Signature Path Dictionary for Nested Object Query Processing

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

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

Abstract

Predicate evaluation and object traversal are two critical issues for nested object query processing. Aiming at these two issues, we introduce a new method, the signature path dictionary, which combines signature techniques with the path dictionary organization designed for fast object traversals. We derive cost formulae for its storage overhead as well as the retrieval and update costs. Comparing to a previously proposed indexing organization, path signature, the signature path dictionary is superior in all aspects.

1 Introduction

Object-Oriented Database Systems (OODBSs) have advanced significantly in the last decade. In addition to facilitating the design and engineering of traditional database applications, OODBSs provide data modeling mechanisms to support new database applications such as CAD/CAM, CASE, office automation, multimedia systems, and geographical information systems. Among the rich modeling facilities, the concept of *class* allows object-oriented database systems to define 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.

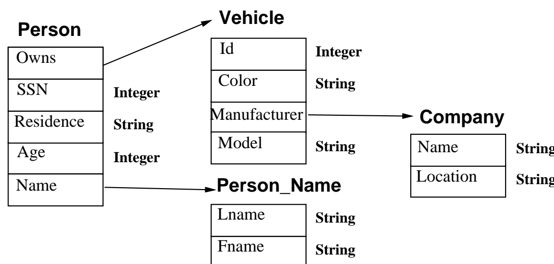


Figure 1: Aggregation hierarchy.

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.

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

1.1 Nested Object Query Processing

OODBSs support queries involving nested objects. These queries are called *nested queries*. There are two basic approaches to evaluating a nested query: *top-down* and *bottom-up* evaluations [5]. 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. Note that when every reference from an object O to another object O' (e.g., `Owns`) is accompanied with an inverse reference from O' to O (e.g., `Owned_by`), the aggregation hierarchy becomes bidirectional, resulting in no difference between the top-down and the bottom-up approaches. In this paper, however, we assume there is no inverse references.

There are many kinds of nested queries, and several indexing organizations have been proposed to support nested query processing [1,2,6,7,9]. However, an access method doesn't necessarily support all of the nested queries. 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. Thus, nested queries may be classified by the 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.

1.2 Signature Techniques

The object signature S_i of an object O is a superimposed bit string generated from O's attribute values. The signature of a simple attribute is obtained by hashing the attribute value. As we will see later, the signature of a complex attribute can be obtained in a number of ways. An object signature is formed by superimposing the signatures of its attributes. Object signatures are stored sequentially in a signature file. A query specifying certain values to be searched for is transformed into a query signature S_Q in a similar way. The query signature is then compared to every object signature in the signature file. There are three possible outcomes of the comparison: (1) the object matches the query; that is, for every bit set in S_Q the corresponding bit in the object signature is also set (i.e. $S_Q \wedge S_i = S_Q$); (2) the object doesn't match the query (i.e. $S_Q \wedge S_i \neq S_Q$); and (3) the signature comparison indicates a match but the object in fact does not match the search criteria. The last case is called a *false drop*. In order to eliminate false drops, the object must be examined after the object signature signifies a match.

The signature file is used to screen out most of the unqualified objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object accesses are avoided. Signature files have a much lower storage overhead and a simpler file structure than inverted indexes. They are particularly good for multi-attribute retrieval when the attributes have equal chance of being specified in the query. Since queries in an OODBS could be very flexible, inverted indexes would be very complex if all possible access paths are supported.

Several signature techniques for OODBSs have been proposed. The tree signature scheme creates the signature of an object by superimposing the signatures of all of the simple and nested attributes of the object in the aggregation hierarchy, whereas the path signature method generates an object signature based on the simple and nested attributes along a given path [7]. The signature replication technique [9] generates object signatures from the simple 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. Moreover, a signature technique for the support of set value access has been proposed [4]. In this paper, we propose a different signature organization which provides both associative search and object traversal facilities for query processing in OODBS.

The rest of the paper is organized as follows. In Section 2, we first briefly describe the path signature scheme, which is compared with the new organization in performance evaluation. In Section 3, we introduce the concept and implementation of the signature path dictionary. Cost models for both organizations and performance comparisons are presented in Section 4. Section 5 concludes the paper.

