Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

An Efficient Query Processing Algorithm on Object Databases with Multi-valued Attributes

Jou-Hui Lin and Arbee L.P. Chen
Institute of Computer Science, NTHU, Hsinchu, Taiwan, R.O.C.

Abstract

The signature-based indexing techniques support evaluating complex queries against all attributes of arbitrary classes in OODBs. However, the false drop is an intrinsic characteristic of signature-based techniques. The phenomenon becomes serious in OODBs with multi-valued attributes. In this paper, we propose to partition the schema graph for reducing false drop probability. Besides, we analyze quantitatively our approach and the multiindex technique (MX) in terms of storage cost and retrieval cost. The results show that our approach requires less storage overhead and achieves better performance in the average case than MX.

1. Introduction

Object-oriented databases provide more semantic constructs than relational ones. The domain of an attribute of an object can be any class. A class and its aggregation relationships form a hierarchy structure called *class-composition hierarchy*. An attribute of which domain is a primitive class (e.g., interger, string, etc.) is defined as the *primitive attribute*; otherwise, it is defined as the *complex attribute*. Attributes may be either multi-valued or single-valued. For a class-composition hierarchy rooted at class C, all primitive attributes of classes in this hierarchy are *nested attributes* of class C.

A predicate in an object query is of the form **<path-expression *operator* value>** where **value** is called the *predicate value*. A *path* is expressed as $C_1.A_1.A_2..A_n$ where class C_i is the domain of attribute A_{i-1} of class C_{i-1} for any $1<i\leq n$. C_1 is called the *starting class* of the path; A_n is called the *predicate attribute* of the path. A *single-valued predicate* is a predicate of which the predicate attribute is single-valued; likewise, a *multi-valued predicate* is a predicate of which the predicate attribute is multi-valued. Several indexing techniques have been proposed to efficiently utilize OODBs [1][2][3][4][8]. Whereas the efforts are concentrated extensively on the single-valued predicates [1][2][4][8], the work concerning multi-valued predicates is seldom mentioned [3].

Traditionally an index provides a direct association between its predicate attribute and its starting class. In general, the indices are maintained only for some specific paths that are frequently specified in the predicates. The predicate is evaluated by a forward traversal if no index is constructed for this path.

Signature files are originally applied in the field of text retrieval [5][6][7]. Their capabilities in OODB environment have gradually been discussed [3][4][8]. A signature-based index is constructed for a specific class. Related information of an object is involved in its signature; therefore, a signature-based index can evaluate a query against arbitrary nested attributes of the specific class. In contrast to the traditional techniques, it maintains a single index for a specific class. In addition, signature files are generally promising as set access facilities [3]. However, the study is limited to one class. We extend the exploration to a class-composition hierarchy. Moreover, complex attributes which are multi-valued are allowed. To the best of our knowledge, that has not been reported.

An *atomic signature* is a bit string generated by hashing a data term. For an object, each nested attribute which is single-valued derives its atomic signature. Similarly, each element in a multi-valued nested attribute is hashed to get its atomic signature. An *object signature* is then generated by OR-ing, namely *superimposing*, all atomic signatures of this object. Data collision can occur due to characteristics of hashing and superimposing. More atomic signatures an object signature derives from, more serious the phenomenon becomes. Consequently, we propose to partition a given schema graph into several components. A signature-based index is constructed for each component; likewise, a signature-based index is constructed for each multi-valued primitive attribute. We can process arbitrary queries via these indices. The detail is described later.

The remainder of this paper is organized as follows. An overview of indexing techniques is given in Section 2. A new signature-based scheme is proposed in Section 3. Section 4 develops simulations based on models of storage cost and retrieval cost. Performance analyses are done for some query types. Finally, we conclude the paper in Section 5.

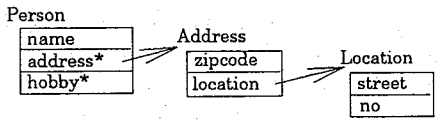


Figure 2.1: A schema graph

2. Survey of Indexing Techniques

An example schema graph of a simple database is shown in Figure 2.1 for the discussion. The '*' symbol next to an attribute name indicates that the attribute is multi-valued. Such a representation is used throughout this paper.

2.1 Multiindex Technique

An entry in the nested index (NX) for a given path $C_1.A_1.A_2..A_n$ is of the form <*v*, ***oid***>. It denotes that an object in class C_1 with ***oid*** has *v* as the value of its nested attribute A_n. It is impossible to create an NX for each possible path in a class-composition hierarchy due to high storage overhead.

