A New Parallel Signature File Method for Efficient Information Retrieval

Jeong-Ki Kim and Jae-Woo Chang

Department of Computer Engineering, Chonbuk National University
Chonju, Chonbuk 560-756, Korea
Phone: +82-652-70-2414 (FAX: 70-2263)
E-mail : {dblab,jwchang}.nms.chonbuk.ac.kr

Abstract

The signature file method has been widely advocated as an efficient index scheme to handle new applications demanding a large volume of textual databases. Moreover, it has recently been extended to support multimedia data. In order to achieve good performance when handling multimedia data, the signature file approach has been required to support parallel database processing. Therefore, in this paper we propose a horizontally-divided parallel signature file (HPSF) method using extendible hashing and frame-slicing techniques. In addition, we also propose a heuristic processor allocation method so that we may assign a set of signatures to a given number of processors in a uniform way. To show the efficiency of HPSF, we evaluate the performance of HPSF in terms of retrieval time, insertion time, and storage overhead. Finally, we show from the performance results that HPSF outperforms the conventional parallel signature file methods regarding retrieval performance and dynamic operating measures used to combine both retrieval and insertion time.

Keywords: dynamic signature files, parallel databases, extendible hashing, frame-slicing technique, performance evaluation

1 Introduction

The signature file method has widely been advocated as an efficient access method to deal with many applications demanding a large volume of textual databases, such as library, office information and medical information systems[1, 2]. Therefore, the signature file approach has become a well-known concept for implementing associative retrieval on data files kept in a stable store. Recently, the use of signature files has been extended to support multimedia data, such as images, voice and video[3]. Many recent database management systems (DBMS) used to support multimedia data require a dynamic storage structure which performs not only retrieval operations but also insertion, deletion and update operations in an efficient manner. As a result, several dynamic signature file methods have been proposed; S-tree[4], Quick filter[5], TDSF[6], and HS file[7].

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
CIKM '95, Baltimore MD USA
© 1995 ACM 0-89791-812-6/95/11..$3.50

The *signature file* is an abstraction which acts as a filtering mechanism to reduce the number of block accesses for a query. A *signature* is a bit string formed from the terms which are used to index a record. Indexing using signature files assigns a signature to every record in the data file. Signature files typically make use of a superimposed coding technique to create a record signature[8]. When we assume that a record consists of n terms, each term is converted into a bit string, called *term signature*, using a hash function. The record signature is formed by superimposing (inclusive ORing) the n term signatures as shown in Figure 1. The number of 1's in a signature is called *weight*. To answer a query, we first examine the signature file rather than the data file, to immediately discard non-qualifying records. For this, a set of terms in a query is hashed to form a query signature in the same way used for the record signature. If the record signature contains 1's in the same positions as the query signature (i.e. record signature $\supseteq$ query signature), the record can be considered as a potential match. However, there can be a case where the record signature may qualify as a query signature, but the record itself does not satisfy the query. This is called a *false drop*[8].

H(term) : Hash function		Term signatures
H(Computer)	=	0001 1000 0000 1001
H(Communication)	=	0100 0000 1100 1000
H(Database)	=	1000 1100 0000 1000
Record signature	=	1101 1100 1100 1001

Figure 1: Record signature construction

In a database processing environment, the speed of microprocessors increases rapidly with the development of computer technology, while the performance of disks increases at a much lower rate. This leads to a major bottleneck. To solve the problem, parallel database systems using multiple disks have been developed[9, 10]. A key issue in parallelization is: how to create opportunities for parallel accessing without resorting to redundant storage or declustering data on disks. When we use parallel signature files in a multiprocessor environment, we can achieve significant increases

in speed by means of concurrent access to databases. In this paper, we propose a *horizontally-divided parallel signature file (HPSF) method* using extendible hashing and frame-slicing techniques, which is appropriate for both parallel and dynamic database environments. In addition, we propose a heuristic processor allocation method so that we may assign a set of signatures to a given number of processors in a uniform way.

The remainder of this paper is organized as follows. In section 2, we describe an overview of the conventional parallel signature file methods. In section 3, we propose a new dynamic signature file method, called HPSF, using extendible hashing and frame-slicing techniques. In section 4, in order to show the efficiency of HPSF, we evaluate the performance of HPSF in terms of retrieval time, insertion time, and storage overhead. In section 5, we compare the performance of HPSF with those of the conventional parallel signature file methods. Finally, section 6 offers our conclusions.