2 Path Signature Scheme

For every class C in a given path of the aggregation hierarchy, there exists a signature file such that every object O in C has a signature entry $\langle sig, oid_list \rangle$ in the signature file, where *sig* is the signature of O and *oid_list* is a list of object identifiers corresponding to O and its nested objects located on the path.

The signature of an object is created as follows:

1. the signature of a primitive object is generated by hashing the primitive value.
2. the signature of an object not on the path is generated by hashing on its object identifier.
3. the signature of an object on the specified path is generated from superimposing the signatures of all of its attributes.

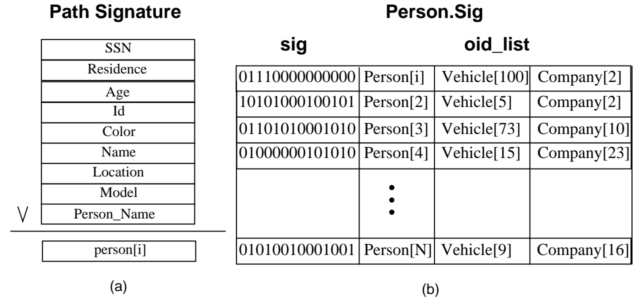


Figure 2: Path Signature Scheme. (a) Creation of an object signature; (b) structure of a path signature file.

Figure 2(a) and (b) show the creation of an object signature and the signature file for class `Person` located on the path `Person.Vehicle.Company`. The signature of `Person`[i] is the superimposition of the signatures for `SSN`, `Residence`, `Age`, `Name`, and `Owns`. The signature of `Name` is generated by hashing its OID (`Lname` and `Fname` are not used). The signature of `Owns` is generated from all of the primitive values along the path (i.e., `Id`, `Color`, `Model`, `Name`, and `Location`). In the signature file of class `Person`, the signature of object `Person`[i] is associated with a list of OIDs, including `Person`[i] and its nested objects. The path signature files are indexed by an identity index which, given an OID, allows the query processor to efficiently locate the signature entries containing the OID. The identity index may be implemented as a tree-structured organization such as B-tree. Because an OID may appear in several signature files and in different signature entries, we use the $\langle File\#, Page\# \rangle$ to specify the locations of the signature entries.

There are two types of nested queries: TP and PT queries. To retrieve objects for a TP query Q, we may evaluate the query by top-down or bottom-up approaches. For the top-down approach, the query signature S_Q has to be formed first. For each entry $\langle sig_i, oidlist_i \rangle$ in the signature file associated with the target class, S_Q is compared with S_i. If they match, $oidlist_i$ is then put into a set. Finally, for each $oidlist_i$ in the set, objects are retrieved from the predicate class to eliminate false-drop. If the search condition of the query is satisfied, the OIDs of the target objects are returned. For the bottom-up approach, the query signature S_Q is compared with the signatures in the signature file of the predicate class. Matched objects from the predicate class are accessed to eliminate false drops. The OIDs of the truly qualified objects are used as key to search the identity index for addresses of the signature entries in the target class's signature file. The OIDs of the qualified target objects are returned from the signature entries of the target signature file.

On the other hand, PT queries may be processed using top-down approach only, because the predicate class is on top of the target class. Therefore, the signatures associated with the target class do not maintain information about objects in the predicate class.

When an object O is modified, we have to update the signatures of O and its ancestor objects to maintain the accuracy of the signatures. The maintenance of signatures is

divided into two tasks: 1) compute the signature of an object; 2) spread the update of the signature from O to its ancestor objects. The signatures of simple and complex attributes not on the path are generated by hashing. The signature for O's nested objects on the path can be found from the signature file associated with the nested class. Therefore, the signature of the nested objects may be obtained by scanning the identity index and access to the signature file. The new OID list for the modified object may be generated by attaching the OID list of the nested object to the modified object. The new signature and OID list are used to update the signature entry corresponding to O in the signature file. The update has to be applied on all ancestor objects of O.

3 Signature Path Dictionary

Path dictionary is a secondary organization proposed by the authors to support efficient object traversal for nested query processing [6]. In this paper, we combine the associative search capability of the signature files with the path dictionary to introduce a new index organization for nested object queries.

3.1 Path Dictionary

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. Path dictionary is a secondary organization which 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 the actual 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.

