A Partitioned Signature File Structure for Multiattribute and Text Retrieval

Dik Lun Lee   Chun-wu Leng

Department of Computer and Information Science
The Ohio State University
2036, Neil Ave
Columbus, OHIO 43210-1277

e-mail address: {dlee, leng}@cis.ohio-state.edu

appeared in ICDE '90

Abstract

Signature file has been shown to be a very good access method for text and multiattribute retrieval [6,13]. It has a low storage overhead and a simple structure. However its speed is poor compared to an inverted file, since every signature in the file must be searched. In this paper, we introduce a partitioning method to reduce the search space required on the signature file. A partitioned signature file is better than the multilevel signature file [15] and the S-tree [3] in that it has an extremely small storage and processing overhead.

We outline three partitioning schemes and discuss their performance. We describe the data structure necessary to support the partitioning schemes and the algorithms for signature insertion, deletion and retrieval.

1 Introduction

Signature files, in particular, the superimposed coding method [14], is an excellent access structure for supporting conjunctive queries. For instance given a relation $R = (A_1, A_2, ..., A_n)$, a partial match query may specify a conjunctive condition on a number of attributes, such as $A_i = v_i \wedge A_j = v_j \wedge ...$ where $v_i, v_j, ...$ are constants. For text retrieval, a document (or a logical block of a document) can be considered as a record containing n keywords and the query may ask for all documents (or logical blocks) that contain a set of keywords. The difference between text retrieval and multiattribute retrieval in database is that typically the order of the keywords in the text is not important for text retrieval. As such, superimposed coding is particularly suitable for text retrieval since it scrambles all the keywords in the text by encoding them randomly in the superimposed codes (see [6] for survey of this subject). For convenience, we describe our partitioned signature file based on text retrieval throughout this paper.

A signature file is basically an abstraction of a text file (or record). It provides an effective way of screening out documents (or records) which don't match with the query. The main

This research has been supported by the Applied Information Technologies Research Center (AITRC), Columbus, Ohio, Project No. EES 4509 and EES 4628.

advantage of signature files is its simple file structure and low storage overheads. Unlike inverted files which may have more than 100% storage overhead, a signature file is typically 10-20% the size of the primary database and is therefore very suitable for large databases [2].

A lot of research has been done on the improvement of the performance of a signature file. One direction of the research studies optimal algorithms for generating signatures from the text and the estimation of false drop probability [7,8]. The goal is to improve the overall system performance by reducing the false drop rate. Another direction is to improve the search speed on the signature file itself by reducing the amount of data in the signature file that needs to be searched. These two approaches are orthogonal to each other in the sense that both approaches can be used to yield even better results. The research presented in this paper is along the second direction. We present a new way to organize the signature file so that the simple structure of the signature file is preserved and that the search can be restricted to a possibly very small portion of the signature file.

Work has been done on increasing the speed of searching the signature file [11]. The techniques range from using a transposed file organization [14] to the exploitation of special hardware [1,9]. However, most of the techniques require the search of every signature in the signature file. Thus, when the file is large, the performance will degrade accordingly. Multi-level signature files have been proposed to alleviate the problem [12,15,16]. In a two-level signature file, the lower-level signature file contains signatures generated from individual text blocks, and, in addition, signatures are generated for groups of text blocks and stored in the higher-level signature file. The higher-level signature files provide a coarse filtering and, therefore, only a portion of the lower-level signature file is typically searched. This method requires extra storage and processing overhead due to the multi-level structure. Moreover, for queries with a low selectivity (i.e., a large number of signatures are qualified), this approach may be even slower than a single-level signature file due to the overhead incurred in the multi-level file organization. The S-tree also uses a tree-like structure and shares the same problems with the multi-level signature file [3].

In this paper, we propose a "flat" structure similar to hashing. The signature file is divided into partitions according to some "key" derived from the signatures. Therefore, each partition contains signatures having the same key. When a query is received, the same method is used to derive a key from the query signature, and only those partitions with a key matched with the query key are searched; partitions not matching the key need not be searched. However, as discussed later, a search on a signature file does not look for exact match as standard hashing does, thus introducing some unique and interesting problems of its own.

The main problem of the partitioned signature file is to find a good partitioning scheme and evaluate its performance. We studied elsewhere three partitioning algorithms and evaluated in detail their performance both analytically and with simulation [10].

In Section 2, the principle of signature file and signature partitioning is reviewed. A summary of the three partitioning algorithms and their performance is presented in Section 3. We then discuss in Section 4 the data structures required for supporting the partitioning methods and, in Section 5, the algorithms for the insertion, deletion and retrieval of signatures.

