Linear Spiral Hashing for Expansible Files

Ye-In Chang, *Member*, *IEEE*, Chien-I. Lee, and Wann-Bay ChangLiaw

Abstract—The goal of dynamic hashing is to design a function and a file structure that allow the address space allocated to the file to be increased and reduced without reorganizing the whole file. In this paper, we propose a new scheme for dynamic hashing in which the growth of a file occurs at a rate of $\frac{n+k}{n}$ per full expansion, where n is the number of pages of the file and k is a given integer constant which is smaller than n, as compared to a rate of two in linear hashing. Like linear hashing, the proposed scheme (called linear 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, linear spiral hashing can maintain a more stable performance through the file expansions and have much better storage utilization than linear hashing. From our performance analysis, linear spiral hashing can achieve nearly 97 percent storage utilization as compared to 78 percent storage utilization by using linear hashing, which is also verified by a simulation study.

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 [3], for example, for a Web-based database [17]. That is, the address space allocated to a file can be increased and reduced without reorganizing the whole file. (Note that the retrieval time based on the hashing approach is O(1) as compared to O(*log n*) in a B-tree approach, where n is the file size.) Over the past decade, many dynamic hashing schemes have been proposed. These dynamic hashing schemes can be divided into two classes: one needs an index, the other one does not need an index. Extendible hashing [1], [6], [19], [22], [26] and dynamic hashing [9], [29], [30] belong to the first class. Linear hashing [4], [5], [10], [11], [12], [14], [15], [16], [18], [20], [23], [27], [28] and spiral storage [2], [7], [8], [21], [24], [25] 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 pages. The first linear hashing scheme was proposed by Litwin [18]. 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. By having two hashing functions active at a time, a file can be expanded without reorganizing the whole records.

- *Y.-I. Chang and W.-B. ChangLiaw are 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 Institute of Information Education, National Tainan Teachers College, Tainan, Taiwan, Republic of China. E-mail: leeci@ipx.ntntc.edu.tw.*

Manuscript received 11 Nov. 1996; revised 9 Nov. 1998.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 107444.

Since in linear hashing, all the records on the split page will be redistributed among this page and a new added page at the end of the file, the storage utilization of this page will suddenly drop to only half of the original storage utilization. Moreover, this phenomenon will cause that the performance in the access time and the storage utilization oscillates after an expansion. To maintain a stable performance through the file expansions, many strategies have been proposed [10], [12], [15], [27]. Among these strategies, linear hashing with partial expansions as first presented by Larson [10], [12] is a generalization of Litwin's linear hashing [18]. 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, the doubling of the file (i.e., a full expansion) is carried out by a series of partial expansions. In [27], they have proposed another way to perform partial expansions, in which data records in all of the buddy pages are redistributed into those old pages and the new added page. Larson also has presented another strategy to maintain a stable performance through the file expansions by changing the expansion sequence [15].

Martin's spiral storage [21], [25] 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 reuse space freed on the left for physical implementation. Unified hashing [24], modified unified hashing [7], and cascading hashing [8] are also proposed based on the similar idea 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 $\frac{n+k}{n}$ per full expansion, where n is the number of pages of the file and k is a given integer constant which is smaller than n, as compared to a rate of two in linear hashing. Like linear

1041-4347/99/$10.00 © 1999 IEEE

hashing, the proposed scheme (called linear 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, linear spiral hashing can maintain a more stable performance through the file expansions and have better storage utilization than linear hashing. The basic idea of linear spiral hashing is similar to Martin's spiral storage. However, while Martin uses an exponential spiral, our scheme uses a linear spiral. Based on an exponential spiral, the expected density of records on the left end of the file will be the highest one and is decreasing from the left to the right of the file; i.e., the load distribution of the pages is not uniform all the time [21]. As compared to the exponential spiral approach, our linear spiral scheme not only reduces the address calculation cost, but also can provide a much uniform load distribution due to the linear function. To reduce the number of disk accesses for overflow records, linear spiral hashing applies *separators* [13], which makes use of a small in-core table to direct search so that the records in the overflow pages can be retrieved in one disk access. Therefore, the retrieval of any record in linear spiral hashing is guaranteed to be in at most two disk accesses. From our performance analysis, linear spiral hashing can achieve nearly 97 percent storage utilization as compared to 78 percent storage utilization by using linear hashing, which is also verified by a simulation study.

The rest of the paper is organized as follows. Section 2 describes the basic idea of linear spiral hashing. Section 3 gives a formal description of linear spiral hashing. Section 4 presents the performance analysis for linear spiral hashing. Section 5 discusses the simulation results of linear spiral hashing, and compares it with linear hashing [18] and linear hashing with partial expansions [10]. Finally, Section 6 contains a conclusion.

2 Basic Idea

In this section, we describe the basic idea of linear spiral hashing. First, we briefly describe Martin's spiral storage [21], [25]. For convenience, we describe the case of $k = 1$. In a dynamic hashing scheme without using an index, the data records are stored in chains of pages linked together. A chain *split* occurs under a certain condition, 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 Key, the logical address can be derived by the following steps:

$$Key -> m(Key) -> X$$

$$-> \text{Logical address(denoted as } Y),$$

where $m(Key)$ is a hash function which distributes the records uniformly on the interval [0, 1). The value of X is derived from the function: $X = \lceil c - m(Key) \rceil + m(Key)$, 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 may be seen graphically in Fig. 1. Note that as

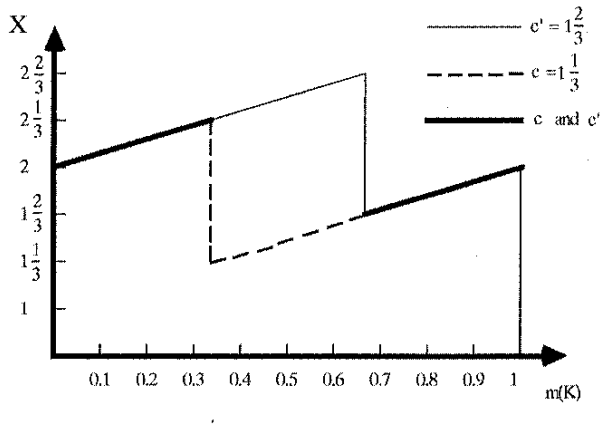


Fig. 1. The X function.

