Alternating Hashing for Expansible Files

Ye-In Chang, *Member*, *IEEE*, and Chien-I Lee

Abstract—In this paper, we propose a generalized approach for designing a class of dynamic hashing schemes which 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. Based on this generalized approach, we derive a new dynamic hashing scheme called *alternating hashing*, in which, when a split occurs in page k, the data records in page k will be redistributed to page k and page $(k + 1)$, or page k and page $(k - 1)$, according to whether the value of *level* d is even or odd, respectively. (Note that a *level* is defined as the number of full expansions happened so far.) From our performance analysis, given a fixed load control, the proposed scheme can achieve nearly 97% storage utilization as compared to 78% storage utilization by using linear hashing.

Index Terms—Access methods, dynamic storage allocation, file organization, file system management, hashing.

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 and distribution of keys while maintaining fast retrieval time [2]. Basically, dynamic hashing schemes can be divided into two classes: one needs an index, the other one does not need an index. Extendible hashing [4] and dynamic hashing [5] belong to the first class. Linear hashing [3], [6], [7], [8], [10], [11], [13], [15], [16] and spiral storage [1], [14] belong 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. In linear hashing, a file is expanded by adding a new page at the end of the file when a split occurs and re-locating a number of records to the new page by using a new hashing function. The new hashing function doubles the size of the address space created by the old hashing function. Therefore, after a *full expansion* (defined in Section 2), the number of pages is doubled. To maintain stable performance through file expansions in linear hashing, many strategies have been proposed in which linear hashing with partial expansions as first presented by Larson [6], [8] is a generalization of Litwin's linear hashing [13]. 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 which 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 *alternating hashing*, in which, when a split occurs in page k, the data records in page k will be redistributed to page k and page $(k + 1)$, or page k and page $(k - 1)$, according to whether the value of *level d* is even or odd, respectively. (Note that a *level* is defined as the number of full expansions happened so far.) From our performance analysis and simulation, given a fixed load control, alternating hashing can achieve nearly 97%, 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.) 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 decreases storage utilization).

2 The Generalized Approach and Alternating Hashing

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. Let each key be mapped into a string of binary bits b_i first, i.e., $H(\text{key}) = (b_{q-1}, ..., b_1, b_0) = c$. Then, this scheme addresses records by using a series of *split functions*, $h_0, h_1, ..., h_q$, where each function h_i maps c to a non-negative 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 [13]. A *level* is defined as the number of full expansions happened so 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: $c \rightarrow \{0, ..., s_0 - 1\}$, 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, $h_1, h_2, ..., h_i$, are defined as follows:

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

$$h_{i+1}(c) = (h_i(c) + w(i)b_i) \bmod (s_0 + i + 1), \qquad \text{for } i \geq 0,$$

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 $\{0, 1, ..., s_0 + i\}$; 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.

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

- *Y.-I. Chang is with the Department of Applied Mathematics, National Sun Yat-Sen University, Kaohsiung, Taiwan, Republic of China. E-mail: changyi@math.nsysu.edu.tw.*
- *C.-I. Lee is with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China. E-mail: leeci@winston.cis.nctu.edu.tw.*

Manuscript received Jan. 23, 1995; revised Nov. 27, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96071.

1041-4347/97/$10.00 © 1997 IEEE

$$h_{i+1}(c) = \begin{cases} h_i(c) + (-1)^i b_i, & \text{if } h_{i+1}(c) \geq 0 \\ i+1, & \text{otherwise.} \end{cases}$$

In general, when an insertion causes a split and $sp = k$, the data records in page k will be redistributed into page k and page $(k + 1)$, or page k and page $(k - 1)$, according to whether the value of level d is even or odd, respectively. If d is an even number, the data records in page k will be reinserted into page k or $(k + 1)$, according to whether the value of bit b_d is 0 or 1, respectively; otherwise (i.e., d is an odd number), the data records in page k will be reinserted into page k or $(k - 1)$, according to whether the value of bit b_d is 0 or 1, respectively. When d is an even number and $sp = d$, i.e., sp has pointed to the maximum index of pages on level d, then a new page $(d + 1)$ is added at the end of the file, and the data records in page d are redistributed to page d or page $(d + 1)$, according to whether the value of bit b_d is 0 or 1, respectively. When d is an odd number and $sp = 0$, then a new page $(d + 1)$ is added at the end of the file, and the data records in page 0 are redistributed to page 0 or page $(d + 1)$, according to whether the value of bit b_d is 0 or 1, respectively.