2 Superimposed Coding and Signature File Partitioning

B_i	i^{th} block
k	length of a key
K_{M_i}	key for the i^{th} partition
K_{S_i}	key obtained from the i^{th} block signature
$K_{S_i}^V$	key value, the bit string which forms the key
$K_{S_i}^P$	key position, the starting position of the key
K_{S_Q}	key obtained from the query signature
m	length of a signature
M_i	i^{th} partition
p	page size
S_i	i^{th} signature
$\bar{S}_i$	non-key portion of the i^{th} signature
S_Q	query signature
$\bar{S}_Q$	non-key portion of a query signature
w_X	no. of ones (weight) of bit vector X
$\bar{w}_X$	no. of zeros of bit vector X
z	no. of zeros in a key

Table 1: List of Symbols.

In superimposed coding, the database is divided into a number of logical blocks, each containing about the same number of distinct words. A block B_i is associated with a signature S_i which is a fixed length bit vector $S_i^1 S_i^2 ... S_i^m$, where m is the length of the signature and $S_i^j \in \{0, 1\}$. (See Table 1 for a summary of the symbols used in this paper.) To obtain the signature, each non-trivial word in the text block is hashed into a bit vector, called a *word signature*. The number of one's in a word signature is in general fixed. The *block signature* (or simply signature for short) is generated by superimposing all word signatures for that block.

When a word is searched for, a *query signature*, S_Q, of the word is generated from the query, Q, in the same way as described above. S_Q is then compared with every block signature. A block signature, S_i, is *qualified*, if and only if for all bit positions in S_Q which are set to 1, the corresponding bit positions in S_i are also set to 1 (known as the *inclusion* condition). Formally, the set of qualified signatures is defined by:

$$\{S_i \mid (S_i \wedge S_Q) = S_Q\} \tag{1}$$

In this formula bitwise conjunction is used. A disqualified signature indicates that the associated text block is guaranteed not to contain the word specified in the query.

Since a signature is generated for each text block, the size of the signature file and therefore the search time involved are linear with the size of the text file. In this paper, we investigate the partitioning of a signature file with a deterministic algorithm. The method relieves us from searching every partition in order to answer a query and thus reducing the search space involved.

The main problem in signature partitioning is to determine the hash function to map signatures into partitions. The process is similar to hashing but the retrieval operation is quite different from that of hashing. Standard hashing looks for an exact match of a key in the

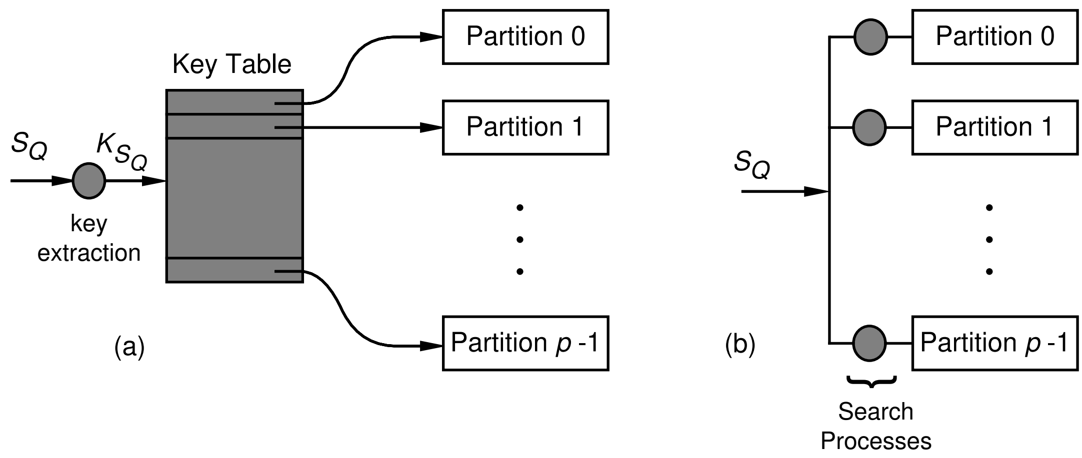

Figure 1: The partitioned signature file approach.

hash table, whereas an inclusion condition as indicated in Equation (1) must be satisfied in a signature file.

To simplify our discussion and analysis, we assume that S_i's and S_Q are random bit vectors uniformly distributed in the range of 0 to 2^{m-1}; each bit position is equally likely to have a 0 or 1 on it. Since S_i's are random bit vectors, substrings taken from S_i's are themselves random bit vectors and hence can be used as keys.