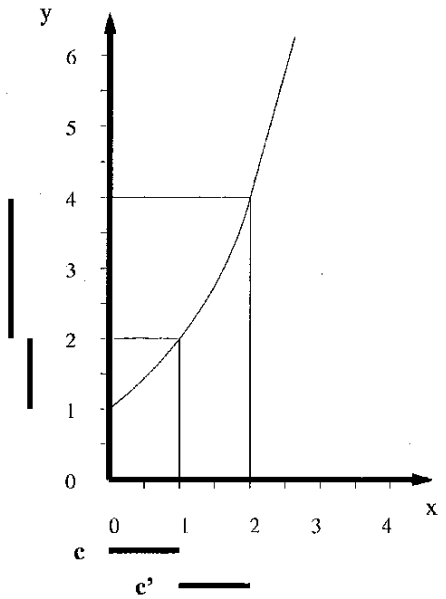


Fig. 2. The growth function $y = f(X) = 2^X$ used in the spiral storage approach.

the value of c changes, the interval of X values stays, but the starting and ending values are c and $(c + 1)$, respectively. A logical address Y is $\lfloor y \rfloor$, where $y = f(X)$. 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 the parameter c increases. The effect of the function is therefore to increase the logical space dynamically.

While in our proposed linear spiral hashing, given $k = 1$ and the number of initial pages $s_0 = 2$, the growth function can be viewed as show in Fig. 3 based on the given growth rate $\frac{n+1}{n}$, where n is the size of the current file. Based on this figure, we can derive the related growth functions $y = f(X)$, as shown in Fig. 4, and their related inverse functions $X = f^{-1}(y)$.

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, split pointer *first* points to page 0. When a file is split, the value of c is readjusted to eliminate the *first* page by the following way: $c\prime = f^{-1}(first + 1)$. All records in the old *first* page (left)

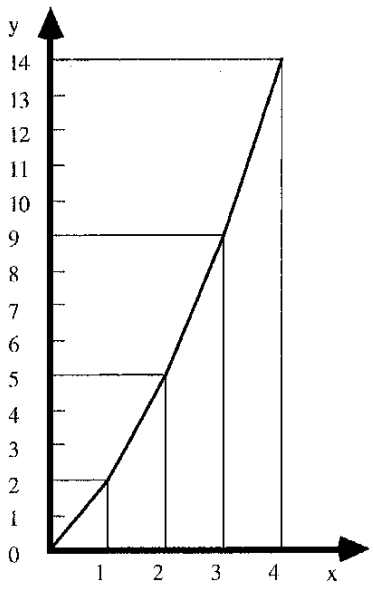


Fig. 3. The growth function $y = f(X)$ when $s_0 = 2$ and $k = 1$.

are remapped into a new larger space on the right. Thus, both file boundaries move. 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. Since many computer systems would have difficulty with a file where both boundaries move, a logical to physical address mapping is employed to reuse space freed on the left. In our approach, we always reuse the freed physical page for the last new added logical page as shown in Table 2, where P is the physical address, n is the current file size and the cross-point of table row n and column P represents the corresponding logical address Y. (Note that the physical address P can be derived from the logical address Y by calling the function *physical* as described in Subsection 3.1 later.)

In linear spiral hashing, a *full expansion* occurs when $c = f^{-1}(first)$ is an integer. (Note that a full expansion occurs when a split occurs at a page next to which is a new added page. In linear spiral hashing, initially, $first = 0$ and $c = f^{-1}(first) = 0$, and the range of X of the current file is [0, 1); therefore, when the range of X of the current file is changed to [1, 2), a full expansion occurs. That is, when $c = f^{-1}(first)$ is an integer, a full expansion occurs.) A *level* (denoted as d) is defined as the number of full expansions happened so far and $d = \lfloor c \rfloor$. On each level d, the pages are split in the order from the small number to the large number of pages. 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. 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 $m(Key) < (c - d)$, i.e., the page where data record with key $= Key$ is stored has been split, then y_{d+1} is applied; otherwise, y_d is used.) Moreover, there are at most $(s_0 + (d + 1)k)$ pages on level d, where s_0 is the initial file size and k is the new added page number per full expansion.

In general, when an insertion causes a splitting, the data records in page *first* will be redistributed to pages *last* and $(last + 1)$, or pages $(last + 1)$ and $(last + 2)$, according to the value of $m(Key)$, where *last* is the logical page number of the last page in the current file is equal to $(\lceil y_d(y_d^{-1}(first) + 1) \rceil - 1)$ (described in Section 4). Since linear spiral hashing only adds k more page after a full expansion and $k < s_0$ (i.e., the growth rate is $\frac{n+k}{n}$, which is smaller than 2), this scheme can maintain a more stable performance through the file expansions and provide better storage utilization than linear hashing.

3 The Algorithms

In this section, given an integer constant k, we give a formal description of the address computation algorithm for linear spiral hashing. We also describe retrieval, insertion, file split, deletion and file contraction algorithms used in linear 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 of the file initially, k be the number of new added page per full expansion ($k < s_0$), and

$y = 2X$	$0 \le X < 1$	$X = \frac{y}{2}$	$0 \le y < 2$
$y = 3X - 1$	$1 \le X < 2$	$X = \frac{y+1}{3}$	$2 \le y < 5$
$y = 4X - 3$	$2 \le X < 3$	$X = \frac{y+3}{4}$	$5 \le y < 9$
$y = 5X - 6$	$3 \le X < 4$	$X = \frac{y+6}{5}$	$9 \le y < 14$
(a)		(b)	

Fig. 4. The (a) growth functions and (b) inverse growth functions, when $s_0 = 2$ and $k = 1$.

TABLE 1
Existing Logical Page Numbers Y When $s_0 = 2$ and $k = 1$

n	c	Y													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	0	0	1												
3	½		1	2	3										
3	1			2	3	4									
4	⅓				3	4	5	6							
4	⅔					4	5	6	7						
4	2						5	6	7	8					
5	¾							6	7	8	9	10			
5	1¾								7	8	9	10	11		
5	1¼									8	9	10	11	12	
5	3										9	10	11	12	13

l be the level number. Then the relationship between growth function y_l for each level l and variables s_0, k, and X can be computed in the following way by observing the relationship shown in Fig. 3:

$y_0 = s_0X;$
$y_1 = (s_0 + k)X - k;$
$y_2 = (s_0 + 2k)X - k(1 + 2);$
$y_3 = (s_0 + 3k)X - k(1 + 2 + 3);$
$y_l = (s_0 + lk)X - k\sum_{i=1}^{l} i;$
$y_l = (s_0 + lk)X - kl\frac{l+1}{2}.$

Let $l = \lceil c - m(Key) \rceil$ and X be $l + m(Key)$, then y_l can be rewritten as follows:

TABLE 2
Placement of Logical Pages When $s_0 = 2$ and $k = 1$

n \ P	0	1	2	3	4
2	0	1			
3	3	1	2		
3	3	4	2		
4	3	4	6	5	
4	7	4	6	5	
4	7	8	6	5	
5	7	8	6	10	9
5	7	8	11	10	9
5	12	8	11	10	9
5	12	13	11	10	9

$$y_l = \frac{1}{2}kl^2 + \left(s_0 - k\left(\frac{1}{2} - m(Key)\right)\right)l + s_0m(Key).$$