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 pattern 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 Fig. 1a. In this function, initially, all the data records are mapped into page 0 by $h_0(c) = 0$ and hence, *address* = 0. The *address* after the for-loop statement represents the home page number after the dth full expansion. Depending on whether or not *address* < *sp*, the final home page number is determined.

3.2 Overflow Handling and Retrieval

In [11], Larson applied *separators* [9] 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. To understand what a *separator* is, let's define a *probe sequence* first [11]. 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*, $p(K) = (p_1(K), p_2(K), ..., p_n(K))$, $(n \geq 1)$, defines the order in which the pages will be checked when inserting or retrieving the record. That is, every *probe sequence* is a permutation of the set $\{1, 2, ..., n\}$. For each data record with key = K, its *signature sequence*, $s(K) = (s_1(K), s_2(K), ..., s_n(K))$, is a q-bit integer. (Note that $q \geq 1$ and q should be large enough such that the values of the signatures of all data records can be in $\{0, (2^q - 2)\}$ [9], [11]. When a data record with key = K probes page $p_i(K)$, *signature* $s_i(K)$ is used, $1 \leq i \leq n$. Implementation of $p(K)$ and $s(K)$ are discussed in detailed in [9]. 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 which uniquely separates the two

```
function home_address(key) : integer;
  var c : integer;          /* = H(key) */
  i : integer;          /* an index */
  address : integer;
  begin
    c = H(key);
    address = 0;          /* i.e., h0(c) */
    for i = 0 to (d-1) do
    begin
      address = address + (-1)^i × b_i(c);
      if address < 0 then address = i + 1;
    end;
    if address < sp then address = address + (-1)^d × b_d;
    if address < 0 then address = d + 1;
    return (address);
  end;
```

(a)

```
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) )
    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);
    end;
  end;
```

(b)

```
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 i
          such that s_ij(K) < separator_ij↑.value;
        if not found then /* the overflow pages are full */
        begin
          append an overflow page to the external file of home page i;
          recompute the probe sequences and signature sequences;
        end;
        find an entry j in separator table i
          such that s_ij(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;
```

(c)

Fig. 1. The algorithms: a) function *home_address*; b) function *retrieval*; c) procedure *file_split*.

groups is called a *separator*, and is stored in a *separator table*. The value stored is the lowest *signature* occurring among those records which 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. And, the initial values of separators are strictly greater than all signature values. For example, using q bits as described before, the initial values of separators are set to $(2^q - 1)$, meaning that their corresponding pages are empty, initially [9], [11].)

Since in [11], 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, alternating hashing also applies *separators* but only for overflow pages. To apply *separators* to handle overflow pages in alternating 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 only when overflow pages 1, 2, ..., $(j - 1)$ are full.) 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, $1 \leq j \leq m$. 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 which has overflow pages, we store a *separator table* in each home page.

As shown in Fig. 1b, 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 $separator_{ij}$, $1 \leq j \leq m$, 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, File Split and File Contraction

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* which are stored in the home page may also have to be updated. 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 and page $(sp + 1)$ or page sp and page $(sp - 1)$, according to whether the value of level d is even or odd, respectively. If $sp = d$, d is increased by 1 and sp is reset to 0. The results of the above actions are equal to update sp (and d) first and then reinsert those data records which are in the page where the old sp points to by using the new hashing function h_{d+1}. The description of procedure *file_split* is shown in Fig. 1c. (Note that to reduce the number of disk accesses, we use a buffer mechanism to reduce the overhead of reinsertion.) Whenever the number of deletions of a file drops below a control condition, a contraction occurs. The description of procedure *file_contraction* can be found in [12].