To support random queries, it must make indexing the entire class-composition hierarchy possible. A simple index is created for each possible path with unit length via the concept of the multiindex (MX). There exists a set of n simple indices for path $C_1.A_1.A_2..A_n$. Each simple index is allocated to $C_i.A_i$ ($1 \le i \le n$). The last index is first used for evaluating predicates. The results are used as keys for the next search preceding the last one and so on until the first one is reached. In contrast to NX, the storage overhead is more regular and reasonable. MX organization reduces the storage overhead while providing efficient retrieval.

2.2 Signature-based Technique

Each nested attribute of an object in class C derives its signature. Subsequently, these signatures are superimposed together to form the object signature [8]. These object signatures are stored sequentially in the signature file. Besides, there exists a corresponding OID file. When a query of which all predicates originating from class C is given, a *query signature* is formed from predicate values in a similar way. Each entry in the signature file is examined. For all bit positions set to "1"s in the query signature, if the same bit positions in an entry are also set to "1"s, the corresponding object becomes a candidate, namely a *drop*. The candidates obtained may not be what we really want. These false qualified objects are called *false drops*. The objects which actually satisfy the predicates are called *actual drops*. *False drop probability* is defined as

$$\frac{\text{false drops}}{N - \text{actual drops}}$$

where N is the number of objects in class C. The signature-based indexing technique performs well when the false drop probability is low.

2.2.1 Discussion of Single-valued Predicates

Let us consider the following query:

select *Address*
where *Address.location.**street*** is-equal **"Central"**
and *Address.location.**no*** is-equal **"100"**

For each object in class *Address*, its nested attributes *zipcode*, *street* and *no* are hashed by function SIG(). They are superimposed together to form an object signature. The query signature is generated by superimposing SIG("Central") on SIG("100"). The signature file for class *Address* is examined, then the qualified objects are verified.

2.2.2 Discussion of Multi-valued Predicates

Let us consider another query given as follows:

select *Person*
where *Person.**hobby*** has-subset **{"Swimming", "Hiking"}**

Superimposing SIG("Swimming") on SIG("Hiking") generates the query signature. For each object in class *Person*, each element in attribute *hobby* derives its atomic signature. The *set signature* is formed by superimposing all atomic signatures obtained above. The set signature is assigned to the corresponding object signature if queries are against attribute *hobby*.

Otherwise, the object signature is determined by superimposing all the set signatures and atomic signatures of the rest nested attributes. False drop probability increases greatly as a result of the increment of atomic signatures from which the object signature is derived. Higher false drop probability requires more object verifications. That is illustrated by the following query:

select *P*
from *Person P*, *A* **is-in** ***P.address***
where *A.zipcode* is-equal **"108"**
and *A.location.street* is-equal **"Central"**
and *A.location.no* is-equal **"100"**

An *object graph* based on the schema in Figure 2.1 is shown in Figure 2.2. The object signature of object P1 is formed by OR-ing SIG("Austen"), SIG("108"), SIG ("Garden"), SIG("27"), SIG("300"), SIG("Central"), SIG("100"), SIG("Music") and SIG("Movie"). The query signature of the above query is derived in a similar way. The object signature of P1 involves SIG("108"), SIG("Central") and SIG("100") which determine the query signature. P1 is a drop; however, P1 does not satisfy the above query in fact.

The control of false drop probability is important to achieve better performance. Consequently, such a schema is partitioned for reducing the false drop probability in our approach. The phenomenon described above can be avoided.

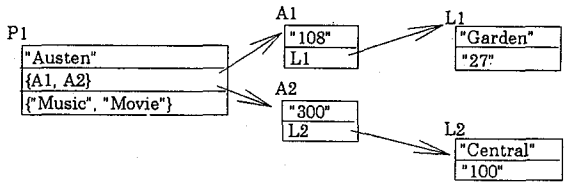


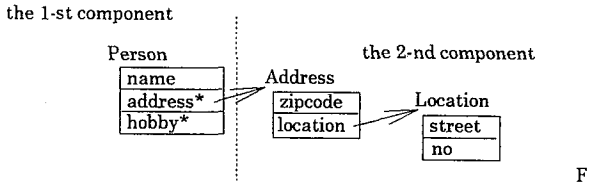
Figure 2.2: An object graph of the schema in Fig. 2.1

3. New Signature-based Scheme

In this section, after describing how to partition a given schema, we show how data are processed and organized. Subsequently, predicates are classified for later discussion. Query processing employing the developed structure is introduced finally. In the following, a *value_set attribute* means that a primitive attribute is multi-valued; while an *oid_set attribute* means that a complex attribute is multi-valued. Our approach is abbreviated to the SIG Scheme.

