Using Path Information for Query Processing in Object-Oriented Database Systems

Dik L. Lee and Wang-chien Lee
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277
{dlee, wlee}@cis.ohio-state.edu

Abstract

This paper argues that most queries in object-oriented databases require traversing from one object to another in the aggregation hierarchy. Thus, the connections between objects through object identifiers are essential to the efficiency of query processing and should be represented separately from the database. We introduce the concept of path dictionary and describe how it supports queries of different types. We evaluate the storage overhead, query and update costs of the path dictionary. Compared to the path index, the path dictionary has better overall query and update performance and lower storage overhead.

1 Introduction

As a result of the wide-spread acceptance of object-oriented database systems (OODBSs) and the emerging standardization of the object model and query language [2], implementation issues such as query processing and indexing become a critical factor to the success of OODBSs. Although indexing and signature file techniques have been proposed to support query processing in OODBSs [1,3,5,6,8], they in general introduce large storage overhead and maintenance cost. In this paper, we investigate the problems of indexing and query processing in OODBSs and propose a new indexing scheme and the associated query processing methods.

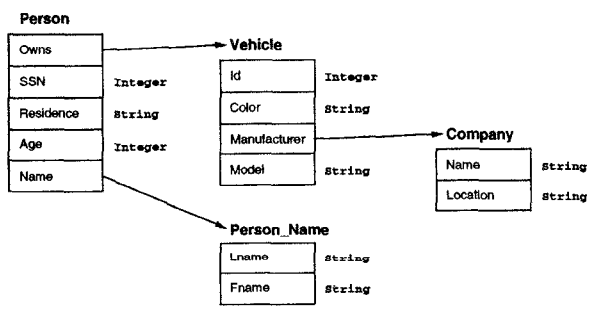


Figure 1: Aggregation hierarchy.

CIKM '94- 11/94 Gaitherburg MD USA

Instead of defining the terminology formally, we use Figure 1 to illustrate some key concepts used in this paper. Figure 1 shows the class definitions for `Person`, `Vehicle`, `Person_Name`, and `Company`. These four classes form an *aggregation hierarchy*. The class `Person` has three *primitive 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 primitive attributes, `Id`, `Color`, and `Model`, and a complex attribute `Manufacturer`, which has `Company` as its domain. `Company` and `Person_Name` each consists of two primitive attributes.

Every object in the database is identified by a unique *object identifier* (*OID*). By storing the OID of an object as an attribute value of another object, we establish an aggregation/association relationship between these two 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 as the *parent* object of O. Objects are nested according to the aggregation hierarchy.

There are two types of queries involving nested attributes. The first type of queries specifies a class from which objects are to be retrieved and a set of predicates on the (nested) attributes of the class. For example, "retrieve all persons who are 50 years old and own vehicles manufactured by Ford" can be expressed as:

```
retrieve Person where Person.Age = 50
        and Person.Vehicle.Company.Name = "Ford"    [Q1]
```

The search condition against the class `Person` involves two predicates: the first predicate involving the simple attribute `Age` of `Person` is called a *simple predicate*, and the second involving the nested attribute `Name` is called a *complex predicate* [4].

The second type of queries retrieves the nested attributes of a given set of objects. For example, "retrieve the manufacturers of the cars owned by persons at the age of 50" can be expressed as:

```
retrieve Person.Vehicle.Company.Name
        where Person.Age = 50                       [Q2]
```

The search condition consists of one simple predicate on the class `Person`, but the query retrieves a nested attribute of `Person`.

To facilitate our discussion, the classes from which objects are retrieved are called *target classes*, and the classes involved in the predicates are called *predicate classes*. In query Q1, `Person` is both a target class and a predicate class,

while Company is a predicate class. In query Q2, Person is a predicate class and Company is a target class.

There are two basic approaches to evaluating a query involving nested attributes: *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. Since a child object is not assumed to carry the OID of its parent object, we must compare the OID of the child object against the corresponding complex attribute in the parent class in order to find the parent object(s) of the child object. This is similar to a relational join when we have more than one child object to start with. 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 bi-directional. Thus, there is no difference between the top-down and the bottom-up approaches. In this paper, we assume there is no inverse references.

To answer Q1 in the top-down approach, we retrieve every object in the class Person, screen out the persons who are not 50 years old, and, for each qualified Person object, retrieve the Vehicle objects and their nested Company objects to check if the manufacturers' name is Ford. In the bottom-up approach, the objects in the class Company are retrieved to examine if their names are Ford. The OID's of the Ford companies are maintained in a set S. Then the vehicle objects in class Vehicle are examined to identify those vehicles made by the companies in S. The qualified vehicle objects are collected in a set S'. Finally, the Person objects are retrieved to see if they are 50 years old and own a vehicle in S'. The efficiency of these two methods depends on the selectivity of the two predicates and the number of objects connected directly or indirectly to the qualified objects in Person and Company after the predicates are applied.

