CHAPTER 2

Chapter 2

Signature based information retrieval

The advent of internet has made the topic of information retrieval gaining more interest nowadays. As the volume of data escalates in all computer-based applications everyday, extracting useful data from such huge data repository becomes mandatory. There are many tools used to retrieve data from large databases in which indexing plays a vital role. Among the many indexing techniques reported in the literature the signature file approach is preferred for its efficient evaluation of set-oriented queries and easy handling of insert and update operations.

2.1 Overview of text retrieval methods

Many text retrieval methods have been proposed in the literature. However, they fall in one of the following large classes [FALOUTSOS 85]:

i) Full text scanning: Given a search pattern, the whole data base is scanned until the qualifying documents are discovered and returned to the user. The method requires no space overhead and minimal effort on insertions and updates. The main disadvantage is the retrieval speed and so scanning of a large data base may take too much time.

ii) Inversion: This method uses an index. An entry of this index consists of a word along with a list of pointers. These pointers point to documents that contain this word. Many commercial systems like ORBIT, LEXIS have adopted this approach. The main advantage of this method is its retrieval speed. However, it may require large storage overhead for the index. Moreover, insertions of new documents require expensive updates of the index.

iii) Signature files: The documents are stored sequentially in the “text file”. Their abstractions are stored sequentially in the “signature file”. When a query arrives, the signature file is scanned sequentially and a large number of non-qualifying documents are discarded. The method is faster than full text scanning because the size of the signature file is much smaller. It requires much smaller space overhead than inversion. The signature file method can handle queries on parts of words and can tolerate errors.

iv) Clustering: In this method, similar documents are grouped together to form clusters. Usually, they are stored physically together, too. Clustering has been extensively examined in the literature of library science and information retrieval. The space overhead and the retrieval time are rather small. However, it can not handle insertions easily.

v) Multi-attribute hashing: This method is applicable on bibliographic data bases. All the keywords of a title are hashed to yield a signature for the title. A sophisticated one-to-one function transforms this signature into an address of the hash table. The interesting property of this method is that the number of buckets to be searched decreases

exponentially with the number of search terms in the query. The hashing functions and their characteristics have been discussed by [CARTER 77] and [DEUTSCHER 75].

2.2 Signatures – A rational option

In order to choose the best access method we have to consider the operational characteristics of the specific environment. The main features are:

- Large data bases
- Large insertion rates, deletions and updates

Under the above considerations, the signature file method seems to be a reasonable choice [FALOUTSOS 85]. Signatures are hash coded abstractions of the original data. It is a binary pattern of predefined length with fixed number of 1s. The attributes' signatures are superimposed to form object's signature. Initially applied on text data as discussed in [FALOUTSOS 84], [FALOUTSOS 85a], [LEE 95], [ZOBEL 98] and [LARSON 84] it has now been used in other applications like office filing [FALOUTSOS 87a], relational and OODBs [TOUSIDOU 02], [YONG 94], [LEE 92] and hypertext [FALOUTSOS 90]. A signature is a bit string formed from a given value. Compared to other index structures, signature file is more efficient in handling new insertions and queries on parts of words. Other advantages include its simple implementation and the ability to support a growing file. But it introduces information loss which can be minimized by carefully selecting the signature extraction method.

A commonly used benchmark for the comparison of indexing schemes is the inverted file method. Use of inversion in information retrieval is analyzed and presented by Stellhorn in his paper [STELLHORN 77]. [KENT 90] has reported in the comparative analysis of inversion and signature file method that the inverted file method is efficient for single term queries only and is expensive for multi-term queries. Inverted file method storage is of expensive. On the contrary, signature file methods provide direct support, at low cost, for the indexing of word parts and word phrases. Lee et al. [LEE 92] deduced that signature files have much lower storage overhead and a simpler file structure than inverted indexes. They are particularly good for multi-attribute retrieval when attributes have equal chance of being specified in the query. Since queries in OODBSs could be very flexible, an inverted index would be very complex if all possible access paths are supported. However, the signature file approach offers a much simpler solution at the expense of some retrieval speed.

According to [FALOUTSOS 87a] another advantage of signature files over inverted files is the ease of insertions. For each new message, its signature is created and appended at the end of the signature file. In comparison, inverted files require rewriting of parts of the index. Also signature files can be applied on optical disks, which are ideal for archiving. Optical disks can be written upon only a limited number of times. This is a problem for the inverted file method, because the index has to be rewritten on every insertion or batch of insertions. Signature files create no problems because the signatures of the new messages can be appended easily at the end of the signature file. However it is expected to be slower than inversion for large files.

2.2.1 Signature extraction methods

Work on indexing signature files can be generally divided into two categories: the *exhaustive approach* and a more *restrained* one. In order to achieve a good performance in the first case, researchers had focused in developing good signature extraction methods for the reduction of false-drop probability and the improvement of the retrieval performance in signature files as indicated in [FALOUTSOS 84] and [DERVOS 98]. Techniques for signature extraction such as Word Signature, Superimposed Coding, Run Length Encoding, Bit-block Compression and Variable Bit-block Compression have been studied by FALOUTSOS, DERVOS and ROBERTS in their papers during 80's and 90's. In general, the encoding scheme sets a constant number say m, of 1s in the range [1..F], where F is the length of the signature. The resulting binary pattern with m number of 1s and (F-m) number of 0s is called a word signature. These methods are discussed below.

i) Word Signature: In this method each word of the document is hashed into a bit pattern of length f. These patterns ("word signatures") are concatenated to form the document signature. In Figure 2.1 the document consists of four words. Each word yields a 4-bit word-signature (f =4). Searching is performed in the obvious way. For example, on a single word query the signature of the search word is extracted and all the document signatures are searched. Those that contain the signature of the search word are retrieved. In order to improve the performance, common words (e.g., "the", "a", etc.) may be ignored.

Document	free	text	retrieval	methods
	↓	↓	↓	↓
Word sign.	0000	0100	0111	1011
Document signature	0000	0100	0111	1011

Figure 2.1 Word Signature method

ii) Superimposed coding: The second method [CHRISTO 84] is based on superimposed coding. The method works as follows. Each document is divided into "logical blocks". A logical block is defined as a piece of text that contains a constant number D of distinct, non-common words. Each such word yields a bit pattern of size F. These bit patterns are OR-ed together to form the block signature. The concatenation of the block signatures of a document forms the document signature.

The signature creation of a word in superimposed coding is rather sophisticated. Each word yields m bit positions (not necessarily distinct) in the range 1-F. The corresponding bits are set to "l", while all the other bits are set to "0". In Figure 2.2, It is assumed that each logical block consists of D = 2 words only. The signature size F is 12 bits and m = 4 bits per word. For example, the word "free" sets to "1" the 3-rd, 7-th, 8-th, and 11-th bits (m = 4 bits per word). In order to allow searching for parts of words, the following method is suggested: Each word is divided into successive, overlapping triplets (e.g., "fr", "fre", ree", "ee" for the word "free"). Each such triplet is hashed to a bit position by applying a hashing function on a numerical encoding of the triplet. In case that a word has I triplets, with I>m, the word is allowed to set 1 (non-distinct) bits. If I <m, the additional bits are set using a random number generator, initialized with a numerical encoding of the word.

Word	Signature
free	001 000 110 010
text	000 010 101 001
block signature	001 010 111 011

Figure 2.2 Superimposed coding method

Searching for a word is handled as follows: The signature of the word is created. Suppose that the signature contains "1" in positions 2, 3, 6, and 9. Each block signature is examined. If the above bit positions (i.e, 2, 3, 6, and 9) of the block signature contain "l", then the block is retrieved. Otherwise, it is discarded. More complicated boolean queries can be handled easily. In fact, conjunctive (AND) queries result in a smaller number of false drops. Even sequencing of words can be handled. The signature size F affects directly the number of false drops. Superimposed coding applied for multi key access method has been discussed by [DAVIS 87] et al.

iii) Compression-based methods: The next method is based on compression. In this the document is split into logical blocks, as in Superimposed Coding. The idea is that we use a (large) bit vector of B bits and we hash each word into one (or perhaps more, say n) bit position(s), which are set to "1" as in Figure 2.3. The resulting bit vector will be sparse and therefore it can be compressed.