Let K_{S_i} and K_{S_Q} be the keys for a block signature, S_i, and a query signature, S_Q, respectively; and $\bar{S}_i$ and $\bar{S}_Q$ denote the non-key portions of S_i and S_Q respectively. Signatures having the same key are grouped into the same partition. Since a partition contains signatures of the same key, we use K_{M_i} to refer to the key of the partition, M_i, and k to denote the length of a key. More precisely,

$$M_i = \{\bar{S}_i \mid K_{S_i} = K_{M_i}\} \tag{2}$$

When a query signature S_Q is received, it is compared to the keys of the partitions. K_{S_Q} is obtained by extracting the substring from S_Q at the same position at which the key of the partition is obtained. M_i is activated when K_{S_Q} includes K_{M_i}:

$$(K_{S_Q} \wedge K_{M_i}) = K_{S_Q} \tag{3}$$

Condition (3) is checked first when S_Q is received. Partitions satisfying (3) are further searched for signatures with $\bar{S}_i$ satisfying $\bar{S}_Q$. The partitioning approach can be used in a sequential environment as well as in a parallel system. Figure 1a illustrates the general organization in a sequential environment. The partitions are stored on secondary storage. In order to identify the partitions to be searched, some data structure (key table in the figure) must be used to map the query signature to the set of partitions to be searched. In a parallel system, a partition is stored in a processor (either on disk or memory resident) and query processing can be done simultaneously in each processor (see Figure 1b). It doesn't need any centralized data structure as in a sequential environment.

3 Algorithms and Performance Evaluation

We studied three partitioning schemes: the *Fixed Prefix* method, the *Extended Prefix* method, and the *Floating Key* method. Detailed analytical and simulation results for their performance have been obtained [10]. The performance of a partitioning scheme is defined by the search space reduced. The reduction of search space can be observed by studying two ratios: (1) the ratio of the number of signatures searched to the number of signatures in the signature file (*the signature reduction ratio*); and (2) the ratio of the number of partitions searched to the number of partitions in the signature file (*the partition reduction ratio*). The first ratio indicates the amount of processing (or comparisons) required. The second ratio indicates the amount of disk accesses required if a partition is retrieved with one disk access.

3.1 Fixed Prefix Partitioning

The simplest way to select a key is to take the first k bits of a signature as the key. Signatures with the same k-prefix are grouped in the same partition. Likewise, the first k bits from S_Q can be taken as K_{S_Q}. Only partitions whose keys include the query's key are searched further.

This method is very attractive since it is extremely simple and offers significant advantage in speed with little overhead in the partitioning algorithm [10]. However, some partitions are activated more frequently than others. This can be seen from an extreme case that the partition with key "111...11" satisfies condition (3) regardless of the query and thus will be searched for every query. This partition will become the bottleneck of the system. To alleviate the problem, it would require more buffer and/or faster disks be used for that partition. One of the goals for the next two partitioning algorithms is to make the accesses to the partitions as uniform as possible so that the workload is evenly distributed across the partitions.

3.2 Extended Prefix Partitioning

In order to achieve a uniform workload distribution on the partitions, we can make the probability that a partition is searched a constant. We discover that the probability solely depends on the number of zeros in a partition's key, and that the probability decreases as the number of zeros in the key increases [10]. Therefore, when the keys are chosen, we have to make sure that every key contains at least one zero and, further, the number of zeros in each key must be the same.

In the Extended Prefix method, the key of a signature is chosen to be the shortest prefix which contains a pre-defined number, z, of zeros. Figure 2(a) is an example for partitioning 20 signatures with m equal 6 and z equal 2. The keys of the signatures are underlined in the figure. They are shown only for clarity. In reality, the signature keys are represented by the partition keys. Since z is typically very small (< 10) in order to be practical, we assume that a signature must contain at least z zeros.

A major problem with this method is that the partition size is highly non-uniform. In particular, partitions with $w_{K_i} = w_{S_i}$ hold only one unique signature. This can be seen from the example in Figure 2(a). We will show how this can be dealt with in our dynamic storage allocation scheme.

partition 1	$\underline{00}$ 0111
	$\underline{00}$ 1011
	$\underline{00}$ 1111
	$\underline{00}$ 1010
partition 2	$\underline{010}$ 011
	$\underline{010}$ 101
	$\underline{010}$ 111
partition 3	$\underline{0110}$ 01
	$\underline{0110}$ 11
partition 4	$\underline{01110}$ 0
partition 5	$\underline{100}$ 000
	$\underline{100}$ 111
	$\underline{100}$ 110
partition 6	$\underline{1010}$ 01
	$\underline{1010}$ 10
partition 7	$\underline{10110}$ 0
partition 8	$\underline{1100}$ 00
	$\underline{1100}$ 11
partition 9	$\underline{11010}$ 0
partition 10	$\underline{11110}$ 0

(a)

partition 1	$\underline{00}$ 0111
	$\underline{00}$ 1011
	$\underline{00}$ 1111
	$\underline{00}$ 1010
partition 2	01 $\underline{00}$ 11
	10 $\underline{00}$ 00
	11 $\underline{00}$ 00
	11 $\underline{00}$ 11
partition 3	0111 $\underline{00}$
	1011 $\underline{00}$
	1101 $\underline{00}$
	1111 $\underline{00}$
partition 4	$\underline{01}$ 0101
	$\underline{01}$ 0111
	$\underline{01}$ 1001
	$\underline{01}$ 1011
partition 5	$\underline{10}$ 0111
	$\underline{10}$ 0110
	$\underline{10}$ 1001
	$\underline{10}$ 1010

(b)

Figure 2: Distribution of signatures based on (a) Extended Prefix, and (b) Floating Key.