3.2 Data Structure and Implementation

In the signature path dictionary, an object signature is generated by superimposing only the bit strings generated from the simple attributes of the object. Instead of associating the signature files with classes, the signatures are associated with the OIDs of their corresponding objects in the path dictionary.

The signature path dictionary consists of a series of recursive expressions which maintain abstracted attribute and path information for objects in the database. Since the linking structure denoted by the expression resembles an inward subtree, we call the expression an *s-expression.* An s-expression records the information about all paths terminating at an object in a leaf class.

The s-expression for the path $C_1.C_2....C_n$ is defined as follows:

$S_1 = \Theta_1$, where Θ_1 is null or a pair $\langle O_1, SIG_1\rangle$; O_1 is the OID of an object in class C_1 and SIG_1 is its signature.

$S_i = \Theta_i(S_{i-1}[, S_{i-1}])$ $1 < i \leq n$, where Θ_i is null or a pair $\langle O_i, SIG_i\rangle$, in which O_i is the OID of an object in class C_i and SIG_i is its signature.

S_i is an s-expression of i levels, in which the list associated with Θ_i contains all of the OIDs and signatures 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.

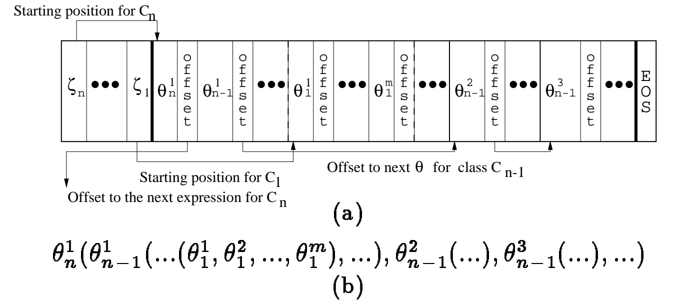
The signature path dictionary for $C_1.C_2...C_n$ consists of a sequence of n-level s-expressions. The leading object in an s-expression, which does not necessarily belong to C_n, is the terminal object of the paths denoted by the s-expression. The number of s-expressions in the path dictionary is equal to the number of objects, which do not have a nested object on the path. Figure 3 shows examples of the s-expressions in a signature path dictionary. SIG_C, SIG_V, and SIG_P are, respectively, the signatures for objects in `Company`, `Vehicle`, and `Person` classes. It maintains all the abstracted attribute and path information for the objects located on the path `Person.Vehicle.Company`.

Path = Person.Vehicle.Company

```
Company[1]SIGC[1](Vehicle[5]SIGV[5](Person[3]SIGP[3],Person[7]SIGP[7]),
                  Vehicle[12]SIGV[12](Person[4]SIGP[4]))
Company[2]SIGC[2](Vehicle[6]SIGV[6](),Vehicle[9]SIGV[9](),
                  Vehicle[11]SIGV[11]())
((Person[9]SIGP[9]))
```

Figure 3: Examples of the s-expressions.

Since an object signature is abstracted from the simple attributes of the object, it will only support the evaluation of predicates involving its resident class. However, the path information is available from the path dictionary, the signatures of an object's nested and ancestor objects can be easily obtained.



$$\theta_n^1(\theta_{n-1}^1(...(\theta_1^1, \theta_1^2, ..., \theta_1^m), ...), \theta_{n-1}^2(...), \theta_{n-1}^3(...), ...)$$
(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 Fig. 4(b) is the corresponding s-expression. ζ_i in the header of an s-expression points to the first occurrence of θ_i^j, which consists of OID_i^j and SIG_i^j. OID_i^j is the OID of the jth C_i object in the s-expression and SIG_i^j is its signature. Every θ in the signature path dictionary has an associated offset; the offset associated with θ_i^j points to θ_i^{j+1}. To retrieve the OID and the signature of every object in C_i, we start with ζ_i, which will lead us to the first θ_i in the s-expression, and following the associated offset value we can reach the next θ_i, and so on. Thus, we can quickly scan through all OIDs and the associated signatures of the objects in a class, skipping the OIDs and signatures of objects in irrelevant classes. The offset associated with the object of class C_n, however, is pointing to the θ_n 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. Consequently, free space may be left 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.

In order to efficiently locate an s-expression, 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. The identity index is organized as a separate search tree on top of the path dictionary. The leaf nodes of the index store the identifiers and s-expression locations of the objects.

