# CHAPTER 5

Chapter 5

# Nested query processing

One of the best features of OODBSs is that any real world entity can easily be modeled as an object. Every object has a set of attributes which may be simple or complex along with a set of methods that apply on these attributes. Each object has a unique Object Identifier (OID). When one object is referred as a complex attribute of another object, OID of the referred object is included in the referring object. This establishes an aggregation relationship between the two classes and the embedded object is said to be nested in the parent object.

## 5.1 Signatures and nested queries

In OODBSs each object is identified by a unique system-assigned OID. Objects can be nested by storing the child OIDs in the complex attributes of the parent objects. Thus, child objects can be forward referenced from their parents through the OIDs stored in the parents. Conversely, backward reference from children to parents can be created by explicitly creating a complex attribute in the child object referencing the parent. The nesting structure of objects can easily be represented using signatures. Since the signature of an object is the superimposed signature of its constituent attributes signatures, the embedded values of nested hierarchy as well is easily reflected in signatures.

Consider the example of nested object hierarchy (partly extracted from schema 3.10) shown in Figure 5.1. In OODBSs the search condition in a query is expressed as a boolean combination of predicates of the form <attribute operator value>. The attribute may be a nested attribute of the target class. For example, the query "List all Universities located in US offering Computer Science programme" can be expressed as

Select University

where Univeristy. Location = "US"

and University. Dept. Programme. Name = "Computer Science"

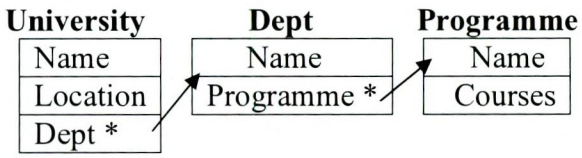


***Figure 5.1 A nested object hierarchy***

In this query the target class University has two predicates involving attributes Location and Dept. With the use of proper intermediate structure for indexing the search process can be speeded up; otherwise the sequential searching process will filter data one level at a time from the target class in a top-down manner. In this chapter an indexing structure for signatures is used to speed up the retrieval process which is summarized as follows:

- The signatures of each class in the class hierarchy are stored in separate SD-tree structures forming a hierarchy.
- Objects of classes with nested attributes form signatures by superimposing method.

- In each level except the target class the only matching OIDs' signatures obtained from the previous level are inserted in SD-tree for further searching.
- The object query is stored in a tree like structure to promote parallel comparison as in [CHEN 04].

The query signature tree for the path specified in the above query is constructed as in Figure 5.2. To remove irrelevant signatures as quickly as possible [CHEN 04] employs entries in signature file at each level a triple of the form <osig, oid, list> where list is the set of object identifiers referred from the current object. Although we adapt the lead of [CHEN 04] for query tree, use of SD-tree for classes at each level cuts down multiple path traversals thus adding more to optimality.

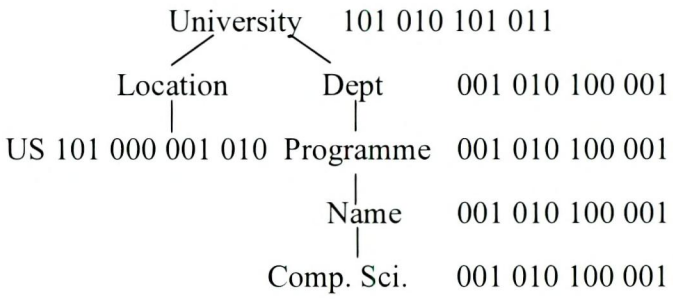


***Figure 5.2 Query signature tree***

## 5.2 Query Algorithm

The algorithm for query evaluation is outlined in Figure 5.3. We use $Stack_q$ to store signatures of the query tree bottom-to-top. Signatures of the target class are initially inserted in SD-tree.

**Algorithm** SD-tree based-retrieval

*Input:* An Object query Q.

*Output*: Set of signatures (OIDs) satisfying Q.

1. Push the signatures of query tree into $Stack_q$ from leaves to root.
2. Let S be the list of signatures of class pointed by the query signature.
3. Insert the signatures of S in SD-tree.
4. While $Stack_q$ not empty do

   Begin

   $S_q \leftarrow$ Pop $Stack_q$;

   Search($S_q$); // in the SD-tree

   Move the matching OIDs to S;
5. Go to step (3).
6. For the output signatures in S check false drops.

***Figure 5.3 Algorithm for nested query evaluation***

Compared to the top-down-hierarchy-retrieval algorithm of [CHEN 04] algorithm 5.3 saves time in step (3) by inserting only the filtered signatures from the previous level in every forward move. This is same as in [CHEN 04] but the search in step(4) retrieves the matching OIDs in a single access from SD-tree structure which reduces the average number of nodes searched and hence the retrieval time.