2 Conventional Parallel Signature File Methods

There are basically three different structures for multiprocessor database computers: Shared Everything(SE), Shared Disk(SD), and Shared Nothing(SN)[9]. In SE, all disk and memory modules are shared by processors, and therefore data is equally accessible from all processors. In SD, each processor can directly access any disk, but each processor has its own private memory. In SN, each processor has its own private memory and the dedicated disk devices are connected by a high speed network as illustrated in Figure 2. There has been considerable debate about which structure is the most suitable for a database computer implementation. It is generally agreed that SN is outstanding for achieving good system performance because the problem of a data coherency control does not occur. However, since SN is very sensitive to the distribution of data on disks, it may lead to considerable performance degradation due to a *data skewing*. In spite of the data skew problem, SN is the most used structure for implementing the conventional parallel signature file methods, i.e., FSF[11], CAT[12], and PBSSF[13].

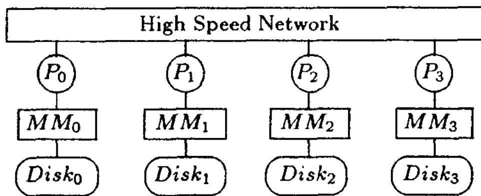

Figure 2: Parallel architecture (Shared nothing)

Lin proposed a concurrent frame signature file (CAT) as an extension of the frame-sliced signature file (FSSF). The CAT aims at allowing concurrent accesses to a signature file, and makes use of a *clustered frame scheme* to construct a signature. The clustered frame scheme selects only n frames per term among all the frames in order to create a term signature from a single term in the record. A common method used to parallelize FSSF whould be to partition the signature file horizontally into segments, which are then assigned into different processors. However, the common method's problem is that it is unable to predict where the n frames are located because a frame selection is made at run time. CAT achieves better retrieval performance than the common method because CAT needs only one random disk access per frame in each section (Figure 3). However, the CAT method has a disadvantage in that it cannot support a dynamic database environment.

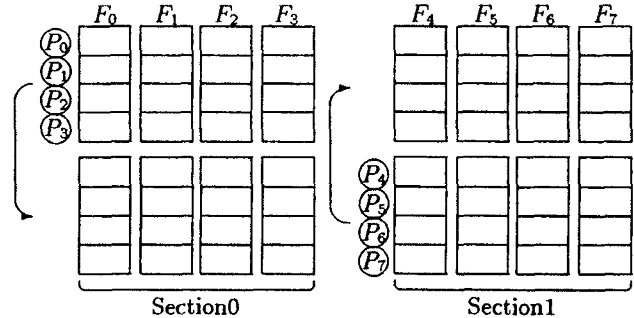

Figure 3: CAT structure

Grandi et al. proposed a fragmented signature file (FSF) which combines a quick filter[5] with the frame-slicing approach so that they could dynamically manage a large file with fast responses on user queries (Figure 4). The FSF method presents a scheme to divide a single signature into several frames, which are computed as $n \cdot 2^k \geq p$, where p, n, and 2^k represent the number of processors, frames, and partitions, respectively. Each frame is assigned to processors whose number are determined by:

$$HP(FS,k,Z_j) = \begin{cases} \sum_{l=0}^{k-1} b_{l+1}2^l & if \ \sum_{l=0}^{k-1} b_{l+1}2^l < Z_j \\ \sum_{l=0}^{k-2} b_{l+1}2^l & otherwise \end{cases}$$

where FS is the frame-slice and k is calculated as $\lfloor \log_2(p/n) \rfloor$, and $2^{k-1} < Z_j \leq 2^k$. When we make use of clustered frames, the retrieval performance of FSF becomes better as the number of frames is increased. However, when we adopt unclustered frames, the retrieval performance is better with a decrease in the number of frames.