3.3 Retrieval and Update Operations

Assume that a signature path dictionary is available for the path $C_1.C_2...C_n$. For a given query Q with predicate classes $C_{p_1}, C_{p_2}, ..., C_{p_m}$, where $m \leq n$ and $1 \leq p_1 < p_2 < ... < p_m \leq n$, a predicate signature Sig_{p_i} is generated for each predicate class C_{p_i}, where $1 \leq i \leq m$. To answer the query, the signature path dictionary is sequentially scanned to match the object signatures with the corresponding predicate signatures. The matched objects are retrieved from the database to eliminate false drops. Finally, the OIDs of the qualified objects in the target class are derived from the s-expressions and returned to the users.

To answer the query "retrieve the persons who own cars made by GM", the signature path dictionary is scanned and the signatures of `Company` objects are used to match with the signature generated for "GM". The matched `Company` objects are then accessed to verify if they are really called "GM". The `Person` objects are returned if they are in the s-expressions of the `Company` objects which satisfies the predicates.

We have to consider the updates on simple attributes and complex attributes of an object separately. When a simple attribute of an object is updated, we first recompute the signature of the modified object; search through the identity index to locate the s-expression of the modified object; and update the signature of the modified object. The identity index need not be changed.

To update the complex attribute of an object, we recompute the signature of the object. The path dictionary and identity index have to be updated accordingly to maintain accurate path information. Suppose O_i has its complex attribute changed from O_{i+1} to O'_{i+1} (O_i, O_{i+1} and O'_{i+1} are identified by ϕ_i, ϕ_{i+1} and ϕ'_{i+1}, respectively). We search the path dictionary through the identity index to find the s-expressions containing ϕ_i and ϕ'_{i+1}. Then, ϕ_i and its ancestor list are moved from the ancestor list of ϕ_{i+1} to the ancestor list of ϕ'_{i+1}. Meanwhile, the identity index has to be updated by changing the s-expression addresses for objects in the moved ancestor list.

4 Analytical Model

We formulate cost models to estimate the storage overhead and query performance of the signature path dictionary. Given a path $P = C_1C_2...C_n$, we use the following parameters to describe the characteristics of classes in the path.

- A_i: the complex attribute of C_i used to connect C_i to C_{i+1}, $1 \leq i \leq n-1$.
- D_i: the number of distinct values for attribute A_i.
- N_i: the number of objects in class $C_i, 1 \leq i \leq n$.
- P: the page size.
- Pf: false drop probability of a signature.
- Ps: selectivity of a query.
- Sr_i: the ratio of shared references from C_i to C_{i+1}.
- f: fanout from a nonleaf node in the identity index.
- ff: the length of a file pointer.
- k: average number of attributes in a class.
- m: length of a signature in bits.
- pp: the length of a page pointer.
- s: the number of bit strings which are superimposed into a signature.
- w: number of 1's in a a bit string generated from hashing.
- EL: the length of EOS.
- FSL: the length of the free space field in FSD.
- $OFFL$: the length of an offset field in the signature path dictionary.
- $OIDL$: the length of an object identifier.
- $OBJL$: the average size of an object.
- SL: the length of the start field in the signature path dictionary.

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*.

The following are assumptions used in our analysis:

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.

Table 1: Parameters of the cost models.

$EL = 4$	$OIDL = 8$	$OBJL = 320$	$SL = 2$	$pp = 4$
$FSL = 2$	$OFFL = 2$	$P = 4096$	$ff = 1$	$k = 10$

Table 1 lists the values chosen for the performance comparison. Based on the assumption, the number of objects in class C_i may be derived as follows.

$$N_i = Sr_i Sr_{i+1} ... Sr_{n-1} \cdot N_n, \text{ where } 1 \leq i \leq n.$$

4.1 False Drop Probability

For a given signature length of m bits and the number of bit strings superimposed into a signature s, the minimal false drop probability for a single value query is [3,8]:

$$Pf \approx 0.5^{m \cdot ln2/s}$$

4.1.1 Path Signature

Assume that the average number of attributes in a class is k. The optimal false drop probability for signatures corresponding to objects in class C_i is:

$$Pf_i^{ps} = 0.5^{m \cdot ln2/((n-i+1) \cdot k)}$$

Assume that a query involves only the target class and a predicate class and that the classes on the path have equal probability to be used in a query. The average of the optimal false drop probabilities for signatures corresponding to objects in classes $C_1, C_2, ..., C_n$ is:

