1}}$$

$$\ldots\ldots$$

$$(k-2)\ \overbrace{C_j.A_j.A_{j+1}},\ \overbrace{C_{j+2}.A_{j+2}.\ldots.A_{j+k-1}}\,.$$

$$(k-1)\ \overbrace{C_j.A_j},\ \overbrace{C_{j+1}.A_{j+1}.\ldots.A_{j+k-1}}\,.$$

The cost of one these configurations is given as the sum of the costs of the two subpaths that make up the configuration. For example *ov_cost(configuration*$_2$*)* $=$ *cost*$(C_j.\ A_j.\ A_{j+1}.\ldots A_{j+k-3})$ + *cost*$(C_{j+k-2}.A_{j+k-2}.A_{j+k-1})$.

Note that the configurations and costs of the subpaths into which S_j can be split have already been evaluated at some previous steps.

The costs of all k+2 choices are then evaluated and the choice with the minimum cost is selected. Note that in the resulting configuration, S_j can be split in more than two subpaths. This happens when a split configuration is chosen for S_j. In the chosen configuration S_j is split into two subpaths S_i and S'_i. For each of these two subpaths the optimal configuration has been determined at some previous step and the resulting configuration for S_j is the concatenation of the configurations of S_i and S'_i. If, for example, S_i has in turn a split configuration consisting of two subpaths S_h and S'_h, then the overall configuration of S_j is the concatenation of the configurations of S_h, S'_h, S'_i and therefore S_j is split into three subpaths.

At step n the algorithm considers all possible paths of length n. There is only one path of this length—the input path $\mathcal{P}$. The algorithm generates a number of choices equal to n+2 where the first three choices are the nested index, the path index, or no index. The remaining $n - 1$ choices are the split configurations obtained by splitting $\mathcal{P}$ into two subpaths. The costs of all the choices are evaluated. The choice with the minimum cost is the configuration selected for the input path $\mathcal{P}$.

In all the steps the costs for each subpath are evaluated with respect to the derived workload of the subpath.

We assume that when considering a configuration without index, all possible execution strategies are considered and the one with the minimum cost is selected.

In presenting the algorithm we will make use of some additional notations:

S_i^j denotes a subpath of length j having as starting class C_i.

cost_NX, cost_PX, and *cost_I* denote the overall subpath cost when a nested index, a path index, and a simple index are used respectively.

cost_θ denotes the cost of the most efficient execution strategy (Bertino and Martino, 1993) when no index is allocated.

Given two paths $\mathcal{P}$ and $\mathcal{P}'$ and two configurations $\chi(\mathcal{P})$ and $\chi(\mathcal{P}')$ of degree k and k', such that:

$\chi(\mathcal{P})=\{T_1, T_2, \ldots, T_k\}$
$\chi(\mathcal{P}')=\{T_1', T_2', \ldots, T_k'\}$
cat$[\chi(\mathcal{P}), \chi(\mathcal{P}')]$ is a configuration of degree $k+k'$ defined as the sequence

$$\{T_1, T_2, \ldots, T_k, T_1', T_2', \ldots, T_k'\}$$

The algorithm is organized in the following steps:

- *STEP 1.* Consider all subpaths of length 1. The number of these subpaths is n, and each subpath has the form $S_i^1=C_i.\ A_i$. For each S_i^1

  1. evaluate $DW(S_i^1)$ for $i=1,\ldots, n$;
  2. evaluate *cost_I*;
  3. evaluate *cost_θ*;
  4. $cost(S_i^1)=min\{cost\_I, cost\_\theta\}$

- *STEP 2.* Consider all subpaths of length 2. The number of these subpaths is $n - 1$ and each subpath has the form $S_i^2=C_i.\ A_i.\ A_{i+1}$. For each S_i^2

  1. evaluate $DW(S_i^2)$;
  2. if $r_{i+1}=1$, and $\alpha_{i+1}=0$, evaluate *cost_NX*; otherwise $cost\_NX=\infty$;
  3. evaluate *cost_PX*;
  4. evaluate *cost_θ*;
  5. $cost_1 = cost(S_i^1) + cost(S_{i+1}^1)$ where
     - S_i^1 is a subpath of length 1 such that the starting class of S_i^1 is C_i;
     - S_{i+1}^1 is a subpath of length 1 such that the starting class of S_{i+1}^1 is C_{i+1};

  (note that $cost(S_i^1)$ and $cost(S_{i+1}^1)$ have been evaluated at the previous step).

6. $cost(S_i^2) = min\{cost\_NX, cost\_PX, cost\_\theta, cost_1\}$
7. In this step the optimal configuration for the subpath S_i^2 is determined.
$\chi\ (S_i^2) = \{<C_i.\ A_i.\ A_{i+1}, NX>\}$ if *cost_NX* is the minimum cost; else
$\chi\ (S_i^2) = \{<C_i.\ A_i.\ A_{i+1}, PX>\}$ if *cost_PX* is the minimum cost; else
$\chi\ (S_i^2) = \{<C_i.\ A_i.\ A_{i+1}, \theta>\}$ if *cost_θ* is the minimum cost; else
$\chi\ (S_i^2) =$ cat $[\chi\ (S_i^1), \chi\ (S_{i+1}^1)\ ]$.

- *STEP 3.* Consider all subpaths of length 3. The number of these subpaths is $n-2$ and each subpath has the form $S_i^3 = C_i.\ A_i.\ A_{i+1}.\ A_{i+2}$. For each S_i^3
  1. evaluate $DW(S_i^3)$;
  2. If $r_{i+1}=1$, $r_{i+2}=1$, $\alpha_{i+1}=0$, and $\alpha_{i+2}=0$, evaluate *cost_NX*; otherwise $cost\_NX=\infty$;
  3. evaluate *cost_PX*;
  4. evaluate *cost_θ*;
  5. $cost_1 = cost(S_i^2) + cost(S_{i+2}^1)$ where
     - S_i^2 is a subpath of length 2 having as starting class C_i;
     - S_{i+2}^1 is a subpath of length 1 having as starting class C_{i+2};
  6. $cost_2 = cost(S_i^1) + cost(S_{i+1}^2)$ where
     - S_i^1 is a subpath of length 1 having as starting class C_i;
     - S_{i+1}^2 is a subpath of length 2 having as starting class C_{i+1};
  7. $cost(S_i^3) = min\{cost\_NX, cost\_PX, cost\_\theta, cost_1, cost_2\}$
  8. In this step the optimal configuration for the subpath S_i^3 is determined.
  $\chi\ (S_i^3) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}, NX>\}$ if *cost_NX* is the minimum cost; else
  $\chi\ (S_i^3) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}, PX>\}$ if *cost_PX* is the minimum cost; else
  $\chi\ (S_i^3) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}, \theta>\}$ if *cost_θ* is the minimum cost; else
  $\chi\ (S_i^3) =$ cat $[\chi(S_i^2), \chi\ (S_{i+2}^1)\ ]$ if $cost_1$ is the minimum cost; else
  $\chi\ (S_i^3) =$ cat $[\chi(S_i^1), \chi(S_{i+1}^2)]$.