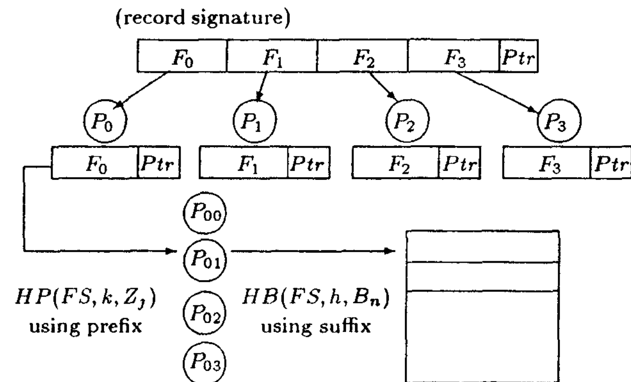

Figure 4: FSF structure

3 A Horizontally-divided Parallel Signature File Method

Based on extendible hashing and frame-slicing techniques, we propose a *horizontally-divided parallel signature file (HPSF) method*. In addition, we propose a heuristic processor allocation method which distributes both descriptor keys and their frame blocks in a uniform way so that we may make equivalent keys to be fairly assigned to a given number of processors. This leads to a better retrieval performance than conventional methods regarding database accesses as well as efficient utilization of multiple processors. Here, the ***equivalent keys*** are defined as a set of keys which are searched to find whether they potentially match the hashing key of a query signature. For example, if a hashing key is determined to be '1010', equivalent keys can be calculated from '1*1*', i.e., 1010, 1011, 1110 and 1111. Figure 5(a) shows that these equivalent keys are uniformly distributed among four processors by using a heuristic processor allocation method. Therefore, sixteen descriptor keys are assigned into four processors by means of a repeating sequence of ↓↑↑↓ directions. Four equivalent keys can be accessed by each processor. On the other hand, Figure 5(b) indicates that the four equivalent keys (1010, 1011, 1110, and 1111) are assigned into only processors 2 and 3, resulting in the inefficient utilization of the four processors. Since it is very difficult to find an optimal processor allocation method, it is reasonable to design an efficient process allocation method based on a heuristic approach.

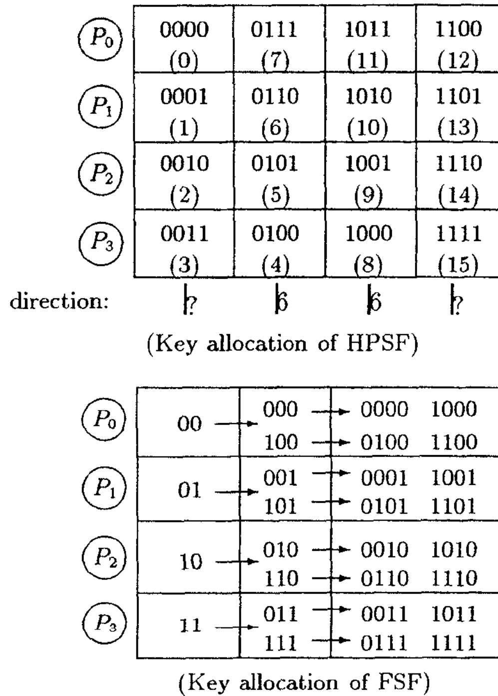

Figure 5: Allocation methods

When a collection of frame blocks is split due to the overflow of record signatures, a newly created key must be moved into the processor whose number is determined as:

$$p(K, N_p, N) = \begin{cases} K\%N_p & if\ b(K, N_p, B) = 0 \\ (K\%N_p) \oplus (N_p - 1) & if\ b(K, N_p, B) = 1 \end{cases}$$

where K is a key value calculated by an extendible hash function, N_p presents the number of processors, and $\oplus$ means a bitwise exclusive-OR operation. Also, B represents a processor allocation pattern, which is obtained by 0x96 (=10010110) in the experiment. Function $b()$ indicates an allocation direction, which is calculated as

$$b(K, N_p, B) = [B \gg ((K/N_p)\%bitsize(B))]\ \&\ 0x01.$$

When $b(K, N_p, B)$=0, a forward direction is presumed; when $b(K, N_p, B)$=1, a backward direction. For example, when K=6(=101) and N_p=4, then b(6,4,0x96) = 1. This means a backward direction and $p()$=1. Therefore, a record signature with K=6 is assigned into processor 1.

Meanwhile, each processor determines which sequence its frame blocks should be searched. In general, it is necessary to access frame blocks according to a query signature weight because query signature frames with high weights lead to a greater filtering effect by accessing them first. For example, the searching order of the query signature in Figure 6 is decided as F_1, F_3, and F_0, based on the weights of the four query frames. Meanwhile, when we access frame blocks according to a searching order, a frame block with non-qualifying signatures is excluded in the next searching sequence. The higher a query signature weight, the lower the probability of potential matching with a record signature. In addition, when we divide a query signature into several signature frames, we do not need to access frame blocks which correspond to zero-weight query frames because they qualify all the frames in frame blocks. Therefore, the zero-weight query frame of F_2 is removed from the searching order (Figure 6).

