CHAPTER 3

Chapter 3

Signature Declustering tree

Signature file based access methods initially applied on text have now been used to handle set-oriented queries in OODBs. All the proposed methods use either efficient search method or tree based intermediate data structure to filter data objects matching the query. Use of search techniques retrieves the objects by sequentially comparing the positions of 1s in it. Such methods take longer retrieval time. On the other hand tree based structures traverse multiple paths making comparison process tedious. This chapter describes a new indexing technique for OODB using the dynamic balancing of B+ tree called ***Signature Declustering (SD)***-tree. In this work the positions of 1s in the signatures are distributed over a set of leaf nodes. Using this for a given query signature all the matching signatures can be retrieved cumulatively in a single node. Also for signature insertion and query searching an optimal search path is calculated so that the entire process is speeded up. Experiments have been conducted to analyze the time and space overhead of the SD-tree by varying the signature length and the distribution of signature weight. The study clearly indicates the advantage of fast retrieval time in a way quite different from the other methods suggested in the past.

3.1 The structure of SD-Tree

In this section we describe the structure of SD- tree. The tree is constructed for the given signature length value (F) regardless of the number of signatures in the input file (n). There are three types of nodes:

- Internal nodes
- Leaf nodes
- Signature nodes

The internal nodes form the upper tree and leaf nodes at the last but one level as in B+ tree [COMER 79], [ELMASRI 89] . The signature nodes are at the bottom level of the SD-tree. The internal nodes and leaf nodes are somewhat similar to the internal nodes and leaf nodes of B+ trees respectively. Let us see the structure of the nodes in detail. To make discussion simple, we assume the tree order as 3 for a signature file with 8 block signatures of length 12.

3.1.1 Structure of an internal node

An internal node of SD-tree is illustrated in Figure 3.1. Like B+ tree the internal node has pointers and keys that alternate each other. For a tree of order 3 the internal node has two keys K1 and K2 and three pointers P1, P2, P3. These pointers are tree pointers pointing to the nodes at the lower level. While searching, the left tree pointer is followed for values less than or equal to the node value, else right pointer is followed for values greater than the node value as in B+ tree.

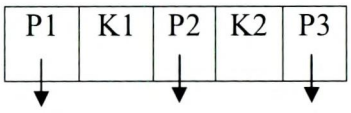

Figure 3.1 A typical structure of an internal node

3.1.2 Structure of a leaf node

The leaf nodes appear in the last but one level of the SD–tree. Like B+ tree all the key values appear in ascending order of their values in the leaf nodes and are connected to promote sequential search. But unlike B+ tree in SD-tree each value is followed by a signature node instead of data pointer. This is depicted in Figure 3.2. Pointers P1, P2 point to the corresponding signature nodes for K1, K2 and P3 is the sequential pointer to next leaf node.

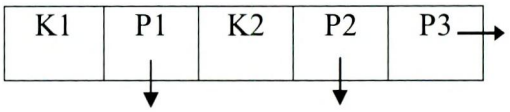

Figure 3.2 A typical leaf node entry

3.1.3 Structure of a signature node

The structure of a signature node is shown in Figure 3.3. The signature node for Ki has 2^{i-1} binary combinations denoting the possible prefixes. When a signature S_u with 1 in the i^{th} position is to be inserted the intermediate prefix formed so far is compared with the binary combinations in the signature node at Ki and u is inserted in the list.

