

Extended Spiral Hashing for Expansible Files¹

YE-IN CHANG, CHIEN-I LEE* AND WANN-BAY CHANGLIAW

*Department of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan 804, R.O.C.*

E-mail: changyi@math.nsysu.edu.tw

**Institute of Information Education
National Tainan Teachers College
Tainan, Taiwan 700, R.O.C.
E-mail: leeci@ipx.ntntc.edu.tw*

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 scheme for dynamic hashing in which the growth of a file occurs at a rate of $\frac{t}{s}$ per a full expansion, where s and t are given integers and $\frac{t}{s}$ is smaller than two, as compared to a rate of two in linear hashing. (Note that s is used to denote the number of pages of a file before any split occurs in a full expansion, and t is used to denote the number of pages of the file after a full expansion is finished through a number of split operations.) Therefore, extended spiral hashing can maintain more stable performance through file expansions and has much better storage utilization than does linear hashing. Basically, the proposed scheme is based on a modified spiral storage approach, in which the load distribution is uniform after a full expansion. Therefore, extended spiral hashing can also provide better performance than can the original spiral storage approach. Moreover, we have used a modified separator strategy for overflow handling such that retrieval of any data record in extended spiral hashing is upper-bounded by two disk accesses. From our performance analysis and simulation, extended spiral hashing can achieve nearly 96% storage utilization as compared to 78% storage utilization using linear hashing and 88% storage utilization using the spiral storage approach.

Keywords: 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 [6]. 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 class needs an index, the other class does not need an index. Extendible hashing [1, 9, 20, 25, 29] and dynamic hashing [12, 33, 34] belong to the first class. Linear hashing [7, 8, 13-15, 17-19, 21, 22, 26, 30-32] and spiral storage [5, 10, 11, 24, 27, 28] belong to the second class.

Received August 5, 1996; accepted January 10, 1997

Communicated by Arbee L. P. Chen.

¹This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-82-0408-E-110-135.

Among these dynamic hashing schemes, linear hashing dispenses with the use of an index at the cost of requiring overflow pages. The first linear hashing scheme was proposed by Litwin [19]. In linear hashing, a file is expanded by adding a new page at the end of the file when a split occurs and relocating 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. With two hashing functions active at a time, a file can be expanded without reorganizing all of the records.

Since in linear hashing, all the records on a split page will be redistributed among that page and a new added page at the end of the file, storage utilization of that page will suddenly drop to only half of the original storage utilization. Moreover, this phenomenon will cause the performance in terms of access time and storage utilization to oscillate after an expansion. To maintain stable performance throughout file expansions, many strategies have been proposed [2-4, 13, 15, 18, 30]. Among these strategies, linear hashing with partial expansion as first presented by Larson [13, 15] is a generalization of Litwin's linear hashing [19]. 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 pages and the new added page (called a partial expansion). That is, doubling of the file (i.e., a full expansion) is carried out by means of a series of partial expansions. In [30], Ramamohanarao et al. proposed another way to perform partial expansions, in which data records in all of the buddy pages are redistributed into the old pages and the new added page. Larson [18], Lorentzos et al. [22] and Ruchte et al. [32] presented other strategies to maintain stable performance through out file expansions by changing the expansion sequence.

Martin's spiral storage [24, 28] is a different approach to dynamic hashing without using an index, in which the logical address space of a file can be visualized as shrinking on the left and growing on the right. That is, when a file is expanded, records in a page on the left are moved to a new larger space on the right in terms of the logical address space. Moreover, a logical to physical address mapping strategy is employed to re-use space freed on the left for physical implementation. Unified dynamic hashing [27], modified unified hashing [10] and cascading hashing [11] have also proposed based on an idea similar to that of Martin's spiral storage.

In this paper, we propose a new scheme for dynamic hashing in which the growth of a file occurs at a rate of $\frac{t}{s}$ per full expansion, where s and t are given integers and $\frac{t}{s}$ is smaller than two, as compared to a rate of two in linear hashing. (Note that s is used to denote the number of pages of a file before any split occurs in a full expansion, and t is used to denote the number of pages of the file after a full expansion is finished through a number of split operations.) Like linear hashing, the proposed scheme (called extended spiral hashing) requires no index; however, the proposed scheme may or may not add one more physical page, instead of always adding one more page in linear hashing, when a split occurs. Therefore, extended spiral hashing can maintain more stable performance through file expansions and has better storage utilization than does linear hashing. Moreover, while Martin uses an exponential spiral, our scheme uses a linear spiral. Based on an exponential spiral, the expected density of records at the left end of the file is highest and decreases from the left side to the right side of the file; i.e., the load distribution of the pages is not uniform all the time [24]. As compared to the exponential spiral approach, our extended spiral scheme not only reduces the address calculation cost, but also can provides a much

more uniform load distribution due to the linear function. To reduce the number of disk accesses for overflow records, extended spiral hashing applies separators [16], which makes use of a small in-core table to direct searching so that the records in the overflow pages can be retrieved in one disk access. Therefore, the retrieval of any record in extended spiral hashing is guaranteed to be in at most two disk accesses. From our performance analysis and simulation, extended spiral hashing can achieve nearly 96% storage utilization as compared to 78% storage utilization using linear hashing and 88% storage utilization using the spiral storage approach.

The rest of this paper is organized as follows. Section 2 describes the basic idea of extended spiral hashing. Section 3 gives a formal description of extended spiral hashing. Section 4 presents the performance analysis for extended spiral hashing. Section 5 discusses the simulation results of extended spiral hashing, and compares them with those of linear hashing [19], linear hashing with partial expansions [13], Ramamohanarao et al.'s dynamic hashing [30] and the spiral storage approach [28]. Finally, Section 6 presents conclusions.

2. BASIC IDEA

In this section, we describe the basic idea of extended spiral hashing. For convenience, we describe the case of $\frac{t}{s} = \frac{3}{2}$. In a dynamic hashing scheme without use of 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. Based on the spiral storage approach, given a data record with a key K , the physical address can be derived by the following steps:

$$K \rightarrow m(K) \rightarrow X \rightarrow \text{Logical_address}(Y) \rightarrow \text{Physical_address}(P),$$

where $m(K)$ is a hash function which distributes the records uniformly on the interval $[0, 1)$. The value of X is derived from the function $X = \lceil c - m(K) \rceil + m(K)$, where the parameter c is fixed by the file size. c increases as the file size increases. The range of X is always one unit from c to $(c + 1)$ (i.e., $X \in [c, c+1)$). During file growth or contraction, the variable c is incrementally readjusted. The function X can be seen graphically in Fig. 1. Note that as the value of c changes, the interval of the X values stays, but the starting and ending values are c and $(c + 1)$, respectively. A logical address Y is computed using a growth function $f: Y = \lfloor f(x) \rfloor$. As can be seen in Fig. 2, where $y = f(x) = 2^x$, the growth function f permits the range of X to grow as the value of parameter c increases. The effect of the function is, therefore, to increase the logical space dynamically.

In extended spiral hashing, let a split pointer *first* point to the next logical page to be split (i.e., *first* is the logical page number of the first page in the current file), and initially, let split pointer *first* point to page 0. When a file is split, the value of c is readjusted to eliminate the *first* page in the following way: $c' = f^{-1}(\text{first} + 1)$. All the records in the old *first* page are logically remapped into a new larger space at the end of the current file. Thus, both file boundaries move. A *full expansion* occurs when a split occurs in a page next to which is a new added page. A *level* (denoted as d) is defined as the number of full expansions which have occurred so far and $d = \lfloor c \rfloor$. In each level d , the pages are split in order from the small number to the large number of pages. After all the pages in the current level

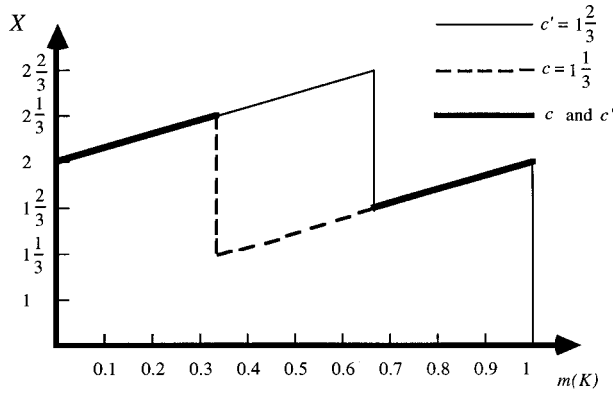


Fig. 1. The X function.