$$Pf_{avg}^{ps} = \frac{2}{n(n+1)} \sum_{i=1}^{n} Pf_i^{ps}(n-i+1)$$

4.1.2 Signature Path Dictionary

The optimal false drop probability for signature path dictionary is:

$$Pf^{spd} = 0.5^{m \cdot ln2/k}$$

4.1.3 Comparison

In the following comparisons, we assume a path of 4 classes. The number of objects in C_4 is assumed to be 10,000. The average reference ratio between consecutive classes is set to 3. Also, the average number of attributes in a class is set to 10. Using the formulae developed above, we compare the false drop probability of the signature path dictionary and the average false drop probability of the path signature files. In the comparison, we vary the signature lengths to observe the change of the false drop probabilities.

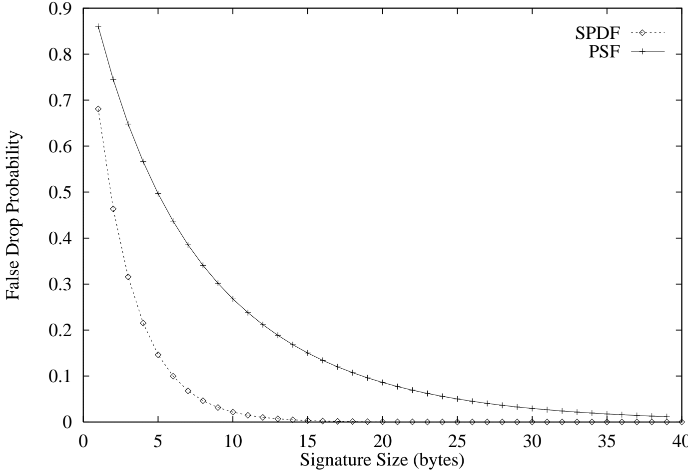


Figure 5: False drop probability vs signature size.

Figure 5 shows, as we expected, that the signature path dictionary has lower false drop probability than the path signature method. This is because the signature path dictionary only abstracts information from an object's simple attributes object into the object signature, while the path signature maintains the abstracted information from both simple and nested attributes.

4.2 Storage Overhead

4.2.1 Path Signature

For every object in C_i, there is an entry in the signature file associated with C_i. The size of the entry is:

$$SE_i = SIGL + (n - i + 1) \cdot OIDL$$

There are N_i objects in C_i. Thus the number of disk pages for C_i's signature file is:

$$SSF_i = \begin{cases} \lceil N_i / \lfloor P/SE_i \rfloor \rceil, & \text{if} \quad SE_i \leq P \\ N_i \lceil SE_i / P \rceil, & \text{if} \quad SE_i > P \end{cases}$$

For the identity index, the size of a leaf node corresponding to an object in C_i is:

$$XP_i = OIDL + (ff + pp) \cdot \sum_{j=1}^{i} Sr_j \cdot Sr_{j+1} \cdots Sr_i$$

The number of leaf pages of the identity index corresponding to objects in class C_i is:

$$LP_i = \begin{cases} \lceil N_i / \lfloor P/XP_i \rfloor \rceil, & \text{if} \quad XP_i \leq P \\ N_i \lceil XP_i / P \rceil, & \text{if} \quad XP_i > P \end{cases}$$

The total number of leaf pages needed for the identity index of the path signature files is:

$$TLP_{ps} = \sum_{i=1}^{n} LP_i$$

The number of nonleaf pages is:

$$NLP_{ps} = \lceil TLP_{ps}/f \rceil + \lceil \lceil TLP_{ps}/f \rceil / f \rceil + \ldots + X,$$

where $X < f$. If $X \neq 1$, NLP_{ps} is increased by 1 to account for the root node. As a result, the overall storage overhead for the path signature files for $C_1.C_2...C_n$ is:

$$PSS_{ps} = \sum_{i=1}^{n} SSF_i + TLP_{ps} + NLP_{ps}$$

4.2.2 Signature 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 $(OIDL + SIGL + OFFL)$ bytes in an s-expression. The average number of objects in an s-expression is:

$$NOBJ = 1 + Sr_{n-1} + Sr_{n-1} Sr_{n-2} + \ldots + Sr_{n-1} Sr_{n-2} \ldots Sr_1$$