B1	Signatures having prefix B1
B2	Signatures having prefix B2
...	...
Bn	Signatures having prefix Bn

Figure 3.3 A typical signature node entry

3.1.4 Overall structure of SD-tree

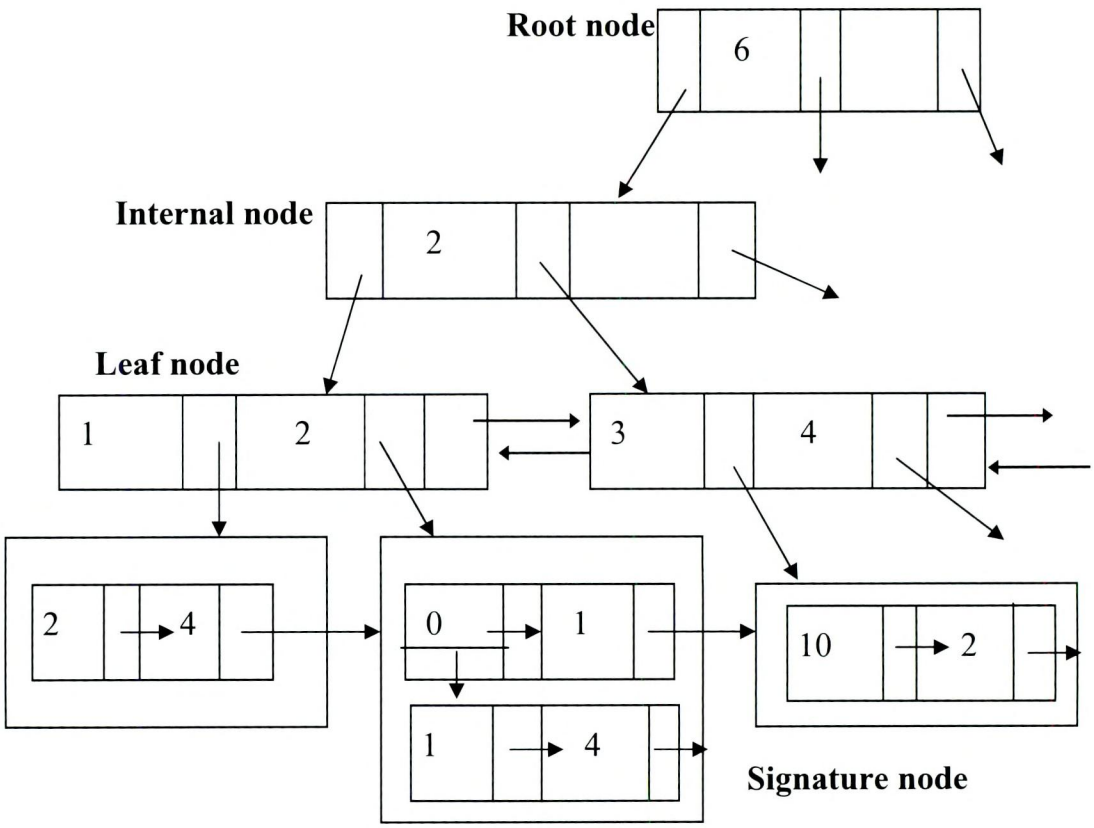

Figure 3.4 Sample SD-tree

Consider the SD-tree partially shown in Figure 3.4. The tree has been constructed for signature length F = 12. Consider the signature ***S1:0100 1001 0010***. To insert S1 with first occurrence of 1 at position 2, access the leaf node with value 2, follow its signature node and write the signature value as 1 (for S1) with the prefix 0 (for bit 1). The next 1 at position 5 is inserted in the signature node following the leaf node 5 with prefix 0100. The other bits are also inserted in a similar way. For ***S2:1010 1100 0000*** signature number 2 is inserted in signature node 1 with no prefix and signature node 3 with prefix 10 (for bits 1 and 2). In the same way it can be easily understood for signatures ***S3: 0001 1000 0011*** and ***S4: 1100 0100 0001*** too. While inserting a signature there are two possible options to move to next 1s position in the leaf node. One is via the sequential pointer between leaf nodes and the other is the top-down traversal of tree from root.

