Optimization of Object-Oriented Queries Using Path Indices

E.Bertino and C.Guglielmina
Dipartimento di Matematica - Universita' di Genova
Via L.B. Alberti 4, 16132 Genova (Italy)
e-mail: bertino@igecuniv.bitnet

Abstract

This paper addresses the problem of efficiently evaluating nested predicates in object-oriented databases. In a previous paper [Bert 89] we have introduced the notion of path index that associates the values of a nested attribute with the instances of the class root of a given aggregation hierarchy. In that paper we have evaluated the performance of the path index in the case of queries containing a single predicate. In the present paper we consider the usage of the path index in the framework of more general queries containing several predicates.

1 Introduction

Object-oriented programming languages imply navigational access to objects. However, this capability alone is not adequate for applications which must deal with a large number of objects. Therefore advanced object-oriented DBMSs provide associative query capabilities, in addition to navigational access to objects [Bert 91]. Because of the nested object structures, object-oriented query languages allow restrictions on objects based on predicates on both nested and non-nested attributes of classes. An example of a query, against the aggregation hierarchy of figure 1, is the following:

Q1 = Retrieve all red vehicles manufactured by Fiat Company.

In this query there is a predicate against the attribute Color and a predicate against the nested attribute Name. We will refer to a predicate defined on a nested attribute to as *nested predicate*. In order to support query predicates on a class nested attribute, object-oriented query languages usually provide some forms of *path-expression*. A path-expression specifies an implicit join between an object O and an object referenced by O. Therefore in object-oriented query languages it is useful to distinguish between the *implicit join*, deriving from 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, for example the language defined in [Bane 88], only support implicit join. The motivation for this is based on the argument that in relational systems joins are mostly used to recompose entities that were decomposed for normalization and to support relationships among entities. In object-oriented data models there is no need of normalizing objects, since these models directly support complex objects. Moreover, relationships among entities are supported through object references; thus the same function of joins as used in the relational model to support relationships is provided more naturally by path-expressions. It, therefore, 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 'Vehicle.Manufacturer.Name' denoting the nested attribute 'Name' of class Vehicle. The evaluation of a query with nested predicates may cause the traversal of objects along aggregation hierarchies. Examples of strategies for evaluations of these queries can be found in [Bert 90, Kim 90].

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 precompute these joins and to define specialized indexing techniques supporting fast traversal of aggregation hierarchies. The concepts of nested index and path index have been proposed in [Bert 89] as access mechanisms to provide efficient support for queries involving implicit joins. [Kemp 90] proposes a technique, called *access support relation*, which is equivalent to the path index defined in [Bert 89]. Moreover, the

0-8186-2660-7/92 $3.00 © 1992 IEEE

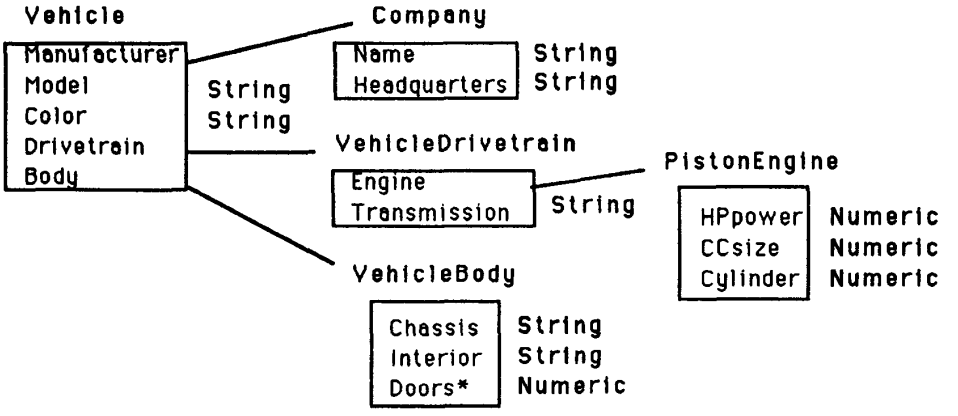

Figure 1: An example of database schema

usage of nested index in the framework of complex queries (containing Boolean combinations of several predicates) has been investigated and some results are reported in [Bert 90]. By contrast, no similar analysis has been reported concerning the path index organization, since both [Bert 89] and [Kemp 90] evaluate the performance of the path index in the case of simple queries, that is, queries containing a single nested predicate. In the present paper we provide the same analysis carried in [Bert 90] but with the usage of the path index. In particular we will point out the types of queries for which a path index is more suited. The goals of this analysis are the description of the new query strategies arising from the usage of path indices, and the derivation of indications for index allocation in an object oriented database.

The remainder of this paper is organized as follows. Section 2 reviews the path index organization, and introduces some new query processing strategies that take advantage of the features of this new organization. Section 3 presents some results concerning the usage of the path index when used in complex queries.

