Path Dictionary: A New Approach to Query Processing in Object-Oriented Databases

Wang-chien Lee
Dept of Computer and 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
University of Science and Technology
Clear Water Bay, Hong Kong

dlee@cs.ust.hk
FAX: (852) 2358-1477

October 3, 1995

Abstract

We present a new access method, called the path dictionary index (PDI) method, for supporting nested queries on object-oriented databases. PDI supports object traversal and associative search, respectively, with a path dictionary and a set of attribute indexes built on top of the path dictionary. We discuss issues on indexing and query processing in object-oriented databases, describe the operations of the new mechanism, develop cost models for its storage overhead and query and update costs, and compare the new mechanism to the path index method. The result shows that the path dictionary index method is significantly better than the path index method over a wide range of parameters in terms of retrieval and update costs and that the storage overhead grows slowly with the number of indexed attributes.

1 Introduction

Object-oriented database system (OODBS) has been one of the most prominent areas of database research in the last decade. Many experimental prototypes [7,17,25], and commercial systems [8,14,18] have been introduced in the past decade. In addition to facilitating the design and engineering of traditional database applications, OODBSs provide data modeling mechanisms which support flexible data types, useful relationships (e.g., aggregation/association and specialization/generalization relationships) and allow users to define, query, and update nested entities. This functionality meets the needs of new database applications such as CAD/CAM, CASE, office automation, multi-media systems, and geographic information systems.

The need to reduce the cost of developing, operating, and supporting the above mentioned applications prompted the rapid development of OODBSs. However, to the success of OODBSs, implementation and performance issues play an important role. Efficient techniques for query processing and indexing are critical [1,3,10,12,13,15]. Although indexing techniques have been proposed to support query processing in OODBSs [4,5,16,24], 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.

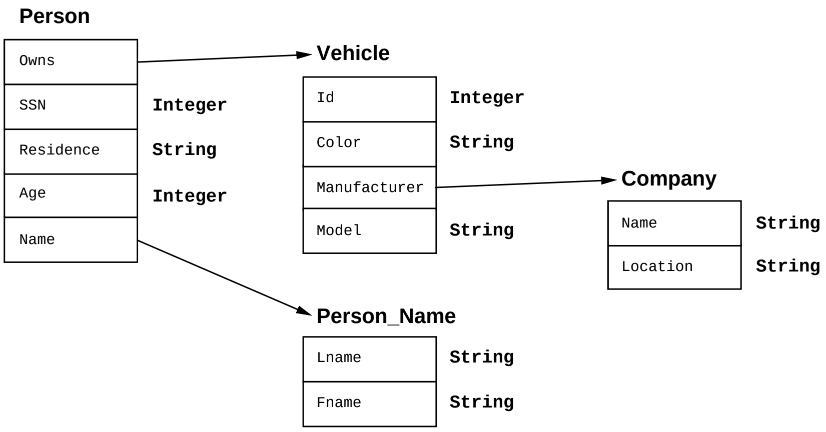

Figure 1: Aggregation hierarchy.

The object-oriented data model is based on the following basic concepts. In the data model, a real-world entity is represented as an object, which consists of methods and attributes. While methods, which are procedures and functions associated with the object, define the reactions of the object in response to messages from other objects, attributes represent the inner state of the object. Objects sharing the same methods and attributes are grouped into *classes*. The concept of class allows 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.

Another feature of OODBS is the specialization/generalization relationships between classes. In this paper, we call it *inheritance* relationship. The inheritance relationship organizes classes into an *inheritance hierarchy*. Inheritance allows a class C to be defined as a specialization of another class C'. C is called the *subclass* of C' and C' is a *superclass* of C. A subclass inherits attributes and methods from its superclasses. A subclass can have more attributes and methods than its superclass.

Since similar objects are grouped into a hierarchy of classes, whether to create a single index for the whole hierarchy of classes or to build an index for each of the classes is an interesting indexing problem.

Figure 2 is an example of inheritance hierarchy among the class `Vehicle` and its subclasses `Automobile`, `DomesticAuto`, `ForeignAuto`, and `Truck`. In addition to the attributes inherited from the superclass `Vehicle`, objects in class `Automobile` have an additional attribute `Model`. Therefore, the objects in class `DomesticAuto` have attributes `Id`, `Color`, `Manufacturer`, `Model`, and `Country`.

Every object in an OODBS is identified by an *object identifier* (OID). The OID of an object

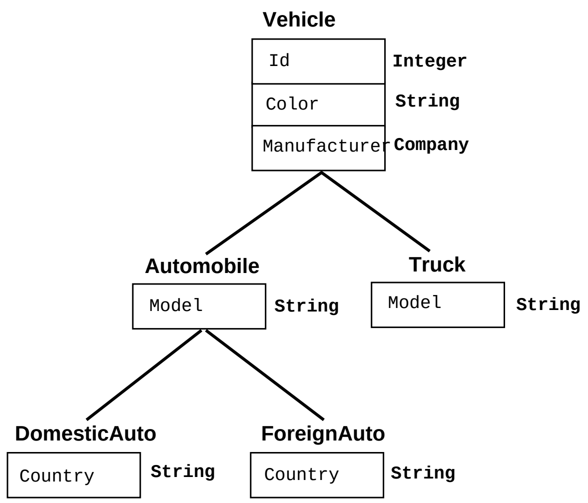

Figure 2: Inheritance hierarchy.

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.

OODBSs support queries involving nested objects. These queries are called *nested queries*. 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: 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: $>, \geq, <, \leq, *$, and ?[1].
     - Inequality: $\neq$.
   - Complex: The predicate is specified on a complex attribute. Depending on whether or not an OID is specified in the predicate class, this class can be further divided into:
     - Exist: An OID is specified in the predicate.
     - Nonexist: No OID is specified in the predicate.

[1] * and ? are partial string matching operators.

In the following, we give examples for each type of the nested queries. We will use some of these examples to illustrate the query processing strategies described in this paper. Note that `Company`[i] and `Person`[i] denote the OIDs of the ith object in class `Company` and `Person`, respectively.

Q1: Retrieve persons who owns cars made by "GM". (TP-Simple-Equality)

Q2: Retrieve persons who owns cars made by "G*". (TP-Simple-Range)

Q3: Retrieve persons who owns cars not made by "GM". (TP-Simple-Inequality)

Q4: Retrieve manufacturers of the cars owned by persons at the age of 50. (PT-Simple-Equality)

Q5: Retrieve manufacturers of the cars owned by persons older than 50. (PT-Simple-Range)

Q6: Retrieve manufacturers of the cars owned by persons not at the age of 50. (PT-Simple-Inequality)

Q7: Retrieve persons who have a car made by `Company`[1]. (TP-Complex-Exist)

Q8: Retrieve persons who don't have a car. (TP-Complex-Nonexist)

Q9: Retrieve vehicles which are owned by `Person`[1]. (PT-Complex-Exist)

Q10: Retrieve vehicles which are not owned by any person. (PT-Complex-Nonexist)

There are three basic approaches to evaluating a nested query: *top-down*, *bottom-up* and *mixed* 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 required 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 bi-directional, resulting in no difference between the top-down and the bottom-up approaches. In this paper, however, we assume there is no inverse references.

Let's consider the above query examples. To answer Q1 in the top-down approach, the system has to retrieve all of the objects in class `Person`, then retrieve the `Vehicle` objects of the `Person` objects and their nested `Company` objects to check the manufacturers' names. Finally, those persons who own GM cars are returned. In the bottom-up approach, the objects in class `Company` are retrieved to examine if their names are GM. The OID's of the GM 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 find out if their cars are one of the vehicles in S'. The other TP queries, such as Q2, Q3, Q7 and Q8, can be evaluated similarly.

The top-down approach is more effective for PT queries, which retrieve nested attributes of some specific collection of objects. Take Q4 as an example, the objects in `Person` are first retrieved to examine their ages. Those objects with `Age` of 50 are then traversed along the path of

`Person.Vehicle.Company` to retrieve the names of the automobile makers. The bottom-up approach, in contrary with the top-down approach, is cumbersome in this query. It requires all of the objects in class `Company` to join to the objects in class `Vehicle`, and the result is further joined to the `Person` class. Then, the names of the auto makers corresponding to the 50-year-old auto owners are returned.

The other PT queries, except for Q10, may be evaluated in a manner similar to Q4. To answer Q10, we have to retrieve person objects and collect in a set S the vehicle objects which have owners. Then, the set S is subtracted from the `Vehicle` class to return the vehicle objects which have no owners.