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:

$$y_l = y_{l-1} + kl + s_0 - k + km(Key).$$

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

```
function home_address(Key) : integer;
var
    l, Y : integer;
    c, X : real;
begin
    c = y_d^{-1}(first);
    l = ⌈c − m(Key)⌉;
    X = l + m(Key);
    Y = ⌊y_l(X)⌋;
    home_address = physical(Y, l);
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 the reverse growth function y_d^{-1} based on the current value of *first* and d. Then, we let X be $\lceil c - m(Key) \rceil + m(Key)$ and the logical address Y be $\lfloor y_l(X) \rfloor$. (Note that if $m(Key) < (c - d)$, i.e., the page where data record with key $= Key$ is stored has been split, then $l = d + 1$; otherwise, $l = d$.) Finally, we call the following function $physical(Y, l)$ to reuse space freed on the left as explained before:

```
function physical(Y, l) : integer;
var
    low, high : real;
    anc_low, anc_high : integer;
```

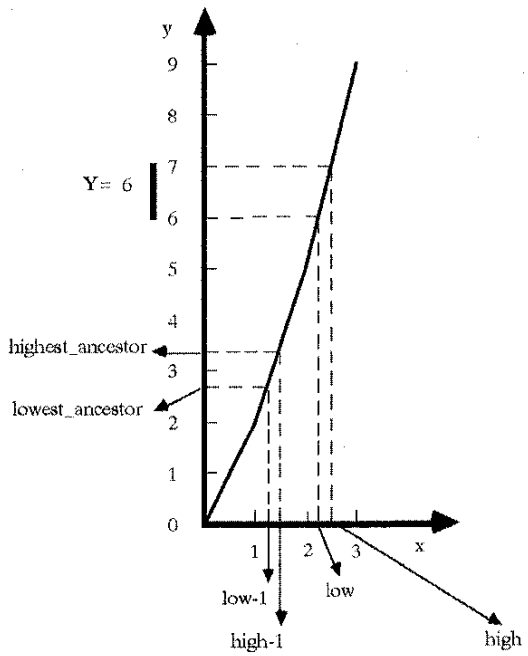


Fig. 5. Physical address mapping.

```
begin
    if Y ≤ s0 − 1 then
        physical = Y
    else
    begin
        low = y_l^{-1}(Y);
        high = y_l^{-1}(Y + 1);
        anc_low = ⌊y_{l-1}(low − 1)⌋;
        anc_high = ⌊y_{l-1}(high − 1)⌋;
        if anc_low < anc_high then;
            physical = physical(anc_low, l − 1)
        else physical = Y − anc_low;
    end;
end;
```

In this function $physical(Y, l)$, given a logical page Y and a current level l, it determines the related physical address [25]. This requires a determination of how the given logical address was instantiated. There are three possible cases:

1. It was one of the original allocation.
2. It was a reuse 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 newly allocated space, its physical address is the number of pages existent just before its creation. If the page was put into 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 could 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 is 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 keyspace range is always one unit long from c to $(c + 1)$. One can map from X back to the page ancestor addresses by the following computation:

$$lowest\_ancestor = y_{l-1}(low - 1), and$$
$$highest\_ancestor = y_{l-1}(high - 1).$$

The actual lower page is $anc\_low = \lfloor lowest\_ancestor \rfloor$, and the higher page is $anc\_high = \lfloor highest\_ancestor \rfloor$. 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 to a new larger space. When a page is totally deallocated, that space can be immediately reuse. 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 find how this recycled page was instantiated. The address is always reduced and the recursion completes.

The above recursive function can be rewritten in the following way without recursion:

```
function physical(Y, l) : integer;
var
    z, anc_low, anc_high : integer;
    low, high : real;
begin
    anc_low = Y;
    repeat
        z = anc_low;
        if z ≤ s0 − 1 then
            physical = z
        else
        begin
            low = y_l^{-1}(z);
            high = y_l^{-1}(z + 1);
            anc_low = ⌊y_{l-1}(low − 1)⌋;
            anc_high = ⌊y_{l-1}(high − 1)⌋;
        end;
    until(anc_low ≥ anc_high);
    physical = z − anc_low;
end;
```

3.2 Overflow Handling and Retrieval

In [15], Larson applied *separators* [13] 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's define a *probe sequence* first [15]. 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 $= Key$, its *probe sequence*, $p(Key) = (p_1(Key), p_2(Key), ..., p_n(Key)), (n \geq 1)$, 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 $= Key$, its *signature sequence*, $s(Key) = (s_1(Key), s_2(Key), ..., s_n(Key))$, is a $q-$bit integer. (Note that $q \geq 1$ and q should be large enough such that signatures of all data records can be in $\{0, (2^q - 2)\}$ [13], [15].) When a data record with key $= Key$ probes page $p_i(Key)$, the *signature* $s_i(Key)$ is used, $1 \leq i \leq n$. Implementation of $p(Key)$ and $s(Key)$ are discussed in detail in [13]. 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 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 [13], [15].)

Since, in [15], 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, linear spiral hashing also applies *separators* but only for overflow pages. To apply *separators* to handle overflow pages in linear 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 $= Key$, let its *probe sequence* be $p_i(Key) = (p_{i1}(Key), p_{i2}(Key), ..., p_{im}(Key)) = (1, 2, ..., m)$, $m \geq 1$. (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 $= Key$, let its *signature sequence* be $s_i(Key) = (s_{i1}(Key), s_{i2}(Key), ..., s_{im}(Key))$. When an overflow record of home page i with key $= Key$ probes page $p_{ij}(Key)$, the *signature* $s_{ij}(Key)$ 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 recomputed to include this new 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 [15]. By using *separators* and the above modification, any data record can be found in at most two disk accesses.

As a file grows, the total size of *separator tables* of all the home pages (which have overflow pages) can be too large to be loaded into main memory at the same time. Moreover, to reduce the number of disk accesses for loading a *separator table* for a certain home page 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 that there is no change for the data records in the home page but a data insertion/deletion has caused data record movements between overflow pages, the related home page still should be written back to the disk before it is removed from main memory. That is, one more disk access is needed in this case, since the contents of the *separator table* has been changed. Therefore, we still can guarantee that the cost of data retrieval is at most two disk accesses. As shown in Fig. 6, 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 and File Split

When a data record is inserted, its home page is searched first. If the size of its home page has exceeded the page size b, then one of its related overflow pages is searched according to its *probe sequences*. In the case that a data record insertion causes relocations of some other records in overflow pages, related *separators* 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 on this home page and its overflow pages have to be extended and recomputed to include this new 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 number of inserted data records exceeds a predetermined 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 $(last + 1)$, or pages $(last + 1)$ and $(last + 2)$, depending on the value of $m(Key)$, 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 update *first* (and d) first and then reinsert those data records which are in the page where the old *first* points to by using the new hashing function y_{d+1}. Next, we show how to derive the current level l given the value of y. Since the growth rate

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

Fig. 6. Function *retrieval*.

of the file is always $\frac{n+k}{n}$, when n is the size of the current file, the following formula shows how y is bounded after $(l-1)$ full expansions:

$$s_0 + (s_0 + k) + \ldots + (s_0 + (l-1)k) \leq y < s_0 + (s_0 + k) + \ldots + (s_0 + lk);$$

$$s_0 + (s_0 + k) + \ldots + (s_0 + (l-1)k) \leq y.$$

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

$$kl^2 + (2s_0 - k)l - 2y \leq 0;$$

$$l \leq \frac{(k - 2s_0) + \sqrt{(2s_0 - k)^2 + 8ky}}{2k};$$

$$l = \left\lfloor \frac{(k - 2s_0) + \sqrt{(2s_0 - k)^2 + 8ky}}{2k} \right\rfloor.$$

The description of 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 reinsertion. That is, we first perform the reinsertion in a buffer. Then, we write one page back to disk from the buffer at a time, instead of reinserting one data record back to disk at a time in the process of reinsertion.)

3.4 Deletion and File Contraction

When a data record is deleted, we immediately try to move another data record 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, 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, by the way, the *separators* must be updated. Second, if the deleted data record is stored in the 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 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 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 $m(Key)$, where $last$ is the logical page number of the last page in the current file. Therefore, when a contraction occurs, we should collect those data records which were stored in page $(first-1)$ before the last file split but now are stored in those pages mentioned above, and move those data records back to page $(first-1)$. The results of the above actions are equal to update $first$ (and d) first and then reinsert all the data records stored in those above pages. (Note that for those data records in pages $last$, $(last-1)$ and $(last-2)$, they may not be moved out from page $(first-1)$ and 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 those pages to determine its original page from which the data record was moved out.) The description of procedure *file_contraction* is shown in Fig. 8.