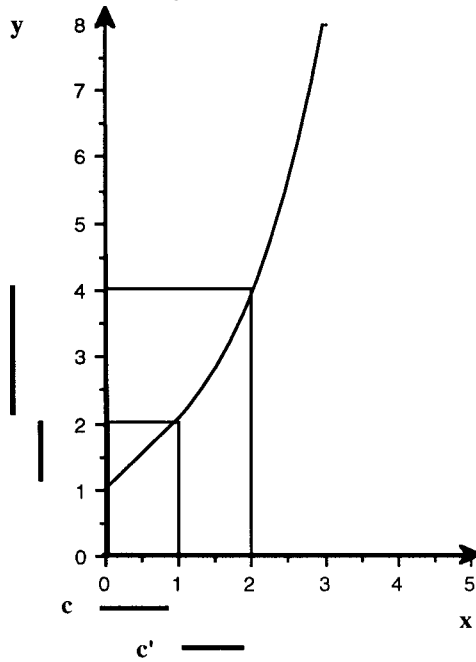


Fig. 2. The growth function $y=f(x)=2^x$.

d have been split, i.e., after a full expansion, the value of level d is increased by 1. For each level d , y_d or y_{d+1} is used to locate a page, depending on the current value of c , where y_d is the growth function used in level d . (Note that if $m(K) < (c - d)$, i.e., the page where data record with key = K is stored has been split, then y_{d+1} is applied; otherwise, y_d is used.)

In extended spiral hashing, given $\frac{t}{s} = \frac{3}{2}$ as the growth rate per full expansion and the number of initial page $s_0=2$, the growth function can be viewed as shown in Fig. 3, where each individual line represents different levels. For example, initially (i.e., level 0), there are two pages. After a full expansion occurs (i.e., after the splits in level 0 are finished),

there are $2 \times \frac{3}{2} = 3$ pages. Next, after one more full expansion occurs (i.e., after the splits in level 1 are finished), there are $2 \times (\frac{3}{2})^2 = \frac{9}{2}$ pages. Based on those individual lines in Fig. 3, we can derive the related growth functions $y = f(x)$ as shown in Fig. 4 (described in detail in Subsection 3.1) and their related inverse functions $x = f^{-1}(y)$. Table 1 shows the relationship between the growth of the logical address space Y , where $Y = \lfloor y \rfloor$, and the size of the current file n (described in detail in Subsection 3.1). (Note that both file boundaries move as n is increased.) Since many computer systems would have difficulty with a file where both boundaries moved, a logical to physical address mapping is employed to re-use space freed on the left. In our approach, we always re-use the freed physical page for the last newly added logical page as shown in Table 2, where P is the physical address and n is the size of the current file. For example, in Table 1, when n is increased from 2 to 3, logically, page 0 is split into pages 2 and 3. However, as shown in Table 2, physically, when n is increased from 2 to 3, the freed physical page 0 is re-used for the new added logical page 3 (described in detail in Subsection 3.1).

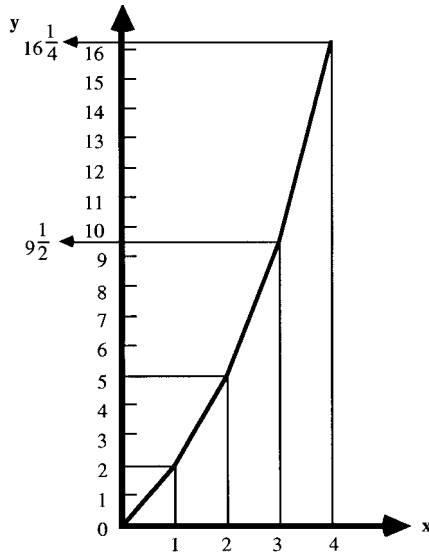


Fig. 3. The growth function $y = f(x)$ when $s_0 = 2$ and $\frac{t}{s} = \frac{3}{2}$.

$y = 2x,$	$0 \leq x < 1.$	$x = \frac{1}{2}y,$	$0 \leq y < 2.$
$y = 2(\frac{3}{2}x - \frac{1}{2}),$	$1 \leq x < 2.$	$x = \frac{1}{3}y + \frac{1}{3},$	$2 \leq y < 5.$
$y = 2(\frac{9}{4}x - 2),$	$2 \leq x < 3.$	$x = \frac{2}{9}y + \frac{8}{9},$	$5 \leq y < 9\frac{1}{2}.$
$y = 2(\frac{27}{8}x - \frac{43}{8}),$	$3 \leq x < 4.$	$x = \frac{4}{27}y + \frac{43}{27},$	$9\frac{1}{2} \leq y < 16\frac{1}{4}.$
	(a)		(b)

Fig. 4. The growth functions and inverse growth functions when $s_0=2$ and $\frac{t}{s} = \frac{3}{2}$: (a) the growth functions; (b) the inverse growth functions.

Table 1. Existing logical page numbers Y when $s_0=2$ and $\frac{t}{s} = \frac{3}{2}$.

n	c	Y																
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	0	0	1															
3	$\frac{1}{2}$		1	2	3													
3	1			2	3	4												
4	$\frac{4}{3}$				3	4	5	6										
4	$\frac{5}{3}$					4	5	6	7									
5	2						5	6	7	8	9							
5	$\frac{20}{9}$							6	7	8	9	10						
6	$\frac{22}{9}$								7	8	9	10	11	12				
6	$\frac{24}{9}$									8	9	10	11	12	13			
7	$\frac{26}{9}$										9	10	11	12	13	14	15	
7	$\frac{83}{27}$											10	11	12	13	14	15	16

Table 2. Placement of logical pages when $s_0=2$ and $\frac{t}{s} = \frac{3}{2}$.

$n \backslash P$	0	1	2	3	4	5	6
2	0	1					
3	3	1	2				
3	3	4	2				
4	3	4	6	5			
4	7	4	6	5			
5	7	9	6	5	8		
5	7	9	6	10	8		
6	7	9	12	10	8	11	
6	13	9	12	10	8	11	
7	13	9	12	10	15	11	14
7	13	16	12	10	15	11	14

In general, when an insertion causes a split, the data records in page *first* will be redistributed to pages *last* and (*last* + 1), pages *last*, (*last* + 1) and (*last* + 2), or pages (*last* + 1) and (*last* + 2) according to the value of $m(K)$, where *last* is the logical page number of the last page in the current file and is equal to $(\lceil y_d(y_d^{-1}(first) + 1) \rceil - 1)$ (described in Section 4). Moreover, there are at most $\lceil s_0(\frac{t}{s})^d \rceil$ pages in level d , where s_0 is the initial file size

and $\frac{t}{s}$ is the growth rate of file. In extended spiral hashing, since the growth rate per full expansion is smaller than 2, this scheme can maintain more stable performance through file expansions and provide better storage utilization than can linear hashing.

3. THE ALGORITHMS

In this section, given $\frac{t}{s}$, we will give a formal description of the address computation algorithm for extended spiral hashing. We will also describe retrieval, insertion, file split, deletion and file contraction algorithms used in extended spiral hashing. In these algorithms, the following variables are used globally: (1) b : the size of a home page in number of records; (2) w : the size of an overflow page in number of records; (3) $first$: the split pointer and the initial value = 0; (4) d : the level, i.e., the number of finished full expansions and the initial value = 0.

3.1 Address Computation

Let s_0 be the number of pages in the file initially, $\frac{t}{s}$ be the growth rate of a file, and l be the level number. Then, the relationship between growth function y_l for each level l and variables s_0 , $\frac{t}{s}$ and x can be computed in the following way by observing the relationship shown in Fig. 3:

$$\begin{aligned} y_0 &= s_0 x; \\ y_1 &= s_0 \left(\left(\frac{t}{s} \right) x - \left(\frac{t}{s} - 1 \right) \right); \\ y_2 &= s_0 \left(\left(\frac{t}{s} \right)^2 x - \left(\left(\frac{t}{s} \right)^2 - \frac{t}{s} + \left(\frac{t}{s} \right)^2 - 1 \right) \right); \\ y_3 &= s_0 \left(\left(\frac{t}{s} \right)^3 x - \left(\left(\frac{t}{s} \right)^3 - \left(\frac{t}{s} \right)^2 + \left(\frac{t}{s} \right)^3 - \frac{t}{s} + \left(\frac{t}{s} \right)^3 - 1 \right) \right); \\ y_l &= s_0 \left(\left(\frac{t}{s} \right)^l x - \sum_{i=0}^{l-1} \left(\left(\frac{t}{s} \right)^l - \left(\frac{t}{s} \right)^i \right) \right); \\ y_l &= s_0 \left(\left(\frac{t}{s} \right)^l x + \left(\frac{t}{s} \right)^l \left(\frac{s}{t-s} - l \right) - \frac{s}{t-s} \right). \end{aligned}$$

Let x be $l+m(K)$; then y_l can be rewritten as follows:

$$y_l = s_0 \left(\left(\frac{t}{s} \right)^l \left(\frac{s}{t-s} + m(K) \right) - \frac{s}{t-s} \right).$$

Since the value of y_l may not be an integer, we let $Y_l = \lfloor y_l \rfloor$ be the logical address. Moreover, the relationship between y_l and y_{l+1} can be derived as follows:

$$y_l = \left(\frac{t}{s} \right) y_{l-1} + s_0 \left(1 - \left(\frac{t}{s} \right)^l \right).$$