3.1 Partition Rule

A connection in a schema graph between an oid_set attribute and its domain class is chosen and then removed. Such an operation is continued until there is no such a connection. Hence, the schema graph is partitioned into several disconnected components. A signature-based index is available for each value_set attribute in the schema. For each object in class C which defines a value_set attribute A, a *set signature* SIG_{set} is generated. For a component, all nested attributes except the value_set attributes of an object in the root class of the component are hashed, and the results obtained are superimposed together to get a *component signature* SIG_{comp}. Set signatures and component signatures are stored separately.



F

igure 3.1: The partitioned schema based on that in Fig. 2.1

The schema in Figure 2.1 is partitioned into two components, as illustrated in Figure 3.1. The first component exists a signature file for attribute *hobby*. Besides, each entry of the signature file for the first component is from attribute *name* of the corresponding object. As to the second component, attributes *zipcode*, *street* and *no* of an object in class *Address* determine its component signature.

It is inevitable to maintain as many indices as classes to handle queries against arbitrary classes in a schema graph if the signature-based indexing technique [8] is applied. An atomic signature may be used more than once to generate object signatures in that approach. However, each atomic signature only contributes to a superimposing computation according to the generation of the SIG_{set} and the SIG_{comp} in our approach. Therefore, our approach achieves not only a smaller storage overhead but also a lower false drop probability than the scheme previously proposed.

3.2 Index Organization

There exists a simple index between any two adjacent classes. The values of all primitive attributes derive set signatures and component signatures, as described in Section 3.1. They are stored separately and their corresponding OID files are also maintained.

Each object in the root class of a component has its SIG_{comp}. Meanwhile, a SIG_{set} is available if there exists a value_set attribute in this class. These signatures are stored separately. For each one of the rest classes in this component, a SIG_{set} is available for each object of which the corresponding class defines a value_set attribute. Several signature file organizations have been proposed. *Sequential Signature File* (SSF) is the simplest one in which signatures are sequentially stored . A full scan is needed in evaluating a query. *Bit-Slice Signature File* (BSSF) stores signatures in a column-wise manner. A bit-slice file is for each bit position of the signatures. Only a part of bit-slice files have to be retrieved so that retrieval cost is lower. Therefore, we employ BSSF. The detail is described later. The organization is illustrated in Figure 3.2.

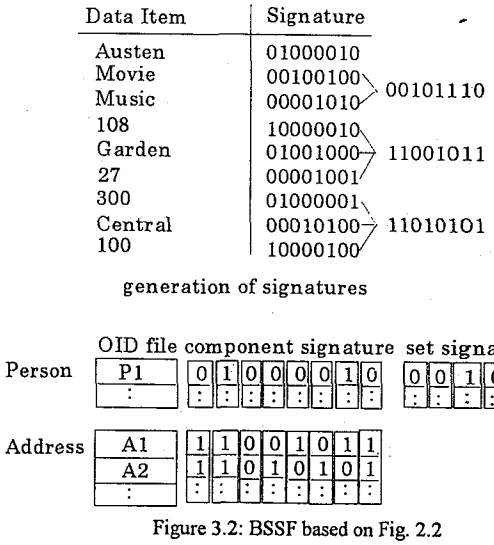


Figure 3.2: BSSF based on Fig. 2.2

3.3 Types of Predicates

We classify the predicates in our discussion into two main groups. One contains all *simple path expressions* in which

there are no oid_set attributes included. The other contains *complex path expressions* which include at least one oid_set attribute.

Some sample queries illustrating above situations are given below.

Query1: select *Address*
where *Address.**zipcode*** is-equal "100"

Query2: select *Address*
where *Address.**hobby*** has-subset/is-subset
{"Swimming", "Hiking", "Jogging"}

The above queries are simple path expressions. An example of complex path expressions is described as follows:

Query3: select *P*
from *Person P*, ***Addr is-in P.address***
where *Addr.location.street* is-equal "Central"

The keyword exists (each) preceding an oid_set attribute *A* indicates that the predicate must hold for at least one object (for each object) in the set specified by *A*. We only consider exists due to easier estimation of the number of objects satisfying predicates. For the purpose of generalization, Query3 is simplified as follows.