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

$$SS = SL \cdot n + (OIDL + SIGL + OFFL) NOBJ + EL$$

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

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

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

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

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

$$TLP_{spd} = \lceil NOBJ \cdot N_n / \lfloor P/(OIDL + pp) \rfloor \rceil$$

The number of nonleaf pages is:

$$NLP_{spd} = \lceil TLP_{spd}/f \rceil + \lceil \lceil TLP_{spd}/f \rceil / f \rceil + \ldots + X,$$

where $X < f$. If $X \neq 1$, NLP_{spd} is increased by 1 to account for the root node. Therefore, the number of pages needed for the path dictionary is:

$$SPDS = FSD + SSP + TLP_{spd} + NLP_{spd}$$

4.2.3 Comparison

We use the same parameters and assumptions as we used in evaluating the false drop probability. We evaluate the storage overhead with signature size from 1 to 40 bytes for the path signature and signature path dictionary organizations.

Figure 6 shows that the signature path dictionary has a smaller storage overhead than the path signature for various signature sizes. As we will see later in the comparison for retrieval cost, the signature path dictionary and the path signature reach the best retrieval performance when the signature size is 15 bytes and 51 bytes, respectively. In other words, the signature path dictionary introduces a much smaller storage overhead than the path signature does to achieve the best retrieval performance.

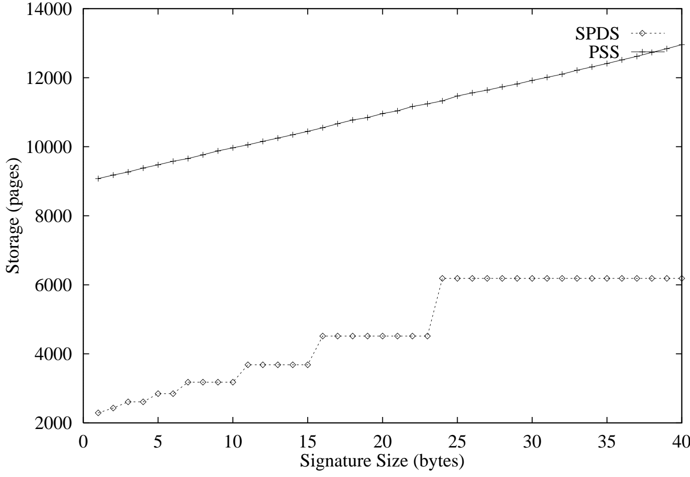


Figure 6: Storage vs signature size.

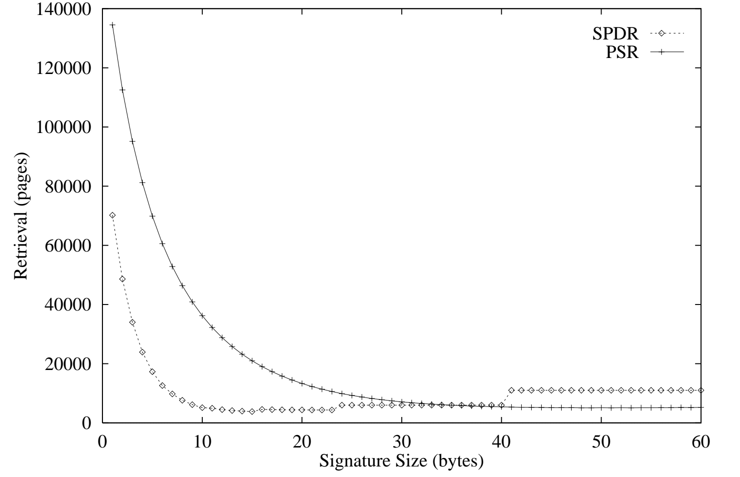


Figure 7: Retrieval cost vs signature size.

4.3 Retrieval Cost

To simplify our analysis, we assume that there is only one predicate attribute in the queries. Let C_t and C_p be the target and predicate classes of the queries, respectively.