To compute the final home page number (i.e., the physical address) after d full expansions, the function *home_address* is defined as follows:

```

function home_address(K) : integer;
var
  l, Y: integer;
  c, x: real;
begin
  c =  $y_d^{-1}(\textit{first})$ ;
  l =  $\lceil c - m(K) \rceil$ ;
  x =  $l + m(K)$ ;
  Y =  $\lfloor y_l(x) \rfloor$ ;

  home_address = physical(Y);
end;

```

In this function, we have to decide whether y_d or y_{d+1} is to be used (i.e., $l = d$ or $l = d + 1$). Therefore, we must derive the current value of c first by means of the reverse growth function y_d^{-1} based on the current values of *first* and d . Then, we let x be $\lceil c - m(K) \rceil + m(K)$ and the logical address Y be $\lfloor y_l(x) \rfloor$. (Note that if $m(K) < (c - d)$, i.e., the page where the data record with key = K is stored has been split, then $l = d + 1$; otherwise, $l = d$.) Finally, we call the following function *physical*(Y) to re-use space freed on the left side as explained before:

```

function physical(Y): integer;
var
  low, high : real;
  l1, l2, anc_low, anc_high : integer;
begin
  if  $Y \leq s_0 - 1$  then
    physical = Y
  else
    begin
       $l1 = \left\lfloor \log_{\frac{t}{s}} \left( 1 + \frac{Y}{s_0} \left( \frac{t}{s} - 1 \right) \right) \right\rfloor$ ;
       $l2 = \left\lfloor \log_{\frac{t}{s}} \left( 1 + \frac{Y+1}{s_0} \left( \frac{t}{s} - 1 \right) \right) \right\rfloor$ ;
      low =  $y_{l1}^{-1}(Y)$ ;
      high =  $y_{l2}^{-1}(Y + 1)$ ;
      anc_low =  $\lfloor y_{l1-1}(\textit{low} - 1) \rfloor$ ;
      anc_high =  $\lfloor y_{l2-1}(\textit{high} - 1) \rfloor$ ;
      if anc_low < anc_high then
        physical = physical(anc_low)
      else physical =  $Y - \textit{anc\_low}$ ;
    end;
end;

```

The function $physical(Y)$, given the value of Y , has to derive the current level l . Since the growth rate of the file is $\frac{t}{s}$, the following formula shows how Y is bounded after $(l - 1)$ full expansions:

$$\begin{aligned} \left[s_0 \left(1 + \frac{t}{s} + \dots + \left(\frac{t}{s} \right)^{l-1} \right) \right] &\leq Y < \left[s_0 \left(1 + \frac{t}{s} + \dots + \left(\frac{t}{s} \right)^l \right) \right]; \\ Y &< \left[s_0 \left(1 + \frac{t}{s} + \dots + \left(\frac{t}{s} \right)^l \right) \right]; \\ Y &< s_0 \left(1 + \frac{t}{s} + \dots + \left(\frac{t}{s} \right)^l \right); \\ Y &< s_0 \left(1 - \left(\frac{t}{s} \right)^{l+1} \right) / \left(1 - \frac{t}{s} \right). \end{aligned}$$

Therefore, given a value Y , l can be computed as follows:

$$\begin{aligned} \left(\frac{t}{s} \right)^{l+1} &> 1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right); \\ l &> \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right) - 1; \\ \left\{ \begin{aligned} l &= \left\lceil \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right) - 1 \right\rceil, & \text{if } \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right) \neq \text{integer}; \\ l &= \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right), & \text{if } \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right) = \text{integer}. \end{aligned} \right. \\ \text{Therefore, } l &= \left\lceil \log_{\frac{t}{s}} \left(1 + \frac{Y}{s_0} \left(\frac{t}{s} - 1 \right) \right) \right\rceil. \end{aligned}$$

This function $physical(Y)$, given a logical page Y , determines the related physical address [28]. This requires determination of how the given logical address was instantiated. There are three possible cases: (1) it was one of the original allocations; (2) it was a re-use of a freed page on the left; (3) it was a newly allocated page. If $Y \leq (s_0 - 1)$, its physical address is Y . If the page was put into a newly allocated space, its physical address is the number of pages which existed just before its creation. If the page was put into a recycled space, one determines its recycled ancestor and then recurs to find the first allocation for the ancestor page. This process involves finding the fractional page addresses, called the ancestor range, which when deallocated can map keys to the logical page under consideration. Fig. 5 shows this process graphically. Consider page Y . the range of keys mapping to page Y extends from the lower boundary at low up to $high$. The range of keys which will be remapped into page Y lies at boundaries one below from $(low - 1)$ to $(high - 1)$. This is because the active key space range is always one unit long from c to $(c + 1)$. One can map from x back to the page ancestor addresses by means of the following computation:

$$\begin{aligned} \text{lowest_ancestor} &= y_{l-1} (low-1), \text{ and} \\ \text{highest_ancestor} &= y_{l-1} (high-1). \end{aligned}$$

The actual lower page is $anc_low = \lfloor \text{lowest_ancestor} \rfloor$, and the higher page is $anc_high = \lfloor \text{highest_ancestor} \rfloor$. The ancestor range mapping to a page is always smaller than one unit page. This is true since a deallocated logical page is always mapped

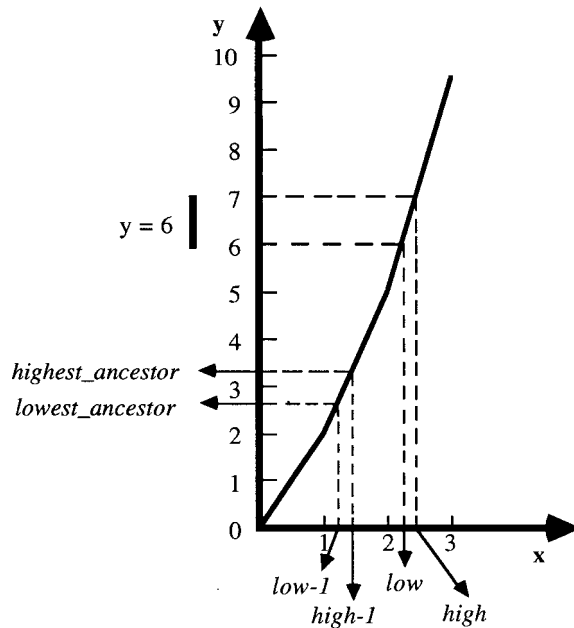


Fig. 5. Physical address mapping.

to a new larger space. When a page is totally deallocated, that space can be immediately reused. We need to determine whether the low ancestor address and high ancestor address are within the same page. Since pages always expand to a larger space, $lowest_ancestor_highest_ancestor < 1$. Thus if $anc_high > anc_low$, then page Y was instantiated from the recycled page anc_low . Otherwise, page Y was instantiated from newly allocated space.

If page Y was instantiated with a newly allocated page, one needs to know the number of active locations when it was instantiated. This is given by $(Y - anc_low)$ since Y was then the last page and anc_low was the initial page at that time. If the page was instantiated from a recycled page, the problem reduces to finding out how this recycled page was instantiated. The address is always reduced and the recursion is completed.

3.2 Overflow Handling and Retrieval

In [18], Larson applied *separators* [16] 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 [18]. Assume that all of the data records are stored in an external file consisting of n pages, and that each of these n pages has a capacity of b records. For each data record with $key = K$, its *probe sequence*, $p(K) = (p_1(K), p_2(K), \dots, p_n(K))$, ($n \geq 1$), defines the order in which the pages will be checked when a record is inserted or retrieved. That is, every *probe sequence* is a permutation of the set $\{1, 2, \dots, n\}$. For each data record with $key = K$, its *signature sequence*, $s(K) = (s_1(K), s_2(K), \dots, s_n(K))$, 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 the data records can be in $\{0, (2^q-2)\}$ [16,18]. 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 $s(K)$ is discussed in detailed in [16]. 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. 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 groups is called a *separator* and is stored in a *separator table*. The value stored is the lowest *signature* occurring among those record 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. Also, the initial values of separators are strictly greater than all the signature values. For example, using q bits as described above, the initial values of separators are set to (2^q-1) , meaning that their corresponding pages are initially empty [16, 18].)