Word	Signature
free	0000 0000 0000 0010 0000
text	0000 0001 0000 0000 0000
retrieval	0000 1000 0000 0000 0000
methods	0000 0000 0000 0000 1000
Block signature	0000 1001 0000 0010 1000

Figure 2.3 Compression-based method

There are many variations of the signature generation method based on compression. Commonly used techniques are given below:

a) Bit-Block-compression: In this method the sparse vector is divided into groups of consecutive bits (bit-blocks). For each bit-block we create a signature, which is of variable length and consists of at most three parts as in Figure 2.4 where the bit-block size b = 4.

Part I) It is one bit long and it indicates whether there are any "1"s in the bit-block (1) or the bit-block is empty (0). In the latter case the bit-block signature stops here.

Part II) It indicates the number s of "1"s in the bit-block. It consists of s -1 "1"s and a terminating zero.

Part III) It contains the offsets of the "1"s from the beginning of the bit-block.

	b ↔				
Sparse vector	0000	1001	0000	0010	1000
Part I	0	1	0	1	1
Part II		10		0	0
Part III		00 11		10	00

Figure 2.4 Bit-Block compression method

0 | 1 10 00 11 | 0| 1 0 10 | 1 0 00

Figure 2.5 Signature storage by concatenation

0 1 0 1 1 | 10 0 0 | 00 11 10 00

Figure 2.6 Signature storage by parts concatenation

Figure 2.5 and Figure 2.6 illustrate two different ways of storing the block signature of Figure 2.4. In the first alternative (Figure 2.5), the parts of each bit block signature are stored consecutively, and the bit-block signatures are concatenated. The vertical lines mark the bit-block signature boundaries. In the second alternative (Figure 2.6), the first parts of all the bit-block signatures are stored consecutively, and then the second parts, and so forth. The latter alternative allows faster searching [FALOUTSOS 85], [FALOUTSOS 87a].

b) Run-Length encoding: In this method the number of zeros between two successive "1"s in the sparse vector is recorded. The advantage of the method is the excellent compression. The disadvantage of the Run Length encoding method is the slow searching. In order to determine whether a bit is a "1" in the sparse vector, the encoded lengths of all the preceding intervals have to be decoded and summed. Moreover, in order to skip to the next signature, the total length of the current signature has to be determined. This can not be done, unless the entire signature is scanned or unless pointers are used. In contrast, the length of a signature in the Bit-Block Compression method can be determined from Parts I and II.

c) Variable Bit-Block compression: Another important consideration is that the Bit-Block Compression method can be slightly modified to become insensitive to changes in the number of words D per block. This is desirable because the need to split documents in logical blocks is eliminated, thus making the resolution of complex queries much easier. There is no need to "remember" whether some of the terms of query have appeared in one of the previous logical blocks of the message under inspection [FALOUTSOS 87a]. This method is referred to as Variable Bit-Block Compression.

Many researchers have compared the signature extraction methods listed above from a practical point of view by considering additionally the following:

i) Speed to search a block signature.

ii) Performance on complicated queries.

iii) Ability to answer queries on parts of words.

iv) Preservation of the information sequencing.

The observations inferred are as follows:

- The fastest method for searching a signature seems to be superimposed coding [FALOUTSOS 85].
- Only superimposed coding can handle queries on parts of words [FALOUTSOS 85].
- Superimposed coding is perhaps the most common method used [LEE 95].
- Signature files typically use superimposed coding to create the signature of a document [FALOUTSOS 88].
- Superimposed coding requires a few simple bit operations on searching and it eliminates duplicates automatically [FALOUTSOS 84].

2.2.2 Applications of signatures

Signatures are applied in database access methods useful for text retrieval such as full text scanning, inversion and clustering, multi attribute retrieval methods like hashing and signature files. Such applications are discussed in [FALOUTSOS 85a], [FALOUTSOS 85b], [LARSON 84]. Here the documents are stored sequentially in the "Text File".

Signatures which are abstractions of the documents are stored in the "Signature File". The latter serves as a filter on retrieval. It helps in discarding a large number of non-qualifying documents. Signatures have been applied in areas rich in text documents like telephone directory [ROBERTS 79], office systems [FALOUTSOS 85b], [FALOUTSOS 85], [FALOUTSOS 87a], Optical and magnetic disk access [FALOUTSOS 88], data base management system and library automation. Other applications include indexing method for large text file [ZOBEL 98], [KENT 90], access method for formatted data [ROBERTS 79], to speed up searching in editor, to compress a vocabulary, for a spelling checking program and in differential file [FALOUTSOS 87].

2.2.3 Physical representations of signature files

Different approaches have been discussed by researchers to represent signature file in a way conducive for evaluating queries which are listed below. This section follows the lead of [CHEN 06] for figures.

i) Sequential Signature File (SSF)

The signatures are sequentially stored in a file as in Figure 2.7. In single level every signature must be accessed and tested individually. Since signatures are abstractions of original data with smaller size, the method is faster than sequential scan of objects themselves. This method is easy to implement and requires low storage space and low update cost. The disadvantage is that more the number of objects exist, the more is the time spent on scanning signature file [FALOUTSOS 88], [ISHIWAKA 93]. Therefore it is generally slow in retrieval. To support faster access multilevel signature methods are suggested.

Signature file	OIDs
1 0 1 0 1 0 0 1	O_1
0 1 1 0 0 0 1 1	O_2
0 0 1 0 1 1 0 1	O_3
1 1 1 0 1 0 0 0	O_4
0 0 1 1 1 0 0 1	O_5
1 1 1 0 0 0 1 0	O_6
0 1 0 1 0 0 1 1	O_7
0 1 0 1 0 1 1 0	O_8

Figure 2.7 A typical sequential signature file

ii) Bit-Sliced Signature File (BSSF)

A signature file can be stored in a column-wise manner. That is, the signatures in the file are vertically stored in a set of files [ISHIWAKA 93]. Concretely, if the length of the signatures is F, then all the signatures will be stored in F files, in each of which one bit per signature for all the signatures is stored as shown in Figure 2.8.

With such a data structure, the signatures are checked slice-by-slice (rather than signature-by-signature) to find matching signatures. To demonstrate the retrieval process, consider a query signature s_q = 10110000. First, we check the first bit-slice file shown in Figure 2.8 and find that only three positions: first, fourth and sixth positions match the first bit in s_q. Then, we check the second bit-slice file. This time, however, only those three positions in the second file will be checked. Since the second bit in s_q is 0, no positions will be filtered. Next, we check the third bit-slice file against the third bit in s_q. Since all the three positions in it are set to 1, the same positions in a next bit-slice file, i.e., in the fourth bit-slice file will be checked against fourth bit in s_q. Since none of the three positions in the fourth bit-slice file matches this bit in the query, the search stops and reports nil. From this process, we can see that only part of the m bit slice files have to

be scanned. Therefore, the search cost is lower than that of a sequential file. However, update cost becomes larger. For example, an insertion of a new signature requires about m disk accesses, one for each bit slice file.