```
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 = ⌊((k−2s0)+√((2s0−k)²+8k*first))/2k⌋;
        for each record with key = Key in buffer B do
        begin
            i = home_address(Key);
            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 sij(Key) < separatorij↑.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 sij(Key) < separatorij↑.value do
                begin
                    if the page pointed by separatorij↑.pointer is full then
                      move out the record whose key is separatorij↑.value to Buffer B;
                    write the data record with key = Key to the overflow page pointed
                    by separatorij↑.pointer;
                    updated separatorij↑.value if necessary;
                end;
            end;
        end;
end;
```

Fig. 7. Procedure *file_split*.

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 [3], [18]: 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 [18]. 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 [10], [27]. Therefore, we prefer to use the load control as the split control strategy as that in [27], [28].

In this section, we present the performance analysis of linear spiral hashing by using the load control strategy. In this performance analysis model, we assume that the keys for data records are uniformly distributed and independent to each other, and the page size is measured in number of record slots. The hash function which distributes the records uniformly on 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. The overhead for updating *separator* tables for home pages is ignored. We also assume that the number of overflow pages for each home pages is minimum. In other words, if a home page has $t, t \geq 0$, overflow records then there will be $\lceil \frac{t}{w} \rceil$ overflow pages for this home page. 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, *first*, the average retrieval cost and the storage utilization, that is, to analyze these properties of a file as a function of N. The various properties that we are interested are discussed as follows.