The advantage we expect from this method is that the probability that a partition is searched is absolutely uniform. In a parallel implementation in which one partition is assigned to one processor (e.g., in a connection machine or hypercube), the extended prefix method may yield a more uniform workload on each processor.

3.3 Floating Key Partitioning

We can observe that if keys are selected only from the prefix, signatures with a sequence of ones followed by a sequence of zeroes may result in partitions with very long keys and small sizes, yet most of the bits in the key are ones which cannot be used to reduce the chance of that partition being searched.

In the Floating Key method, we avoid the problem by choosing a fixed key length, say, k and selecting the k-substring in the signature which contains the least number of ones. Specifically, each consecutive, non-overlapping, k-substrings of the signature is examined from left to right, and the leftmost substring with the smallest weight is chosen as the key. The key in this method composes of the *key value* ($K_{S_i}^V$, the substring which forms the key), and the *key position* ($K_{S_i}^P$, the starting bit position of the key in the signature). $K_{S_i}^V$ is a k-bit bit pattern, and $K_{S_i}^P$ takes the values of $1, k+1, 2k+1, ...$, etc.

This algorithm attempts to maximize the number of zeros in a key. Since the probability that a partition is activated by a query decreases as the number of zeros in the partition's key increases, this algorithm tends to minimize the number of partitions searched for a query. This has been verified by analytical and simulation results. We found that both the partition and signature reduction ratios are the smallest among the three methods. We found that the reduction in search space could be as high as 90% for a query weight of 100, which is not unusual in a text retrieval environment (when there are about 10 query terms in the query).

Our conclusion about the three methods we studied is that the performance of the floating key method is the best. Although the partition size and the workload are non-uniform, the variation is acceptable in a practical system. A minor drawback of this method is that the algorithm for obtaining the key from a signature and for comparing between keys is more complex, but this should not be a serious problem as computation time is negligible compared to I/O costs.

4 Data Structures

In a parallel environment, such as the one shown in Figure 1(b), it is rather straightforward to determine which partitions must be searched upon the receipt of a query. The query signature can simply be sent to all search processes, which extract the query key according to the partitioning scheme implemented, and compare the key to the corresponding partition keys. Partitions whose keys don't match with the query signature's key can simply terminate the search and become available for the next query. The mapping of signatures to the partitions is done in parallel and distributed among the search processes; no sophisticated data structures are needed.

In a sequential environment, however, some data structure must be used to hold the partition keys, and the key of the query signature must be compared to the partition keys in the data structure sequentially. In the rest of this paper, we study the data structures for supporting the partitioning schemes and the algorithms for the insertion, deletion, and search of the signatures.

Although three different partitioning methods have been introduced, the data structures required to support them are essentially the same as the one depicted in Figure 1(a). The only difference among the different schemes is the key extraction function. The floating key method can be considered as a generalization of both the fixed and extended prefix methods while the fixed prefix method is a special case of the extended prefix method and the floating key method. We introduce a general data structure which supports the floating key method directly but can be adapted to the other two methods with little change.

4.1 Overall structure

Figure 3 depicts the general data structure for supporting the partitioning schemes. After receiving a signature, the key extraction algorithm outputs a 3-tuple containing the starting bit position of the key in the signature, $K^P_{S_i}$, the key length, k, and the signature, S_i, itself. The key-position directory maps $K^P_{S_i}$ to a trie representing the keys starting at $K^P_{S_i}$. $K^P_{S_i}/k$ is used as an offset in the key-position directory to an entry pointing to the corresponding trie. For the first two methods, $K^P_{S_i}$ is always one. Therefore, the directory contains only one entry and

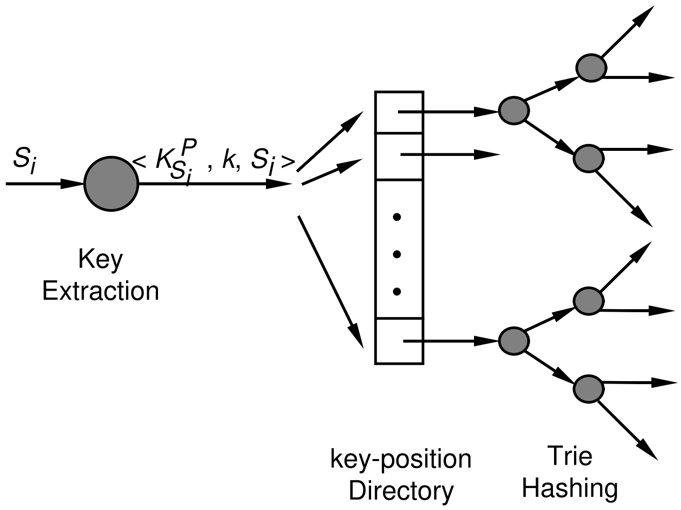

Figure 3: A general data structure.