## 5.3 Experimental results

In order to evaluate the performance of our approach, let us consider the query format as in [YONG 94] which is shown in Figure 5.4. For each class Ci ($1 \leq i \leq n-1$) the domain of each attribute Ai is the subsequent class $C_{i+1}$. Each $p_i$ is the query predicate of $C_i$ in the query path.

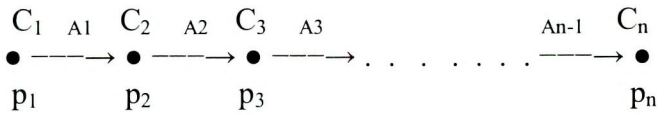


***Figure 5.4 A general query tree***

Parameters used for performance evaluation follow the lead of [CHEN 04] as listed below:

$N_i$ Total number of objects in $C_i$.

$P_i$ Probability that an object in $C_i$ satisfies predicate $p_i$ .

$v_i$ Number of objects visited in the algorithm in class $C_i$.

$P_f$ False drop probability of signatures.

$P_q$ Probability that a signature matches the query signature in the search procedure.

d Average out-degree of objects in any level of hierarchy.

$P_g$ : Probability that the tree edge traversed leads to matching signatures.

These parameters are used to estimate the number of objects retrieved and the average number of nodes checked during nested query processing for both Top-down-hierarchy-retrieval (TDHR) and SD-tree-based-retrieval (SDTR).

### 5.3.1 Top-down-hierarchy-retrieval

Using the analysis of [YONG 94] the total number of objects retrieved (v) is given by

$$v = \sum_{i=1}^{n} vi = v1(1+d \sum_{i=2}^{n} (Pi + (1 - Pi)\, Pf) \prod_{j=1}^{i} P\, j-1)$$

where (Pi + (1 – Pi) Pf) is the probability that an object is not removed while comparing with the referenced objects' signatures. TDHR method cuts down irrelevant objects' comparison at each level in the query path. If this impact is denoted by the probability Pq then, the total number of objects visited in TDHR method (v') is given by,

$$v' = \sum_{i=1}^{n} vi' = v1.\, Pq.(1+d \sum_{i=2}^{n} (Pi + (1 - Pi)\, Pf) \prod_{j=1}^{i} P\, j-1)$$

### 5.3.2 SD-tree-based-retrieval

SDTR follows the same procedure as in TDHR and hence the predicate check probability $P_q$ is present here also. In addition SDTR employs better object retrieval in the search procedure. This is due to the fact that in the first level of query processing the target class' signatures are inserted and the search method reduces the average number of nodes checked substantially. Moreover, for a signature tree with n signatures, each is of length m with k set bits the average number of nodes checked during query processing [CHEN 06] is given by $O\ (\ n^{1\ -\ (k\ /m)}\ )$. For optimal solution k/m = 0.5 and hence the value becomes $O(\sqrt{n})$ against the O(T) in SD-tree where T is the threshold value fixed as h+1 for tree height h. This gain is propagated for all query predicates in the search path. Let the probability of this gain is $P_g$, then the number of objects visited (v'') is given by

$$v'' = \sum_{i=1}^{n} vi'' = v_1 . P_q . P_g \left(1 + d \sum_{i=2}^{n} (P_i + (1 - P_i) P_f) \prod_{j=1}^{i} P_{j-1}\right)$$

Since most of the work in this line has been carried out on *abstract data sets*, the TDHR and SDTR methods are compared with the out degree of objects in each level (i.e) d = 1 as shown in Figure 5.5.

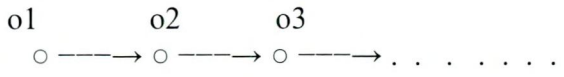


***Figure 5.5 Experiment Data Set***

Figure 5.6 shows the comparison results for TDHR and SDTR for the readings listed in Table 5.1.

### 5.3.3 Time complexity

For analysis, we assume three classes C1, C2 and C3 in the class hierarchy with values $P_i$ = 0.1, $P_f$ = 0.01, $P_q$ = 0.5 and $P_g$ = 0.01. (Note that $P_g$ is minimum because h < n). It is clear from the graph that considerable time is saved by employing SD-tree structure to filter data in the object hierarchy. Also for various input file size the number of nodes checked during query processing in signature tree and SD-tree was observed. The observed values are listed in Table 5.2 and results plotted in Figure 5.7.