The performance of the top-down and the bottom-up approaches is strongly dependent on the distribution of objects located in the classes to be traversed. No matter where the predicate classes are located, the bottom-up approach will always scan through all objects in the classes on the path. However, the number of objects fetched by the top-down approach is at most the number of objects in the top class times the number of classes on the path.

As a result, it is intuitive to conclude that the top-down evaluation is more appropriate than the bottom-up method when the ancestor classes have fewer objects than the other classes on the paths or when the number of path traversals are reduced due to the predicate evaluations in the top classes. On the other hand, the bottom-up evaluation is more suitable when the ratio of object sharing from ancestor classes to their nested classes is high. In other words, the bottom-up approach prevails when the numbers of objects in ancestor classes are much larger than that of their nested classes.

Generally speaking, the top-down approach has an advantage over the bottom-up approach for queries which have predicate classes located near the top of the paths to be traversed. If the number of qualified objects in these predicate classes is small, the number of forward traversals will be small. For example, if there are 10 persons who are 50 years old in the class `Person`, at most 10 `Vehicle` objects and 10 `Company` objects will be fetched. This kind of queries only benefits the top-down method and has no advantage for the bottom-up method. Another disadvantage of the bottom-up approach is that it requires a lot of internal memory due to the breath-first style of the join operations. On the other hand, the top-down approach may choose a depth-first style of forward traversal which doesn't require much internal memory.

Many access methods have been proposed to support complex queries in OODBSs. In particular, three techniques, namely, indexing [2,3,5,24], signature file [9,21,28] and data dictionary [19,20], have been proposed recently. In this paper, we discuss the problems with query processing in OODBSs and present the path dictionary organization.

The rest of the paper is organized as follows. Section 2 summarizes the access methods proposed in the literature. Section 3 introduces the concept of path dictionary and possible approaches to implement the path dictionary. The implementation of the s-expression scheme for path dictionary and its retrieval and update operations are discussed in Section 4 and Section 5 respectively. In Section 6, we present the performance analysis and comparison of various techniques discussed. Finally, we conclude the paper in Section 7.

2 Related Work

A relational database consists of a group of separated relations, which are related through primitive key values. The join operation is used to connect these relations. In object-oriented databases, objects of various classes are related by object identifiers, which leads to the special structure of nested objects. Traversal through the bridges built upon OIDs is a natural way of evaluating OODBS queries. Therefore, nested queries implies traversal of objects along the path between the target class and the nested attributes.

From our discussion on traversal methods, we can see that a significant part of the query processing cost is spent on accessing intermediate objects between the target class and the predicate classes. Techniques based on indexing or signature file methods have been proposed to expedite the processing of queries. According to our observation, the essence of these techniques is to reduce physical traversals of intermediate objects between the target class and the predicate classes.

2.1 Indexing Techniques

The idea behind indexing techniques for nested query processing is to map a value of certain attribute to some ancestor objects which directly or indirectly own the attribute values. The indexing mechanisms implicitly create a direct reverse link from a nested attribute to an ancestor class. As a result, the goal of bypassing the intermediate objects is achieved by scanning indexes. Indexing techniques are effective and will be efficient as long as the overhead they introduce is smaller than the saving gained from avoiding intermediate object traversal. Unfortunately, most indexing techniques require costly storage overhead and expensive index maintenance. Therefore, they can't be applied on too many attributes. Only some frequently queried target classes and predicate attributes can be chosen to create indexes.

2.1.1 Indexing Aggregation Hierarchy

Multiple Index [5,24] is the first of the indexing techniques for OODBSs. It creates an index for each edge on the path from a nested attribute to the target class. It is like creating a reverse link for each edge along the path. To answer a query involving the indexed attribute and target class, index scans may be used to replace physical access to intermediate objects for backward traversals from the nested attribute to the objects in target class. Although several indexes are created for a given path and thus many index scans are necessary for a query evaluation, this organization is flexible for creating indexes sharing a path without introducing much duplicated overhead.

Nested Index and *Path Index* [5] map a specific nested attribute to the target class and to the classes located along the given path, respectively. Like multiple index, they separately create implicit reverse links from the nested attribute to the target class and the classes appearing on the path. Only one index scan is needed to reach the target classes from the nested attribute. Although both techniques are very effective, they require high storage cost and expensive update maintenance. Thus, they are very expensive when many attributes are indexed. Further, the nested index requires system-supported reverse links among objects in the path to efficiently update the index [5].

These indexing techniques cannot support all of the nested queries we classified. PT queries implicitly suggest a forward traversal to the nested attributes. Thus, they cannot benefit from the reverse links built by the indexes. Support of TP-Simple-Inequality queries is problematic, since inequality cannot be easily supported by indexing techniques. We can create indexes for TP-Complex-Exist queries using OIDs as the key. However, the applicability of these indexes is limited, because the only meaningful operator for the kind of indexes is equality. Finally, TP-Complex-Nonexist queries are not supported by these indexing techniques either.

Field Replication Technique [26], as its name suggested, replicates attributes of nested objects into their ancestor objects. Therefore, nested attribute values that would normally be accessed through forward traversal are replicated such that expensive traversals may be avoided. The problems with field replication are that it imposes a structure change to the original database and that its update cost is expensive. In order to improve the update performance, *inverted path* was introduced to implement reverse links along the path [26]. The idea of inverted path is similar to that of multiple index. Therefore, this organization can be used to support backward traversals and some of the forward traversals where the nested attributes are replicated.

Join Indices [27] were proposed for improving joins in relational database systems. It may be created by joining two relations, say R and S, and project the corresponding tuple identifiers from R and S into a (join index) relation of parity 2. In order to facilitate fast access to the join index relation, two copies of the join index are usually maintained. One copy is clustered on the tuple identifiers of R and the other is clustered on that of S. The join indices can also be used in OODBSs. A join index may be used for each direct connection of classes along a given path. Therefore, it may be implemented as two sets of multiple indexes, which will allow both directions of traversal. The tradeoff for the bi-directional traversal is to double the storage overhead.

Access support relations [11] is a generalization of the join indices for OODBSs. Instead of supporting traversal (or join) of two connected classes (relations), access support relations support the traversal along a path of arbitrary length. The relations may be created by joining all of the classes along the path and project the object identifiers from the classes on the path. Similar to join indices, two copies of an access support relation are stored and clustered correspondingly on the OIDs of objects in the two end classes of the path. Therefore, traversals from either end class of the path to any class on the path can be supported.

Direct Links [22] maintain links connecting objects in two separate classes for fast object traversal. Since objects in the intermediate classes between the target class and the predicate class usually are not directly related to the query, much computing cost will be saved if they are not accessed during query processing. Therefore, the direct links between two classes provide short cuts for object traversals. The direct links are similar to projecting the OIDs of the end classes on the access support relations. Thus, it may go from one end of the path to the other end efficiently. Moreover, in order to facilitate associative search of the direct links, indexes can be built to map attributes of either end classes to the direct links organization. Therefore, both forward and backward traversals are supported with reasonable storage overhead. However, like nested index, system-supported reverse links among objects in the path is needed to efficiently update the direct links.

2.1.2 Indexing Inheritance Hierarchy

Issues in building a single index for the classes in an inheritance hierarchy were studied in [16]. It compares the *Single-Class Index*, which maintains a conventional index for each class of the hierarchy, to the *Class-Hierarchy Index*, which maintains a single index for the whole hierarchy of classes. The conclusion is that the class-hierarchy index is superior to the single-class index as long as there are more than two classes in the inheritance hierarchy [16]. However, the study in [16] is confined to single level indexes for primitive attributes of a class without considering nested attributes.

H-trees is a hierarchical indexing organization supporting efficient associative search on objects based on the inheritance hierarchy [23]. The organization is tailored to supporting object retrieval from a single class as well as from an inheritance hierarchy of classes. The H-tree indexes of the classes are structured in accordance with the inheritance hierarchy. A B^+-tree is created for each class. The nested indexes are connected to their parent indexes by pointers, which associate the nodes in the nested H-trees to their parent H-trees according to the indexed values. As a result, searching for a value against an inheritance hierarchy of classes requires only a full search on the root H-tree and partial searches on the nested H-trees.

The study of Nested-Inherited Index [2] takes both of the aggregation and inheritance relationships into account. Actually the inheritance index scheme is an extension of the path index to cover the inheritance relationship among classes. Yet another improvement of the nested-inherited index over the path index is to store the ancestor information of objects in a network of indexed auxiliary records. This mechanism facilitates the update maintenance of the nested-inherited index. However, the storage overhead involved is tremendous.