Since query Q2 retrieves the nested attributes of a specific collection of objects, the top-down approach is more efficient. First, we retrieve the objects in Person and check their ages. For those Person objects with Age 50, we traverse along the path Person.Vehicle.Company to retrieve the names of their car makers. However, the bottom-up approach is cumbersome for this query, since it requires the objects in Company to join to the objects in Vehicle, which in turn join to the Person objects before the predicate on Age can be evaluated. This is equivalent to performing a sequence of join *before* applying a selection operation.

From the discussion above, we can see that both the top-down and bottom-up approaches spend a significant part of the query processing cost on accessing intermediate objects connecting two objects. In other words, traversals between the target classes and the predicate classes are expensive. To alleviate this problem, indexes can be used to create implicit reverse links from a nested attribute to the target class. For example, an index can be built to map company names to the person objects, so that, given a company name, the persons who owns the company's vehicles can be directly retrieved without accessing to the vehicle objects. However, this index would not benefit queries such as Q2, because Q2 implies a forward traversal from a class (Person) to the nested attributes (Company). Therefore, another index must be built to map Person objects to Manufacturer objects to bypass Vehicle objects. We can see that in order to support every query pattern a large number of indexes must be used. Unfortunately, since indexes require costly storage overhead and index maintenance, we can only select a number of frequently used target classes and nested attributes for indexing.

Signature file techniques have been proposed to reduce the overall storage overhead and maintenance cost but at the same time support a large variety of queries [6]. For instance, a path signature encodes all the attributes values of the objects along a path, so that by examining the signatures we can discard paths which don't satisfy the predicates without accessing to the objects in the database. The signature file techniques generally have a lower storage overhead and a simpler file structure than indexing techniques. Unfortunately, signature files introduce false drops which lower search performance.

In this paper, we propose a new technique, called the *path dictionary*, to support efficient query evaluation in OODBSs. The idea is to separate the path information from the actual attribute values and store it in a separate path dictionary. Since attribute values are not stored in the path dictionary, the path dictionary is small and can be searched very efficiently. Both forward and backward traversals can be efficiently supported with a careful design of the dictionary.

The rest of the paper is organized as follows. Section 2 introduces the concept, logical organization, and implementation of the path dictionary. Section 3 describes the database operations when a path dictionary is available. Section 4 describes the cost model for estimating the cost of storage and various database operations. Section 5 summarizes the paper.

2 Path Dictionary

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. However, these links shouldn't be confused with pointers in hierarchical databases since they don't enforce a hierarchical structure. Unlike relational databases, which use join operations to connect objects (tuples), OODBSs use OIDs embedded in complex attributes as the main mechanism in accessing nested objects. Unfortunately, previous work didn't fully exploit the linkage information in query processing. In this paper, we examine how the linkage information can be used for processing different types of queries.

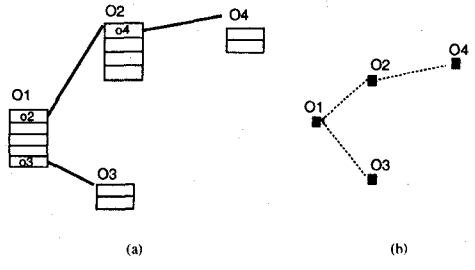


Figure 2: Path information.

Figure 2(a) shows an instance of the aggregation hierarchy in Fig. 1. Fig. 2(b) is the corresponding path information represented as a graph. The graph can be considered as a *conceptual* path dictionary, which can be implemented in different ways. General speaking, the path dictionary extracts the complex attributes from the database to represent the connections between objects. Since simple attribute values are not stored in the path dictionary, it is much faster to traverse the nodes in the path dictionary than objects

in the database. Therefore, we can use the path dictionary to reduce the number of accesses to the database, and, in particular, to avoid accessing intermediate objects when we traverse from one class to another.

To simplify the complexity of the path dictionary, we assume that a path dictionary contains information about a single path in the aggregation hierarchy. Complex graphs may be decomposed into paths, and queries traversing more than one path may use separate path dictionaries for query evaluation. Of course, the path dictionaries corresponding to an aggregation hierarchy can be collectively considered as a single conceptual path dictionary.