- *STEP k.* Consider all subpaths of length k ($k<n$). The number of these subpaths is $n+1-k$ and each subpath has the form $S_i^k = C_i.\ A_i.\ A_{i+1}.\ A_{i+2}.\ldots.A_{i+k-1}$. For each S_i^k
  1. evaluate $DW(S_i^k)$;

2. If $r_{i+1}=1$, $r_{i+2}=1$, ..., $r_{i+k-1}=1$, and $\alpha_{i+1}=0$, $\alpha_{i+2}=0$, ..., and $\alpha_{i+k-1}=0$ evaluate *cost_NX*; otherwise *cost_NX*$=\infty$;
3. evaluate *cost_PX*;
4. evaluate *cost_θ*;
5. For $l=1, k-1$:
   $cost_l = cost(S_i^{k-l}) + cost(S_{i+k-l}^l)$ where
   - S_i^{k-l} is a subpath of length $k-l$ having as starting class C_i;
   - S_{i+k-l}^l is a subpath of length l having as starting class C_{i+k-l}
6. $cost_{l'}=min\{cost_1, cost_2, \ldots, cost_{k-1}\}$
7. $cost(S_i^k)=min\{cost\_NX, cost\_PX, cost\_\theta, cost_{l'}\}$
8. In this step the optimal configuration for the subpath S_i^k is determined.
   $\chi\ (S_i^k) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}.\ldots.A_{i+k-1}, NX>\}$ if *cost_NX* is the minimum cost; else
   $\chi\ (S_i^k) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}.\ldots.A_{i+k-1}, PX>\}$ if *cost_PX* is the minimum cost; else
   $\chi\ (S_i^k) = \{<C_i.\ A_i.\ A_{i+1}.\ A_{i+2}\ldots A_{i+k-1}, \theta>\}$ if *cost_θ* is the minimum cost; else
   $\chi\ (S_i^k) = \text{cat}[\chi(S_i^{k-l'}), \chi\ (S_{i+k-l'}^{l'})]$ if $cost_l'$ is the minimum cost.

- *STEP n.* Consider the subpath of length n. There is only one subpath of this length and coincides with the input path $\mathcal{P}$.

1. evaluate $DW(S_1^n)$;
2. If $r_2=1, \ldots, r_n=1$, $\alpha_2=0$, ..., and $\alpha_n=0$ evaluate *cost_NX*; otherwise *cost_NX*$=\infty$;
3. evaluate *cost_PX*;
4. evaluate *cost_θ*;
5. For $l=1, n-1$:
   $cost_l = cost(S_1^{n-l}) + cost(S_{n-l}^l)$ where
   - S_1^{n-l} is a subpath of length *n-l* having as starting class C_1;
   - S_{n-l}^l is a subpath of length l having as starting class C_{n-l}
6. $cost_{l'}=min\{cost_1, cost_2, \ldots, cost_{n-1}\}$
7. $cost(S_1^n)=min\{cost\_NX, cost\_PX, cost\_\theta, cost_{l'}\}$

8. In this step the optimal configuration for the subpathS_1^n is determined.
$\chi\ (S_1^n) = \{<C_1.\ A_1.\ A_2 \ldots A_n,\ NX>\}$ if *cost_NX* is the minimum cost; else
$\chi\ (S_1^n) = \{<C_1.\ A_1.\ A_2 \ldots A_n,\ PX>\}$ if *cost_PX* is the minimum cost; else
$\chi\ (S_1^n) = \{<C_1.\ A_1.\ A_2 \ldots A_n,\ \theta>\}$ if *cost_θ* is the minimum cost; else
$\chi\ (S_1^n) = \mathrm{cat}[\chi(S_1^{n-l'}),\ \chi\ (S_{n-l'}^{l'})]$ if $cost_{l'}$ is the minimum cost.

The optimal configuration for the path $\mathcal{P}$ is given by $\chi\ (S_1^n)$.

The formal correctness proof of this algorithm is presented in Appendix B. Here we provide some informal justification. We note that a correct algorithm is one that would consider all possible ways of splitting a path (including the case of not splitting the path), and for each one of these ways would consider all possible index organizations. The algorithm, at *STEP n*, considers first the case of not splitting the path, and then the three possible organizations (nested index, path index, and no index). Then it considers all possible ways of splitting the path into two subpaths. Note that there is no need at *STEP n* to consider splitting the path into a larger number of subpaths (e.g., three or more), since these have already been considered when evaluating the subpaths of length lower than n at the previous steps of the algorithm. The algorithm considers all possible ways of splitting the path into two subpaths. The optimal configuration of each of these subpaths is in turn obtained by evaluating all possible index organizations and all possible ways of splitting them into two subpaths.

4.1 Complexity Evaluation

The complexity of the algorithm is evaluated in terms of the number of configurations whose costs must be evaluated. Given a path P whose length is n, the number of configurations $nc(P)$ that are evaluated is given in the worst case by the following expression:

$$nc(P) = 2 * n + \sum_{i=2}^{n}((n+1-i) \times (3+i-1)).$$

This expression is obtained as follows:

- The term $2*n$ is the number of configurations that are examined at *STEP 1*. At this step all subpaths of P of length 1 are considered. Since P has length n, the number of such subpaths is n. For each path of length 1, we consider only two configurations: (1) allocation of a simple index; (2) no allocation of an index.

