Indexing Techniques for Queries on Nested Objects

ELISA BERTINO, MEMBER, IEEE, AND WON KIM

Abstract—**In relational databases, an attribute of a relation may only have a single primitive value, making it cumbersome to model complex artifacts of interest to a wide variety of applications. An object-oriented and nested relational model of data removes this difficulty by introducing the notion of nested objects, that is, by allowing the value of an object to be another object or a set of other objects. This means that a class (relation) consists of a set of attributes, and the values of the attributes are objects that belong to other classes (relations); that is, the definition of a class (relation) forms a hierarchy of classes (relations). All attributes of the nested classes are nested attributes of the root of the hierarchy. Just as a secondary index on an attribute or a combination of attributes is useful for expediting the evaluation of a query on a relation, a secondary index is useful for evaluating queries on a nested class in an object-oriented database or a nested relation in a nested relational database. In this paper, we introduce three index organizations for use in the evaluation of a query in an object-oriented or nested relational database. We develop detailed models of the three indexes. Using the models, we evaluate the storage cost, retrieval cost, and update cost of these indexes, and make a number of observations about the use of these indexes for evaluating queries for object-oriented or nested relational databases.**

Index Terms—**Access methods, complex objects, database performance and measurement, index selection, nested relations, object-oriented databases, query optimization.**

I. Introduction

The normalized relational model of data restricts the value of an attribute to a single primitive value, such as an integer, a real, a string, or a boolean. Users and researchers of databases have found that this restriction poses a serious problem in modeling complex artifacts, such as a design object or a multimedia document. The current research into object-oriented databases [2], [3], [9], [16] and nested relational databases [1], [7], [8], [11], [17] is partially motivated by the need to remove this problem with the relational model of data. One important element common to object-oriented databases and nested relational databases is the view that the value of an attribute of an object is an object or a set of objects. A class C (or a relation R) consists of a number of attributes, and the value of an attribute A of an object belonging to the class C (or the relation R) is an object or a set of objects belonging to some other class C′ (or a relation R′). The class C′ (relation R′) is called the domain of the attribute A of the class C (or relation R). The class C′ (relation R′) in turn consists of a number of attributes, and their domains are other classes (relations). In other words, in object-oriented databases and nested relational databases, a class or a relation is in general a hierarchy of classes or relations. This hierarchy is called a *class-attribute hierarchy* in [13]. Fig. 1 is an example of a class-attribute hierarchy; the hierarchy is rooted at the class Vehicle, and the * symbol next to an attribute indicates that the value of the attribute is multivalued. An attribute of any class on a class-attribute hierarchy is logically an attribute of the root of the hierarchy, that is, the attribute is a *nested attribute* of the root class. For example, in Fig. 1, the Location attribute of the class Division is a nested attribute of the class Vehicle.

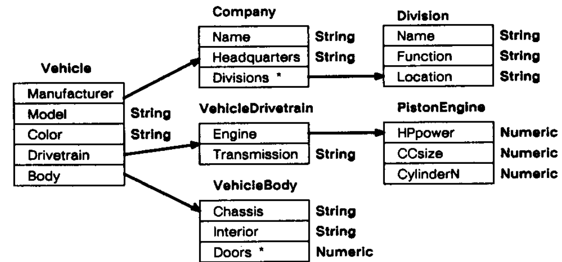


Fig. 1. A class-attribute hierarchy.

In normalized relational databases, the search conditions in a query are expressed as a boolean combination of predicates of the form < attribute operator value >. The query is directed to one or more relations, and the attribute specified in a predicate is an attribute of one of the relations. The nested structure of the definition of a class or a relation in object-oriented or nested relational databases requires an important generalization of the notion of a predicate. In object-oriented or nested relational databases, the search conditions in a query on a class or relation can still be expressed as a boolean combination of predicates of the form < attribute operator value >. However, the attribute may be a nested attribute of the class or the relation. For example, the following is a query that may be issued against the class Vehicle as defined in the class-attribute hierarchy of Fig. 1.

Retrieve all red vehicles manufactured by Fiat.

The query consists of a predicate against the nonnested attribute Color and a predicate against the nested attribute Name. We will call a predicate on a nested attribute a *nested predicate* and a predicate on a nonnested attribute a *simple predicate*. An important issue in supporting queries in object-oriented or nested relational databases is

Manuscript received March 22, 1989; revised May 12, 1989.
The authors are with the Microelectronics and Computer Technology Corporation, Austin, TX 78759.
IEEE Log Number 8930115.

1041-4347/89/0600-0196$01.00 © 1989 IEEE

the efficient evaluation of queries involving nested predicates.

To expedite the evaluation of queries, relational database systems typically provide a secondary index using some variation of the B-tree structure [4], [6] or some hashing technique. An index is maintained on an attribute or a combination of attributes of a relation. Since object-oriented and nested relational databases require the notion of an attribute to be generalized to a nested attribute, the notion of secondary indexing must also be generalized to indexing on a nested attribute. The class (relation) in which the attribute is defined may be different from the class on which the index is maintained.

References [15] and [13] first provided preliminary discussions of the notion of secondary indexing on a sequence of nested attributes. In this paper, we introduce three index organizations for evaluating queries that involve nested predicates, and compare them in terms of storage cost, retrieval cost, and update cost. Largely for presentational simplicity, we will cast the discussions in the context of object-oriented databases; however, the results of this paper are clearly applicable to nested relational databases.

The remainder of this paper is organized as follows. Section II defines some terms and introduces three index organizations for evaluating nested predicates. Section III describes the data structures and operations on the indexes. Section IV summarizes the parameters, and assumptions, of the index models that will be developed in subsequent sections. Sections V–VII develop the models for the storage cost, retrieval cost, and update cost, respectively, and compare the costs for the three indexes. Section VIII provides a comparison of the combined retrieval and update costs for the three indexes. Section IX concludes the paper.

II. Three Index Organizations

In this section, after defining several terms we will use throughout this paper, we will introduce three basic index organizations for evaluating nested predicates.

A. Definitions

Definition 1: Given a class-attribute hierarchy $\mathcal{H}$, 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 $\mathcal{H}$;
- $A(1)$ is an attribute of class $C(1)$;
- $A(i)$ is an attribute of a class $C(i)$ in $\mathcal{H}$, such that $C(i)$ is the domain of attribute $A(i-1)$ of class $C(i-1)$, $1 < i \leq n$;

len $(\mathcal{P}) = n$ denotes the length of path $\mathcal{P}$;

the number of classes along path $\mathcal{P}$ is equal to len $(\mathcal{P})$;

class $(\mathcal{P}) = C(1) \cup \{C(i)$ such that $C(i)$ is the domain of attribute $A(i-1)$ of class $C(i-1)$, $1 < i \leq n\}$.

The path length indicates the number of classes, including $C(1)$, that must be traversed to reach the nested attribute $A(n)$. Note that all classes along a path must be nonprimitive classes, while the domain of the last attribute in the path can be either primitive or nonprimitive.

A class can have several paths starting from it. The following are example paths for the class-attribute hierarchy in Fig. 1:

P1:	Vehicle.Manufacturer.Name	len(P1) = 2
P2:	Vehicle.Manufacturer.Headquarters	len(P2) = 2
P3:	Vehicle.Body.Chassis	len(P3) = 2
P4:	Vehicle.Color	len(P4) = 1.

We assume that each nonprimitive object is identified by an object identifier, which consists of the class identifier of the object class concatenated with the identifier of the object within the class. For example, Vehicle[i] denotes the i-th instance of the class Vehicle. Different object identification schemes are possible, as discussed in [13]. Further, we assume that basic objects are identified by their value.

Definition 2: Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, an *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 $C(1)$; and
- $O(i)$ is the value of the attribute $A(i-1)$ of object $O(i-1)$, $1 < i \leq n+1$.

Note that each object $O(i)$, $1 < i \leq n+1$, in an instantiation is an instance of the class $C(i)$, the domain of attribute $A(i-1)$ in class $C(i-1)$, or of any subclass of $C(i)$.

The objects shown in Fig. 2 are instances of some of the classes shown in Fig. 1. The following are example instantiations of the path P1 = Vehicle.Manufacturer.Name:

- Vehicle[i].Company[j].Renault
- Vehicle[j].Company[j].Renault
- Vehicle[k].Company[i].Fiat.

Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, and an object $O(i)$ of a class $C(i)$ along the path, we will use the term *forward traversal* to denote access of objects $O(i+1), \ldots, O(n)$ such that $O(i+1)$ is referenced by object $O(i)$ through attribute $A(i), \ldots, O(n)$ is referenced by object $O(n-1)$ through attribute $A(n-1)$. Objects on a path may be traversed in a reverse direction of the path, that is, $O(i-1), \ldots, O(2)$, such that $O(i-1)$ references object $O(i)$ through attribute $A(i-1), \ldots, O(1)$ references object $O(2)$ through attribute $A(1)$. It may not be possible to support a *backward traversal*, unless it is possible to directly determine the objects that reference a given object O. For example, in GemStone and ORION a reference from an object to another is unidirectional.

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 class $(\mathcal{P})$ where $k = n - j + 2$; and
- $O(i)$ is the value of the attribute $A(i-1)$ of object $O(i-1)$, $1 < i \leq j$.

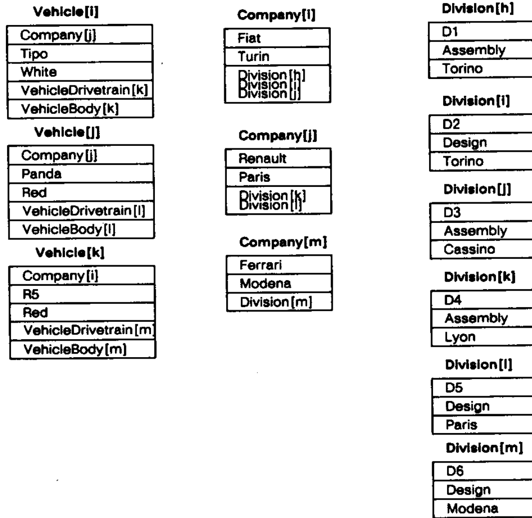


Fig. 2. Instances of classes of Fig. 1.

Examples of a partial instantiation of the path P1 = Vehicle . Manufacturer . Name are:

- Company [m] . Ferrari
- Company [i] . Fiat.

Definition 4: Given a partial instantiation $p = O(1).O(2) \ldots O(j)$ of $\mathcal{P}$, p is *nonredundant* if no instantiation $p' = O'(1).O'(2) \ldots O'(k)$, $k > j$ exists such that $O(i) = O'(k - j + i)$, $1 \leq i \leq j$.

Of the example partial instantiations, Company [m]. Ferrari is nonredundant, since there are no instances of the class Vehicle having Ferrari as manufacturer. Company [i] . Fiat is redundant since there are two instantiations of which Company [j] . Fiat is part.

B. Index Organizations

In the following we will refer to the first object and the last object in an instantiation as the *starting object* and the *ending object*, respectively.

