NORTH-HOLLAND

Climbing Hashing for Expansible Files[*]

YE-IN CHANG

Department of Applied Mathematics, National Sun Yat-Sen University, Kaohsiung, Taiwan, Republic of China

and

CHIEN-I LEE

Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China

ABSTRACT

The goal of dynamic hashing is to design a function and a file structure that allow the address space allocated to the file to be increased and reduced without reorganizing the whole file. In this paper, we propose a new dynamic hashing scheme called *climbing hashing*, which requires no index and has the growth of a file at a rate of $\frac{n+1}{n}$ per full expansion, where n is the number of pages of the file, as compared to a rate of two in linear hashing. In climbing hashing, when a split occurs, the relative position of the new page (into which a data record may move), to the current page (where the data record is now), is proportional to the number of full expansions. Therefore, it seems like the data record is climbing in the files. (Note that a *level* is defined as the number of full expansions that have happened thus far.) From our performance analysis, given a fixed load control, the proposed scheme can achieve nearly 96% storage utilization as compared to 78% storage utilization by using linear hashing, which is also verified by a simulation study. Moreover, the proposed scheme can be generalize to have the growth of a file at a rate of $\frac{n+t-1}{n}$ per full expansion, where t is an integer larger than 1. As t is increased, the average number of overflow pages per home page is reduced, resulting in a decrease of the average number of disk accesses for data retrieval.

1. INTRODUCTION

The goal of dynamic hashing is to design a function and a file structure that can adapt in response to large, unpredictable changes in the number

[*]This work was supported in part by the National Science Council of China under grant NSC-82-0408-E-110-135.

0020-0255/95/$9.50
SSDI 0020-0255(95)00048-T

and distribution of keys while maintaining a fast retrieval time [1]. That is, the address space allocated to a file can be increased and reduced without reorganizing the whole file. Over the past decade, many dynamic hashing schemes have been proposed. These dynamic hashing schemes can be divided into two classes: one needs an index, the other one does not need an index. Extendible hashing [3, 12] and dynamic hashing [4, 17] belong to the first class. Linear hashing [2, 5–7, 9–11, 13–16] belongs to the second class.

Among these dynamic hashing schemes, linear hashing dispenses with the use of an index at the cost of requiring overflow space. The first linear hashing scheme was proposed by Litwin [11]. In linear hashing, a file is expanded by adding a new page at the end of the file when a split occurs and relocating some of the data records in the split page to the new page by using a new hashing function. To maintain stable performance through file expansions in linear hashing, many strategies have been proposed. Among these strategies, linear hashing with partial expansions as first presented by Larson [5, 7] is a generalization of Litwin's linear hashing [11]. This method splits a number of *buddy* pages together at one time, and the data records in each of those buddy pages are redistributed into the related old page and the new added page.

In this paper, we propose a generalized approach for designing a class of dynamic hashing schemes that require no index and have the growth of a file at a rate of $\frac{n+1}{n}$ per full expansion, where n is the number of pages of the file, as compared to a rate of two in linear hashing. Since the growth rate of the proposed approach is smaller than that of linear hashing, the proposed approach can maintain more stable performance through file expansions and better storage utilization than linear hashing. Based on this generalized approach, we derive a new dynamic hashing scheme called *climbing hashing*. In climbing hashing, when a split occurs, the relative position of the new page (into which a data record may move), to the current page (where the data record is now), is proportional to the number of full expansions. Therefore, it seems like the data record is climbing in the files. From our performance analysis, given a fixed load control, climbing hashing can achieve nearly 96%, as compared to 78% storage utilization using linear hashing, when the keys are uniformly distributed. (Note that a load control denotes the upper bound of the number of new inserted records before the next split can occur.) Climbing hashing can have even much better storage utilization than linear hashing, when the key are not uniformly distributed. Moreover, the proposed scheme can be generalized to set the growth of a file at a rate of $\frac{n+t-1}{n}$ per full expansion, where t is an integer larger than 1. As t is increased, the average number of overflow pages per home page is reduced, resulting in a decrease of the average number of disk accesses for data retrieval (while also decreasing storage utilization).

The rest of the paper is organized as follows. Section 2 presents the generalized approach and climbing hashing. Section 3 gives descriptions of climbing hashing. Section 4 presents the performance analysis of climbing hashing. Section 5 discusses the simulation results of climbing hashing, linear hashing, and linear hashing with partial expansions. In Section 6, we generalize climbing hashing to have the growth of a file at a rate of $\frac{n+t-1}{n}$ per full expansion. Finally, Section 7 contains a conclusion.

2. A CLASS OF DYNAMIC HASHING SCHEMES

In this section, we first present a generalized approach for designing a class of dynamic hashing schemes. Next, we derive a new dynamic hashing scheme based on this approach.

2.1. *THE GENERALIZED APPROACH*

In a dynamic hashing scheme without using an index, the data records are stored in chains of pages linked together. A page *split* occurs under certain conditions, for example, whenever the number of records exceeds a positive integer value denoted by L. Let each key be mapped into a string of binary bits b_i first, i.e., $H(\text{key}) = (b_{q-1}, \ldots, b_1, b_0) = c$. Then, this scheme addresses records by using a series of *split functions*, $h_0, h_1, \ldots, h_q$, where each function h_i maps c to a nonnegative integer. Let a split pointer sp point to the next page to be split, and initially, split pointer sp points to page 0. A *full expansion* occurs when a split occurs at a page next to which is a new added page [11]. A *level* is defined as the number of full expansions that have happened thus far. For each level d, h_d or h_{d+1} is used to locate a page depending on whether $h_d(c) \geq sp$ or not. On each level d, the pages are split in the order from page 0 to the maximum index of pages on that level. After all the pages on the current level d have been split, i.e., after a full expansion, the value of level d is increased by 1 and the splitting process starts again from page 0.

Based on the above strategy to handle file expansions, we can give a class of dynamic hashing schemes with a growth rate of $\frac{n+1}{n}$ per full expansion by defining the relationship among h_i in the following way. Let $h_0(c)$ be the function to load the file initially and $h_0: c \rightarrow \{0, \ldots, s_0 - 1\}$, where s_0 is the number of pages of the file initially. Let $w(i)$ be a function with $w: i \rightarrow Z - \{0\}$, where Z denotes the set of integer numbers. (Note that $w(i)$ denotes the distance from the current page $h_i(c)$ to the new page $h_{i+1}(c)$. The rest of the split functions, $h_1, h_2, \ldots, h_i$, are defined as follows:

$$h_0(c) = c \bmod s_0;$$

$$h_{i+1}(c) = (h_i(c) + w(i)b_i) \bmod (s_0 + i + 1),$$

for $i \geq 0$, where b_i is the value of the ith bit of c; that is, $0 \leq h_{i+1}(c) \leq i + 1 + h_0(c)$.

