Combining Indexing Technique with Path Dictionary for Nested Object Queries

Wang-Chien Lee
Dept. of Computer & Information Science
The Ohio State University
Columbus, Ohio 43210-1277, USA

wlee@cis.ohio-state.edu, FAX: 614-292-2911

Dik Lun Lee*
Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Hong Kong

dlee@cs.ust.hk, FAX: 852-358-1477

Abstract

A path dictionary encodes the connections among objects in the aggregation hierarchy. It has been shown to be an efficient mechanism for supporting nested object queries [6]. In this paper, we present a new method, called the path dictionary index method, which combines indexing techniques with the path dictionary method. We describe the operations of the new mechanism and develop cost models for its storage overhead and query and update costs. Finally, we compare the new mechanism to the path index method [3]. The result shows that the path dictionary index method is significantly better than the path index method over a wide range of parameters.

1 Introduction

The concept of *class* allows object-oriented database systems (OODBSs) to model complex data more precisely and conveniently than the relational data model. A class may consist of *simple attributes* (e.g., of domain `integer` or `string`) and *complex attributes* with user-defined classes as their domains. Since a class C may have a complex attribute with domain C', an *aggregation relationship* can be established between C and C'. Using arrows connecting classes to represent aggregation relationship, a directed graph, called the *aggregation hierarchy*, may be built to show the nested structure of the classes. Figure 1 is an example of an aggregation hierarchy, which consists of four classes, `Person`, `Vehicle`, `Person_Name`, and `Company`. The class `Person` has three *simple attributes*, `SSN`, `Residence` and `Age`, and two *complex attributes*, `Owns` and `Name`. The domain classes of the attributes `Owns` and `Name` are `Vehicle` and `Person_Name`, respectively. The class `Vehicle` is defined by three simple attributes, `Id`, `Color`, and `Model`, and a complex attribute `Manufacturer`, which has `Company` as its domain. `Company` and `Person_Name` each consists of two simple attributes.

Every object in an OODBS is identified by an *object identifier* (OID). The OID of an object may be stored as attribute values of other objects. If an object O is referenced as an attribute of object O', O is said to be *nested* in O' and O' is referred to as the *parent* object of O. Objects are nested according to the aggregation hierarchy.

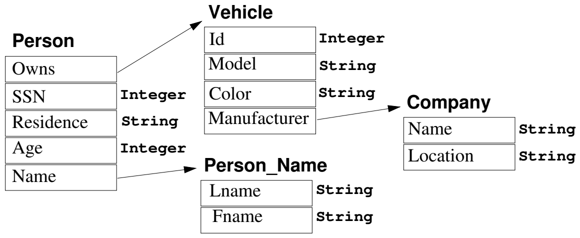

Figure 1: Aggregation hierarchy.

OODBSs support queries involving nested objects. These queries are called *nested queries*. There are two basic approaches to evaluating a nested query: *top-down* and *bottom-up* evaluations. The top-down approach traverses the objects starting from an ancestor class to a nested class. Since the OID in a parent object leads directly to a child object, this approach is also called a *forward traversal* approach. On the other hand, the bottom-up method, also known as *backward traversal*, traverses up the aggregation hierarchy. A child object, in general, does not carry the OID of (or an inverse reference to) its parent object. Therefore, in order to identify the parent object(s) of an object, we have to compare the child object's OID against the corresponding complex attribute in the parent class. This is similar to a relational join when we have more than one child object to start with. Mixed evaluation is a combination of the top-down and bottom-up approaches, which is often used for complex queries. Note that when every reference from an object O to another object O' (e.g., `Owns`) is accompanied with an inverse reference from O' to O (e.g., `Owned_by`), the aggregation hierarchy becomes bidirectional, resulting in no difference between the top-down and the bottom-up approaches. In this paper, however, we assume there is no inverse references.

Many access methods have been proposed to support the complex queries in OODBSs. In particular, three techniques, namely, indexing [1,2,3], signature file [4,7] and path dictionary [6], have been proposed recently. (A similar idea to path dictionary has also been used in [5]). In this paper, we further develop our previous implementation of the path dictionary approach [6] and combine it with index-

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

*The author is on leave from the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210, USA.

ing, resulting in a new object access mechanism, the path dictionary index (PDI). We find that the storage overhead and the retrieval and update costs of PDI is better than that of the path index [3].

There are many kinds of nested queries. However, an access method doesn't necessarily support all of them. Even with the same access method, different kinds of queries may be evaluated differently. To facilitate our discussion, we define *target classes* as the classes from which objects are retrieved and *predicate classes* as the classes involved in the predicates of the query. We classify nested queries by the following factors:

1. Relative positions of the target and predicate classes on the aggregation hierarchy:
   - TP: The target class is an ancestor class of the predicate classes.
   - PT: The target class is a nested class of the predicate classes.
   - MX: The target class is an ancestor class of some predicate class and a nested class of some predicate class.
2. The complexity of the predicates:
   - Simple predicates: The predicate is specified on a simple attribute. Based on the operators used in the predicates, this class of nested queries is further divided as follows.
     - Equality: $=$.
     - Range: $>, \geq, <$ and $\leq$.
     - Inequality: $\neq$.
   - Complex predicates: The predicate is specified on a complex attribute. Depending on whether or not an OID is specified in the predicate, this class can be further divided into:
     - Exist: An OID is specified in the predicate.
     - Nonexist: No OID is specified in the predicate.