Bit-slice files								OIDs
1	0	1	0	1	0	0	1	O_1
0	1	1	0	0	0	1	1	O_2
0	0	1	0	1	1	0	1	O_3
1	1	1	0	1	0	0	0	O_4
0	0	1	1	1	0	0	1	O_5
1	1	1	0	0	0	1	0	O_6
0	1	0	1	0	0	1	1	O_7
0	1	0	1	0	1	1	0	O_8

Figure 2.8 A typical bit-sliced signature file

A more efficient approach based on the prevention of reading unnecessary signature bits was introduced in [ISHIWAKA 93], where several variations of the bit-sliced signature file are presented. [GRANDI 92] et al. has discussed another variation of the partitioned signature file method suiting parallel computing architectures. For queries that result in high weights, a method of organizing signatures based on partitioning was proposed by [ZEZULA 91]. The author further reports a new way of organizing signature files combining key-based and bit-sliced partitioning strategies in [ZEZULA 95]. Other variations of this signature file method that support database processing for multimedia data have been presented in [KIM 95] and [RABITTI 89].

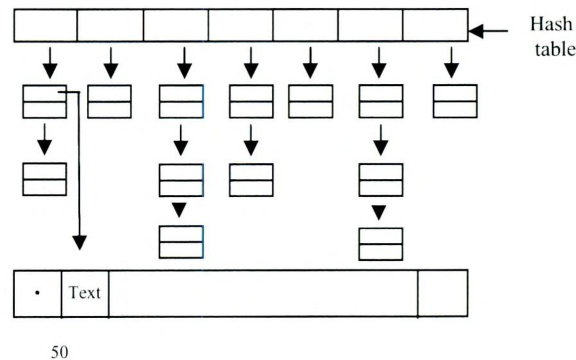


Figure 2.9 A typical CBSSF organization

iii) Compressed Bit-Sliced Signature File (CBSSF)

A bit-slice file can also be compressed. According to [KOCBERBER 99], if a proper hash function is chosen, which maps each keyword to a signature with only one bit set to 1 (i.e., k = 1), a signature file will contain much less 1s than 0s. In this case, a signature file can be considered as a sparse matrix, and can be effectively compressed. A simple way to compress a bit-slice file is to replace each 1 with an address where the corresponding keyword can be found. All the addresses in a slice are then stored in a collection of buckets that are linked together.

In Figure 2.9, an array is used, in which each entry is a pointer to the head of a linked list of buckets. Each of them contains several addresses. Assume that we have a word "Text" that hashes to a signature with the first bit set to 1 (i.e., hash ("Text") = 1), and that it appears in the document starting at the 50th byte of the text file. Then, searching the first linked bucket list, we will find the position where the word "Text" appears. Space optimization is achieved at the cost of query evaluation time since the probability of false drops will be definitely increased in the case of sparse signature files.

According to [CHRISTO 84], when a signature file is half-populated with 1s and half-populated with 0s, we have the lowest false drop probability.

iv) S-Tree

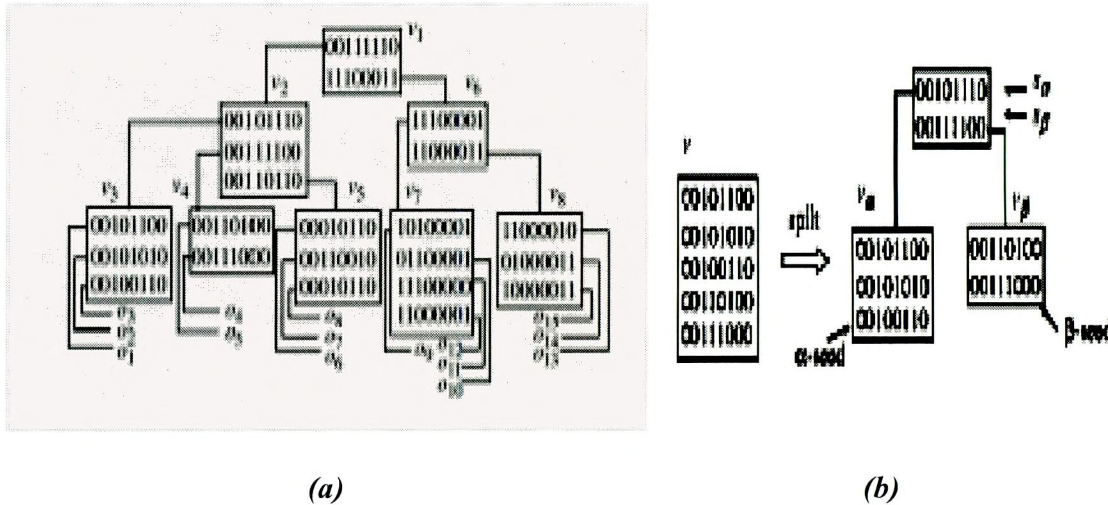
S-Tree is a height balanced multi way tree [DEPPISCH 86] similar to B+-tree. Each internal node corresponds to a page, which contains a set of signatures and each leaf node contains a set of entries of the form < s, oid >, where oid is the object identifier and s is the signature of the corresponding object. Let v be the parent node of v_0. Then, there exists a signature in v, whose value is obtained by superimposing all the signatures in v_0. See Figure 2.10 for illustration. To retrieve a query signature s_q = 00110000, we search the S-tree top-down. However, more than one path may be visited. For example, the first signature in the root v_1 of Figure 2.10(a) leads us to its child node v_2 because the third and fourth bits are set to 1. In v_2, the second and third signatures match s_q. Then, we go to the leaf node v_4 and v_5. In v_4, we find two matching candidates o_4 and o_5, and in v_5, we have only one o_7.

The construction of an S-tree is an insertion-splitting process. At the very beginning, the S-tree contains only an empty leaf node and signatures in a file are inserted into it one by one. When a leaf node v becomes full, it will be split into two nodes and at the same time, a parent node v_{parent} will be generated if it does not exist. In addition, two new signatures will be put in v_{parent}. Assume that the capacity of v is K (i.e., v can accommodate K signatures.) Then, when we try to insert the (K + 1)th signature into v, it has to be split into two nodes v_α and v_β. To do this, we will pick a signature which has the heaviest signature weight (i.e., with the most 1s) in v. It is called the α-seed and will be put in v_α. Then, we select a second signature, which has the maximum

number of 1s in those positions where α has 0. That is, the signature provides the maximal weight increase to α. This signature is called the β-seed and put in v_β. Any of the rest K - 1 signature is assigned to v_α or v_β, depending on whether it is closer to v_α or v_β. The two new signatures (denoted s_α and s_β) to be put into the parent node are obtained by superimposing the signatures in v_α and v_β, respectively. This is shown in Figure 2.10.

The advantage of this method is that the scanning of a whole signature file is replaced by searching several paths in S-tree. However, the space overhead is almost doubled. Furthermore, due to superimposing, the nodes near the root tend to have heavy weights and thus have low selectivity.

The S-tree has been further improved by Tousidou et al. They elaborated the selection of α-seeds and β-seeds so that their distance was increased. However, this kind of improvement is achieved at cost of time, i.e., by checking more signatures, which makes the insertion of a signature into an S-tree extremely inefficient. In [TOUSIDOU 00] a number of new split methods namely linear split, quadratic split, cubic split and hierarchical clustering for S-tree were proposed to improve query response time. In [TOUSIDOU 02] a new hybrid scheme combining linear hashing, S-tree and parametric weighted filter was used to evaluate subset-superset queries.



(a) *(b)*

Figure 2.10 S-tree organization and node splitting

v) Multilevel Signature file

In general, to increase the selectivity of a signature in an internal node, longer signatures should be used, or the page for a node should not be fully populated. Both of them require more space. The multilevel signature file method discussed in [LEE 95] follows the same principle as the S-tree. The difference between them is in the way that signatures at a higher level are constructed. In the multilevel signature file method, a signature at a higher level is a superimposed code generated directly from a group of p text blocks, instead of superimposing p signatures at the lower level. In other words, a signature at a higher level can be considered as being generated from a bigger block containing more words. Assume that any signature at the lowest level (leaf level) is created from a block of size D. Then, any signature at the second lowest level is generated from a block of size pD. For a better understanding, consider the S-tree shown in Figure 2.10(a) once again. Corresponding to this S-tree, we may have a multilevel signature file as shown in Figure 2.11. As in S-tree, any signature at the lowest level corresponds to an object (or a text block). But, any signature at a higher level is

constructed in a different way. We pay attention to the level L_1 shown in Figure 2.11, at which each signature is not generated by superimposing some signatures at the level L_0, but created directly from the text blocks.