From the above definitions of the relationships between functions h_{i+1} and h_i, where $i \geq 0$, the address space returned from function h_{i+1} is in the set of $\{0, 1, \ldots, s_0 + i\}$; that is, the file size s_{i+1} on level $(i+1)$ is (s_0+i+1). Consequently, the growth rate of a file is $\frac{n+1}{n}$ per full expansion, where n is the number of pages of the file.

For example, given $s_0 = 1$ and $w(i) = 1$, we have $h_{i+1}(c) = h_i(c) + b_i$. In this case, when an insertion causes a split on level i and $sp = k$, $0 \leq k < i$, the data records in page k will be redistributed to page k or page $(k + 1)$ according to whether the value of bit b_i is 0 or 1, respectively, i.e., according to the value of $h_{i+1}(c)$. When a split occurs on level i and $sp = i$, i.e., sp has pointed to the maximum index of pages, then a new page $(i + 1)$ is added at the end of the file and the data records in page i are redistributed to page i or page $(i + 1)$ according to whether the value of bit b_i is 0 or 1, respectively.

2.2. *CLIMBING HASHING*

Based on the above proposed generalized approach, now we derive a specific dynamic hashing scheme called *climbing hashing*. Let $s_0 = 1$ (i.e., $h_0(c) = 0$) and $w(i) = i$, then

$$h_0(c) = 0,$$

$$h_1(c) = h_0(c) + b_0,$$

$$h_{i+1}(c) = (h_i(c) + ib_i) \bmod (i + 2),$$

for $i \geq 1$, that is, $0 \leq h_{i+1}(c) \leq i + 1$.

In general, when an insertion causes a split and $sp = k (k \leq 1)$ on level $d (> 0)$, the data records in page k will be redistributed into page k or page $(k + d)$, according to whether the value of bit b_d is 0 or 1, respectively, as shown in Figure 1(a). Note that $0 \leq h_d(c) \leq d$, i.e., there are at most $(d + 1)$ pages in the system when the current level $= d$. When a split occurs in page $k (2 \leq k \leq d)$, where $(k + d)$ has exceeded the maximum index of pages on level $d + 1$ (i.e., $(d + 1)$), data records in page k will be redistributed into page $(k - 2)$ [$= (k + d) \bmod (d + 2)$] or still stay in page

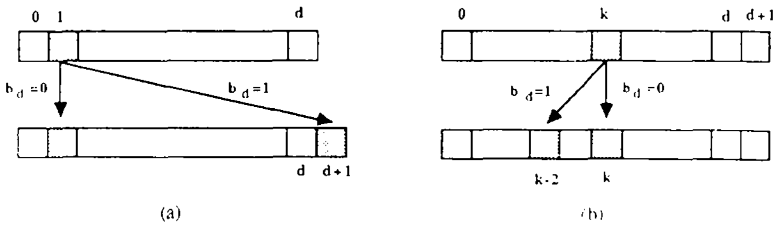

Fig. 1. A splitting operation in climbing hashing.

k, according to whether the value of b_d is 1 or 0, respectively, as shown in Figure 1(b).

3. THE ALGORITHMS

In this section, we give descriptions of address computation, retrieval, insertion, file split, and file contraction algorithms. In these algorithms, the following variables are used globally: (1) b: the size of a home page in terms of the number of records; (2) w: the size of an overflow page in terms of the number of records; (3) sp: the split pointer with an initial value = 0; (4) d: the level with an initial value = 0.

3.1. ADDRESS COMPUTATION

Let function H(key) map a key into random binary bit patterns of length q, for q sufficiently large. Let function $b_i(c)$ return the value of the ith bit of the binary pattern, which is denoted by c (= H(key)). To compute the final home page number after d full expansions, function *home-address* is shown in Figure 2. In this function, initially, all the data records are mapped into page 0 by $h_0(c) = 0$ and hence, *address* = 0. Then, the for-loop

```
function home_address(key) : integer;
    var c : integer;                        /* = H(key) */
        i : integer;                        /* an index */
        address : integer;
begin
        c = H(key)
        address = 0;            /* i.e., $h_0(c)$ */
        if $d$ > 0 then address = address + $b_0$;
        for i = 1 to (d-1) do
                        address = (address + i × $b_i$(c)) mod (i + 2);
        if address < sp then address = (address + d × $b_d$(c)) mod (d + 2);
        return (address);
end;
```

Fig. 2. Function *home_address*.

statement traces the home page number (denoted as *address*) through d full expansions. For the unfinished $(d+1)$th full expansion, a page may have been split or not. Depending on whether or not $address < sp$, the final home page number is determined.

3.2. OVERFLOW HANDLING AND RETRIEVAL

In [10], Larson applied *separators* [8] for home pages to linear hashing to guarantee that any data record can be retrieved in one disk access, where overflow records are distributed among the home pages. This method, *separators*, is based on hashing and makes use of a small in-core table, for each home page if needed, to direct the search. To understand what a *separator* is, let us define a *probe sequence* first [10]. Assume that all of the data records are stored in an external file consisting of n pages, and each of those n pages has a capacity of b records. For each data record with key $= K$, its *probe sequence*, $p(K) = (p_1(K), p_2(K), \ldots, p_n(K))$, $(n \geq 1)$, defines the order in which the pages will be checked when inserting or retrieving the record. For each data record with key $= K$, its *signature sequence*, $s(K) = (s_1(K), s_2(K), \ldots, s_n(K))$, is a q-bit integer. When a data record with key $= K$ probes page $p_i(K)$, the *signature* $s_i(K)$ is used, $1 \leq i \leq n$. Implementation of $p(K)$ and $s(K)$ are discussed in detailed in [8]. Consider a home page j to which r, $r > b$, records hash. In this case, at least $(r-b)$ records must be moved out to their next pages in their *probe sequences*, respectively. Only at most b records are stored on their current *signatures*, and records with low *signatures* are stored on the page whereas records with high *signatures* are moved out. A *signature* value that uniquely separates the two groups is called a *separator*, and is stored in a *separator table*. The value stored is the lowest *signature* occurring among those records that must be moved out. (Note that a *separator table* has two entries: one is a *separator* value and the other one is a pointer to a page.)