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 [2], [13]: uncontrolled and controlled splitting. The uncontrolled splitting means that a split occurs whenever a collision occurs. In the 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 [13]. 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 [6], [15]. Therefore, we prefer to use the load control as the split control strategy as that in [15], [16].

In this section, we present the performance analysis of alternating 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 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 $\left\lceil \frac{k}{w} \right\rceil$ 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 $ns(N) = 0$ when $0 \leq N \leq s_0 L$ or $ns(N) = \left\lceil \frac{N - s_0 L}{L} \right\rceil$ when $N > s_0 L$. (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 alternating hashing, the growth rate of a file is $\frac{n+1}{n}$ per full expansion, the number of home pages expanded (denoted by m) is given by

$$s_0 + (s_0 + 1) + \ldots + (s_0 + (m - 1)) \leq ns(N) < s_0 + (s_0 + 1) + \ldots + (s_0 + m).$$

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, $\frac{(m + 2s_0 - 1)m}{2} \leq ns(N)$ and $m = \left\lfloor \frac{\sqrt{8ns(N) + (2s_0 - 1)^2} - 2s_0 + 1}{2} \right\rfloor$.

Then, the maximum index of home pages for the file is s $(= (s_0 + m - 1))$, and sp is $\left(ns(N) - \frac{(m + 2s_0 - 1)m}{2} \right)$. The load distribution for each home page is different in alternating 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 uniformly distributed. Initially, we have $X_0^1 = 1$, $X_1^1 = 1$. The value of X_i^d is $\left(X_i^{d-1} + X_{i-1}^{d-1}\right)$ when d is odd ($d > 1$), $0 \leq i \leq d$. On the other hand, the value of X_i^d is $\left(X_i^{d-1} + X_{i+1}^{d-1}\right)$ when d is even ($d > 1$), $0 \leq i < (d - 1)$. Moreover, the value of X_{d-1}^d is X_{d-1}^{d-1} and the value of X_d^d is X_0^{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 alternating hashing (when s is even) and $sp = 0$, that is, alternating hashing splits backwards during the sth full expansion, the probability $P(0, i, s)$ for home page i ($0 \le i < s$) is $\frac{P(0,i,s-1)+P(0,i+1,s-1)}{2}$ and the probability $P(0, s, s)$ for home page s is $\frac{P(0,0,s-1)}{2}$. During the $(s + 1)$th full expansion, since $(s + 1)$ is odd, alternating hashing splits forwards. 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 1, the probability $P(1, 0, s)$ is $\frac{P(0,0,s)}{1+P(0,0,s)}$, the probability $P(1, 1, s)$ is $\frac{P(0,0,s)+P(0,1,s)}{1+P(0,0,s)}$ and the probability $P(1, i, s)$ ($2 \le i \le s$) is $\frac{P(0,i,s)}{1+P(0,0,s)}$. Moreover, when $1 < sp \le s$, the probability $P(sp, i, s)$ of the page left of page $(sp + 1)$ (i.e., $0 \le i < (sp + 1)$) is $\frac{P(0,i,s)+P(0,i-1,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$ (with $P(0, -1, s) = 0$), while the probability $P(sp, i, s)$ of the page right of page $(sp + 1)$, including page $(sp + 1)$, (i.e., $(sp + 1) \le i \le s$) is $\frac{P(0,i,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$.