For example, the first signature at L_1 is generated by taking o_1, o_2, and o_3 as a single block. So, it will have a different length than the signatures in L_0. Obviously, such a data structure needs more space than S-trees. However, the filtering power of any signature at a higher level is effectively increased. It is because the ratio of 1s over the signature length at a higher level is kept unchanged. Recall that in an S-tree, a signature in an internal node may be populated with more 1s than in a leaf node, decreasing its selectivity significantly.

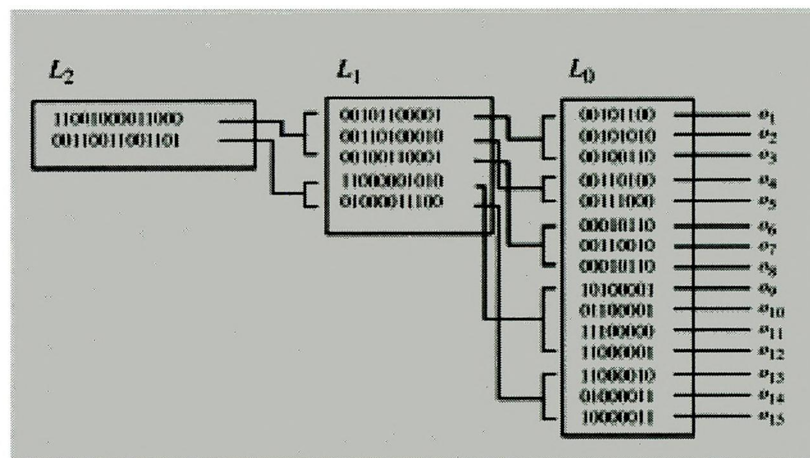


Figure 2.11 A typical Multilevel Signature file organization

vi) Signature Graph

The signature file is organized as a trie like structure [CHEN 04a], [CHEN 06]. However, the path visited in the graph to find a signature that matches a given query signature corresponds to a signature identifier which is not a continuous piece of bits,

differentiating the signature graph from trie. The signature graph for a given signature file $S = s_1, s_2, \ldots, s_n$ is constructed as follows.

At the very beginning, the tree contains an initial node: a node v with p (v) pointing to the first signature and skip (v) = 0. Then, we take the next signature to be inserted into the graph. Let s be the next signature we wish to enter. We traverse the graph from the root and each encountered node will be marked. Let v be a node encountered and assume that skip (v) = i. If v is not marked and i > 0, check s[i] and mark v. If s[i] = 0, we go left. Otherwise, we go right. If i = 0 or v is marked, we compare s with the signature s' pointed to by p (v). Signature s' can not be the same as s since in S there is no signature which is identical to anyone else. But several bits of s can be determined, which agree with s'. Assume that the first k bits of s agree with s'; but s differs from s' in the (k + 1) th position, where s has the digit b and s' has 1 - b. We construct a new node u with skip (u) = k + 1 and p (u) pointing to s. Let $w_1 \rightarrow w_2 \ldots \rightarrow w_j \rightarrow v$ be the accessed path. Then, make u the left child of w_j if v is a left child of w_j; otherwise, make u the right child of w_j. If b = 1, we make v be the left child of u and the right pointer of u pointing to itself. If b = 0, we make v be the right child of u and the left pointer of u pointing to itself.

Though in a signature graph, signatures are represented compactly, the search path is not same for all queries. In other words the graph is not balanced. In the worst case it degrades to a signature file. Figure 2.12 shows the signature file and the corresponding signature graph.

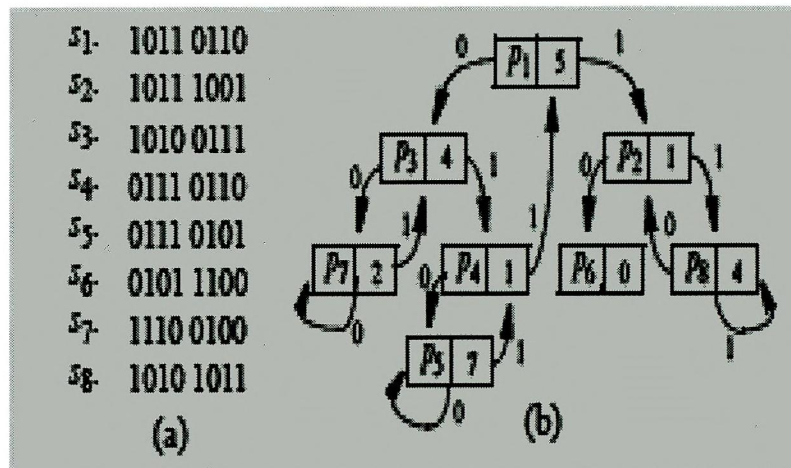


Figure 2.12 A typical signature file and signature graph

vii) Signature tree

A signature tree works for a signature file like a trie [KNUTH 73] for a text. But in a signature tree, each path is a signature identifier (defined below) which is not a continuous piece of bits, quite different from a trie, in which the bits (or characters) labeling a path are consecutive. In fact, the signature identifiers can be considered as a generalization of the concept of position identifiers [AHO 74], [DEPPISCH 86], extended to handle inexact matching. As mentioned above, by the inexact matching, we ask for matches at the 1 bit positions of a query and indifferent at the 0 bit positions. In comparison with the other representations for a signature file the signature tree has the following advantages:

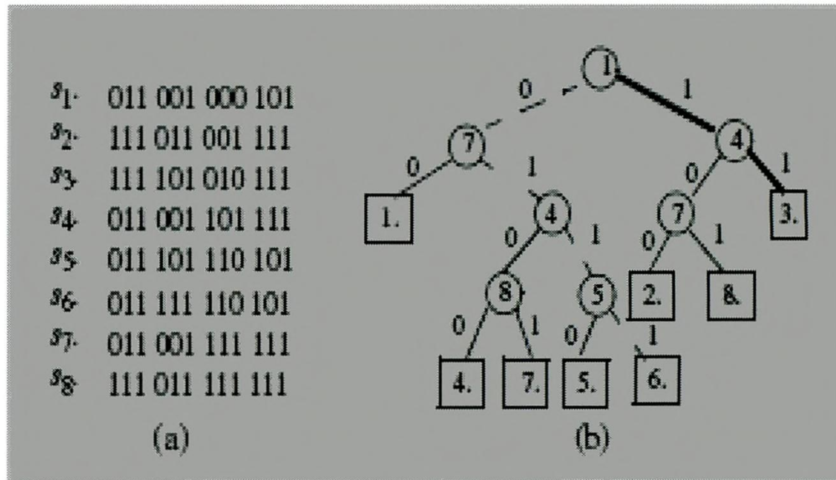
i) The slice checking in the bit-slice method is replaced with a single bit checking and less time is needed for both the insertion and deletion of signatures.