Since in [10] overflow records are distributed among the home pages, the costs of file-split, insertion, and maintaining *separators* will be expensive. To avoid this disadvantage and efficiently search a data record stored in overflow pages, climbing hashing also applies *separators*, but only for overflow pages. To apply *separators* to handle overflow pages in climbing hashing, we need the following modification. Assume that for each home page i, its overflow records are stored in an external file consisting of m pages, and that each of these m pages has a capacity of w records. For each overflow record of home page i with key $= K$, let its *probe sequence* be $p_i(K) = (p_{i1}(K), p_{i2}(K), \ldots, p_{im}(K)) = (1, 2, \ldots, m)$, $m \geq 1$. (Note that to increase storage utilization, we probe overflow page j until

```
function retrieval(key) : pointer;
    var i, j : integer;
begin
        i = home_address(key);
        if data record is found in page i then return( physical_address(i) );
        /* function physical_address returns the actual physical address of home page i */
        else
        begin
                for each entry j in the separator table i do
                begin
                        if $s_{ij}$(key) < $separator_{ij}$↑.value then
                        begin
                                if data record is found in page pointed by $separator_{ij}$↑.pointer
                                then return ($separator_{ij}$↑.pointer)
                                else return (nil);
                        end;
                end;
                return (nil);        /* nil denotes that the record is not found */
        end;
end.
```

Fig. 3. Function *retrieval*.

overflow page $(j-1)$ is full when a data record is inserted.) For each overflow record of home page i with key $= K$, let its *signature sequence* be $s_i(K) = (s_{i1}(K), s_{i2}(K), \ldots, s_{im}(K))$. When an overflow record of home page i with key $= K$ probes page $p_{ij}(K)$, the *signature* $s_{ij}(K)$ is used, $i \le j \le m$. By using *separators* and the above modification, any data record can be found in at most two disk accesses.

As a file grows, the total size of *separator tables* of all the home pages (which have overflow pages) can be too large to be loaded into main memory at the same time. Moreover, to reduce the number of disk accesses for loading a *separator table* for a certain home page that has overflow pages, we store a *separator table* in each home page. A *separator table* is loaded into main memory whenever its related home page is read into main memory, and it is written back to the disk whenever its home page is written back to the disk. In the case that there is no change for the data records in the home page but a data insertion/deletion has caused data record movements between overflow pages, the related home page still should be written back to the disk before it is removed from main memory. That is, one more disk access is needed in this case, since the contents of the *separator table* has been changed. Therefore, we still can guarantee that the cost of data retrieval is at most two disk accesses. As shown in Figure 3, the function *retrieval*(key) is used to locate the actual physical address (either in a home page or one of its related overflow pages), where $separator_{ij}$, $1 \le j \le m$, represents the *separator* for the jth overflow page of home page i.

In this function, home page i is searched first, which is one disk access. If the data record cannot be found in home page i, its overflow pages are tried by using *separators*. If the data record exists in those overflow pages, one more disk access is needed; otherwise, 0/1 more disk access is needed. Therefore, at most two disk accesses are needed.

3.3. INSERTION AND FILE SPLIT

When a data record is inserted, its home page is searched first. If the size of its home page has exceeded the page size b, then one of its related overflow pages is searched according to its *probe sequences*. In the case that a data record insertion causes relocations of some other records in overflow pages, related *separators* that are stored in the home page may also have to be updated. In this case, one more disk access is needed to write the home page back to the disk, since the *separator table* is included in the home page.

Whenever the growth of a file exceeds a split control condition, a split occurs. In this case, data records in page sp (including its overflow pages) have to be redistributed to page sp or page $((sp+d) \bmod (d+2))$, according to whether the value of b_d is 0 or 1, respectively. If $sp = d$, d is increased by 1 and sp is reset to 0. The results of the above actions are equal to updating sp (and d) first and then reinserting those data records that are in the page where the old sp points by using the new hashing function h_{d+1}. The description of procedure *file_split* is shown in Figure 4. (Note that to reduce the number of disk accesses, we use a buffer mechanism to reduce the overhead of reinsertion.)

3.4. FILE CONTRACTION

Whenever the number of deletions of a file drops below a control condition, a contraction occurs. In climbing hashing, we collect the data records that are stored in page $(sp - 1)$ and page $((sp - 1 + d + 1) \bmod (d + 2))$ back to page $(sp - 1)$, when $sp > 0$ and level $= d$. If $sp = 0$ and level d, we collect the data records that are stored in page $(d - 1)$ and page $((d - 1 + d) \bmod (d + 1))$ $(=d - 3)$ back to page $(d - 1)$. The description of procedure *file_contraction* is shown in Figure 5.

4. PERFORMANCE ANALYSIS

In all dynamic hashing schemes without using an index, a split occurs under a certain condition. There are two kinds of strategies [1, 11]: uncontrolled and controlled splitting. Uncontrolled splitting means that a split

```
procedure file_split();
    var i, j : integer;
        B : buffer;
begin
        read home page sp and its overflow pages into buffer B
        and release these pages from the disk
        sp := sp + 1;
        if sp > d then
        begin
                sp := 0;
                d := d + 1;
        end;
        for each record with key = K in buffer B do
        begin
            i := home_address(K);
            if home page i is not full then
                write this record to home page i
            else
            begin
                find an entry j in separator table $s_i$ such that [illegible](K) < separator$_{ij}$.value do
                begin
                    if the page pointed by separator$_{ij}$.pointer is full then
                        move out the record whose key is separator$_{ij}$.value to Buffer B;
                    write the data record with key = K to the overflow page pointed
                    by separator$_{ij}$.pointer;
                    updated separator$_{ij}$.value if necessary;
                end;
            end;
        end;
end;
```

Fig. 4 Procedure *file_split*.

occurs whenever a collision occurs. In controlled splitting, a split occurs when the number of inserted data records exceeds a load control (L), or when storage utilization exceeds a load factor (A), $0 < A < 1$. (Note that a load control denotes the upper bound of the number of new inserted records before the next split can occur, and a load factor is a storage utilization threshold.) In general, the controlled strategy can provide better storage utilization than the uncontrolled strategy, which is verified in [11]. Moreover, when the load factor is used as the split control strategy, the system will suffer more unstable performance during a full expansion, as stated in [5, 15]. Therefore, we prefer to use the load control as the split control strategy, as in [15, 16].