To ensure optimal search path the *threshold* (t) is fixed at h+1 (h being the tree height plus one more level for accessing signature nodes). As long as the number of sequential pointers traversed in leaf nodes is within the specified (t) value the sequential pointers are followed. This is calculated by finding the difference (d) between two consecutive 1s in a signature divided by the number of entries per leaf node. The condition here is $d / (p-1) \leq t$, where p is the order of the tree. The division here is integer division. When the above condition is not true, a new tree access is initiated from root. Similarly to promote optimality between queries the difference between the last set bits of two consecutive queries is compared with threshold and leaf node's forward / backward pointers are used accordingly.

3.2 Tree search and updates

This section lists the algorithms for signature insert, delete and search operations on SD-tree. A global flag F is set to 0 indicating the search path from the root of the tree by default. In the procedure after the first 1's insertion, depending on the d value F may be set to 1.

3.2.1 Insertion

The algorithm for signature insertion is outlined in Figure 3.5.

```
Insert (Su)
Input : The signature to insert Su;
1.  Let i1, i2, ….. in  be the positions of 1 in Su ;
     t = h+1;  F ← 0;  // for tree of order p
2. Move i2 to in to queue Q.  B = NULL;
3. If (i1  = 1) then begin
              Access leaf node i1;   // from root
              New(node);
              insert u;
              B = strcat ( B, '1'); // to denote bit 1 position
           end
        else begin
             for k = 1 to i1 - 1 do
             B = strcat ( B, '0');
             Access leaf node i1;   // from root
             New(node); write (B);
             insert u ;
             B = strcat ( B, '1');
           end
        f = i1; // store the current bit position
        F ← 1; // enable sequential search in leaf nodes
```

```
4. While Q not empty do
     begin
         read x from Q;
         if (x = f+1) then
         begin
            Access leaf node x;  // via leaf node pointers
            If (not(B)) then  New(node); // create node
            Write(B);
            Write u @ prefix B;
         end
      else begin
             d = x - f;
             If (d /(p-1)) > t then  F ← 0;
             for k = f +1 to x-1 do
               B = strcat (B, '0');
             Access leaf node x;
             If (not (B)) then New (node);
             Write (B);
             Write u @ prefix B
           end
           B = strcat (B, '1');
           f = x;
      end.  // until queue is empty
```

Figure 3.5 Insertion algorithm

The insertion algorithm reads the positions of set bits in the given signature to insert and move them to a queue. Every set bit position is read from queue one at a time and inserted in the corresponding signature node. The algorithm records the binary prefix as bits are inserted. Every time before insertion in the signature node the binary prefix is checked for existence. If so, the bit position is inserted; else new node with the prefix formed so far is created and then bit position is inserted.

Optimality is ensured between two successive insertions by checking the distance of two set bits. If this distance is less than or equal to the height of the tree, then sequential pointers of leaf nodes are followed for further insertion; else tree traversal is initiated from root.

3.2.2 Searching

Figure 3.6 outlines the steps to search for signatures matching a given query signature S_q. In the procedure F ← 0 always and the algorithm lands up directly in the signature node corresponding to last 1 from root.

```
Search(Sq)
1.    Input  : The (query) signature to search.
2.    Output : The list of signatures matching the given signature.
      Let i1, i2, … in  be the positions of 1 in Sq ;
      F ← 0;  // for access from root
      Move i2 to in-1 to queue Q.  B = NULL;
3.    If (i1 = 1) then B = strcat ( B, '1');
      else begin
            for  k = 1 to i1 - 1 do
            B = strcat (B, '0');
            B = strcat (B, '1');
           end;
      f = i1;    // Store the current bit position
4.    While Q not empty do
      begin
          read x from Q;
          If (x = f+1) then B = strcat (B, '1')
```

```
else begin
                for k = f+1 to x-1
                B = strcat (B, '0');
                B = strcat (B, '1');
               end;
       end.  // until Q is empty
5.     Access leaf node i_n.  // from root node
       Search for B.
       If Found() then read and output the list of signatures.
```

Figure 3.6 Search algorithm

Searching for a signature in SD-tree is very simple. For a given query signature the algorithm forms the binary pattern until the last set bit. Then the signature node corresponding to the last set bit position is accessed, binary pattern is compared and the list following that is read and output. This way all the matching signatures are retrieved cumulatively in a single access.