2.2 Signature File Techniques

The signature file techniques, in contrary with the indexing techniques, use abstracted information stored in signature files to avoid actual retrieval of intermediate objects located on the paths from the top class to the nested attributes. An object signature is an abstraction of the information stored in the (nested) attributes of the object. When processing a query, a query signature is formed to match with object signatures. An object signature which fails to match the query signature guarantees that the corresponding object can be ignored. Consequently unnecessary object accesses are avoided. Only the objects passing the signature matching are traversed to eliminate false-drops. The signature file techniques generally have a much lower storage overhead and a simpler file structure than indexing techniques. They are particularly good for queries which requires forward traversals (i.e., PT queries) and queries involving a large number of the attributes. The latter is because values from many attributes, not just one attribute, are encoded in the signatures.

Tree signature and *path signature* are two of the signature file schemes proposed by the authors [21]. The tree signature scheme generates the signature of an object by hashing all of its direct and nested primitive attributes into a signature. Therefore, the signature is an abstraction of information directly stored or nested in the object. The path signature scheme generates the signature of an object by hashing all of its direct primitive attributes and nested primitive attributes located in a given path. Consequently, the abstraction of information in the path signature scheme is limited to the primitive attributes along the path. For a given aggregation hierarchy, the information embedded in a path signature is less than that of a tree signature. The cost and complexity of maintenance for the path signature scheme is less than that of the tree signature scheme, because the scope of the index is confined to a path, not the whole aggregation hierarchy. Since there are fewer attributes involved, the path signature scheme is more effective in filtering unqualified objects. On the other hand, the tree signature scheme provides a more general support for queries involving any attribute in the database.

Signature replication technique [28] generates object signatures from the direct attribute values of the objects. Instead of using only OIDs, the object signature of an object and its OID are stored as complex attributes of the parent objects. Therefore, the object signatures are first used to screen out unqualified nested objects before performing forward traversal. Like the field replication technique, this organization will change the structure of the original database. Further, it has expensive update and maintenance costs. Also, for a nested query involving a long path with predicate class at the far end, the signature replication method may save only the final step in the traversal instead of bypassing all intermediate classes in the path.

3 Path Dictionary Index

In this paper, we propose a new technique, called the *path dictionary index* (PDI), to support efficient query evaluation in object-oriented databases. The PDI is a separate access structure for the object database. It consists of two parts: the path dictionary supports efficient object traversal and the identity and the attribute indexes support associative search. The identity and attribute indexes are built on top of the path dictionary. Figure 3 illustrates the overall architecture of the path dictionary index. Upon the receipt of a query, the query processor will efficiently evaluate the predicates, if any, using the attribute indexes and then traverse to the target classes using the path dictionary. In other words, the PDI approach reduces the cost of query processing by supporting both associative search and object traversals.

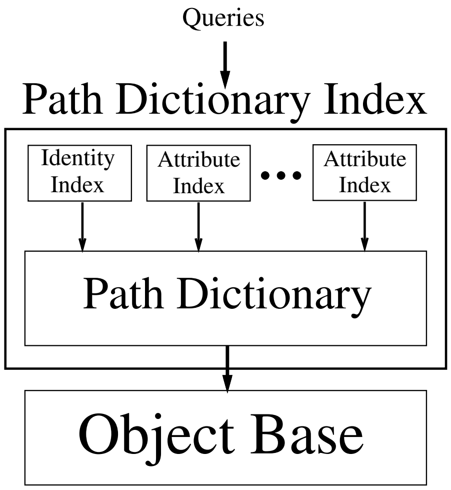

Figure 3: Path dictionary index.

3.1 Path Dictionary

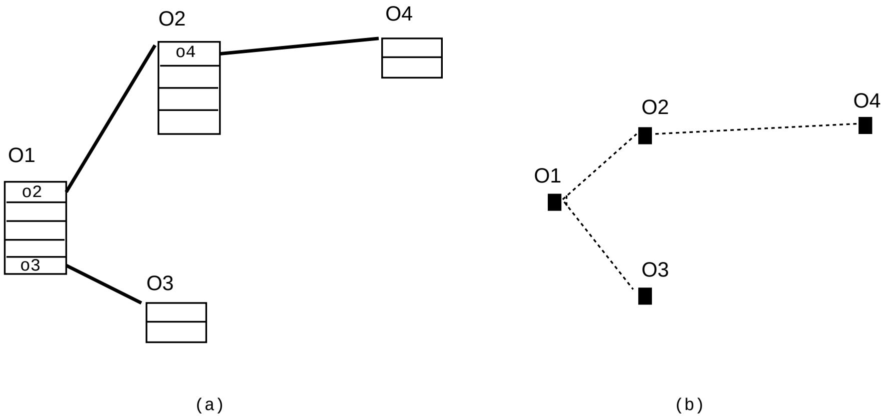

Figure 4: (a) A database instance, (b) Path information.

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. Figure 4(a) shows some object instances corresponding to the aggregation hierarchy in Fig. 1. Fig. 4(b) is a *conceptual* path dictionary storing the connections among the objects in the database. General speaking, the path dictionary extracts the complex attributes from the database to represent the connections between objects. Since primitive 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, the path dictionary can be used to reduce the number of accesses to the database, and, in particular, to avoid accessing intermediate objects when traversal from one class to another is performed.

Compared to other approaches which use reverse links from nested attributes to the target objects or store abstract information of (nested) attributes with the objects, the path dictionary prevents unnecessary object accesses by storing the path information among the objects in a separate access structure. The path dictionary provides shortcuts for both forward and backward traversals of the objects on a given path. As a result, it is suitable for general queries, whether they imply top-down or bottom-up evaluation.

When the connections between objects is very complex, the path dictionary can be decomposed into a number of simpler path dictionaries. For instance, a long path may be decomposed into several small path segments for design and efficiency reasons. The configuration issues involved with path and nested indexes were discussed in [6]. In this paper, we assume that only one path dictionary is built for an aggregation hierarchy.

3.2 Attribute Index

While the path dictionary supports fast traversal among objects, it by itself will not help predicate evaluation which involves finding objects meeting certain conditions specified on their attribute values. To facilitate associative search, the PDI provides attribute indexes which map attribute values to the OIDs in the path dictionary corresponding to the attribute values. As usual, attributes which have high selectivity and are frequently used in queries should be indexed.

Instead of mapping attribute values directly to objects (as in the nested index and path index methods), the attribute indexes map attribute values to path information stored in the path dictionary. The path dictionary serves as a shared structure for object traversal and as a level of indirection from attribute values to the physical objects. The separation of support for traversal and associative search contributes to the low storage overhead and maintenance cost of the PDI approach.

As a result of the separation, as many attributes indexes as necessary can be built on top of the path dictionary without incurring extraordinary growth in storage overhead. The attribute indexes provide general support for various kinds of queries as well as reduce the cost of query evaluation. The more attributes involved in a query, the more options are available for query optimization.

The attribute indexes can be organized as tree-structures, such as B^+-trees. However, in order to share the path information with other attribute indexes and to reduce redundant updates on path dictionary, the location of path information in the dictionary should be stored as the leaf nodes of the indexes.

3.3 Identity Index

Since OIDs are used to describe the path information among objects, it is often necessary to obtain from the path dictionary path information associated with a given OID. In order to efficiently support this operation, an identity index is provided to map OIDs to the locations in the path dictionary where the OIDs can be found. Since identity search is important for retrieval and update, the identity index significantly reduces the cost for retrieval and update operations. Similar to the attribute indexes, the identity index is organized as a separate search tree on top of the path dictionary.

3.4 Design Considerations

When designing a new access method to support nested query, we considered the following requirements:

- It must be a secondary file organization.
- It must support both forward and backward traversals.
- It must be easily coupled with attribute indexes to support associative search techniques.
- It must support multiple access methods.
- It must support various nested queries efficiently.

The path dictionary index is designed to meet each of the requirements. It is desirable to retain the organization of the original database and keep the index structures transparent to the users, so we like the access structure to be implemented as a secondary file without the need of modifying the database structure at the system level. Also, we like the access method to be general enough to support both forward and backward traversals and a large variety of queries (i.e., the queries we classified in section 1). Since most database queries involve associative search on simple attributes, the method must support both traversal and associative search. Also, we like the structure to be general enough to support many query evaluation plans for optimization. Finally, we like the structure to be useful for various kinds of queries instead of being useful only for certain kinds of queries. The design of PDI as presented above meets each of these requirements.