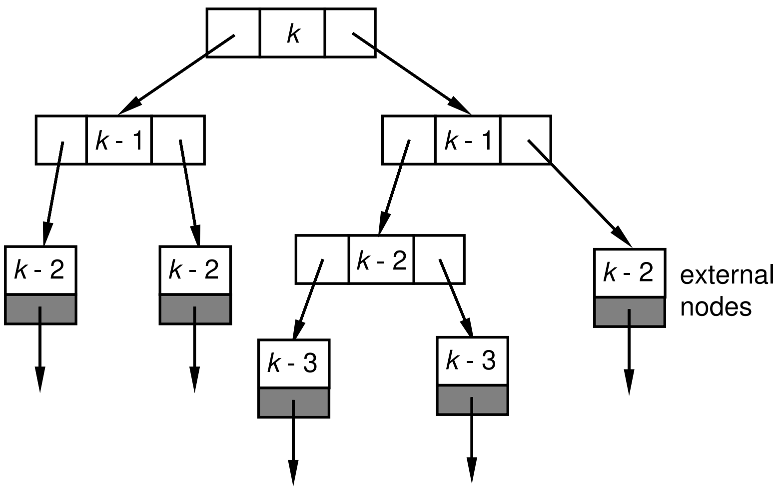

Figure 4: A trie encoding a set of keys.

can be eliminated in practice. However, we assume the directory exists for all three methods so that our description applies to all three of them.

A trie encodes the set of keys extracted at a certain starting position. It is used to hash signatures to their corresponding partitions during insertion and to determine the partitions to be searched during query processing. Figure 4 shows the details of a trie. The trie in the figure represents the set of keys: {00, 01, 100, 101, 11}. The external nodes point to the partitions of the corresponding keys. It can be seen from the figure that the trie is a natural way of representing variable-length keys. Therefore, it can handle all three schemes without difficulty. In fact, it is an important feature for us to deal with the dynamic growth/shrinkage of the partitions by *not* encoding the full key into the trie.

In addition to the pointers pointing to its children, a node records some information about the key. For the fixed prefix and the floating key method, it records the number of unused bits in the key, and for the extended key method, the number of unused zeros in the key. The example in Figure 4 is for the fixed prefix and floating key method. For the extended prefix method, k should be replaced with z. We will elaborate later how this information is used in operations on the trie.

This data structure has some distinct advantages:

1. The same structure can be employed for various partitioning schemes by revising the way to extract the key from a signature.
2. It is efficient for the insertion, deletion, and search of signatures. The algorithms will be described in the next section. We can also see later that the time complexity depends on the number of partitions which match the query but not on the total number of partitions in the system.
3. The trie can dynamically allocate storage space to a partition. It is efficient in storage utilization, even for small files. In addition, it reduces the number of disk accesses required for large partitions. This will be elaborated in the next section.

4.2 Dynamic Storage Allocation

The characteristics of the partitioning schemes pose two major problems on the physical design. The first is that the number of partitions generated from a signature file can vary greatly. As the signature file grows, the number of partitions grows as well. Therefore, it is impossible to pre-allocate a fixed number of partitions. Second, the size of each partition can be highly non-uniform, especially for the extended prefix and the floating key method. If a fixed page size is used and a partition occupies at least one page, a lot of storage will be wasted since many partitions contain only a few signatures. On the other hand, a large partition will span over several pages thus requiring multiple disk accesses. To alleviate these problems, it would require a dynamic storage scheme in which partitions can be allocated dynamically. Moreover, the size of the partitions must be incrementally expandable. The trie structure described above solves both problems in a way similar to dynamic hashing [5].

In order to simplify our discussion, we only examine what would happen on one trie. Initially, the trie contains only one external node (the root). This node has a pointer pointing to an empty page. Then signatures are gradually inserted into this page and eventually the page will be full. The system will split the overflow page into two pages, and the external node pointing to that page becomes an internal node with two external children, each of them points to one of the newly created pages. When a large signature file is to be allocated, a trie with a certain depth d can be created in advance, with the effect of pre-allocating 2^d pages and avoiding a large number of splits. When the depth of the trie reaches k, with at most 2^k pages generated, we can either increase the key length k or link pages to the full page for storing the extra signatures.