Query3': select *Person*
where *Person.*exists
***address**.location.street* is-equal "Central"

When the number of predicates in which "exists address"s appear in Query3' is more than one, all "***exists address***"s are bound to the same object in class *address*. The prior works [1][2][3] only take a single path query into consideration. In fact, queries may contain predicates on several paths originating from the same class and having overlapping subpaths.

3.4 Query Processing

A component is a unit of processing; hence, for all predicates in a given query, they are initially grouped by the components in which their predicate attributes locate. Predicates locating in the same component are classified into two types. One is for those against the multi-valued attributes. A *set predicate signature* $\mathbf{PSIG}_{set}$ is derived by superimposing all atomic signatures from the predicate value in a multi-valued predicate. The other is for those against the single-valued attributes. These predicates are called a *component predicate*. For each component which includes a component predicate, a *component predicate signature* $\mathbf{PSIG}_{comp}$ is generated by superimposing all atomic signature from the predicate values in these single-valued predicates.

Multi-valued predicates are first processed. The $\mathbf{PSIG}_{set}$ checkes the corresponding bit-slice files. If a SIG_{set} qualifies the condition implied by the $\mathbf{PSIG}_{set}$, the corresponding object must be accessed and verified. Let us first consider the has-subset case. The checking procedure is specified as follows:

> For each bit position which is set to "1" in the $\mathbf{PSIG}_{set}$, the corresponding bit-slice file is retrieved. These bit-slice files are bit-wise AND-ed together. For a SIG_{set} of which AND-ed result is "1", the corresponding object becomes a drop.

Next, let us consider the is-subset case. The checking procedure is specified as follows:

> For each bit position which is set to "0" in the $\mathbf{PSIG}_{set}$, the corresponding bit-slice file is retrieved. These bit-slice are bit-wise OR-ed together. For a SIG_{set} of which OR-ed result is "0", the corresponding object becomes a drop.

The objects satisfying a multi-valued predicate form an OID set. Take the intersection of the OID sets if the class currently processed is the nearest common class of some multi-valued predicates. The elements in the resulting OID set serve as keys for the next search on the simple index associating the current class with the preceding class. The above procedure is repeated until objects, satisfying all multi-valued predicates in the current component, in the root class of the component are determined.

Next, we focus on the single-valued predicates. The SIG_{comp} of the object available is compared with the $\mathbf{PSIG}_{comp}$ to filter out impossible objects. Such a procedure is similar to the checking procedure applied in the has-subset case. The forward traversal is applied along all the single-valued predicates. The traversal of these predicates is preorder. Logically, all the single-valued and the multi-valued predicates in the class-compositon hierarchy rooted at the root class of the current component have been processed. The query processing is finished until all the components have been processed.

3.5 Illustrative Example

To illustrate the scheme we described thus far, a query is given as follows:

select *Person*
where *Person.**hobby*** is-subset
{"Swimming", "Movie", "Music"}
and *Person.*exists *address.**zipcode*** is-equal "300"
and *Person.*exists *address.location.**street*** is-equal
"Central"

The second and the third predicates belong to the second component. The first predicate is in the first component. $\mathbf{PSIG}_{comp}$ for the second component is generated by superimposing SIG("300") on SIG("Central"). The

corresponding bit-slice files are examined. Object A2 is a qualified object, as observed in Figure 3.2. A2 is retrieved to check attribute *zipcode*. Through attribute *location*, object L2 is verified. It is confirmed that A2 satisfies the predicates.

P1 is obtained via the MX connecting class *Address* to class *Person*. Next, we start to process the first component. $PSIG_{set}$ for the first predicate is generated by OR-ing SIG("Swimming"), SIG("Movie") and SIG("Music"). P1 is what we want to retrieve because P1.*hobby* is actually a subset of {"Swimming", "Movie", "Music"}.

4. Simulations

The goal of the simulations is to confirm that the SIG scheme performs better than the MX technique in general. We compare the retrieval cost of the two approaches for queries which are boolean combinations of any kinds of predicates discussed in Section 3.3. Then, we investigate the influence of some parameters.