Definition 5: Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, a *nested index* (*NX*) on $\mathcal{P}$ is defined as a set of pairs (O, S), where $S = \{O'$ such that $O(1).O(2) \ldots O(n + 1)$ instantiation of $\mathcal{P}$ exists, where $O' = O(1)$ and $O = O(n + 1)\}$. The first element of the pair (O, S) is the index key.

A nested index provides a direct association between an ending object and corresponding starting objects along a path. A nested index can be implemented using some variation of the B-tree structure or some hashing algorithm.

Example 1: Let us consider the class-attribute hierarchy in Fig. 1. A nested index on the path P1 = Vehicle . Manufacturer . Name will associate a distinct value of the Name attribute with a list of object identifiers of Vehicle whose Manufacturer is an instance of the Company class whose Name is the key value. For the objects shown in Fig. 2, the nested index will contain the following pairs:

- (Fiat, {Vehicle [k]})
- (Renault, {Vehicle [i], Vehicle [j]}).

Definition 6: Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, and $p = O(1).O(2) \ldots O(j)$ instantiation of $\mathcal{P}$, $j \leq n + 1$, $\pi < m > (p) = O(1).O(2) \ldots O(m)$, $m < j$ denotes the projection of p on the first m elements.

For example $\pi < 2 >$(Vehicle [i] . Company [j] . Renault)

= Vehicle [i] . Company [j].

Definition 7: Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, a *path index* (*PX*) on $\mathcal{P}$ is defined as a set of pairs (O, S) where $S = \{\pi < j - 1 > (p(i))$ such that:

- $p(i) = O(1).O(2) \ldots O(j)$ is a nonredundant instantiation (either partial or not) of $\mathcal{P}$;
- $O(j) = O$ (that is O is the ending object of $p(i))\}$.

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

Given an index key, a path index records all instantiations ending with the key; in a nested index only the starting objects of the path instantiations are recorded. Note that when n = 1, the nested index and path index are identical and they reduce to the indexes used in most relational database systems.

Example 2: Let us consider the objects shown in Fig. 2. The path index will contain the following pairs:

- (Fiat, {Vehicle [k] . Company [i]})
- (Renault, {Vehicle [i] . Company [j], Vehicle [j]. Company [j]})
- (Ferrari, {Company [m]}).

Note 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 company with a specified name or to retrieve vehicles whose manufacturer has a specified name.

In both the nested index and the path index, the key values can be instances of a primitive class or a nonprimitive class, depending on the domain of the last attribute in the path. In the latter case, the key values are object identifiers, while in the former case they are primitive values. An index whose key values are object identifiers is called an *identity index* in [16]. The only meaningful operator for an identity index is identity equality [12]. For an index whose keys are primitive values, equality is based on value and additional operators may be meaningful, such as $\leq$ or $\geq$.

A path may be split into several subpaths, and a different index may be used for each subpath. A special case is when each subpath has length 1; in fact, this is the way GemStone supports nested indexing [15]. This is equivalent to the conventional technique of intersecting a combination of separate indexes.

Definition 8: Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, a *multiindex* (*MX*) is defined as a set of n nested indexes NX.1, NX.2, ..., NX.n, where NX.i is a nested index defined on the path C(i).A(i), $1 \leq i \leq n$.

Example 3: For the path P1 = Vehicle . Manufacturer . Name, the multiindex consists of two

indexes. The first index is on the subpath Vehicle . Manufacturer and would contain the following pairs:

- (Company [i], { Vehicle [k] })
- (Company [j], { Vehicle [i], Vehicle [j] }).

The second index is on the subpath Company . Name and would contain the following pairs:

- (Fiat, { Company [i] })
- (Renault, { Company [j] })
- (Ferrari, { Company [m] }).

III. Index Structure and Operations

The data structure that we will use to model the various indexes is based on a B-tree. A B-tree organization is used, for example, in ORION [14] and in the relational system SQL/DS [10].

A. Index Structure

The format of the nonleaf node is identical in all the indexes. A nonleaf 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 fanout, f, is between d and 2d, where d is the order of the B-tree. The fanout of the root node can be between 2 and 2d records.

The format of the leaf node is identical in the nested and multiindexes; it is similar to the leaf node of that used in [14]. An index record in a leaf node consists of the record-length, key-length, key-value, the number of elements in the list of unique identifiers (UID's) of the objects which hold the key value in the indexed attribute, and the list of UID's. Fig. 3(a) shows the format of a leaf node. In the case of a nested index the UID's are not those of the objects in the class that directly contains the indexed attribute, but they are of objects that indirectly contain the indexed attribute. We assume that the list of UID's is ordered. Also we assume that when the record size is larger than the page size a directory is kept at the beginning of the record recording for each page storing the record, the page address, and the highest UID stored in that page. In this way, when deleting or adding a UID in the record, it is possible to determine directly the page to be updated.

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 instantiations having as ending object the key value, and the list of instantiations. Each instantiation is implemented as an array of dimension equal to the number of classes along the path. Therefore, given a path $\mathcal{P}$ = C(1).A(1).A(2) . . . A(n), if C(1), C(2) . . . , C(n) are the classes along the path, the number of elements in the array is n. The 1st element in the array is the UID of an instance of class C(1), the 2nd element is the UID of the instance of class C(2) referenced by attribute A(1) of the instance identified by the 1st element of the array, and so on. Fig. 3(b) shows the format of a leaf node. Fig. 4 provides examples of leaf-node records in a path index for the objects shown in Fig. 2. We assume that when the record size is larger than the page size, the elements of the array are ordered and a directory is kept at the beginning of the record.

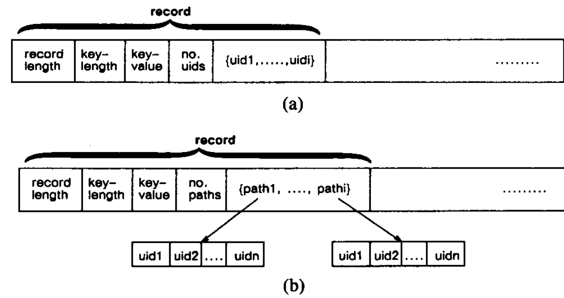


Fig. 3. (a) A leaf node in a nested or multiindex. (b) A leaf node in a path index.

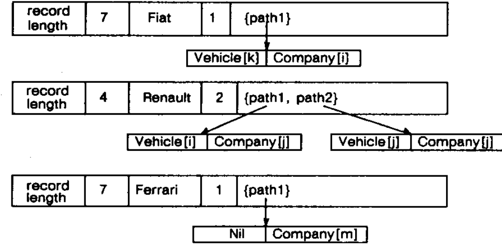


Fig. 4. Leaf-node records in a path index.

B. Index Operations

1) Nested Index: Given a path $\mathcal{P}$ = C(1).A(1).A(2) . . . 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 a lookup of a single index. Therefore, the cost of evaluating a nested predicate is the same as if the attribute A(n) were a direct attribute of class C(1). This index also has the lowest storage overhead.

However, updates to a nested index are costly. Let us consider a path $\mathcal{P}$ = C(1).A(1).A(2) . . . A(n), and an object O(i), $1 \leq i \leq n$, which is an instance of a class C(i) along the path. Suppose that O(i) has an object O(i + 1) as the value of the attribute A(i) and that O(i) is updated to assign to A(i) a new object O′(i + 1). An update to a nested index proceeds as follows.

1) Determine the set *S.old* of the value(s) of the nested attribute A(n) with respect to O(i + 1); a forward traversal of the path is done starting from object O(i + 1). Usually *S.old* contains only one element, unless some attribute along the path after A(i) is multivalued.

2) Determine the set *S.new* of the values(s) of the nested attribute A(n) with respect to O′(i + 1); to do this, the path must be forward traversed from object O′(i + 1). If *S.old* = *S.new*, no updates to the index are required. Otherwise, step 3) is executed.

3) Determine the set *R* of the UID(s) of object(s) of class C(1) that contains direct or indirect references to

O(i); the path is reverse traversed from object O(i). If i = 1, $\boldsymbol{R} = \{O(i)\}$.

4) The index must be updated as follows:

i) if $\boldsymbol{S}.new \subset \boldsymbol{S}.old$, then $\Delta = \boldsymbol{S}.old - \boldsymbol{S}.new$; for each key value K in Δ, the leaf-node record of K is updated by eliminating from the set of UID's the UID's in the set $\boldsymbol{R}$.

ii) if $\boldsymbol{S}.old \subset \boldsymbol{S}.new$, then $\Delta = \boldsymbol{S}.new - \boldsymbol{S}.old$ for each key value K in Δ, the leaf-node record of K is updated by adding to the set of UID's the UID's in the set $\boldsymbol{R}$.

iii) otherwise $\Delta.1 = \boldsymbol{S}.old - \boldsymbol{S}.new$ and $\Delta.2 = \boldsymbol{S}.new - \boldsymbol{S}.old$; for each key value K in $\Delta.1$, the leaf-node record of K is updated by eliminating from the set of UID's the UID's in the set $\boldsymbol{R}$; and for each key value K in $\Delta.2$, the leaf-node record of K is updated by adding to the set of UID's the UID's in the set $\boldsymbol{R}$.

Let us consider the path P1 = Vehicle.Manufacturer.Name and the nested index of Example 1. Suppose that Vehicle[i] changes the Manufacturer from Company[j] to Company[i]. We need two forward traversals of length 1, but no backward traversal. We have to access the object Company[j] and determine the value of the attribute Name, which yields $\boldsymbol{S}.old = \{\text{Renault}\}$. Then we have to access the object Company[i] and determine the value of the attribute Name, which yields $\boldsymbol{S}.new = \{\text{Fiat}\}$. $\boldsymbol{R} = \{\text{Vehicle}[i]\}$. Then updates to the nested index involve the following:

- Vehicle[i] is eliminated from the set of UID's associated with the key value Renault;
- Vehicle[i] is added to the set of UID's associated with the key value Fiat.

If O(i) is the updated object, the forward traversal has length $lf = n - i$, where n is the path length. The backward traversal has length $lb = i - 2$ if $i > 2$, and $lb = 0$ if $i \leq 2$. If no reverse references are kept in objects, the nested index cannot be used, unless updates are very infrequent. If objects along the path are clustered, the number of accesses to secondary storage may be lower than the number of object accesses.

Note that when the updated object belongs to the last class along the path (i.e., i = n) and the key values are unique with respect to the instances of class C(n) [i.e., there are no two instances of C(n) having the same values for A(n)], no forward or backward path traversal is needed. Suppose that all instances of the class Company have different names, and that the name of Company[i] is changed from Fiat to Toyota. The index update is performed simply by looking up the key value Fiat; saving the list of UID's associated with this key; removing this key from the index; and finally inserting the new name, associating with it the saved list of UID's. In this case, the update cost is the same as if the index were not nested, since there are no additional costs due to object traversal.