4.3.1 Path Signature

Due to space constraint, we only derive formulae for the top-down approach. To evaluate a query which uses classes C_i and C_j, where $1 \le C_i \le C_j \le n$, as predicate and target classes, respectively, we first scan through the path dictionary associated with C_i. Let Pf_i^{ps} and Ps^{ps} be the false drop probability of the signatures and the selectivity of the query, respectively. Assuming equal probability for each class to be used in a query, the average number of page accesses for a query is:

$$PSR = \frac{\sum_{i=1}^{n}\sum_{j=i}^{n}(SSF_i + (Pf_i^{ps} + Ps^{ps}) \cdot N_i \cdot \lceil OBJL/P \rceil)}{n(n+1)/2}$$

4.3.2 Signature Path Dictionary

The signature path dictionary must be scanned to answer a query Q with C_t as the target class and C_p as the predicate class, where $1 < t, p < n$. Let Pf^{spd} and Ps^{spd} be the false drop probability of the signatures and the selectivity of the query, respectively. The average number of pages accessed is:

$$SPDR = SSP + (Pf^{spd} + Ps^{spd}) \cdot \frac{\sum_{i=1}^{n} N_i}{n} \cdot \lceil OBJL/P \rceil$$

4.3.3 Comparison

Figure 7 compares the average retrieval cost of the signature path dictionary and path signature. It shows that the retrieval cost of the signature path dictionary rapidly drops and then increases again as the signature size increases. The path signature shows similar behavior, though it is not obvious in the figure. The figure also shows that the best retrieval performance for the signature path dictionary is better than that of the path signature.

4.4 Update Cost

To simplify the analysis, we do not include the cost due to page overflow. In the following, we assume that the complex attribute A_{i+1} of O_i is changed from ϕ_{i+1} to ϕ'_{i+1} (ϕ_{i+1} and ϕ'_{i+1} are OIDs of O_{i+1} and O'_{i+1}, respectively).

4.4.1 Path Signature

We first derive the update cost when the complex attribute of O is updated. The number of page accesses for searching the identity index based on an object in class C_i is:

$$SII_i = h + \lceil XP_i/P \rceil,$$

where $1 \le i \le n$ and $h =$ height of the identity index -1.

We need to search the identity index twice to access the leaf nodes corresponding to O_{i+1} and O'_{i+1}. The number of page accesses is $2 \times SII_{i+1}$. Meanwhile, the number of page accesses for searching the identity index and accessing the leaf node corresponding to O is SII_i.

The $\langle File\#, Page\# \rangle$ pairs on the leaf node corresponding to O_i has to be moved from the leaf node corresponding to O_{i+1} to that corresponding to O'_{i+1}. Thus, the number of page accesses is:

$$WLN_i = 2 \cdot \lceil XP_{i+1}/P \rceil,$$

where $1 \le i \le n-1$.

Based on the leaf nodes corresponding to O_i and O'_{i+1}, we access to the signature entries for O_i and O'_{i+1}, recompute the signature and reconstruct the entry for O_i, and write the entry back to its signature file. Therefore, the number of page accesses needed is:

$$WB_i = \lceil SE_{i+1}/P \rceil + 2 \cdot \lceil SE_i/P \rceil,$$

where $1 \le i \le n-1$.

When the updated object is in class C_1, it's not necessary for update spreading. For other classes, the number of page accesses for spreading the update of signatures to the object's ancestor objects is:

$$\begin{aligned} SPA_i &= (2\lceil SE_{i-1}/P \rceil + \lceil OBJL/P \rceil) \cdot (Sr_{i-1} + Sr_{i-2} \cdot Sr_{i-1} \\ &\quad + \cdots + Sr_1 \cdot Sr_2 \cdots Sr_{i-1}), \end{aligned}$$

where $2 \le i \le n$.

Thus, to update a complex attribute of the path signature organization, the total number of page accesses is:

$$PSU_{complex} = \frac{\sum_{i=1}^{n-1} 2SII_{i+1} + SII_i + WLN_i + WB_i + SPA_i}{n-1}$$

On the other hand, if a simple attribute of an object O_i in class C_i is modified, we have to update the signatures of O_i

and its ancestor objects. The identity index is not changed. The update cost includes scanning the identity index for the objects nested in O to obtain its signature and the addresses of the signature entries corresponding to O and O's ancestors, access to them, and update their corresponding signature entries. Therefore, the total number of page accesses is:

$$PSU_{simple} = \frac{\sum_{i=1}^{n-1}(SII_{i+1} + WB_i + SPA_i) + SII_n + WB_n + SPA_n}{n}$$