The storage allocation scheme is the same for the three partitioning schemes, with the exception that the number stored in the nodes indicates the number of unused bits in the key for the fixed prefix and floating key method and the number of unused zeros in the key for the extended prefix scheme. The reason why we need to trace the remaining key length or zero bits in the key is that a page should not be split any further when the maximum key length is reached (i.e., all k bits are encoded in the trie). When this happens, an overflow page can be expanded with a link list structure. The following provides the details in a split operation:

1. Create two new pages, a left page and a right page, and two external nodes pointing to these pages.
2. Separate the signatures in the overflow page into two groups. The first group contains all the signatures whose $K^P_{S_i}$th bit is zero, and the remaining signatures belong to the second group. Then store the first group in the newly created left page and the second group in the right page after removing the $K^P_{S_i}$th bit from each signature.
3. Replace the old external node, which points to the overflow page before split, with a new internal node which contains two pointers to the new external nodes. Move the number stored in the old external node to the new internal node.
4. Set the node value of each new external node to the value in its parent minus one. However, there is an exception for the extended prefix method. In this scheme, the number in each node indicates the number of zero bits remaining in the key so far; therefore, the number in the left child should be reduced by one but the number in the right child remains the same as its parent's value.

Our splitting technique provides a simple and efficient way to separate an overflow page into two pages having the same prefix in the key. Unlike the split in S-tree [3], our technique is not required to reconstruct the whole trie along the path in which overflow occurs but only scans the overflow page to distinguish the $K^P_{S_i}$th bit of the signatures. In the next section, three major operations of this system, namely, the insertion, deletion, and search of the signatures, are introduced. It can be seen that the trie structure associated with our splitting technique is very promising for the signature file environment.

Figure 5 and 6 show the data structure corresponding to the examples in Figure 2a and 2b. In the example for the extended prefix method, partitions 4, 7, 9, and 10 contain only one signature. If each of them takes one physical disk page, the storage utilization will be very low. Furthermore, the trie will be very unbalanced if we use the whole key in the trie. In the dynamic trie hashing method, partitions 3 and 4, 6 and 7, and 8 and 9 are grouped together. For example, partitions 3 and 4 are stored in the third page (from the left). The external node indicates that still one zero in the key is unused. The page can be split further when overflow occurs, and partitions 3 and 4 will be separated. We can also see that the trie is quite balanced, and that partition 1 uses two pages linked together to accommodate its four signatures (assuming a page size of 3) since the full length of the key has already been used. In the example for the floating key method, a key can start at position 1, 3 or 5. Therefore, there are three corresponding tries. In the Figure 6, the partitions are stored in their own page since the full key has been encoded in the trie. The principle is virtually the same as that of the extended prefix method, except that more than one trie are used.

5 Operations on the signature file system

We have discussed the importance of page split in supporting the operations of the system. Usually, for a regular signature file system, the most frequent operations acting on a file are

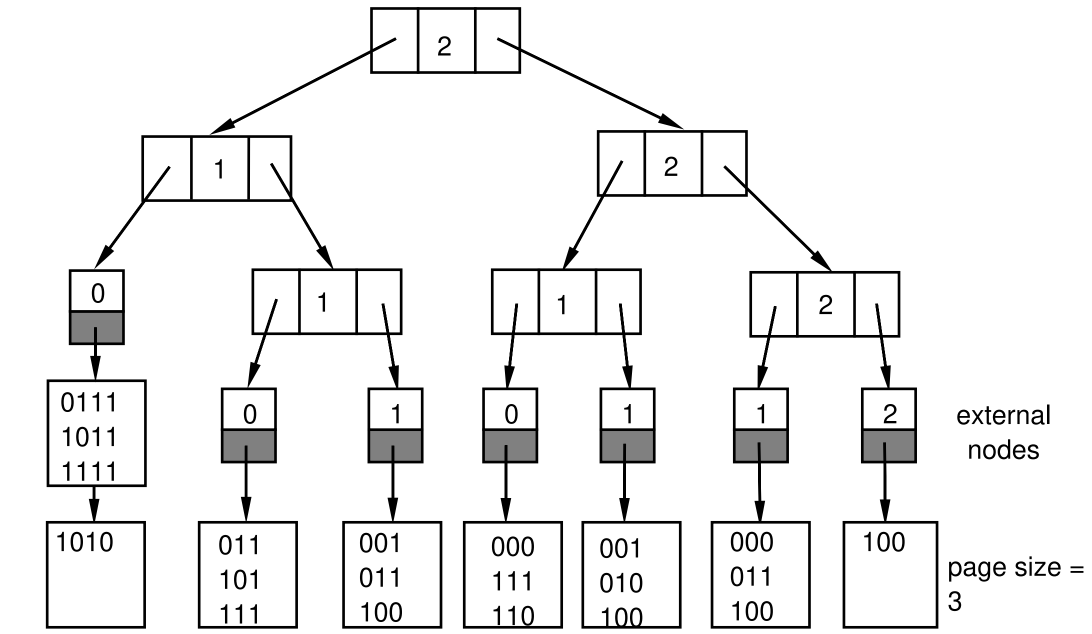

Figure 5: Dynamic trie structure for the extended prefix method.