Since in [18], overflow records are distributed among the home pages, the costs of file-split, insertion and maintaining *separators* will be high. To avoid this disadvantage and to efficiently search a data record stored in overflow pages, extended spiral hashing also applies *separators* but only to overflow pages. To apply *separators* to handle overflow pages in extended spiral 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 $p_i(K) = (p_{i1}(K), p_{i2}(K), \dots, p_{im}(K)) = (1, 2, \dots, m)$, $m \geq 1$. (Note that to increase storage utilization, we will 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 $s_i(K) = (s_{i1}(K), s_{i2}(K), \dots, s_{im}(K))$. When an overflow record of home page i with $key = K$ probes page $p_{ij}(K)$, the *signature* $s_{ij}(K)$ is used, $1 \leq j \leq m$. Moreover, when the external file for the overflow records of a home page is full, First, we have to add a page at the end of the external file. next, all the *probe sequences* and *signature sequences* of the data records on this home page and its overflow pages have to be extended and re-computed to include this newly added page. That is, the number of overflow pages for a home page, m , will be changed, depending on the number of overflow data records of a home page [18]. 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 the *separator tables* of all the home pages (which have overflow pages) may be too large for loading into main memory at one time. Moreover, to reduce the number of disk accesses needed to load a *separator table* for a certain home page which 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 where there is no change of 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* have been changed. Therefore, we can still guarantee that the cost of data retrieval will be at most two disk accesses. As shown in Fig. 6, the function $retrieval(key)$ is used to locate the actual physical address (either in a home page or in one of its related overflow pages), where $separator_{ij}$, $1 \leq j \leq m$, represents the *separator* for the j th overflow page of home page i .

```

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  $s_{ij}(key) < separator_{ij} \uparrow .value$  then
      begin
        if data record is found in page pointed by  $separator_{ij} \uparrow .pointer$ 
        then return ( $separator_{ij} \uparrow .pointer$ )
        else return (nil);
      end;
    end;
  return (nil); /*nil denotes that the record is not found */
end;
end;

```

Fig. 6. Function *retrieval*.

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 using *separators*. If the data record exists in these 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 page size b , then one of its related overflow pages is searched according to its *probe sequences*. In the case where a data record insertion causes re-location of other records in overflow pages, related *separators* which are stored in the home page may also have to be updated. Moreover, when the external file for the overflow records of a home page is full, we add a page at the end of the external file, and all the *probe sequences* and *signature sequences* of the data records in this home page and its overflow pages have to be extended and re-computed to include this newly added page. 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 *first* (including its overflow pages) have to be redistributed to pages *last* and $(last + 1)$, pages *last*, $(last + 1)$ and $(last + 2)$, or pages $(last + 1)$ and $(last + 2)$, depending on the value of $m(K)$, where *last* is the logical page number of the last page in

the current file. Then, *first* is increased by one, and new level *d* is computed. The results of the above actions are equal to updating *first* (and *d*) first and then re-inserting those data records which are in the page where the old *first* points to by using the new hashing function y_{d+1} . Procedure *file_split* is shown in Fig. 7. (Note that to reduce the number of disk accesses, we use a buffer mechanism to reduce the overhead of re-insertion. That is, we first perform re-insertion in a buffer. Then, we write one page back to disk from the buffer at a time instead of re-inserting one data record back to disk at a time in the process of re-insertion.)

3.4 Deletion and File Contraction

When a data record is deleted, we immediately try to move another data record in to fill the hole left by the deleted data record. There are two cases. First, if the deleted data record is stored in a home page which has overflow pages, then we only move one of the data records stored in the last overflow page (i.e., overflow page *m*) back to the home page and fill the hole; in addition, the *separators* must be updated. Second, if the deleted data record is stored in overflow page *i* ($1 \leq i \leq m$), we should move one of the data records from overflow page (*i* + 1) back to fill the hole created by the deletion in overflow page *i*. In the same way, the hole created by the above movement in overflow page (*i* + 1) will be filled by moving one of the data records in to overflow page (*i* + 2). This process will not be terminated until one of the data records in overflow page *m* is moved back to overflow page (*m* - 1). In this process, the related *separators* must also be updated.

Whenever the number of deleted data records exceeds a predetermined contracted control condition as in file split, a contraction occurs. In this case, we should collect the data records which have been redistributed in the last file split operation and move them back to page (*first* - 1). Since when a split occurs in page (*first* - 1), the data records in page (*first* - 1) (including its overflow pages) have to be redistributed to pages (*last* - 1) and *last*, or (*last* - 2), (*last* - 1) and *last*, depending on the value of $m(K)$, where *last* is the logical page number of the last page in the current file, when contraction occurs, we should collect these data records which were stored in page (*first* - 1) before the last file split but now are stored in the pages mentioned above, and move these data records back to page (*first* - 1). The results of the above actions are equal to updating *first* (and *d*) first and then re-inserting all the data records stored in those above pages. (Note that the data records in pages *last*, (*last* - 1) and (*last* - 2) may not be moved out from page (*first* - 1), and that we do not record any information about the original page for each data record in a page; therefore, we have to compute the home address for each data record in these pages to determine its original page from which the data record was moved out.) Procedure *file_contraction* is shown in Fig. 8.

4. PERFORMANCE ANALYSIS

In all dynamic hashing schemes which do not use an index, a split occurs under certain conditions. There are two kinds of strategies [6,19]: uncontrolled and controlled splitting. Uncontrolled splitting means that a 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

```

procedure file_split();
  var i, j : integer;
      B : buffer;
begin
  read page first and its overflow pages into buffer B;
  set page first and its overflow pages to empty;
  first = first + 1;
   $d = \left\lfloor \log_{\frac{t}{s}} \left( 1 + \frac{\textit{first}}{s_0} \left( \frac{t}{s} - 1 \right) \right) - 1 \right\rfloor$ ;
  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
             $s_{ij}(K) < \textit{separator}_{ij} \uparrow . \textit{value}$ ;
          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  $s_{ij}(K) < \textit{separator}_{ij} \uparrow . \textit{value}$  do
            begin
              if the page pointed by  $\textit{separator}_{ij} \uparrow . \textit{pointer}$  is full then
                move out the record whose key is  $\textit{separator}_{ij} \uparrow . \textit{value}$  to Buffer B;
                write the data record with key = K to the overflow page pointed
                  by  $\textit{separator}_{ij} \uparrow . \textit{pointer}$ ;
                updated  $\textit{separator}_{ij} \uparrow . \textit{value}$  if necessary;
            end;
          end;
        end;
    end;
end;
end;
end;

```

Fig. 7. Procedure *file_split*.

```

procedure file_contraction();
  var i, j : integer;
  B: buffer;
begin
   $last = \lceil y_d(y_d^{-1}(first) + 1) \rceil - 1$ ;
  read page (last - 2), page (last - 1) and last
  and their overflow pages into buffer B;
  set page (last - 2), page (last - 1) and last
  and their overflow pages to empty;
  first = first - 1;
   $d = \left\lfloor \log_{\frac{t}{s}} \left( 1 + \frac{first}{s_0} \left( \frac{t}{s} - 1 \right) \right) - 1 \right\rfloor$ ;
  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  $s_{ij}(K) < separator_{ij} \uparrow .value$ ;
        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  $s_{ij}(K) < separator_{ij} \uparrow .value$  do
        begin
          if the page pointed by  $separator_{ij} \uparrow .pointer$  is full then
            move out the record whose key is  $separator_{ij} \uparrow .value$  to Buffer B;
            write the data record with key = K to the overflow page pointed
            by  $separator_{ij} \uparrow .pointer$ ;
            updated  $separator_{ij} \uparrow .value$  if necessary;
          end;
        end;
      end;
  end;
end;

```

Fig. 8. Procedure *file_contraction*.

load control denotes the upper bound of the number of newly 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 can the uncontrolled strategy, as verified in [19]. Moreover, when the load factor is used as the split control strategy, the system will suffer from more unstable performance during a full expansion as stated in [13, 30]. Therefore, we prefer to use the load control as the split control strategy as in [30,31].

In this section, we will present the results of performance analysis of extended spiral hashing using the load control strategy. In this performance analysis model, we assume that the keys for data records are uniformly distributed and independent of each other, and that the page size is measured in terms of the number of record slots. The hash function distributes the records uniformly in the interval $[0, 1)$. 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 q , $q \geq 0$, overflow records, then there will be $\left\lceil \frac{q}{w} \right\rceil$ overflow pages for this home page. When the search cost is computed, all the 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, *first*, the average retrieval cost and storage utilization; that is, we are able to 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

$$ns(N) = 0, \quad 0 \leq N \leq s_0 L,$$

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

(Note that to reduce the number of splits, we assume that the split control is not started until the first s_0 pages are filled with $s_0 L$ records in the performance analysis.) In extended spiral hashing, the value of *first* is equal to the number of splits, i.e., $first = ns(N)$. Since in extended spiral hashing, the growth rate of a file is $\frac{t}{s}$, the number of levels can be computed using the following formula as derived in Section 3.1, given $Y = first = ns(N)$:

$$d = \left\lceil \log_{\frac{t}{s}} \left(1 + \frac{first}{s_0} \left(\frac{t}{s} - 1 \right) \right) \right\rceil.$$

Since the range of X for the current file is always between c and $(c + 1)$, and $y_d(c) = first = ns(N)$, based on the current value of *first*, we can derive the values of *median* and *last* in the following way: (Note that the value of *median* is the last logical page number in level d , and that the value of *last* is the last logical page number of the current file.)

$$c = y_d^{-1}(first);$$

$$d' = d + 1;$$

$$median = \lceil y_{d'}(\lfloor c + 1 \rfloor) \rceil - 1;$$

$$last = \lceil y_{d'}(c + 1) \rceil - 1.$$

Consequently, the number of pages of the current file is $(last - first + 1)$.