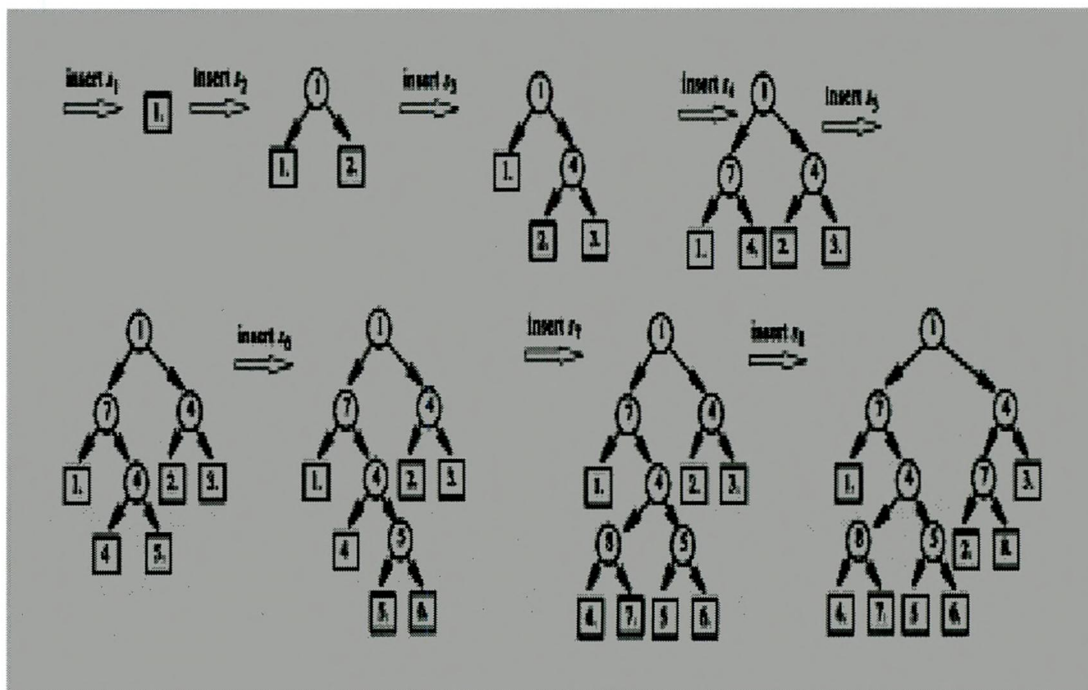
ii) The checking of signatures in an internal node of an S-tree is changed to a binary tree searching and much less space is needed for the tree structure.

The procedure to construct a signature tree for a given signature file S = $s_1, s_2, .. s_n$ is as follows. At the very beginning, the tree contains an initial node: a node containing a pointer to the first signature. Then, we take a next signature and insert it into the tree. Let s be the next signature we wish to enter. We traverse the tree from the root. Let v be the node encountered and assume that v is an internal node with sk (v) = i. Then, s[i] will be checked. If s[i] = 0, we go left. Otherwise, we go right. If v is a leaf node, we compare s with the signature s_0 pointed to by v. s cannot be the same as v since in S, there is no signature which is identical to anyone else. But, several bits of s can be determined, which agree with s_0. Assume that the first k bits of s agree with s_0; but, s differs from s_0 in the (k + 1) th position, where s has the digit b and s_0 has 1 - b. We construct a new node u with sk (u) = k + 1 and replace v with u. If b = 1, we make v and the pointer to s be the left and right child of u, respectively. If b = 0, we make v the right child of u and the pointer to s the left child of u.

Figure 2.13 (b) shows a signature tree for the signature file shown in Figure 2.13(a). In this signature tree, each edge is labeled with 0 or 1 and each leaf node is a pointer to a signature in the signature file. In addition, each internal node v is marked with an integer sk(v) to tell which bit will be checked. Consider the path going through the nodes marked 1, 7, and 4. If this path is searched for locating some signature s, then three bits of s : s[1], s[7],and s[4] must be checked. If s[4] = 1, the search will go to the right child of the node marked "4." This child node is marked with 5 and then the fifth bit of s : s[5] will be checked. Also the path consisting of the dashed edges in Figure 2.13(b) corresponds to the identifier of s6 : s6 (1,7,4,5) = (1,0) (7,1) (4,1) (5,1). Similarly, the identifier of s3 is s3 (1,4) = (1,1) (4,1) which is shown as the path consisting of thick

edges. Figure 2.13(b) traces the tree generation steps against the signature file shown in Figure 2.13(a).





(c)

Figure 2.13 A typical signature tree with sample trace of tree generation

2.3 Motivation for the work

The signature tree developed by Chen [CHEN 06] is having the following drawbacks:

- Signatures are inserted considering both 0s and 1s whereas actual weight age is for set bits only.
- Insertion path is dictated by the existing tree structure.
- To process a query, bits appearing in the tree from root node are compared with query signature pattern for 0s and 1s and not by its set bits.
- For a 0-bit in the query both left and right sub tree is followed that leads to multiple traversals.

These observations laid the foundation for the current work. We study a new indexing technique for OODBSs using the dynamic balancing of B+ tree called Signature Declustering (SD)-tree in which 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.

2.4 Signature-based object-oriented query processing

Advanced database application areas, such as computer aided design, office automation and multimedia data present the need to manipulate efficiently complex objects. Object-oriented databases can come to the aid of such areas that contain both formatted and unformatted, complex data by supplying set and tuple constructors and by making use of the existing index structures. Several experimental and commercial systems, such as Gemstone, Orion, and O2 have been developed. Many Benchmarks to represent

OODBMS' performance have also been reported in the literature. One such benchmark namely 007 has been analyzed by Carey et al. in [CAREY 93]. The rich data modeling capability of OODBSs have laid the path for analyzing different types of query handling like conjunctive queries, inclusive queries, nested queries and so on. For example the conjunctive queries on OODBSs have been studied by [CHAN 92]. Object query evaluation using calculus and their optimization has been reported by [FEGARAS 00] et al. and various query evaluation techniques have been analyzed by [GRAEFE 93] and [TANIAR 98].

During the last few years, several indices have been proposed in order to support the manipulation of object-oriented and object-relational data models [TOUSIDOU 02]. Indexing techniques defined in the framework of OODBMS can be classified as *structural* and *behavioral* [BERTINO 95]. *Structural indexing* techniques is based on object attributes and is very important because most object-oriented query languages allow query predicates to be issued against object attributes. Structural indexing techniques proposed so far can be classified as techniques providing support for nested predicates and indexing techniques supporting queries issued against an inheritance hierarchy. *Behavioral indexing* aims at providing efficient execution for queries containing method invocations. It is based on pre-computing or caching a method results and storing them into an index. However, the major problem in this approach is how to detect changes to objects that invalidate the results of a method. Kemper et al. [KEMPER 90] and Xie et al. [XIE 94] present structures (that follow structural indexing) manage to cope with path expressions, which are fundamental in OODBs, but they have not dealt with data that contain set-valued attributes. Due to this need, the use of signature files

was introduced. Set-valued attributes are now being represented by bit vectors, called signatures.

A signature is basically an abstraction of information stored in the attributes of an object. It is a superimposed bit string generated from the attribute values of the object being represented. Signatures are stored in a signature file to screen out the unqualified objects. According to [LEE 92], the following features of signatures support the flexible query handling in OODBSs.

- A signature failing to match the query signature guarantees that the corresponding object can be ignored.
- Signature files have lower storage overhead and simpler file structure.

Another direction of signature-based OODB indexing was the use of tree-based signature file organizations such as the two-level and multi-level index, and the most popular one is the S-tree [DEPPISCH 86]. S-trees are height balanced dynamic structures similar to B+-trees that have been proposed to improve the searching performance in signature based organizations. An improvement of the S-tree construction is presented in [TOUSIDOU 00] in order to achieve a better clustering of the stored signatures. Combining signature files with pat trees to support queries in document databases is the report of the study by [CHEN 99] et al. Recently, signatures have also been used in order to perform join operations between two relations containing set-valued attributes as in [RAMASAMY 00].

Common ground in most of the signature-based methods is that they were originally presented for use in text databases. Only lately do authors refer to the particular properties, needs and problems that are met in OODBs, such as the constrained domain of set-valued attributes in contrast to the non constrained one of text-databases, and the superset, perfect match and subset queries that is frequently asked.