3.5 S-expression Scheme

In the following, we present the s-expression scheme and discuss the multi-link and path schemes for the path dictionary implementation.

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.[2] 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.

Note that, by our definition, s-expressions comprise OIDs of objects. However, through out the paper, we might use "objects" to refer to the OIDs of objects in the s-expression when it does not cause confusion.

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

[2]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.

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 5: Examples of the s-expression scheme.

paths denoted by the s-expression. Thus, the number of s-expressions corresponding to a path equals to the number of objects along the path which don't have a nested object on the path. Several s-expressions are shown in Figure 5. 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:

`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. On the other hand, an s-expression may contain null ancestor lists indicating that the object is not referenced by any other object. For instance, in the third s-expression in Fig. 5, 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, thus avoiding redundant partial path information introduced in other schemes.

Due to the inherent tree structure, the s-expression scheme supports naturally 1:1 and 1:N relationships. N:M relationship can be easily supported by extending the s-expression. The analysis in this paper, however, is based on 1:1 and 1:N relationships, because these are most common in database applications. Furthermore, since the goal of the analysis is to compare the performance of PDI against the nested and path index methods, given the decoupling of traversal and associative search support the performance advantage of PDI for N:M relationships is even more conspicuous than that of 1:1 and 1:N relationships.

4 Implementation of the s-expression Scheme

Figure 6(a) illustrates the data structure of an s-expression for $C_1C_2...C_n$. SP_i in the header points to the first occurrence 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. The offset associated with θ_i, $2 \leq i \leq n$, points to the next occurrence of θ_i in the s-expression. The OIDs for class C_1 don't have offset

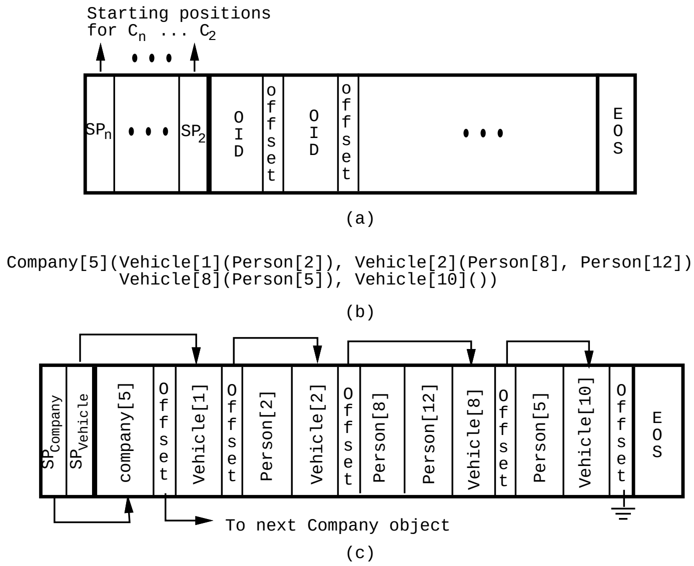

Figure 6: Data structure of an s-expression.

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. 6(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. 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.

Figure 7(a) illustrates the physical structure of the path dictionary index, which consists of the free space directory, the s-expression pages, and attribute and identity indexes. Fig. 7(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. In the identity index, the OIDs are used as the key value. The *s-address* in the leaf node is the address of the s-expression corresponding to the OID in the same leaf node. The page pointers in a nonleaf nodes are pointing to the next level of nonleaf nodes or to the leaf nodes.

Figures 7(d) and (e) show the structures of an attribute index's leaf node record and non-

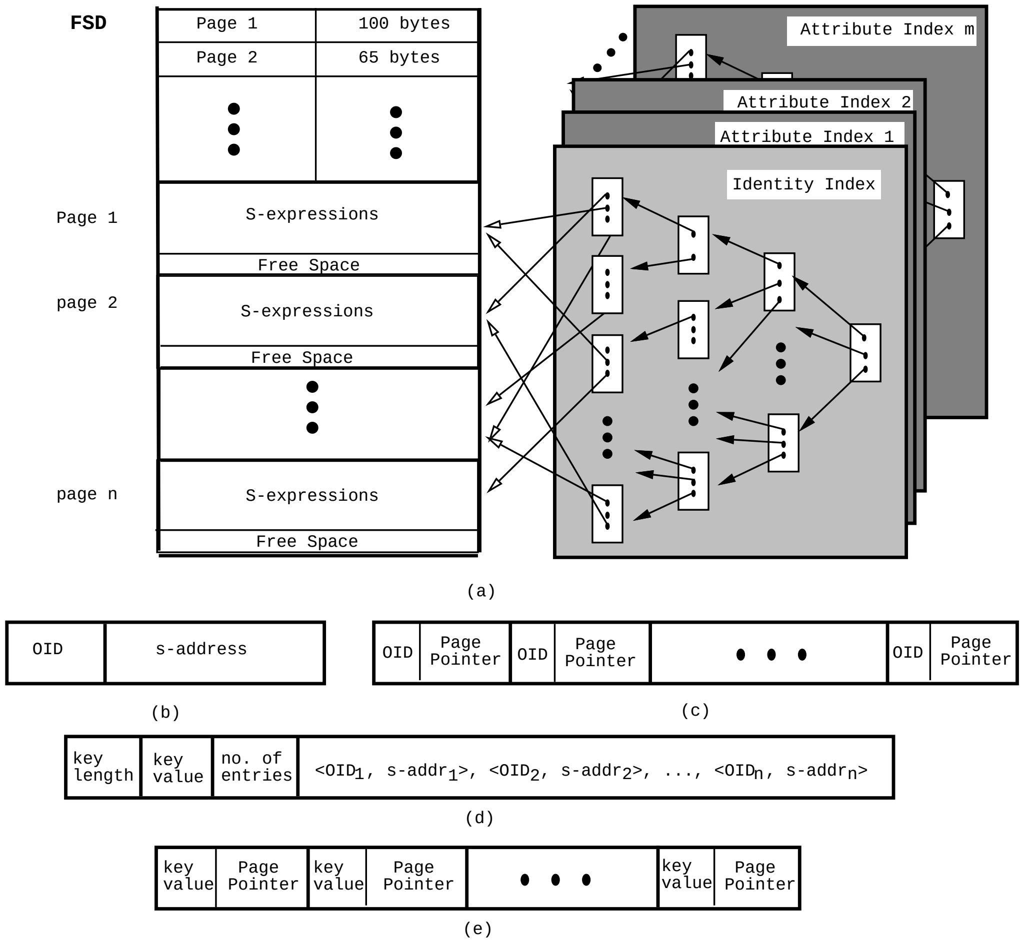

Figure 7: Path Dictionary Index. (a) Structure of the path dictionary index; (b) leaf node record of the identity index; (c) nonleaf node of the identity index; (d) leaf node record of an attribute index; (e) nonleaf node of an attribute index.

leaf 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 all of the objects in the predicate classes.

5 Retrieval and Update With Path Dictionary Index

In the following, we discuss the strategies used to process nested queries and updates with the path dictionary index.

5.1 Retrieval Operations

In order to simplify our discussion, we assume that the query has only one predicate attribute, which is indexed by an attribute index, $Index_p$. We specify a nested query Q as having C_t as the target class and C_p as the predicate class, where $1 \leq t, p \leq n$. 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: Attribute indexes have advantages of processing simple predicates with equality and range operations. For inequality operation, the path dictionary is still better than the conventional traversal approaches.

Equality – 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 own 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.

Range – For the range query, we use the lowest key value in the range to search the attribute index for leaf node record containing the lowest key value. Then we sequentially search the leaf node records until the record containing the highest key value in the range is reached. From those leaf node records, we obtain the addresses of the s-expressions corresponding to the predicate objects with an attribute value in the specified range. As before, we obtain the s-expressions and return the OIDs for C_t from the s-expressions. This strategy prevents repeated scanning on the attribute index.

Use Q5 as an example. Assume that the attribute `Age` of `Person` is indexed by $Index_{age}$. We use "50" as the search key on $Index_{age}$ to arrive at the leaf node record corresponding to 50. Starting from the next leaf node record, which is corresponding to the next age greater than 50, we sequentially scan the leaf nodes and use the addresses in the records to access the corresponding s-expressions. From the s-expressions, the OIDs of the `Company` objects are returned. In this example, the scanning of the leaf nodes continues until there is no more leaf node records left.