Formally, a *path dictionary* is created for a path, $C_1.C_2...C_n$, in the aggregation hierarchy. A path dictionary is a secondary file containing nesting information about the objects in the classes along the path.

The implementation of a conceptual path dictionary must:

- support fast traversal among objects,
- support different types of queries,
- have low storage overhead,
- retain the organization of the database (i.e., the path dictionary is a secondary mechanism, and should not dictate a query processing plan).

We have considered three different schemes for implementing the path dictionary [7]:

1. multi-link scheme: enumerate all the links between pairs of directly connected objects,
2. path scheme: enumerate all the paths connecting two terminal objects,
3. *s*-expression scheme: encode the path information in a recursive list structure.

Due to space constraints, we only describe the *s*-expression scheme in this paper.

2.1 S-expression Scheme

The *s*-expression scheme encodes into a recursive expression all paths terminating at the same object in a leaf class. Since the linking structure denoted by the expression resembles an inward subtree, we call the expression an *s-expression*. The *s*-expression for the path $C_1.C_2....C_n$ is defined as follows:

$S_1 = O_1$, where O_1 is the OID of an object in class C_1 or null.

$S_i = O_i(S_{i-1}[, S_{i-1}])$ $1 < i \leq n$, where O_i is the OID of an object in class C_i or null and S_{i-1} is an *s*-expression for the path $C_1.C_2....C_{i-1}$.

S_i is an *s*-expression of i levels, in which the list associated with O_i contains all of the OIDs of the ancestor objects of O_i. We call it the *ancestor list* of O_i. Except for the objects in C_1, every object on the path has an ancestor list.

The path dictionary for $C_1.C_2...C_n$ 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. The number of *s*-expressions in the path dictionary equals to the number of objects along the path, which do not have a nested object on the path.

Figure 3 is an example of the *s*-expression scheme. It maintains all the linkage information for the objects located on the path `Person.Vehicle.Company`. For example, the first *s*-expression in the figure indicates that there are three paths:

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]),
Vehicle[8](Person[5]), Vehicle[10]())
((Person[9]))

Figure 3: The *s*-expression scheme for a path dictionary.

`Person`[3].`Vehicle`[5].`Company`[1]
`Person`[7].`Vehicle`[5].`Company`[1]
`Person`[4].`Vehicle`[12].`Company`[1]

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. Note that we don't show null OIDs in an *s*-expression for simplicity. 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. 3, the ancestor list for `Vehicle`[3] is empty, meaning that it 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 paths introduced in other schemes [7].

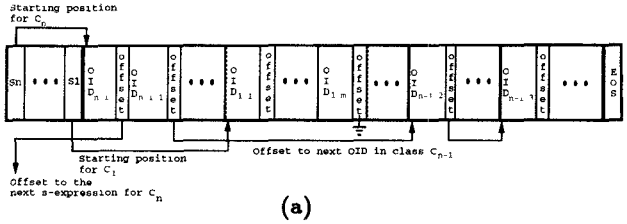
To evaluate Q1 with the path dictionary, we retrieve from the database all `Company` objects corresponding to "Ford" and keep their OIDs in a set S. Then, the path dictionary is searched to locate the *s*-expressions corresponding to the objects in S. Finally, the `person` objects derived from the *s*-expressions are retrieved and their ages are verified. The qualified objects (with `Age`=50) are returned. To answer Q2 with the path dictionary, we retrieve the `Person` objects and check their ages. The OIDs of the qualified `Person` objects are kept in a set S. The path dictionary are then searched for the *s*-expressions corresponding to the objects in S. Finally, all `Company` objects in those *s*-expression are returned. From the examples above, we can see that with the path dictionary we don't have to access any intermediate objects in the database (`Vehicle` objects in the examples) and that it can support both types of queries involving nested attributes.

In addition to keeping track of linkage information, the path dictionary also provides information about shared partial paths, which may be valuable for processing certain kinds of queries. For example, to answer the query "retrieve persons who don't have a car," we can search the path dictionary for the pattern "((`Person`?))" and return the matching `Person` objects. Without the path dictionary, we will have to examine each of the person objects and check if the `Owns` attribute is null or not. Furthermore, a query such as "retrieve the vehicles which are not owned by any person" is very difficult to answer without the path dictionary, since it requires a scan through the `Vehicle` class to collect all OIDs in it, and another scan through the `Person` class to collect all OIDs under the `Owns` attribute (i.e., all vehicles with owners). Then the difference of the two result sets is returned. Using the path dictionary, we simply return all the vehicle objects with an empty ancestor list (i.e., vehicle

objects matching the pattern "`Vehicle?()`").