2 Path index

In this Section, we first present some definitions, most of them from [Bert 89], that are needed for the subsequent discussion. Then we describe the concepts of query processing strategies.

Definition 1. Given an aggregation hierarchy H a ***path*** $\mathcal{P}$ is defined as $C_1.A_1.A_2.\ldots.A_n$ $(n \geq 1)$ where:

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

$\text{len}(\mathcal{P}) = n$ denotes the length of the path;
$\text{class}(\mathcal{P}) = C_1 \bigcup \{C_i \ / \ C_i$ is domain of attribute A_{i-1} of class C_{i-1}, $1 < i \leq n\}$ denotes the set of the classes along the path. The number of classes along the path is equal to the path length;
$\text{dom}(\mathcal{P})$ denotes the class C domain of attribute A_n of class C_n.

Definition 2. Given a path $\mathcal{P}$=$C_1.A_1.A_2.\ldots.A_n$, a *complete instantiation* of $\mathcal{P}$ is defined as a sequence of $n+1$ objects, denoted as $O_1.O_2.\ldots.O_{n+1}$ where:

- O_1 is an instance of class C_1;
- O_i is the value of attribute A_{i-1} of object O_{i-1} (i.e. $O_{i-1}.A_{i-1} = O_i$ if A_{i-1} is single-valued; $O_i \in O_{i-1}.A_{i-1}$ otherwise), $1 < i \leq n+1$.

A complete path instantiation is such that it contains an instance of each class along the path and it terminates with an instance of the class domain of the path.

Definition 3. Given a path $\mathcal{P}$=$C_1.A_1.A_2.\ldots.A_n$, a *partial instantiation* of $\mathcal{P}$ is defined as a sequence of objects $O_1.O_2.\ldots.O_j$, $j < n+1$ where:

- O_1 is an instance of a class C_k in $\text{class}(\mathcal{P})$ such that $k+j-1 = n+1$;
- O_i is the value of attribute A_{i-1} of an object O_{i-1}, $1 < i \leq j$.

Definition 4. Given a partial instantiation $p = O_1.O_2.\ldots.O_j$ of $\mathcal{P}$, p is *non-redundant* if no instantiation (either complete or partial) $p' = O'_1.O'_2.\ldots.O'_k$ of $\mathcal{P}$, $k > j$ exists such that $O_i = O'_{k-j+i} (i = 1, \ldots, j)$.

An instantiation is redundant if it is contained in another, longer instantiation.

Definition 5. Given a path $\mathcal{P}$= $C_1.A_1.A_2.\ldots.A_n$, a *path index* on $\mathcal{P}$ is defined as a set of pairs (O,S)

where

$$S = \{\pi_{<j-1>}(p_i) \text{ such that :}$$

1. $p_i = O_1.O_2.....O_j$ $(1 \leq j \leq n+1)$ is either a complete instantiation or a non-redundant partial instantiation of $\mathcal{P}$;
2. $O_j = O\}$.

The first element of the pair (O,S) is the index key.

Referring to an hypothetical database of the example schema shown in figure 1, examples of entries for the path index on the path P=Vehicle.Manufacturer.Name are:
(Renault,{Vehicle[k].Company[i]})
(Fiat,{Vehicle[i].Company[j],Vehicle[j].Company[j]}),
where 'Class'[i] stands for the i-th instance of the class with name 'Class'. The path index can be seen as a generalization of the join index introduced in [Vald 87].

In the remainder on the discussion, we will briefly refer to the nested index organization, defined in [Bert 89], of which we will give an informal definition. Given a path in an aggregation hierarchy, a nested index provides a direct association between the objects (or values) at the end of the path and the object instances of the class which is the root of the path. For example a nested index on the previous path and for the same hypothetical objects shown in the previous example will contain the entries:
(Fiat,{Vehicle[i],Vehicle[j]}) (Renault,{Vehicle[k]}).
Therefore, the difference of the nested index with respect to the path index is that in the former only the object at the beginning of the path is kept in the index, while in the latter all objects along the path are kept.

Index structure and operations

The data structure that we will use to model the various index organizations is a B^+-tree. The format of the non-leaf node is identical in all the index 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 a leaf-node record in a path index 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. The 1-st element of the array is the OID of an instance of class C_1, the 2-nd element is the OID of the instance of class C_2 referenced by attribute A_1 of the instance identified by the 1-st element of the array, and so on.

Query processing strategies

In order to analyze the performance of path indices in the framework of query processing, we summarize here the main topics of query processing and execution strategies. A query can be conveniently represented by a *query graph* [Kim 90]. The query execution strategies vary along two dimensions. The first dimension concerns the method used to traverse the query graph. The second dimension is the technique used to retrieve data from the classes that are traversed for the evaluation of nested predicates. We consider two traversal methods [Kim 90]:

- *Forward traversal*: the first class visited is the target class of the query (root of the query graph). The remaining classes are traversed starting from the target class in any depth-first order. The forward traversal strategy for query Q1 is (Vehicle Company).
- *Reverse traversal*: the traversal of the query graph begins at the leaves and proceeds bottom-up along the graph. The reverse traversal strategy for query Q1 is (Company Vehicle).