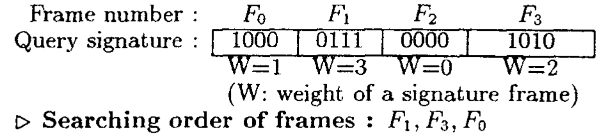

Figure 6: Searching order of query signature frames

For example, we assume to answer a query containing two terms, "computer" and "file", as shown in Figure 7. A query signature is made from the query. Next, since a hashing key in the query signature is '1010', equivalent keys are '1010, 1011, 1110, and 1111'. So P_0 finds one of the equivalent keys (i.e. '1011') in the descriptor and accesses the frame group G_{11}. According to the searching order (Figure 6), P_0 compares query signature frames with record signature frames. In this case, we can obtain the first signature as a qualifying signature and finally access the document by the *Ptr* information of the record signature. To construct the HPSF, insertion, retrieval and deletion algorithms are as follows:

[Algorithm 1] Insertion Algorithm
Input :
  R_n : new record
Output :
  U' : database structure after R_n is inserted
Variable :
  S : signature made from R_n
  P_i : i-th processor, $0 \leq i < N_p$

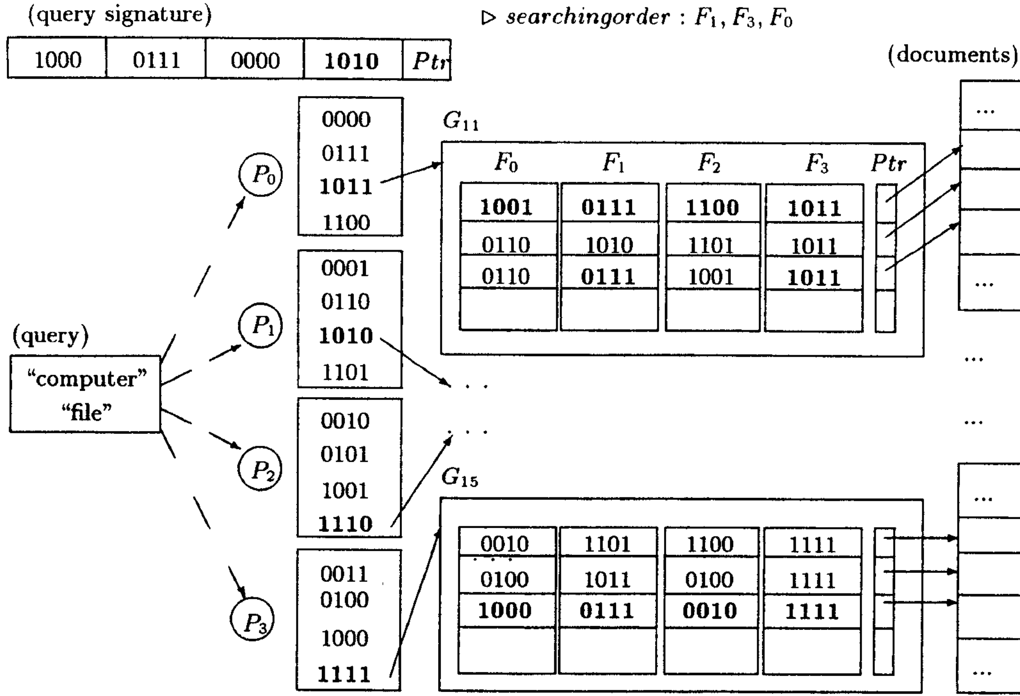

Figure 7: HPSF structure

T_s : blocking factor of a fame block
D : set of descriptor keys,
$\{d[i] | d[i] \in D, i$ is the value of descriptor key $\}$
$d[i](klen, cnt, pointer)$: key length, number of signature, pointer to frame group
$FG[i]$: group of frame blocks which is indicated by d[i].pointer

Process :

Make S from R_n and transfer S into every processor P_i;
Every processor P_i computes a hashing key KEY from S ;
for(; processor number $== p(KEY, N_p, B)$;
computer new KEY from S){
  if($d[KEY].cnt == T_s$) {   /* $FG[KEY]$ is full */
    $d[KEY].klen++$;
    $NEXT = KEY + 2^{d[KEY].klen}$;
      /* $NEXT$ is a new descriptor key value */
    $d[NEXT].pointer = \&(FG[NEXT])$
    and $d[NEXT].klen = d[KEY].klen$;
    Classify all the frames of $FG[KEY]$
    into $FG[KEY]$ and $FG[NEXT]$;
    Move $d[NEXT]$ and $FG[NEXT]$
    into new $p(NEXT, N_p, B)$ processor;}
  else {   /* $d[KEY].cnt < T_s$ */
    Divide S into each frame ;
    Store frames in every frame block of $FG[KEY]$;
    $d[KEY].cnt$ ++ ; break; }
} /* for loop */