- At step i-th of the algorithm, we consider all subpaths of P whose length is i. The number of such subpaths is $n+1 - i$. For each subpath of length i,

Table 4. Data parameters, workload, reverse reference specification

- Data parameters
  - N_1=200,000 D_1=20,000 k_1=10 $PC(C_1)$=10,000
  - N_2= 20,000 D_2=10,000 k_2= 2 $PC(C_2)$= 1,000
  - N_3= 10,000 D_3=10,000 k_3=1 $PC(C_3)$= 500
  - N_4= 10,000 D_4=10,000 k_4=1 $PC(C_4)$= 500
  - N_5= 10,000 D_5=10,000 k_5=1 $PC(C_5)$= 500
- Workload
  - α_1=0.1 ν_1=0.05 δ_1=0.05
  - α_2=0.1 ν_2=0.05 δ_2=0.05
  - α_3=0.1 ν_3=0.05 δ_3=0.05
  - α_4=0.1 ν_4=0.05 δ_4=0.05
  - α_5=0.0 ν_5=0.1 δ_5=0.1
- Reverse reference specification: r_i=1 $(1 < i \leq 5)$.
- $r[dom(P)]=0$

we consider the following configurations: path index, nested index, no index. Moreover, we consider a number of configurations that consist of splitting the subpaths of length i into pairs of subpaths. The number of such pairs for a subpath of length i is equal to $i - 1$. Therefore, we find that the number of configurations for a subpath of length i is (3 + i - 1). Note that this is the worst case, since the nested index organization is not considered when there are no reverse references, and thus the number of configurations is (2 + i - 1).

By developing the above expression for nc we obtain that the total number of configurations to be considered is $c = (n^3 + 9 \times n^2 + 2 \times n)/6$. Therefore, the complexity of the algorithm is polynomial. In terms of storage, the algorithm requires that the optimal configuration cost for each subpath be considered. Since the total number of subpaths is $n + \sum_{i=2}^{n} (n+1 - i) = (n^2 + n)/2$, the space required is proportional to this expression which is linear with the square of n, that is, with the path length.

4.2 Illustrative Examples

To illustrate the algorithm, we consider the path P=C_1. A_1. A_2. A_3. A_4. A_5. Table 4 presents the data parameters, workload, values, and reverse reference specification.

In the table, we also report the values of parameters k_i which are derived from N_i and D_i.

- *STEP 1.* The subpaths of length 1 are considered. Their costs are as follows:
  - $S_1^1=C_1.A_1$ $cost\_I$=0.9 $cost\_\theta$=0.4
    $cost(S_1^1)$=0.4 $\chi(S_1^1)=\{<C_1.A_1,\theta>\}$
  - $S_2^1=C_2.A_2$ $cost\_I$=0.85 $cost\_\theta$=0.4
    $cost(S_2^1)$=0.4 $\chi(S_2^1)=\{<C_2.A_2,\theta>\}$
  - $S_3^1=C_3.A_3$ $cost\_I$=1.05 $cost\_\theta$=0.6
    $cost(S_3^1)$=0.6 $\chi(S_3^1)=\{<C_3.A_3,\theta>\}$
  - $S_4^1=C_4.A_4$ $cost\_I$=1.25 $cost\_\theta$=0.8
    $cost(S_4^1)$=0.8 $\chi(S_4^1)=\{<C_4.A_4,\theta>\}$
  - $S_5^1=C_5.A_5$ $cost\_I$=1.7 $cost\_\theta$=200
    $cost(S_5^1)$=1.7 $\chi(S_5^1)=\{<C_5.A_5,I>\}$

- *STEP 2.* The subpaths of length 2 are considered. Their costs and configurations are as follows:
  - $S_1^2=C_1.A_1.A_2$ for this path the nested index cannot be used because class C_2 has a frequency of predicate evaluation (α_2) which is different from zero. Therefore the following configurations are considered for this subpath:
    * $\{<C_1.A_1.A_2,PX>\}$ $cost\_PX$=2.1
    * $\{<C_1.A_1.A_2,\theta>\}$ $cost\_\theta$=0.8
    * $\{<C_1.A_1,\theta>,<C_2.A_2,\theta>\}$ $cost_1$=0.4+0.4=0.8

    Therefore $\chi(S_1^2)=\{<C_1.A_1.A_2,\theta>\}$ $cost(S_1^2)$=0.8
  - $S_2^2=C_2.A_2.A_3$ for this path the nested index cannot be used because class C_3 has a frequency of predicate evaluation (α_3) which is different from zero. Therefore, the following configurations are considered for this subpath:
    * $\{<C_2.A_2.A_3,PX>\}$ $cost\_PX$=1.8
    * $\{<C_2.A_2.A_3,\theta>\}$ $cost\_\theta$=1
    * $\{<C_2.A_2,\theta>,<C_3.A_3,\theta>\}$ $cost_1$=0.4+0.6=1

    Therefore, $\chi(S_2^2)=\{<C_2.A_2.A_3,\theta>\}$ $cost(S_2^2)$=1
  - $S_3^2=C_3.A_3.A_4$ for this path the nested index cannot be used because class C_4 has a frequency of predicate evaluation (α_4) which is different from zero. Therefore the following configurations are considered for this subpath:

* $\{<C_3.\ A_3.\ A_4, PX>\}$ $cost\_PX$=2
* $\{<C_3.\ A_3.\ A_4, \theta>\}$ $cost\_\theta$=1.4
* $\{<C_3.\ A_3, \theta>, <C_4.\ A_4, \theta>\}$ $cost_1$=0.6+0.8=1.4

Therefore $\chi(S_3^2)=\{<C_3.\ A_3.\ A_4, \theta>\}$ $cost(S_3^2)$=1.4

- $S_4^2=C_4.\ A_4.\ A_5$ for this path the nested index can be used because $\alpha_5=0$ and $r_5=1$. Therefore, the following configurations are considered for this subpath:
  * $\{<C_4.\ A_4.\ A_5, NX>\}$ $cost\_NX$=2.45
  * $\{<C_4.\ A_4.\ A_5, PX>\}$ $cost\_PX$=2.45
  * $\{<C_4.\ A_4.\ A_5, \theta>\}$ $cost\_\theta$=200
  * $\{<C_4.\ A_4, \theta>, <C_5.\ A_5, I>\}$ $cost_1$=0.8+1.7=2.5