A concept orthogonal is that of accessing instances from a given class. Two methods are considered for retrieving data from a visited class [Kim 90]. The first method is called *nested-loop* and consists of instantiating separately each qualified instance of a class. The instance attributes are examined for qualification, if there are simple predicates on the instance attributes. If the instance qualifies, it is passed to its parent node (in the case of reverse traversal) or to its child node (in case of forward traversal). The second method is called *sort-domain* and consists of instantiating all qualified instances of a class at once. Then all qualifying instances are passed to their parent or child node (depending on the traversal strategy used). By combining the graph traversal strategies with the retrieval strategies, we obtain four basic query execution strategies, namely nested-loop forward traversal (NLFT), nested-loop reverse traversal (NLRT), sort-domain forward traversal (SDFT), sort-domain reverse traversal (SDRT). When complex queries are concerned these strategies can be combined giving place to more complex strategies.

A query execution strategy is the decomposition of a query into a set of basic operations: index scan (I), nested-loop join (NJ), sort-merge join (SJ), intersection ($\cap$), selection (S), projection (P), sort-order (O) all with the straightforward semantic. The usage of path indices gives rise to a new projection operation, namely *projection on path instantiation* which impacts significantly the definition of new query strategies.

Definition 6. Given a path $\mathcal{P} = C_1.A_1.....A_n$, an instantiation $I = O_1 \cdots O_{n+1}$ of P and a set S of subscripts such that $S \subseteq \{1,\ldots,n\}$, a *path index projection* of I with respect to S is an m-tuple $(oid_{i_1},\ldots,oid_{i_m})$ where m is the cardinality of S and oid_{i_j} is the i_jth object identifier in I.

With this notation the projection of $O_1.O_2.O_3.O_4.O_5$ with respect to {1,2,4} is the tuple (O_1,O_2,O_4). We point out that the most useful projections in the framework of query strategies are the ones containing the first element of each instantiation. This operation differs from the classical relational projection in that the projected attributes are nested attributes instead of simple ones. Thus it is useful only unitedly with a path index, because in order to be executed it does not require more page accesses, while in case when no index is used it would be simulated by a NLFT-like strategy. Its power and usage in the framework of this study will be shown in the next section.

3 Usage of the path index

The usage of indices in general becomes useful in the case of strategies with reverse traversal or mixed traversal. In this section we will present the analysis of query processing strategies using path indices and compare their performance with that of the strategies using other organizations. In particular we will concentrate on:

1. single-target queries;
2. single key predicates, that is predicates of the form *key=value* and not range queries;
3. queries concerning the target class and not the inheritance hierarchy rooted at the class;
4. queries containing several predicates in conjunction.

Note that the analysis can be easily extended, when the restrictions at points 1) 2) and 3) are lifted, by simply taking into account the changes in the cost expressions. Some of the conclusions may change but the structure of the strategies used to evaluate the performance remain the same. As far as restriction 4) is concerned some of the strategies presented here are totally inadequate for queries with predicates in disjunction. Thus further analysis is needed to give conclusions on that case also. Anyway the cases presented here are in fact common cases.

The complete classification of the possible index allocations for a query with two predicates is summarized in figure 2 where we also number the configurations. In the rest of this paper we will refer to that numbering. Note that we include the possibility of splitting a path into two subpaths and of allocating a separate path index on each subpath (cases 6,8, and 9 only for path P_2). In the remainder we report some results of a complete analysis. In particular, we summarize the case of non overlapping paths, which does not present different trends with respect to a similar analysis presented for the nested index in [Bert 90]. By contrast, we will describe more in detail the case of overlapping paths which appear more interesting since it shows the usage of the projection on path instantiation operation introduced earlier in this paper.

3.1 Parameters

Given a path $\mathcal{P} = C_1.A_1 \ldots A_n$ such that the corresponding classes along P are $\{C_1, C_2, \ldots, C_n\}$, we have the following parameters that describe characteristics of the classes and attributes:

- D_i number of distinct values for attribute A_i, $1 \leq i \leq n$. In particular, when $1 \leq i < n$, this parameter defines the number of distinct references from instances of class C_i to instances of class C_{i+1} through attribute A_i.
- N_i cardinality of class C_i, $1 \leq i \leq n$.
- k_i average number of instances of class C_i assuming the same values for attribute A_i; $k_i = \lceil N_i/D_i \rceil$.
- $OIDL$ length of the object-identifier.
- $PC(C_i)$, $1 \leq i \leq n$, number of disk pages containing instances of class C_i.
- NIA cost of access to a nested index. For this cost, we use the formulation presented in [Bert 89].
- PIA cost of access to a path index. Formulation presented in [Bert 89].
- SIA cost of access to a simple index. Formulation presented in [Bert 89].