End :

[Algorithm 2] Retrieval Algorithm

Input :

R_q : query

Output :

C : set of records retrieved from a query

Variables :

Q : query signature
QF_i : i-th frame of query signature, $0 \leq i < N_f, Q \supseteq QF_i$
$O[i]$: searching order of query frames
N_f : number of frames
$FB[i, j]$: j-th frame block in $FG[i], 0 \leq i < N_f$
K : set of equivalent key, $\{k_i | k_i \in K, k_i \in D\}$

Process :

Make Q from R_q and transfer Q into every processor P_i;
Divide Q into N_f query frames, i.e. QF_i ;
Compute $O[i]$, for example, if i-th frame weight is zero, $O[i] = -1$;
Every processor P_i computes a hashing key KEY from Q;
Every processor P_i computes all equivalent keys $k_i \in K$ from KEY;
while($k_i \in K$ exists in a processor whose number is $p(k_i, N_p, B)$) {
  for($j = 0; j < N_f; j++$) {
    if($O[j] == -1$) continue;
    Read $FB[i, O[j]]$ of $FG[i]$ which is pointed by k_i;
    Find record signature frames matched with $QF_{O[j]}$ in $FB[i, O[j]]$;
    if(all frames in $FB[i, O[j]]$ are never matched with $QF_{O[j]}$)
      $O[j] = -1$; }
  Read C which is pointed by matched signatures;
  while($(r \in C) \& (r \supset R_q)$) output r ;
} /* while loop */

End :

[Algorithm 3] Deletion Algorithm

Input :

R_d : record to be deleted

Output :

U'' : database structure after R_d is deleted

Variables :

S_d : record signature to be deleted

Process :

Make S_d from R_d and transfer S_d into every processor P_i;
Every processor P_i computes a hashing key KEY from S_d ;
Find S_d using [Algorithm 2] ;
Delete all frames of S_d in $FG[KEY]$ and its record ;

$$OPKEY = \begin{cases} KEY + 2^{d[KEY].klen-1} & if b(KEY, klen) = 0 \\ KEY - 2^{d[KEY].klen-1} & if b(KEY, klen) = 1 \end{cases}$$

for($d[KEY].cnt--; d[KEY].cnt + d[OPKEY].cnt \leq T_s;$ $d[KEY].klen--$) {
  if($OPKEY < KEY$) Exchange $OPKEY$ with KEY;
  Merge $FG[KEY]$ and $FG[OPKEY]$ into $FG[KEY]$;
  $d[KEY].cnt = d[KEY].cnt + d[OPKEY].cnt$;
  Delete $d[OPKEY]$ and $FG[OPKEY]$;
  Compute again an opposite key $OPKEY$ from KEY;
}

End:

4 Performance Analysis

In this section, we develop an analytic performance model of the HPSF method. For this, Table 1 lists the input and design parameters as well as their meanings. The input parameters describe the database used in a given application and the design parameters are chosen by a designer to evaluate the performance of HPSF. We first define false drop probability, F_d, as the probability that a nonqualified signature is accidentally considered as a qualified one. Therefore, the false drop probability is computed as follows:

$$F_d = \frac{M_f}{N - M_a}$$

Here N is the number of signatures, M_f is the number of false match signatures, and M_a is the number of actual match signatures. When a signature length is relatively large, F_d can be computed as:

$$F_d \approx \lim_{b \to \infty} \left[1 - \left(1 - \frac{k}{B}\right)^m\right]^k = \left(1 - e^{-\frac{k \cdot m}{b}}\right)^k$$

Therefore, we can gain values k and b based on the fact that the optimal distribution of a signature is achieved when t/b is 1/2 [8].

4.1 Retrieval Time

To estimate retrieval performance, we assume that a query contains w terms and a record signature follows a uniform distribution. The retrieval time is measured by the number of block accesses when retrieving a query with w terms. For the analysis, we define $e(r, m, n)$ as a probability that any one of n blocks includes any one of r records when r is the number of randomly selected records and m is a blocking factor[14]. We make use of the following measures for searching both signature files and qualifying records.