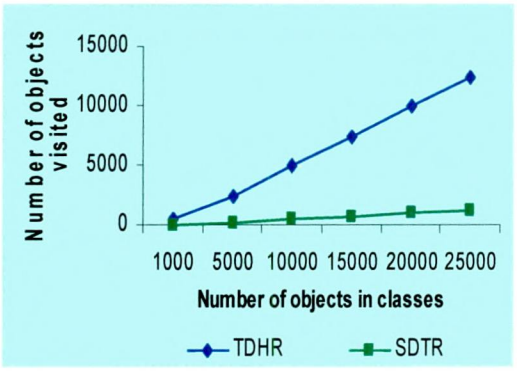


*Figure 5.6 Comparison results*

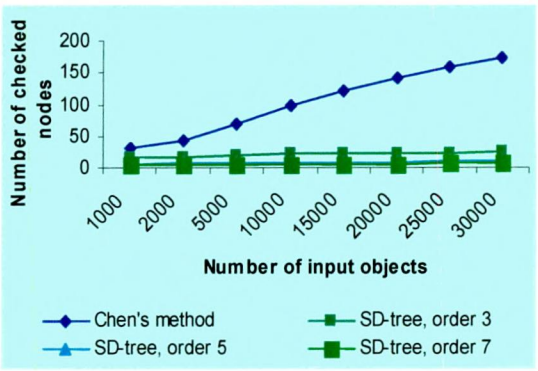


*Figure 5.7 Time complexity*

The graph indicates that the reduction of the number of checked nodes is substantial in SD-tree and by further increasing the tree order the number goes down. This is because higher the order more the number of key values accommodated in B+ tree nodes.

To evaluate nested queries we implement SD-tree in Java using the same hardware set up specified in section 3.4. The ***biological dataset of Gentleman-lab*** is taken for experiments fixing the data block size D as 3. For optimal results signature length (F) and the number of set bits (m) are fixed as 16 and 4 (to satisfy F ln 2 = mD). There are three classes C1, C2 and C3 considered in the hierarchy and the number of objects varied from 1000 to 3000 with out degree 1 as in Figure 5.5. The corresponding readings observed are listed in Table 5.3. The time taken for processing queries was noted for signature tree based retrieval and SD-tree based retrieval is shown in Figure 5.8. It is obvious that SD-tree based retrieval process saves considerable searching time.

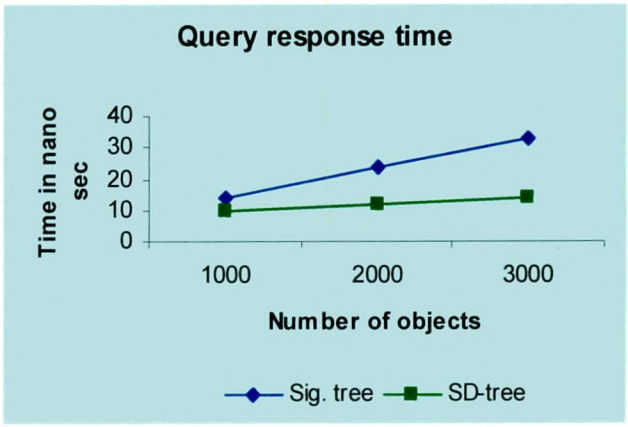


*Figure 5.8 Query response time*

*Table 5.1 Number of visited objects*

| Number of objects in classes | TDHR | SDTR |
|---|---|---|
| 1000 | 500 | 50 |
| 5000 | 2500 | 250 |
| 10000 | 5000 | 500 |
| 15000 | 7500 | 750 |
| 20000 | 10000 | 1000 |
| 25000 | 12500 | 1250 |

*Table 5.2 Number of checked nodes*

| Number of input objects | Chen's method | SD-tree | | |
|---|---|---|---|---|
| | | d = 3 | d = 5 | d = 7 |
| 1000 | 31 | 17 | 7 | 5 |
| 2000 | 44 | 18 | 8 | 5 |
| 5000 | 70 | 21 | 9 | 6 |
| 10000 | 100 | 22 | 10 | 7 |
| 15000 | 122 | 23 | 10 | 7 |
| 20000 | 141 | 24 | 10 | 7 |
| 25000 | 158 | 24 | 11 | 8 |
| 30000 | 173 | 25 | 11 | 8 |

*Table 5.3 Query response time (in* **Sec x 10** $^{-9}$*)*

| No. of objects | Signature tree | SD-tree |
|---|---|---|
| 1000 | 14 | 10 |
| 2000 | 24 | 12 |
| 3000 | 33 | 14 |

### 5.3.4 Space overhead of SD-tree

Every indexing technique is not without its trade-offs. This is true for the structure proposed in this paper also. For a signature file of size N, the space overhead of signature tree which is a balanced binary tree is given by [CHEN 04],

$$N \times \log_2 m + 2 \sum_{i=0}^{k} 2^i (i+1) \quad \text{where } k=\log_2 N \text{ and } m = \text{signature length}$$