```
procedure file_contraction();
    var i, j : integer;
        B : buffer;
begin
    read pages last and (last-1) and (last-2) and their overflow pages into buffer B;
    set pages last and (last-1) and (last-2) and their overflow pages to empty;
    first = first - 1;
    d = ⌊((k-2s0)+√((2s0-k)^2+8k*first))/2k⌋;
    for each record with key = Key in buffer B do
    begin
        i = home_address(Key);
        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 sij(Key) < separatorij⌈.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 sij(Key) < separatorij⌈.value do
            begin
                if the page pointed by separatorij⌈.pointer is full then
                    move out the record whose key is separatorij⌈.value to Buffer B;
                write the data record with key = Key to the overflow page pointed
                by separatorij⌈.pointer;
                updated separatorij⌈.value if necessary;
            end;
        end;
    end;
end;
```

Fig. 8. Procedure *file_contraction*.

The number of splits performed is given by

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

$$\mathrm{ns}(N) = \left\lfloor \frac{N - s_0 L}{L} \right\rfloor, \quad N > (s_0 L).$$

(Note that to reduce the number of splits, we assume that the split control is started when the first s_0 pages are filled with $s_0 * L$ records in the performance analysis.) In linear spiral hashing, the value of *first* is equal to the number of splits, i.e., $first = \mathrm{ns}(N)$. Since, in linear spiral hashing, the growth rate of a file is $\frac{n+k}{n}$, the number of level can be computed by the following formula as derived in Section 3.3, given $Y = first = \mathrm{ns}(N)$:

$$l = \left\lfloor \frac{(k - 2s_0) + \sqrt{(2s_0 - k)^2 + 8k * first}}{2k} \right\rfloor.$$

Since the range of X for the current file is always between c and $(c+1)$, and $y_l(c) = first = \mathrm{ns}(N)$, based on the current value of *first*, we can derive the value of *last*, which the last logical page of the current file, by the following way:

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

$$l' = l + 1;$$

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

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

The probability $Pr(first, i)$ of load distributions for logical pages i, given the value of *first*, are different in linear spiral hashing as shown in Table 3. In general, after a full expansion, i.e., after level l is increased by one (or when $y^{-1}(first)$ is an integer), the probability $Pr(first, i)$ is the same in each page i of the current file, and is equal to $\frac{1}{s_0 + lk}$, $first \le i \le last$; that is, the data records are uniformly distributed in the file. During the process of a full expansion, let *median* be $y_{l'}(l')$, where $l' = l + 1$, then the probability $Pr(first, i)$ can be divided into the following three classes:

$$Pr(first, i) = \frac{1}{s_0 + lk}, \qquad first \le i \le median;$$

$$Pr(first, i) = \frac{1}{s_0 + l'k}, \quad median + 1 \le i \le last - 1;$$

TABLE 3
The Probability of Load Distribution

level (l)	*first*	logical page number (Y)												
		0	1	2	3	4	5	6	7	8	9	10	11	12
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	1/4	1/12						
	4					1/3	1/4	1/4	1/6					
2	5						1/4	1/4	1/4	1/4				
	6							1/4	1/4	1/4	1/5	1/20		
	7								1/4	1/4	1/5	1/5	1/10	
	8									1/4	1/5	1/5	1/5	3/20

$$Pr(first, i) = 1 - \frac{median - first + 1}{s_0 + lk} - \frac{last - median - 1}{s_0 + l'k}, i = last.$$

The pages between *first* and *median* are not split yet in the current level l; therefore, the probability $Pr(first, i)$ of each of those pages is still the same as the probability after l full expansions. The pages between $(median + 1)$ and $(last - 1)$ are newly added during the process of the $(l+1)'$th full expansion; therefore, the probability of each of those pages is the same as the probability after $(l + 1)$ full expansions. For page *last*, $Pr(first, last)$ is equal to 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 $W(t)$ be a function to denote the number of overflow pages of a home page with t data records inserted and be defined as follows:

$$W(t) = \left\lceil \frac{t + SU - b}{w} \right\rceil.$$

Note that, since we store the separator table in the home page, the used space SU for the separator table is

$$\left(\left\lceil (\lceil \frac{t-b}{w} \rceil) \times \frac{1}{sd} \right\rceil \right)$$

data records, where the size of a data record is sd times of the size of a separator entry (including the separator value and the pointer). Let $\mathrm{Bin}(t; N, \mathrm{P})$ denote the binomial distribution, i.e., $\mathrm{Bin}(t; N, \mathrm{P}) = (C_t^N \mathrm{P}^t (1 - \mathrm{P})^{N-t})$. The probability for logical page i $(first \le i \le last)$ containing t data records is $\mathrm{Bin}(t; N, Pr(first, i))$. The expected number of overflow pages for logical page i is obtained as:

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

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 the storage utilization can be obtained as follows:

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

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

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

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

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

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

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

$$RA_i(N) = \sum_{t=0}^{b}(t\text{Bin}(t, N, Pr(first, i))) + \sum_{t=b+1}^{N}((t+(t-b))\text{Bin}(t, N, Pr(first, i))).$$

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 tth ($t \leq N$) 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, $2L$, ..., $ns(N)L(ns(N)L \leq N)$), the total split cost for N inserted data records can be obtained by:

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

Then, we consider the average cost of inserting a data record when there are t data records 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 st 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:

$$AC_i(t) = \frac{2b(1+OP_i(t)) + 2w(\sum_{z=3}^{OP_i(t)+1} z) + (0.5)(2w)(2)}{b + w(OP_i(t) - 1) + 0.5w} = \frac{2b(1+OP_i(t)) + w(OP_i(t)-1)(4+OP_i(t)) + 2w}{b + w(OP_i(t) - 1) + 0.5w}.$$

Note that there are $OP_i(t)$ overflow pages of home page i, and each of overflow pages has w records, expect the last overflow page. We assume that the number of records in the last overflow page is $0.5w$ on the average. Consequently, the total number of records which are stored in home page i and its overflow pages is $(b + w(OP_i(t) - 1) + 0.5w)$. When a record is inserted into the file and the home address of this record is page i, there are $(1 + OP_i(t))$ possible positions in which the record is inserted. Let's consider the case in which a record r is inserted into home page i, which causes another record y in home page i to be moved out to overflow page 1. Then, one record which has been stored in overflow page 1 has to be forced to move out to overflow page 2, and so on. For this case, the total number of pages which have to be updated for inserting a record in home page i is $(1 + OP_i(t))$, i.e., the number of read/write operations is $2(1 + OP_i(t))$. Since there are b records which are stored in home page i, the total number of read/write operations for inserting a record after these b records in home page i is $2b(1 + OP_i(t))$. In the same way, when a record in inserted into overflow page 1, one record in overflow page 1 has to be moved out to overflow page 2, and so on. The number of overflow pages which have to be updated is $OP_i(t)$. Moreover, since the separator table which is stored in home page i has been changed, one more page (i.e., home page i) has to be updated. Therefore, the total number of read/write operations for inserting a record after these w records in overflow page 1 is $2w(1 + OP_i(t))$.