Table 1: Input and design parameters

Parameters	Meanings
b	Signature length
m	Number of distinct words in a record
k	Weight of a word signature (i.e. bits being set by one word)
L_d	Length of a descriptor key
N	Number of records
G	Size of a disk page(block)
N_p	Number of processors
N_f	Number of frames
L_f	Frame size
α	Proportion of page utilization
T_s	Blocking factor of a frame block
w	Number of words in a query

$P_w(S), P_w(Q)$: Weight ratio of record and query signatures respectively,
$1 - \left(1 - \frac{k}{b}\right)^m, 1 - \left(1 - \frac{k}{b}\right)^w$.

$W(S), W(Q)$: Record signature weight and query signature weight,
$P_w(S) \cdot b,\ P_w(Q) \cdot b$.

$W(S_i), W(Q_i)$: i-th frame weight of record and query signatures respectively,
$P_w(S) \cdot L_f,\ P_w(Q) \cdot L_f$.

$P_{w=j}(Q, h)$: Proportion of h bits with j weight in query signatures,
$\left\{\binom{b-h}{W(Q)-j}\binom{h}{j}\right\} \Big/ \binom{b}{W(Q)}$.

$E(W(Q, h))$: Expected weight of h bits in query signatures,
$\begin{cases} P_w(Q) \cdot h & if\ W(Q) \ll b \\ \sum_{j=0}^{min(h, W(Q))} j \cdot P_{w=j}(Q, h) & otherwise. \end{cases}$

$K(h)$: Number of equivalent keys when a hashing key has h bits in query signatures,
$2^{h - E[W(Q,h)]}$.

$V(K, \beta)$: Number of equivalent keys in i-th processor, $K(h) \cdot \beta$. When the equivalent keys are uniformly distributed among N_p processors, β is $1/N_p$.

$P_m(L_f, W(S_i), W(Q_i))$: Probability that one of the signature frames in i-th frame blocks is matched with a query signature where i is a searching order,
$\begin{cases} 2^{-W(Q_i)} & if\ L_f \gg W(Q_i)\ and\ \frac{W(S_i)}{L_f} = \frac{1}{2} \\ \binom{W(S_i)}{W(Q_i)} \Big/ \binom{L_f}{W(Q_i)} & otherwise. \end{cases}$

M_i : Number of qualifying signature frames in i-th frame blocks,
$M_{i+1} = M_i \cdot P_m(L_f, W(S_i), W(Q_i)),\ M_1 = V(K, \beta) \cdot T_s \cdot \alpha$.

$R_f(i)$: Number of i-th frame blocks which is accessed for query signatures, $R_f(i+1) = e(M_i, T_s \cdot \alpha, R_f(i)) \times R_f(i)$, $R_f(1) = V(K, \beta)$.

According to these measures, we can calculate the retrieval time of HPSF:

$$R_{HPSF} = K(h) + \sum_{i=1}^{N_f} R_f(i).$$

4.2 Storage Overhead

In addition, we make use of the following measures for estimating storage overhead to maintain signature files in all processors.

O_{sf} : Storage space for maintaining frame blocks to store signature frames, $\frac{N \, N_f}{T_s \cdot \alpha} + \frac{N}{T_P \, \alpha}$ where $T_s = \frac{G}{L_f}, T_p = \frac{G}{L_p}, L_p$ is the length of a pointer.

O_{df} : Storage space for maintaining descriptor files to store descriptor keys, $2^h \cdot L_d$.

According to these measures, we can calculate the storage overhead of HPSF:

$$O_{HPSF} = O_{sf} + O_{df}.$$

4.3 Insertion Time

For the analysis, we use the following measures for inserting record signatures.

I_{Hfb} : Number of frame block accesses when not split, N_f

P_{sp} : Probability that frame blocks are split where $1/T_s$ is a proportion that a frame is split when inserting from the first signature to the T_s-th signature, $\frac{1}{N}(1 + (N - T_s)\frac{2}{T_s}) = \frac{2}{T_s} - \frac{1}{N}$.

I_{Hfsp} : Number of additional frame block accesses when split, $N_f \cdot P_{sp}$. These frame blocks are moved to another processor through the network.

I_{Hdb} : Number of descriptor block accesses, $1+P_{sp}$.

According to these measures, we can calculate the insertion time of HPSF:

$$I_{HPSF} = I_{Hfb} + I_{Hfsp} + I_{Hdb} + \gamma,$$

where γ is an additional time which is required in a network to move the split frame blocks.