2.4.1 Queries on set-valued attributes

OODBS need efficient support for manipulation of complex objects. In particular, support of queries involving evaluation of set predicates is often required in handling complex objects. Many schemes to apply signature file techniques to the support of set value accesses, and quantitatively evaluate their potential capabilities have been studied. Further, a significant amount of today's stored data consists of records with set-valued attributes. Such attributes are extensively used in OODBSs to represent an object's multi-valued attribute. The overall goal of OODB research is to provide efficient complex object management facilities. OODBMSs offer data modeling constructs such as the set constructor and the tuple constructor, so that the user can specify structures of complex objects. In order to support complex objects efficiently, it is necessary to investigate methods to accelerate set value manipulation in query processing.

Signatures are used to indicate the presence of individuals in sets. For example, in an object-oriented database they would be used to represent a set-valued attribute of an object [TOUSIDOU 02]. In general for two sets x and y checking whether $x = y$ is performed in two steps. First we check if $sig(x) = sig(y)$, and only if this holds we compare the actual sets. The same holds for the subset predicate $(x \subseteq y) \Rightarrow (sig(x) \subseteq$

sig(y)). The signatures are usually much smaller than the set instances and binary operations on them are very cheap, thus the two-step query processing can save many computations.

An inclusion (or subset) query searches for all objects containing certain attributes. Given an inclusion query with argument O' its query signature q is obtained by using the same methodology. The answer to the inclusion query is the collection of all objects O for which $O' \subseteq O$. If s is the signature of an object O, then it is true that

$$O' \subseteq O \Rightarrow q \subseteq s$$

where the right part of the above expression represents the fact that signature s has an 1 in all positions, where the query signature has also an 1. A superset query searches for all objects containing at least some of the queried attributes and only those. Here, the answer to this query is the collection of all objects O for which $O' \supseteq O$. It is similarly easy to show that

$$O' \supseteq O \Rightarrow q \supseteq s$$

where the right part of the above expression represents the fact that there is no position in the signature s having a value of 1 when the query signature has a 0 in the corresponding position. Thus, signatures can provide a filter for testing subset and superset queries on attributes, because if the subset (superset) condition does not hold for the signature, then it does not hold for the object neither. Since the inverse of the above relation is not necessarily true, some objects that do not satisfy the query may be retrieved as well which are called false drops.

As an example, consider a sample database of company from [TOUSIDOU 00] with the following class definitions, where notation '[]' is used for tuple constructors and '{}' is used to represent set-valued attributes:

Dept = [dname: str; projs: {**Proj**}, ...]

Proj = [pname: str; emps: {**Emp**}, Mnger: **Emp**]

Emp = [ename: str; projs: {**Proj**}, hobbies: {str}.,,].

Consider the query 'Find all employees who like to spend their free time cooking or fishing or playing basketball', or

$$hobbies \supseteq \{cooking, fishing, basketball\}.$$

Set-valued attributes, such as *hobbies,* can be represented by bit vectors, called *signatures.* Each distinct element of an attribute's domain is represented by a distinct signature.

Object-oriented and object-relational DBMS support set valued attributes, which are a natural and concise way to model complex information. The basic query type on set-valued attribute is inclusive or partial-match query, which retrieves all objects containing specific attributes. Inclusive queries can also be characterized as subset or superset queries, searching for all sets which are subsets or supersets of a given query set. Research in this area has focused on processing of keyword-based selection queries. A range of indexing methods has been proposed, among which signature based techniques [DEPPISCH 86], [ISHIWAKA 93] and inverted files [ZOBEL 92] dominate. These indexes have been extensively revised and evaluated for various selection queries on set-valued attributes [HELMER 03], [ZOBEL 98]. The performance of signature-file

organizations in OODBs for indexing set-valued objects has been studied in [ISHIWAKA 93] where two signature file organizations, the sequential signature file (SSF) and the bit-sliced signature file (BSSF) are considered and their performance is compared with that of the nested index (NIX) for queries involving the set inclusion operator ($\subseteq$). The authors conclude that the bit-sliced signature file (BSSF) achieves better performance than the sequential signature file approach (SSF) by almost 50% (of the time cost) in the best case. But the storage cost of BSSF is twice as that of SSF, and the update cost of BSSF is triple as that of SSF or more. The basic advantage of sequential signature files lies in their efficiency in handling new insertions and queries on parts of words.

When compared to tree-based indexing, sequential signature files suffer from two drawbacks:

- They can not be used to evaluate range queries; and
- For each query processed, the entire signature file needs to be scanned, which involves high processing and I/O costs.

RD-trees have been proposed for indexing set-valued data and when used with signatures they exhibit similar performance to that of S-trees [HELLERSTEIN 94]. Besides inclusive queries, Helmer et al. [HELMER 97] analyzes nested-loop and signature-hash algorithms for set-valued objects under the join query with subset/superset predicates. In [LEE 92], [LEE 96] and [YONG 94] the use of signatures in path expressions has been studied. Nested and path index, multi index, access support

relations and join index hierarchies for object-oriented and object-relational systems have been discussed by [BERTINO 89], [STEIN 86], [KEMPER 92] and [XIE 94] respectively.

Signatures are preferred to encode sets for three reasons [HELMER 03].

- They are of fixed length and hence convenient for index structures.
- Set comparison operators on signatures can be easily implemented by efficient bit operations.
- Signatures are more space efficient compared to the conventional set representation.

In order to improve selectivity in the upper level of signature-based tree structures Tousidou et al. [TOUSIDOU 00] proposed methods like linear split, quadratic split, cubic split etc. In [TOUSIDOU 02] the variation of a new method that combines linear hashing and S-tree is proposed to handle subset-superset queries. Norvag [NORVAG 99] has showed how logical OIDs stored in an index structure reduced average object access cost. A trie tree superimposed on an inverted file used as an index for subset-superset query evaluation is discussed by Terrovitis et al [TERROVITIS 06]. Nikos [MAMOULIS 03] has applied the various join types on set-valued attributes and inferred that signature-based methods are appropriate for set equality joins rather than inverted files.

2.4.2 Queries on nested object hierarchy

The increased complexity of data model supported by OODBMSs burdens it to address new issues and requirements in the design and analysis of suitable access mechanisms.

To be viable, the OO approach to data management must be supported by suitable techniques that directly implement the basic concepts of the object-oriented paradigm [BERTINO 94]. In OODBSs, an entity is represented as an object, which consists of methods and attributes. Methods are procedures and functions associated with an object defining actions taken by the object in response to messages received. Attributes represent the state of the object. Objects having the same set of attributes and methods are grouped into the same class. A class is either a primitive class or a complex class. Objects in the respective classes are called primitive objects and complex objects. A primitive class, such as an integer or string, is not further broken down into attributes or substructures. A complex class is defined by a set of attributes, which may be primitive, or complex with user-defined classes as their domains. A class C may have a complex attribute with domain C' establishing relationship between C and C'. The relationship is called the *aggregation* relationship. When arrows connecting classes are used to represent the aggregation relationship, an aggregation hierarchy (or, say, a nested object hierarchy) can be constructed to show the nested structure of the classes. Objects are nested according to the aggregation hierarchy [LEE 92]. 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. 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 is one of the key features which make the indexing problem of OODBs different from that of relational databases.

As a result of its wide acceptance of OODBSs, some implementation issues such as query processing and indexing become a crucial factor in the success of OODBS. OODBS which deal with complex objects require expensive traversal costs to process queries. [SHIN 96a] proposed a new signature scheme, called the *s-signature* on the path dictionary to efficiently support query processing of different types of queries. Shin et al. generated a signature from each object in an s-expression, which is an expression scheme of the path dictionary, and superimposed those signatures to form an s-signature for the s-expression. Each s-expression may contain several paths terminating at the same object. The s-signature provides an efficient filtering mechanism, so as not to access the database at the initial stage of query processing, which is required at the original path dictionary scheme. The s-signature also provides an efficient access method for the path dictionary instead of the sequential scanning.