In general, for inserting a record in overflow page $k(1 \leq k < OP_i(t))$, the total number of read/write operations is $2w(1 + (OP_i(t) - k + 1))$. The reason is that the overflow pages that have to be updated are overflow pages $k, (k+1), ..., OP_i(t))$. Therefore, the number of these updated pages is $(OP_i(t) - k + 1)$. Since the separator table in home page i has been changed, home page i also has to be updated. Finally, in the last case (case $(1 + OP_i(t))$), for inserting a record into the last overflow page $OP_i(t)$, the total number of read/write operations for these records $q(q \leq w)$ which are stored in overflow page $OP_i(t)$ is $2q(1+1)$. For simplification, we let q be the average value $\frac{w}{2}$.

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

$$AC(t) = \sum_{i=0}^{st} P(sp, i, st) AC_i(t).$$

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 4a shows the results derived from the above formulas, where $k = 1, s_0 = 2$, $N = 10^6$, $b = 10, 20, 40$, and $80, w = 0.5b$, and $L = 0.8b$, $L = b$, and $L = 1.2b$ in linear spiral hashing. From this table, we observe that the storage utilization can be up to 99 percent, where the cost of successful and unsuccessful search is in terms of the number of disk accesses.

5 Simulation Results

In this section, we show the simulation results of linear spiral hashing, linear hashing [18] and linear hashing with partial expansions [10], under two different split control strategies. In this simulation study [27], we assume that N input data records are uniformly distributed. The environment control variables are the size of a home page (b) and the size of an overflow page (w) and a load control (L) which controls when a split should occur. In this simulation, the storage utilization and the average number of disk accesses for successful and unsuccessful searches are the main performance measures considered. Moreover, overflow pages are handled by *separators* in all these approaches. When the average successful/unsuccessful search cost is concerned, we consider $2N$ search requests, where N searched data records are present in the file and

TABLE 4
Performance: (a) Analysis Results; (b) Simulation Results

Parameters			Analysis Results			
b	w	L	INS	ss	us	uti
10	5	8	16.5	1.874	2.000	0.968
10	5	10	17.0	1.892	2.000	0.993
10	5	12	17.7	1.909	2.000	0.984
20	10	16	10.3	1.825	2.000	0.953
20	10	20	10.9	1.844	2.000	0.954
20	10	24	11.3	1.852	2.000	0.957
40	20	32	6.7	1.755	2.000	0.917
40	20	40	7.1	1.774	2.000	0.955
40	20	48	7.7	1.793	2.000	0.956
80	40	64	4.6	1.650	2.000	0.892
80	40	80	5.1	1.672	2.000	0.895
80	40	96	5.6	1.719	2.000	0.911

(a)

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

Parameters			Simulation Results			
b	w	L	INS	ss	us	uti
10	5	8	16.2	1.884	2.000	0.966
10	5	10	17.1	1.894	2.000	0.992
10	5	12	17.9	1.905	2.000	0.980
20	10	16	10.1	1.829	2.000	0.966
20	10	20	10.8	1.849	2.000	0.952
20	10	24	11.5	1.859	2.000	0.956
40	20	32	6.7	1.759	2.000	0.917
40	20	40	7.2	1.779	2.000	0.961
40	20	48	7.8	1.799	2.000	0.952
80	40	64	4.7	1.652	2.000	0.892
80	40	80	5.1	1.679	2.000	0.892
80	40	96	5.5	1.719	2.000	0.909

(b)

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

the other N searched data records are absent. When the average insertion cost is concerned, we consider the average result of 10 random different insertion sequences.

Table 4b shows the simulation results of linear spiral hashing, where $k = 1, s_0 = 2, N = 10^6, w = 0.5b$, and $L = 0.8b$, $L = b$, and $L = 1.2b$, respectively. Compared with the analysis results shown in Table 4a, the simulation results shown in Table 4b are very close to those shown in Table 4a.

Simulation results of linear spiral 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 Table 5a, Table 5b, Table 5c, and Table 5d, respectively, where $k = 1$, $s_0 = 2$, $N = 10^6$, $w = 0.5b$, and $L = 0.8b$, $L = b$, and $L = 1.2b$. From these tables, we observed that as the sizes of a home page and an overflow page are increased, storage utilization may be decreased in all these four methods. The reason is that the larger the size of a page is, the larger the average unused space in a home page or an overflow page may be, which resulting in a decrease of storage utilization. Linear spiral hashing has the highest storage utilization among these four methods. When $b = 20$, $w = 10$, and $L = 16$, linear spiral hashing can achieve nearly 97 percent storage utilization, as compared to 78 percent storage utilization in linear hashing, and in linear hashing with partial expansions under the same conditions. (Note that linear hashing with partial expansions was proposed to improve the retrieval performance of linear hashing since it can provide more stable performance oscillation than linear hashing during a full expansion. However, based on the same reason, linear hashing with partial expansions will result in higher average insertion cost than linear hashing as stated in [10], [12], [27], [28]. Moreover, linear hashing with partial expansions does not give any large help in improving storage utilization of linear hashing.) 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 linear spiral hashing to that of linear hashing is decreased from:

$$\frac{16.2}{5.5}(\approx 2.9) \text{ to } \frac{2.9}{2.6}(\approx 1.1),$$

when L is increased. The reason is that, when L is increased, the ratio of the number of newly added pages of linear spiral hashing to that of linear hashing is increased under a fixed N. (Note that this ratio is always smaller than 1.) Obviously, since storage utilization and the average insertion cost (and the average retrieval cost) are always a trade-off, linear spiral hashing will need higher average insertion cost and average retrieval cost than the other three methods.

Fig. 9 shows the relationship between storage utilization and the number of inserted data records in linear spiral hashing and linear hashing, where $k = 1, s_0 = 2$, $b = 10, w = 5$, and $L = 8$. From this figure, we observe that linear spiral hashing has more stable and higher storage utilization than linear hashing. That is, the oscillation in performance during a full expansion in linear spiral hashing is smaller than the one in linear hashing. The

TABLE 5
Simulation Results Under the Split Control of the Load Control (L): (a) Linear Spiral Hashing; (b) Linear Hashing; (c) Linear Hashing with Two Partial Expansions; (d) Linear Hashing with Three Partial Expansions

Parameters	Linear Spiral Hashing			
b w L	INS	ss	us	uti
10 5 8	16.2	1.884	2.000	0.966
10 5 10	17.1	1.894	2.000	0.992
10 5 12	17.9	1.905	2.000	0.980
20 10 16	10.1	1.829	2.000	0.966