The remainder of the paper is organized as follows. Section 2 introduces *s*-expressions, an implementation technique for the path dictionary, and its integration with indexes. Section 3 discusses the retrieval and update operations with path dictionary index. Section 4 presents the performance analysis and comparison of path index and path dictionary index. Finally, we conclude the paper in Section 5.

2 Path Dictionary Index

A *path dictionary index* (PDI) consists of a path dictionary, an identity index, and a number of attribute indexes. In a previous paper [6], we proposed the concept of path dictionary, presented the *s*-expression scheme for its implementation and evaluated the storage overhead, and query and update costs. Compared to the path index [3], the path dictionary has better overall query performance and lower storage overhead, but the update performance is inferior. In this paper, we revise the path dictionary and augment it with an *identity index* and *attribute indexes* to speed up its performance. Instead of sequentially scanning the path dictionary, the indexes support rapid access to *s*-expressions in the path dictionary, thus dramatically improving update as well as retrieval performance at the cost of only a small increase in storage overhead.

Path = Person.Vehicle.Company

Company[1](Vehicle[5](Person[3], Person[7]), Vehicle[12](Person[4]))
Company[2](Vehicle[6](), Vehicle[9](), Vehicle[11]())
Company[3](Vehicle[3]())
Company[4](Vehicle[4](), Vehicle[7](Person[1], Person[6]))
Company[5](Vehicle[1](Person[2]), Vehicle[2](Person[8], Person[12]),
Vehicle[8](Person[5]), Vehicle[10]())
((Person[9]))

Figure 2: Examples of *S*-expression.

2.1 S-expression Scheme

A *path dictionary* is created for a path, $C_1C_2...C_n$, in the aggregation hierarchy. It is a secondary file containing nesting information about the objects in the classes along the path.

The *s*-expression scheme encodes into a recursive expression all paths terminating at the same object in a leaf class. The *s*-expression for the path $C_1C_2...C_n$ is defined as follows:

$S_1 = \theta_1$, where θ_1 is the OID of an object in class C_1 or null.

$S_i = \theta_i(S_{i-1}[, S_{i-1}])\ \ 1 < i \leq n$, where θ_i is the OID of an object in class C_i or null, and S_{i-1} is an *s*-expression for the path $C_1C_2...C_{i-1}$.

S_i is an *s*-expression of i levels, in which the list associated with θ_i contains recursively the OIDs of all ancestor objects of θ_i.[1] We call it the *ancestor list* of θ_i. Except for the objects in C_1, every object on the path has an ancestor list, which may be empty.

The path dictionary for $C_1C_2...C_n$ consists of a sequence of n-level *s*-expressions. The leading object in an *s*-expression, which does not necessarily belong to C_n, is the terminal object of the paths denoted by the *s*-expression. Several *s*-expressions are shown in Figure 2. They represent the linkage information for the objects on the path `Person.Vehicle.Company`. In the examples, we use `Person`[i], `Vehicle`[i], and `Company`[i] to refer to the OIDs of the ith objects in `Person`, `Vehicle` and `Company`, respectively. The first *s*-expression in the figure indicates that there are three paths, all terminating at `Company`[1], and that `Person`[3] and `Person`[7] connect to `Company`[1] through the common node `Vehicle`[5]. It is possible that the first i levels of an *s*-expression are all null, which means the object on level $i+1$ is the terminal object for the subtree represented by the *s*-expression. For instance, the last *s*-expression in the figure, ((`Person`[9])), indicates that `Person`[9] has no car and therefore no manufacturer for the car. On the other hand, an *s*-expression may contain null ancestor lists, which indicates that the object is not referenced by any other object. For instance, in the third *s*-expression in Fig. 2, the ancestor list for `Vehicle`[3] is empty, meaning that the vehicle doesn't have an owner. An advantage of the *s*-expression scheme is that every object on the path appears only once in the path dictionary, which avoids redundant partial path information introduced in other schemes [8].

2.2 Implementation of the *s*-expression Scheme

Figure 3(a) illustrates the data structure of an *s*-expression for $C_1C_2...C_n$. SP_i in the header points to the first occur-

[1] Although θ_i denotes the OID of an object, we use it to refer to the object itself, as in this case, when no confusion arises.

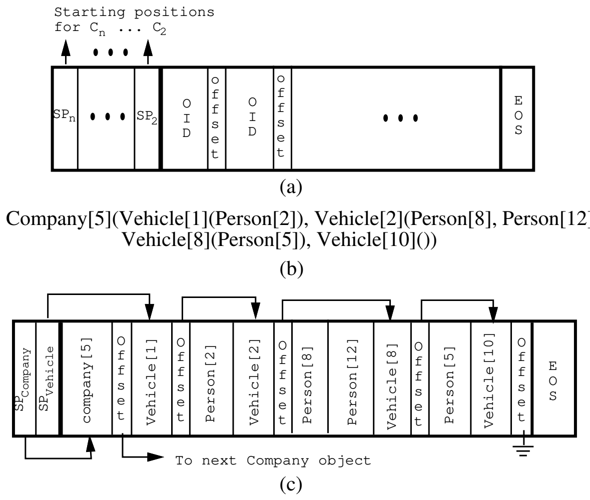

Figure 3: Data structure of an S-expression.