TABLE 1
THE VARIANCE OF THE LOAD DISTRIBUTION IN ALTERNATING HASHING

Level	Page number (1)								Mean	Variance
d	0	1	2	3	4	5	6	7		
1	1	1							1	0
2	2	1	1						1.3	0.2
3	2	3	2	1					2	0.5
4	5	5	3	1	2				3.2	2.56
5	5	10	8	4	3	2			5.3	7.88
6	15	18	12	7	5	2	3		9.1	30.1
7	15	33	30	19	12	7	7	5	16	99.2

Mean = $\frac{1}{d+1}\sum_{i=0}^{d} X_i^d \left(=\frac{2^d}{d+1}\right)$ Variance = $\frac{1}{d+1}\sum_{i=0}^{d}\left(X_i^d - Mean\right)^2$

On the other hand, if s is odd and $sp = 0$, that is, alternating hashing split forwards during the sth full expansion, the probability $P(0, i, s)$ for home page i ($0 \le i < s$) is $\frac{P(0,i,s-1)+P(0,i-1,s-1)}{2}$ and the probability $P(0, s, s)$ for home page s is $\frac{P(0,s-1,s-1)}{2}$ (with $P(0, -1, s - 1) = 0$). During the $(s + 1)$th full expansion, since $(s + 1)$ is even, alternating hashing splits backwards. 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 a new added home page (i.e., page $(s + 1)$), the probability $P(1, i, s)$ ($0 \le i \le s$) is $\frac{P(0,i,s)}{1+P(0,0,s)}$ and the probability $P(1, s + 1, s)$ for the new added home page $(s + 1)$ is $\frac{P(0,0,s)}{1+P(0,0,s)}$. Moreover, when $1 < sp \le s$, the probability $P(sp, i, s)$ of the page left of page $(sp - 1)$ (i.e., $0 \le i < (sp - 1)$) is $\frac{P(0,i,s)+P(0,i+1,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$, while the probability $P(sp, i, s)$ of the page right of page $(sp - 1)$, including page $(sp - 1)$, (i.e., $(sp - 1) \le i \le s$) is $\frac{P(0,i,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$ and the probability $P(sp, s + 1, s)$ for the new added home page $(s + 1)$ *is* $\frac{P(0,0,s)}{1+\sum_{k=0}^{sp-1} P(0,k,s)}$.

From the above load distribution analysis, we observe that during the $(s + 1)$th full expansion, the maximum used index (n) of home pages is s if s is even (or s is odd and $sp = 0$). Otherwise, n is $(s + 1)$. Let $W(t)$ be a function to denote the number of overflow pages of a home page with t data records inserted and let it be defined as follows:

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

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 ($0 \le i \le n$) 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 as $OP_i(N) = \sum_{t=0}^{N} (W(t)\mathrm{Bin}(t; N, P(sp, i, s)))$. 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 given by $UTI(N) = \frac{N}{(n+1)(b+wOP(N))}$.

By using *separators* for handling overflow records, the expected cost of an unsuccessful search for home page i ($0 \le i \le n$) in terms of the number of disk accesses is $US_i = 1$ when $OP_i = 0$, or $US_i = 2$ when $OP_i > 0$. 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 ($0 \le i \le n$) plus its overflow pages, which can be obtained by

$$RA_i(N) = \sum_{t=0}^{b} (t\mathrm{Bin}(t, N, P(sp, i, s))) + \sum_{t=b+1}^{N} ((t + (t-b))\mathrm{Bin}(t, N, P(sp, i, s)))$$

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 first consider the split cost at the insertion of the tth ($t \le N$) data record, which is given by $SC(t) = 1 + OP(t) + 2(1 + OP(t + 1))$. (Note that since we apply a buffer mechanism, $(1 + OP(t))$ disk accesses are need to read the split page and its overflow pages into the buffer, and $2(1 + OP(t + 1))$ disk accesses are needed to write the split results.) Since a split occurs only when t is $L, 2L, \ldots, ns(N)L$ ($ns(N)L \le N$), 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 which 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:

$$AC_i(t) = \frac{2b(1+OP_i(t)) + 2w(OP_i(t) + OP_i(t) - 1 + \ldots + 1)}{b + wOP_i(t)}$$
$$= \frac{2b(1+OP_i(t)) + wOP_i(t)(1+OP_i(t))}{b + wOP_i(t)}.$$

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 2a shows the results derived from the above formulas, where $s_0 = 1$, $N = 10^6$, $b = 10, 20, 40$, and 80, $w = 0.5b$, and $L = 0.8b$, $L = b$, and $L = 1.2b$ in alternating hashing. From this table, we observe that the storage utilization can be up to nearly 97%.