In OODBSs a class consists of a set of attributes whose values sometimes are objects that belong to other classes; that is the definition of a class forms a hierarchy of classes. All the attributes of the nested classes are nested attributes of the root of the hierarchy. A branch of such a hierarchy is called a *path* [BERTINO 94]. Access Support Relations (ASR) proposed in [KEMPER 90] is equivalent to the path index. The authors suggest that a path can be split into several sub paths and different access support relations may be allocated on each sub path. Path index, nested index, and multiple indexes using an inverted file have been proposed in [BERTINO 89]. The path index, which is recommended for an aggregation hierarchy with a long path and virtually equivalent to the ASR, unfortunately requires high storage overhead and costly index

maintenance if it wants to support various key fields. However, the filtering capability of the tree signature scheme is weaken if the sub tree is large. The path signature has expensive update cost and storage overhead because it has a lot of redundant signatures and OIDs.

Among the two techniques proposed in [BERTINO 89] for fast evaluation of queries having nested predicates, one is to use nested index to index nested attributes and the other is to use field replication techniques [YONG 94] to avoid forward traversals. The field replication technique copies attribute values of the frequently referred object and stores them into the referring object. As such, during forward traversal process there is no need to access the referred object to retrieve its attribute values. However the field replication technique might cause high storage cost and data inconsistency problem. Indexing techniques can efficiently support backward traversals too. Again this requires high storage and maintenance cost. These overheads may constrain to limit indices for multiple attributes of the classes. Therefore on the whole, indices are maintained only for important attributes. The path dictionary scheme introduced by Lee et al. [LEE 94], [LEE 95a], [LEE 98] has lower storage overhead.

An example of a nested object hierarchy is extracted from [BERTINO 89] and shown in Figure 2.14, where an attribute of any class can be viewed as a nested attribute of the root class.

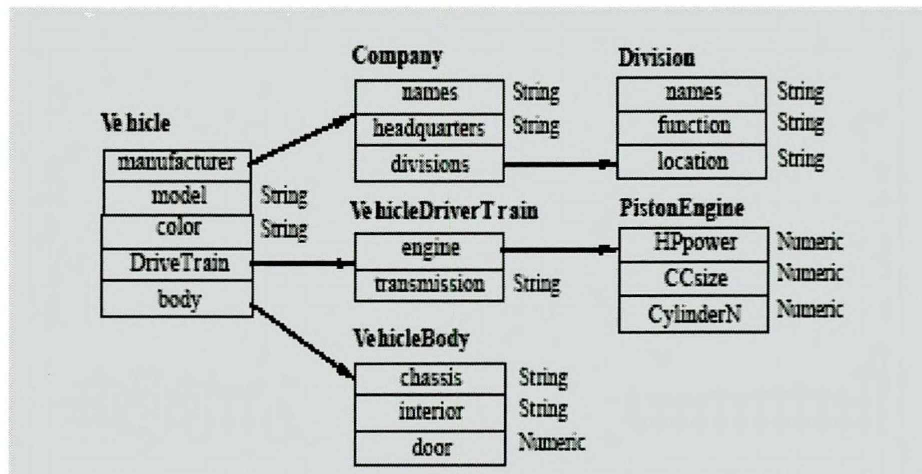


Figure 2.14 An example of nested object hierarchy

The hierarchy is rooted at class Vehicle. An attribute of any class on a class-attribute hierarchy is logically an attribute of the root of the hierarchy, that is, the attribute is a *nested attribute* of the root class. For example in Figure 2.14, the location attribute of the class Division is a nested attribute of the class Vehicle. In OODBs, 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 "retrieve all red vehicles manufactured by a company with a division located in Ann Arbor" can be expressed as:

select Vehicle

where Vehicle.color = "red"

and Vehicle.Company.Division.location = "Ann Arbor"

The search condition against the class Vehicle consists of two predicates one involving the attribute 'color' and the other involving the nested attribute 'location'. Without indexing structures, the above query can be evaluated in a top-down manner as follows. First, the system has to retrieve all of the objects in the class Vehicle and single out those that are red in color. Then, the system retrieves the Company objects referenced by the red vehicles and checks the locations of the divisions of the manufacturers. Finally, those red vehicles made by a company that has a division located in "Ann Arbor" are returned.

[CHEN 04] introduces a new indexing method to speed up this process. This method is based on the technique of signature files and can be summarized as follows:

(i) All signature files are organized into a hierarchical structure to facilitate the implementation of a step-by-step filtering strategy.

(ii) Each signature file is stored as a tree structure to speed up signature file scanning.

In terms of an aggregation hierarchy, a signature file hierarchy can be constructed as follows:

(i) The signature of an object is generated by superimposing the signatures of all its primitive and complex attributes.

(ii) The signature of a primitive attribute is obtained by hashing on the attribute value; the signature of a complex attribute is the signature of the object it references.

(iii) Let C be a class, and let o_1, ..., o_l be its objects; there exists a signature file S such that each o_i (i = 1, ..., l) has an entry <osig, oid> in S.

(iv) Let S_i and S_j be two signature files associated with classes C_i and C_j, respectively. If there exists an arrow from C_i to C_j, then there is implicitly an arrow from S_i to S_j.

To cut off irrelevant data as early as possible, the top-down approach is used to retrieve all the objects along the path from the target class to its nested attributes specified in the search condition of the query. Then, the value of the nested attribute is checked to decide if it is a desired object or not. With the signature file, the query is evaluated as follows. A query signature s_q for the query Q is generated. s_q is compared with every signature stored in the signature file associated with the target class. If a signature matches s_q, the path is traversed to verify the nested attribute.

Consider the above mentioned query once again. Assume that the signatures for "red" and "Ann Arbor" are s_{red} = 100 110 000 100 and $s_{Ann\ Arbor}$ = 010 000 100 110, respectively. First, we construct $s_q = s_{red} \vee s_{Ann\ Arbor}$ = 100 110 000 100 ∨ 010 000 100 110 = 110 110 100 110. It matches the entry in the signature file of *Vehicle* shown in Figure 2.15. Then, the corresponding OID is put in *S*. The next step of the algorithm checks the 'color' attribute of OID. If it matches s_{red}, we traverse the paths from this *Vehicle* object to those *Division* objects reachable over some *Company* objects. Such objects of *Division* are checked to eliminate false drops. The signature of the *Division* object shown in Figure 2.15 matches $s_{Ann\ Arbor}$. Then, the attribute value of "Location" of this object is checked to see whether it really matches "Ann Arbor".