Insertion and deletion are similar to update, except that they require only one forward traversal.

2) 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), $1 \leq i \leq n$, takes a lookup of a single index. Once the set of 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 pages 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.

Again, let us assume that an object O(i), $1 \leq i \leq n$, which is an instance of a class C(i) along the path, is updated by replacing object O(i + 1), the value of attribute A(i), with a new object O'(i + 1). To update the index, the following steps must be performed.

1) Determine a set $\boldsymbol{S}.old$ of the value(s) of the nested attribute A(n) with respect to O(i + 1); the path is forward traversed from object O(i + 1). $\boldsymbol{S}.old$ contains only one element, unless some attribute along the path after A(i) is multivalued.

2) Determine a set $\boldsymbol{SP}.new$ of partial instantiations from O'(i + 1) to the attribute A(n), and a set $\boldsymbol{S}.new$ of the value(s) of A(n) with respect to O'(i + 1) by a forward traversal of the path from O'(i + 1).

3) For each key value K in $\boldsymbol{S}.old$, access the leaf-node record of K, and for each instantiation p such that the i-th element of the array is equal to O(i) and the (i + 1)-th element of the array is equal to O(i + 1), execute the following steps:

- save in a temporary set $\boldsymbol{T}$ the portion of p from the 1st element to the (i − 1)-th element of the array;
- eliminate p from the set of instantiations associated with K.

4) Generate the new instantiations having O(i) at the i-th position and O'(i + 1) at the (i + 1)-th position. Let us denote the set of new instantiations as $\boldsymbol{P}$. The i-th element of $\boldsymbol{P}$ is generated by concatenating the i-th element in $\boldsymbol{T}$, O(i), O'(i + 1), and the 1st element in $\boldsymbol{SP}.new$; the (i + 1)-th element of $\boldsymbol{P}$ is generated by concatenating the (i + 1)-th element in $\boldsymbol{T}$, O(i), O'(i + 1), and the 1st element in $\boldsymbol{SP}.new$; ... The $(c(1) \times c(2))$-th element of $\boldsymbol{P}$ is generated by concatenating the c(1)-th element in T, O(i), O'(i + 1), the c(2)-th element in $\boldsymbol{SP}.new$, where c(1) and c(2) are the cardinalities of the sets T and $\boldsymbol{SP}.new$, respectively.

5) For each key value K in $\boldsymbol{S}.new$, access the leaf-node record of K and modify the set of instantiations associated with K by adding the instantiations in the set $\boldsymbol{P}$.

Note that unlike the nested index, a path index does not require a backward traversal, since the paths are stored in the leaf-node records. Therefore, this index can be used even if backward references are not kept in objects. Further, when updates are performed on objects of the last class on the path [i.e., C(n)], no forward traversal is needed.

As an example, let us consider a path $\mathcal{P}$ = C(1).A(1).A(2).A(3).A(4), and an instantiation p = O(1).O(2).O(3).O(4).O(5). The set of instantiations in the leaf-node record for the key value O(5) will

contain the instantiation O(1).O(2).O(3).O(4). Let us assume that we modify object O(2) by replacing a reference to object O(3) with a reference to object O'(3); the value of attribute A(3) in object O'(3) is an object O'(4); and the value of attribute A(4) in object O'(4) is an object O'(5). The path index on the path P is updated as follows.

1) Starting from O(3), execute a forward traversal of length 2; that is, access object O(3) and determine O(4), the value of attribute A(3). Then access object O(4) and determine the value of its attribute A(4). We have $S.old = \{O(5)\}$.

2) Starting from O'(3), execute a forward traversal, keeping track of all objects found along the path. We have $S.new = \{O'(5)\}$ and $SP.new = \{O'(4)\}$.

3) Access the leaf-node record with a key value equal to O(5). We have:

- $T = \{O(1)\}$;
- eliminate the instantiation O(1).O(2).O(3).O(4).O(5) from the set of instantiations in this record.

4) Generate the new instantiations by concatenating each element of T [i.e., O(1)], O(2), O'(3), and each element of $SP.new$ [i.e., O'(4)]. We now have

$$P = \{O(1).O(2).O'(3).O'(4)\}.$$

5) Access the leaf-node record with a key value equal to O'(5) and add the instantiations in the set P to the set of instantiations associated with O'(5).

3) Multiindex: Given a path $\mathcal{P}$ = C(1).A(1).A(2) . . . A(n), a multiindex requires a lookup of (n − i + 1) indexes to evaluate a predicate against a nested attribute A(n) of a class C(i), 1 ≤ i ≤ n. As described in [15], the index NX(n) is scanned first and the list of UID's obtained is sorted. Then the index NX(n − 1) is accessed, performing a lookup with the sorted list, and so forth until the index NX(i) is accessed. In particular, to evaluate a predicate with respect to class C(1), n indexes must be accessed. Therefore, the multiindex has lower performance than a nested index. It may have lower or higher performance than the path indexing, depending on the patterns of reference among objects. We will provide a more precise comparison in Section V.

Let us consider the multiindex in Example 3. Suppose that we wish to retrieve all vehicles manufactured by Fiat. The multiindex can be used as follows. First, the index on the subpath Company.Name is accessed looking up for Fiat. The set of UID's retrieved is {Company[i]}. Then, the index on the subpath Vehicle.Manufacturer is accessed for the UID's in the set retrieved by the previous index lookup. Since the set contains only one element, a single index lookup is needed. The lookup of Company[i] in the index on the subpath Vehicle.Manufacturer returns the set of UID's {Vehicle[k]}. This set is the answer to the query.

The primary advantages of multiindexing is the efficiency in updates, since neither forward or backward traversals are required. Suppose that an object O(i), 1 ≤ i ≤ n, which is an instance of a class C(i) along the path, is updated by replacing object O(i + 1), the value of attribute A(i), with an object O'(i + 1). Index NX.i is updated by removing the UID of object O(i) from the set of UID's associated with O(i + 1), and adding it to the set of UID's associated with O'(i + 1).

IV. Parameters of the Index Cost Models

In this section we summarize the parameters used and assumptions we make in the cost models of index we will develop in Sections V–VII.

The parameters that we consider in the model have been grouped as follows. Some of these parameters are provided as input, while others are derived from the input parameters. All lengths and sizes are in bytes.

Logical Data Parameters: Given a path $\mathcal{P}$ = C(1).A(1).A(2) . . . A(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 ≤ i ≤ n. In particular, when 1 ≤ 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 ≤ i ≤ n.
- $k(i)$ Average number of instances of class C(i), assuming the same value for attribute A(i). As pointed out in [14], $k(i) = \lceil N(i)/D(i) \rceil$.
- *UIDL* Length of the object-identifier.

System Parameters:

- *IO* Time needed to fetch a page.
- *P* Page size.

Index Parameters:

- *d* Order of a nonleaf node.
- *f* Average fanout from a nonleaf node. $d \le f \le 2d$, for nonleaf nodes other than the root node. $2 \le f \le 2d$, for the root node.
- *pp* Length of a page pointer.
- *kl* Average length of a key value for the indexed attribute, [i.e., A(n)], for a nested index, for a path index, and for the n-th index in a multiindex.
- *kll* Size of the key-length field.
- *rl* Size of the record-length field.
- *nuid* Size of the no. uids (n. paths) field.
- *ol* Sum of *kll*, *rl*, and *nuid*.
- *L* Average length of a nonleaf-node index record for a nested index, for a path index, and for the n-th index in a multiindex; $L = kl + kll + pp$.
- *L'* Average length of a nonleaf-node index record for the i-th (1 ≤ i ≤ n) index a multiindex; $L' = UIDL + kll + pp$.
- *DS* Length of the directory at the beginning of the record, when the record size is greater than the page size.

TABLE I

$UIDL = 8$	$kl = 8$	$kll = 2$	$rl = 2$	$nuid = 2$	$ol = 6$	$pp = 4$
$L = 14$	$L' = 14$	$f = 218$	$d = 146$	$P = 4096$		

XN Average length of a leaf-node index record for a nested index.
XP Average length of a leaf-node index record for a path index.
$XM(i)$ Average length of a leaf-node index record for the i-th ($1 \leq i \leq n$) index the multiindex.
LP Number of leaf level index pages.
NLP Number of nonleaf level index pages.
np Number of pages occupied by a record when the record size is larger than the page size.

In Sections V–VII we will compare the three indexes with respect to the storage, retrieval, and update costs. The parameters $k(i)$ ($1 \leq i \leq n$) model the degree of reference sharing, which impacts the costs most significantly. Two objects share a reference if they reference the same object.

Assumptions: To simplify our model, we make a number of assumptions.

1) There are no partial instantiations. This implies that $D(i) = N(i+1)$; that is, 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.

2) All key values have the same length. As discussed in [14], this implies that all nonleaf node index records have the same length in all indexes.

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

4) All attributes are single-valued.

As in [14], for some parameters we have adopted values from the B-tree implementation of indexes in ORION. Also we have assumed that the average length of a key value is equal to the size of a UID. The values for these parameters are listed in Table I.

V. Storage Cost

In formulating the storage costs of the three indexes, we use a model similar to that developed in [14]. We first provide the cost formulas for the three indexes, and then compare them using these formulas.

A. Cost Model

1) Nested Index: $k(1, n)$ is the average number of instances of class C(1) having the same value for the nested attribute A(n).

$$k(1, n) = \prod_{i=1}^{n} k(i) = \frac{N(1)}{D(n)}$$

The number of leaf pages is

$$LP = \lceil D(n) / \lfloor P/XN \rfloor \rceil, \text{ where}$$
$$XN = k(1, n) * UIDL + kl + ol \quad \text{if } XN \leq P$$

$$LP = D(n) * \lceil XN/P \rceil, \text{ where}$$
$$XN = k(1, n) * UIDL + kl + ol + DS \text{ and}$$
$$DS = \lceil [k(1, n) * UIDL + kl + ol]/P \rceil * (UIDL + pp) \quad \text{if } XN > P.$$

The estimation of parameters $k(1, n)$ and LP is valid under the assumption that there are no partial instantiations. If there were partial instantiations, this estimation represents the upper bound.

The number of nonleaf pages is evaluated as follows. Let $LO = \min(D(n), LP)$. Then the number of nonleaf pages is

$$NLP = \lceil LO/f \rceil + \lceil \lceil LO/f \rceil / f \rceil + \cdots + X$$

where each term is successively divided by f until the last term X is less than f. If the last term X is not 1, 1 is added to the total (for the root node).

The number of terms in the expression for NLP represents the number of nonleaf nodes that must be accessed when scanning the index. We denote it by h and we will use it in evaluating the retrieval cost in Section VI. The *height* of the index is therefore equal to $h + 1$.