The parameters k_i $(1 \leq i \leq n)$ model the degree of reference sharing, which impacts the cost most significantly. Two objects share a reference if they reference the same object.

In order to simplify the model, we make a number of assumptions; this is because the aim of the present paper is to develop query execution strategies rather than being exhaustive in terms of model which can always be extended in a second time.

1. Each instance of a class C_i is referenced by instances of class C_{i-1}, $1 < i \leq n$. This implies that

there are no partial instantiations and, from the point of view of the parameters, that $D_i = N_{i+1}$.

2. All key values have the same length. As discussed in [Bert 89], this implies all non-leaf node index records have the same length in all indices.
3. The values of the attributes are uniformly distributed among instances of the class defining the attributes.
4. All attributes are single-valued.
5. The indices are not clustered, meaning that instances are not stored according to the order in which their OID are stored in the leaf-node index records.

The cost functions presented in this study are concentrated on the most significant parameters that is on the degrees of reference sharing k_i and the cardinality of the classes, which mostly affects the cost of accessing the path and simple indices (PIA and SIA).

3.2 Non overlapping paths

The usage of a path index instead of a nested index does not change the query strategies considered in the case of nested indices [Bert 90]. That is because when the paths are not overlapping these two organizations are virtually the same as far as the development of new strategies is concerned. As an example let us consider the case of two paths starting from the same class:

- $\mathcal{P}_1 = C_1.A_1.A_2.....A_n$
- $\mathcal{P}_2 = C_1.A_1'.A_2'.....A_m'$

and a query containing two predicates $pred_n$ and $pred_m$, one for each path.

Let us suppose that a path index is allocated on each path. It should be clear that any query strategy that can be imagined for such situation uses only the starting class and the ending attributes of the paths, and in no way it is possible to profit by the peculiarity of path indices, that is the presence of the whole path instantiations tied to each key value. We are then using a path index as if it were a nested index for which we already have cost formulas and results. The same considerations apply to the case of simple predicates also. The only difference here is that the formulas for the cost of PIA should substitute the cost of NIA. The reader can refer to [Bert 90] for a detailed description of the strategies and cost formulas. Here we only give the graphical representation of the results obtained using path indices instead of nested ones and we summarize the results as follows:

- for cases 1) and 2) (nested predicate and simple predicate):
  - the costs of the strategies using the path index are always worse than the equivalent ones using nested indices, though not significantly [1];
  - the longer the path is the worse the performance of path indices grows; this is easily explained by the intrinsic spatial complexity of path indices;
  - the performance of both nested and path indices with respect to the strategies not using them depends on the degrees k_i as outlined in figure 3.
- for cases 3) (both nested predicates with only one index allocated):
  - when no index is allocated on P_2 and the paths are short (n=m=2) the usage of the path index does not differ from the usage of the nested index;
  - when the paths are longer (n=5, m=6) the cost of path indices grows as shown in figure 4a.
- for case 4) (both nested predicates with both indices allocated):
  as expected the costs of the usage of path indices grows sensibly with respect to the usage of the nested indices as shown in figure 4b.

3.3 Overlapping paths

We concentrate then on the case of queries with predicates on the ending attributes on overlapping paths originating from the same class. More formally given two paths

- $\mathcal{P}_1 = C_1.A_1.A_2.....A_n$
- $\mathcal{P}_2 = C_1.A_1'.A_2'.....A_m'$

they are overlapping if there exists a subscript $j < min(n,m)$ such that $A_i = A_i'$ $(1 \leq i \leq j)$ and $A_i \neq A_i'$ for $i > j$.

We now introduce the notion of splitted configurations. A splitted configuration is one where the path is splitted into several subpaths and an index, either path or nested, is allocated on each subpath. The present work, together with [Bert 90], is interested in giving indications about retrieval costs only for the problem of optimal index configuration on a set of paths. It is then clear that a non splitted organization

[1] Note, however, that path index has a better performance in general for update operations. In this paper, we are mainly concerned with retrieval performance in the case of complex queries, so we will make no other consideration on update costs.