In this section, we present the performance analysis of climbing hashing under the split control of the load control L. In this performance analysis model [15], we assume that the keys for data records are distributed uniformly and independently to each other, and that the page size is measured in terms of number of record slots. The size of a home page is denoted by

```
procedure file_contraction();
   var i, j : integer;
       B : buffer;
begin
      if sp == 0
      then read home page (d - 1) and page (d - 3) including related overflow pages
           into buffer B and release these pages from the disk
      else read home page (sp - 1) and page ((sp - 3) mod (d + 2)) including related overflow pages
         into buffer B and release these pages from the disk;
      sp = sp - 1;
      if sp < 0 then
      begin
            d = d - 1;
            sp = d;
      end;
      for each record with key = K in buffer B do
      begin
         i = home_address(K);
         if home page i is not full then
            write this record to home page i
         else
         begin
            find an entry j in separator table i such that s_{i,j}(K) < separator_{i,j}↑.value do
            begin
               if the page pointed by separator_{i,j}↑.pointer is full then
                  move out the record whose key is separator_{i,j}↑.value to Buffer B.
               write the data record with key = K to the overflow page pointed
               by separator_{i,j}↑.pointer;
               updated separator_{i,j}↑.value if necessary;
            end;
         end;
      end,
end;
```

Fig. 5. Function *file_contraction*.

b and the size of an overflow page is denoted by w. We also assume that the number of overflow pages for each home page is a minimum. In other words, if a home page has k, $k \geq 0$, overflow records, then there will be $\lceil \frac{k}{w} \rceil$ overflow pages for this home page. The overflow data records are handled by using *separators*, as stated in Section 3.2. When the search cost is computed, all records are assumed to have the same probability of retrieval.

Let s_0 be the number of pages of a file initially and N be the number of data records inserted into the file. Given N, we are able to derive information about the current state of the file, such as the number of used home pages *sp*, the average retrieval cost, and the storage utilization; that is, we can analyze these properties of a file as a function of N. The various properties that we are interested in are discussed below.

The number of splits performed is given by

$$ns(N) = 0, \qquad 0 \le N s_0 L,$$

$$ns(N) = \left\lceil \frac{N - s_0 L}{L} \right\rceil, \qquad N > s_0 L.$$

(Note that to reduce the number of splits, we assume that the first split is not started until the first s_0 pages are filled with $s_0 L$ records in this performance analysis.) Since in climbing hashing, the growth rate of a file is $\frac{n+1}{2}$ per full expansion, the number of home pages expanded (denoted by m) is given by

$$s_0 + (s_0 + 1) + \cdots + (s_0 + (m - 1))$$

$$\le ns(N) < s_0 + (s_0 + 1) + \cdots + (s_0 + m).$$

The first page will be added after *sp* scans over s_0 pages, the second page will be added after *sp* scans over $(s_0 + 1)$ pages, and so on; therefore, the mth page is added to the file after $\sum_{i=s_0}^{s_0+m-1} i$ splits. Therefore, $\frac{(m+2s_0-1)m}{2} \le ns(N)$ and $m = \lfloor \frac{\sqrt{8ns(N)+(2s_0-1)^2}-2s_0+1}{2} \rfloor$. Then, the maximum index of home pages for the file is $s(=(s_0 + m - 1))$, and *sp* is $(ns(N) - \frac{(m+2s_0-1)m}{2})$.

The load distribution for each home page is different in climbing hashing, as shown in Table 1. The value shown in the intersection position of level d and page number i is the number of records stored after d full expansions and is denoted by X_i^d, when there are 2^d data records whose keys are

TABLE 1
The Variance of the Load Distribution in Climbing Hashing[a]

Level	Page number (i)							
d	0	1	2	3	4	5	Mean	Variance
1	1	1					1	0
2	1	2	1				1.3	0.2
3	2	2	2	2			2	0
4	4	4	2	4	2		3.2	0.9
5	6	8	4	4	6	4	5.3	1.9

[a]Mean $= (1/(d+1)) \sum_{i=0}^{d} X_i^d (= (2^d/(d+1)))$.
Variance $= (1/(d+1)) \sum_{i=0}^{d} (X_i^d - \text{Mean})^2$.

uniformly distributed. Initially, we have $X_0^1 = 1$, $X_1^1 = 1$. When d is 2, the value of X_0^2 is X_0^1, the value of X_1^2 is $(X_0^1 + X_1^1)$, and the value of X_2^2 is X_1^1. Moreover, when $d > 2$, the value of $X_i^d (0 \leq i < (d - 1))$ is $(X_i^{d-1} + X_{i+2}^{d-1})$, the value of X_{d-1}^d is $(X_{d-1}^{d-1} + X_0^{d-1})$, and the value of x_d^d is X_1^{d-1}.

Let $P(sp, i, s)$ be the probability of a data record hashed into home page i after s full expansions when the split pointer points to page sp. In general, after s full expansions in climbing hashing and $sp = 0$, the probability $P(0, i, s)$ for home page $i (0 \leq i \leq (s - 1))$ is $\frac{P(0,i,s-1)+P(0,i+2,s-1)}{2}$, the probability $P(0, s-1, s)$ for home page $(s-1)$ is $\frac{P(0,0,s-1)+P(0,s-1,s-1)}{2}$, and the probability $P(0, s, s)$ for home page s is $\frac{P(0,1,s-1)}{2}$. During the $(s+1)$th full expansion, after a split occurs in home page 0 (i.e., $sp = 1$) and all the data records of home page 0 have been redistributed to home page 0 and home page s, the probability $P(1, i, s)$, $0 \leq i < s$ is $\frac{P(0,i,s)}{1+P(0,0,s)}$ and the probability $P(1, s, s)$ is $\frac{P(0,0,s)+P(0,s,s)}{1+P(0,0,s)}$. After a split has occurred in home page 1 (i.e., $sp = 2$) and all the data records of home page 1 have been redistributed to home page 1 and a new added home page (i.e., page $(s+1)$), the probability $P(2, i, s) (0 \leq i < s)$ is $\frac{P(0,i,s)}{1+P(0,0,s)+P(0,1,s)}$, the probability $P(2, s, s)$ is $\frac{P(0,0,s)+P(0,s,s)}{1+P(0,0,s)+P(0,1,s)}$ and the probability $P(2, s+1, s)$ for the new added page $(s+1)$ is $\frac{P(0,1,s)}{1+P(0,0,s)+P(0,1,s)}$. Moreover, when $2 < sp \leq s$, the probability $P(sp, i, s)$ of the page to the left of page $(sp - 2)$ (i.e., $0 \leq i < (sp - 2)$) is $\frac{P(0,i,s)+P(0,i+2,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$, while the probability $P(sp, i, s)$ of the page to the right of page $(sp - 2)$, including page $(sp - 2)$ (i.e., $(sp - 2) \leq i < s$), is $\frac{P(0,i,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$, the probability $P(sp, s, s)$ for home page s is $\frac{P(0,0,s)+P(0,s,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$ and the probability $P(sp, s+1, s)$ for the new added home page $(s+1)$ is $\frac{P(0,1,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$.