2) Path Index: PN is the average number of path instantiations ending with the same value for the nested attribute A(n).

$$PN = \prod_{i=1}^{n} k(i)$$

The number of leaf pages is

$$LP = \lceil D(n) / \lfloor P/XP \rfloor \rceil, \text{ where}$$
$$XP = PN * UIDL * n + kl + ol \quad \text{if } XP \leq P$$

(each instantation is implemented as an array of UID's of length n)

$$LP = D(n) * \lceil XP/P \rceil, \text{ where}$$
$$XP = PN * UIDL * n + kl + ol + DS \text{ and}$$
$$DS = \lceil [PN * UIDL * n + kl + ol]/P \rceil * (UIDL * n + pp) \quad \text{if } XN > P.$$

The number of nonleaf pages is evaluated as in the previous case.

3) Multiindex: We first express $S(n)$, the storage size of the n-th index in a multiindex. The number of leaf pages in the n-th index is obtained as follows:

$$XM(n) = k(n) * UIDL + kl + ol$$
$$LP(n) = \lceil D(n) / \lfloor P/XM(n) \rfloor \rceil \quad \text{if } XM(n) \leq P$$
$$LP(n) = D(n) * \lceil XM(n)/P \rceil \quad \text{if } XM(n) > P.$$

The number of nonleaf pages is formulated as in the case of the nested index or the path index.

The storage size $S(i)$ of the i-th index ($1 \le i < n$) is given by formulating the number of leaf pages as follows:

$$XM(i) = (k(i) + 1) * UIDL + ol$$

(the key values for the intermediate indexes along the path are object identifiers)

$$LP(i) = \lceil D(i) / \lfloor P/XM(i) \rfloor \rceil \quad \text{if } XM(i) \le P \quad 1 \le i < n$$

$$LP(i) = D(i) * \lceil XM(i)/P \rceil \quad \text{if } XM(i) > P \quad 1 \le i < n.$$

The number of nonleaf pages is expressed as in the other two types of index.

Therefore, the total size of the multiindex is the sum of $S(i)$, $1 \le i \le n$. In the following we will denote by $h(i)$, $1 \le i \le n$, the number of nonleaf pages that must be accessed when scanning the i-th index. It is evaluated as in the previous cases.

B. Comparison

Using the above formulations, we evaluated the index sizes for the three types of index. It is quite obvious that the nested index has the lowest size, independently of the object-reference topology. However, which is the better between the path index and the multiindex depends on the degree of reference sharing among objects.

The cost formulas were evaluated for the case of path length 2 (a path involving two classes). The reason is that for greater path lengths, it is in general preferable to split the path in several subpaths of lengths 1, 2, or 3 and allocate either a nested index or a path index on each subpath. In order to model different degrees of reference among objects, we have fixed the cardinality $N(1)$ to 200 000 and varied the parameters $D(1)$ and $D(2)$ so that the parameter $k(1)$ would assume values in the set {1, 5, 10, 50, 100} and $k(2)$ would assume values in the set {1, 5, 10, 50}. Table II in the Appendix shows the values for these parameters and the computed value of $k(1, 2)$ (significant only for the nested and path indexes). Note that when $k(1)$ and $k(2)$ assume value 1, there is no sharing of references.

We compared the three indexes with respect to different values of parameters $k(i)$. For the storage cost, we note that the path index has a certain degree of redundancy with respect to the multiindex. Some instantiations may share some references. In this case, the referenced objects (and all objects following in the instantiations) are replicated in the arrays implementing the instantiations. This degree of redundancy is modeled by the parameters $k(i)$.

Fig. 5(a) and (b) shows the storage requirements for the three indexes for different values of $k(1)$ and $k(2)$ equal to 1 and 5. The graphs for $k(2)$ equal to 10 and 50 exhibit a similar behavior. It can be seen from the figure that when there is no sharing of references among objects of class

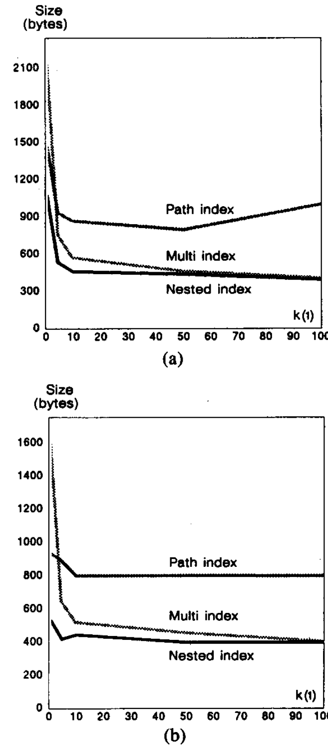


Fig. 5. (a) Storage overhead $k(2) = 1$. (b) Storage overhead $k(2) = 5$.

C(1), the path index has less storage overhead than the multiindex. However, when the degree of reference sharing increases, the same information is replicated several times in the path index, increasing the storage overhead relative to the multiindex. Also note that when the degree of reference sharing increases, the multiindex and the nested index have almost the same storage overhead. This is due to Assumption 1) (cf. Section IV). Under this assumption, an increase in the degree of reference sharing of class C(1) implies that the size of class C(2) decreases.

VI. Retrieval Cost

In this section we compare the retrieval cost of the three types of index, for both single-key queries and range-key queries. A *single-key query* is a query with a single predicate of the form: key = value. A *range-key query* has a single predicate of the form (key < value), or (key > value), or (key between value-1 and value-2).

A. Single-Key Queries

1) Cost Model: For both the nested index and the path index, predicate evaluation requires scanning only one index, while for the multiindex n indexes must be accessed. In the following, we will formulate A, the number of index pages to be accessed to evaluate a single-key predicate.

(1) Nested Index and (2) Path Index: Let X be the size of a leaf-node record.

The number of index pages accessed is

$$A = h + 1 \qquad \text{if } X \leq P$$
$$A = h + \lceil X/P \rceil \qquad \text{if } X > P$$

(h is the number of nonleaf nodes that must be accessed; cf. Section V-A) where $X = XN$ for the nested index, and $X = XP$ for the path index.

(3) Multiindex: We first formulate the number of index pages accessed for the n-th index, and then for the i-th index $1 \leq i < n$.

The number of pages accessed in the n-th index is

$$A(n) = h(n) + 1 \qquad \text{if } XM(n) \leq P$$
$$A = h(n) + \lceil XM(n)/P \rceil \qquad \text{if } XM(n) > P.$$

To determine the number of pages accessed for an i-th index, we have to determine the number of UID's retrieved by a scan of the $(i + 1)$-th index. This number, denoted as $NUID(i)$, is determined as follows:

$$NUID(n - 1) = k(n)$$
$$NUID(i) = k(i + 1) * k(i + 2) * \cdots k(n - 1) * k(n), \qquad 1 \leq i < n - 1.$$

To evaluate the number of leaf pages that must be accessed given $NUID(i)$ UID's, we use a formula developed in [19]. This formula determines the number of pages hit when accessing k records randomly selected from a file containing n records grouped into m pages:

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

Applying this formula, we obtain the number of index leaf pages accessed (denoted by $AL.i$) in scanning the i-th index

$$AL(i) = H(NUID(i), LP(i), D(i)) \qquad \text{if } XM(i) \leq P$$
$$AL(i) = NUID(i) * \lceil XM(i)/P \rceil \qquad \text{if } XM(i) > P.$$

To determine the number of nonleaf pages to be accessed, we can again use the formula of [19] to determine the pages hit at level $h(i)$ in a B-tree, where k is equal to $AL(i)$, m is the number of index pages at level $h(i)$, and n is the number of index records at level $h(i)$. To determine the pages hit at level $h(i) - 1$ in the B-tree, the same reasoning can be applied, until the root is reached. However, to simplify the model we assume that the number of nonleaf pages accessed is equal to $h(i)$. This value is the lower limit. Therefore, the number of pages accessed in the i-th index is

$$A(i) = H(NUID(i), LP(i), D(i)) + h(i) \qquad \text{if } XM(i) \leq P$$
$$A(i) = NUID(i) * \lceil XM(i)/P \rceil + h(i) \qquad \text{if } XM(i) > P.$$

2) Comparison: Using the cost formulas derived above, we computed the number of pages accessed to evaluate a single-key query for the three indexes. The cost formulas were first evaluated for the case of path length 2, and the other parameters assume the same values as those used for the storage cost.

We evaluated the retrieval cost with respect to the parameters $k(i)$, as for the storage cost. The product of these parameters determines the size of the leaf-node record for the nested index and path index, and therefore it may have a nonnegligible impact on the retrieval cost when the size exceeds the page size. For the multiindex the values of the parameters $k(i)$ are even more crucial. In fact, after the $(i + 1)$-th index is scanned, the product $k(n) * k(n - 1) * \cdots * k(i + 1)$ determines the number of UID's that must be looked up in the i-th index. Unless the number of UID's is very large, each of these UID's causes an index lookup. Therefore, the number of index lookups in the i-th index is proportional to this product.

Fig. 6(a) and (b) shows the retrieval costs for the three indexes for different values of $k(1)$, and for values of $k(2)$ equal to 1 and 50. We also evaluated the retrieval costs for values of $k(2)$ equal to 5 and 10. The retrieval cost in the case of $k(2)$ equal to 5 and 10 is the same as the case of $k(2)$ equal to 1 for the nested index, and equal in most cases for the path index, while it goes down for the multiindex.

When the degree of reference sharing of class $C(2)$ is low ($k(2)$ equal to 1), the number of pages accessed for the multiindex is not much higher than that for the other two indexes. However, when $k(2)$ has higher values, the difference increases. This happens because the number of UID's of instances of class $C(2)$ that are retrieved from the second index is given by $k(2)$. For each one of these UID's, the first index must be accessed to determine the UID's of instances of class $C(1)$ associated with that UID. This factor affects significantly the retrieval cost of the multiindex. Therefore, as $k(2)$ increases, the number of accesses to the first index increases; while in the other indexes a single index scan is needed.

Note that the retrieval cost of the nested index and path index is the same for varying values of $k(1)$ and $k(2)$, as long as the record size does not exceed the page size. The retrieval cost of the two indexes becomes different once the record size exceeds the page size; this is because the record size increases faster for the path index than the nested index. Therefore, in Fig. 6(b) we see that for values of $k(1)$ between 10 and 20, the number of page accesses for the nested index is still stable, while it increases for the path index for $k(1)$ greater than 10.

An overall conclusion that can be drawn is that the factor which most significantly affects the retrieval cost of the multiindex is the value of $k(2)$. The access cost increases linearly with the value of $k(2)$. For the other two indexes, the factor that affects the retrieval cost most significantly is the value of the product $k(1) * k(2)$ when it exceeds a certain threshold value. In particular, the ac-

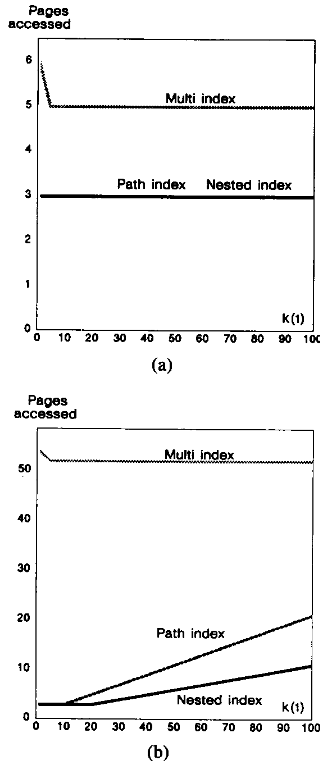


Fig. 6. (a) Performance of single-key queries for $k(2) = 1$ and number of classes = 2. (b) Performance of single-key queries for $k(2) = 50$ and number of classes = 2.

cess cost is constant for

$k(1) * k(2) \leq 1000$ for the nested index; and

$k(1) * k(2) \leq 500$ for the path index.