  Therefore $\chi(S_4^2)=\{<C_4.\ A_4.\ A_5, NX>\}$ $cost(S_4^2)$=2.45

- *STEP 3.* The subpaths of length 3 are considered. Their costs and configurations are as follows:
  - $S_1^3=C_1.\ A_1.\ A_2.\ A_3$. The following configurations are considered for this subpath:
    * $\{<C_1.\ A_1.\ A_2.\ A_3, PX>\}$ $cost\_PX$=3.6
    * $\{<C_1.\ A_1.\ A_2.\ A_3, \theta>\}$ $cost\_\theta$=1.4
    * cat[$\chi(S_1^2)$, $\chi(S_3^1)$] $cost_1$=0.8+0.6=1.4
    * cat[$\chi(S_1^1)$, $\chi(S_2^2)$] $cost_2$=0.4+1=1.4

    Therefore, $\chi(S_1^3)=\{<C_1.\ A_1.\ A_2.\ A_3, \theta>\}$ $cost(S_1^3)$=1.4.
  - $S_2^3=C_2.\ A_2.\ A_3.\ A_4$. The following configurations are considered for this subpath:
    * $\{<C_2.\ A_2.\ A_3.\ A_4, PX>\}$ $cost\_PX$=3.05
    * $\{<C_2.\ A_2.\ A_3.\ A_4, \theta>\}$ $cost\_\theta$=1.8
    * cat[$\chi(S_2^2)$, $\chi(S_4^1)$] $cost_1$=1+0.8=1.8
    * cat[$\chi(S_2^1)$, $\chi(S_3^2)$] $cost_2$=0.4+1.4=1.8

    Therefore, $\chi(S_2^3)=\{<C_2.\ A_2.\ A_3.\ A_4, \theta>\}$ $cost(S_2^3)$=1.8.
  - $S_3^3=C_3.\ A_3.\ A_4.\ A_5$. The following configurations are considered for this subpath:
    * $\{<C_3.\ A_3.\ A_4.\ A_5, PX>\}$ $cost\_PX$=3.5
    * $\{<C_3.\ A_3.\ A_4.\ A_5, \theta>\}$ $cost\_\theta$=200.6
    * cat[$\chi(S_3^2)$, $\chi(S_5^1)$] $cost_1$=1.4+1.7=3.1
    * cat[$\chi(S_3^1)$, $\chi(S_4^2)$] $cost_2$=0.6+2.45=3.05

Therefore $\chi(S_3^3)$= cat[$\chi(S_3^1)$, $\chi(S_4^2)$]= $\{<C_3.\ A_3, \theta>, <C_4.\ A_4.\ A_5, NX>\}$
$cost(S_3^3)$=3.05.

- *STEP 4.* The subpaths of length 4 are considered. Their costs and configurations are as follows:
  - $S_1^4 = C_1.\ A_1.\ A_2.\ A_3.\ A_4$. The following configurations are considered for this subpath:
    - $\{<C_1.\ A_1.\ A_2.\ A_3.\ A_4, PX>\}$ $cost\_PX$=5.4
    - $\{<C_1.\ A_1.\ A_2.\ A_3.\ A_4, \theta>\}$ $cost\_\theta$=2.2
    - cat[$\chi(S_1^3)$, $\chi(S_4^1)$] $cost_1$=1.4+0.8=2.2
    - cat[$\chi(S_1^2)$, $\chi(S_3^2)$] $cost_2$=0.8+1.4=2.2
    - cat[$\chi(S_1^1)$, $\chi(S_2^3)$] $cost_3$=0.4+1.8=2.2

    Therefore $\chi(S_1^4)$= $\{<C_1.\ A_1.\ A_2.\ A_3.\ A_4, \theta>\}$ $cost(S_1^4)$=2.2
  - $S_2^4 = C_2.\ A_2.\ A_3.\ A_4.\ A_5$. The following configurations are considered for this subpath:
    - $\{<C_2.\ A_2.\ A_3.\ A_4.\ A_5, PX>\}$ $cost\_PX$=4.85
    - $\{<C_2.\ A_2.\ A_3.\ A_4.\ A_5, \theta>\}$ $cost\_\theta$=201
    - cat[$\chi(S_2^3)$, $\chi(S_5^1)$] $cost_1$=1.8+1.7=3.5
    - cat[$\chi(S_2^2)$, $\chi(S_4^2)$] $cost_2$=1+2.45=3.45
    - cat[$\chi(S_2^1)$, $\chi(S_3^3)$] $cost_2$=0.4+3.05=3.45

    Therefore, $\chi(S_2^4)$= [$\chi(S_2^2)$, $\chi(S_4^2)$] = $\{<C_2.\ A_2.\ A_3, \theta>, <C_4.\ A_4.\ A_5, NX>\}$ $cost(S_2^4)$=3.45.

- *STEP 5.* The subpath of length 5 is considered, $S_1^5 = C_1.\ A_1.\ A_2.\ A_3.\ A_4.\ A_5$. The following configurations are considered:
  - $\{<C_1.\ A_1.\ A_2.\ A_3.\ A_4.\ A_5, PX>\}$ $cost\_PX$=7.8
  - $\{<C_1.\ A_1.\ A_2.\ A_3.\ A_4.\ A_5, \theta>\}$ $cost\_\theta$=201.2
  - cat[$\chi(S_1^4)$, $\chi(S_5^1)$] $cost_1$=2.2+1.7=3.9
  - cat[$\chi(S_1^3)$, $\chi(S_4^2)$] $cost_2$=1.4+2.45=3.85
  - cat[$\chi(S_1^2)$, $\chi(S_3^3)$] $cost_3$=0.8+3.05=3.85
  - cat[$\chi(S_1^1)$, $\chi(S_2^4)$] $cost_4$=0.4+3.45=3.85

  Therefore $\chi(S_1^5)$= cat[$\chi(S_1^3)$, $\chi(S_4^2)$] =
  $\{<C_1.\ A_1.\ A_2.\ A_3, \theta>, <C_4.\ A_4.\ A_5, NX>\}$ $cost(S_1^5)$=3.85.

The resulting configuration for P is given by $\chi(S_1^5)$. The optimal configuration consists of splitting the path into two subpaths. The first is $C_1.\ A_1.\ A_2.\ A_3$. No index is allocated on this subpath. The second subpath is $C_4.\ A_4.\ A_5$. A nested index is allocated on this subpath.