2.2 Implementing the S-expression Scheme

The organization of the path dictionary and the dictionary searching strategy has considerable impact on the effectiveness of the path dictionary approach. The best way to utilize a path dictionary is to keep the entire dictionary in main memory. However, the path dictionary may be too large to store in main memory. In this paper, we assume that the path dictionary is stored on disk entirely and that a sequential scan is used to locate an s-expression in the path dictionary. In general, indexes can be built on the path dictionary and part of the path dictionary can be cached in main memory to speed up search. These issues will be addressed in a future paper.



(a)

$$OID_{n,1}(\ OID_{n-1,1}(...(OID_{1,1}, OID_{1,2}, ..., OID_{1,m}), ...),\ OID_{n-1,2}(...), OID_{n-1,3}(...), ...)$$

(b)

Figure 4: Data structure of an s-expression.

Figure 4(a) illustrates the data structure of an s-expression for $C_1.C_2...C_n$ and Figure 4(b) is the corresponding s-expression. S_i in the header points to the first occurrence of a class C_i OID in the s-expression. $OID_{i,j}$ is the OID of the j-th C_i object in the s-expression; this is used to access to the objects in the database. Every OID in the path dictionary has an associated offset; the offset associated with $OID_{i,j}$ points to the next occurrence of a C_i OID in the s-expression. To retrieve all the OIDs for class C_i, we start with S_i, which will lead us to the first C_i OID in the s-expression, and following the associated offset value we can reach the next C_i object, and so on. Thus, we can quickly scan through all the OIDs for a class, skipping the OIDs of irrelevant classes. The offset associated with the object of class C_n, however, is pointing to the OID of class C_n in the *next* s-expression, because there is at most one OID of class C_n in an s-expression. At the end of the s-expression is a special end-of-s-expression (EOS) symbol.

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 to an s-expression may cause a page to overflow. When a page overflows, a new page is allocated and some of the s-expressions are moved to the new page. Updates, deletions, and splits may result in unused space in a 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.

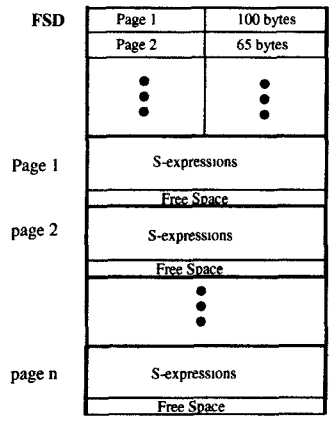


Figure 5: File structure of the path dictionary.

Figure 5 illustrates the physical structure of a path dictionary which comprises the free space directory and the s-expression pages.

3 Database Operations with Path Dictionary

In this section, we describe how various database operations (i.e., queries and updates) are executed with the path dictionary. We assume that a path dictionary for the path $C_1.C_2...C_n$ is available.

Consider a query Q which has C_t as the target class and C_p as the predicate class, where $1 < t, p < n$. First, the objects in C_p are accessed for predicate evaluation. The OIDs of the qualified objects are collected in a set S. Then, the path dictionary is scanned to locate the s-expressions which contain the OIDs in S. From the s-expressions, the OIDs for class C_t objects are obtained, and the objects can then be retrieved from the database. With the path dictionary, we avoid accessing from the database any objects between C_t and C_p.

When changes are made to the database, the path information in the dictionary must be updated. Since updates to primitive attributes won't change the links among objects, they can proceed as normal and have no effect on the path dictionary. When complex attributes are modified, however, the path dictionary must be updated as follows. Suppose O_i has a nested object O_{i+1}, and an update operation changes O_{i+1} to O'_{i+1}. The path dictionary is searched to find all s-expressions containing the OIDs of O_{i+1} and O'_{i+1}. Then, every occurrence of O_i and its ancestor list are removed from the ancestor list of O_{i+1} and inserted into the ancestor list of O'_{i+1}.

An *insertion* operation is to insert an object into a complex attribute of another object. The difference between insert and update is that the complex attribute is null before insertion. For example, `Person`[9], who didn't have a car, just bought `Vehicle`[4]. We have to insert `Vehicle`[4] into the complex attribute `Owns` of `Person`[9]. In general, to insert an object O_{i+1} into a complex attribute of O_i, the path dictionary is searched to find the s-expressions S_{i+1} and S_i which, respectively, contain the OIDs of O_{i+1} and O_i. O_i and its ancestor list is moved from S_i to the ancestor list of O_{i+1} in S_{i+1}.