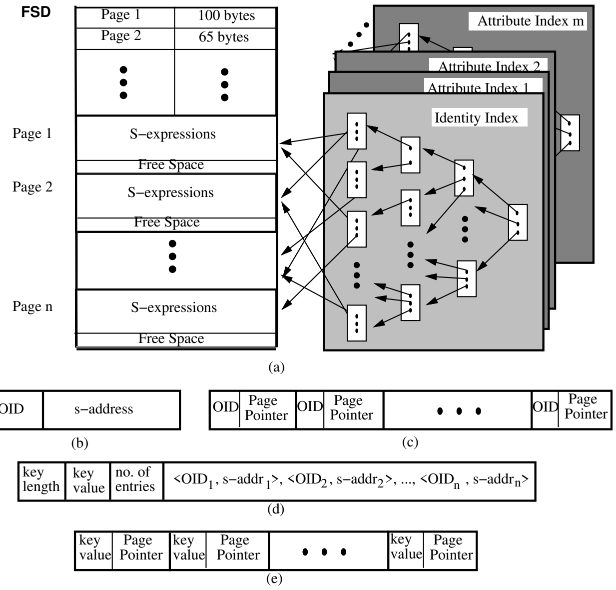

Figure 4: Path Dictionary Index. (a) Structure of the path dictionary index; (b) leaf node record of the identity index; (c) nonleaf node of the identity index; (d) leaf node record of an attribute index; (e) nonleaf node of an attribute index.

rence of θ_i in the s-expression. Following the SP_i fields is a series of $\langle$*OID*, *Offset*$\rangle$ pairs. At the end of the s-expression is a special end-of-s-expression (EOS) symbol. The data structure mimics the nesting structure in the s-expression. The OIDs in the data structure are in the same order as the OIDs in the unwrapped s-expression corresponding to the data structure. The offset associated with θ_i, $2 \le i \le n$, points to the next occurrence of θ_i in the s-expression. The OIDs for C_1 don't have offset fields, since all θ_1's referencing the same θ_2 are stored consecutively right after θ_2; for the same reason, SP_1 is not needed either, since θ_1's can be located by tracing θ_2's. Using the SP_i and offset values, we can easily trace the nested relationship among objects in an s-expression. For example, to obtain the ancestor list associated with θ_i, we simply collect the OIDs stored after θ_i until we reach the OID pointed to by θ_i's offset. An s-expression and its representation are shown in Fig. 3(b) and (c), respectively.

An advantage of this representation is that it allows fast retrieval of OIDs in the same class. To retrieve all OIDs for class C_i, we start with SP_i, which will lead us to the first θ_i in the s-expression, and following the associated offset value we can reach the next θ_i, and so on. Thus, we can quickly scan through all OIDs in a class, skipping the OIDs of irrelevant classes. Notice that the offset associated with θ_n is pointing to θ_n in the *next* s-expression, because there is at most one OID of class C_n in an s-expression.

S-expressions are stored sequentially on disk pages. In order to reduce the number of page accesses, an s-expression is not allowed to cross page boundaries unless the size of the s-expression is greater than the page size. If an s-expression is too long to fit into the space left in a page, a new page is allocated. Consequently, free space may be left in a page. Updates and insertions may cause a page to overflow, which requires a new page to be allocated and some of the s-expressions in the overflown page to be moved to the new page. In order to effectively keep track of the free space available in the pages, a free space directory (FSD), which records the pages with free space above a certain threshold, is maintained at the beginning of the path dictionary.

The identity index allows us to quickly locate all s-expressions containing a given OID. Since search on the s-expressions is important for retrieval and update operations, the identity index significantly reduce the cost for retrieval and update operations. Figure 4(a) illustrates the physical structure of the path dictionary index, which consists of the free space directory, the s-expression pages, and augmented indexes. Fig. 4(b) and (c) show the structures of a leaf node record and a non-leaf node for B$^+$-tree implementation of the identity index, respectively.

An attribute index is built on a simple attribute to map attribute values into s-expressions in the path dictionary. It is the same as a regular index (e.g., a B$^+$-tree). For instance, given an attribute index on `Age` of `Person`, we can quickly identify all s-expressions referring to persons of a certain age. Figure 4(d) and (e) shows the structures of an attribute index's leaf node record and nonleaf node page. The OIDs and s-expression addresses (denoted as s-addr) are used to access the s-expressions of the corresponding OIDs. Attribute indexes improve the path dictionary's performance in predicate evaluation and range query processing, because single-value predicates and range predicates can be performed by efficient index scanning rather than accessing to all of the objects in the predicate classes.

3 Retrieval and Update with Path Dictionary Index

In this section, we specify a nested query Q as having C_t as the target class and C_p as the predicate class, where $1 \le t, p \le n$. We use θ_t and θ_p to denote OIDs of objects in class C_t and C_p.

We assume that a path dictionary index for the path $C_1 C_2 ... C_n$ has been created. $Index_p$ is the attribute index based on an attribute of C_p. The path dictionary supports all classes of the nested queries. The following strategies are applicable to both TP and PT queries.