Note that, in the example, the best query execution strategy is based on reverse traversal (in all cases except for subpaths that are last in the configuration). The reason for this is that there are reverse references among objects in the example. This allows the system to determine the instances of a class C_i that reference a given instance O of class C_{i+1} directly from O by using reverse references. Therefore, there is no need to access all instances of class C_i, as would have been needed without reverse references. Intuitively, we can see that the configuration has been chosen because an index on the last subpath avoids accessing all instances of the last class (C_5) to evaluate the predicate. Not having an index on the last subpath would have implied a total scanning of class C_5. By contrast, it is not convenient to allocate an index on the first subpath since there are reverse references among objects. Therefore, once the instances of class C_4 that verify the predicates are determined, it is possible to determine the instances of classes C_1, C_2, C_3 by simply navigating backward using the reverse references. This is particularly efficient since the degree of reference sharing is rather low. For example, given an object O, instance of class C_4, there is only one instance of class C_3 that references O. On the other hand, given an object O', instance of class C_3, there are two instances of class C_2 that reference O'. Therefore, since objects contain reverse references, the reverse traversal is very efficient, and this eliminates the need for the index on the first subpath.

To further assess this point we consider the data parameters and workload in Table 4, while there are no reverse references among objects. In this case, the configuration chosen by the algorithm (we omit the steps for brevity) is $\{<C_1.\ A_1, I>, C_2.\ A_2.\ A_3, PI>, <C_4.\ A_4.\ A_5, PI>\}$. The overall cost of this configuration is 5.25. Under this configuration, the path has been split into three subpaths and an index allocated on each subpath. In this case, since there are no reverse references, the query execution strategies based on instance access are very expensive.

Among the two configurations, the first has a lower overall cost. This shows that reverse references are useful not only for supporting referential integrity and enforcing certain types of constraint (Kim et al., 1989), but they can also be used in some cases to provide efficient query execution strategies.

To show the influence of the degree of reference sharing, we consider another example where these degrees have higher values for some classes than those of the previous examples. Table 5 presents the data parameters, workload, and reverse reference specification.

The configuration chosen is $\{<C_1.\ A_1.\ A_2.\ A_3, PI>, <C_4.\ A_4.\ A_5, NI>\}$. In this case, even if there are reverse references, it is more efficient to split the path into two subpaths and to allocate a path index on the first, and a nested index on the second. The nested index is allocated on the second subpath because, as in the

Table 5. Data parameters, workload, reverse reference specification

- Data parameters
  - N_1=200,000 D_1=200,000 k_1= 1 $PC(C_1)$=10,000
  - N_2= 20,000 D_2= 20,000 k_2=10 $PC(C_2)$=10,000
  - N_3= 20,000 D_3= 2,000 k_3=10 $PC(C_3)$= 1,500
  - N_4= 2,000 D_4= 200 k_4=10 $PC(C_4)$= 100
  - N_5= 500 D_5= 100 k_5= 2 $PC(C_5)$= 10
- Workload
  - α_1=0.1 ν_1=0.05 δ_1=0.05
  - α_2=0.1 ν_2=0.05 δ_2=0.05
  - α_3=0.1 ν_3=0.05 δ_3=0.05
  - α_4=0.1 ν_4=0.05 δ_4=0.05
  - α_5=0.0 ν_5=0.1 δ_5=0.1
- Reverse reference specification: r_i=1 ($1 < i \leq 5$).
- $r[dom(P)]=0$

previous example, there is no retrieval from class C_5 (α_5=0.0). This configuration is the most efficient because, when the degree of reference sharing increases, the number of object accesses in the reverse traversal becomes quite high. In this situation, it is preferable to allocate an index.

5. Summary and Future Work

In this article we presented a formal definition of access mechanisms to support the evaluation of nested predicates on a path. A path is defined as a sequence of classes, such that the first class has a domain of an attribute of the second class of the path, the second class has a domain of an attribute of the third class of the path, and so forth. We presented an algorithm that defines the optimal index configuration for a path, and we gave parameters defining the access patterns and logical data characteristics.

The algorithm presented defines the optimal index configuration when only a path is considered. We note that queries may contain nested predicates on several paths originating from the same class. Therefore, future work includes the extension to the case of multiple paths when these paths have overlapping subpaths. We note, however, that the case we have considered in this article (i.e., a single nested predicate) is quite significant. In fact, a nested predicate is equivalent in a relational query language to a restriction on a relation attribute, and to several joins among different relations.

It is also important to observe that the algorithm proposed should be included in a more general methodology for index allocation. We believe that existing methodologies, such as those proposed by Finkelstein et al. (1988), Reuter and Kinzinger (1984), Lam et al. (1988), and Rullo and Sacca (1988) can be extended to deal with object-oriented databases and the novel indexing techniques. Finally, another open research issue concerns how to integrate the indexing techniques described in this article with the class hierarchy indexing technique proposed by Kim et al. (1989).

References

Andrews, T. and Harris, C. Combining language and database advances in an object-oriented development environment. *Proceedings of the Second International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, FL, 1987.

Banerjee, J., Chou, H.T., Garza, J., Kim, W., Woelk, D., Ballou, N., and Kim, H.J. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, 1987.

Banerjee, J., Kim, W., and Kim, H.K. Queries in object-oriented databases. *Proceedings of the IEEE International Conference on Data Engineering*, Los Angeles, CA, 1988.

Batory, D.S. Modeling the storage architecture of commercial database systems. *ACM Transactions on Database Systems*, 10(4):463-528, 1985.

Bayer, R. and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3): 173-189, 1972.

Bertino, E. and Kim, W. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196-214, 1989.

Bertino, E. Query optimization using nested indices. *Proceedings of the Second International Conference on Extending Database Technology (EDBT90),* Venice, Italy, March 26-30, 1990, Lecture Notes in Computer Science 416, Springer-Verlag.

Bertino, E., Negri, M., Pelagatti, G., and Sbattella, L. Object-oriented query languages: The notion and the issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223-237, 1992.

Bertino, E. and Martino, L. *Object-Oriented Database Systems: Concepts and Architectures*. New York: Addison-Wesley, 1993.

Bertino, E. On index configuration in object-oriented databases. Extended version, August 1993. In: Kim, W. and Lochovsky, F., eds., *Object-Oriented Concepts, Databases, and Applications.* New York: Addison-Wesley, 1989, pp. 283-308.