A *delete* operation on an object's complex attribute replaces the OID stored in the complex attribute with null. In other words, it removes the link between an object and its nested object but without destroying the nested object.

For example, Person[4]'s car was stolen. The link between Person[4] and its car is removed, but the car still exists in the database.

To delete a nested object O_{i+1} from O_i. The path dictionary is searched to find the s-expression S_{i+1} containing the OID of O_{i+1}. After removing O_i and its ancestor list from S_{i+1}, a new s-expression for O_i and its ancestors is created.

A *new* operation creates a new object. When the new object is created but before it is stored in the database, we have to keep track of its parent objects and the object it references. For example, a new car Vehicle[13] is built by Company[1]. The Vehicle object is likely to be created in main memory, assigned values to its attributes, and then stored in the database. To update the path dictionary to reflect the new path information, the OIDs of the parent objects referencing the new object N and the OIDs of their ancestor objects have to be moved into the ancestor list associated with N. Then, the OID of N and its ancestors are inserted into the ancestor list of its nested objects. In order to do this, the path dictionary is searched for s-expressions corresponding to N's parent objects. If found, the OIDs of the parent objects and their ancestors are moved into N's ancestor list. If N references an object O, search path dictionary for the s-expression containing the OID of O and insert N's OID and its ancestors into O's ancestor list.

Object deletion has two possible semantics. The first one only deletes the object itself. The objects nested in the deleted object will not be affected. For example, a car is junked, but its manufacturer still exists. The other semantics is to also delete all the objects nested in the deleted object. For example, when a person is removed from the database, the name object associated with him should also be removed. To distinguish these two semantics, we call the first operation *destruction* and the second *destroy*.

To update the path dictionary when an object O is destructed, we first search the path dictionary for the s-expression containing the OID of O. Then, the OIDs of O and its ancestor list are removed from the s-expression, and a new s-expression for each parent object in O's ancestor list is created.

If O is destroyed, the path dictionary is searched for the s-expression containing the OID of O. Remove the s-expression from the dictionary, and make a new s-expression for each parent object of O.

During deletion and destruction, dangling references in the parent objects to the deleted/destructed object must be removed (nullified). The path dictionary supports this function easily, because it maintains an ancestor list for each object, from which the parent objects can be accessed.

4 Storage Cost and Performance Evaluation

In this section, we formulate the cost models to estimate the storage overhead and query processing performance with the path dictionary. Then, we compare the path dictionary with path index [1] in terms of their storage, retrieval, and update cost. We select path index for comparison, because its structure and applicability to queries is close to that of the path dictionary. The nested index has outstanding performance for certain kind of queries (i.e., queries on a target class to which the indexed attribute is mapped to) [1]. The applicability of the nested index is limited and it requires reverse links built in by the system to support update operations. Therefore, we exclude it from our comparison. A complete comparison among the path dictionary and other techniques (e.g., the nested index, the tree and path signature methods) is under way. In order to facilitate our comparison, we adopt some common parameters from [1]. We use the following parameters to describe the characteristics of classes and their attributes on the path, $C_1.C_2...C_n$, and the structures of the path index and the path dictionary.

- 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 attribute A_i.
- 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.
- k_i: the average number of objects in class C_i referencing the same object in C_{i+1}.
- $UIDL$: 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.
- kl: average length of a key value in the path index.
- ol: the total 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.

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

Performance is based primarily on the number of I/O accesses. Since a *page* is the basic unit for data transfer between the main memory and the external storage device, 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 [1], 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. The values of complex attributes are uniformly distributed among instances of their domain classes.
4. All attributes are single-valued.

and adopt the parameter values in [1]. Table 1 lists the values chosen for the path dictionary.

4.1 Storage Overhead

Path Dictionary

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

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

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

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

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

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

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

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

Therefore, the number of pages needed for the path dictionary is:

$$PDS = FSD + SSP.$$

Path Index

Based on the cost model developed for the path index [1], the number of leaf pages is:
If $XP \leq P$:

$$LP = \lceil D_n / \lfloor P/XP \rfloor \rceil,$$

where $XP = k_1 k_2 ... k_n \cdot UIDL \cdot n + kl + ol$.

If $XP > P$:

$$LP = D_n \lceil XP/P \rceil,$$

where $XP = k_1 k_2 ... k_n UIDL n + kl + ol + DS$, $DS = \lceil (k_1 k_2 ... k_n \cdot UIDL \cdot n + kl + ol)/(UIDL \cdot n + pp) \rceil$.

The number of nonleaf pages is:

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

where $LO = min(D_n, LP)$ and $X < f$. If $X \neq 1$, 1 is added to NLP for the root node.