When $k(1) * k(2)$ is greater than the above thresholds, the access cost increases with the product. However, we note that this increase is less than a linear increase as in the case of the multiindex. The cost for the nested index and path index increases when the leaf-record size is larger than the page size. When record size is smaller than the page size, the access cost is constant.

The above thresholds for the nested index and the path index are explained as follows. For $k(1) * k(2) \leq 500$ (cf. expression for *XN* in Section V-A), the record size in the nested index is smaller than the page size. Therefore, only one leaf page access is needed to access a leaf-node record. For $500 < k(1) * k(2) \leq 1000$, a leaf-node record occupies two pages. However, the index height decreases by 1, offsetting the increase in the number of leaf-page accesses, and keeping the access cost constant. However, when $k(1) * k(2) > 1000$, the leaf node record occupies more than two pages. The decrease in the index height is not enough to offset this, and the cost starts increasing. The same discussion applies to the path index when the record size is greater than the page size for $k(1) * k(2) > 250$.

On the basis of all the experiments we did, we generalize the above discussion to the case of a path length n. As we discussed before, the nested index and the path index have constant access times when the leaf-record size does not exceed a certain threshold value. In particular, on the basis of the expressions given for *XN* and *XP* in Section V-A, we have the following.

1) The nested index has a constant access time, if $k(1) * k(2) * \cdots * k(n) \leq 500$; otherwise the access time grows linearly with $(k(1) * k(2) * \cdots * k(n))/500$.

2) The path index has a constant access time, if $k(1) * k(2) * \cdots * k(n) \leq (500/n)$; otherwise the access time grows linearly with $(k(1) * k(2) * \cdots * k(n))/(500/n)$.

Now let us see how the above threshold is obtained for the nested index. (The derivation for the path index is similar.) Let us consider the expression for *XN* (when *XN* is smaller than the page size) given in Section V-A:

$$XN = k(1) * k(2) * \cdots * k(n) * UIDL + kl + ol.$$

Assuming that the terms *kl* and *ol* are negligible, we have that $XN \leq 4096$ (page size) if

$$k(1) * k(2) * \cdots * k(n) \leq (4096/UIDL).$$

Since *UIDL* is equal to 8, we see that the threshold is 500.

As we discussed in Section VI-A-1), the multiindex requires a lookup of n indexes for a path length n. For the i-th index, the number of keys that must be looked up is given by $k(i+1) * k(i+2) * \cdots * k(n)$. If we consider only the accesses to the leaf pages and assume that for each key a page must be accessed (the real number is slightly smaller and it is given by Yao's formula), we have the following conclusion.

3) The multiindex has an access time that grows linearly with

$$1 + k(n) + k(n-1) * k(n) + k(n-2) * k(n-1) * k(n) + \cdots + k(2) * k(3) * \cdots * k(n-1)\, k(n).$$

B. Range-Key Queries

In formulating range-key queries, we will make use of the following additional parameters:

NREC number of records per leaf-node, if X is the record size then $NREC = \lfloor P/X \rfloor$ if $X \leq P$; $NREC = 1$ otherwise.

NRQ number of key values in the range specified for a given query.

We assume that in a range search, the index is traversed to determine the leaf node containing the lowest value in the range. Then a sequential search is performed on the leaf nodes until the node containing the highest value in the range is determined. That is, we assume that the leaf nodes of the index are linked.

1) Cost Model: As in the case of single-key queries, for both the nested index and the path index, predicate evaluation requires scanning only one index, while for the multiindex n indexes must be accessed.

(1) Nested Index and (2) Path Index: Let X be the size of a leaf-node record.

The number of index pages accessed is

$$A = h + \lceil NRQ/NREC \rceil \quad \text{if } X \leq P$$
$$A = h + NRQ * \lceil X/P \rceil \quad \text{if } X > P$$

where $X = XN$ for the nested index, and $X = XP$ for the path index.

When the record size is smaller than the page size, the expression $\lceil NRQ/NREC \rceil$ determines the number of leaf pages to be accessed to retrieve all records associated with the keys in the range.

(3) Multiindex: The expression for the number of index pages accessed in the n-th index is similar to that for the nested and path indexes

$$A(n) = h(n) + \lceil NRQ/NREC \rceil \quad \text{if } XM(n) \leq P$$
$$A(n) = h(n) + NRQ * \lceil XM(n)/P \rceil \quad \text{if } XM(n) > P.$$

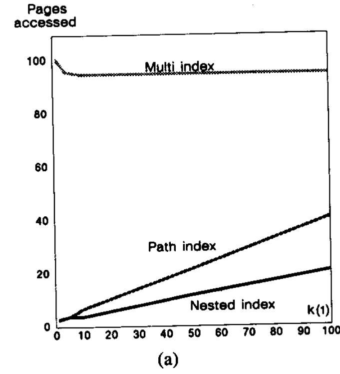
To determine the number of pages accessed for an i-th index, we use the approach used for single-key queries. The number of UID's retrieved by the scan of the (i + 1)-th index is as follows:

$$NUID(n-1) = NRQ * k(n)$$
$$NUID(i) = NRQ * k(n) * k(n-1) * \cdots * k(i+1) \quad 1 \leq i < n-1.$$

In fact, when accessing the n-th index (i.e., the last index), NRQ keys are searched. For each of these keys, $k(n)$ UID's are retrieved. Therefore, when accessing the (n − 1)-th index $NRQ * k(n)$ keys are searched. For each of these keys, $k(n-1)$ UID's are retrieved. Therefore, the total number of UID's retrieved by searching the (n − 1)-th index is given by $NRQ * k(n) * k(n-1)$, which is the number of keys to be searched in the (n − 2)-th index, and so forth, until the 1st index is scanned.

2) Comparison: We studied the case of path length 2, using the same values for the parameters as those used for the storage cost. The parameter NRQ assumes the value 10 or 20.

Fig. 7(a) shows the retrieval cost of the three indexes for different values of $k(1)$, for $k(2)$ equal to 10, and for $NRQ = 10$; while Fig. 7(b) is for $NRQ = 20$. The multiindex has the worst retrieval cost, since the number of UID's retrieved in each index lookup increases in the case of range queries compared to the case of single-key queries. The path index has better retrieval cost than the multiindex, but worse than the nested index. This is due to the greater size of the leaf-node records in the path index compared to that in the nested index.



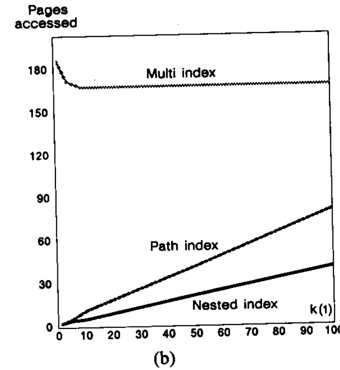


Fig. 7. (a) Performance of range queries for $k(2) = 10$ and $NRQ = 10$. (b) Performance of range queries for $k(2) = 10$ and $NRQ = 20$.

VII. Update Cost

Given a path $\mathcal{P} = C(1).A(1).A(2) \ldots A(n)$, and an object $O(i)$, $1 \leq i \leq n$, an index on the path may have to be updated when the current value of attribute $A(i)$ [i.e., object $O(i+1)$] is replaced with an object $O'(i+1)$. In this section we will derive the cost formulas for update operations. The cost for update, delete, and instance creation will be denoted by U, D, and I, respectively. We will not include the cost of updating, deleting, or creating the object itself, since this cost is common to all indexes. We only account for the cost related to updating the index. Further, to simplify the analysis, as in [18], we do not include the costs due to index page splits; that is, we only consider the cost of leaf-page update.

A. Cost Model

We will use the following additional parameters.

- CFT Cost of a forward traversal.
- CBT Cost of a backward traversal.
- CBM Cost of a B-tree update.

1) Nested Index: We will assume that reverse references are supported. If reverse references are not supported, the nested index cannot be used (if there are updates). We also assume that reverse references from object O to object O′ are stored in O. As discussed already, to update the nested index it is necessary to execute two forward traversals, one backward traversal, and one B-tree update. Of course, the backward traversal and the B-tree

update are necessary only if the value of A(n) for O(i + 1) is different from the value for O′(i + 1). We use *pdiff* to denote the probability that the value of A(n) for O(i + 1) is different from the value for O′(i + 1). Therefore, U is on average

$$U = 2 * CFT + pdiff * (CBT + CBM).$$

To formulate CFT, we use the fact that the number of objects that must be accessed is $n - i$. For each object, the physical address of the object must be determined first, and then the object itself must be fetched. Therefore,

$$CFT = 2 * (n - i).$$

The formulation of CBT is based on the fact that the number of objects that must be accessed in the backward traversal is

$$NO = \sum_{j=2}^{i-1} \left(\prod_{i=j}^{i-1} k(i) \right) \quad \text{if } i > 2$$

$$NO = 0 \quad \text{otherwise.}$$

Since for each object two I/O operations are necessary, we obtain

$$CBT = 2 * NO \quad \text{if } i > 2$$

$$CBT = 0 \quad \text{otherwise.}$$

Note that if the length of the path is two (i.e., $n = 2$) and the updated object belongs to the second class in the path (i.e., $i = 2$), $CFT = 0$ and $CBT = 0$.

The probability *pdiff* is given by the following expression:

$$pdiff = 1 \quad \text{if } i = n;$$

$$pdiff = 1 - \left[\frac{\left(\prod_{j=i+1}^{n} k(j) \right) - 1}{D(i) - 1} \right] \quad \text{if } i < n.$$

The cost of a B-tree update is the cost of removing the UID of object O(i) from the record associated with the old value of A(n) and the cost of adding it to the new value. Therefore, the cost of each of these operations is the cost of finding the leaf node containing the record and the cost of reading and writing the leaf node. We denote the cost of each such operation as CO. The average value of CBM is formulated as follows:

$$CBM = CO * (1 + pl)$$

where pl is the probability that the old and new values of A(n) are on different leaf nodes.

$$CO = h + 2 \quad \text{if } XN \leq P$$

when modifying a leaf page, a page is accessed to read the leaf page containing the updated record, and another to write this page; in addition, h pages are accessed to determine the leaf node containing the record to be updated.

$$CO = h + 2 + (np - 1)/np \quad \text{if } XN > P \quad (1)$$

when the record size is larger than the page size and np is the number of leaf pages needed to store a record ($np = \lceil XN/P \rceil$), $h + 1$ of I/O's are necessary to access the leaf page that contains the header of the record. From the header of the record, it is possible to determine the page from which a UID must be deleted or to which a UID must be added. If this page is different from the page containing the header of the record, a further access must be performed. The probability of this is given by $(np - 1)/np$.