3.2.3 Deletion

The algorithm to delete a signature from SD-tree is given in Figure 3.7. Deletion algorithm follows the reverse of insertion. That is the position of set bits are extracted besides forming the binary prefixes in each step. Then, for every set bit position the corresponding signature node is accessed, prefix pattern compared and the signature number is deleted from the list.

```
Delete (Su)
1.    Input : Su, the signature to delete.
      Let i1, i2, ….. in  be the positions of 1 in Su.
2.    For each ik (1 ≤ k ≤ n) form prefix B as in Insert (Su).
3.    Access the leaf node and follow the signature node;
4.    Access prefix B and search for u.
      If present, delete it.
5.    Repeat steps (2) through (5) for all iks.
```

Figure 3.7 Deletion algorithm

3.3 Signature tree and SD-tree

In this section we compare analytically the results of signature tree [CHEN 06] with that of the SD-tree. The observed results are listed in Table 3.1.

Table 3.1 Signature tree Vs SD-tree

Parameter	Signature tree	SD-tree	Inference
Time complexity	O(nF)	O(nm)	m < F; Faster insertion
Tree height	$O(\log_2 n)$	$O(\log p\ (F/(p-1)))$	p>2 ; Shorter tree
Search cost	$O(\lambda.\log_2 n)$	$O(\log_p F+a)$	F < n; Cost< Sig. tree
Space complexity	$n\log_2 F + 2\sum_{i=0 \text{ to } k} 2^i (i+1)$	O(F(a+f))	< Sig. tree when k>F

In Table 3.1,

n - Number of signatures in signature file

F – Length of signature

m - Number of set bits

p – Order of SD-tree

λ – Number of path traversed in query searching

a – Average no. of signatures / signature node

k – $\log_2 n$

f - Average no. of prefix values / signature node

The first column of Table 3.1 lists the parameters generally used in search and update algorithms of indexing structures as observed from [CHEN 06]. The second column is the same calculated for SD-tree. From Table 3.1 it is apparent that the time complexity of SD-tree is O(nm) as against the O(nF) for signature tree. This is due to the fact that in the SD-tree the concern is for only set bits which results in lesser time complexity.

Since the construction of SD-tree depends only on the length of the signature regardless of the data base size, the height of the tree is greatly reduced. Also, SD-tree is a B+ tree like structure which has lesser number of nodes compared to the signature tree which is a balanced binary tree structure. Further searching for a signature traverses multiple paths in signature tree whereas in SD-tree the process accesses directly the signature node of the last set bit. This reduces the search cost tremendously.

The space complexity of signature tree can be defined clearly because it is a balanced binary tree. But for SD-tree, though it is a B+ tree based structure the space complexity of the structure is increased by the presence of signature nodes. Hence the complexity of the whole structure depends on the average number of entries per signature node and the number of binary prefixes present in each node. Nevertheless when the number of bits used to represent the database size n exceeds the signature length (i.e.) k>F which is true normally for large databases, SD-tree shows better performance.

To infer, SD-tree shows potential improvement of various complexities and hence is an effective structure for signature-based applications.

3.4 Experimental results

To validate the proposed structure we implement SD-tree in Java and for every test run the tree is constructed statically before signature insertion. The parameters considered in the experiments' data sets are signature length (F), signature weight (m) and signature weight distribution (swd). The experiments were carried out in a standalone system with Intel Pentium IV processor. The input file size was fixed as 50000 objects.The main memory size is 512 MB and the hard disk capacity is 80 GB.