The total number of pages needed for the path index $PIS = LP + NLP$.

Comparison

Using the formulae developed above, we compare the storage cost of the path dictionary and the path index. We choose a path of 4 classes in the comparison. Also, we fix the cardinality of N_1 to 200000 and the average size of an object to 80 bytes. For the formulae above, we find that the size of the path index varies with $K_1 K_2 K_3 K_4$, so we fix K_4 to 1 for convenience. Therefore, we can vary the ratios of shared references among the classes, i.e., k_1, k_2 and k_3, to observe the change of storage needed for the path dictionary and the path index.

Figure 6 shows the storage overhead for the path dictionary and the path index under different values of k_1, k_2 and k_3. To observe the impact of k_i on the storage overhead of the path dictionary, we vary each k_i, $1 \leq i \leq 3$, from 1 to 1000, and, while k_i is being varied, all k_j, $1 \leq j \leq 3$ and $j \neq i$ is set to 1. We use $PDS1, PDS2$ and $PDS3$, respectively, to represent the storage costs for the path dictionary when k_1, k_2 or k_3 is being varied, and PIS to represent the storage cost of the path index. The result shows that $PDS1 \leq PDS2 \leq PDS3 \leq PIS$. The reason for $PDS1 \leq PDS2 \leq PDS3$ is that when the ratios of shared references for the top classes of the path increase, the number of objects in the bottom classes decreases. In other words, due to assumption (1) and (3), the number of objects in the database are decreasing with respect to the increase of k_i values. $PDS1$ has fewer objects in the database than $PDS2$, which in turn has fewer objects in the database than $PDS3$. PIS has the highest storage cost due to the redundancy in shared references. It replicates the OIDs of the shared nested objects in the leaf nodes of the path index. When the number of objects in the database decreases with respect to the increase of k_i values, the replication of shared nested OIDs increases.

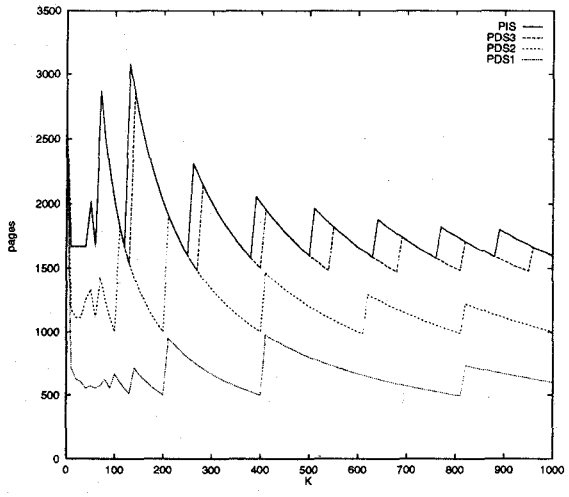


Figure 6: Storage overhead.

4.2 Retrieval Cost

Path Dictionary

To answer a query Q which has C_t as target class and C_p as predicate class, where $1 < t, p < n$, the path dictionary approach will have to retrieve all of the objects in class C_p for predicate evaluation, then completely scan the path dictionary to return objects in C_t. Therefore, the number of pages accessed for object retrieval is:

$$PDR = \lceil N_p S_p / P \rceil + SSP.$$

Path Index

The cost model developed for retrieval with the path index is as follows [1]. The number of pages accessed for object retrieval is:

$$PIR = \begin{cases} h + 1 & \text{if } XP \leq P \\ h + \lceil XP/P \rceil & \text{if } XP > P, \end{cases}$$

where h = height of path index $- 1$.

Without Path Dictionary/Path Index

To answer a query without a path dictionary or path index, we have to consider the following conditions:

1. $t < p$: The query may be answered using the top-down approach, the bottom-up approach, or a combination of both. Assuming that objects in each class are clustered. The number of page accesses for top-down approach:

$$PTR = \lceil N_t S_t / P \rceil + N_t (\lceil S_{t+1}/P \rceil + ... + \lceil S_p/P \rceil).$$

The number of page accesses for bottom-up approach is:

$$PBR = \lceil N_p S_p/P \rceil + \lceil N_{p-1} S_{p-1}/P \rceil + ... + \lceil N_t S_t/P \rceil.$$

The number of pages needed for retrieval without path dictionary or path index is: $NOP = min(PTR, PBR)$.