Simple predicates: We use the attribute value specified in the predicate to search attribute index $Index_p$ for the corresponding addresses of the s-expressions. Through the addresses, we can obtain the s-expressions and derive from the s-expressions the OIDs for C_t. PDI allows us to avoid accessing any objects from the database. Assuming that the attribute `Name` of `Company` is indexed by $Index_{name}$, we can answer a query "retrieve persons who owns cars made by GM" by first searching $Index_{name}$ using "GM" as the search key to obtain the addresses of the s-expressions corresponding to "GM". After the s-expressions are accessed through the addresses, the OIDs of the `Person` objects in the s-expressions are returned. Owing to space limitations, we only present the strategy for queries with equality operator; the other classes of queries can be derived in a similar way [8].

Complex predicates: Attribute indexes have great advantages on predicate evaluation. Unfortunately, they don't benefit queries with complex predicates, which require scanning the identity index or sequentially searching the path dictionary.

The strategies for answering this class of nested queries are different depending on the existence of θ_p in the predicate.

Exist: If θ_p is specified in the predicate, we can use the identity index to locate the s-expressions containing θ_p from the path dictionary and derive θ_t from the s-expression for predicate evaluation. If the relationship between θ_p and θ_t satisfies the predicate, θ_t is returned. For example, to answer a query "retrieve persons who have a car made by `Company`[1]", we search the identity index using `Company`[1] as the key, obtain from the path dictionary the s-expressions corresponding to `Company`[1], and derive from the s-expressions the OIDs of `Person` objects.

Nonexist: If no θ_p is specified in the predicate, we will scan the path dictionary for the s-expressions in which the predicate on C_t and C_p is satisfied, and return θ_t. Take "retrieve persons who don't have a car" as an example. we sequentially search the s-expressions in the path dictionary for the pattern "((`Person?`))" and return the matching `Person` objects. Without the path dictionary, we will have to examine every `Person` object in the database and check if the `Owns` attribute is null or not. On the other hand, to evaluate "retrieve vehicles which are not owned by any person" (the PT case), we sequentially scan the path dictionary and simply return all the vehicle objects with an empty ancestor list (i.e., vehicle objects matching the pattern "`Vehicle?()`"). Without the path dictionary, the query would be very expensive since it requires a scan through the `Vehicle` class to collect all OIDs in it, another scan through the `Person` class to collect all OIDs under the `Owns` attribute (i.e., all vehicles with owners), and a set difference between the two result sets.

When changes are made to the database, the path information in the dictionary must be updated. Operations such as update, insertion, deletion, creation, destruction and destroy will require updates to the path dictionary. Due to space constraints, we only describe the update operation. The PDI has to be updated in the following situations:

1. When an indexed simple attribute is modified: the corresponding attribute index has to be updated, while the path dictionary and the identity index need not be changed. Suppose one of the indexed attributes of an object, identified by θ, is modified. Let A_θ be the address of the s-expression containing θ. The update of the attribute indexes is accomplished by two index scans: one to delete A_θ from the leaf node corresponding to the old attribute value, and the other to insert A_θ to the leaf node corresponding to the new attribute value.
2. When one of the complex attributes connecting the path is modified: Suppose object O_i changes its complex attribute from O_{i+1} to O'_{i+1} (O_i, O_{i+1} and O'_{i+1} are identified by θ_i, θ_{i+1} and θ'_{i+1}.) If none of the direct attributes of class C_i and none of the direct attributes of C_i's ancestor classes are indexed, we have to search the path dictionary through the identity index to find the s-expressions containing θ_i and θ'_{i+1}. Then, θ_i and its ancestor list are moved from the ancestor list of θ_{i+1} to the ancestor list of θ'_{i+1}. Meanwhile, the identity index has to be updated by changing the old s-expression address in θ_i's leaf node to the new address. However, if some direct attributes of class C_i or C_i's ancestor classes are indexed, we also have to update those attribute indexes, which is the same as described above.

Assume that the attribute `Age` of class `Person` is indexed by $Index_{age}$. Let's consider the following update examples.

The update "change `Person`[1]'s age from 50 to 51" will not change the path dictionary and the identity index, but $Index_{age}$ has to be searched twice to move `Person`[1]'s s-expression address from the leaf node corresponding to 50 to the leaf node corresponding to 51. Next, to "change `Person`[1]'s car from `Vehicle`[7] to `Vehicle`[10]", we first use the identity index to locate the s-expressions corresponding to `Vehicle`[7] and `Vehicle`[10]. The path dictionary is updated by moving `person`[1] from the s-expression corresponding to `Vehicle`[7] to the s-expression corresponding to `Vehicle`[10]. The identity index is then updated by changing the leaf node of `Person`[1] from pointing to the s-expression corresponding to `Vehicle`[7] to that corresponding to `Vehicle`[10]. Finally, the attribute index $Index_{age}$ has to be updated by removing the s-expression address corresponding to `Vehicle`[7] and inserting the s-expression address corresponding to `Vehicle`[10] into the leaf node of $Index_{age}$ corresponding to the age of `Person`[1].