Inequality – The attribute indexes cannot be used for predicates with inequality operation. However, the path dictionary can still improve the processing of this class of queries. The objects in C_p are retrieved from the database for predicate evaluation. The OIDs of the qualified objects, θ_p's, are collected in a set P. The OIDs in P are used as keys to search the identity index for the s-expression addresses. Then the s-expressions in the path dictionary are accessed to return the OIDs of the qualified objects in the target class (i.e., θ_t's), with which the target objects can be retrieved from the database. Using the path dictionary, we avoid accessing from the database any objects between C_t and C_p.

To answer query Q3, "retrieve persons who own cars not made by GM", all of the objects in class `Company` are accessed to collect OIDs of GM company objects into a set P. The OIDs in P are then used as keys to search the identity index for s-expressions. Finally the OIDs of objects in the target class are derived from the found s-expressions and returned.

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 Q7 "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 Q8 "retrieve persons who don't have a car" as an example. The s-expressions in the path dictionary are sequentially searched for the pattern "((`Person`?))" and the matching `Person` objects are returned. 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 Q10 "retrieve vehicles which are not owned by any person" (the PT case), we sequentially scan the path dictionary and simply return all of 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.

For queries with predicates on more than one indexed attributes, the evaluation is accomplished by first separately scanning the attribute indexes, with the results unioned or intersected according to the Boolean condition in the query. For each index, addresses of the s-expressions, corresponding to objects which passed the predicates, are collected. Next, these sets of s-expression addresses are unioned or intersected to generate a set S in accordance with boolean combination of the search conditions. Using addresses in S to access the s-expressions. The OIDs of objects corresponding to unindexed predicate classes are derived from the s-expressions and used to access to the objects in the database. After the evaluation of the search condition is completed, the OIDs of the qualified objects in target class are returned from s-expression.

Comparing to the nested index, the path dictionary index approach needs to derive the target objects from the s-expressions in the path dictionary, while the nested index will directly return the qualified target objects through index scan. However, with queries involving unindexed attributes, the nested index needs to traverse the objects in the database in order to evaluate the predicate, while the path dictionary index can directly access to the objects in the target class. Besides, with queries involving more than one indexed attribute, the cost of index scans is about the same for both methods. Although the path dictionary index may cost more when the number of s-expressions accessed is large, we expect the path dictionary index to have about the same retrieval performance as the nested index for queries with typical selectivity and reasonably restricted conjunctive conditions.

5.2 Update Operations

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. In the paper, we only describe the update operation.

When changes are made to the database, the path information in the dictionary must be updated. Since updates to simple attributes won't change the links among objects, they have no effect on the path dictionary. When complex attributes are modified, however, the path dictionary must be updated. However, owing to the attribute indexes, updates on the simple attributes of the objects located along the indexed path induces updates on the PDI. 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.

   Note that an alternative approach is to traverse from O_{i+1} and O'_{i+1} to their nested attributes, then use the attribute values to scan through the attribute index to locate the s-expressions and update the path dictionary. The update of the attribute index may be done while locating the s-expression addresses. Whether this method is better than the previous one depends on whether or not traversing through the nested attributes is more expensive than going through the identity index.

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

6 Storage Cost and Performance Evaluation

In this section, we formulate the cost models for the path index, path dictionary, and path dictionary index methods to analyze their storage overhead and query processing performance. Then, we compare their performance in terms of their storage, retrieval, and update costs. 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. We didn't select the nested index method even though it has outstanding performance for certain kind of queries (i.e., queries on a target class to which the indexed attribute is mapped to) [5], because the applicability of the nested index is limited. Most importantly, it requires reverse links built in by the system to support effective update operations. The path dictionary and path dictionary index are general enough to provide significant support for both TP and PT queries. The path dictionary method can serve as the baseline performance for the path dictionary index method, where no attribute index is used.

In order to facilitate our comparison, we adopt some common parameters from [5]. 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 the three 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 main memory and external storage, we use it to estimate the storage overhead and the performance cost. All lengths and sizes used above are in *bytes*.

Table 1: Parameters of the cost models.

P	=	4096
$UIDL$	=	8
pp	=	4
$OFFL$	=	2
SL	=	2

FSL	=	2
EL	=	4
kl	=	8
ol	=	6
f	=	218

To directly adopt the formulae developed in [5], 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 [5]. Table 1 lists the values chosen for the path dictionary.

6.1 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 class on the path $C_1C_2...C_n$, the number of pages needed is [5]:

$$LP = \begin{cases} \lceil U_{n,j}/\lfloor P/XP \rfloor \rceil, & \text{where } XP = PN \cdot UIDL \cdot n + kl + ol \quad \text{if } XP \leq P \\ U_{n,j}\lceil XP/P \rceil, & \text{where } XP = PN \cdot UIDL \cdot n + kl + ol + DS, \\ & DS = \lceil [PN \cdot UIDL \cdot n + kl + ol]/P \rceil (UIDL \cdot n + pp) \quad \text{if } XP > P, \end{cases}$$

and $PN = k_1k_2...k_{n-1}q_{n,j}$. The number of nonleaf pages is:

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

where $LO = min(U_{n,j}, LP)$ and $X < f$. If $X \neq 1$, NLP is increased by one to account for the root node. The total number of pages needed for the path index is:

$$PIS = LP + NLP.$$

Path Dictionary

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:

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

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 total number of objects in the database is:

$$TOBJ = N_1 + N_2 + ... + N_n = NOBJ \cdot N_n.$$

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

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

The number of nonleaf pages is:

$$NLP_{identity} = \lceil LP_{identity}/f \rceil + \lceil \lceil LP_{identity}/f \rceil / f \rceil + ... + X,$$

where $X < f$. If $X \neq 1$, $NLP_{identity}$ is increased by 1 to account for the root node. The total number of pages needed for the identity index is:

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

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

$$PDS = FSD + SSP + IIP.$$

Path Dictionary Index

The number of pages for the dictionary part of PDI is the same as the path dictionary. In the following, we develop the cost model for the attribute index part of PDI.

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:

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

where $LO_{A_{i,j}} = min(U_{i,j}, LP_{A_{i,j}})$ and $X < f$. If $X \neq 1$, $NLP_{A_{i,j}}$ is increased by one to account for the root node. 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:

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

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, path dictionary, and path dictionary index. We choose a path of 4 classes in the comparison. For the path index and path dictionary index, a primitive attribute in the bottom class of the path, $A_{4,1}$, 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*, *PDS* and *PDIS* to represent the storage overhead of the path index, path dictionary and path dictionary index, respectively.

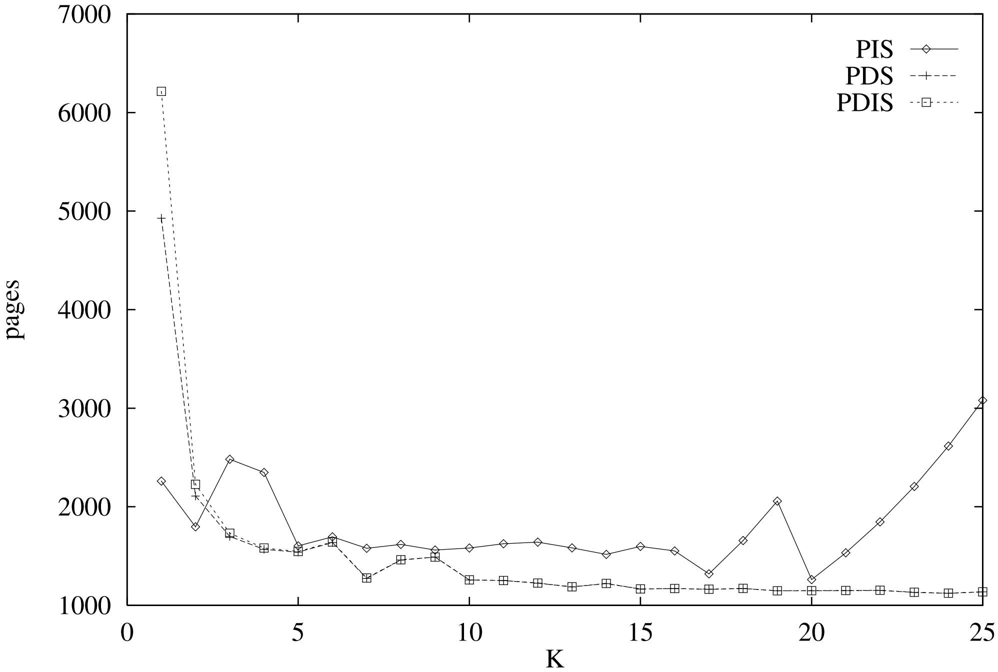

Figure 8: Storage overhead.