5 SIMULATION RESULTS

In this section, we show the simulation results of alternating hashing, linear hashing [13] and linear hashing with partial expan-

TABLE 2
PERFORMANCE: a) ANALYSIS RESULTS OF ALTERNATING HASHING; b) SIMULATION RESULTS OF ALTERNATING HASHING; c) LINEAR HASHING; d) LINEAR HASHING WITH TWO PARTIAL EXPANSION; e) LINEAR WITH THREE PARTIAL EXPANSIONS

Parameters			alanysis results of Alternating Hashing			
b	w	L	INS	ss	us	uti
10	5	08	6.6	1.932	1.970	0.940
10	5	10	6.4	1.938	2.0	0.958
10	5	12	6.3	1.939	2.0	0.959
20	10	16	4.3	1.870	1.949	0.934
20	10	20	4.0	1.878	1.989	0.950
20	10	24	3.9	1.879	1.989	0.972
40	20	32	3.4	1.780	1.988	0.962
40	20	40	3.2	1.782	2.0	0.968
40	20	48	3.1	1.820	2.0	0.972
80	40	64	2.9	1.632	1.980	0.944
80	40	80	2.8	1.720	1.987	0.958
80	40	96	2.8	1.728	2.0	0.966

(a)

Parameters			simulation results of Alternating Hashing			
b	w	L	INS	ss	us	uti
10	5	08	6.7	1.934	1.977	0.943
10	5	10	6.6	1.939	1.947	0.948
10	5	12	6.2	1.939	1.947	0.957
20	10	16	4.3	1.879	1.947	0.930
20	10	20	4.1	1.879	1.947	0.948
20	10	24	4.0	1.879	1.947	0.966
40	20	32	3.3	1.784	1.988	0.962
40	20	40	3.2	1.779	2.0	0.962
40	20	48	3.2	1.819	1.980	0.971
80	40	64	2.9	1.642	1.960	0.943
80	40	80	2.9	1.728	1.968	0.962
80	40	96	2.9	1.728	2.0	0.926

(b)

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

(c)

Parameters			Linear Hashing with Two Par. Exp.			
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

(d)

Parameters			Linear Hashing with Three Par. Exp.			
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

(e)

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

sions [6] under two different split control strategies. In this simulation study [15], we assume that N input data records are uniformly distributed [8]. 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 these three approaches. Table 2b shows the simulation results of alternating hashing under the split control of the load control L, where $N = 10^6$, $w = 0.5b$, and $L = 0.8b$, $L = b$ and $L = 1.2b$, respectively. Compared with the analysis results shown in Table 2a, the simulation results shown in Table 2b are very close to those shown in Table 2a.

Simulation results of alternating 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 2b, 2c, 2d, and 2e, respectively, where $N = 10^6$, $w = 0.5b$, and $L = 0.8b$, $L = b$, and $L = 1.2b$. From these tables, alternating hashing has the highest storage utilization among these four methods. When $b = 40$, $w = 20$, and $L = 48$, alternating hashing can achieve 97% 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 alternating 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 alternating hashing to that of linear hashing is increased under a fixed N. (Note that this ratio is always smaller than 1.) Fig. 2. shows the relationship between storage utilization and the number of inserted data records in alternating hashing and linear hashing, where $b = 80$, $w = 40$, and $L = 80$. From this figure, we observe that alternating hashing has more stable and higher storage utilization than linear hashing. That is, the oscillation in performance during a full expansion in alternating hashing is smaller than the one in linear hashing.

Recall that the growth rate of alternating hashing 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

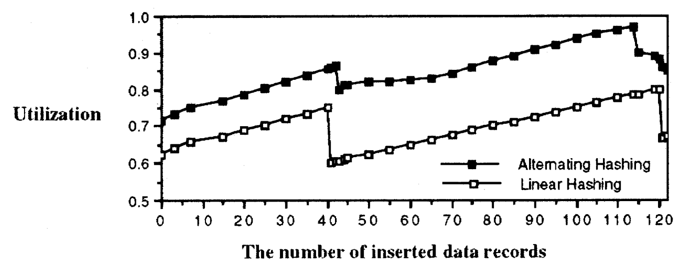

Fig. 2. The relationship between storage utilization and the number of inserted data records.