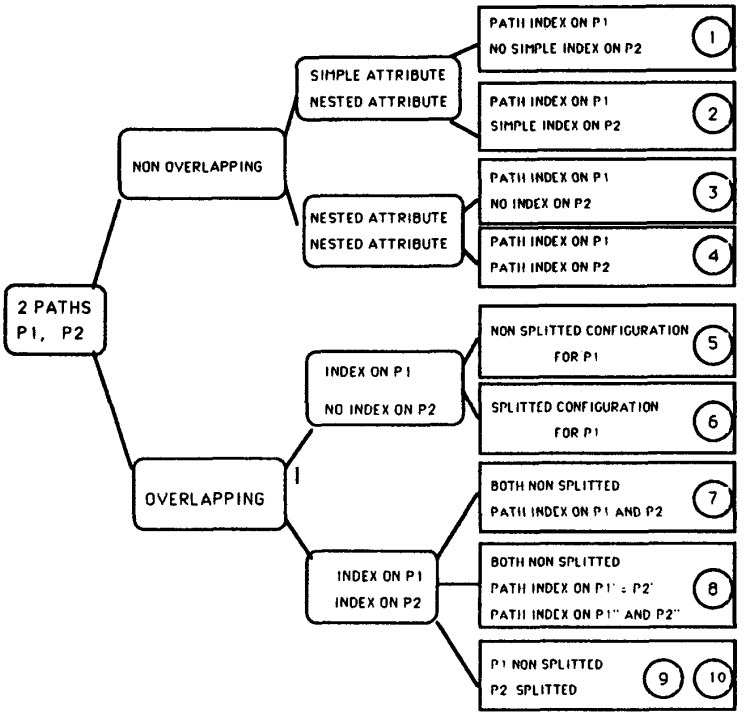

Figure 2: Possible configurations

is more efficient in the case of non overlapping paths, but it can be successful for overlapping paths too. The situation of the paths along the aggregation hierarchy is outlined in figure 5 in which we give the name of the paths we are going to refer to. In the following we will use the notation outlined in figure 5. The splitted configurations considered are then those having C_{i+1} as the splitting class.

3.3.1 Path index on P_1 and no index on P_2: Case 5

When the non splitted configuration is chosen, four different strategies arise. Those strategies are the first example of the use of the new operation which we have already described, the projection that creates tuples of object identifiers. The only difference between those strategies is in the way the classes from C_{i+1} to C_m are visited.

In detail we have:

(A) $pred_n$ is evaluated with an access to the path index defined on P_1; projection operation on the path_instantiations retrieved, creating couples (O_1,O_i); sorting of the second elements of the couples and NLFT from those objects to C_m to evaluate $pred_m$.

(B) the same strategy described in A but the instances from C_i to C_m are retrieved using a SDFT.

(C) the same strategy described in A but the instances from C_i to C_m are retrieved using a NLRT.

(D) the same strategy described in A but the instances from C_i to C_m are retrieved using a SDRT.

As in the case of non overlapping paths we present the cost formulae for the strategies using FT only, the RT being too costly in most of the cases. Moreover we concentrate on NL strategies.

Cost formula:

$$Cost(A) = PIA(P_1) + 2 \times (k_i \times \cdots \times k_n) \times (m - i)$$

3.3.2 Path indices on P_1' and P_1''; no index on P_2: Case 6

The splitted configuration is chosen. The situation becomes the one shown in the figure 5, with indices

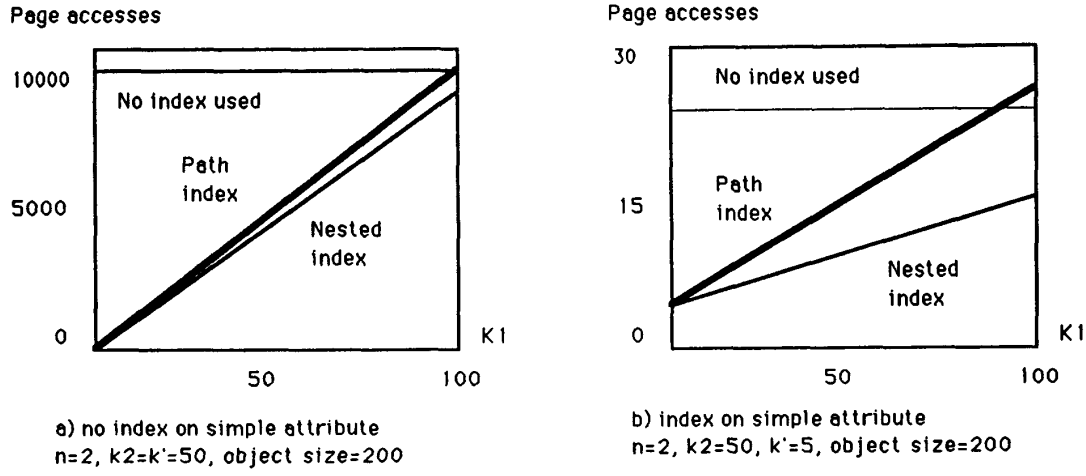

Figure 3: Graphical representation of some cases for non overlapping paths: cases 1) 2)

allocated on paths P_1' and P_1''. Note that one of the properties of the path index is that it directly supports the splitted configuration even if two distinct indices are not allocated. To see this we note that the answer to a query of the kind: *find all objects of class C_i that satisfy $pred_n$ on the nested attribute A_n*, requires a single index lookup and no further page access to follow the references through path P_1'. Anyway here we show the query strategies that arise:

(A) an index access on path P_1''; NLFT from C_i to C_m to evaluate $pred_m$; index access on path P_1' for each retrieved object;

(B) the same but with SDFT;

(C) **(D)** strategies not using the index on P_1'' : NLFT (SDFT) from C_i to C_n and to C_m to evaluate the two predicates; then an access on path P_1' for each qualifying instance;

(E) **(F)** strategies not using the index on P_1' : one access on path P_1''; then NLFT (SDFT) from C_i to C_m to evaluate $pred_n$; then NLFT from C_1 to C_i.