From the load distribution analysis, we observe that during the $(s+1)$th full expansion, the maximum used index (n) of home pages is s in climbing hashing when $0 \leq sp \leq 1$ and is $(s+1)$ when $2 \leq sp \leq s$. Let $W(t)$ be a function to denote the number of overflow pages to a home page with t data records inserted and let it be defined as follows:

$$W(t) = 0, \qquad 0 \leq t \leq b,$$
$$W(t) = j, \qquad (b + (j-1)w + 1) \leq t < (b + jw).$$

Let $\mathrm{Bin}(t; N, P)$ denote the binomial distribution, i.e., $\mathrm{Bin}(t; N, P) = C_t^N P^t (1 - P)^{N-t})$. The probability that home page $i (0 \leq i \leq n)$ contains t data records is $\mathrm{Bin}(t; N, P(sp, i, s))$. The expected number of overflow pages for home page i is obtained is

$$OP_i(N) = \sum_{t=0}^{N}(W(t)\text{Bin}(t; N, P(sp, i, s))).$$

Then, the average number of overflow pages for the file after inserting N data records is given by

$$OP(N) = \frac{\sum_{i=0}^{n} OP_i(N)}{n+1},$$

and the storage utilization can be obtained as follows:

$$UTI(N) = \frac{N}{(n-1)(b+wOP(N))}.$$

By using *separators* for handling overflow records, the expected cost of an unsuccessful search for home page $i(0 \leq i \leq n)$ in terms of the number of disk accesses is

$$US_i = 1, \qquad OP_i = 0,$$

$$US_i = 2, \qquad OP_i > 0.$$

Then, the average number of disk accesses for an unsuccessful search is given by

$$US(N) = \sum_{i=0}^{n}(US_i(N)P(sp, i, s)).$$

For the successful search, we first consider the expected number of disk accesses for retrieving all the data records in home page $i(0 \leq i < n)$ plus its overflow pages, which can be obtained by

$$RA_i(N) = \sum_{t=0}^{b}(t\ \text{Bin}(t, N, P(sp, i, s)))$$

$$+ \sum_{t=b+1}^{N}((t+(t-b))\text{Bin}(t, N, P(sp, i, s))).$$

Then, the average number of disk accesses for a successful search can be calculated by

$$SS(N) = \frac{\sum_{i=0}^{n} RA_i(N)}{N}.$$

For the average insertion cost, we fist consider the split cost at the insertion of the tth ($t \leq N$) data record, which is given by

$$SC(t) = 1 + OP(t) + 2(1 + OP(t+1)),$$

where a buffer mechanism is applied. Since a split occurs only when t is $L, 2L, \ldots, ns(N)L (ns(N)L \leq N)$, the total split cost for N inserted data records can be obtained by

$$TSC(N) = \sum_{i=1}^{ns(N)} SC(iL).$$

Then, we consider the average cost of inserting a data record when there are t data records that have been inserted. (Note that given the number of data records t, we can obtain the corresponding split pointer sp' and the number of full expansion s' as explained before.) Since a data insertion may cause the other data records to be reinserted, the average number of disk accesses for inserting the $(t+1)$th data record in page i is as follows:

$$\begin{aligned} AC_i(t) &= \frac{2b(1 + OP_i(t)) + 2w(OP_i(t) + OP_i(t) - 1 + \cdots + 1)}{b + wOP_i(t)} \\ &= \frac{2b(1 + OP_i(t)) + wOP_i(t)(1 + OP_i(t))}{b + wOP_i(t)}. \end{aligned}$$

Then, the average number of disk accesses for inserting a data record in any page i among those $(s'+1)$ pages is given by

$$AC(t) = \sum_{i=0}^{s'} P(sp', i, s') AC_i(t).$$

Finally, we can obtain the average insertion cost in the insertion process of N data records (including the split cost), which is given by

$$INS(N) = \frac{TSC(N) + \sum_{t=0}^{N-1} AC(t)}{N}.$$

Table 2(a) shows the results derived from the above formulas, where $s_0 = 1$, $N = 10^6$, $b = 10, 20, 40$, and 80; $w = 0.5b$; $L = 0.8b$, and $L = b$ and $L = 1.2b$ in climbing hashing. From this table, we observe that the storage utilization can be up to nearly 96%.

TABLE 2

Performance: (a) Analysis Results; (b) Simulation Results[a]

Parameters			Analysis Results				Parameters			Simulation Results			
b	w	L	INS	ss	us	uti	b	w	L	INS	ss	us	uti
10	5	08	6.6	1.932	1.934	0.940	10	5	08	6.7	1.929	1.927	0.945
10	5	10	6.4	1.938	2.0	0.958	10	5	10	6.5	1.939	1.855	0.950
10	5	12	6.2	1.939	2.0	0.959	10	5	12	6.1	1.939	1.855	0.959
20	10	16	4.2	1.870	1.899	0.934	20	10	16	4.2	1.879	1.855	0.934
20	10	20	4.1	1.876	1.904	0.950	20	10	20	4.0	1.879	1.855	0.952
20	10	24	3.9	1.878	1.905	0.962	20	10	24	3.9	1.879	1.855	0.961
40	20	32	3.2	1.780	1.988	0.952	40	20	32	3.3	1.759	2.0	0.961
40	20	40	3.2	1.785	2.0	0.956	40	20	40	3.2	1.779	2.0	0.943
40	20	48	3.1	1.803	2.0	0.958	40	20	48	3.2	1.799	2.0	0.943
80	40	64	2.9	1.632	1.998	0.924	80	40	64	3.0	1.639	2.0	0.925
80	40	80	2.8	1.650	2.0	0.943	80	40	80	2.9	1.679	2.0	0.943
80	40	96	2.8	1.728	2.0	0.947	80	40	96	2.9	1.719	2.0	0.943
			(a)							(b)			

[a]b: the size of a home page; w: the size of an overflow page; L: load control; INS: insertion cost; ss: successful search cost; us: unsuccessful search cost; uti: storage utilization.

5. SIMULATION RESULTS

In this section, we show the simulation results of climbing hashing, linear hashing [11], and linear hashing with partial expansions [5] under two different split control strategies. In this simulation study, we assume that N input data records are uniformly distributed [7]. The environment control variables are the size of a home page (b), the size of an overflow page (w), and a load control (L) [or a load factor (A)]. Storage utilization, average insertion cost, average successful search cost, and average unsuccessful search cost are the main performance measures considered. These costs are measured in terms of the number of disk accesses. Moreover, overflow pages are handled by *separators* in all three of these approaches.