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 8 shows that $PIS < PDS < PDIS$ when $K = 1$ and 2, and that $PDS < PDIS < PIS$ 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 methods is similar to that of a leaf node record in the path index, except that the path dictionary methods have additional storage overhead 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 and the path dictionary index have better storage overhead than the path index.

In practice, we usually have more than one attribute in the path to be indexed. In order to compare the overall storage overhead for the indexes created, we calculate the total cost of creating n indexes on attributes of class C_4 for the path index and the path dictionary index. We vary n from 1 to 10 to observe the change of storage overhead for the three methods. In this comparison, we fix the ratios of the shared references among the classes along the path and the ratios of the shared key values for each attribute indexes (i.e., k's and q's are set to 3).

Figure 9 shows that *PIS* increases dramatically, because the path index creates a separate index

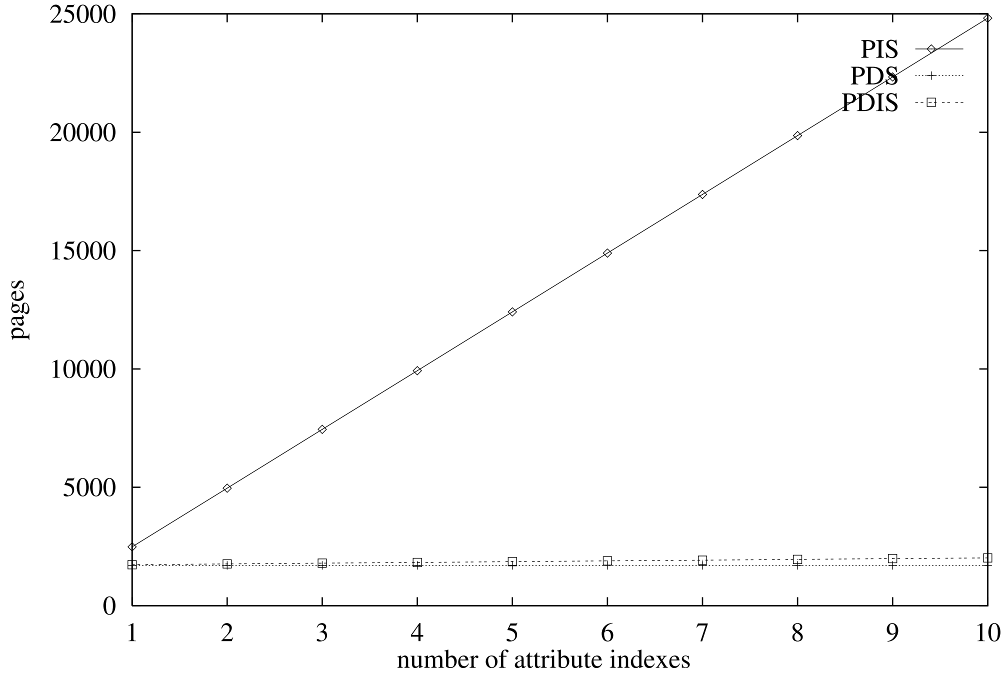

Figure 9: Storage overhead for multiple attribute indexes ($K = 3$).

for each attribute indexed. *PDS*, shown here as a reference, is constant since it doesn't create any index. On the other hand, the increase on *PDIS* is due only to the storage for building the attribute indexes. Thus, we can see that *PDIS* increases linearly with the number of attribute indexes, but at a much lower rate than that of *PIS*.

6.2 Retrieval Cost

Using the classification introduced in Section 1, queries on nested objects can be classified as *TP*, *PT* and *MX*. To simplify our analysis, we assume that there is only one predicate attribute in the queries. Therefore, we will only consider *TP* and *PT* queries in our discussion.

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 [5]. The number of pages accessed for retrieval is:

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

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

Path Dictionary

The path dictionary may be applied to *TP* and *PT* queries. To answer a query Q which has C_t as the target class and C_p as the 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, search the identity index to locate the addresses of the s-expressions, then access the s-expressions in the path dictionary to return the objects in C_t. Therefore, the number of pages accessed is:

$$PDR = \lceil N_p S_p / P \rceil + N_{p|Q}(h_{identity} + 1 + \lceil SS/P \rceil),$$

where $N_{p|Q}$ is the number of objects in class C_p, which satisfy the predicates in Q, and $h_{identity}$ = height of the identity index $-$ 1.

Path Dictionary Index

Likewise, the path dictionary index supports both *TP* and *PT* queries. To answer 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 then access the path dictionary. Therefore, the number of pages accessed is:

$$PDIR = h_{attr.} + \lceil XP_{attr.}/P \rceil + N_{p|Q} \cdot \lceil SS/P \rceil$$

where $h_{attr.}$ = 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*, *PDR* and *PDIR* to represent the retrieval costs of the the path index, the path dictionary 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 indexed 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.

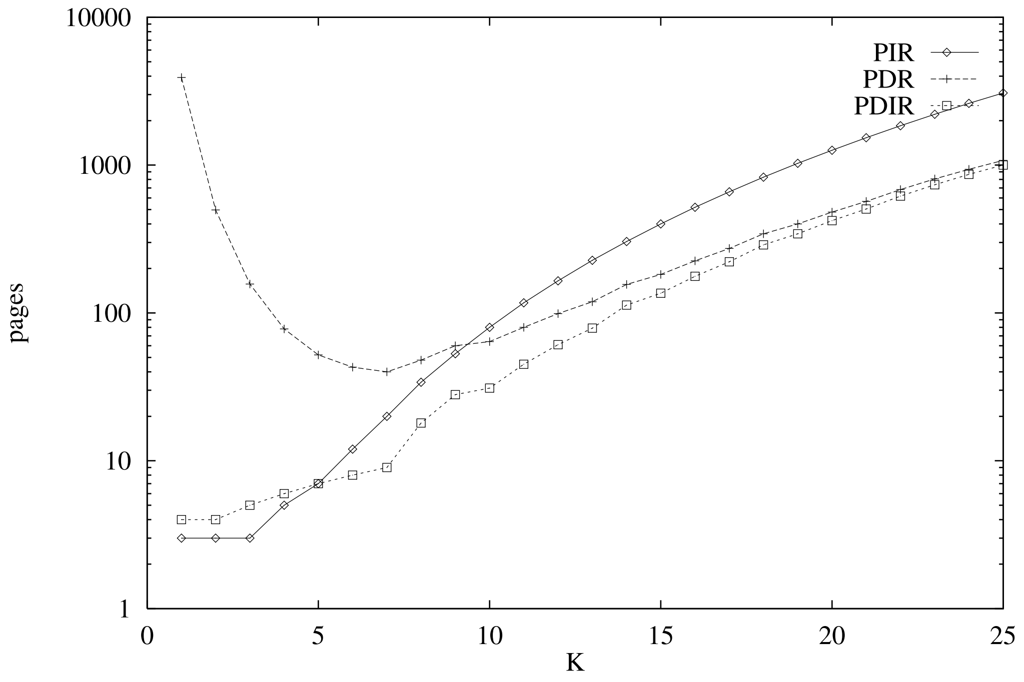

Figure 10: Retrieval cost.

Figure 10 indicates that the path index has the lowest retrieval cost initially, while the path dictionary index is a close second. 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; after $k = 10$, the path dictionary also has a lower retrieval cost than the path index. Therefore, we may conclude that for a query with an indexed attribute in the predicate:

$$\begin{array}{lll} PIR < PDIR < PDR & \text{when} & K < 5 \\ PDIR < PIR < PDR & \text{when} & 5 \leq K < 10 \\ PDIR < PDR < PIR & \text{when} & 10 \leq K \end{array}$$

The path dictionary and the path dictionary index are more general mechanisms than the path index in terms of improving the overall performance for different kinds of queries. The path dictionary mechanisms may be used to process any kind of queries which has predicate attributes in the classes located along the path. The path index, however, can only be used to process queries with predicates on the indexed attributes.

To compare the overall retrieval performance of the three methods, we select the following mix of queries for evaluation:

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.

Note that we only include the TP class of the queries in the list. 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 [19] to compute the retrieval cost.

As before, we vary the average ratio of shared reference and shared key values, K, from 1 to 25 to observe the retrieval performance of the methods with respect to K. Figure 11 shows that *PDR* and *PDIR* have 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 as shown previously.

6.3 Range Query Cost

Range query is one of the important operations for database query. In this section, we develop cost models for the path dictionary and the path dictionary index. Using the formulae, we compare the cost of range query with respect to the percentage of objects satisfying the query predicate.