TABLE 3
THE RELATIONSHIP BETWEEN PERFORMANCE AND L IN LINEAR HASHING

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 = 250	3.05	1.819	2.0	0.95
L = 300	3.04	1.839	2.0	0.96
L = 335	3.03	1.859	2.0	0.97
L = 350	3.03	1.859	2.0	0.97
alternating L = 40	3.21	1.779	2.0	0.96

L : load control
INS : insertion cost
ss : successful search cost
us : unsuccessful search cost
uti : storage utilization

TABLE 4
SIMULATION RESULTS UNDER THE SPLIT CONTROL OF THE LOAD FACTOR (A)

Load Factor	Alternating Hashing				Linear Hashing			
A	INS	ss	us	uti	INS	ss	us	uti
0.50	57	1.934	1.880	0.500	2.62	1.0	1.0	0.500
0.55	44	1.897	1.847	0.554	2.60	1.0	1.0	0.549
0.60	38	1.887	1.927	0.598	2.61	1.0	1.0	0.599
0.65	25	1.930	1.913	0.654	2.66	1.0	1.0	0.649
0.70	24	1.880	1.887	0.700	2.73	1.0	1.0	0.699
0.75	15	1.931	1.889	0.751	2.85	1.0	1.0	0.749
0.80	14	1.884	1.923	0.798	3.00	1.032	1.093	0.800
0.85	10	1.899	1.846	0.851	3.17	1.115	1.337	0.849
0.90	5	1.934	1.910	0.901	3.35	1.324	1.904	0.858
0.95	5	1.935	1.945	0.947	3.28	1.671	2.0	0.892

A : load factor INS : insertion cost ss : successful search cost us : unsuccessful search cost uti : storage utilization

average insertion/retrieval cost in linear hashing and alternating hashing when both approaches achieve the same storage utilization, we try to run linear hashing under different choices of L. Table 3 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 that when both approaches have the same storage utilization (or the similar 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 measure, 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 alternating hashing. At the same time, the number of overflow pages in linear hashing is greater than the one in alternating hashing. Consequently, the average retrieval cost in alternating hashing is better than the one in linear hashing.

Table 4 shows the simulation results of alternating hashing and linear hashing under the split control of the load factor (A), where $N = 10^6$, $b = 10$, and $w = 5$. In alternating 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 alternating hashing, factor two dominates the performance of the average insertion cost. As A is increased, which implies that the storage utilization threshold is increased, oscillation in performance during a full expansion is increased as stated in [6], [13]. Since the growth rate of alternating hashing is $\frac{n+1}{n}$ per full expansion as compared to 2 in linear hashing, alternating hashing will result in smaller oscillation during a full expansion than linear hashing. From Table 4, as A is increased from 0.5 to 0.95, the ratio of the average insertion cost of alternating hashing to that of linear hashing is decreased from $\frac{57}{2.62}$ to $\frac{5}{3.28}$. Moreover, when $A > 0.85$, alternating 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 the one in alternating hashing is.

The proposed scheme can be extended 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., $H_t(key) = c = (c_{q-1}, c_{q-2}, \ldots,$

c_1, c_0) ($0 \le c_i < t$ and $0 \le i < q$). Let $h_0(c) = m_0$ be the function to load the file initially, where $0 \le m_0 \le (m - 1)$ and m denotes the number of pages of a file initially. The rest of the split functions, h_1, h_2, ..., h_i for extended alternating hashing are defined as follows:

$$h_0(c) = m_0, \quad \text{where } 0 \le m_0 \le (m-1)$$

$$h_{i+1}(c) = h_i(c) + (-1)^i c_i \quad \text{if } h_i(c) + (-1)^i c_i \ge 0, \quad \text{or}$$

$$h_{i+1}(c) = (i+1)(t-1) + (h_i(c) + (-1)^i c_i + (m-1) + 1, \quad \text{otherwise};$$

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

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. Therefore, if we are care about fast retrieval (and a low average insertion cost) more than high storage utilization, we choose a t with a large value in extended alternating hashing [12].