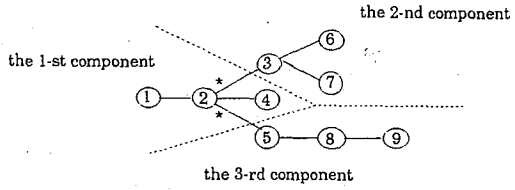


Figure 4.1: The schema graph used in simulations

The schema graph employed in the simulations is shown in Figure 4.1. Queries are randomly generated. First, the starting class of a query is randomly selected from classes in the schema graph. The number of classes involved in a query is also randomly determined. The query graph is generated by randomly forming branches. Meanwhile, we randomly determine whether there exists a predicate against a class in the query graph. The type and the cardinality of the predicate value for a predicate is also randomly determined. For some parameters, we adopt typical values [1]. We also assume that the size of a key is equal to the length of an OID. They are given in Table 4.1.

kl	oidl	noid	f	P
8	8	2	218	4096

Table 4.1: Values of some parameters

We first discuss false drop probability which determines the number of objects which need to be verified. Next, parameters used and assumptions made in cost models are listed. Then, the models of storage cost and retrieval cost are formulated. Simulations and their results are finally shown.

4.1 False Drop Probability

A false drop is an intrinsic characteristic of signature-based techniques. It directly affects the performance of query processing. False drop probability greatly depends on the signature size. If a larger signature size is applied, false drop probability is smaller. However, it increases the storage overhead. There exists a trade-off between storage cost and retrieval cost. Factors affecting false drop probability are listed as follows:

- m number of "1"s in an atomic signature
- m_p number of "1"s in a set predicate signature or a component predicate signature
- D_{t,A_i} cardinality of the target set of attribute A_i
- D_{p,A_i} cardinality of the predicate set on attribute A_i
- D_{t,l_i} number of primitive attributes which are single-valued in the i-th component
- D_{p,l_i} number of single-valued predicates against the i-th component
- F signature size in bits
- F_{d,A_i} false drop probability for a multi-valued predicate against attribute A_i
- F_{d,l_i} false drop probability for a component predicate against the i-th component

Let us consider the example in Section 3.5. $D_{p,hobby} = 3$, $D_{t,t_1} = 1$, $D_{p,l_1} = 0$, $D_{t,t_2} = 3$ and $D_{p,l_2} = 2$. D_{t,A_i} is determined by the system and can be varied during simulations.

For the predicate against primitive attribute A_i and $D_{t,A_i} \geq D_{p,A_i}$, F_{d,A_i}, as developed in [3], approximates

$$\left(1 - e^{-\frac{m}{F}D_{t,A_i}}\right)^{mD_{p,A_i}}$$

Similarly, for the predicate against primitive attribute A_i and $D_{t,A_i} \leq D_{p,A_i}$, F_{d,A_i} approximates

$$\left(1 - e^{-\frac{m}{F}D_{p,A_i}}\right)^{mD_{t,A_i}}$$

Next, let us consider F_{d,l_i}. Each object in the root class of the i-th component has a SIG_{comp} determined from all single-valued attributes of this component. A $PSIG_{comp}$ is determined from all single-valued predicates of which predicate attributes locate in this component. Such a situation is logically considered as the $D_{t,A_i} \geq D_{p,A_i}$ case. Therefore, the number of single-valued attributes, D_{t,l_i}, in the i-th component substitutes for D_{t,A_i}; likewise, the number of single-valued predicates, D_{p,l_i}, in the same component substitutes for D_{p,A_i}. F_{d,l_i} becomes

$$\left(1 - e^{-\frac{m}{F}D_{t,l_i}}\right)^{mD_{p,l_i}}$$

4.2 Notations and Assumptions

Parameter notations used to measure the performance are given in the following:

N_i number of objects in class C_i

K_i number of objects in class C_i which have the same OID(s) in complex attribute A_i

V_i cardinality of the domain of a single-valued attribute of class C_i

MV_i cardinality of the domain of a value_set attribute of class C_i

$OIDl$ length of an OID in bytes

P page size in bytes

LC_i look-up cost for OID file of class C_i

MXC_{A_i} cost for evaluating a single-valued predicate against attribute A_i using multiindex structure

$SIGC_{I_i}$ cost for evaluating a component predicate against the i-th component using the SIG scheme

$SIGC_{A_i}$ cost for evaluating a multi-valued predicate against attribute A_i using the SIG scheme

To simplify cost models, we make assumptions as follows:

- A specific schema graph is given.
- Each object of C_j is referenced by some objects of C_i where class C_j is the domain of an attribute of class C_i.
- Distribution of attribute values of a class is uniform.
- All attributes of a class are independent; hence, the value of an attribute does not affect the value of any other attribute.
- All objects in a class have the same D_{t,A_i} for each multi-valued attribute A_i.