Given the value of *first*, the probability $Pr(first, i)$ of the load distributions for logical pages *i*, where $first \leq i \leq last$, is different in extended spiral hashing as shown in Table 3. To compute the probability, two classes are considered. For the first class, all of the pages of the current file are in the same level *d*; i.e., it satisfies the condition (*median = last*). For the second class, some of the pages are in level *d*, and some of the pages are in level (*d + 1*), i.e., *median ≠ last*.

Table 3. The probability of load distribution.

level (<i>l</i>)	<i>first</i>	logical page number (<i>Y</i>)																
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	1/2	1/2															
	1		1/2	1/3	1/6													
1	2			1/3	1/3	1/3												
	3				1/3	1/3	2/9	1/9										
	4					1/3	2/9	2/9	2/9									
2	5						2/9	2/9	2/9	2/9	1/9							
	6							2/9	2/9	2/9	5/27	4/27						
	7								2/9	2/9	5/27	4/27	4/27	2/27				
	8									2/9	5/27	4/27	4/27	4/27	4/27			
	9											5/27	4/27	4/27	4/27	4/27	4/27	2/27
3	10												4/27	4/27	4/27	4/27	4/27	1/9

The probability $Pr(first, i)$ of the first class can be divided into the following two subclasses:

$$Pr(first, i) = \frac{1}{s_0 (\frac{l}{s})^d}, \quad first \leq i < last;$$

$$Pr(first, i) = 1 - \frac{last - first}{s_0 (\frac{l}{s})^d}, \quad i = last.$$

In class one, all of the pages are not split yet in the current level *d*, and the last page may not be full because $y_d^{-1}(\lfloor c + 1 \rfloor)$ may not be an integer. Therefore, the probability $Pr(first, i)$ of each of the pages between *first* and (*last - 1*) is the same as the probability after *d* full expansions. For page *last*, $Pr(first, last)$ is equal to the remaining value.

The probability $Pr(first, i)$ of the second class can be divided into the following four subclasses:

$$\begin{aligned}
Pr(first, i) &= \frac{1}{s_0 \left(\frac{t}{S}\right)^d}, & first \leq i < median; \\
Pr(first, i) &= y_d^{-1}(median + 1) - y_d^{-1}(median), & i = median; \\
Pr(first, i) &= \frac{1}{s_0 \left(\frac{t}{S}\right)^{d'}}, & median + 1 \leq i < last; \\
Pr(first, i) &= c + 1 - y_{d'}^{-1}(last), & i = last.
\end{aligned}$$

The pages between $first$ and $(median-1)$ are not split yet in the current level d ; therefore, the probability $Pr(first, i)$ of each of these pages is still the same as the probability after d full expansions. Although page $median$ is the last page in level d , it may contain some records which need to be added during the $(d + 1)$ 'th full expansion. (Note that this is because $y_{d'}^{-1}(\lfloor c + 1 \rfloor)$ may or may not be an integer.) Therefore, the probability of page $median$ is the length from $y_{d'}^{-1}(median)$ to $y_{d'}^{-1}(median + 1)$. (Note that since the interval of X values stays between c and $c + 1$, the length in the x -axis represents the probability.) The pages between $(median + 1)$ and $(last - 1)$ are newly added during the process of the $(d + 1)$ 'th full expansion; therefore, the probability of each of these pages is the same as the probability after $(d + 1)$ full expansions. For page $last$, $Pr(first, last)$ is equal to the length from $y_{d'}^{-1}(last)$ to $(c + 1)$, i.e., the remaining value.

After computing the probability $Pr(first, i)$ for each page i of the current file, we can start to analyze the other performance measures. Let $W(q)$ be a function used to denote the number of overflow pages of a home page with q data records inserted and let it be defined as follows:

$$\begin{aligned}
W(q) &= 0, & 0 \leq q \leq b, \\
W(q) &= j, & (b + (j-1)w + 1) \leq q \leq (b + jw).
\end{aligned}$$

Let $\text{Bin}(q; N, P)$ denote the binomial distribution, i.e., $\text{Bin}(q; N, P) = \binom{N}{q} P^q (1-P)^{N-q}$. The probability of logical page i ($first \leq i \leq last$) containing q data records is $\text{Bin}(q; N, Pr(first, i))$. The expected number of overflow pages for logical page i is obtained as

$$OP_i(N) = \sum_{q=0}^N (W(q) \text{Bin}(q; N, Pr(first, i))).$$

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

$$OP(N) = \frac{\sum_{i=first}^{last} OP_i(N)}{last - first + 1}$$

and storage utilization can be obtained as follows:

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

Using *separators* to overflow records, the expected cost of an unsuccessful search for home page i ($first \leq i \leq last$) in terms of the number of disk accesses is

$$\begin{aligned} US_i &= 1, & OP_i &= 0, \\ US_i &= 2, & OP_i &> 0. \end{aligned}$$

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

$$US(N) = \sum_{i=first}^{last} (US_i(N) Pr(first, i)).$$

For an successful search, we first consider the expected number of disk accesses needed to retrieve all the data records in home page i ($first \leq i \leq last$) plus its overflow pages, which can be obtained by

$$\begin{aligned} RA_i(N) &= \sum_{q=0}^b (q \text{Bin}(q; N, Pr(first, i))) \\ &+ \sum_{q=b+1}^N ((q + (q - b)) \text{Bin}(q; N, Pr(first, i))). \end{aligned}$$

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

$$SS(N) = \frac{\sum_{i=first}^{last} RA_i(N)}{N}.$$

For the average insertion cost, we first consider the split cost at the insertion of the q th ($q \leq N$) data record, which is given by

$$SC(q) = 1 + OP(q) + 2(1 + OP(q + 1)).$$

(Note that since we apply a buffer mechanism, $(1 + OP(q))$ disk accesses are needed to read the split page and its overflow pages into the buffer, and $2(1 + OP(q + 1))$ disk accesses are needed to write the split results.) Since a split occurs only when q is $L, 2L, \dots, ns(N)L$ ($ns(N)L \leq N$), the total split cost for N inserted data records can be obtained by

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

Then, we can consider the average cost of inserting a data record when there are q data records which have been inserted. (Note that given the number of data records q , 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 re-inserted, the average number of disk accesses needed to insert the $(q + 1)$ th data record in page i is as follows:

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

Then, the average number of disk accesses needed to insert a data record in any page i among these $(s' + 1)$ pages is given by

$$AC(q) = \sum_{i=0}^{s'} P(sp', i, s') AC_i(q).$$

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

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

Table 4 -(a) shows the results derived using the above formulas, where $s = 2$, $t = 3$, $s_0 = 2$, $N = 10^6$, $b = 10, 20, 40$ and 80 , $w = 0.5 * b$ and $L = 0.8 * b$, $L = b$ and $L = 1.2 * b$ in extended spiral hashing. From this table, we observe that storage utilization can reach nearly 97%, where the cost of successful and unsuccessful search is given in terms of the number of disk accesses.

Table 4. Performance: (a) analysis results; (b) simulation results.

Parameters			Analysis		Results	
b	w	L	INS	ss	us	uti
10	5	08	4.5	1.332	1.970	0.920
10	5	10	5.4	1.438	2.0	0.968
10	5	12	6.3	1.539	2.0	0.969
20	10	16	3.3	1.370	1.949	0.954
20	10	20	4.4	1.478	1.989	0.900
20	10	24	5.4	1.579	1.990	0.902
40	20	32	3.4	1.380	1.988	0.792
40	20	40	4.2	1.482	2.0	0.868
40	20	48	4.3	1.520	2.0	0.842
80	40	64	2.9	1.292	1.980	0.844
80	40	80	3.4	1.420	1.987	0.898
80	40	96	3.8	1.528	2.0	0.898

Parameters			Simulation		Results	
b	w	L	INS	ss	us	uti
10	5	08	4.6	1.364	2.0	0.919
10	5	10	5.3	1.489	2.0	0.970
10	5	12	6.0	1.574	2.0	0.970
20	10	16	3.9	1.351	1.995	0.956
20	10	20	4.6	1.479	2.0	0.896
20	10	24	5.3	1.559	2.0	0.885
40	20	32	3.5	1.339	2.0	0.793
40	20	40	4.0	1.459	2.0	0.869
40	20	48	4.5	1.543	1.978	0.943
80	40	64	3.0	1.294	1.983	0.847
80	40	80	3.5	1.414	1.966	0.892
80	40	96	3.8	1.519	2.0	0.892

(a)

b : the size of a home page
w : the size of an overflow page
L : load control

(b)

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

5. SIMULATION RESULTS

In this section, we will present the simulation results of extended spiral hashing, linear hashing [19], linear hashing with partial expansions [13], Ramamohanarao's dynamic hashing [31] and the spiral storage approach [28], under two different split control strategies. (Note that linear hashing with partial expansions and Ramamohanarao's dynamic

hashing were proposed to improve the retrieval performance of linear hashing since they can provide more stable performance than can linear hashing during a full expansion. The growth rate for each partial expansion is not a constant in Larson's linear hashing with partial expansions while it is a constant $\frac{g+1}{g}$ in Ramamohanaro's dynamic hashing when g is the number of pages per group. For example, in Larson's linear hashing with partial expansions, in the case of one full expansion consisting of two partial expansions, the growth rate is $\frac{3}{2}$ during the first partial expansion and $\frac{4}{3}$ during the second. However, Ramamohanarao's dynamic hashing will result in many round-up pages.)