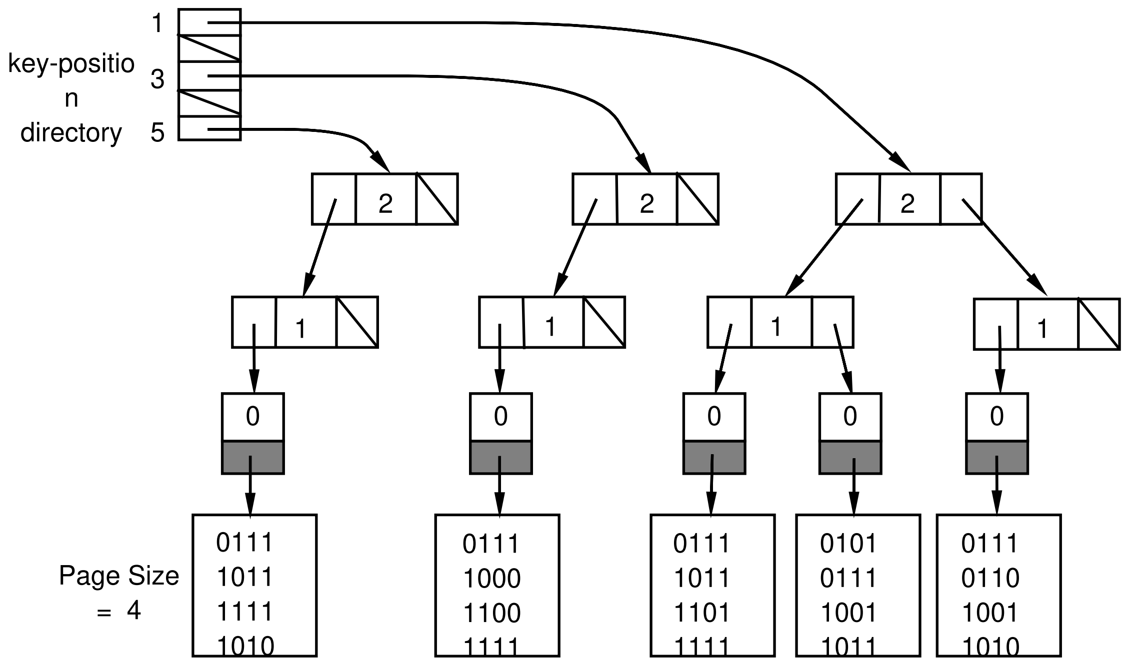

Figure 6: Dynamic trie structure for the floating key method.

insertion and retrieval [4]. It is especially important for the retrieval operation to be efficient in storage utilization as well as to avoid unnecessary comparison of unqualified partitions. The following description sketches how these operations work.

5.1 Insertion

When a signature is received, the key extraction function returns the starting position of the key, $K^P_{S_i}$, and the key length, k. From the starting position, we can locate the trie from the key-position directory and traverse down the trie according to the bit pattern starting at $K^P_{S_i}$. When a node is visited, the bit at $K^P_{S_i}$ is deleted from the signature. (The bit is physically removed so that the $K^P_{S_i}+1$ bit becomes the $K^P_{S_i}$ bit after the deletion.) The deleted bit also determines which child should be visited next. When the leaf page is reached, several options are available to handle the remaining signature string.

1. If the page pointed to by the leaf is not full, just put the signature into that page.
2. If the page pointed to by the leaf is full and the number in the external node is greater than zero, split the overflow page.
3. If the page pointed to by the leaf is full and the value is zero, then allocate a new page, link it to the overflow page, and store the signature in the new page.

5.2 Deletion

Deletion is similar to a retrieval except that it traverses down to the leaf which exactly matches with the key, and then followed by the disposal of the signature. We will talk about the retrieval in the next section. Here only the portion about merging two brother pages to one page is described. The idea of merging is the same as splitting except that it performs the opposite operation at each step. For example, when two pages are merged, we should insert a zero or a one to the signatures in the left and right page, respectively, right in front of the $K^P_{S_i}$ bit position in the signature.

The major problem for merging two pages is that it is necessary to count the total number of signatures in the two pages every time a deletion is performed. It is a quite time-consuming procedure due to the sequential scanning of both pages. We can solve this problem by either adding one more field in each external node indicating the number of signatures in the page it points to or just ignoring the merging process, assuming that the frequency of deletion will never keep up with the frequency of insertion.

5.3 Retrieval

Without the need of a sequential comparison, the trie structure can find all the qualified partitions at a complexity proportional to the number of qualified pages instead of the number of all pages in the trie. However, despite the similarity of the three methods in the operations described so far, the fixed prefix and the extended key method require the search of only one trie (since there is only one trie in these schemes) but the floating key method must search all tries for each possible key derived from the query signature. The procedure for retrieval is described below.