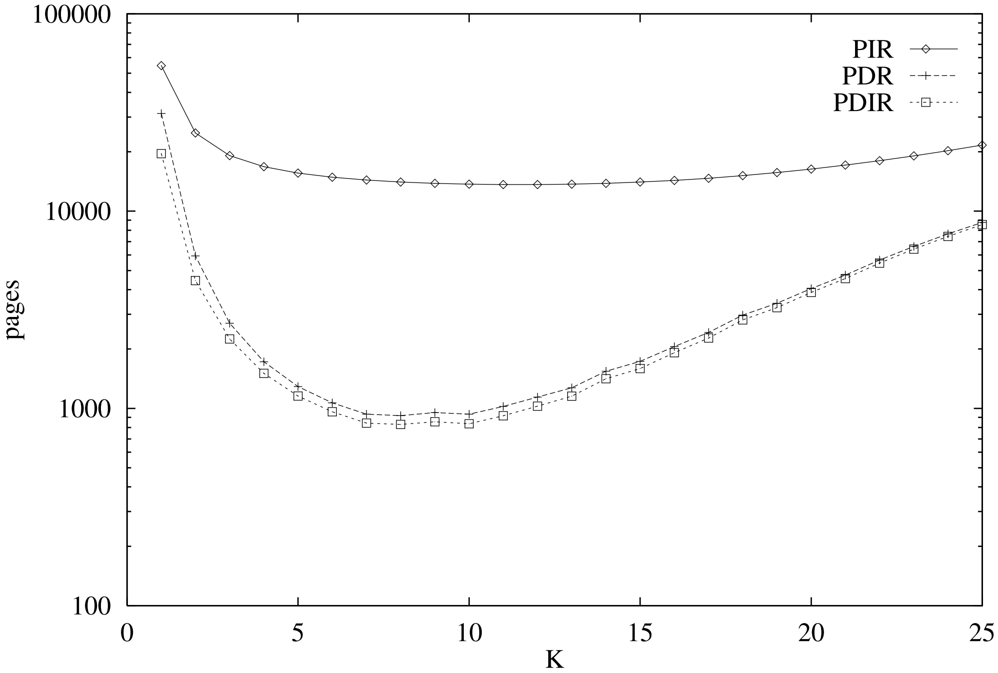

Figure 11: General retrieval cost.

Path Index

Based on [5], the number of page accesses is:

$$PIRQ = \begin{cases} h + \lceil NRQ/NREC \rceil & \text{if} \quad XP \leq P \\ h + \lceil NRQ/\lceil XP/P \rceil \rceil & \text{if} \quad XP > P, \end{cases}$$

where h = height of the path index $-$ 1, XP is the size of a leaf node record for the path index, NRQ is the number of key values specified in the query, and $NREC$ is the number of records in a leaf node page.

Path Dictionary

For range queries on the path dictionary, the formula is the same as the formula for single value retrieval:

$$PDRQ = \lceil N_p S_p / P \rceil + N_{p|Q}(h + 1 + \lceil SS/P \rceil),$$

where $N_{p|Q}$ is the number of objects in class C_p satisfying the predicates in Q and h = height of the identity index $-$ 1.

Path Dictionary Index

To answer a range query with the path dictionary index, an attribute index scan is performed to find the leaf node record corresponding to the lower bound value of the range, then sequentially access to the leaf node pages in the range, and finally retrieve all of the s-expressions specified in these records to return the target objects. Therefore, the number of pages accessed is:

$$PDIRQ = \begin{cases} h + \lceil NRQ/NREC \rceil + N_{p|Q} \lceil SS/P \rceil & \text{if} \quad XP \leq P \\ h + \lceil NRQ/\lceil XP/P \rceil \rceil + N_{p|Q} \lceil SS/P \rceil & \text{if} \quad XP > P, \end{cases}$$

where h = height of the attribute index $-$ 1, XP is the size of a leaf node record for the attribute index, NRQ is the number of key values specified in the query, and $NREC$ is the number of records in a leaf node page.

Comparison

Instead of varying the ratio of references between classes, we change the selectivity of the predicate specified in the query, $RANGE$, from 1% to 100%. In the comparison, we fix k_1, k_2, k_3 and $q_{4,1}$ to 3. The number of key values satisfying the predicate, NRQ, is $U_{4,1} \cdot RANGE$. Therefore, the number of objects in C_4 satisfying the query predicate, $N_{p|Q}$, is $NRQ \cdot q_{4,1}$. $N_{p|Q}$ is used in computing the cost of range queries for the path dictionary and the path dictionary index.

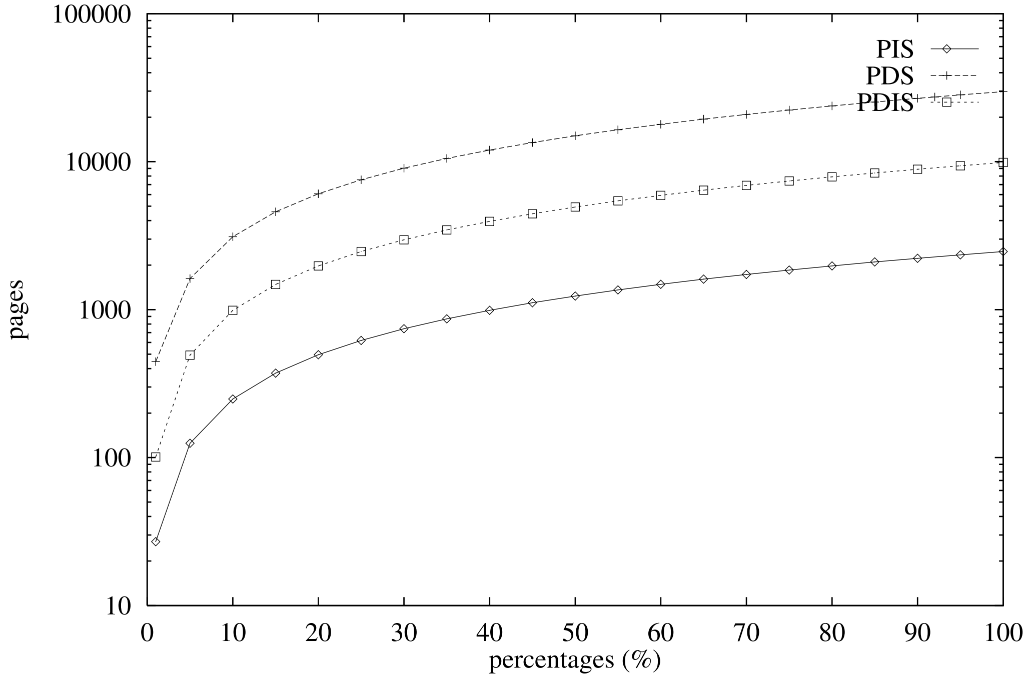

Figure 12: Range Retrieval Cost($k_1 = k_2 = k_3 = q = 3$).

Figure 12 shows that the path index has a much better performance on answering range queries than the other two approaches. This is because the leaf node records of the path index are sorted based on the key values of the indexed attribute. After accessing to the record corresponding to the lowest key value, the OIDs of the target objects may be returned by sequentially scanning the leaf nodes until the highest range value is reached. On the other hand, the path dictionary index has to access to s-expressions in the path dictionary to return the OIDs of the target objects after they obtain the addresses of s-expressions. The path dictionary approach has the worst performance, because the evaluation of predicates is based on accessing the objects in the predicate class.

6.4 Update Cost

Due to space constraints, we only present the cost model for update operation. Other update operations, such as insertion and deletion, may be derived in a similar way. To simplify the analysis, we do not include the cost 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 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 traversals to $A_{n,1}$:

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

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 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:

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

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

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

Path Dictionary

To update the complex attribute A_{i+1} of O_i from θ_{i+1} to θ'_{i+1}, the identity index is searched to read and write back the s-expressions corresponding to θ_{i+1} and θ'_{i+1}. To simplify our analysis, we assume that θ_{i+1} and θ'_{i+1} are in different s-expressions and that they are in different pages. Therefore, the number of page accesses for update is:

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

where $h_{identity}$ = height of the identity index $-$ 1.

Path Dictionary Index

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

1. An indexed simple attribute 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:

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

   where $h_{attr.}$ = height of the attribute index $-$ 1.

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 the same as that of the path dictionary.

3. If one of the attributes in C_i or C_i's ancestor classes is indexed, the attribute index has to be updated too. Therefore, the number of page accesses for update with PDI is:

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

   where $XP_{attr.}$ is the size of a leaf node record in the attribute index.