Cost formulae:

- $\text{Cost(A)} = \text{PIA}(P_1'') + 2 \times (k_i \times \cdots \times k_n) \times (m - i) + \lceil (k_i \times \cdots \times k_n)/D_m' \rceil \times \text{PIA}(P_1')$
- $\text{Cost(C)} = 2 \times N_i \times (n - i) + 2 \times (k_i \times \cdots \times k_n) \times (m - i) + \lceil (k_i \times \cdots \times k_n)/D_m' \rceil \times \text{PIA}(P_1')$
- $\text{Cost(E)} = \text{PIA}(P_1'') + 2 \times (k_i \times \cdots \times k_n) \times (m - i) + A_{set}(C_1, A_i, (k_i \times \cdots \times k_n)/D_m')$.
  In this formula $A_{set}(C, A, U)$ denotes the cost of determining which instances of class C assume values for the nested attribute A in a set of OIDs of cardinality U.

From this formulae we obtain the result R3.1:
$\text{Cost(A)} > \text{Cost(C)}$ when
$(k_i \times \cdots \times k_n) > 1000 \times N_i$ (R3.1).
Therefore, with the splitted configuration for one path, when the degrees on the second subpath (P_1'') are high it is better not to use the index on that path, but is is preferable a NLFT instead. We now compare the splitted to the non splitted configuration when the indices are allocated on one path only, in the particular case of NLFT strategies.

The result is shown in figure 6 and it is obtained comparing cost expression Cost(A) to cost expression Cost(A) presented in subsection 3.3.1. It reveals that when D_m', the number of distinct values for attribute A_m', is very low the usage of the non-splitted configuration is preferable to the splitted one.

3.3.3 Path indices on P_1 and P_2: Case 7

In this case the non splitted configuration is chosen for both paths so the situation is the one in figure 5 with indices allocated on paths P_1 and P_2. This case gives rise to the strategies:

(A) the obvious one : accessing the two indices separately and then intersecting the results;

(B) any other strategy falls into the previous cases being the ones that do not use one of the indices.

The only cost expression we give is:
$\text{Cost(A)} = \text{PIA}(P_1) + \text{PIA}(P_2)$.
We now compare the configuration that has indices on

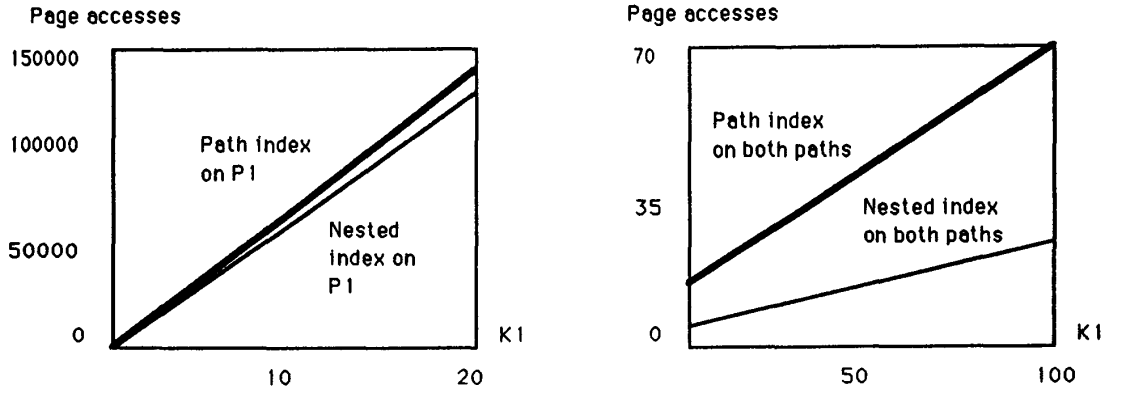

Figure 4: Graphical representation of some cases for non overlapping paths:cases 3) 4)