4.3 Cost Model

Actual drops are investigated after formulating the storage costs and retrieval costs of the MX technique and the SIG scheme.

4.3.1 Storage Cost

MX and SIG organizations are similar except information involved in the primitive attributes. Only the difference is considered.

MX Technique: An MX structure is based on a B-tree. A leaf entry is composed of a key value and the list of OIDs for objects which hold the key value in the corresponding attribute. Parameters for MX are shown as follows:

kl size of a key value in bytes

k_i number of objects in class C_i, which share the same value in primitive attribute A_i

$noid$ size of a field which records the number of OIDs

ll_i size of a leaf entry for primitive attribute A_i

lp_i number of leaf pages for primitive attribute A_i

$LP_{j,i}$ number of leaf pages for complex attribute A_i of which domain class is C_j

nlp_i number of nonleaf pages for primitive attribute A_i

f average fanout of a nonleaf node

The size of a leaf entry ll_i is

$$kl + k_i * OIDl + noid$$

where $k_i = N_i / V_i$ for a single-valued attribute and $k_i = D_{t,A_i} * N_i / MV_i$ for a multi-valued attribute. The number of leaf pages is

$$lp_i = \begin{cases} \lceil V / \lfloor P / ll_i \rfloor \rceil & \text{if } ll_i \le P \\ V * \lceil ll_i / P \rceil & \text{if } ll_i > P \end{cases}$$

where $V = V_i$ for a single-valued attribute and $V = MV_i$ for a multi-valued attribute. Then, the number of nonleaf pages is

$$nlp_i = \lceil lp_i / f \rceil + \lceil \lceil lp_i / f \rceil / f \rceil + \cdots + X$$

where the last term X is less than f. If X is not 1, 1 is added to nlp_i. The number of terms is h_i.

The total storage cost of MX is the sum of the number of leaf and nonleaf pages of all primitive attributes in the hierarchy.

SIG Scheme: The SIG organization consists of two kinds of files: OID files and bit-slice files. For the root class of each component, there exists an OID file. In addition, there exists an OID file for each class defining value_set attributes. The size of OID file for such a class C_i in pages is

$$\lceil N_i / \lfloor P / OIDl \rfloor \rceil .$$

Then, there exists two kinds of bit-slice files: for SIG_{set} and for SIG_{comp}. Only component root classes and classes which define value_set attributes maintain bit-slice files. The storage cost of bit-slice files for such a class C_i is

$$\lceil N_i / 8P \rceil * F .$$

The total storage cost of SIG is the sum of storage costs of all OID files and bit-slice files.

4.3.2 Retrieval Cost

Queries we investigate include predicates on several paths originating from the same class and having overlapping subpaths. Therefore, we do not process a unit until all units which succeed the class have been evaluated. A unit means

a class for MX technique. However, a unit denotes a component for SIG scheme. The number of satisfactions of the current class can be estimated easily due to independence of attributes in a class.

Since the I/O cost determines mainly the performance, we only consider the number of disk page accesses during query processing. To clarify the presentation of the formulations, we conduct the cost models based on a schema and query graph in Figure 4.2 which is a link list structure rather than a tree structure. Class C_{i+1} $(1 \le i(n)$ is the domain of complex attribute A_i of class C_i. There exists a predicate $Pred_i$ $(1 \le i \le n)$ against primitive attribute A_i of class C_i. The schema is partitioned into m components. $C_{r(\ell_i)}$ denotes the root class of the i-th component.

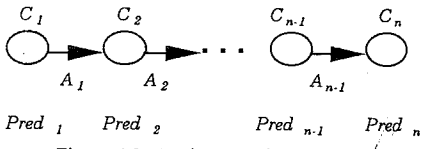


Figure 4.2: A schema and query graph

MX Technique: For a single-valued predicate, retrieve the OIDs corresponding to the predicate value. Otherwise, each element in the predicate set derives the corresponding OIDs. Take the intersection of the OID sets if $D_{t,A_i} \ge D_{p,A_i}$, or take the union of the OID sets. However, the results of the latter must be retrieved to verify whether they actually satisfy the $Pred_i$.