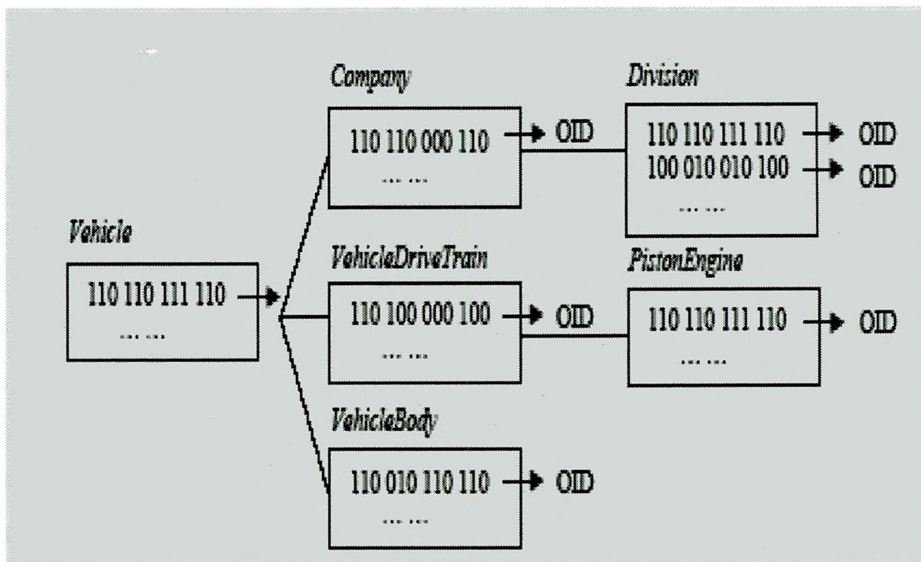
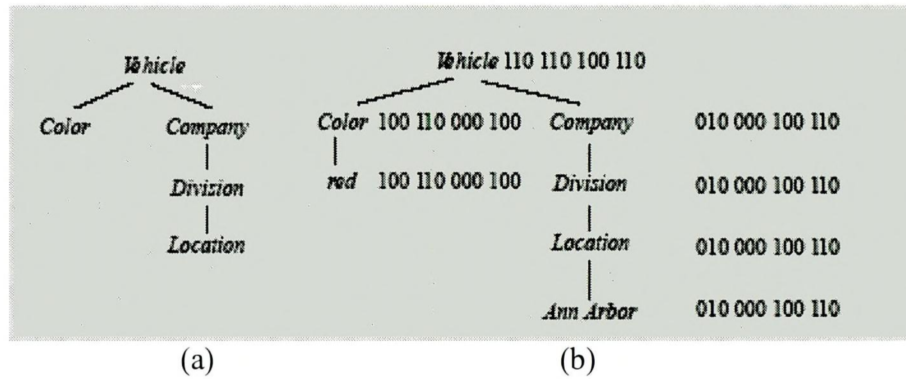


Figure 2.15 Signature and signature file hierarchy

From this example, we can see that the signature files are used only to locate the relevant objects of the target class. The optimization possibility provided by the signature file hierarchy is not employed at all. It is not efficient because all the sub trees rooted at the relevant objects of the target class have to be searched exhaustively. To overcome this drawback, Chen [CHEN 04] uses query tree and query signature tree by means of which attention is paid to the query structure to make the signature file hierarchy useful. For the same exemplar query, the query tree and the query signature tree are as shown in Figures 2.16 (a) and (b), respectively. In this way traversal of object hierarchies is optimized by building signature file hierarchies which removes all irrelevant branches as early as possible.



(a) (b)

Figure 2.16 Query tree and query signature tree

There are many kinds of nested queries supported by OODBMSs and several indexing organizations have been proposed to support nested query processing as in [LEE 94], [LEE 92], [SHIN 96] and [YONG 94]. In object-oriented databases, the attribute of an object can have the OID of another object as a value. Through this OID, objects can refer to other objects. That is each object of a class can refer to an object or objects of other class by defining the referred class as the domain of the attribute of the referring class. Those classes related with their attributes form a hierarchical structure called class-attribute hierarchy. A predicate that has a nested attribute is called a nested predicate. To evaluate queries with nested predicates three traversals namely forward traversal, backward traversal and mixed traversal are proposed in [KIM 88].

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. In OODBSs with aggregation relationship the class from which objects are retrieved is called *target class* and the class with attributes involved in query predicates is called *predicate class* [LEE 94]. There are two basic approaches to evaluate nested query; *top-down* and *bottom-up* as discussed in [LEE 96], [YONG 94]. The top-down approach also known as *forward traversal* traverses the object starting from an ancestor class to a nested class. On the contrary, the bottom-up method which is otherwise known as

backward traversal traverses up the aggregation hierarchy. The performance of the top-down and the bottom-up approaches depend on a number of factors [LEE 92]: the sizes of the classes along the paths from the target class to the nested attributes, the conditions specified on the classes and the selectivity of the conditions, whether indexes are available for those classes and whether backward references are supported. The top-down approach is in general more efficient because only forward references are needed. Nested queries may be classified by the relative positions of the target and a predicate class in the aggregation hierarchy as follows [LEE 94]:

- ***TP queries***: These types of queries have a set of predicates on attributes of the target class.
- ***PT queries***: These types of queries retrieve the nested objects of a given set of objects that is specified by predicates.
- ***MX queries***: The target class of the query is between two predicate classes.

Many indexing structures to handle nested queries from OODBSs using signature-based techniques have been proposed. In [LEE 96] signature technique combined with path dictionary to result a new secondary index called signature path dictionary is presented. Signature based hybrid schemes combining hashing and S-tree for retrieval of objects with set-valued attributes is introduced in [TOUSIDOU 02]. In [NORVAG 99] the use of signatures in maintaining OID index to reduce the object retrieval cost is discussed. Paper [SHIN 96] discusses the use of S-signature on S-expression for different query handling in OODBSs. Lee et al. [LEE 92] reported tree signature and path signature methods for

single class queries. The tree signature scheme creates the signature of an object by superimposing all its attributes' signatures (both simple and nested) in the aggregation hierarchy whereas the signature of a primitive attribute is obtained by hashing on the attribute values and the signature of a complex attribute is the signature of the object it references. For every class C in the aggregation hierarchy, there exists a signature file S such that every object O in C has an entry <*sig, oid*> in S, where *sig* is the signature of O and *oid* is the OID of O. The filtering capability of the tree signature scheme is weaken if the sub tree is large.

However, in the path signature scheme, the signatures of complex attributes located on the path and those not located on the path are generated in different ways. The signatures of complex attributes located on the path are generated by superimposing the signatures of its nested attributes, while the signatures of other complex attributes are generated by using the object identifiers stored in the attributes as the keys for hashing. That is, for every class C in a given path of the aggregation hierarchy, there exists a signature file S such that every object O in C has an entry <*sig, oid-list*> in S, where *sig* is the signature of O and *oid-list* is the object identifiers of O and its nested objects located in the path. Another difference between the tree and path signature scheme is that the path signature scheme uses a list of OIDs, which includes the object identifiers of O and its nested objects on the path, in the signature files.

Comparing to other index schemes the tree signature scheme provides a more general support for queries involving any attributes in the database. The path signature scheme on the other hand pays more attention on attributes in a given path of classes.

Every object in the path is indexed by a signature. Since there are less attributes involved, the path signature scheme is more effective on filtering unqualified objects.

The signature file technique can be conveniently used to index nested object hierarchies. In [YONG 94] signatures have been applied in forward OO query processing to avoid physical OID access in case of non-matching query. Signature for each object is generated from its attribute values and stored into the referring objects to support forward traversals. Using this technique nested predicates can be checked by using these signatures without accessing referred objects. Only when the predicates are successfully matched with the signature will the forward traversals continue to read values of the referred objects. Also, the characteristic that the signature file has low storage cost, will allow all attribute values of objects to be included to make the signatures. This makes it possible to evaluate any predicates on the classes. In the case where nested indices are also defined, it is possible to evaluate queries using mixed traversals which have both the nested index and the signatures. The signature replication technique used generates object signatures from simple attributes of the object. It proposes an optimal method by means of which signatures are distributed over a nested object hierarchy and the signature of a referred object is stored in the referring one. Then, such a signature can be used to check the predicate involving the corresponding object before it is visited. In this way, some evaluation time can be saved. Inherited multi index and nested inherited index have been studied in [BERTINO 95]. [TOUSIDOU 02] reports new hybrid schemes for OO queries combining hashing and tree based methods.

One serious problem common to all the above mentioned schemes is the object retrieval time of query matching for large databases. None of the structures is able to retrieve matching oids in a single access. In the following section we briefly describe the structure of SD-tree that overcomes the difficulties listed. We follow the lead of [CHEN 04] for query signature tree creation and evaluation in this report.