3.4.1 Time Complexity

The time taken by an algorithm expressed as a function of the size of a problem is called the *time complexity* [AHO 74]. Like in other signature applications we use the *response*

time as the performance measure [KOCBERBER 99]. It is observed that the time taken for tree construction when F = 10 and p = 3 is 0.002 seconds. To analyze the *query response time* the signature weight distribution was fixed as 100%, 70%, 50% and 30% for signature lengths 10 and 30. Furthermore, the weight of the signature was biased in upper byte (U), lower byte (L) or uniformly distributed (M) and time values noted. The observed readings are listed in Table 3.2 and graphs plotted are shown in Figure 3.8. It is clear from Figure 3.8 that the query search time is lesser than signature insertion time.

Table 3.2 Time complexity of SD-tree of order 3 (in* Sec x 10^{-9} *)

Weight	Length	Byte	Search Time	Insert Time
100%	10	L	37520	95847
		M	37520	95847
		U	37520	95847
70%		L	34881	53683
		M	28207	60922
		U	26307	86418
50%		L	36987	42189
		M	23221	48763
		U	24694	68955
30%		L	33220	26312
		M	21820	32008
		U	19120	49008
100%	30	L	53675	242452
		M	53675	242452
		U	53675	242452
70%		L	49553	152152
		M	38202	171544
		U	19027	204017
50%		L	46586	119986
		M	24752	135416
		U	36290	150807
30%		L	43286	80614
		M	17295	105488
		U	19512	124780

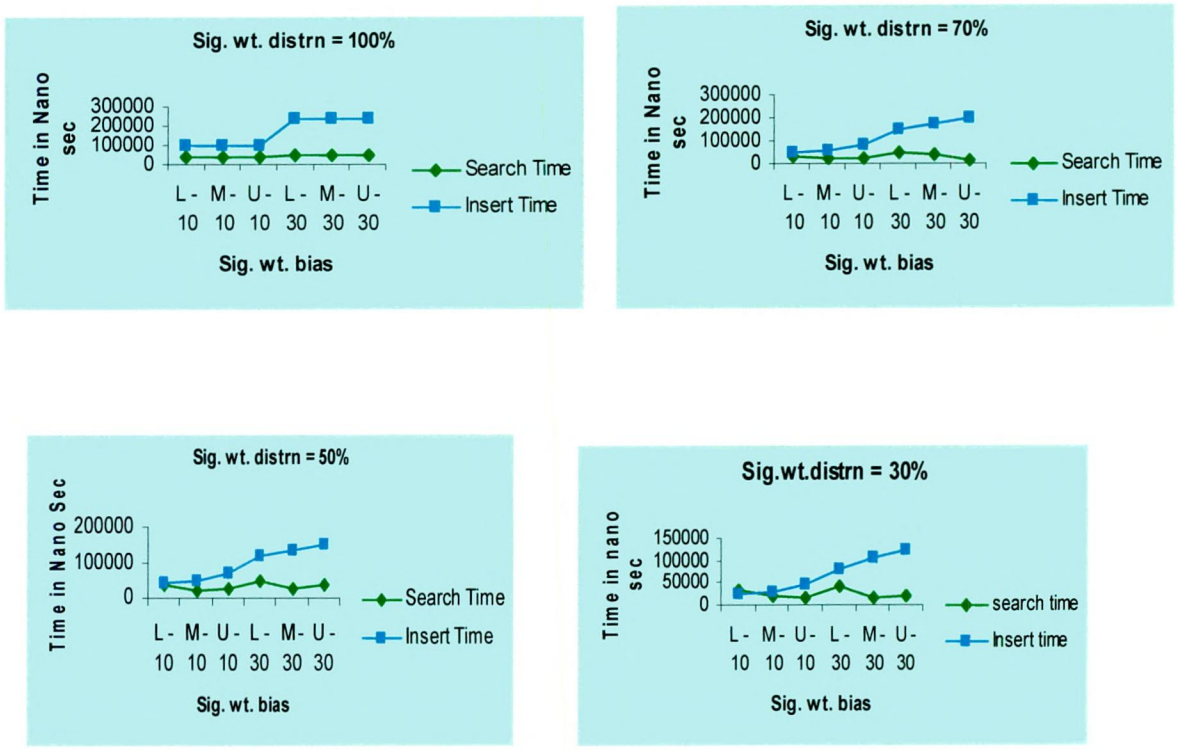

Figure 3.8 Time complexity

Table 3.3 Space overhead of SD-tree (in Bytes)