The probability that the current and new value of A(n) are on different leaf nodes is

$$pl = 1 \quad \text{if } XN > P$$

when the record size is larger than the page size, each record is stored in different pages;

$$pl = 1 - \left[(NREC - 1)/(D(n) - 1) \right] \quad \text{if } XN \leq P \quad (2)$$

where $NREC$ is the number of index records per leaf page.

The costs of index updates for object deletion and creation are equal, and are evaluated similarly. The major difference is that only one forward traversal is necessary and the cost of a B-tree update includes only the removal (or insertion) of a UID from a leaf-node record. Therefore, $D = I = CFT + CBT + CBM$, where CFT and CBT are the same as in the case of object update, while $CBM = CO$.

(2) Path Index: As we have seen already, to update a path index it is necessary to execute two forward traversals to determine the values of A(n) for O(i + 1) and for O′(i + 1), and one B-tree update. Note that unlike the previous case, the B-tree update is necessary even if the value of A(n) for O(i + 1) is equal to the value of O′(i + 1). In fact, even if the value of A(n) is the same, we have to modify the leaf-node record associated with A(n) to modify the instantiations affected by the update. However, no backward traversal is necessary. Therefore, $U = 2 * CFT + CBM$.

The expression for CFT is the same as that for the nested index.

The expression for CBM is slightly different from that for the nested index. In fact, two different leaf nodes are updated when the old and new values of A(n) for O(i) are different, and the records associated with the old and new values are stored in different leaf nodes. Therefore,

$$CBM = CO * (1 + pdiff * pl),$$

where CO is the same as expression (1) in which XP is substituted for XN.

The cost of index updates for object deletion and creation is similar to that for the nested index. Therefore, $D = I = CFT + CBM$, where the expressions for CFT and CBM are the same as the nested index.

(3) Multiindex: Only the i-th B-tree must be updated. Therefore,

$$U = CBM = CO * (1 + pl),$$

where CO is given by expression (1) in which $XM(i)$ is substituted for XN, and by expression (2) in which $D(i)$ is substituted for $D(n)$.

The cost of index updates for object deletion and creation is given by

$$D = I = CO.$$

B. Comparison

Using the cost formulas developed in the previous section, we compared the update costs of the three indexes. We considered updates to both C(1) and C(2). Again, we compared the three indexes on the basis of the values of parameters $k(i)$. These parameters are relevant for the nested index and path index for the following reasons. First, the record size depends on the product of parameters $k(i)$, and therefore it may influence the update costs when the record size exceeds the page size. Second, for the nested index these parameters determine how many objects must be accessed in a backward traversal.

Fig. 8(a)–(d) shows the update costs for the case when class C(1) is updated. Note that when the degree of object sharing of class C(2) is low [Fig. 8(a)], the update costs for the nested index and path index are constant. When the value of $k(1) * k(2)$ is equal to 500, the cost of update increases for the path index. This increase results from the increased size of the leaf-node record. The record size in the first index for the multiindex depends only on $k(1)$; when class C(1) is updated, only the first index is updated. In the path index, the record size depends on the product $k(1) * k(2)$, and therefore the cost of update increases in general by 1 when the product increases. Note, however, that in Fig. 8(c) the cost for the nested index and the path index decreases. This happens because the number of nonleaf nodes that must be scanned to determine the records to be updated (parameter h in the cost equations) decreases from 2 ($k(1) < 50$) to 1 ($k(1) > 50$). There is a corresponding increase in the number of leaf pages needed to store a record: for $k(1) \leq 50$ this number is 1 for the nested index and 2 for the path index, and for $k(1) > 50$ it is 2 for the nested index and 3 for the path index. However, for $k(1) > 50$, the probability of an additional page access is 0.5 for the nested index and 0.7 for the path index (cf. expression (1) for CO). As such:

$CO = 4$ for $k(1) < 50$ and $U = 12$ for both the nested index and the path index;

$CO = 3.5$ for $k(1) > 50$ and $U = 11$ for the nested index; and

$CO = 3.7$ for $k(1) > 50$ and $U = 11.4$ for the path index.

However, when the record size increases so that the record occupies more than three pages, the probability of an additional page access starts to increase, offsetting the gain

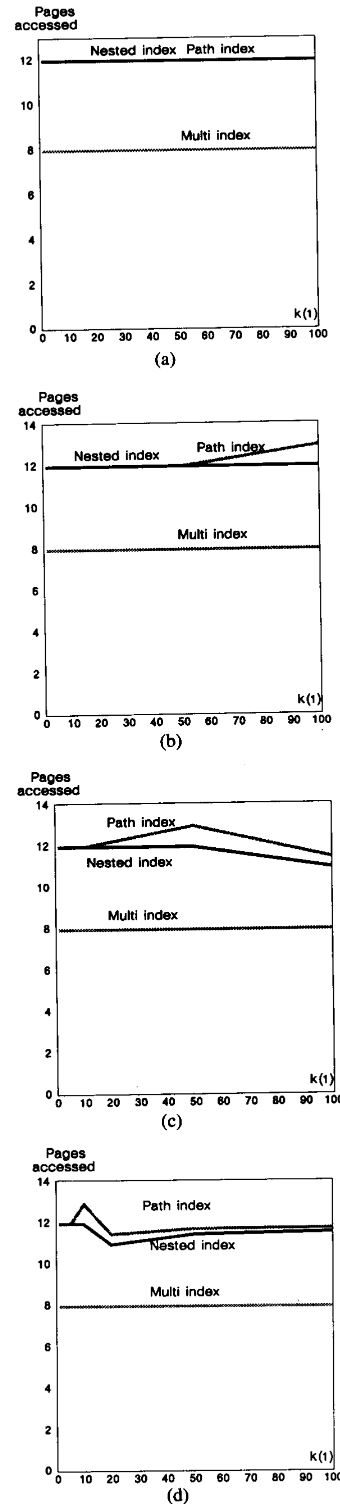


Fig. 8. (a) Update cost $k(2) = 1$, class modified is C(1), number of classes = 2. (b) Update cost $k(2) = 5$, class modified is C(1), number of classes = 2. (c) Update cost $k(2) = 10$, class modified is C(1), number of classes = 2. (d) Update cost $k(2) = 50$, class modified is C(1), number of classes = 2.

due to the reduced tree height. This explains the initial decrease and the subsequent increase in the costs for the nested index and path index in Fig. 8(d). Therefore, for values of the product $k(1) * k(2)$ greater than 1000, we can expect the cost function assumes values that are closer to 12.

When class C(2) is updated, instead of C(1), no forward or backward traversals are needed. When there is no reference sharing ($k(1) = 1$ and $k(2) = 1$), all three indexes have the same update costs. When object sharing increases, the update cost of the multiindex decreases. The decrease in the multiindex is due to the fact that the size of the second index [i.e., the index on class C(2)] decreases as the value of $k(1)$ increases. In fact, when $k(1)$ increases, the cardinality of class C(2) decreases (because we assume no partial instantiations). The update cost for the nested index and path index has a trend similar to that observed for the case of updates on class C(1). The cost is constant, then decreases and then increases again. This is explained in the same way.

An important point is that when the path has length 2 and reverse references are stored in the objects, the nested index can be as effective as the multiindex. In fact, when the indexed attribute A(2) in an object O(2) of the second class is updated, we do not need to access the objects of the first class to determine the objects that reference O, since O contains reverse references to these objects. Therefore, once O is accessed for an update, the reverse references can be determined. When an object O(1) of the first class is updated by assigning to attribute A(1) a different instance of the second class, as we discussed earlier, two forward traversals are necessary to access the objects O′ and O″, the old and new values of A(1), respectively. But these two objects must be accessed anyway to update the reverse references to O(1) (i.e., removing it from O′ and adding to O″). When they are accessed, the value of A(2) can be determined. If a multiindex were used in such a situation, the overhead of accessing objects O′ and O″ would be unavoidable, without the benefit of a faster predicate evaluation.

When the path length is greater than 2, a backward traversal is required for the nested index, thereby increasing the update cost relative to the path index. To assess the effect of the backward traversal, let us consider the case of path length 3. The values for $k(1)$, $k(2)$, and $k(3)$ are the same as those assumed for the cost of single-key queries (Section VI-A). Fig. 9(a)–(d) shows the update costs for some values of $k(2)$ and $k(3)$ when class $C(3)$ (last class in the path) is updated. We can see that for the nested index, the dominant factor is the backward traversal, since $k(3) * k(2)$ objects must be accessed (see the expression for *CBT* in the previous section). For example, when $k(3) = 1$ and $k(2) = 10$ [Fig. 9(b)], ten objects must be accessed. Since for each object two accesses are necessary, the backward traversal costs 20 accesses. Thus, the costs for the nested index are constant in Fig. 9(a) and (b) for fixed combinations of $k(3)$ and $k(2)$, and are almost independent of the values of $k(1)$.

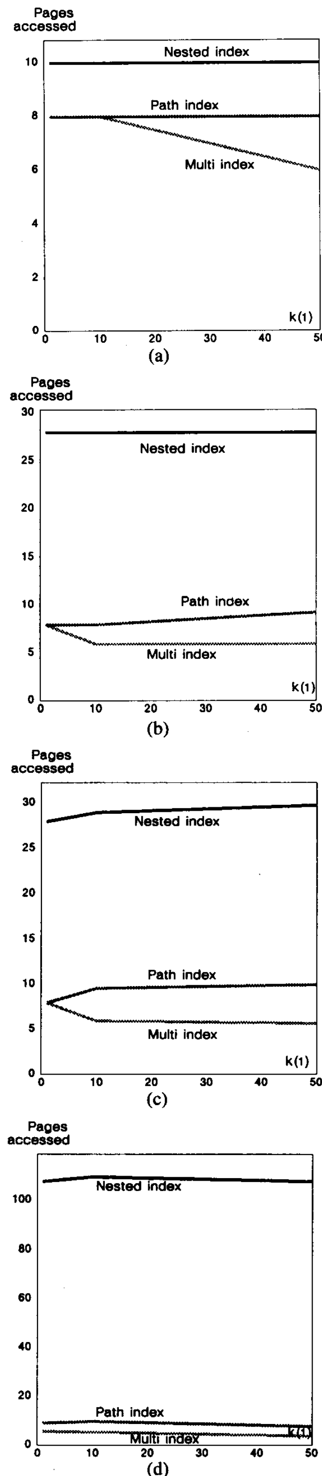


Fig. 9. (a) Update cost $k(2) = 1$, $k(3) = 1$, class modified is C(3), number of classes = 3. (b) Update cost $k(2) = 10$, $k(3) = 1$, class modified is C(3), number of classes = 3. (c) Update cost $k(2) = 10$, $k(3) = 10$, class modified is C(3), number of classes = 3. (d) Update cost $k(2) = 50$, $k(3) = 10$, class modified is C(3), number of classes = 3.