20 10 20	10.8	1.849	2.000	0.952
20 10 24	11.5	1.859	2.000	0.956
40 20 32	6.7	1.759	2.000	0.917
40 20 40	7.2	1.779	2.000	0.961
40 20 48	7.8	1.799	2.000	0.952
80 40 64	4.7	1.652	1.963	0.892
80 40 80	5.1	1.679	2.000	0.892
80 40 96	5.5	1.719	2.000	0.909

(a)

Parameters	Linear Hashing			
b w L	INS	ss	us	uti
10 5 8	2.7	1.010	1.040	0.788
10 5 10	2.9	1.136	1.434	0.858
10 5 12	3.1	1.243	1.699	0.858
20 10 16	2.5	1.012	1.034	0.781
20 10 20	2.7	1.143	1.423	0.784
20 10 24	2.8	1.233	1.677	0.784
40 20 32	2.3	1.002	1.003	0.781
40 20 40	2.5	1.145	1.407	0.781
40 20 48	2.7	1.234	1.656	0.781
80 40 64	2.2	1.001	1.003	0.757
80 40 80	2.4	1.132	1.376	0.781
80 40 96	2.6	1.222	1.938	0.781

(b)

Parameters	Linear Hashing with Two Par. Exp.			
b w L	INS	ss	us	uti
10 5 8	3.1	1.015	1.047	0.790
10 5 10	3.3	1.144	1.445	0.858
10 5 12	3.5	1.243	1.697	0.858
20 10 16	2.7	1.016	1.046	0.781
20 10 20	2.9	1.153	1.438	0.784
20 10 24	3.1	1.241	1.689	0.784
40 20 32	2.4	1.011	1.031	0.781
40 20 40	2.6	1.154	1.438	0.781
40 20 48	2.8	1.245	1.686	0.781
80 40 64	2.3	1.000	1.000	0.781
80 40 80	2.5	1.157	1.436	0.781
80 40 96	2.7	1.247	1.686	0.781

(c)

Parameters	Linear Hashing with Three Par. Exp.			
b w L	INS	ss	us	uti
10 5 8	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	2.000	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	2.000	0.724

(d)

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

reason is that when a split occurs, linear hashing always redistribute data records of a certain page i into page i and a new added empty page. The property of stable storage utilization in linear spiral hashing has distributed the overhead of insert/split operations uniformly as data record are inserted, while the unstable storage utilization in linear hashing may suddenly cause a large overhead of insert/split operations.

Recall that the growth rate of linear spiral hashing is $\frac{n+k}{n}$ per full expansion, which is not a constant since n is changed during file growth, where n is the current size of the file. To compare the average insertion/retrieval cost in linear hashing and linear spiral hashing when both approaches achieve the same storage utilization, we try to run linear hashing under different choices of L. Table 6 shows that storage utilization in linear hashing can be

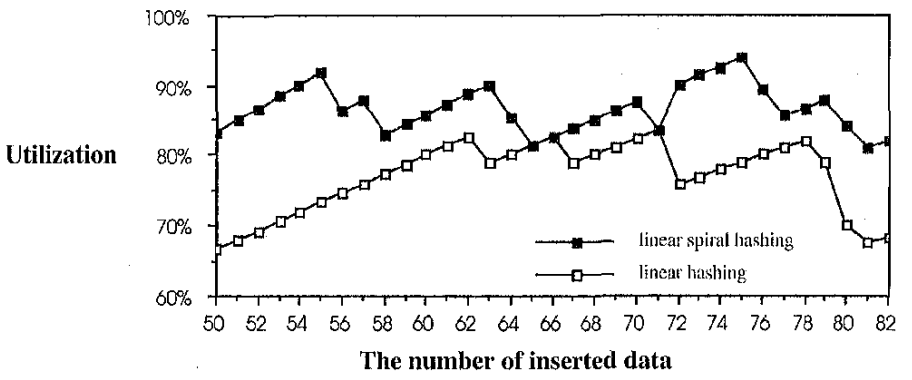


Fig. 9. The relationship between storage utilization and the number of inserted data records.

increased as L is increased, at the cost of increasing the average retrieval cost, where $k = 1, s_0 = 2, b = 20, w = 10$, and $N = 10^6$. From this table, we observe that when both approaches have the same average unsuccessful search cost ($L = 31$ in linear hashing), linear spiral hashing has higher storage utilization but a higher average insertion cost and a higher average successful search cost than linear hashing.

Moreover, when both approaches have the similar average insertion cost ($L = 93$ in linear hashing), linear spiral hashing has higher storage utilization but a higher average successful search cost than linear hashing. When both approaches have the same storage utilization ($L = 106$ in linear hashing), linear spiral hashing can have a lower average insertion cost but a higher average successful search cost than linear hashing. 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 linear spiral hashing. When both approaches have the same average successful search cost ($L = 126$ in linear hashing), linear spiral hashing has a lower average insertion cost but lower storage utilization than linear hashing.

Table 7 shows the simulation results of linear spiral hashing and linear hashing under the split control of the load factor (A), where:

$$k = 1, s_0 = 2, N = 10^6, b = 10, \text{ and } w = 5.$$

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. While in linear spiral hashing, only one new home page is added to the file after a full expansion, instead of n new home pages in linear hashing, which results in higher storage utilization. Therefore, almost a split occurs after a data record is inserted because the storage utilization in linear spiral hashing always exceeds the load factor, which results in a higher average insertion cost than the one in linear hashing. Although the number of splits in linear spiral hashing is larger than that in linear hashing, the number of home pages in linear spiral hashing is less than that in linear hashing which results in a higher average retrieval cost in linear spiral hashing as shown in Table 7. Moreover, when $A > 0.85$, linear hashing cannot retain the storage utilization up to A. The reason is that the higher A is, the higher the ratio of performance oscillation during a full

TABLE 6
The Relationship Between Performance and L in Linear Hashing

Load Control	INS	ss	us	uti
L = 20	2.92	1.153	1.438	0.784
L = 30	4.00	1.347	1.968	0.784
L = 31	4.11	1.359	2.000	0.784
L = 60	7.82	1.669	2.000	0.888
L = 90	10.60	1.779	2.000	0.934
L = 93	10.90	1.789	2.000	0.938
L = 100	11.58	1.799	2.000	0.943
L = 106	12.15	1.819	2.000	0.952
L = 120	13.70	1.839	2.000	0.961
L = 126	14.35	1.849	2.000	0.961
linear spiral L = 20	10.82	1.849	2.000	0.952

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

TABLE 7
Simulation Results Under the Split Control of the Load Factor (A)

Load Factor	Linear Spiral Hashing				Linear Hashing			
A	INS	ss	us	uti	INS	ss	us	uti
0.50	11.28	1.550	2.000	0.890	2.62	1.000	1.000	0.500
0.55	11.31	1.550	2.000	0.892	2.60	1.000	1.000	0.549
0.60	11.35	1.550	2.000	0.896	2.61	1.000	1.000	0.599
0.65	11.42	1.550	1.989	0.901	2.66	1.000	1.000	0.649
0.70	11.58	1.555	1.994	0.901	2.73	1.000	1.000	0.699
0.75	11.76	1.560	2.000	0.913	2.85	1.000	1.000	0.746
0.80	12.09	1.563	1.992	0.921	3.00	1.032	1.093	0.800
0.85	12.78	1.580	2.000	0.956	3.17	1.115	1.337	0.849
0.90	13.53	1.629	1.998	0.892	3.35	1.324	1.904	0.858
0.95	13.97	1.760	2.000	0.947	3.28	1.671	2.000	0.892

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

expansion in linear hashing to the one in linear spiral hashing is.

In linear spiral hashing, under the split control of the load control L and $k = 1$, nL more data records are distributed into $(n+1)$ home pages per full expansion, instead of $2n$ home pages in linear hashing, which results in better storage utilization in linear spiral hashing than the one in linear hashing as has been proved by both analysis and simulation results. However, high storage utilization implies that there may be many overflow pages for each home page, resulting in a large number of disk accesses for data retrieval and insertion operations. Therefore, we look for a compromise between high storage utilization and fast data retrieval. Fig. 10 shows the growth functions with different values of k in linear spiral hashing and the growth function in linear hashing. As k is increased in linear spiral hashing, the number of new added logical pages is increased, which results in a lower average retrieval cost and a lower average insertion cost but lower storage utilization than linear hashing. On the other hand, as k is decreased in linear spiral hashing, the number of new added logical pages is decreased, which results in higher storage utilization but a higher average retrieval cost and a higher average insertion cost than linear hashing. That is, if we care about fast retrieval (and a low average insertion cost) more than high storage utilization, we choose a k with a large value in linear spiral hashing. Therefore, linear spiral hashing provides a flexible choice between these two requirements.