Table 2(b) shows the simulation results of climbing hashing under the split control of the load control L, where $N = 10^6$, $w = 0.5b$ and $L = 0.8b$, and $L = b$ and $L = 1.2b$, respectively. Compared with the analysis results shown in Table 2(a), the simulation results shown in Table 2(b) are very close to those shown in Table 2(a).

Simulation results of climbing hashing, linear hashing, linear hashing with two partial expansions per full expansion, and linear hashing with three partial expansions per full expansion under the split control of the load control L are shown in Tables 3(a), (b), (c) and (d), respectively,

TABLE 3

Simulation Results Under the Split Control of the Load Control (L): (a) Climbing Hashing; (b) Linear Hashing; (c) Linear Hashing with Two Partial Expansions; (d) Linear Hashing with Three Partial Expansions[a]

Parameters			Climbing Hashing			
b	w	L	INS	ss	us	uti
10	5	08	6.7	1.929	1.927	0.945
10	5	10	6.5	1.939	1.855	0.950
10	5	12	6.1	1.939	1.855	0.959
20	10	16	4.2	1.879	1.855	0.934
20	10	20	4.0	1.879	1.855	0.952
20	10	24	3.9	1.879	1.855	0.961
40	20	32	3.3	1.759	2.0	0.961
40	20	40	3.2	1.779	2.0	0.943
40	20	48	3.2	1.799	2.0	0.943
80	40	64	3.0	1.639	2.0	0.925
80	40	80	2.9	1.679	2.0	0.943
80	40	96	2.9	1.719	2.0	0.943

(a)

Parameters			Linear Hashing			
b	w	L	INS	ss	us	uti
10	5	08	2.7	1.010	1.040	0.788
10	5	10	2.9	1.136	1.434	0.858
10	5	12	3.1	1.243	1.699	0.858
20	10	16	2.5	1.012	1.034	0.781
20	10	20	2.7	1.143	1.423	0.784
20	10	24	2.8	1.233	1.677	0.784
40	20	32	2.3	1.002	1.003	0.781
40	20	40	2.5	1.145	1.407	0.781
40	20	48	2.7	1.234	1.656	0.781
80	40	64	2.2	1.001	1.003	0.757
80	40	80	2.4	1.132	1.376	0.781
80	40	96	2.6	1.222	1.938	0.781

(b)

Parameters			Linear Hashing with Two Partial Expansions			
b	w	L	INS	ss	us	uti
10	5	08	3.1	1.015	1.047	0.790
10	5	10	3.3	1.144	1.445	0.858
10	5	12	3.5	1.243	1.697	0.858
20	10	16	2.7	1.016	1.046	0.781
20	10	20	2.9	1.153	1.438	0.784
20	10	24	3.1	1.241	1.689	0.784
40	20	32	2.4	1.011	1.031	0.781
40	20	40	2.6	1.154	1.438	0.781
40	20	48	2.8	1.245	1.686	0.781
80	40	64	2.3	1.000	1.000	0.781
80	40	80	2.5	1.157	1.436	0.781
80	40	96	2.7	1.247	1.686	0.781

(c)

Parameters			Linear Hashing with Three Partial Expansions			
b	w	L	INS	ss	us	uti
10	5	08	3.1	1.015	1.114	0.790
10	5	10	3.4	1.136	1.445	0.858
10	5	12	3.5	1.243	1.610	0.863
20	10	16	2.7	1.026	1.526	0.781
20	10	20	2.9	1.038	1.786	0.784
20	10	24	3.1	1.159	1.957	0.784
40	20	32	2.4	1.025	1.656	0.781
40	20	40	2.6	1.038	1.936	0.781
40	20	48	2.8	1.160	2.0	0.781
80	40	64	2.3	1.024	1.666	0.781
80	40	80	2.5	1.038	1.958	0.781
80	40	96	2.7	1.160	2.0	0.724

(d)

[a] b: the size of a home page; w: the size of an overflow page; L: load control; INS: insertion cost; ss: successful search cost; us: unsuccessful search cost; uti: storage utilization.

where $N = 10^6$, $w = 0.5b$ and $L = 0.8b$, and $L = b$ and $L = 1.2b$. From these tables, climbing hashing has the highest storage utilization among these four methods. When $b = 40$, $w = 20$, and $L = 40$, climbing hashing can achieve 96% storage utilization, as compared to 78% storage utilization in linear hashing and in linear hashing with partial expansions under the same conditions. Under a fixed N, as L is increased from 8 to 96, the number of file splits is decreased, which results in a decrease of the average insertion cost in all these three methods. Moreover, the ratio of the average insertion cost of climbing hashing to that of linear hashing is decreased from $\frac{6.7}{2.7}(\approx 2.5)$ to $\frac{2.9}{2.6}(\approx 1.1)$ when L is increased. The reason is that when L is increased, the ratio of the number of newly added pages of climbing hashing to that of linear hashing is increased under a fixed N. (Note that this ratio is always smaller than 1.) Obviously, since storage utilization and the average insertion cost (and the average retrieval cost) are always a trade-off, climbing hashing will need higher average insertion cost and average retrieval cost than the other three methods. However, in the next section, we will extend climbing hashing such that it can provide a lower average insertion cost than linear hashing at the cost of decreasing storage utilization.

Recall that the growth rate of climbing hasing is $\frac{n+1}{n}$ per full expansion, which is not a constant since n is changed during file growth, where n is the current size of the file. To compare the average insertion/retrieval cost in linear hashing and climbing hashing when both approaches achieve the same storage utilization, we try to run linear hashing under different choices of L. Table 4 shows that storage utilization in linear hashing can be increased as L is increased, at the cost of increasing the average retrieval cost, where $b = 40$, $w = 20$, and $N = 10^6$. From this table, we observe

TABLE 4
The Relationship Between Performance and L in Linear Hashing[a]

Load Control	*INS*	*ss*	*us*	*uti*
$L = 40$	2.53	1.145	1.407	0.78
$L = 50$	2.76	1.256	1.717	0.78
$L = 60$	2.93	1.325	1.906	0.78
$L = 65$	2.96	1.359	2.0	0.78
$L = 100$	3.03	1.519	2.0	0.83
$L = 200$	3.04	1.779	2.0	0.93
$L = 225$	3.04	1.799	2.0	0.94
$L = 250$	3.05	1.819	2.0	0.95
$L = 300$	3.04	1.839	2.0	0.96
$L = 350$	3.03	1.859	2.0	0.97
climbing $L = 40$	3.27	1.779	2.0	0.94

[a]L: load control, *INS*: insertion cost; *ss*: successful search cost; *us*: unsuccessful search cost; *uti*: storage utilization.