2. $t > p$: The bottom-up approach is equivalent to joining all classes between C_t and C_p, evaluating the query on the result, and projecting on the attributes we want. Therefore, this query should be answered with a top-down approach, which needs:

$$PTR = \lceil N_p S_p/P \rceil + N_{p|Q}(\lceil S_{p+1}/P \rceil + \lceil S_{p+2}/P \rceil + ... + \lceil S_t/P \rceil),$$

where $N_{p|Q}$ is the number of objects in class C_p which satisfies the predicates in Q.

3. $t = p$: This query does not involve nested attributes, so the number of accesses is: $PR = \lceil N_p S_p/P \rceil$.

Comparison

We use the same parameters and assumptions as we used in evaluating the storage cost. We first compare the costs of object retrievals (1) with the path dictionary, (2) with the path index, and (3) without the path dictionary or path index. The query used in the first comparison of the retrieval cost has C_1 as the target class, C_4 as the predicate class, and the indexed attribute as the predicate attribute. Different k values can be varied for comparison. However, due to space and time constraints, we only evaluate the retrieval cost with respect to the parameter k_2 increasing from 1 to 1000, while other k values are fixed at 1.

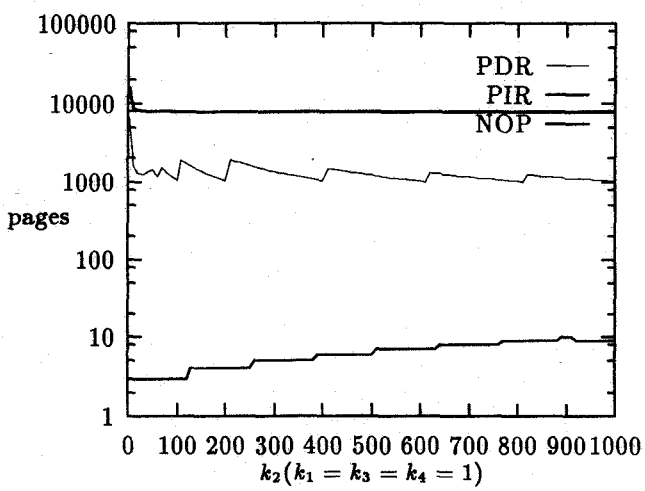


Figure 7: Retrieval cost.

Figure 7 shows that, for the queries with a predicate specified on an indexed attribute, the path index approach is superior to the other two approaches. However, we also observe that the path dictionary approach does reduce the retrieval cost compared to the case without any indexes.

The path dictionary is a more general mechanism than the path index in terms of improving the general performance for different kinds of queries. The path dictionary may be used to process queries with predicate classes located anywhere on the path, while the path index can only be used to process queries with predicates on the indexed attributes. To compare the overall performance of the path dictionary and the path index approaches for different kinds of queries, we evaluate the total cost of the following queries against the classes on the path:

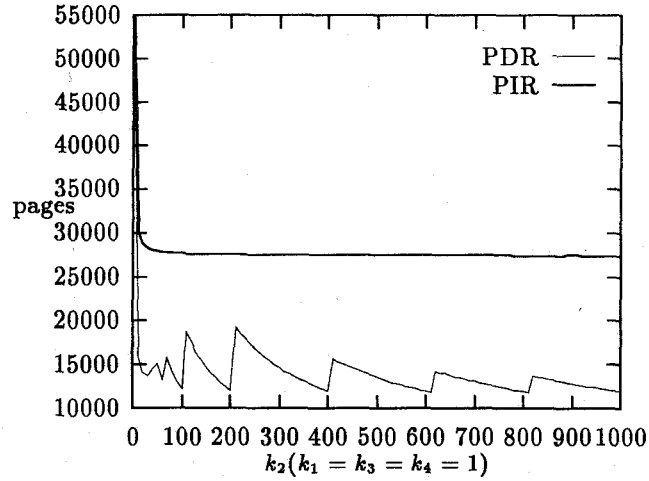


Figure 8: General retrieval cost.

1. Three queries in which the indexed attribute 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 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.

Figure 8 shows that the path dictionary approach has a better overall performance than the combination of path indexing and traditional query evaluation. Comparing to the path dictionary approach, the cost of traditional retrieval approach is so expensive that creating a path index on a single attribute is not sufficient on improving the overall performance of object retrieval. Creating a path index for each of the attributes in the database will dramatically improve the retrieval performance. However, the storage overhead is unacceptable because of the redundancy in the information stored in the indexes which index on different attributes of a path or index on partially shared paths.

4.3 Update Cost

Path Dictionary

Due to space constraints, we only present the cost model for update operation. The cost formulae for insertion, deletion, creation, destruction and destroy can be found in [7]. To simplify the analysis, we do not include the costs due to page overflow caused by insertion or update operations.