Bjornerstedt, A. and Hulten, C. Version control in an object-oriented architecture. In: W. Kim, and F. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications.* New York: Addison-Wesley, 1989, pp. 451-485.

Breitl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, H., and Williams, M. The GemStone data management system. In: Kim, W., and Lochovsky, F., eds., *Object-Oriented Concepts, Databases, and Applications.* New York: Addison-Wesley, 1989, pp. 283-308.

Comer, D. The ubiquitous B-tree. *ACM Computing Surveys,* 11(2):121-137, 1979.

Deux, O. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering,* 2(1):91-108, 1990.

Finkelstein, S., Schkolnick, M., and Tiberio, P. Physical database design for relational databases. *ACM Transactions on Database Systems,* 13(1): 91-128, 1988.

Fishman, D.H., Beech, D., Cate, H., Chow, E., Connors, T., Davis, J., Derrett, N., Hoch, C., Kent, W., Lyngback, P., Mahbod, B., Neimat, M., Ryan, T., and Shan, M. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems,* 5(1):48-69, 1987.

Jenq, P., Woelk, D., Kim, W., and Lee, W.L. Query processing in distributed ORION. MCC Technical Report, No. ACA-ST-035-89, January 1989.

Lam, H., Su, S., and Kogant, N. A physical database design evaluation system for CODASYL databases. *IEEE Transactions on Software Engineering,* 14(7):1010-1022, 1988.

Lang, S.D., Driscoll, J., and Dou, J. A unified analysis of batched searching and tree-structured files. *ACM Transactions on Database Systems,* 14(4):604-618, 1989.

Kemper, A. and Moerkotte, G. Access support in object bases. *Proceedings of the ACM-SIGMOD Conference on Management of Data,* Atlantic City, NJ, 1990.

Kim, W. A foundation for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering,* 2(3):327-341, 1990.

Kim, K.C., Kim, W., Woelk, D., and Dale, A. Acyclic query processing in object-oriented databases. *Proceedings of the Entity-Relationship Conference,* Rome, 1988. Also: MCC Technical Report, No. ACA-ST-287-88, September 1988.

Kim, W., Bertino, E., and Garza, J.F. Composite objects revisited. *Proceedings of the ACM-SIGMOD Conference on Management of Data,* Portland, OR, 1989. Also: MCC Technical Report, No. ACA-ST-387-88, Nov. 1988.

Kim, W., Kim K.C., and Dale A. Indexing techniques for object-oriented databases. In: Kim, W. and Lochovsky, F.H., eds., *Object-Oriented Concepts, Databases, and Applications,* New York: Addison-Wesley, 1989.

Maier, D. and Stein, J. Indexing in an object-oriented DBMS. *Proceedings of the IEEE Workshop on Object-Oriented DBMS,* Asilomar, CA, 1986.

Reuter, A. and Kinzinger, H. Automatic design of the internal schema for a CODASYL database system. *IEEE Transactions on Software Engineering,* 10(4):358-375, 1984.

Rullo, P. and Sacca, D. An automatic physical designer for network model databases. *IEEE Transactions on Software Engineering,* (14):9, 1293-1306, 1988.

Schkolnick, M. and Tiberio, P. Estimating the cost of updates in a relational database. *ACM Transactions on Database Systems,* 10(2):163-179, 1985.

Skarra, A., Zdonik, S., and Reiss, S. An object server for an object-oriented database system. *Proceedings of the First International Workshop on Object-Oriented Database Systems*, Asilomar, CA, 1986.

Valduriez, P., Khoshafian, S., and Copeland, G. Implementation techniques of complex objects. *Proceedings of the International Conference on Very Large Data Bases,* Kyoto, Japan, 1986.

Valduriez, P. Join indices. *ACM Transactions on Database Systems*, 12(2):218-246, 1987.

Whang, K.Y, Wiederhold, G., and Sagalowicz, D. Separability: An approach to physical database design. *IEEE Transactions on Computer Systems*, 33(3):209-222, 1984.

Yao, S.B. Approximating block accesses in database organizations. *ACM Communications*, 20(4):260-261, 1977.

Yu, C., Suen, C., Lam, K., and Siu, M.K. Adaptive record clustering. *ACM Transactions on Database Systems*, 14(2):147-167, 1985.

Appendix A: Proof of Proposition 1

Given the configuration $\chi(P)$ of degree k, each class along path P belongs to only one of the subpaths into which P is split. Therefore, expression (1) is developed as follows:

$$\sum_{i=1}^{n} \alpha_i \times cost\_a[C_i, pred] + \nu_i \times cost\_u[C_i] + \delta_i \times cost\_d\_i[C_i] =$$

$$\sum_{i=1}^{k} \sum_{h=0}^{l_i-1} \alpha_{(i_s+h)} \times cost\_a[C_{(i_s+h)}, pred] +$$

$$\nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}] + \delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}] =$$

recalling the cost function for $cost\_a[C_i, pred]$)

$$\sum_{i=1}^{k-1} \sum_{h=0}^{l_i-1} [\alpha_{(i_s+h)} \times [cost\_a[SC_k, pred] + [\sum_{j=i+1}^{k-1} Q_{set}(SC_j, NUID_j)] +$$

$$Q_{set}(C_{(i_s+h)}, NUID_i)] + \nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}] +$$

$$\delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}]] + \sum_{h=0}^{l_k-1} [\alpha_{(k_s+h)} \times cost\_a[C_{(k_s+h)}, pred] +$$

$$\nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}] + \delta_{(k_s+h)} \times cost\_d\_i[C_{(k_s+h)}]] =$$

$$\sum_{i=1}^{k-1}\sum_{h=0}^{l_i-1}[\alpha_{(i_s+h)} \times [\sum_{j=i+1}^{k-1} Q_{set}(SC_j, NUID_j)]+$$

$$\alpha_{(i_s+h)} \times Q_{set}(C_{(i_s+h)}, NUID_i) + \alpha_{(i_s+h)} \times cost\_a[SC_k, pred]+$$

$$\nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}] + \delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}]]+$$

$$\sum_{h=0}^{l_k-1}[\alpha_{(k_s+h)} \times cost\_a[C_{(k_s+h)}, pred] + \nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}]+$$