TABLE 5
Simulation Results Under the Split Control of the Load Factor (A)[a]

Load Factor	Climbing Hashing				Linear Hashing			
A	INS	ss	us	uti	INS	ss	us	uti
0.50	104	1.196	1.189	0.498	2.62	1.0	1.0	0.500
0.55	85	1.215	1.335	0.545	2.60	1.0	1.0	0.549
0.60	66	1.260	1.262	0.599	2.61	1.0	1.0	0.599
0.65	52	1.321	1.496	0.647	2.66	1.0	1.0	0.649
0.70	44	1.340	1.495	0.698	2.73	1.0	1.0	0.699
0.75	34	1.409	1.666	0.745	2.85	1.0	1.0	0.749
0.80	27	1.470	1.720	0.800	3.00	1.032	1.093	0.800
0.85	19	1.570	1.887	0.847	3.17	1.115	1.337	0.849
0.90	7	1.764	1.926	0.894	3.35	1.324	1.904	0.858
0.95	5	1.939	1.855	0.950	3.28	1.671	2.0	0.892

[a]b: the size of a home page; w: the size of an overflow page; A: load factor; INS: insertion cost; ss: successful search cost; us: unsuccessful search cost; uti: storage utilization.

that when both approaches have the same storage utilization (or the same average successful search cost, or the same average unsuccessful search cost, or the same average insertion cost), one will have better performance than the other in some performance measures, while having worse performance than the other in some other performance measures. The reason is that as L is increased a lot in linear hashing, the number of file splits is decreased in linear hashing. Therefore, given a fixed N and the same storage utilization, the number of home pages in linear hashing is less than the one in climbing hashing. At the same time, the number of overflow pages in linear hashing is greater than the one in climbing hashing. Consequently, the average retrieval cost in climbing hashing is better than that in linear hashing.

Table 5 shows the simulation results of climbing hashing and linear hashing under the split control of the load factor (A), where $N = 10^6$, $b = 10$, and $w = 5$. In climbing hashing, when A is increased from 0.5 to 0.95, the number of file splits is decreased, which results in a decrease of the average insertion cost. While in linear hashing, as A is increased from 0.5 to 0.95, the average insertion cost is increased. The reason is that as A is increased, the number of overflow pages is increased (which is denoted as factor one), while the number of file splits is decreased (which is denoted as factor two). As A is increased, factor one dominates the performance of the average insertion cost in linear hashing; while in climbing hashing, factor two dominates the performance of the average insertion cost. As A is

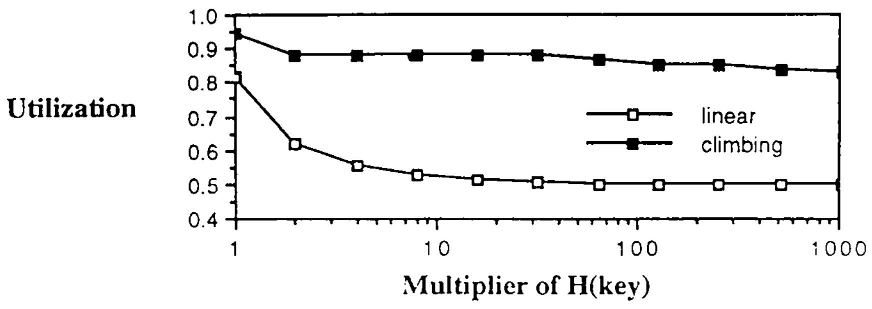

Fig. 6. The relationship between storage utilization and non-uniform key distribution.

increased, which implies that the storage utilization threshold is increased, oscillation performance during a full expansion is increased, as stated in [5, 11]. Since the growth rate of climbing hashing is $\frac{n+1}{n}$ per full expansion as compared to 2 in linear hashing, climbing hashing will result in smaller oscillation during a full expansion than linear hashing. From Table 5, as A is increased from 0.5 to 0.95, the ratio of the average insertion cost of climbing hashing to that of linear hashing is decreased from $\frac{10.4}{2.62}$ to $\frac{5}{3.28}$. Moreover, when $A > 0.85$, climbing hashing can have higher storage utilization than linear hashing. The reason is that the higher A is, the higher the ratio of performance oscillation during a full expansion in linear hashing to that in climbing hashing.

The above simulation results are based on the assumption that the input data records are uniformly distributed. Now, let us examine one more interesting result when the input data records are not uniformly distributed. Consider a special case in which almost all of the data records are, unfortunately, hashed into the same home page. This can be simulated by letting keys of data records be multiplied by $2^1, 2^2, \ldots, 2^9$ and 2^{10}. Figure 6 shows a comparison of storage utilization between climbing hashing and linear hashing under this case, where $N = 10^6$, $b = 10$, $w = 5$, $L = 10$. (Note that in Figure 6, the X axis has been replaced by the logarithmic function of X with base 2.) In this case, climbing hashing can provide even better storage utilization than linear hashing as the multiplier is increased. When the multiplier is increased (i.e., the number of data records into the same home page is increased), storage utilization in linear hashing even drops below 50%, while climbing hashing still keeps the storage utilization above 85%.

This reason can be explained as follows. Assume that k splits occur in linear hashing and the file initially contains one page; in this case, k more pages are added. Under the same number of splits, there are s more pages added in climbing hashing, where $(1+2+\cdots+s) \leq k < (1+2+\cdots+(s+1))$, as explained in Section 4. Therefore, $\frac{(s+1)s}{2} \leq k$ and $s = \lfloor \frac{\sqrt{8k+1}-1}{2} \rfloor$; i.e., after k splits occur, $\lfloor \frac{\sqrt{8k+1}-1}{2} \rfloor$ pages are added in climbing hashing as compared to k pages in linear hashing. When $L = b$ and $w = 1$, the storage utilization is $\frac{(k+1)b}{\lfloor \frac{\sqrt{8k+1}-1}{2} \rfloor b+kb}$ in climbing hashing as compared to $\frac{(k+1)b}{(k+1)b+kb}$ in linear hashing, where there are kb overflow records. As k is increased, storage utilization is near 1 in climbing hashing, while it is about $\frac{1}{2}$ in linear hashing.

6. EXTENSION

In this section, we extend the proposed scheme to have a growth rate of $\frac{n+t-1}{n}$ per full expansion ($t \geq 2$); i.e., $(t-1)$ more pages are added per full expansion, such that the number of disk accesses for data retrieval and insertion operations can be reduced.