4 Storage Cost and Performance Evaluation

In this section, we formulate the cost models for the path index and path dictionary index methods to analyze and compare their storage overhead and query processing performance. We select the path index as a reference point in the comparison, because it can be generally applied to queries with different target classes as long as the classes are on the indexed path (i.e., TP queries). However, the path index can't be used for PT queries, because its structure implies a bottom-up evaluation. The path dictionary and path dictionary index are general enough to provide significant support for both TP and PT queries.

In order to facilitate our comparison, we adopt some

Table 1: Parameters of the cost models.

P	=	4096	FSL	=	2
$UIDL$	=	8	EL	=	4
pp	=	4	kl	=	8
$OFFL$	=	2	ol	=	6
SL	=	2	f	=	218

common parameters from [3]. We use the following parameters to describe the characteristics of the classes and their attributes on the path, $C_1C_2...C_n$, and the structures of these two organizations.

- N_i: the number of objects in class $C_i, 1 \leq i \leq n$.
- S_i: the average size of an object in class C_i.
- A_i: the complex attribute of C_i used on the path, $1 \leq i \leq n$.
- D_i: the number of distinct values for complex attribute A_i.
- k_i: the ratio of shared reference between objects in class C_i and values for A_i. ($k_i = N_i/D_i$.)
- $A_{i,j}$: the jth simple attribute of C_i, $1 \leq i \leq n$.
- $U_{i,j}$: the number of distinct values for simple attribute $A_{i,j}$ of class C_i.
- $q_{i,j}$: the ratio of shared attribute value between objects in class C_i and values for attribute $A_{i,j}$. ($q_{i,j} = N_i/U_{i,j}$.)
- K: the average ratio of shared references, i.e., k's, and shared attribute values, i.e., q's.
- $UIDL$: the length of an object identifier.
- P: the page size.
- pp: the length of a page pointer.
- f: average fanout from a nonleaf node in the path index, identity index, and attribute indexes.
- kl: average length of a key value in path index and attribute indexes.
- ol: the sum of the key length, record length, and number of path fields in the path index.
- $OFFL$: the length of an offset field in the path dictionary.
- SL: the length of the start field in the path dictionary.
- FSL: the length of the free space field in the free space directory.
- EL: the length of EOS.

Performance is measured by the number of I/O accesses. Since a *page* is the basic unit for data transfer between the main memory and the external storage, we use it to estimate the storage overhead and the performance cost. All lengths and sizes used above are in *bytes*.

To directly adopt the formulae developed in [3], we follow their assumptions:

1. There are no partial instantiation, which implies that $D_i = N_{i+1}$.
2. All key values have the same length.
3. Attribute values are uniformly distributed among the objects of the class defining the attribute.
4. All attributes are single-valued.

Further, we adopt the parameter values in [3]. Table 1 lists the values chosen for the path dictionary.

4.1 Storage Overhead

Path Index

To create a path index for a primitive attribute, $A_{n,j}$, of the class C_n, which maps the key values of $A_{n,j}$ to every classes on the path $C_1C_2...C_n$, the number of pages needed for leaf nodes is [3]:

If $XP \leq P$:

$$LP = \lceil U_{n,j}/\lfloor P/XP \rfloor \rceil,$$

where $XP = k_1k_2...k_{n-1} \cdot q_{n,j} \cdot UIDL \cdot n + kl + ol$.

If $XP > P$:

$$LP = U_{n,j} \lceil XP/P \rceil,$$

where $XP = k_1k_2...k_{n-1} \cdot q_{n,j} \cdot UIDL \cdot n + kl + ol + DS$, $DS = \lceil (k_1k_2...k_{n-1} \cdot q_{n,j} \cdot UIDL \cdot n + kl + ol)/P \rceil (UIDL \cdot n + pp)$.

The number of nonleaf pages is:

$$NLP = \lceil LO/f \rceil + \lceil \lceil LO/f \rceil /f \rceil + ... + 1,$$

where $LO = min(U_{n,j}, LP)$.

The total number of pages needed for the path index is:

$$PIS = LP + NLP.$$

Path Dictionary Index

Each object in the path dictionary, except for those in the root class of the path, is associated with an offset. Therefore, an object will take at most $(UIDL + OFFL)$ bytes in an s-expression. The average number of objects in an s-expression is:

$$NOBJ = 1 + K_{n-1} + K_{n-1}K_{n-2} + \cdots + K_{n-1}K_{n-2}...K_1.$$

Thus, the average size of an s-expression is:

$$SS = SL \cdot (n-1) + (UIDL + OFFL)NOBJ + EL.$$

The number of pages needed for all of the s-expressions on the path is:

$$SSP = \begin{cases} \lceil N_n/\lfloor P/SS \rfloor \rceil & \text{if} \quad SS \leq P \\ N_n \lceil SS/P \rceil & \text{if} \quad SS > P. \end{cases}$$

The number of pages needed for the free space directory is:

$$FSD = \lceil SSP(pp + FSL)/P \rceil.$$

The number of leaf pages needed for the identity index is:

$$LP_{identity} = \lceil NOBJ \cdot N_n / \lfloor P/(UIDL + pp) \rfloor \rceil.$$

The number of the nonleaf pages for identity index is:

$$NLP_{identity} = \lceil LP_{identity}/f \rceil + \lceil \lceil LP_{identity}/f \rceil /f \rceil + \cdots + 1.$$