To update the complex attribute O_{i+1} of O_i to O'_{i+1}, a path dictionary scan is necessary. The average number of page accesses to locate the s-expressions containing O_{i+1}

and O'_{i+1} is half the size of the path dictionary. Finally, the pages containing s-expressions of O_{i+1} and O'_{i+1} have to be written back to the path dictionary. To simplify our analysis, we assume that O_{i+1} and O'_{i+1} are in different s-expressions and that they are in different pages. Therefore, the number of page accesses for update is: $PDU = SSP/2 + 2\lceil SS/P \rceil$.

Path Index

In order to be fair, we also assume that O_{i+1} and O'_{i+1} have different key values and that they are in different leaf nodes of the path index. To search through the nonleaf 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:

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

O_{i+1} and O'_{i+1} each requires a forward traversal to the indexed attribute to determine their key values.

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

Therefore, the number of pages accesses for update with the path index is: $PIU = 2(CO + FT)$.

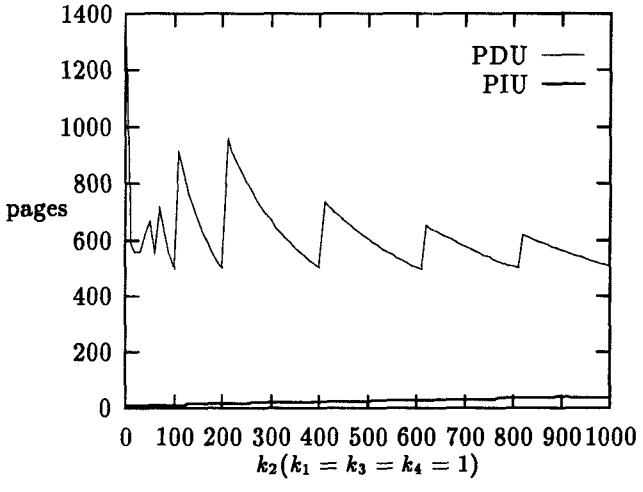


Figure 9: Update cost.

Comparison

The update cost of the path dictionary and the path index is compared in Figure 9. We use the same test cases in the previous comparisons. Comparing to the path index, the path dictionary's update cost is high. This is due to the sequential scan of the path dictionary when locating the s-expressions to be updated. An additional index which maps OIDs to the s-expressions in the path dictionary shall significantly improve the performance [7].

5 Conclusion

Allowing an object to be stored as an attribute of other objects is one of the most important features of the object-oriented database systems. This feature significantly enhances the modeling power of object-oriented data model over other data models. Consequently queries on nested objects has to be supported by OODBSs. The main obstacle for nested queries is the cost of forward or backward traversals along the path between the target and predicate objects. To expedite queries on nested objects, several indexing and signature file approaches have been proposed. However, the storage overhead and maintenance cost of these mechanisms are quite expensive.

One of the common problems of previous indexing techniques is the ineffective support of traversals from an object to its nested objects and vice versa. This paper emphasizes the importance to query processing of keeping nested information on the paths, especially for those frequently traversed ones. We introduce the path dictionary, which is a secondary structure for storing information of a path defined in the aggregation hierarchy. It supports effectively different types of queries. We also developed cost models for the storage overhead, retrieval cost, and various update costs. Using these models, we compare the cost of the path dictionary and path index under different degrees of reference sharing among objects in the classes. We show that the path dictionary yields a dramatic improvement on retrieval cost over the traditional methods while keeping the storage overhead quite reasonable.

We are investigating techniques that combine indexing and signature file techniques with the path dictionary and other implementation issues such as caching the path information in main memory. We believe that the full use of path information embedded among objects will significantly improve the performance of object-oriented database systems.

6 References

[1] 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.

[2] R.G.G. Cattell, ed., *The Object Database Standard: ODMG-93*. Morgan Kauffmann, San Mateo, CA, 1994.

[3] 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.

[4] W. Kim, "A Model of Queries for Object-Oriented Databases," *Proceedings of the IEEE International Conference on Very Large Data Bases*, Amsterdam, 1989, 423–432.

[5] W. Kim, K.-C Kim & A. Dale, "Indexing techniques for object-oriented databases," in *Object-Oriented, Concepts, Databases, and Applications*, W. Kim & F.H. Lochovsky, eds., Addison-Wesley, Reading, MA, 1989, 371–394.

[6] 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.

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

[8] D. Maier & J. Stein, "Indexing in an object-oriented DBMS," *Proceedings of International Workshop on Object-Oriented Database Systems*, 1986, 171–182.