$Pred_i$ does not be processed until all $Pred_j$ $(i(j \le n)$ have been processed. I_i objects of class C_i are available to be considered before evaluating $Pred_i$. After evaluating $Pred_i$, S_i objects of class C_i satisfy predicates which have been processed. I_n is initially equal to N_n. For 1≤i<n,

$$I_i = \begin{cases} S_{i+1} * K_i & \text{if } A_i \text{ is singlevalued} \\ N_i - C(N_{i+1} - S_{i+1}, D_{t,A_i}) * K_i & \text{if } A_i \text{ is multivalued} \end{cases}$$

The total retrieval cost consists of the evaluation of predicates and the mapping from C_i $(2 \le i \le n)$ to C_{i-1}. To estimate the number of page accesses for the latter, we utilize a formula developed in [9]. This formula H(k,m,n) determines the number of page hits when accessing k records randomly selected from a file containing n records grouped into m pages. S_i objects are served as keys to obtain I_{i+1} objects. It costs $H(S_i, LP_{i,i-1}, N_i) + h_{i,i-1}$ page accesses where $h_{i,i-1}$ is the height of the tree for $C_{i-1}.A_{i-1}$ where attribute A_{i-1} is primitive. The cost for evaluating $Pred_i$, which is single-valued is equal to

$$MXC_{A_i} = \begin{cases} h_i + 1 & if \quad ll_i \le P \\ h_i + \lceil ll_i / P \rceil & if \quad ll_i > P \end{cases}$$

The cost for evaluating $Pred_i$, which is multi-valued is

$$MXC_{A_i} * D_{p,A_i} + \alpha(Actual_i + I_i \frac{\sum_{k=1}^{D_{t,A_i}-1} C(D_{p,A_i}, k) * C(MV_i - D_{p,A_i}, D_{t,A_i} - k)}{C(MV_i, D_{t,A_i})})$$

where α is 1 if $D_{t,A_i} \le D_{p,A_i}$, or is 0. We know that the retrieval cost for the m-th component is

$$\sum_{i=r(m)}^{n} (rc_{i,s} + rc_{i,m}) + \sum_{i=n}^{r(m)-1} (H(S_i, LP_{i,i-1}, N_i) + h_{i,i-1}).$$

SIG scheme: For a component, multi-valued predicates are first evaluated and then single-valued ones. As for the m-th component, the $SIGC_{A_i}$ is

$$(1-\alpha)\lceil N_i/8P \rceil m_p + \alpha \lceil N_i/8P \rceil (F - m_p) + LC_i + Actual_i + F_{d,A_i}(I_i - Actual_i)$$

where α is 1 if $D_{t,A_i} \le D_{p,A_i}$, or is 0. The $SIGC_{\ell_i}$ is

$$\lceil N_i/8P \rceil m_p + LC_i + Actual_i + F_{d,\ell_i}(S_i - Actual_i)$$

where i is $r(\ell_m)$. Similarly, the retrieval cost also includes the mapping between two adjacent classes. Therefore, the cost for processing the m-th component is

$$SIGC_{\ell_m} + \sum_{i=r(\ell_m)}^{n} SIGC_{A_i} + \sum_{i=n}^{r(\ell_m)-1} (H(S_i, LP_{i,i-1}, N_i) + h_{i,i-1})$$

4.3.3 Actual Drop

For a multi-valued predicate against primitive attribute A_i, the expected number of objects satisfying $D_{t,A_i} \ge D_{p,A_i}$ is

$$I_i \frac{C(MV_i - D_{p,A_i}, D_{t,A_i} - D_{p,A_i})}{C(MV_i, D_{t,A_i})}.$$

As for the condition of $D_{t,A_i} \le D_{p,A_i}$, the expected number $Actual_i$ is

$$I_i \frac{C(D_{p,A_i}, D_{t,A_i})}{C(MV_i, D_{t,A_i})}.$$

For a single-valued predicate against primitive attribute A_i, the expected number is N_i/V_i.

Since all attributes of a class are independent, the estimation of intermediate results for class C_i is derived by multiplying selectivities of its all attributes by N_i.

4.4 Performance Analysis

4.4.1 Storage Overhead

Using the formulations in Section 4.3.1, we evaluate storage overheads for the two organizations. For $D_{t,A_i} = 10$, SIG scheme requires less storage than MX technique, as observed in Figure 4.3.