On the other hand SD-tree is B+ tree like structure with signature nodes attached to leaf nodes. Hence the space complexity of SD-tree is the sum of the space complexity of B+ tree and signature nodes. The maximum number of records for a B+ tree of order b with height h is $b^h$. In signature nodes for inserting binary prefix and signature numbers nodes are created dynamically. Hence in the worst case a signature node at position i will have $2^{i-1}$ combinations for binary prefix. If n is the average number of signatures inserted per signature node then the total space overhead of SD-tree is given by

$$b^h + n \sum_{i=1}^{m} 2^{i-1}$$

This is less than the space overhead of signature tree. Also in SD-tree the values of b, h and m can be balanced to result in a shorter tree thus minimizing the space complexity of B+ tree structure. For higher values of N with signatures' weight biased in upper bytes, the space complexity of SD-tree may increase due to the replication of signature numbers for all set bits in signature nodes and complexity of signature node structure in upper levels. But for the same file size the time complexity is tremendously reduced compared with signature tree and maintenance cost as well is very low in SD-tree.

## 5.4 A Sample validation model

To validate the proposed SD-tree based retrieval let us consider the query signature tree given in Figure 5.2. Assume that part of the signature file hierarchy constructed for a database with the schema shown in Figure 5.1 is of the form shown in the upper part of Figure 5.9.

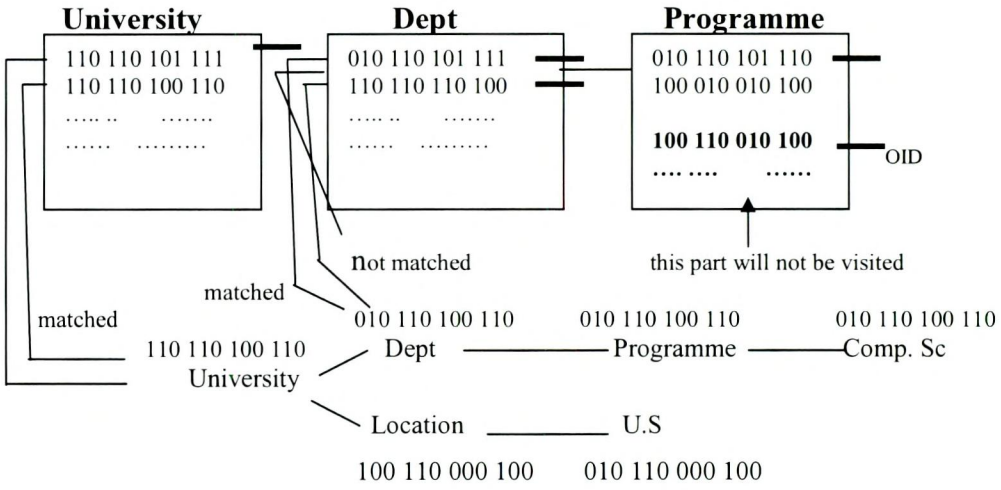


***Figure 5.9 Example of query evaluation***

Since both the top two signatures in the signature file for *University* (say U-file for short) match the corresponding signature in the query signature tree, the signatures referenced by them in the signature file for *Dept* (say D-file for short) are further checked. Assume that the first signature in *D*-file is referenced by the first signature in U-file, while the second one in D-file is referenced by the second one in *U*-file. We can see that the second signature in *D*-file does not match the corresponding signature in the query signature tree. Thus, all those *Programme* object signatures referenced by it will not be checked further (the marked part in Figure 5.9). This is optimal compared to "*top-down-retrieval*" since by means of "*top-down-retrieval*", checking against all *Division* object signatures has to be performed. Using SD-tree for storing signatures of each class further promotes fast searching of matching signatures thus adding more to optimality.

The query signature tree shown in Figure 5.2 is of type TP. The same procedure can be applied for processing queries of type PT also. In PT queries the target class is a nested class of the predicate classes. From Figure 5.1 consider the example query "List all programme names offered by Universities located in US". This query is of type PT which can be expressed as

Select Dept. Programme. Name

where Univeristy. Location = "US"

To process this query the objects of University class satisfying the condition University.Location = "US" are first retrieved. This output list is the input for the

subsequent step. Next in the query processing, only for the objects in the output list obtained, the programme name is retrieved through the link Dept. Programme. Name. As in TP query processing object filtering occurs at every level and the whole process is speeded up.

In this chapter we have proposed an indexing method for handling nested object queries of type TP and PT. In order to optimize search process we employ SD-tree structure that cuts down irrelevant branches as well as the number of checked nodes. This improves the search time complexity. The limitation is that the space overhead is more in SD-tree due to heavier signature nodes in the upper level and signature number replication for all set bits of the signature. Nevertheless the novelty of the structure and its low maintenance cost outweigh this.