5 Experimental Performance Results

In order to verify the analytic model to evaluate the performance of our HPSF method, we implemented the three parallel signature file methods (i.e. CAT, FSF, and HPSF) and ran experiments on a set of 100 thousand records under the UNIX/SUN Sparc 10 environment. The input and design parameters for the experiments are presented in Table 2. In the table, HPSF uses 16 frames and FSF uses 1 frame. Meanwhile, the records used for experiments were collected from the library of Chonbuk National University (CNU), which consist of 12 fields such as title, author, school, degree, date, page, adviser, source, publication, subject, abstract, and keywords. The average record size is about 2.5 Kbytes with 87.8 distinct terms. We also used 4,000 conjunctive queries so as to make a wide range of experiments. For the experiment, we assume that a communication delay is much shorter than the time for a disk block access because we make use of a high-speed network among processors[12, 13].

Table 2: Experimental values

Parameters	Values
b	276 bytes
m	87.8
k	17
N	100,000
G	4096 bytes
N_p	32
N_f	16 (FSF:1)

According to the parameters and assumptions, Figure 8 presents both the theoretical and the experimental retrieval performance of HPSF. From the results, we can see that the analytic and experimental results correlate very well. In addition, Figure 9 shows the retreval time of the three parallel signature file methods. The HPSF achieves a considerably better retrieval performance than FSF and CAT in the entire range. In Figure 9, a cross point between HPSF and CAT occurs for the following reason: when a query has a small number of terms, the HPSF basically owns more frame blocks than CAT because the page utilization of HPSF is lower than that of CAT. On the other hand, when a query has a large number of terms, the HPSF can remove many non-qualifying descriptor keys while CAT must always access all the frames in frame blocks. However, CAT shows the worst performance on the insertion time as shown in Table 3. Therefore, it is difficult to apply CAT to dynamic environments where insertion operations occur frequently. Meanwhile, the false-drop probability of clustered frame signatures used in CAT is about 1000 times higher than that of unclustered frame signatures used in HPSF (Figure 10).

Table 3: Insertion time and Storage overhead

Methods	Insertion time (second)	Storage overhead (%)
CAT	0.30	11.04
FSF	0.03	16.31
HPSF	0.01	15.35