$$\delta_{(k_s+h)} \times cost\_d\_i[C_{(k_s+h)}]] = \sum_{i=1}^{k-1}[Q_{set}(SC_i, NUID_i) \times \sum_{j=1}^{i-1}\sum_{h=0}^{l_j-1} \alpha_{(j_s+h)}]+$$

$$\sum_{h=0}^{l_i-1} \alpha_{(i_s+h)} \times Q_{set}(C_{(i_s+h)}, UID_i) + \nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}]+$$

$$\delta_{(i_s+h)} \times cost\_d\_i[C_{(i_s+h)}]] + cost\_a[SC_k, pred]\times$$

$$[\sum_{i=1}^{k-1}\sum_{h=0}^{l_i-1} \alpha_{(i_s+h)}] + \sum_{h=0}^{l_k-1} \alpha_{(k_s+h)} \times cost\_a[C_{(k_s+h)}, pred]+$$

$$\nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}] + \delta_{(k_s+h)} \times cost\_d\_i[C_{(k_s+h)}] \quad (3)$$

Expression (2) is developed as follows:

$$\sum_{i=1}^{k} cost(S_i) = \sum_{i=1}^{k-1} cost(S_i) + cost(S_k) =$$

$$\sum_{i=1}^{k-1}[Q_{set}(SC_i, NUID_i) \times \sum_{j=1}^{i-1}\sum_{h=0}^{l_j-1} \alpha_{(j_s+h)}+$$

$$\sum_{h=0}^{l_i-1} \alpha_{(i_s+h)} \times Q_{set}(C_{(i_s+h)}, NUID_i) + \nu_{(i_s+h)} \times cost\_u[C_{(i_s+h)}]+$$

$$\delta(i_s + h) \times cost\_d\_i[C_{(i_s+h)}]] + cost\_a[SC_k, pred]\times$$

$$[\sum_{j=1}^{k-1}\sum_{h=0}^{l_j-1} \alpha_{(j_s+h)}] + \sum_{h=0}^{l_k-1} \alpha_{(k_s+h)} \times cost\_a[C_{(k_s+h)}, pred]+$$

$$\nu_{(k_s+h)} \times cost\_u[C_{(k_s+h)}] + \delta_{(k_s+h)} \times cost\_d\_i[C_{(k_s+h)}] \quad (4)$$

Since expressions (3) and (4) are equal, the assertion is proved.

Appendix B: Correctness Proof of Selection Algorithm

We show that the algorithm determines the optimal non-trivial configuration. That is, given $\Theta(\mathcal{P})=\{\chi_1(\mathcal{P}),\chi_2(\mathcal{P}), \ldots, \chi_p(\mathcal{P})\}$ the set of all possible configurations for a path $\mathcal{P}$, the algorithm determines a non-trivial configuration $\chi_i(\mathcal{P})$ such that

$$ov\_cost(\chi_i(\mathcal{P})) \leq ov\_cost(\chi_j(\mathcal{P})) \text{ for} 1 \leq j \leq p$$

where $\chi_i(\mathcal{P}) \in \Theta(\mathcal{P})$ and $\chi_j(\mathcal{P}) \in \Theta(\mathcal{P})$.

In the proof we will make use of the following definition:

Definition 8. Given two subpaths

$$S_1 = C_i.A_i.A_{i+1}.\ldots.A_{i+l_1} \text{ and } S_2 = C_j.A_j.A_{j+1}.\ldots.A_{j+l_2}$$

they can be concatenated if C_j is the domain of attribute A_{i+l_1} of class C_{i+l_1}. The concatenation is denoted and defined as follows:

$$\text{cat}[S_1, S_2] = C_i.A_i.A_{i+1}.\ldots.A_{i+l_1}.A_j.A_{j+1}.\ldots.A_{j+l_2}.$$

For example, given the path P=$C_1.A_1.A_2.A_3.A_4$ and two subpaths S_1=$C_1.A_1.A_2$ and S_2=$C_3.A_3.A_4$,

$$\text{cat}[S_1, S_2] = C_1.A_1.A_2.A_3.A_4.$$

We first prove that the algorithm determines the optimal split configuration. We denote as $\Theta_s(\mathcal{P})$ the set of all possible split configurations for $\mathcal{P}$.

$$\Theta_s(\mathcal{P}) = \{\chi_j(\mathcal{P})/\chi_j(\mathcal{P}) \in \Theta(\mathcal{P}) \text{ and } degree(\chi_j(\mathcal{P})) \geq 2\}.$$

Proposition 2. Given a path $\mathcal{P}$ of length n, the algorithm determines a configuration $\chi_h(\mathcal{P})$, $\chi_h(\mathcal{P}) \in \Theta_s(\mathcal{P})$ such that:

$$ov\_cost(\chi_h(\mathcal{P})) \leq ov\_cost(\chi_j(\mathcal{P})),\ \forall \chi_j(\mathcal{P}) \in \Theta_s(\mathcal{P}).$$

Proof. The proof is ab absurdo. Let's assume that exists a configuration $\chi_y(\mathcal{P})$ such that

$$ov\_cost(\chi_y(\mathcal{P})) < ov\_cost(\chi_h(\mathcal{P}))\ h \neq y\ (4)$$

where $\chi_h(\mathcal{P})$ is the configuration determined by the algorithm. We show that a contradiction follows. Let Σ_h and Σ_y be the subpath specification of the two configurations. We consider two cases. □

Case 1. $\Sigma_h = \Sigma_y = \{S_1, S_2, \ldots, S_m\}$ where m is the degree of the two configurations. This means that, under the two configurations, $\mathcal{P}$ has been split in the same way. Therefore, the two configurations differ because, in at least one subpath, a different index organization has been chosen for $\chi_h(\mathcal{P})$ with respect to $\chi_y(\mathcal{P})$. Let assume that they differ in only one subpath (the proof can easily be extended to the case of several subpaths). That is $T_j \in \chi_h(\mathcal{P})$ and $T_{j'} \in \chi_y(\mathcal{P})$ exist such that

$$j = j' \text{ and } S_j = S_{j'} \text{ and } IT_j \neq IT_{j'}.$$

that is S_j is identical to $S_{j'}$, but different index organizations have been chosen in the two configurations for these subpaths.