In this simulation study [30], we assumed that N input data records were uniformly distributed. The environment control variables were the size of a home page (b) and the size of an overflow page (w) and a load control (L), which controlled when a split would occur. In this simulation, storage utilization and the average number of disk accesses for successful and unsuccessful searches were the main performance measures considered. Moreover, overflow pages were handled by *separators* in all of these approaches. Where the average successful/unsuccessful search cost was concerned, we considered $2N$ search requests, where N searched data records were present in the file and the other N searched data records were absent. Where the average insertion cost was concerned, we considered the average result of 10 random different insertion sequences. In the simulation study of the spiral storage approach, we let the parameter β in the growth function $y = \beta^x$ be $\frac{t}{s}$ since the curve $y = (\frac{t}{s})^x$ would approximate to the line constructed by our proposed growth function as shown in Fig. 3.

Table 4 -(b) shows the simulation results of extended spiral hashing, where $s = 2$, $t = 3$, $s_0 = 2$, $N = 10^6$, $w = 0.5*b$ and $L = 0.8*b$, $L = b$ and $L = 1.2*b$, respectively. Comparing them with the analysis results shown in Table 4 -(a), the simulation results shown in Table 4 -(b) are very similar.

Simulation results of extended spiral hashing with $\frac{t}{s} = \frac{3}{2}$, the spiral storage approach [28] with $\beta = \frac{3}{2}$, linear hashing [19], linear hashing with two partial expansions per full expansion [13] and Ramamohanarao's dynamic hashing [31] with $g = 2$ under split control of the load control L are shown in Tables 5-(a), (b), (c), (d) and (e), respectively, where $N = 10^6$, $w = 0.5b$ and $L = 0.8b$, $L = b$ and $L = 1.2b$. From these tables, we observe that as the size of home pages and of an overflow pages increased, storage utilization could decrease in all five methods. The reason is that the larger the size of a page was, the larger the average unused space in a home page or an overflow page could be, which resulted in a decrease of storage utilization. Extended spiral hashing had the highest storage utilization among these five methods. Compared to the spiral storage approach [28], extended spiral hashing not only reduced the cost for address calculation, but also had much uniform load distribution due to the linear growth function, which resulted in higher storage utilization and a lower average insertion cost.

Moreover, when $N = 10^6$, $b = 20$, $w = 10$, and $L = 16$, extended spiral hashing could achieve 96% storage utilization, as compared to 78% storage utilization in linear hashing and 88% storage utilization in the spiral storage approach under the same conditions. As mentioned before, linear hashing with partial expansions and Ramamohanarao's dynamic

Table 5. Simulation results under split control of the load control (L): (a) extended spiral hashing hashing; (b) the spiral storage (c) linear hashing; (d) linear hashing with two partial expansions; (e) Ramamohanarao's dynamic hashing.

Parameters			Extended Spiral Hashing			
b	w	L	INS	ss	us	uti
10	5	08	4.6	1.364	2.0	0.919
10	5	10	5.3	1.489	2.0	0.970
10	5	12	6.0	1.574	2.0	0.970
20	10	16	3.9	1.351	1.995	0.956
20	10	20	4.6	1.479	2.0	0.896
20	10	24	5.3	1.559	2.0	0.885
40	20	32	3.5	1.339	2.0	0.793
40	20	40	4.0	1.459	2.0	0.869
40	20	48	4.5	1.543	1.978	0.943
80	40	64	3.0	1.294	1.983	0.847
80	40	80	3.5	1.414	1.966	0.892
80	40	96	3.8	1.519	2.0	0.892

(a)

Parameters			Spiral Storage Approach			
b	w	L	INS	ss	us	uti
10	5	08	4.7	1.374	2.0	0.894
10	5	10	5.4	1.499	2.0	0.898
10	5	12	6.2	1.585	2.0	0.917
20	10	16	4.2	1.372	1.995	0.884
20	10	20	5.0	1.499	2.0	0.892
20	10	24	5.8	1.579	1.993	0.904
40	20	32	3.9	1.379	2.0	0.862
40	20	40	4.7	1.499	2.0	0.869
40	20	48	5.6	1.579	1.973	0.892
80	40	64	3.8	1.372	1.964	0.847
80	40	80	4.6	1.486	1.978	0.862
80	40	96	5.4	1.599	2.0	0.862

(b)

Parameters			Linear Hashing			
b	w	L	INS	ss	us	uti
10	5	08	2.7	1.015	1.047	0.790
10	5	10	3.1	1.144	1.445	0.858
10	5	12	3.5	1.246	1.705	0.858
20	10	16	2.5	1.016	1.046	0.781
20	10	20	2.9	1.153	1.438	0.784
20	10	24	3.3	1.247	1.704	0.784
40	20	32	2.4	1.022	1.062	0.781
40	20	40	2.7	1.154	1.438	0.781
40	20	48	3.1	1.256	1.717	0.781
80	40	64	2.3	1.022	1.062	0.781
80	40	80	2.7	1.157	1.436	0.781
80	40	96	3.0	1.269	1.749	0.781

(c)

Parameters			Linear Hashing with two Par. Exp.			
b	w	L	INS	ss	us	uti
10	5	08	3.0	1.074	1.879	0.800
10	5	10	3.2	1.149	1.888	0.863
10	5	12	3.7	1.244	1.999	0.864
20	10	16	2.6	1.017	1.050	0.790
20	10	20	3.0	1.155	1.459	0.809
20	10	24	3.3	1.344	1.800	0.828
40	20	32	2.4	1.039	1.003	0.787
40	20	40	2.6	1.499	2.0	0.808
40	20	48	3.1	1.299	2.0	0.812
80	40	64	2.2	1.099	1.070	0.783
80	40	80	2.6	1.178	1.680	0.803
80	40	96	3.0	1.199	2.0	0.809

(d)

Parameters			Ramamohanarao's dynamic hashing			
b	w	L	INS	ss	us	uti
10	5	08	3.0	1.058	1.999	0.715
10	5	10	3.3	1.117	1.999	0.790
10	5	12	3.6	1.205	1.999	0.836
20	10	16	2.6	1.012	1.000	0.743
20	10	20	2.8	1.099	1.001	0.803
20	10	24	3.2	1.182	1.001	0.819
40	20	32	2.3	1.030	1.000	0.709
40	20	40	2.6	1.052	1.999	0.787
40	20	48	3.0	1.178	1.999	0.813
80	40	64	2.2	1.013	1.000	0.714
80	40	80	2.5	1.078	1.000	0.781
80	40	96	3.0	1.142	1.999	0.819

(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

hashing were proposed in order to improve the retrieval performance of linear hashing. Compared to these two methods used to improved the performance of linear hashing, our extended spiral hashing has achieved the same goal of partial expansions, i.e., to maintain stable performance throughout file expansions; in addition, our method uses a fixed growth rate and requires no round-up pages.

Fig. 9 shows the change of storage utilization as a function of the number of inserted data records, where $s = 2$, $t = 3$, $b = 10$, $w = 5$, and $L = 8$. From Fig. 9, we observe that storage utilization in the spiral storage approach was a little bit more stable than was that in extended spiral hashing. The reason is that the exponential growth function in the spiral storage approach kept the distribution of records across pages fixed during a full expansion, which provide constant performance as stated in [27,28] while the load distribution in extended spiral hashing as shown in Table 3 was not fixed during a full expansion. Moreover, storage utilization in extended spiral hashing was much more stable than was that in linear hashing. The reason is that when a split occurred, linear hashing always redistributed the data records of a certain page i into page i and a newly added empty page. The property of stable storage utilization in extended spiral hashing distributed the overhead of insert/split operations uniformly as data records were inserted while unstable storage utilization in linear hashing could suddenly cause a large overhead of insert/split operations.