6 Conclusion

In this paper, we have proposed a generalized approach for designing a class of dynamic hashing scheme. By reducing the growth rate per full expansion to increase storage utilization, we have derived a new relationship among a series of split functions, resulting in a new dynamic hashing scheme called alternating hashing. Alternating 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 the file. From our mathematical analysis and simulation study, given a fixed load control, alternating hashing can achieve 97% storage utilization as compared to 78% storage utilization using linear hashing, when the keys are uniformly distributed. Moreover, we have extended alternating 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. Therefore, extended alternating hashing provides a flexible choice between these two requirements. Since there are many factors which a file structure designer cares about, including fast data retrieval, a low average insertion cost, high storage utilization, and stable performance through file expansions, our approach provides the designers a useful and flexible formula to reach their goals.

Acknowledgments

This research was supported by the National Science Council of Republic of China under Grant No. NSC-82-0408-E-110-135.

References

[1] J.H. Chu and G.D. Knott, "An Analysis of Spiral Hashing," *The Computer J.*, vol. 37, no. 8, pp. 715-719, 1994.
[2] R.J. Enbody and H.C. Du, "Dynamic Hashing Schemes," *ACM Computing Surveys*, vol. 20, no. 2, pp. 85-113, June 1988.
[3] 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.*, vol. 4, no. 1, pp. 68-82, Feb. 1992.
[4] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Trans. Database Systems*, vol. 4, no. 3, pp. 315-344, Sept. 1979.
[5] P. Larson, "Dynamic Hashing," *BIT*, vol. 18, pp. 184-201, 1978.
[6] P. Larson, "Linear Hashing with Partial Expansions," *Proc. Sixth Int'l Conf. Very Large Data Bases*, pp. 224-232, Oct. 1980.
[7] P. Larson, "A Single-File Version of Linear Hashing with Partial Expansions," *Proc. Eighth Int'l Conf. Very Large Data Bases*, pp. 300-309, Sept. 1982.
[8] P. Larson, "Performance Analysis of Linear Hashing with Partial Expansions," *ACM Trans. Database Systems*, vol. 7, no. 4, pp. 566-587, Dec. 1982.
[9] P. Larson and A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access," *ACM Computing Practices*, vol. 27, no. 7, pp. 670-677, July 1984.
[10] P. Larson, "Linear Hashing with Overflow-Handling by Linear Probing," *ACM Trans. Database Systems*, vol. 10, no. 1, pp. 75-89, Mar. 1985.
[11] P. Larson, "Linear Hashing with Separators—A Dynamic Hashing Scheme Achieving One-Access Retrieval," *ACM Trans. Database Systems*, vol. 13, no. 3, pp. 366-388, Sept. 1988.
[12] C.I. Lee, "Design and Analysis of Dynamic Hashing Schemes Without Indexes," Master's Thesis, Dept. of Applied Mathematics, National Sun Yat-Sen Univ., R.O.C, June, 1993.
[13] W. Litwin, "Linear Hashing: A New Tool for Files and Tables Addressing," *Proc. Sixth Int'l Conf. Very Large Data Bases* , pp. 212-223, Oct. 1980.
[14] G.N. Martin, "Spiral Storage: Incrementally Augmentable Hash Addressed Storage," Univ. of Warwick, Theory of Computation Report, no. 27, Conventry, England, Mar. 1979.
[15] K. Ramamohanarao and J.W. Lloyd, "Dynamic Hashing Schemes," *The Computer J.*, vol. 25, no. 4, pp. 478-485, 1982.
[16] K. Ramamohanarao, "Recursive Linear Hashing," *ACM Trans. Database Systems*, vol. 9, no. 3, pp. 369-391, Sept. 1984.