swd	F = 10	F = 20	F = 30
30%	72	144	216
50%	120	240	360
70%	168	336	1344

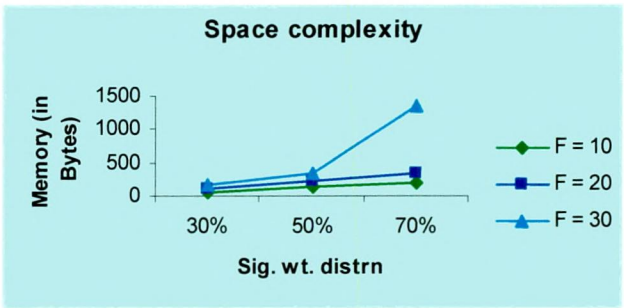

Figure 3.9 Space overhead

3.4.2 SD-tree maintenance and space overhead

SD-tree maintenance is quite simple that the operations are reflected only in the signature node. In the experiments the binary prefix pattern nodes are created dynamically. It has been noted that when signatures' weight is biased in upper bytes, the space complexity of SD-tree increases, due to the structure complexity in upper levels. Nevertheless, for a given file size the space complexity is reduced compared with signature tree when k>F which is true for large files and maintenance cost as well is very low in SD-tree. Figure 3.9 shows the space overhead for different values of signature weight distribution and signature length.

Inference: SD-tree uses B+ tree like structure in which the positions of 1s in the signatures are distributed over a set of signature nodes. Using this for a given query signature all the matching signatures can be retrieved cumulatively in a single node. Also for signature insertion and query searching an optimal search path is calculated so that the entire process is speeded up. The readings noted indicate that the time taken for signature insertion grows linearly with the values of signature weight distribution, signature length and the weight bias. Nevertheless the query response time is lesser than signature insertion time.

3.5 A sample validation model

This section discusses the evaluation of sample queries on a hypothetical object base. Figure 3.10 shows the assumed class diagram in UML notation [BOOCH 03], [FOWLER 03]. The classes and their relationships are listed in Table 3.4.

Table 3.4 Class relationships

S. No	Class 1	Class 2	Type of relationship
1.	University	Dept	Composition
2.	University	Student	Aggregation
3.	Student	Programme	Association
4.	Dept	Instructor	Association
5.	Student	Male	Generalization
6.	Student	Female	Generalization
7.	Programme	Subject	Association

The sample hashing outputs of attributes values are listed in Table 3.5.

Table 3.5 Sample attributes

Dept-name		Pgm-name
Mathematics	– 1010 0000	M.E – 0001 0100
Computer science	– 0100 1000	M.Sc (S.E) – 1000 1000
Physics	– 0001 0010	M.Sc (Mat) – 0100 1000
		M. Sc (Phy) – 0011 0000

Inst-name		Stud-name		Male
John	– 1000 0010	David	– 0100 0010	Sex – 1001 0000
Adams	– 1000 1000	Elena	– 1001 0000	Female
James	– 0000 1001	Maria	– 0100 0001	Sex – 0000 1001
Janes	– 0110 0000	Peter	– 0010 0001	
		Grace	– 0000 1100	
		Antony	– 0000 0011	

The attributes' hash coded values are superimposed to produce object's signature. The set of all object signatures of a class corresponds to the signature file. The signature files created for various attribute combinations of objects are listed in Table 3.6.