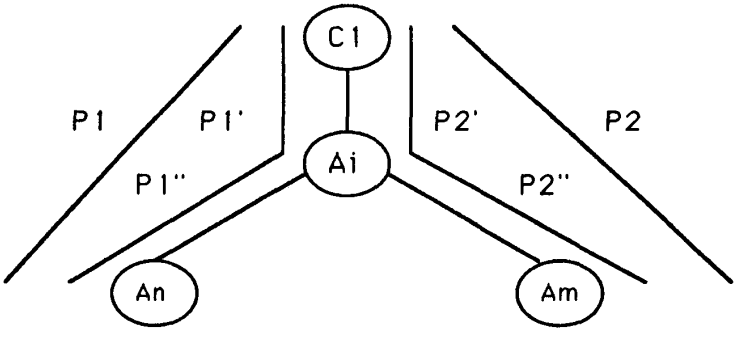

Figure 5: Path schema

both paths (both non-splitted) with the configuration with an index on one path only . Again the result is obtained comparing cost expression Cost(A) of this section to cost expression (A) presented in subsection 3.3.1. We have that the usage of two indices is more convenient with respect to the usage of a single one if: $\frac{k'_1 \times \cdots \times k'_m}{k_1 \times \cdots \times k_n} > 1000 \times \frac{m-i}{m}$ (R3.2)
Note that in this case we do not use the potential of the path indices. Here the usage of the path index is a waste of time and space even if it beats the other strategies in most of the cases, unless heavy update operations on the two paths are foreseen. This comes from the good performance of path indices when updates are concerned.

3.3.4 Path indices on paths P'_1, P''_1 and P''_2: Case 8

Here the splitted configuration is chosen and the first possibility is to have indices allocated on both paths and with both splitted configurations. In particular here the indices on the first two subpaths become the same. Strategies:

(A) the obvious one that is accesses on the indices on paths P''_1 and P''_2 to evaluate $pred_n$ and $pred_m$; then intersection of the retrieved objects; one index access on the index on path P'_1 for each of these objects;

(B) (C) the same of A) but not using the index on P'_1 but performing a NLFT or SDFT from C_1 to C_i instead;

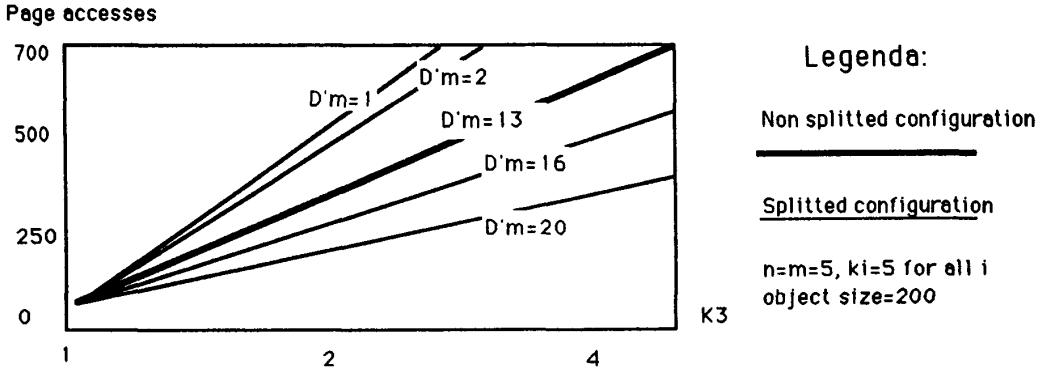

Figure 6: Diagram for the case discussed in section 3.3.2

(D) any other strategy, that is the ones not using the indices on paths P_1'' or P_2'' or both, falls into the strategies presented for case 2).

Costs:

- Cost(A) = PIA(P_1'') + PIA(P_2'') + $\lceil (k_i \times \cdots \times k_n)/D'_m \rceil \times$ PIA(P_1');
- Cost(B) = PIA(P_1'') + PIA(P_2'') + $PC(C_1) + 2 \times N_1 \times (i-1)$;

The comparison between these strategies laeds to the result:

Cost(A) > Cost(B) when $(k_1 \times \cdots \times k_n) > 500 \times D'_m \times [PC(C_1) + 2 \times N_1 \times (i-1)]$. (R3.4)

That is when the degrees of reference sharing along path P_1 are high it is better not to use the index on path P_1', but a NLFT instead.

3.3.5 Path indices on P_1, P_2' and P_2'': Case 9

Another possibility is the one with indices allocated on paths P_1, $P_2'$and P_2'' outlined in figure 5. Here the strategies are:

(A) the straightforward one which uses the indices tout court: one access on the index on path P_2''; one access on the index on path P_2' for each qualifying instance retrieved; one access on the index on path P_1 and then intersection of the results;

(B) a more suited strategy for this case: one access on the index on path P_1; projection that builds the couples (O_1, O_i); one access on the index on path P_2'' and intersection of the results;

(C) any other strategy falls into the previous cases being the ones that do not use indices on one of the paths.

We now present the costs:

- Cost(A) = PIA(P_2'') + $(k_i' \times \cdots \times k_m')$ × PIA(P_2') + PIA(P_1);
- Cost(B) = PIA(P_1) + PIA(P_2'').