The total number of pages needed for the identity index is:

$$IIP = LP_{identity} + NLP_{identity}.$$

For an attribute index based on the jth primitive attribute, $A_{i,j}$, of the class C_i, the average number of pages needed for a leaf node record is:

$$XP_{A_{i,j}} = kl + ol + q_{i,j}(UIDL + pp).$$

The number of leaf node pages needed is:

$$LP_{A_{i,j}} = \begin{cases} \lceil U_{i,j}/\lfloor P/XP_{A_{i,j}} \rfloor \rceil, & \text{if} \quad XP_{A_{i,j}} \leq P \\ U_{i,j} \lceil XP_{A_{i,j}}/P \rceil, & \text{if} \quad XP_{A_{i,j}} > P. \end{cases}$$

The number of nonleaf pages is:

$$NLP_{A_{i,j}} = \lceil LO_{A_{i,j}}/f \rceil + \lceil \lceil LO_{A_{i,j}}/f \rceil /f \rceil + \cdots + 1,$$

where $LO_{A_{i,j}} = min(U_{i,j}, LP_{A_{i,j}})$.

Thus, the total number of pages needed for indexing $A_{i,j}$ is:

$$AIP_{A_{i,j}} = LP_{A_{i,j}} + NLP_{A_{i,j}}.$$

As a result, the number of pages needed for the path dictionary index is:

$$PDIS = FSD + SSP + IIP + AIP_{index_1} + AIP_{index_2} + \cdots + AIP_{index_m},$$

where $index_1, index_2, ..., index_m$ are the attribute indexes created.

Comparison

Using the formulae developed above, we compare the storage overhead of the path index and path dictionary index. We choose a path of 4 classes in the comparison. A primitive attribute, $A_{4,1}$, is chosen for indexing. Also, we fix the cardinality of N_1 to 200000 and the average size of an object to 80 bytes. In the following, we use *PIS* and *PDIS* to represent the storage overhead of the path index and path dictionary index.

To observe the impact of the ratios of shared references and shared key values on the storage overhead, we vary the average ratio, K, from 1 to 25. Figure 5 shows that $PIS < PDIS$ when $K = 1$ and 2, and that $PDIS < PIS$ when $3 \leq K \leq 25$. The explanation for the case of $K = 1$ is that, when there are no shared references and key values in the database, the structure of an s-expression in the path dictionary index is similar to that of a leaf node record in the path index, except that extra storage overhead is incurred for the offset fields in the s-expressions, the identity index, and the leaf node records in the attribute indexes. However, the amount of redundant path information in the path index increases when the ratios of shared references and shared key values increase. Therefore, we can conclude that, in general, the path dictionary index has better storage overhead than the path index.

4.2 Retrieval Cost

Path Index

Since the structure of the path index implies a bottom-up evaluation, it can't be applied to *PT* queries. Therefore, the traditional forward traversal approach is used. The cost model for evaluating *TP* queries with the path index is given in [3]. The number of pages accessed for retrieval is:

$$PIR = h + \lceil XP/P \rceil.$$

where h = height of path index $- 1$, and *XP* is the size of a leaf node record in the path index.

Path Dictionary Index

The path dictionary index supports both *TP* and *PT* queries. To answer a query Q using the path dictionary index, we need to traverse a number of nonleaf nodes and one leaf node record in the attribute index, $Index_{A_{i,j}}$, and

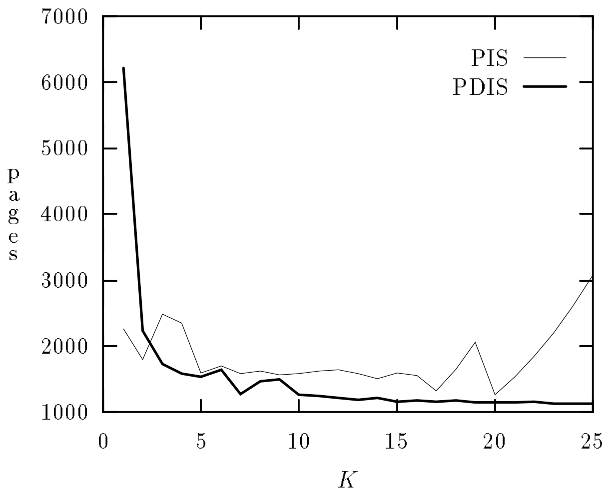

Figure 5: Storage overhead.

then access the path dictionary. Therefore, the number of pages accessed is:

$$PDIR = h_{A_{i,j}} + \lceil XP_{A_{i,j}}/P \rceil + N_{p|Q} \cdot \lceil SS/P \rceil$$

where $N_{p|Q}$ is the number of objects in class C_p satisfying the predicate in Q and $h_{A_{i,j}}$ = height of the attribute index -1.

Comparison

We use the same parameters and assumptions as we used in evaluating the storage cost. We use *PIR* and *PDIR* to represent the retrieval costs of the path index and the path dictionary index, respectively.

First, we assume that the query has C_1 as the target class, C_4 as the predicate class, and $A_{4,1}$ as the predicate attribute. We assume that all k and q values equal to an average ratio, K. As before, we increase K from 1 to 25 to observe the effect on retrieval cost.