Table 3.6 Signature files of classes

Class : Course

Course-name

1. Software engg – 0100 0001
2. Comp. applns – 0010 1000
3. Comp. engg – 0100 0100
4. Applied Maths – 1000 0001
5. Calculus – 0010 0100
6. Nuclear physics – 0101 0000

Class : Male

Stud-name	Pgm-name	Sex	Object signature
1. David	M.Sc (S.E)	Male	1101 1010
2. Peter	M.E	Male	1011 0101
3. Antony	M.E	Male	1001 0111

Class : Female

Stud-name	Pgm-name	Sex	Object signature
1. Elena	M.Sc (S.E)	Female	1001 1001
2. Maria	M.Sc (S.E)	Female	1100 1001
3. Grace	M.Sc (Phy)	Female	0011 1101

Class : Dept

Dept-name	Programs offered	Object signature
1. Computer science	M.E,M.Sc(S.E)	1101 1100
2. Mathematics	M.Sc (Mat)	1110 1000
3. Physics	M.Sc(Phy)	0011 0010

Class : Programme

Pgm-name	Courses	Object signature
1. M.E	Comp.applns, Applied Maths	1011 1101
2.M.Sc (S.E)	Software engg, Comp. engg	1100 1101
3.M.Sc(Phy)	Comp.applns, Nuclear physics	0111 1000

Class : Instructor

Inst-name	Dept-name	Courses	Object signature
1. John	Comp.sc	Software engg, Comp. applns	1110 1011
2.Adams	Mathematics	Applied Maths, calculus	1010 1101
3. James	Comp.sc	Software engg	0100 1001
4. Janes	Physics	Nuclear physics	0111 0010

Class : Student

Stud-name	Pgm-name	Object signature
1. David	M.Sc(S.E)	1100 1010
2. Peter	M.E	0011 0101
3. Antony	M.E	0001 0111
4. Elena	M.Sc(S.E)	1001 1000
5. Maria	M.Sc(S.E)	1100 1001
6. Grace	M.Sc(Phy)	0011 1100

A list of sample queries on the above database and their evaluation is given in Table 3.7. For each query posed the query signature Sq is formed first and the class on which to execute the query is identified. Then the query signature comparison (using SD-tree) on the signature file of the concerned class is initiated which outputs all matching signatures finally.

For example, query 1 in Table 3.7 is to retrieve all students doing M.Sc (S.E). The query signature Sq is the signature of the attribute M.Sc (S.E) which is 1000 1000. This query is to be run on class "Student". Now the search is initiated on the SD-tree of Student class. The last set bit of Sq is at position 5. Hence the search process lands up in signature node corresponds to the leaf node 5 and searches for prefix binary pattern. In Student class the last set bit at position 5 matches with signatures 1, 4, 5 and 6 whereas the prefix matches with only 1, 4 and 5. The corresponding records namely David, Elena and Maria are the matching outputs.

Table 3.7 Sample queries

Criterion	Class accessed	Query signature (Sq)	Matching signatures
Query 1 : List of students doing a given programme			
Pgm-name=M.Sc(S.E)	Student	10001000	1100 1010 (David) 1001 1000 (Elena) 1100 1001 (Maria)
Query 2 : Instructors handling a given course			
M.Sc(S.E) in Courses	Instructor	01000001	1110 1011 (John) 0100 1001 (James)
Query 3 : Dept(s) offering a given course			
1. Comp. applns in Courses 2. Pgm-name = 00010100, 00110000	Programme Dept	00101000	1011 1101 (M.E) 0111 1000 (M.ScPhy) 1101 1100 (Comp.sc) 0011 0010 (Phy)
Query 4 : All instructors of a given dept			
Deptname = Comp.sc	Instructor	01001000	1110 1011 (John) 0100 1001 (James)
Query 5 : Female students attending a given programme			
Pgm-name = M.Sc(S.E)	Female	10001000	1001 1001 (Elena) 1100 1001 (Maria)

The other queries listed in the table can also be interpreted in a similar way. Query 3 needs special explanation. To find the departments offering a given course the query execution is carried out in two steps. This is because the attribute Course-name is a set-valued attribute of Programme class which is again a set-valued attribute within Dept class. Hence, the class Programme is accessed first and searched for signature of Course-name = comp.applns which is 0010 1000 and the output programmes are M.E and M.Sc (Phy). The attribute signatures of thses two programmes are the input for the second step which retrieves the matching departments Comp. sc and Physics from class Dept.