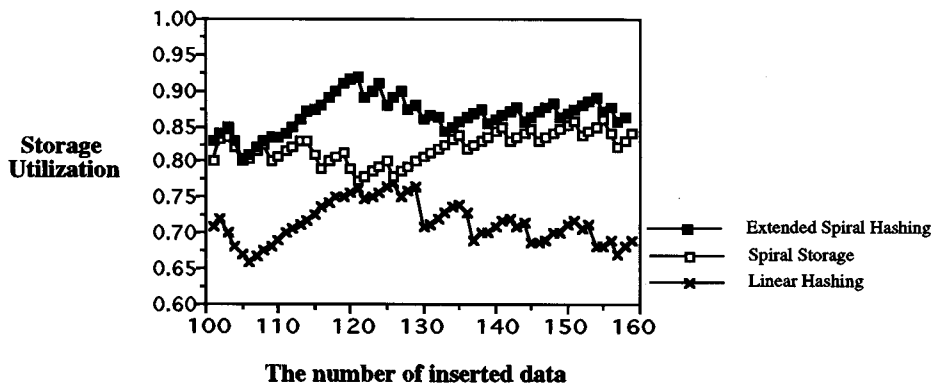


Fig. 9. The relationship between storage utilization and the number of inserted data records.

To compare the average insertion/retrieval cost in linear hashing and extended spiral hashing when both approaches achieved the same storage utilization, we tried to run linear hashing under different choices of L . Table 6 shows that storage utilization in linear hashing could increase as L was increased at the cost of increasing the average retrieval cost and average insertion cost, where $b = 10$, $w = 5$, $\frac{t}{s} = \frac{3}{2}$ and $N = 10^6$. From this table, we observe that when both approaches had the same storage utilization ($L = 70$ in linear hashing), extended spiral hashing could have a lower average retrieval cost and a lower average insertion cost compared to linear hashing.

The reason is that as L increased significantly in linear hashing, the number of file splits decreased. Therefore, given a fixed N and the same storage utilization, the number of home pages in linear hashing was smaller than that in extended spiral hashing. At the same

time, the number of overflow pages in linear hashing was greater than that in extended spiral hashing. Consequently, the average retrieval cost and the average insertion cost in extended spiral hashing were better than those in linear hashing. Moreover, when both approaches had a similar average insertion cost ($L = 20$ in linear hashing), extended spiral hashing had a lower average successful search cost and higher storage utilization than did linear hashing. When both approaches had a similar average retrieval cost ($L = 16$ in linear hashing), extended spiral hashing had higher storage utilization and a higher average insertion cost than did linear hashing.

Table 6. The relationship between performance and L in linear hashing.

Load Control	INS	ss	us	uti
L = 08	2.73	1.015	1.047	0.790
L = 10	3.12	1.144	1.445	0.858
L = 12	3.54	1.246	1.725	0.858
L = 16	4.56	1.374	2.0	0.860
L = 20	5.38	1.499	2.0	0.863
L = 40	9.91	1.749	2.0	0.923
L = 60	14.45	1.834	2.0	0.959
L = 65	15.73	1.849	2.0	0.966
L = 70	16.90	1.859	2.0	0.970
Ext. Spiral L = 10	5.35	1.364	2.0	0.970

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

Tables 7-(a), (b) and (c) show the simulation results of extended spiral hashing with $\frac{t}{s} = \frac{3}{2}$, the spiral storage approach with $\beta = \frac{3}{2}$ and linear hashing under split control of the load factor (A), where $N = 10^6$, $b = 10$ and $w = 5$. In all three methods, as A increased from 0.5 to 0.95, the average insertion cost increased. The reason is that as A increased, the number of overflow pages increased. Since extended spiral hashing could provide a much uniform load distribution, extended spiral hashing had a lower average insertion cost and a better retrieval performance than did the spiral storage approach. Compared to linear hashing, extended spiral hashing had a higher average insertion cost because the number of file splits in extended spiral hashing was larger than that in linear hashing.

Moreover, as A increased, which implied that the storage utilization threshold increased, oscillation in performance during a full expansion increased as stated in [13,19]. Since the growth rate ($\frac{t}{s} = \frac{3}{2}$) of extended spiral hashing was smaller than 2 in linear hashing, extended spiral hashing resulted in smaller oscillation during a full expansion compared to linear hashing. When $A > 0.85$, extended spiral hashing could have higher storage utilization than could linear hashing. The reason is that the higher A was, the higher was the ratio of performance oscillation during a full expansion in linear hashing to that in extended spiral hashing.

Table 7. Simulation results under split control of the load factor (A): (a) extended spiral hashing; (b) the spiral storage; (c) linear hashing.

Parameters	Extended Spiral Hashing			
	A	INS	ss	us
0.50	3.8	1.000	1.000	0.500
0.55	3.7	1.000	1.000	0.549
0.60	3.6	1.000	1.000	0.598
0.65	3.5	1.000	1.000	0.649
0.70	3.5	1.000	1.000	0.699
0.75	3.6	1.000	1.000	0.746
0.80	3.6	1.104	1.775	0.800
0.85	4.2	1.144	1.983	0.849
0.90	5.8	1.500	2.0	0.899
0.95	10.8	1.720	2.0	0.947

(a)

Parameters	Spiral Storage			
	A	INS	ss	us
0.50	3.8	1.000	1.000	0.500
0.55	3.7	1.000	1.000	0.549
0.60	3.6	1.000	1.000	0.598
0.65	3.5	1.000	1.000	0.649
0.70	3.6	1.000	1.000	0.699
0.75	3.7	1.000	1.000	0.746
0.80	4.2	1.000	1.000	0.800
0.85	4.8	1.259	2.0	0.843
0.90	6.7	1.510	2.0	0.896
0.95	13.3	1.780	2.0	0.938

(b)

Parameters	Linear		Hashing	
	A	INS	ss	us
0.50	2.6	1.000	1.000	0.500
0.55	2.5	1.000	1.000	0.549
0.60	2.6	1.000	1.000	0.598
0.65	2.6	1.000	1.000	0.649
0.70	2.7	1.000	1.000	0.699
0.75	2.8	1.000	1.000	0.746
0.80	3.1	1.029	1.096	0.800
0.85	3.3	1.099	1.320	0.847
0.90	3.9	1.333	1.919	0.858
0.95	5.0	1.670	2.0	0.888

(c)

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

Fig. 10-(a) and (b) show the simulation results of extended spiral hashing with different values of the growth rate $\frac{t}{s}$, where $N = 10^6$, $b = 10$, $w = 5$, and $L = 10$, and where the value of $\frac{t}{s}$ was 1.1, 1.2, ..., 1.9, respectively. From these figures, we observe that as the value of $\frac{t}{s}$ increased, storage utilization decreased and the average insertion cost decreased. The reason is that the higher the growth rate was, the larger was the number of home pages appended to the file after a full expansion, resulting in a decrease in storage utilization and a decrease in the number of overflow records; therefore, the average insertion cost decreased.

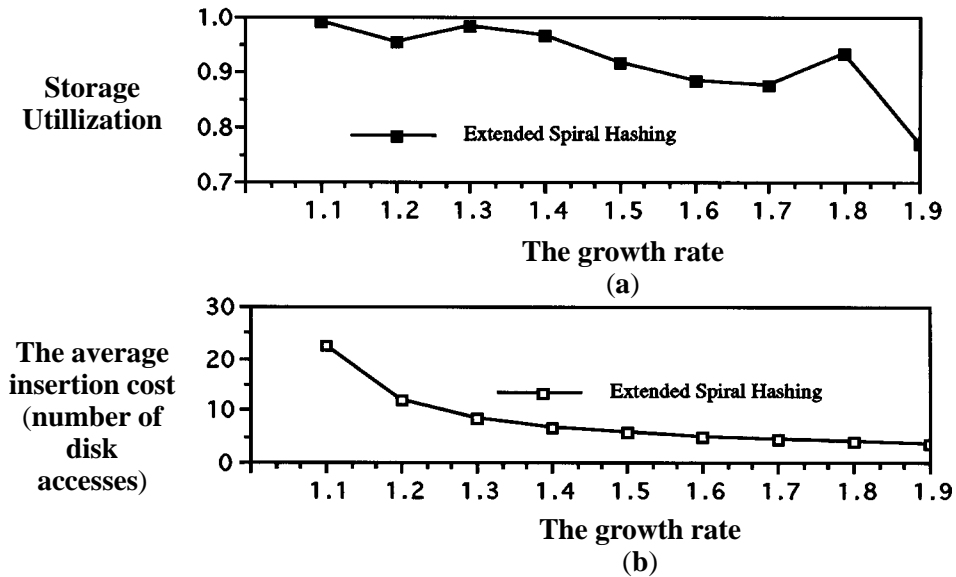


Fig. 10. (a) The relationship between storage utilization and the growth rate; (b) the relationship between the average insertion cost and the growth rate.