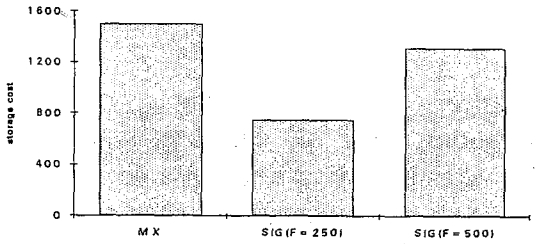


Figure 4.3: Storage cost

The ratios of the storage costs of SIG scheme to those of MX technique are about 49.7% (for F=250) and 87.3% (for F=500).

4.4.2 Retrieval Time

Object retrieval cost is influential using SIG; therefore, a low false drop probability is required. Figure 4.4 shows that the performance, for SIG, gets better due to lower false drop probability if a larger F (signature size) is applied. We vary the m value. The total retrieval cost is larger under a smaller m, as observed in Figure 4.4. With a larger m value, the retrieval cost becomes lower. However, the retrieval cost is less variable when m is larger than 16. When m approximates F/2, the false drop probability increases. Therefore, the retrieval cost becomes larger again.

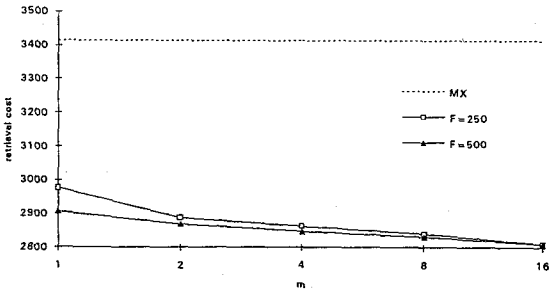


Figure 4.4: retrieval cost

Besides, SIG is superior to MX which is irrelevant to m, as indicated in Figure 4.4.

5. Conclusion and Future Work

In this paper, we present a new indexing scheme based on signature files to support query processing in OODBs with multi-valued attributes. A query can be a boolean combination of predicates on several paths having overlapping subpaths. A signature-based scheme is a good alternative since signature files support indexing retrievals for all attributes of arbitrary classes. However, false drops come with such a scheme. We propose to partition the class-composition hierarchy for decreasing false drops caused by multi-valued attributes while providing efficient retrieval. According to the generation of signatures [8], we know the multi-valued attributes directly influence the number of false drops. Therefore, we eliminate the connections between complex multi-valued attributes and their domain classes. Each component is allocated a signature-based index. Besides, each primitive multi-valued attribute also maintains a signature-based index.

We compare our approach, denoted as SIG, with the multiindex (MX). The quantitative analysis is performed in terms of storage overhead and retrieval time. From the viewpoint of the storage cost, SIG is certainly less costly. Therefore, a small available space for indices benefits from this approach. A random query is a boolean combination of the is-equal, has-subset and is-subset cases. For the retrieval cost, SIG outperforms MX.

The cost analysis is mainly based on the assumption that all the objects in a class which defines multi-valued attributes have the same cardinality. Our future work is to release this con-traint. Besides, other partition rules, different indexing techniques in different components, and update cost analysis deserve further study.

Bibliography

[1]E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects", IEEE TKDE, Vol.1, No.2, pp. 196-214, Jun. 1989

[2]E. Bertino, "Index Configuration in Object-Oriented Databases", VLDB, Vol.3, No.3, pp. 355-399, Jul. 1994

[3]Y. Ishikawa, H. Kitagawa and N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs", ACM SIGMOD, pp. 247-256, 1993

[4]H. S. Yong, S. Lee and H. J. Kim, "Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases", IEEE Data Engineering, pp. 518-525, 1994

[5]C. Faloutsos, "Access Methods for Text", ACM Computing Surveys, Vol.17, No.1, pp. 49-74, Mar. 1985

[6]C. Faloutsos, "Signature files: Design and performance comparison of some signature extraction methods", ACM SIGMOD, pp. 63-82, 1985

[7]D. L. Lee and C. W. Leng, "A Partitioned Signature File Structure for Multiattribute and Text Retrieval", IEEE Data Engineering, pp.389-397, 1990

[8]W. C. Lee and D. L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems", ICSC, pp. 612-622, 1992

[9]S. B. Yao, "Approximating Block Accesses in Database Organizations", ACM Communications, Vol.20, No.4, pp. 260-261, Apr. 1977