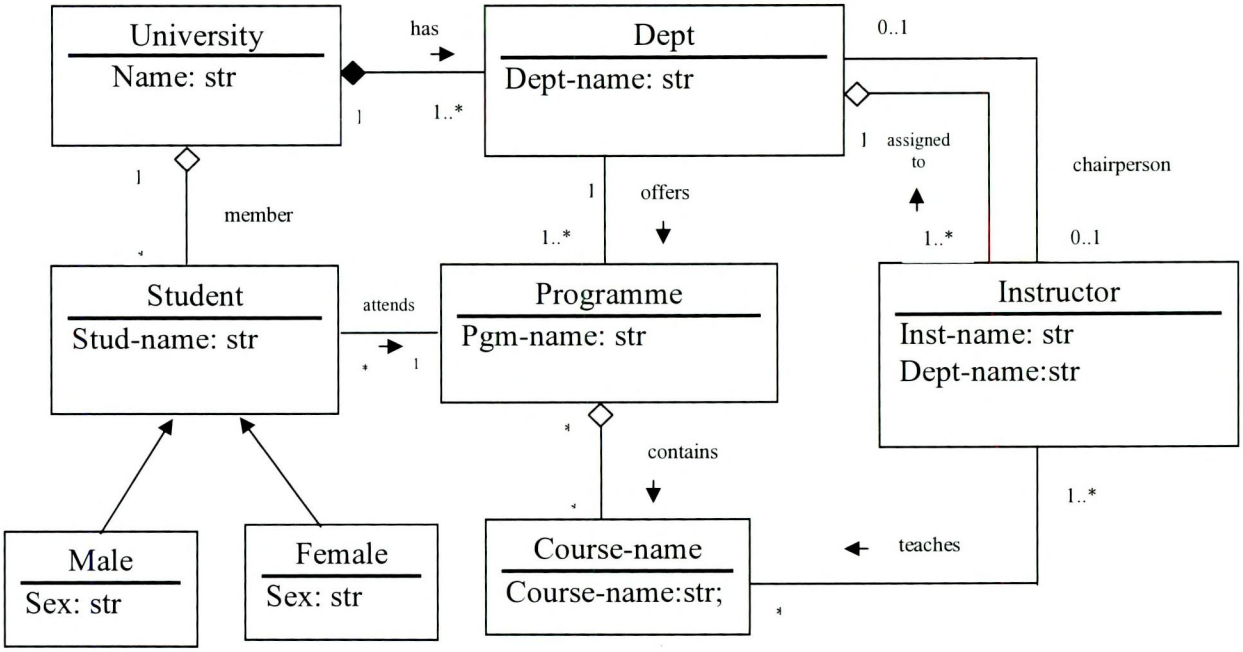

Figure 3.10 Sample object schema

This chapter presents a novel way to represent signatures in a B+ tree like structure called SD-tree. Using this structure for a given query signature all the matching signatures can be retrieved cumulatively in a single node. Also for signature insertion and query searching an optimal search path is calculated to further speed up the process. The structure has been analyzed for query response time by varying the signature length and distribution of 1s in the signature and results plotted. The outputs show that considerable search time is saved. The space overhead in SD-tree may be higher due to the presence of binary prefixes in higher order signature nodes, but the flexibility provided by the SD-tree outweighs all besides simple maintenance and faster query retrieval time.