Here the first result is obvious and can be formulated as:

Proposition 3.1 *Given two overlapping paths P_1 and P_2, such that the configuration is the non-splitted one for path P_1 (path index on P_1) and the splitted one for path P_2 (path indices on P_2' and P_2''), the cost of strategy B is always lower than the cost of strategy A.*

(Note Strategies A and B referred in the proposition are those presented in this subsection)

It is then a fact that the strategies using the projection are always better than the ones that make use of all the indices allocated. As we will see this result gives rise to a generalization. We now compare the configuration analized in this subsection (indices on both paths one of which splitted) to the configuration presented in subsection 3.3.1 (index on one path only), obtaining:

Cost(indices on 2 paths) > Cost(index on 1 path) when $\frac{k_1' \times \cdots \times k_m'}{k_1 \times \cdots \times k_n} > 1000 \times \frac{m-i}{i-1}$. (R3.3)

In other words this means that generally speaking the more the configuration is rich of indices the better, but when the degrees of reference sharing along P_2, the splitted path, are high with respect to the ones along P_1 it is preferable to have one index only.

We can also give interesting results about the comparison between the usage of indices on the two paths (one of which with a splitted configuration) versus the usage of indices on two paths but both with a non splitted configuration. This comes from the comparison between the cost expression for strategy (A) of subsection 3.3.3 and the cost expression for strategy

(B) of this subsection which can be clearly stated as:

Proposition 3.2 *Given two overlapping paths and two configurations defined for these paths:*

- *the configuration with a non-splitted index allocated on the first path and a splitted index for the second path (Splitted-NonSplitted):*
- *the configuration with non-splitted indices allocated on both paths (NonSplitted)*

the cost of NonSplitted configuration is always greater than the Splitted-NonSplitted one.

This is easily proved observing that P_2'' is a subpath of P_2 and the access cost of an index defined on a subpath is always lower than that of the entire path. It is interesting to note that the last result declares the opposite of the same *nested index* case. This comes from the fact that the usage of the projection saves the time used to access the index on path P_2' necessary when nested indices are used.

3.3.6 Path indices on path P_1 and P_2'': Case 10

From the previous analysis and considerations, and particularly from the ones in section 3.3.4, we can say that there is another configuration that we have not yet proposed which arises naturally only with the usage of path indices. This configuration consists of allocating a path index on P_1 and splitting P_2 into two subpaths P_2' and P_2'' but allocating an index only on P_2''. The cost expressions for the strategies arising for this allocation have already been analized in strategy (B) of section 3.3.5. Since proposition 3.2 states that not using the index on path P_1' is always better than using it, we conclude that the best allocation strategy is the one without an index on that path.

4 Conclusions

In this paper, we have analyzed the path index organization in the framework of complex queries, containing conjunctions of predicates, each one involving an implict join. Note that the evaluations of this indexing technique done in [Bert 89] and [Kemp 90] only considered the case of a query containing a single implicit join. In the paper, we have considered implicit joins over both overlapping and non-overlapping paths. The former are paths having common subpaths, while the latter have no common subpaths. It is now interesting to give a partial and informal generalization of the results obtained in the previous section: that is how to use the results in the framework of queries containing more than two predicates defined on the ending attributes of overlapping paths. Among the results we believe the most interesting result comes from the generalization of the discussion presented in subsection 3.3.6: the best index allocation in most of the cases is the one with one path index allocated on the 'longest path' (a path with which the others have a common subpath) and path indices on the subpaths that are not in common. The suited processing strategy is then to access the long index, procede by projection creating tuples $(O_{i_1},\ldots,O_{i_n})$, and then use the other indices to cut the tuples not satisfying one of the predicates.

References

[Bane 88] Banerjee, J., Kim, W., Kim, H.K. Queries in object-oriented databases. *Proc. of IEEE International Conference on Data Engineering*, Los Angeles (Calif.), February 1988.

[Bert 89] Bertino, E., Kim, W. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Engineering*,Vol.1, No.2 (1989), 196-214.

[Bert 90] Bertino, E. Optimization of queries using nested Indices. *Proc. of the Second International Conference on Extending Database Technology Conference (EDBT90)*, Venice (Italy), March 1990.

[Bert 91] Bertino, E., Martino, L. Object-oriented database management systems: concepts and issues. *Computer* (IEEE Computer Society), Vol.24, No.4 (1991), 33-47.

[Kemp 90] Kemper, A., and Moerkotte, G. Access support in object bases. *Proc. of ACM-SIGMOD Conference on Management of Data*, Atlantic City, N.J., May 1990.

[Kim 90] Kim, W. *Introduction to object-oriented databases*. The MIT Press, Cambridge (Mass.), 1990.

[Vald 87] Valduriez, P. Join indices. *ACM Trans. on Database Systems*, Vol. 12, No.2, (1987), 218-246.