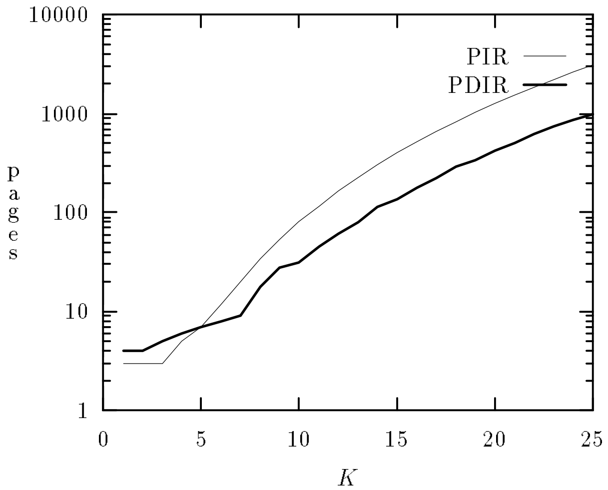

Figure 6: Retrieval cost.

Figure 6 indicates that the path index has lower retrieval cost initially. However, as K increases, the effect of redundant path information in the path index becomes dominant, costing more page accesses. After $k = 5$, the path dictionary index has a lower retrieval cost than the path index.

To compare the overall retrieval performance of the two methods, we select the following mix of queries for evaluation: (We only include the TP class of the queries in the list, because PT queries are not supported by path index.)

1. Three queries in which the indexed attribute, $A_{4,1}$, of C_4 is the only predicate attribute, and each with C_1, C_2 or C_3 as the target class.
2. Three queries in which a non-indexed attribute, $A_{4,2}$, of C_4 is the predicate attribute, and each with C_1, C_2 or C_3 as the target class.
3. Two queries in which C_1 is the target class, and each with C_2 or C_3 as the predicate class.

Since the queries in 2 and 3 are not supported by the path index, we have to use the traditional forward traversal or backward traversal approaches to evaluate these queries. When the queries are not supported by the path index, we use the cost models for retrieval without path index/path dictionary developed in [6] to compute the retrieval cost.

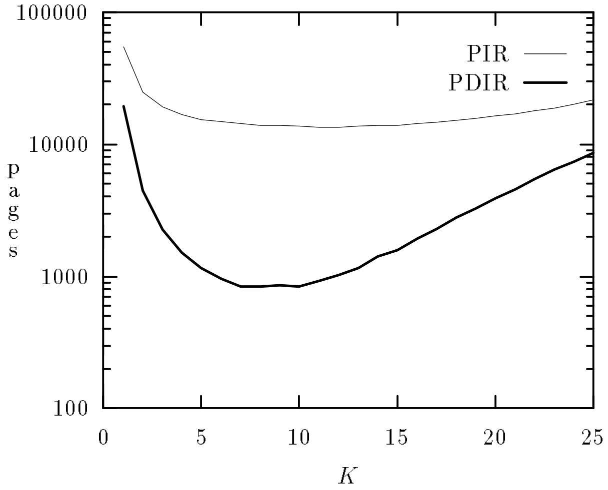

Figure 7: General retrieval cost.

As before, we vary the average ratio of shared references and shared key values, K, from 1 to 25 to observe the retrieval performance of the methods with respect to K. Fig. 7 shows that *PDIR* has a much better overall retrieval performance than *PIR*. The overall performance of *PDIR* will be better if we index more attributes on the path. Likewise, the overall retrieval performance of the database will improve if we create more path indexes on different attributes. However, some queries, such as PT queries, won't be supported at all by the path index method. Also, the cost of building more path indexes is very expensive.

4.3 Update Cost

To simplify the analysis, we do not include the costs due to page overflow caused by insertion or update operations. In the following, we assume that the complex attribute A_{i+1} of O_i is changed from θ_{i+1} to θ'_{i+1} (θ_{i+1} and θ'_{i+1} are OIDs of O_{i+1} and O'_{i+1}).

Path Index

Suppose that the path index is based on the simple attribute, $A_{n,1}$ of the class C_n. To determine the nested attribute values in $A_{n,1}$ for O_{i+1} and O'_{i+1}, we need two forward traversal to $A_{n,1}$:

$$FT = \lceil S_i/P \rceil + \lceil S_{i+1}/P \rceil + \cdots + \lceil S_n/P \rceil.$$

To simplify the cost model, we assume that O_{i+1} and O'_{i+1} have different key values and that they are in different leaf node pages of the path index. To search through the non-leaf nodes of the path index and to read and write the leaf pages for O_{i+1} and O'_{i+1}, the number of page accesses needed is:

$$CO = h + 2\lceil XP/P \rceil.$$

Therefore, the number of pages accesses for update with the path index is:

$$PIU = 2(CO + FT).$$

Path Dictionary Index

There are three different cases in which we have to update the path dictionary index:

1. An indexed simple attribute $A_{i,j}$ is modified: in this case, the update necessary for the PDI is to update the attribute index involved. Since two index scans are needed, the number of page accesses for update with PDI is:

   $$PDIU = 2(h_{A_{i,j}} + 2\lceil XP_{A_{i,j}}/P \rceil),$$

   where $h_{A_{i,j}}$ = height of the attribute index $-$ 1 and $XP_{A_{i,j}}$ is the size of a leaf node record in the attribute index.
2. The complex attribute A_{i+1} of O_i is changed from θ_{i+1} to θ'_{i+1}, and no attribute in C_i and C_i's ancestor classes are indexed. In this case, the update cost is:

   $$PDIU = 2(h_{identity} + 2 + 2\lceil SS/P \rceil),$$

3. If one of the attributes in C_i or C_i's ancestor classes are indexed, e.g., $A_{i,j}$, the attribute index has to be updated too. Therefore, the number of page accesses for update with PDI is:

   $$\begin{aligned} PDIU = \; & 2(h_{identity} + 2 + 2\lceil SS/P \rceil) \\ & + 2(h_{A_{i,j}} + 2\lceil XP_{A_{i,j}}/P \rceil), \end{aligned}$$

   where $h_{A_{i,j}}$ = height of the attribute index $-$ 1 and $XP_{A_{i,j}}$ is the size of a leaf node record in the attribute index.

Comparison

For the first two cases, both of the path index and the path dictionary index are required to update their indexes. For the third case, the path index doesn't have to be updated, because the indexed attribute in the path index must be at the leaf class of the path. However, this is why the path index doesn't support PT queries. In our comparison, we choose the formula for case 2 to compute the update cost, *PDIU*, of the path dictionary index, since it's expected to be higher than the cost for case 1.

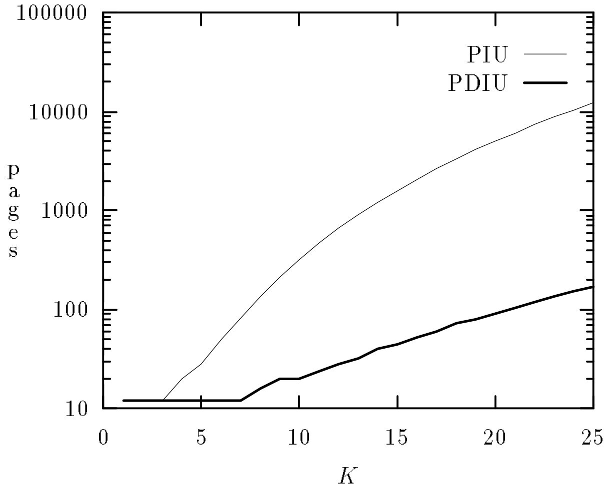

Figure 8: Update Cost.

Figure 8 depicts the update costs of the three methods. Initially, the path index has the same update cost as that of the path dictionary index. However, as K increases, the update cost of the path index dramatically increases, while the update costs for the path dictionary index remain relatively low.

5 Conclusion

In this paper, we classified nested queries in OODBS and introduced the path dictionary and path dictionary index methods for nested query evaluation. We also developed cost models for the storage overhead and retrieval and update costs, and compared the costs to that of the path index method. For most of the comparisons, we varied the average ratio of the shared references and the shared key values to observe its impact on the performance and overhead of the two mechanisms.

When only one attribute is indexed, the storage overhead for the path dictionary index is better than that of the path index, except for very small reference ratios. When more than one attribute are indexed, we expect the storage overhead for the path index method to be drastically larger than that of the path dictionary index. The storage requirement for the path dictionary itself is constant. The extra storage needed for the path dictionary index to index on new attributes is to create the attribute indexes, which is low comparing to creating new path indexes. Thus, it is affordable to create many attribute indexes on the path dictionary.

Generally speaking, the path dictionary index method has better retrieval performance. The path index method is better than the path dictionary organization only when the average ratio of shared references and key values is extremely low. Furthermore, when considering a general mix of nested queries, the performance of the path dictionary index is significantly better than that of the path index. Also, the path index can't be used to support the kind of queries in which the predicate class is located on top of the target class. For the update operation, the performance of the path dictionary index method is better than that of the path index method under all conditions.

We have shown that the path information embedded among objects can be exploited to significantly improve the performance of nest object queries in object-oriented databases. We are currently investigating a new method which combines the signature file technique with the path dictionary and developing cost models for the new organization.

References

[1] E. Bertino, "An Indexing technique for object-oriented databases," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, 1991, 160–170.

[2] E. Bertino, "Optimization of Queries using Nested Indices," *Proceedings of International Conference on Extending Database Technology*, Venice, Italy, March 1990, 44–59.

[3] E. Bertino & W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, June 1989, 196–214.

[4] Y. Ishikawa, H. Kitagawa & N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings of the 1993 SIGMOD Conference*, Washington, DC, June 1993, 247–256.

[5] A. Kemper & G. Moerkotte, "Access support in object bases," *Proceedings of the 1990 SIGMOD Conference*, Atlantic City, NJ, May 1990, 364–374.

[6] D.L. Lee & W.-C. Lee, "Using Path Information for Query Processing in Object-Oriented Database Systems," *Proceedings of Conference on Information and Knowledge Management*, Washington, DC, Nov. 1994, 64–71.

[7] W.-C. Lee & D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," *Proceedings of the 2nd International Computer Science Conference*, Hong Kong, Dec. 1992, 616–622.

[8] W.-C. Lee & D.L. Lee, "Path dictionary: A new approach to query processing in object-oriented databases," in preparation.