Upon the receipt of a query signature, S_Q, extract the corresponding key K_{S_Q} from bit position $K^P_{S_Q}$ to $K^P_{S_Q}+k-1$. For the fixed prefix and the extended prefix method, $K^P_{S_Q}$ is fixed to 1. Thus, there is only one possible K_{S_Q} for these methods. For the floating key method, $K^P_{S_Q}$ can be $1, k+1, 2k+1, ...$ Therefore, there are $\lfloor m/k \rfloor$ K_{S_Q}'s. For each of the key obtained, perform the following steps:

1. Starting from the root of the corresponding trie, when the first bit of K_{S_Q} is zero, both the left and right child are visited; otherwise, only the right child will be visited.
2. Repeat the previous step for each bit in K_{S_Q} until the external nodes are reached. Similar to signature insertion, one bit is eliminated from the key each time we go down one level.
3. All the pages reached from the external nodes found in the previous step contain qualified signatures and should be retrieved for further comparisons.

6 Conclusion

In this paper, we present a new way of organizing a signature file so that an exhaustive search on the signature file can be avoided. Our approach is based on partitioning schemes which divide the signature file into partitions in such a way that each partition contains signatures with the same key. The keys can be examined first before the partitions are actually searched. As shown in our study, many partitions can be rejected simply by examining the keys. We describe three partitioning schemes. The data structure necessary to support the schemes and the algorithms for the insertion, deletion, and search of the signatures are described in detail. We have shown that our approach achieves a substantial reduction in the search space, with an extremely small storage and processing head.

The time complexity for searching the trie structure is of the order of the number of partitions which actually match the query. The complexity for both insertion and deletion operations are $O(k+p)$, where p is the page size, when splitting or merging happened. The data structure and algorithms are simple and adaptable to the growth of the signature file, and the same general data structure can be used in three different schemes.

Acknowledgment

The authors wish to thank J. Cunnyngham, M. McDonald, M. McGill, P.W.Y. Wong, and C. Zeidner for their helpful discussion in this project. Wong made many suggestions which improved the writing of the paper.

REFERENCES

[1] S.R. Ahuja, and C.S. Roberts, "An associative/parallel processor for partial match retrieval using superimposed codes," *Proc. 7th Annual Symp. Comput. Arch.*, France, May 1980, 218–227.

[2] S. Christodoulakis, and C. Faloutsos, "Design considerations for a message file server," *IEEE Trans. Soft. Eng.*, Vol. SE-10, No. 2, Mar. 1984, 201–210.

[3] U. Deppisch, "S-Tree: A dynamic balanced signature index for office retrieval," *Proceedings of the ACM Conference on Research and Development in Information Retrieval*, Pisa, Italy, Sept. 1986, 77–87.

[4] H. C. Du, S. Ghanta, K. J. Maly, and S. M. Sharrock, "An efficient file structure for document retrieval in the automated office environment," *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, Feb. 1987, 165–172.

[5] R. Fagin, "Extensible hashing - A fast access method for dynamic files," *ACM Trans. Database Systems*, Vol. 4, No. 3, Sept. 1979, 315–344, et al..

[6] C. Faloutsos, "Access methods for text," *ACM Computing Surveys*, Vol. 17, No. 1, Mar. 1985, 49–74.

[7] C. Faloutsos, and S. Christodoulakis, "Description and performance analysis of signature file methods for office filing," *ACM Transactions on Office Information Systems*, Vol. 5, No. 3, July 1987, 237–257.

[8] C. Faloutsos, and S. Christodoulakis, "Optimal signature extraction and information loss," *ACM Transactions on Database Systems*, Vol. 12, No. 3, Sept. 1987, 395–428.

[9] D.L. Lee, "A word-parallel, bit-serial signature processor for superimposed coding," *Proceedings of the Second International Conference on Data Engineering*, Los Angeles, Feb. 1986, 352–359.

[10] D.L. Lee, and C.-W. Leng, "Partitioned signature files: Design issues and performance evaluation," *ACM Transactions on Information Systems*, Vol. 7, No. 2, Apr. 1989, 158–180.

[11] D.L. Lee, and F.H. Lochovsky, "Text retrieval machines," in *Office Automation*, D.C. Tsichritzis, ed., Springer-Verlag, New York, N.Y., 1985, 339–375.

[12] J.L. Pfaltz, W.J. Berman, and E.M. Cagley, "Partial match retrieval using indexed descriptor files," *Communications of ACM*, Vol. 23, No. 9, Sept. 1980, 522–528.

[13] R.L. Rivest, "Partial-match retrieval algorithms," *SIAM J. Computing*, Vol. 5, No. 1, 1976, 19–50.

[14] C.S. Roberts, "Partial-match retrieval via method of superimposed codes," *Proc. IEEE*, Vol. 67, No. 12, Dec. 1979, 1624–1642.

[15] R. Sacks-Davis, A. Kent, and K. Ramamohanarao, "Multikey access methods based on superimposed coding techniques," *ACM Transactions on Database Systems*, Vol. 12, No. 4, Dec. 1987, 655–696.

[16] R. Sacks-Davis, and K. Ramamohanarao, "A two level superimposed coding scheme for partial match retrieval," *Information Systems*, Vol. 8, No. 4, 1983, 273–280.