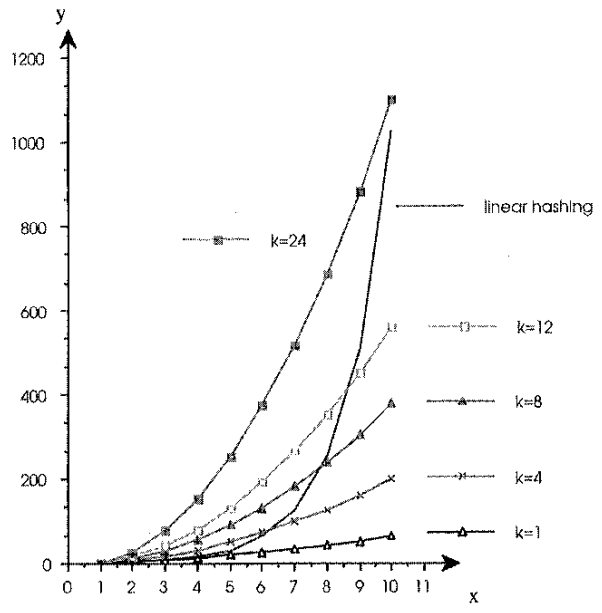


Fig. 10. The growth functions in linear spiral hashing and linear hashing.

6 Conclusion

In this paper, we have proposed a new scheme (called linear spiral hashing) for dynamic hashing in which the growth of a file occurs at a rate of $\frac{n+k}{n}$ per full expansion, where n is the number of pages of the file and k is a given integer constant which is smaller than n, as compared to a rate of two in linear hashing. Because the growth rate of a file is less than two, linear spiral hashing can provide better storage utilization than linear hashing [18]. Moreover, linear spiral hashing can maintain a more stable performance through the file expansions than linear hashing. From our mathematical analysis and simulation study, linear spiral hashing with $k = 1$ can achieve nearly 97 percent storage utilization as compared to 78 percent storage utilization by using linear hashing. As compared to the schemes based on the spiral storage approach [25], linear spiral hashing not only has reduced the cost for address calculation, but also has a

much uniform load distribution due to the linear growth function. Moreover, to compute the logical addresses, no history of split sequence should be traced and only one variable is needed to be recorded in linear spiral hashing (i.e., *first* or c), instead of a table of indexes in [24] or a sequence of split points in [7]. Furthermore, linear spiral hashing has a systematic way in handling file contraction.

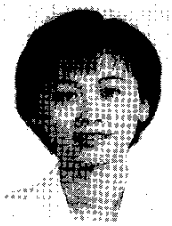
Since high storage utilization and fast data retrieval are always a trade-off in all dynamic hashing schemes, we look for a compromise between high storage utilization and fast data retrieval. Our simulation results show that, if we care about fast retrieval (and a low average insertion cost) more than high storage utilization, we choose a k with a large value in linear spiral hashing. Therefore, linear spiral 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, in part, by the National Science Council of the Republic of China under Grant No. NSC-82-0408-E-110-135.

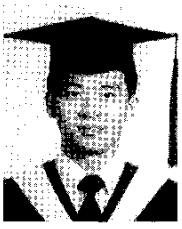
REFERENCES

[1] U. Bechtold and K. Kuspert, "On the Use of Extendible Hashing without Hashing," *Information Processing Letters*, vol. 19, pp. 21-26, 1984.
[2] J. Chu and G.D. Knott, "An Analysis of Spiral Hashing," *Computer J.*, vol. 37, no. 8, pp. 715-719, 1994.
[3] R.J. Enbody and H.C. Du, "Dynamic Hashing Schemes," *ACM Computing Surveys*, vol. 20, no. 2, pp. 85-113, June 1988.
[4] N.I. Hachem and P.B. Berra, "Key-Sequential Access Method for Very Large Files Derived from Linear Hashing," *Proc. Fifth Int'l Conf. Data Eng.*, pp. 305-312, 1989.
[5] 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.
[6] 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.
[7] K. Kawagoe, "Modified Dynamic Hashing," *Proc. Sixth ACM SIGMOD Int'l Conf. Management of Data*, pp. 201-213, 1985.
[8] P. Kjellberg and T.U. Zahle, "Cascade Hashing," *Proc. 10th Int'l Conf. Very Large Data Bases*, pp. 481-492, 1984.
[9] P. Larson, "Dynamic Hashing," *BIT*, vol. 18, pp. 184-201, 1978.
[10] P. Larson, "Linear Hashing with Partial Expansions," *Proc. Sixth Int'l Conf. Very Large Data Bases*, pp. 224-232, 1980.
[11] P. Larson, "A Single-File Version of Linear Hashing with Partial Expansions," *Proc. Eighth Int'l Conf. Very Large Data Bases*, pp. 300-309, 1982.
[12] P. Larson, "Performance Analysis of Linear Hashing with Partial Expansions," *ACM Trans. Database Systems*, vol. 7, no. 4, pp. 566-587, Dec. 1982.
[13] 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.
[14] P. Larson, "Linear Hashing with Overflow-Handling by Linear Probing," *ACM Trans. Database Systems*, vol. 10, no. 1, pp. 75-89, Mar. 1985.
[15] 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.
[16] C.I. Lee, "Design and Analysis of Dynamic Hashing Schemes Without Indexes," masters thesis, Dept. of Applied Math., National Sun Yat-Sen Univ., Republic of China, June 1993.
[17] D. Lester, "Profile of a Web Database," *Database*, vol. 18, no. 6, pp. 46-50, Dec. 1995.
[18] W. Litwin, "Linear Hashing: A New Tool for Files and Tables Addressing," *Proc. Sixth Int'l Conf. Very Large Data Bases*, pp. 212-223, 1980.
[19] D.B. Lomet, "Bounded Index Exponential Hashing," *ACM Trans. Database Systems*, vol. 8, no. 1, pp. 136-165, Mar. 1983.
[20] D.B. Lomet, "Partial Expansions for File Organizations with an Index," *ACM Trans. Database Systems*, vol. 12, no. 1, pp. 65-84, Mar. 1987.
[21] G.N. Martin, "Spiral Storage: Incrementally Augmentable Hash Addressed Storage," *Theory of Computation Report*, no. 27, Univ. of Warwick, Conventry, England, Mar. 1979.
[22] H. Mendelson, "Analysis of Extendible Hashing," *IEEE Trans. Software Eng.*, vol. 8, no. 6, pp. 611-619, Nov. 1982.
[23] J.K. Mullin, "Tightly Controlled Linear Hashing without Separate Overflow Storage," *BIT*, vol. 21, no. 4, pp. 390-400, 1981.
[24] J.K. Mullin, "Unified Dynamic Hashing," *Proc. 10th Int'l Conf. Very Large Data Bases*, pp. 473-480, 1984.
[25] J.K. Mullin, "Spiral Storage: Efficient Dynamic Hashing with Constant Performance," *Computer J.*, vol. 28, no. 3, pp. 330-334, 1985.
[26] E.J. Otto, "Linearizing the Directory Growth in Order Preserving Extendible Hashing," *Proc. Fourth Int'l Conf. Data Eng.*, pp. 580-588, 1988.
[27] K. Ramamohanarao and J.W. Lloyd, "Dynamic Hashing Schemes," *Computer J.*, vol. 25, no. 4, pp. 478-485, 1982.
[28] K. Ramamohanarao, "Recursive Linear Hashing," *ACM Trans. Database Systems*, vol. 9, no. 3, pp. 369-391, Sept. 1984.
[29] M. Scholl, "New File Organizations Based on Dynamic Hashing," *ACM Trans. Database Systems*, vol. 6, no. 1, pp. 194-211, Mar. 1981.
[30] E. Veklerov, "Analysis of Dynamic Hashing with Deferred Splitting," *ACM Trans. Database Systems*, vol. 10, no. 1, pp. 90-96, Mar. 1985.



Ye In Chang received the BS degree in computer science and information engineering from National Taiwan Univerisity, Taipei, Taiwan, in 1986; and the MS and PhD degrees in computer and information science from Ohio State University, Columbus, 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. She is a member of the IEEE and the IEEE Computer Society.

Chien-I Lee received the BS degree in computer science from Feng Chia University in 1987; the MS degree in applied mathematics from National Sun Yat-Sen University in 1993; and the PhD degree in computer science from National Chiao Tung University in June 1997. He then joined the Institute of Information Education at National Tainan Teacher College, Tainan, Taiwan, and is currently an assistant professor. His research interests include object-oriented databases, access methods, multimedia storage servers, video on demand, information retrieval, and web databases.



Wann-Bay ChangLiaw received the BS degree in computer science from Tunghai Unviersity in 1992 and the MS degree in applied mathematics from National Sun Yat-Sen University in 1994. His current research interests include database systems and distributed systems.