4.4.2 Signature Path Dictionary

If a simple attribute of an object O is modified, we have to update O's signature. This entails an identity index scan and write-back of an s-expression. The number of page accesses for scanning the identity index is:

$$SII = h + \lceil LP/P \rceil,$$

where h = height of the identity index $-$ 1. Therefore, the update cost for modifying a simple attribute s:

$$SPDU_{simple} = SII + 2\lceil SS/P \rceil$$

To update the complex attribute A_{i+1} of O_i from ϕ_{i+1} to ϕ'_{i+1}, the identity index is searched to read and write back the s-expressions corresponding to ϕ_{i+1} and ϕ'_{i+1}. On the other hand, the s-expression addresses of the ancestor objects of O_i have to be updated in the identity index. Therefore, the average update cost for modifying a complex attribute is:

$$PDU_{complex} = \frac{(SII+2)}{n-1} \sum_{i=2}^{n-1} \sum_{j=1}^{i-1} Sr_j \cdot Sr_{j+1} \cdots Sr_{i-1} + 2(SII + 2\lceil SS/P \rceil)$$

4.4.3 Comparison

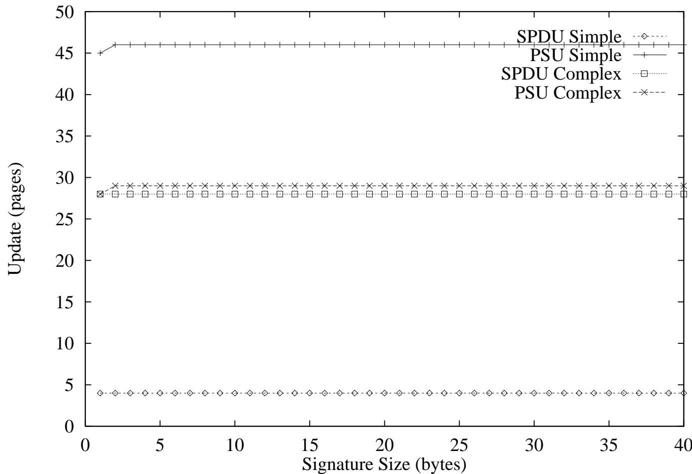


Figure 8: Update cost vs signature size.

Figure 8 compares the update cost of the two organizations with different signature sizes. We show both the update cost for updating a simple attribute and that for updating a complex attribute. It can be observed that the number of page accesses needed for updating the signature path dictionary is less than that for updating the path signature. For the signature path dictionary, the update cost for a simple attribute is lower than that for a complex attribute, because there is no need to spread signature update to its ancestor objects. On the other hand, the path signature is required to spread signature updates to ancestor objects for both of the simple and complex attributes.

5 Conclusion

Nested object query processing is an important problem in the research of object-oriented database systems. Nested object traversal and associative evaluation of the predicates in a query are two of the critical issues involved with nested processing. Path dictionary and signature techniques have been proposed to resolve the above mentioned issues, correspondingly. In this paper, we combine these two technique to propose a new secondary indexing organization, signature path dictionary, to support efficient processing of nested object queries.

In the paper, we present the concept and implementation of the signature path dictionary. Cost models for the storage overhead, retrieval cost and update cost of these two organizations are developed. Using these models, we compare the cost of the signature path dictionary to an enhanced version of the path signature method. The comparison shows that the path dictionary yields a dramatic improvement on the storage, retrieval and update cost.

Next, we want to extend the signature path dictionary technique to distributed object management environments. We believe that caching the signature and path information of objects into the main memory will significantly improve the performance of object-oriented database systems.

References

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

[2] E. Bertino & C. Guglielmina, "Path-index: An approach to the efficient execution of object-oriented queries," *Data and Knowledge Engineering*, North-Holland, 1993, 1–27.

[3] C. Faloutsos & S. Christodoulakis, "Signature files: an access method for documents and its analytical performance evaluation," *ACM Transactions on Office Information Systems*, Vol. 2, No. 4, Oct. 1984, 267–288.

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

[5] 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.

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

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

[8] S. Stiassny, "Mathematical analysis of various superimposed coding methods," *American Documentation*, Vol. 11, No. 2, February 1960, 155–169.

[9] 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.