Comparison

In the first case, both the path index and the path dictionary index are required to update their indexes, while the path dictionary mechanism is not required to do so. For the second case, all of the three mechanisms need an update, while the path dictionary and path dictionary index have the same update cost. For the third case, it's only fair to compare the path dictionary and path dictionary index, because the indexed attribute in the path index must be at the leaf class of the path; this is why the path index doesn't support PT queries and it doesn't need an update in this situation. In our comparison, we choose the formula for case 3 to compute the update cost of the path dictionary index, *PDIU*. Since *PDU* and *PDIU* are the same for the second case, *PIU* and *PDU* are used to compare the update costs between the path index and the other two mechanisms. The difference between *PDIU* and *PDU* is the update overhead on the attribute index required for the path dictionary index method.

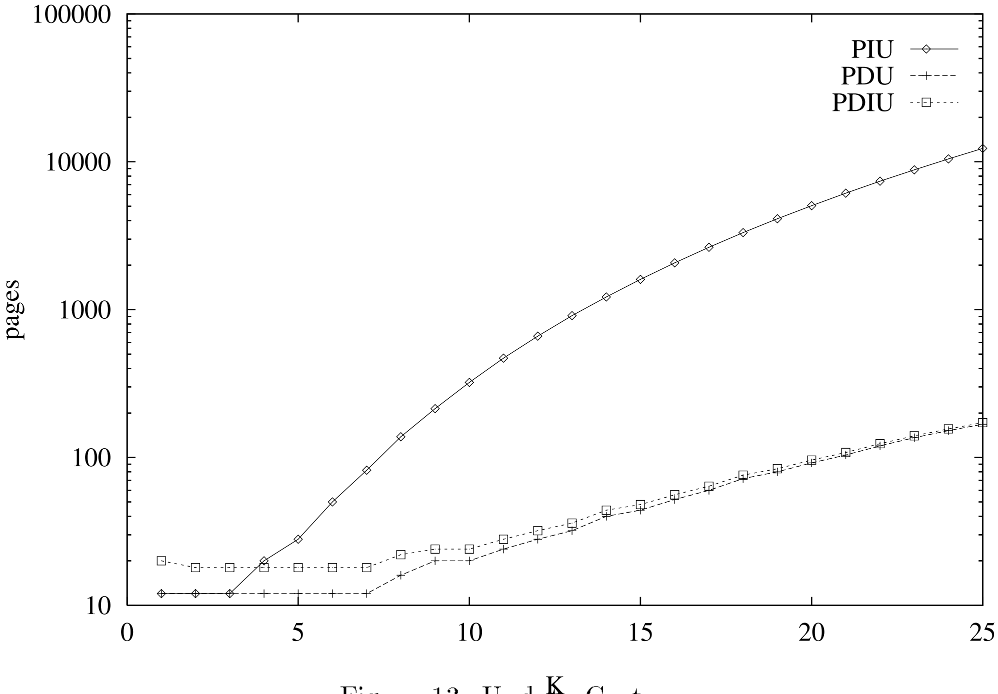

Figure 13: Update Cost.

One of the most important reasons for extending the path dictionary with the identity index is to improve its update performance. The improvement is shown in Figure 13, which depicts the update cost of the three methods. Initially, the path index has the same update cost as that of the path dictionary and the path dictionary index. However, as K increases, the update cost of the path index dramatically increases, while the update cost for the two path dictionary mechanisms remains relatively low. The difference between *PDU* and *PDIU* decreases as K increases, because, owing to assumptions 1 and 3, there are fewer key values in the indexed attribute and fewer objects in C_4 when K is large. Therefore, the size of the attribute index is smaller with large values of K, resulting in smaller update overhead on the attribute index.

7 Conclusion

Aggregate relationship is one of the most frequently found relationships in database applications. In object-oriented database systems, the aggregate relationship is implemented by allowing an object to be stored as a complex attribute of another object. This nested structure among objects suggests the need of nested queries. In this paper, we classify and give examples of nested queries in OODBS. Then, we introduce the path dictionary and path dictionary index methods for nested query evaluation. We also develop cost models for the storage overhead and retrieval and update costs, and compare the costs to the path index method. For most of the comparisons, we vary the average ratio of shared references and shared key values to observe its impact on the performance and overhead of the three mechanisms.

When there is one indexed attribute, the storage overhead for the path dictionary method is better than that of the path dictionary index method, which in turn is better than that of the path index method. When there is more than one indexed attribute, the storage overhead for the path index method is increasingly larger than that of the two path dictionary mechanisms. The storage requirement for the path dictionary is constant, because it is general enough to support all classes of nested queries. For the path dictionary index, the extra storage needed to create the attribute indexes is low comparing to the path index method. Thus, it is affordable to have many attribute indexes on the path dictionary.

Generally speaking, the path dictionary index method has the best retrieval performance. The path index method is better than the other two organizations 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 and path dictionary index is overwhelmingly better than that of the path index. Also, the path index cannot be used to support queries in which the predicate class is an ancestor of the target class.

For the update operation, the path dictionary has the best performance and the path dictionary index is a close second. Under all conditions, both of the path dictionary and path dictionary index have a better performance than the path index.

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. Using the cost models for detailed analysis of available query processing strategies presents a future research topic on query optimization.

References

[1] B. Jenq et al., "Query Processing in Distributed Orion," *Proceedings of International Conference on Extending Database Technology*, Venice, Italy, March 1990, 169–187.

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

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

[4] E. Bertino & P. Foscoli, "An Analytical Model of Object-Oriented Query Costs," *Proceedings of the Fifth International Workshop on Persistent Object Systems*, 1993, 241–261.

[5] 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.

[6] S. Choenni, E. Bertino, H.M. Blanken & T. Chang, "On the Selection of Optimal Index Configuration in OO Databases," *Proceedings of the Tenth International Conference on Data Engineering*, Houston, TX, 1994, 526–537.

[7] O. Deux et al., "The Story of O_2," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, 91–108.

[8] Illustra, "Illustra User Manual," CA, 1994.

[9] 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.

[10] A. Kemper & G. Moerkotte, "Advanced query processing in object bases using access support relations," *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, Aug. 1990, 290–301.

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

[12] K.C. Kim, W. Kim & A. Dale, "Cyclic Query Processing in Object-Oriented DBMS," *Proceedings of the IEEE International Conference on Data Engineering*, 1989, 564–571.

[13] K.C. Kim, W. Kim, D. Woelk & A. Dale, "Acyclic Query Processing in Object-Oriented DBMS," *Proceedings of the Entity-Relationship Conference*, Italy, Nov. 1988.

[14] W. Kim, "UniSQL/X unified relational and object-oriented database system," *June 1994*, Vol. 23, No. 2, 481.

[15] 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.

[16] 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.

[17] W. Kim et al., "Integrating an object-oriented programming system with a database system," *Proceedings of OOPSLA*, 1988, 142–152.

[18] C. Lamb, G. Landis, J. Orenstein & D. Weinreb, "The ObjectStore database system," *Communications of ACM*, Vol. 34, No. 10, Oct. 1991, 50–63.

[19] 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*, Gathersberg, MD, , Nov. 1994, 64–71.

[20] W.-C. Lee & D.L. Lee, "Combining indexing technique with path dictionary for nested object queries," *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA '95)*, Singapore, Apr. 1995, 107–114.

[21] 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.

[22] W.-C. Lee & D.L. Lee, "Short Cuts for Traversals in Object-Oriented Database Systems," *Proceedings of the International Computer Symposium*, Hsinchu, Taiwan, Dec. 1994, 1172–1177.

[23] C.C. Low, B.C. Ooi & H. Lu, "H-trees: a dynamic associative search index for OODB," *Proceedings of the 1992 SIGMOD Conference*, San Diego, CA, June 1992, 134–143.

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

[25] D. Maier & J. Stein, "Development and implementation of an object-oriented DBMS," in *Readings in Object-Oriented Database Systems*, S. Zdonik & D. Maier, eds., Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990, 167–185.

[26] E.J. Shekita & M.J. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS," *Proceedings of the 1989 SIGMOD Conference*, Eugene, Oregon, May 1989, 325–336.

[27] P. Valduriez, "Join indices," *ACM Transactions on Database Systems*, Vol. 12, No. 2, June 1987, 218–246.

[28] H.-S. Yong, S. Lee & H.-J. Kim, "Applying signatures for forward traversal query processing in object-oriented databases," *Proceedings of the 1993 SIGMOD Conference*, Washington, DC, 1993, 518–525.

Contents

List of Figures