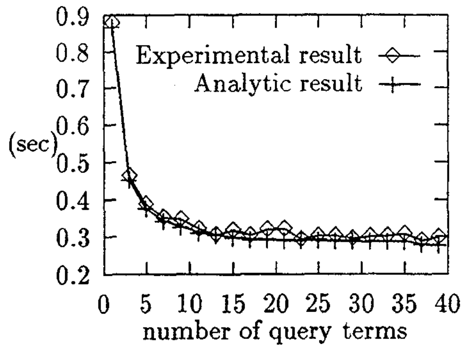

Figure 8: Retrieval time of HPSF

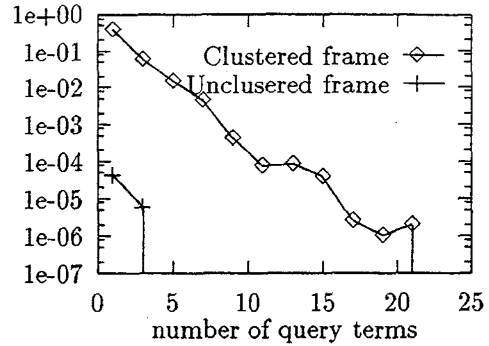

Figure 10: False-drop probabilities

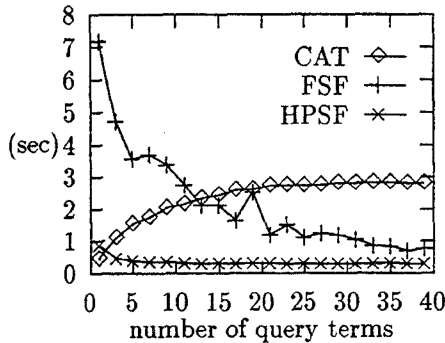

Figure 9: Retrieval time of 3 methods

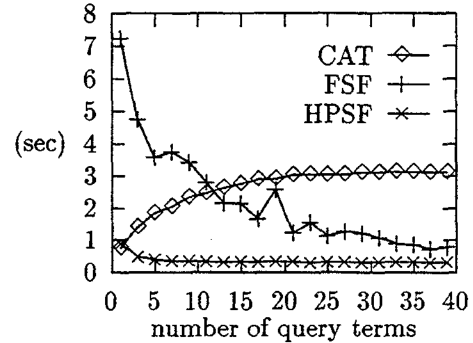

Figure 11: Dynamic operating measures

For a dynamic environment where insertion operations frequently occur, we define a *dynamic operating measure* as $C = R + \delta \cdot I$ which combines two major output parameters (i.e. the insertion and retrieval time). In the formula, δ is a *dynamic degree* indicating the relative weight (occurrence frequency) of data insertion, compared with retrieval. When $\delta = 1$ (i.e. the rate of insertion is the same as that of retrieval), the HPSF achieves the best performance gains with regard to the dynamic operating measure, compared with the CAT and the FSF (Figure 11). Therefore, we can make use of HPSF as an efficient dynamic storage structure for a multiprocessor environment.

6 Conclusion

The signature file method has been widely advocated as an efficient index scheme able to handle many applications demanding a large volume of textual databases. In order to achieve improved performance, the signature file approach has recently been required to support parallel database processing. In this paper, we proposed a horizontally-divided parallel signature file (HPSF) method using extendible hashing and frame-slicing techniques. In addition, we designed a heuristic processor allocation method to assign a set of signature frames into multiple processors in a uniform way. To show the efficiency of HPSF, we presented an analytic model and evaluated the performance of HPSF in terms of the retrieval time, insertion time, and storage overhead. From the performance results, we demonstrated that our HPSF method significantly outperformed FSF on retrieval time and was better than CAT regarding dynamic operating measures combining retrieval time and insertion time.

References

[1] J.W. Chang, J.H. Lee, and Y.J. Lee. "Multikey Access Methods Based on Term Discrimination and Signature Clustering". In *Proc. of 12th Ann Int'l SIGIR of ACM*, pages 176–185, USA, June 1989.

[2] W.W. Chang and H.J. Schek. "A Signature Access Method for the Starbust Database System". In *Proc. of the 15th VLDB Conference*, pages 145-153, Netherland, Aug 1989.

[3] P.B. Berra et al. "Architecture for Distributed Multimedia Database Systems". *Computer Communications*, 13(4):217–231, May 1990.

[4] U. Deppish. "S-tree: A Dynamic Balanced Signature Indexed for Office Retrieval". In *Proc. of the ACM Conf. on RDIR*, pages 77–87, Sept 1986.

[5] P. Zezula, F. Rabitti, and P. Tiberio. "Dynamic Partitioning of Signature Files". *ACM Trans. on OIS*, 9(4):336–369, Oct 1991.

[6] J.K. Kim and J.W. Chang. "A Two-dimentional Dynamic Signature File Method". In *Proc.of Int'l Symp. on ADTI*, pages 63–70, Nara, Japan, Oct 1994.

[7] J.S. Yoo et al. "A Dynamic Signature File Method for Efficient Information Retrieval". In *Proc. of Int'l Symp. on NGDSTA*, pages 108–115, Japan, Sept 1993.

[8] C. Faloutsos and S. Christodoulakis. "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation". *ACM Trans. on OIS*, 2(4):267–288, 1984.

[9] K.A. Hua and C. Lee. "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning". In *Proc. of the 17th VLDB Conference*, pages 525–535, Spain, Sept 1991.

[10] J. Li, J. Srivastava, and D. Rotem. "CMD: A Multidimensional Declustering Method for Parallel Database Systems". In *Proc. of the 18th VLDB Conference*, pages 3–14, Canada, 1992.

[11] F. Grandi, P. Tiberio, and P. Zezula. "Frame-Sliced Partitioned Parallel Signature Files". In *Proc. of 15th Ann Int'l SIGIR of ACM*, pages 286–297, Denmark, June 1992.

[12] Z. Lin. "Concurrent Frame Signature Files". *Distributed and Parallel Databases*, 1(3):231–249, July 1993.

[13] G. Panagopoulos and C. Faloutsos. "Bit-Sliced Signature Files for Very Large Text Databases on a Parallel Machine Architecture". In *Proc. of 4th Int'l Conf. on EDT*, pages 379–392, United Kingdom, 1994.

[14] K. Y. Whang, G. Wiederhold, and D. Sagalowicz. "Estimating Block Accesses in Database Organizations - A Closed Noniterative Formula". *Communication of ACM*, 26(11):940–944, Nov 1983.