Let each key be mapped into a string of t-base digits, i.e., $H_t(\text{key}) = c = (c_{q-1}, c_{q-2}, \ldots, c_1, c_0)$ $(0 \leq c_i < t$ and $0 \leq i < q)$. Let $h_0(c) = m_0$ be the function to load the file initially, where $0 \leq m_0 \leq (m-1)$ and m denotes the initial number of pages of a file. The rest of the split functions $h_1, h_2, \ldots, h_i$ for extended climbing hashing are defined as follows:

$$h_0(c) = m_0,$$

where $0 \leq m_0 \leq (m-1)$

$$h_1(c) = h_0(c) + c_0,$$

$$h_{i+1}(c) = (h_i(c) + ic_i) \bmod (m + (i+1)(t-1)),$$

for $i \geq 1$, that is, $0 \leq h_{i+1}(c) \leq (m-1) + (i+1)(t-1)$.

Table 6 shows the simulation results of extended climbing hashing under the split control L, where $N = 10^6$, $b = 80$, $w = 40$, and $L = 96$, which corresponds to what we have claimed: as t is increased, the growth rate per full expansion is increased, resulting in a decrease of storage utilization and costs of data retrieval and insertion operations. Moreover, costs of data retrieval and insertion operations in extended climbing hashing can even drop below those in linear hashing, at the cost of decreasing storage utilization. Therefore, if we care about fast retrieval (and low insertion cost) more than high storage utilization, we choose a t with a large value in extended climbing hashing. Since high storage utilization and fast data retrieval (and low average insertion cost) are always a tradeoff, the proposed extended climbing hashing provides a flexible choice between these two requirements.

TABLE 6
Simulation Results in Extended Climbing Hashing

Scheme	*INS*	*ss*	*us*	*uti*
climbing	3.01	1.72	2.00	0.94
$t = 3$	2.88	1.59	1.86	0.89
$t = 4$	2.80	1.48	2.00	0.89
$t = 5$	2.74	1.53	1.86	0.79
$t = 6$	2.67	1.42	1.94	0.79
$t = 7$	2.61	1.29	1.80	0.79
$t = 8$	2.56	1.28	1.71	0.78
$t = 9$	2.54	1.33	1.68	0.70
$t = 10$	2.51	1.31	1.67	0.63
$t = 11$	2.48	1.30	1.66	0.56
$t = 12$	2.46	1.29	1.66	0.54
$t = 13$	2.43	1.28	1.65	0.52
$t = 14$	2.39	1.24	1.65	0.50
$t = 15$	2.36	1.19	1.65	0.48
$t = 16$	2.32	1.14	1.61	0.47
linear	2.59	1.22	1.62	0.78

^{a}t: base system; *INS*: insertion cost; *ss*: successful search cost; *us*: unsuccessful search cost; *uti*: storage utilization

7. CONCLUSION

In this paper, we have proposed a new dynamic hashing scheme called climbing hashing. Climbing hashing always adds only one more page after a full expansion: that is, the growth rate of a file is $\frac{n+1}{n}$ per full expansion, when n is the number of pages of the current size of file. From our mathematical analysis and simulation study, given a fixed load control, climbing hashing can achieve 96% storage utilization as compared to 78% storage utilization using linear hashing, when the keys are uniformly distributed. Moreover, when the keys are not uniformly distributed, climbing hashing can still achieve above 85% storage utilization given a fixed load control, while the storage utilization in linear hashing will drop below 50%. Since high storage utilization and fast data retrieval are always a trade-off in all dynamic hashing schemes, we have extended climbing hashing to set a growth rate of a file to $\frac{n+t-1}{n}$ per full expansion in order to find a compromise between high storage utilization and fast data retrieval. Our simulation results show that, if we care about fast retrieval more that high storage utilization, we choose a t with a large value in extended climbing hashing. Therefore, extended climbing hashing provides a flexible choice between these two requirements. Since there are many factors about which a file structure designer cares, including fast data retrieval, a low average insertion cost, high storage utilization, and stable performance through file expansions, our approach provides designers a useful and flexible formula with which to reach their goals.

The authors are grateful to anonymous referees for their careful reading and helpful comments.

REFERENCES

1. R. J. Enbody and H. C. Du, Dynamic hashing schemes, *ACM Computing Surveys* 20(2):85–113 (June 1988).
2. N. I. Hachem and P. B. Berra, New order preserving access method for very large files derived from linear hashing, *IEEE Trans. Knowledge and Data Eng.* 4(1):68–82 (Feb. 1992).
3. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, Extendible hashing— A fast access method for dynamic files, *ACM Trans. Database Syst.* 4(3):315–344 (Sept. 1979).
4. P. Larson, Dynamic hashing, *BIT* 18:184-201 (1978).
5. P. Larson, Linear hashing with partial expansions, in *Proceedings of the 6th International Conference on Very Large Data Bases*, Oct. 1980, pp. 224–232.
6. P. Larson, A single-file version of linear hashing with partial expansions, in *Proceedings of the 8th International Conference on Very Large Data Bases*, Sept. 1982, pp. 300–309.
7. P. Larson, Performance analysis of linear hashing with partial expansions, *ACM Trans. Database Syst.* 7(4):566 587 (Dec. 1982).
8. P. Larson and A. Kajla, "File organization: Implementation of a method guaranteeing retrieval in one access, *ACM Computing Practice* 27(7):670 677 (July 1984).
9. P. Larson, Linear hashing with overflow-handling by linear probing, *ACM Trans. Database Syst.* 10(1):75-89 (Mar. 1985).
10. P. Larson, Linear hashing with separators—A dynamic hashing scheme achieving one-access retrieval, *ACM Trans. Database Syst.* 13(3):366-388 (Sept. 1988).
11. W. Litwin, Linear hashing: A new tool for files and tables addressing, in *Proceedings of the 6th International Conference on Very Large Data Base*, Oct. 1980, pp. 212–223.
12. D. B. Lomet, Bounded index exponential hashing, *ACM Trans. Database Syst.* 8(1):136-165 (Mar. 1983).
13. D. B. Lomet, Partial expansions for file organizations with an index, *ACM Trans. Database Syst.* 12(1):65–84 (Mar. 1987).
14. J. K. Mullin, "Tightly controlled linear hashing without separate overflow storage, *BIT* 21(4):390 400 (1981).

15. K. Ramamohanarao and J. W. Lloyd, Dynamic hashing schemes, *Computer J.* 25(4):478–485 (1982).
16. K. Ramamohanarao, Recursive linear hashing, *ACM Trans. Database Syst.* 9(3):369–391 (Sept. 1984).
17. M. Scholl, New file organizations based on dynamic hashing, *ACM Trans. Database Syst.* 6(1):194–211 (Mar. 1981).

Received 1 January 1994; revised 4 June 1994final ; manuscript accepted 30 November 1994