6. CONCLUSIONS

In this paper, we have proposed a new scheme (called extended spiral hashing) for dynamic hashing in which the growth rate of a file is $\frac{t}{s}$ per full expansion, which is smaller than two, as compared to a rate of two in linear hashing. Since the growth rate of a file is smaller than two, extended spiral hashing can provide better storage utilization than can linear hashing [19]. Moreover, extended spiral hashing can maintain more stable performance through out file expansions than can linear hashing. From our mathematical analysis and simulation study, extended spiral hashing can achieve nearly 96% storage utilization as compared to 78% storage utilization obtained using linear hashing. Compared to schemes based on the spiral storage approach [28], extended spiral hashing not only can reduce the cost of address calculation, but also has a much uniform load distribution due to the linear growth function, which results in higher storage utilization. Moreover, to compute the logical addresses, no history of split sequence need be traced, and only one variable is needed to be recorded in extended spiral hashing (i.e., *first* or *c*) instead of a table of indexes in [27] or a sequence of split points in [10]. Furthermore, extended spiral hashing has a systematic way of handling file contraction.

In [13], Larson improved the performance of linear hashing by using generalized linear hashing, where a full expansion was done in a series of partial expansions, which smoothed the performance oscillation through out file expansions. (Note that in Larson's linear hashing with partial expansions [13], the growth rate for each partial expansion is not a constant.) In [30], Ramamohanarao further improved the performance of Larson's linear

hashing with partial expansions [13] by fixing the growth rate of a file to be a constant $\frac{g+1}{g}$, where g is the number of pages per group. However, Ramamohanarao's approach to partial expansions resulted in many round-up pages. Compared with these two methods used to improved the performance of linear hashing, our extended spiral hashing has achieved the same goal of partial expansions, i.e., to maintain stable performance through out file expansions; at the some time our method uses a fixed growth rate and requires no round-up pages.

Therefore, our extended spiral hashing also can provide better performance compared to these methods. How to combine our extended spiral hashing with a different expanding sequence using, for example, *priority splitting* [22,32], to further improve storage utilization is a subject of future research.

REFERENCES

1. U. Bechtold and K. Kuspert, "On the use of extendible hashing without hashing," *Information Processing Letters*, Vol. 19, No. 1, 1984, pp. 21-26.
2. Y.I. Chang and C.I. Lee, "An incremental approach to dynamic hashing for expansible files," *Journal of Information Science and Engineering*, Vol. 9, No. 4, Dec. 1993, pp. 495-522.
3. Y.I. Chang and C.I. Lee, "Climbing hashing for expansible files," *Information Sciences: an International Journal*, Vol. 86, No. 9, Sept. 1995, pp. 77-99.
4. Y.I. Chang and C.I. Lee, "Alternating hashing for expansible files," accepted by *IEEE Transaction on Knowledge and data Engineering*, Vol. 9, No. 1, Feb. 1997, pp. 179-185.
5. J.H. Chu and G.D. Knott, "An analysis of spiral hashing," *The Computer Journal*, Vol. 37, No. 8, 1994, pp. 715-719.
6. R.J. Enbody and H.C. Du, "Dynamic hashing schemes," *ACM Computing Surveys*, Vol. 20, No. 2, June 1988, pp. 85-113.
7. N.I. Hachem and P.B. Berra, "Key-sequential access method for very large files derived from linear hashing," in *Proceedings of the 5th International Conference on Data Engineering*, 1989, pp. 305-312.
8. N.I. Hachem and P.B. Berra, "New order preserving access method for very large files derived from linear hashing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 1, February 1992, pp. 68-82.
9. R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, "Extendible hashing-A fast access method for dynamic files," *ACM Transactions on Database Systems*, Vol. 4, No. 3, Sept. 1979, pp. 315-344.
10. K. Kawagoe, "Modified dynamic hashing," in *Proceedings of the 6th ACM-Transactions on Database Systems, International Conference on Management of Data*, 1985, pp. 201-213.
11. P. Kjellberg and T.U. Zahle, "Cascade hashing," in *Proceedings of the 10th International Conference on Very Large Data Bases*, 1984, pp. 481-492.
12. P. Larson, "Dynamic hashing," *BIT*, Vol. 18, No. 3, 1978, pp. 184-201.
13. P. Larson, "Linear hashing with partial expansions," in *Proceedings of the 6th International Conference on Very Large Data Bases*, 1980, pp. 224-232.

14. P. Larson, "A single-file version of linear hashing with partial expansions," in *Proceedings of the 8th International Conference on Very Large Data Bases*, 1982, pp. 300-309.
15. P. Larson, "Performance analysis of linear hashing with partial expansions," *ACM Transactions on Database Systems*, Vol. 7, No. 4, Dec. 1982, pp. 566-587.
16. P. Larson and A. Kajla, "File organization: implementation of a method guaranteeing retrieval in one access," *ACM Computing Practices*, Vol. 27, No. 7, July 1984, pp. 670-677.
17. P. Larson, "Linear hashing with overflow-handling by linear probing," *ACM Transactions on Database Systems*, Vol. 10, No. 1, Mar. 1985, pp. 75-89.
18. P. Larson, "Linear hashing with separators - a dynamic hashing scheme achieving one-access retrieval," *ACM Transactions on Database Systems*, Vol. 13, No. 3, Sept. 1988, pp. 366-388.
19. W. Litwin, "Linear hashing: a new tool for files and tables addressing," in *Proceedings of the 6th International Conference on Very Large Data Bases*, 1980, pp. 212-223.
20. D.B. Lomet, "Bounded index exponential hashing," *ACM Transactions on Database Systems*, Vol. 8, No. 1, Mar. 1983, pp. 136-165.
21. D.B. Lomet, "Partial expansions for file organizations with an index," *ACM Transactions on Database Systems*, Vol. 12, No. 1, Mar. 1987, pp. 65-84.
22. N. Lorentzos and Y. Manolopoulos, "Performance of linear hashing schemes for primary key retrieval," *Information Systems*, Vol. 19, No. 5, 1994, pp. 433-446.
23. V.Y. Lum, P.S.T. Yuen and M. Dodd, "Key-to-address transform techniques: a fundamental performance study on large existing formatted files", *Communications of the ACM*, Vol. 14, No. 4, Apr. 1971, pp. 228-239.
24. G.N. Martin, "Spiral storage: incrementally augmentable hash addressed storage," *University of Warwick, Theory of Computation Report*, No. 27, Coventry, England, March 1979.
25. H. Mendelson, "Analysis of extendible hashing," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 6, Nov. 1982, pp. 611-619.
26. J.K. Mullin, "Tightly controlled linear hashing without separate overflow storage," *BIT*, Vol. 21, No. 4, 1981, pp. 390-400.
27. J.K. Mullin, "Unified dynamic hashing," in *Proceedings of the 10th International Conference on Very Large Data Bases*, 1984, pp. 473-480.
28. J.K. Mullin, "Spiral storage: efficient dynamic hashing with constant performance," *The Computer Journal*, Vol. 28, No. 3, 1985, pp. 330-334.
29. E.J. Otto, "Linearizing the directory growth in order preserving extendible hashing," in *Proceedings of the 4th International Conference on Data Engineering*, 1988, pp. 580-588.
30. K. Ramamohanarao and J.W. Lloyd, "Dynamic hashing schemes," *The Computer Journal*, Vol. 25, No. 4, 1982, pp. 478-485.
31. K. Ramamohanarao, "Recursive linear hashing," *ACM Transactions on Database Systems*, Vol. 9, No. 3, Sept. 1984, pp. 369-391.
32. W.D. Ruchte and A.L. Tharp, "Linear hashing with priority splitting," in *Proceedings of the 3rd International Conference on Data Engineering*, 1987, pp. 2-9.
33. M. Scholl, "New file organizations based on dynamic hashing," *ACM Transactions on Database Systems*, Vol. 6, No. 1, Mar, 1981, pp. 194-211.

34. E. Veklerov, "Analysis of dynamic hashing with deferred splitting," *ACM Transactions on Database Systems*, Vol. 10, No. 1, Mar. 1985, pp. 90-96.



Ye-In Chang (張玉盈) was born in Taipei, Taiwan, in 1964. She received the B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986, and the M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, Ohio, in 1987 and 1991, respectively.

Since 1991, she has been on the faculty of the Department of Applied Mathematics at National Sun Yat-Sen University, Kaohsiung, Taiwan, where she is currently a Professor. Her research interests include database systems, distributed systems, multimedia information systems and computer networks.



Chien-I Lee (李建億) was born in Taipei, Taiwan, R.O.C., 1965. He received the B.S. degree in computer science from Feng Chia University in 1987 and the M.S. degree in applied mathematics from National Sun Yat-Sen University in 1993. He received the Ph. D. degree in computer science from National Chiao Tung University in June 1997 and then joined the Institute of Information Education, National Tainan Teacher College, Tainan, Taiwan. He is currently an assistant professor, and his research interests include object-oriented databases, access methods, multimedia storage servers, video-on-demand, information retrieval and web databases.



Wann-Bay ChangLiaw (張廖萬倍) was born in Taipei, Taiwan, R.O.C., in 1969. He received the B.S. degree in computer and information science from Tung Hai University in 1992, and M.S. degree in applied mathematics from National Sun Yat-Sen University in Taipei, Taiwan, in 1994. His research interests include database systems and computer networks.