Fig. 9(c) shows that the costs for the nested index also increase for increasing values of $k(1)$. This happens because, for the combinations of values for $k(1)$, $k(2)$, and $k(3)$, the leaf-record size exceeds the page size; therefore, the probability of an additional page access increases. The same consideration explains the increases in the update cost of the path index. The path index incurs a lower cost than the nested index when the third class is updated, because it does not require the backward traversal. However, the costs of updating the classes $C(1)$ or $C(2)$ are almost equal for both the nested index and path index.

An overall conclusion about the update cost for the nested index and the path index is as follows:

- for path length 2, the nested index is preferable, if reverse references are supported;
- for path length 3, the nested index is preferable if reverse references are supported and updates are only on the first or second class in the path; otherwise the path index is preferable.

We now generalize the discussion to the case of path length n to determine the cost trend for updates (the trends for delete and insert are obtained in similar ways). The update cost for the multiindex is almost constant and it is independent of the path length. For the other two indexes, the dominant cost factors are the forward and backward traversals. The costs of forward and backward traversals depend on the class updated.

The update cost for the path index depends on the path length, while for the nested index it depends on the path length and on the product of the parameters $k(i)$. To see this, let us determine the average update cost, denoted as $\mathcal{Y}$, assuming that updates are equally likely on all classes along the path.

The path index requires two forward traversals. From the expression for *CFT*, the average update cost is

$$\mathcal{Y} = \left[4 * \sum_{i=1}^{n} (n - i)\right] \Big/ n = 2 * (n - 1).$$

We see that the average update cost grows linearly with $2 * (n - 1)$.

The nested index requires a backward traversal in addition to the forward traversal. By using the expression for *CBT*, we obtain the average cost of the backward traversal

$$\big[2 * \big(k(n-1) * k(n-2) * \cdots * k(2) + k(n-2) * k(n-3) * \cdots * k(2) + \cdots + k(3) * k(2) + k(2)\big)\big] / n.$$

By adding this cost to the cost of forward traversal which is the same as the path index, we obtain the average update cost as follows:

$$\mathcal{Y} = 2 * (n - 1) + \big[2 * \big(k(n-1) * k(n-2) * \cdots * k(2) + k(n-2) * k(n-3) * \cdots * k(2) + \cdots + k(3) * k(2) + k(2)\big)\big] / n.$$

VIII. Combined Retrieval and Update Cost

In this section we compare the three indexes on the basis of the combined cost of retrieval and update. As we have seen in the previous sections, the nested and path indexes have lower retrieval costs, but higher update costs. The multiindex incurs lower update costs and higher retrieval costs.

To compare the overall costs, we considered the following mix of retrieval (R), update (U), and delete/create operations (D/I). The overall cost is computed as the average of the costs of the three types of operations multiplied by their relative frequencies.

(90%R, 5%U, 5%D/I), (80%R, 10%U, 10%D/I),
(70%R, 15%U, 15%D/I),
(60%R, 20%U, 20%D/I),
(50%R, 25%U, 25%D/I),
(40%R, 30%U, 30%D/I),
(30%R, 35%U, 35%D/I),
(20%R, 40%U, 40%D/I),
(10%R, 45%U, 45%D/I).

For retrieval, we used only single-key queries with predicates on the first class of the path [i.e., $C(1)$]. We assumed that updates apply equally to all classes along the path. For example, if the mix is (80%R, 10%U, 10%D/I) and the number of classes along the path is 2, 5 percent of the update operations and 5 percent of delete/insert operations are performed on each class.

First let us consider a path length 2. The nested index performs better than the other indexes when the retrieval percentage is 60 percent or more. However, the differences between the nested index and the multiindex are not very high for $k(2) = 1$. When the retrieval percentage is between 50 and 30 percent, the nested index costs slightly higher than the multiindex for $k(2) = 1$ and $k(1) > 1$. This is because for these values of $k(1)$ and $k(2)$, the gain in query performance is not high enough to offset the increased update cost. However, for values of $k(2)$ greater than 1, the nested index has a lower overall cost than the other indexes. Finally, when the retrieval percentage is 20 percent or less, the multiindex has the lowest overall cost if $k(2) \leq 10$. When $k(2)$ has values greater than 10, queries become expensive (cf. Section VI), and the lower update cost cannot offset the increased query cost.

A simple rule for determining which index to use in the case of path length 2 when the overall cost must be minimized is the following:

1) choose the multiindex if

$$k(2) = 1 \text{ or } \big(\text{the operations are mainly updates and } k(2) \leq 10\big); \quad \text{otherwise}$$

2) choose the nested index

if there are reverse references from class C(2)

to class C(1); otherwise

3) choose the path index.

This rule is based on the fact that when there is no sharing of references in the second class ($k(2) = 1$), the overall cost of the multiindex is not much higher (only one page access more, even for the case of retrieval 80 percent or more) than the other two indexes, even when the operations are mainly retrieval. In fact, in this case the performance of queries is good (cf. Section VI). However, the retrieval cost increases as the degree of reference sharing for class C(2) increases. Therefore, even when most of the operations are updates, the increased retrieval cost may outweigh the reduced update cost. The nested index is chosen if there are reverse references from the class C(2) to class C(1); otherwise the path index is chosen. If it is not possible to determine the degree of reference sharing and the frequency of retrieval and update operations, a nested index represents a good choice in most cases, provided of course that reverse references are supported.

Next, we consider a path length 3. Fig. 10(a)–(d) shows some selected results for the overall costs for the three mixes for some values of $k(2)$ and $k(3)$. The path index performs better on the average than the other indexes for all access patterns. In fact, the path index incurs a retrieval cost that is comparable to that of the nested index, while it has on the average a lower update cost than the nested index. However, for high values of the product $k(1) * k(2) * k(3)$ the overall cost of the path index increases [see, for example, Fig. 10(a)] because the retrieval cost that is comparable to that of the nested index, larger than the page size. Therefore, when the record occupies several pages, the increased retrieval cost outweighs the reduced update cost, especially when the percentage of retrieval is high (80 percent or more). Further, when the operations are mainly update (70 percent or more), the multiindex performs better for low values of $k(2)$ and $k(3)$ [compare Fig. 10(c), (d)]. When the values of $k(2)$ and $k(3)$ start increasing, the decrease in the update cost is not enough to offset the increased retrieval cost. When $k(2)$ and $k(3)$ are equal to 1, the overall cost of the mulitiindex is not much higher than the cost of the other indexes.

A simple rule for determining which index to use in the case of path length 3 when the overall cost must be minimized is the following:

1) choose the multiindex if

$(k(2) = 1 \text{ and } k(3) = 1)$

or (the operations are mainly

updates and $k(2) * k(3) \leq 10$);

otherwise

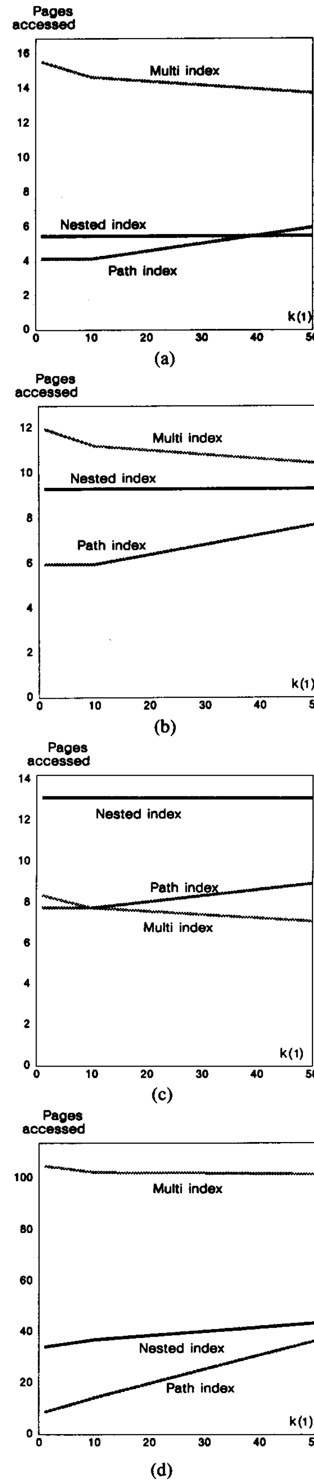


Fig. 10. (a) Overall cost $k(2) = 10$, $k(3) = 1$, mix = (80%R, 10%U, 10%D/I), number of classes = 3. (b) Overall cost $k(2) = 10$, $k(3) = 1$, mix = (50%R, 25%U, 25%D/I), number of classes = 3. (c) Overall cost $k(2) = 10$, $k(3) = 1$, mix = (20%R, 40%U, 40%D/I), number of classes = 3. (d) Overall cost $k(2) = 50$, $k(3) = 10$, mix = (20%R, 40%U, 40%D/I), number of classes = 3.

2) choose the nested index

if there are reverse references from class C(2) to class C(1), and the operations are mainly retrieval and $k(1) * k(2) * k(3) > 100$;

otherwise

3) choose the path index.

This rule is explained in a way similar to the case of path length 2. When the degree of reference sharing of class C(2) and C(3) is 1, the overall cost of the multiindex is not much higher than the other two indexes. However, as the product of $k(2) * k(3)$ goes above 10, the increase in the query cost is not offset by the lower update cost. In the other cases, the nested index is chosen if reverse references are supported, the operations are mainly retrieval, and $k(1) * k(2) * k(3) > 100$. In fact, in this case the nested index may outperform the path index, since the query cost of the path index increases faster than the nested index when $k(1) * k(2) * k(3) > 100$. This increase in the query cost for the path index has a major impact on the overall cost, if most of the operations are retrieval. If, however, this product is less than or equal to 100, or the percentage of retrieval is less than 80 percent, the path index is chosen. If it is not possible to determine the degree of reference sharing and the frequency of retrieval and update operations, a path index represents a good choice in most cases.

We now discuss briefly how our results can be generalized to the case of path length greater than 3. The nested index has a high update cost mostly due to the backward traversal. This cost increases as the degree of reference sharing increases. Therefore, this index cannot be used unless either there are few updates or the degree of reference sharing is very low (equal to 1). The path index does not require backward traversal when executing updates, and therefore has a lower overall cost in most cases. However, the leaf-record size tends to become very large when the degree of reference sharing increases, thereby increasing the query cost.

In general, the best solution for a path of length greater than 3 is to split the path into several subpaths of lengths 1, 2 or 3 and to allocate on each subpath either a nested index or a path index. This approach requires fewer index lookups than with the multiindex. Further, an update on a class requires accesses to only objects of the classes belonging to the subpath of the updated class; that is, it limits any forward and backward traversals to the subpath.