Since the algorithm considers all subpaths of length l at the l-th step, subpath S_j has been considered at the $(l_j + 1)$-th step (cf. Definition 8 in Section 2). Suppose that $IT_j = NX$, and $IT_{j'} = PX$. Since the organization chosen for S_j is the nested index, we have that

$$cost\_NX \leq cost\_PX \ for \ subpath \ S_j \quad (5).$$

We also have that $cost_h(S_j) = cost\_NX$ and $cost_y(S_j) = cost\_PX$, where $cost_h(S_j)$ ($cost_y(S_j)$) denotes the cost of subpath S_j under configuration χ_h (χ_y).

Recalling the definition of the cost of a configuration, we have that:

$$ov\_cost(\chi_y(\mathcal{P})) = \sum_{i=1}^{m} cost_y(S_i)$$

$$ov\_cost(\chi_h(\mathcal{P}) = \sum_{i=1}^{m} cost_h(S_i)$$

Therefore, expression (4) is expanded as follows:

$$\sum_{i=1}^{m} cost_y(S_i) < \sum_{i=1}^{m} cost_h(S_i)$$

The previous expression can be expanded as follows:

$$\sum_{i=1, i \neq j'}^{m} cost_y(S_i) + cost_y(S_{j'}) < \sum_{i=1, i \neq j}^{m} cost_h(S_i) + cost_h(S_j)$$

Since j=j' and $cost_h(S_i)$=$cost_y(S_i)$ for $i \neq j$, we can rewrite the previous expression as follows:

$$\sum_{i=1, i \neq j}^{m} cost(S_i) + cost_y(S_j) < \sum_{i=1, i \neq j}^{m} cost(S_i) + cost_h(S_j)$$

Therefore, we obtain:

$$cost_y(S_j) < cost_h(S_j)$$

and then, by substituting expressions for $cost_y(S_j)$ and $cost_h(S_j)$, we obtain that

$$cost\_PX < cost\_NX \ for \ subpath \ S_j.$$

Since this contradicts expression (5), our thesis follows.

Case 2. $\Sigma_h \neq \Sigma_y$. Let us assume that

$$\Sigma_h = \{S_1, S_2, \ldots, S_m\} \text{ and } \Sigma_y = \{S_1, S_2, \ldots, S_r\}.$$

Because we are considering split configurations, both m and r are greater than 1. Also note that

$\mathcal{P} = cat[S_1, S_2, \ldots, S_m]$ and similarly
$\mathcal{P} = cat[S_1, S_2, \ldots, S_r]$.

Given Σ_h we observe that we can define two subpaths S_h and S'_h as follows:
$S_h = cat[S_1, S_2, \ldots, S_{m-1}]$ and $S'_h = S_m$, such that $\mathcal{P} = cat[S_h, S'_h]$.
$cost(S_h) = \sum_{i=1}^{m-1} cost(S_i)$ and $cost(S'_h) = cost(S_m)$.

Given Σ_y it is always possible to define two subpaths S_y and S'_y as follows:
$S_y = cat[S_1, S_2, \ldots, S_{r-1}]$ and $S'_y = S_r$, such that $\mathcal{P} = cat[S_y, S'_y]$.
$cost(S_y) = \sum_{i=1}^{r-1} cost(S_i)$ and $cost(S'_y) = cost(S_r)$.

Note that l_r is the length of subpath S_r and therefore, is the length of subpath S'_y and $1 \leq l_r \leq n - 1$. In fact l_r cannot be equal to n because $\chi(\mathcal{P})$ is a split configuration. Therefore the length of S_y is $(n - l_r)$ and the starting class of S_y is C_1. Therefore, $S_y = S_1^{n-l_r}$ and $S'_y = S_{1+n-l_r}^{l_r}$. The algorithm evaluates at step n the following expressions:

$$cost(S_h) + cost(S'_h) \ (7) \ and$$

$$cost(S_y) + cost(S'_y) \ (8)$$

Since $\chi_h(\mathcal{P})$ has been chosen by the algorithm, this means that

$$cost(S_h) + cost(S'_h) \leq cost(S_1^{n-l}) + cost(S_{1+n-l}^{l}) \ l = 1, \ldots, n-1 \ (9).$$

In fact the algorithm at step n considers all possible partitions of $\mathcal{P}$ into two subpaths. Expression (9) holds in particular for $l = l_r$. Therefore, we obtain

$$cost(S_h) + cost(S'_h) \leq cost(S_1^{n-l_r}) + cost(S_{1+n-l_r}^{l_r})$$

and thus

$$cost(S_h) + cost(S'_h) \leq cost(S_y) + cost(S'_y) \ (10).$$

By using equalities (7) and (8) the inequality (4) can be expressed as follows:

$$cost(S_y) + cost(S'_y) < cost(S_h) + cost(S'_h) \quad (11)$$

Since expression (11) is in contradiction with expression (10) the thesis follows.

Proposition 3. Given a path $\mathcal{P}$ of length *n*, the algorithm determines a non-trivial configuration $\chi_i(\mathcal{P})$, $\chi_i(\mathcal{P}) \in \Theta(\mathcal{P})$ such that $ov\_cost(\chi_i(\mathcal{P})) \leq ov\_cost(\chi_j(\mathcal{P}))$, $\forall \chi_j(\mathcal{P}) \in \Theta(\mathcal{P})$.

Proof. The optimal index configuration is the configuration having the lowest cost among the nested index (if applicable), the path index, the optimal split configuration, and the configuration without index. At step *n* the algorithm determines the optimal split configuration (by the previous proposition). Then the algorithm compares the cost of the nested index (if applicable), the cost of the path index, the cost of the no-index configuration, and the cost of the selected split organization. Therefore at step *n* the algorithm determines the optimal configuration. The algorithm determines also the non-trivial configuration. This is explained by observing that at step *n*, we always consider a configuration of the form $\{<C_1.\ A_1.\ A_2.....\ A_n,\ \theta>\}$. Therefore, even if among the split configurations, the selected one has been a trivial one (e.g., $\{<C_1.\ A_2....A_i,\ \theta>, <C_{i+1}.\ A_{i+1}.....\ A_n,\ \theta>\}$), the algorithm always considers the equivalent a non-trivial one. When the resulting cost of a trivial configuration is equal to the cost of the non-trivial one, the algorithm always selects the non-trivial configuration (cf. with the choices at the end of STEPS 2....*n*). □