Reference [5] provides an algorithm that determines the most efficient configuration of indexes on the basis of the access patterns and logical data characteristics. In particular, the algorithm determines whether the path must be split, how many subpaths the path should be split into, and the type of index to use for each subpath. As an example, let us consider the path P = C(1).A(1).A(2).A(3).A(4).A(5).

This path may be split into the subpaths

P(1) = C(1).A(1)

P(2) = C(2).A(2).A(3)

P(3) = C(4).A(4).A(5).

A possible configuration consists of a conventional index on $P(1)$, a path index on $P(2)$, and a nested index on $P(3)$.

IX. Concluding Remarks

A nested object is an object which recursively consists of other objects. Nested objects arise naturally in object-oriented and nested relational databases as a consequence of allowing the value of an attribute of an object to be an object or a set of objects. A nested object in an object-oriented and nested relational database is an instance of a class or a relation which is the root of a nested hierarchy of classes or relations. In normalized relational database systems, a secondary index is a useful data structure for expediting the evaluation of queries involving predicates on the indexed attribute. In object-oriented and nested relational databases, the nested definition of a class or a relation requires the notion of secondary indexing to be generalized to predicates on the nested attributes of the target class or relation of a query. A nested attribute is an indirect attribute of a class or a relation that is the root of a nested hierarchy of classes or relations.

In this paper, we introduced three index organizations for use in evaluating queries on nested objects in object-oriented and nested relational databases. For each of the index organizations, we developed a detailed cost model for computing the storage cost, retrieval cost, and update cost. Using these models, we computed and compared the costs of the three indexes under a range of values for key parameters. The most important parameter is the degree of reference sharing among objects for the various classes in the path. As we have shown in the paper, this parameter influences both the retrieval and update costs.

Our general conclusion about retrieval performance is that the nested index has the best retrieval performance, followed by the path index, and then the multiindex. The nested and path index have a constant cost if the product of degrees of reference sharing among the classes does not exceed a certain threshold value. However, the cost of the multiindex is independent of the index-record size and increases linearly with the product of degrees of reference sharing.

Our general conclusion about update performance is that the multiindex is the best. For path length 2, the nested index has a slightly lower cost than the path index. For path length 3, the performance depends on how updates are distributed on the classes on the path. If updates are primarily on the first and second classes, the nested index has a slightly lower cost than the path index. If, however, updates are largely on the last class on the path, the path index costs significantly less than the nested index.

TABLE II

N(1)	D(1) = N(2)	D(2)	k(1)	k(2)	k(1,2)
200,000	200,000	200,000	1	1	1
200,000	40,000	40,000	5	1	5
200,000	20,000	20,000	10	1	10
200,000	4,000	4,000	50	1	50
200,000	2,000	2,000	100	1	100
200,000	200,000	40,000	1	5	5
200,000	40,000	8,000	5	5	25
200,000	20,000	4,000	10	5	50
200,000	4,000	800	50	5	250
200,000	2,000	400	100	5	500
200,000	200,000	20,000	1	10	10
200,000	40,000	4,000	5	10	50
200,000	20,000	2,000	10	10	100
200,000	4,000	400	50	10	500
200,000	2,000	200	100	10	1000
200,000	200,000	4,000	1	50	50
200,000	40,000	800	5	50	250
200,000	20,000	400	10	50	500
200,000	4,000	80	50	50	2500
200,000	2,000	40	100	50	5000

TABLE III

N(1)	D(1)=N(2)	D(2)=N(3)	D(3)	k(1)	k(2)	k(3)	k(1,3)
2,000,000	2,000,000	2,000,000	2,000,000	1	1	1	1
2,000,000	200,000	200,000	200,000	10	1	1	10
2,000,000	40,000	40,000	40,000	50	1	1	50
2,000,000	2,000,000	200,000	200,000	1	10	1	10
2,000,000	200,000	20,000	20,000	10	10	1	100
2,000,000	40,000	4,000	4,000	50	10	1	500
2,000,000	2,000,000	40,000	40,000	1	50	1	50
2,000,000	200,000	4,000	4,000	10	50	1	500
2,000,000	40,000	800	800	50	50	1	2,500
2,000,000	2,000,000	2,000,000	200,000	1	1	10	10
2,000,000	200,000	200,000	20,000	10	1	10	100
2,000,000	40,000	40,000	4,000	50	1	10	500
2,000,000	2,000,000	200,000	20,000	1	10	10	100
2,000,000	200,000	20,000	2,000	10	10	10	1,000
2,000,000	40,000	4,000	400	50	10	10	5,000
2,000,000	2,000,000	40,000	4,000	1	50	10	500
2,000,000	200,000	4,000	400	10	50	10	5,000
2,000,000	40,000	800	80	50	50	10	25,000

We also evaluated the combined retrieval and update cost for various mixes of retrieval, update, delete, and insert operations. Based on these evaluations, we derived some simple rules for the cases of path length 2 and 3; these rules allow us to determine the optimal index organization for most of the cases. If statistics about frequencies of operations and degrees of reference sharing are not available, the general rule is that for path length 2 the nested index should be used, provided that reverse references exist, while for path length 3 the path index should be used. For path lengths greater than 3, the best solution is to split the path into several subpaths of lengths 1, 2, or 3 and to allocate on each subpath either a nested index or a path index. Reference [5] provides an algorithm for defining the optimal configuration of indexes for paths of length greater than 3.

Throughout this paper, we assumed a query with only one predicate on a nested attribute. In general, however, a query can have a boolean combination of predicates. We offer as a topic of future research a detailed quantitative analysis of query processing strategies based on the indexing techniques presented in this paper for queries containing several predicates on both nested and simple attributes.

Appendix

Tables II and III are a list of the values for some of the parameters used in our experiments. Table II shows the values of parameters $N(1)$, $N(2)$, $k(1)$, $k(2)$, and $k(1, 2)$ used in the index evaluations in Sections V–VIII for a path of length 2.

Table III shows the values of the parameters $N(1)$, $N(2)$, $N(3)$, $k(1)$, $k(2)$, $k(3)$, and $k(1, 3)$ used in the index evaluations in Section VIII for a path of length 3.

References

[1] S. Abiteboul and N. Bidoit, "Non first normal form relations to represent hierarchically organized data," in *Proc. ACM SIGACT-SIGMOND Symp. Principles of Database Syst.*, 1984, pp. 191-200.

[2] T. Andrews and C. Harris, "Combining language and database advances in an object-oriented development environment," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst. Languages, Appl.*, Orlando, FL, Oct. 1987, pp. 430-440.

[3] J. Banerjee, H. T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim, "Data model issues for object-oriented applications," *ACM Trans. Office Inform. Syst.*, vol. 5, pp. 3-26, Jan. 1987.

[4] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173-189, 1972.

[5] E. Bertino, "On index configurations in object-oriented databases," document in preparation, 1989.

[6] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surveys*, vol. 11, pp. 121-137, June 1979.

[7] P. Dadam *et al.*, "A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Washington, DC, May 1986, pp. 356-366.

[8] D. Deshpande and D. Van Gucht, "An implementation for nested relational databases," in *Proc. Int. Conf. Very Large Data Bases*, Los Angeles, CA, Aug. 1988, pp. 76-87.

[9] D. H. Fishman *et al.*, "IRIS: An object-oriented database management system," *ACM Trans. Office Inform. Syst.*, vol. 5, pp. 48-69, Jan. 1987.

[10] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corp., Jan. 1981.

[11] IEEE Computer Society, *Database Engineering* (Special Issue on Non-First Normal Form Relational Databases), Z. M. Ozsoyoglu, Ed., Sept. 1988.

[12] S. Khoshafian and G. Copeland, "Object identity," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages Appl.*, Portland, OR, Oct. 1986.

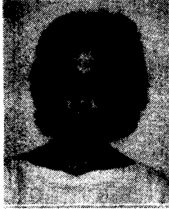
[13] W. Kim, "A foundation for object-oriented databases," MCC Tech. Rep., ACA-ST-248-88, Aug. 1988.

[14] W. Kim, K. C. Kim, and A. Dale, "Indexing techniques for object-oriented databases," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[15] D. Maier and J. Stein, "Indexing in an object-oriented DBMS," in *Proc. Int. Workshop Object-Oriented Database Syst.*, Asilomar, CA, Sept. 23-26, 1986, pp. 171-182.

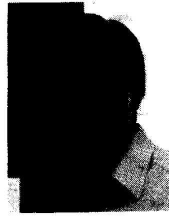
[16] D. Maier *et al.*, "Development of an object-oriented DBMS," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst. Languages, Appl.*, Portland, OR, Oct. 1986, pp. 472-482.

[17] A. Makinouchi, "A consideration of normal form of not-necessarily normalized relations in the relational data model," in *Proc. Int. Conf. Very Large Data Bases*, 1977, pp. 447-453.
[18] M. Schkolnick and P. Tiberio, "Estimating the cost of updates in a relational database," *ACM Trans. Database Syst.*, vol. 10, pp. 163-179, June 1985.
[19] S. B. Yao, "Approximating block accesses in database organizations," *ACM Commun.*, vol. 20, pp. 260-261, Apr. 1977.



Elisa Bertino (M'83) received the doctorate degree in computer science from the University of Pisa, Pisa, Italy, in 1980 with full marks and honors.

Since 1980 she has been a Researcher at the Institute for Information Processing (IEI) of the Italian National Research Council, Pisa. From July 1982 to July 1983, and during the summer of 1984, she was a Visiting Researcher at the IBM Research Laboratory, San Jose, CA, where she worked on the R* project. Since July 1988 she has been a Visiting Researcher at the Microelectronics and Computer Technology Corporation (MCC), Austin, TX, where she is currently working on several issues concerning object-oriented database management systems.



Won Kim received the B.S. and M.S. degrees in physics from the Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign. His Ph.D. dissertation was on query processing in relational database systems.

He joined the Microelectronics and Computer Technology Corporation (MCC), the research consortium of U.S. computer companies, in its early days (1984) and is currently Director of the Object-Oriented and Distributed Systems Laboratory in the Advanced Computing Technology Program of MCC, Austin, TX. He is also a Principal Scientist at MCC. He is the chief architect of the ORION object-oriented database system, intended for applications in the artificial intelligence, computer-aided design, and office systems domains. Prior to joining MCC, he was a Research Staff member for four years at IBM Almaden Research Center, where he participated in a number of database systems research projects, including distributed systems, highly available systems, and engineering design database systems.

Dr. Kim is currently an Associate Editor of the *ACM Transactions on Database Systems*, and for six years has been the Chief Editor of *Data Engineering*, the quarterly bulletin of the IEEE Technical Committee on Data Engineering. He has organized international conferences on Databases for Parallel and Distributed Systems, Deductive and Object-Oriented Databases, and Databases for Advanced Applications.