JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 9, 495-521 (1993)

DOI: 10.6688/JISE.1993.9.4.1

An Incremental Approach to Dynamic Hashing for Expansible Files[1]

YE-IN CHANG AND CHIEN-I LEE

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

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 an incremental approach to a dynamic hashing scheme in which the growth of a file occurs at a rate of $((n+1)/n)$ per full expansion, where n is the number of pages of the file, as compared to a rate of two in linear hashing. Like linear hashing, the proposed approach requires no index; however, the proposed approach may or may not add one more page, instead of always adding one more page in linear hashing, when a split occurs. Therefore, the proposed approach can have much better storage utilization than can linear hashing. To reduce the number of disk accesses for overflow records, the proposed approach applies *separators*; therefore, the retrieval of any record is guaranteed to be in at most two disk accesses. From our performance analysis, the proposed approach can achieve nearly 95% storage utilization as compared to 78% storage utilization by using linear hashing, which is also verified by a simulation study. Moreover, the proposed approach can be generalized to have the growth of a file at a rate of $((n+k-1)/n)$, where k is an integer larger than 1. As k is increased, the average number of overflow pages per home page is reduced, resulting in a decrease of the average number of disk accesses for data retrieval.

Keywords: Access methods, dynamic storage allocation, file organization, file system management, hashing.

1. INTRODUCTION

Hashing techniques are used in file systems to achieve good performance in access time and address records by using an identifier called a *primary key*, or simply *key* [18]. Hashing provides a direct access mechanism which offers no index storage space requirement and no complex storage management scheme. However, as the size of the file grows, *collisions* may occur (i.e., many keys map to the same address) and may cause an *overflow* (i.e., there is no room in the mapped address), resulting in serious performance degradation of insertion and retrieval operations. Therefore, file reorganization is required in this case. In the

Received May 10, 1993; revised September 13, 1993.
Communicated by Wei-Pang Yang.
[1]This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-82-0408-E-110-135.

process of file reorganization, an address space larger than the current one is chosen, and all the records must be rehashed by using a new hashing function. To overcome this difficulty, dynamic hashing is used.

The goal of dynamic hashing is to design a function and a file structure that can adapt in response to large, unpredictable changes in the number and distribution of keys while maintaining fast retrieval time [2, 26]. That is, the address space allocated to a file can be increased and reduced without reorganizing the whole file. Over the past decade, many dynamic hashing schemes have been proposed. These dynamic hashing schemes can be divided into two classes: one needs an index, the other one does not need an index. Extendible hashing [1, 6, 16, 19, 21] and dynamic hashing [7, 25, 27] belong to the first class. Linear hashing [3, 4, 5, 8, 9, 10, 12, 13, 15, 17, 20, 22, 23, 24] belongs to the second class.

Among these dynamic hashing schemes, linear hashing dispenses with the use of an index at the cost of requiring overflow pages. The first linear hashing scheme was proposed by Litwin [15]. 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 2.1), the number of pages is doubled. By having two hashing functions active at the same time, a file can be expanded without reorganizing all the records. Based on Litwin's linear hashing, there are several variants of linear hashing, which try to improve the performance of the basic algorithm. Those variants can be classified into the following four types:

1. Load Distribution: These strategies try to increase the storage utilization or make uniform the load distribution [8, 10, 13, 22]. Among these strategies, linear hashing with partial expansions as first presented by Larson [8, 10] is a generalization of Litwin's linear hashing [15]. This method splits a number of *buddy* pages together at one time, and the data records in each of those buddy pages are redistributed into the related old page and the new added page (called a partial expansion) to maintain more uniform storage utilization through the file, which improves performance. That is, the doubling of the file (i.e., a full expansion) is carried out by a series of partial expansions. For example, when the number of buddy pages equals 2, the first partial expansion increases the file size to 1.5 times the original file size, and the second partial expansion doubles the file size of the original one. In [22], Ramamohanarao has 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 make uniform storage utilization through the file by changing the expansion sequence [13]. For example, instead of splitting pages from 0 to 5, the splitting sequence can be (5, 2, 4, 1, 3, 0), assuming that there are six pages in the system.
2. Overflow Handling: These schemes handle overflow pages in different ways to reduce the number of disk accesses, for example, with linked lists [15], including overflow records in home pages [20], multiple overflow

chains per home page [9], linear probing [12], separators [13] and recursive linear hashing [23].

3. Physical Implementation: These strategies describe efficient ways to implement linear hashing. That is, they show how to map a logical address to a physical address such that the performance of data retrieval, data insertion and file expansion can be improved, for example, by using quanta of different sizes [15], a fixed-size table for quanta [23], and elastic buckets [4, 17].
4. Ordinal Access: These schemes attempt to capture the hashed order in consecutive storage areas so that the order preserving schemes should result in performance improvement for range queries and sequential processing [3, 4, 5, 24].

Since, in linear hashing, all the records on the overflow page will be redistributed between 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 the performance in access time and storage utilization to oscillate after an expansion. To increase storage utilization and maintain stable storage utilization, in this paper, we propose an incremental approach to dynamic hashing. Two incremental schemes are presented: the first one is called *forward incremental hashing*, and the second one is called *backward incremental hashing* since the first one always splits pages forwards, and the second one always splits pages backwards. Both of them require no index and restrict the growth of a file at a rate of $((n+1)/n)$ per full expansion, where n is the number of pages of the file, as compared to a rate of two in linear hashing. These two incremental hashing schemes may or may not add one more page, instead of always adding one more page in linear hashing, when a split occurs; therefore, they can have better storage utilization than linear hashing. To reduce the number of disk accesses for overflow records, the incremental hashing applies *separators* [11], which make use of a small in-core table to direct the search so that the records in the overflow pages can be retrieved in one disk access. Therefore, the retrieval of any record in the proposed schemes is guaranteed to be in at most two disk accesses.

From our performance analysis, the proposed approach can achieve nearly 95% storage utilization as compared to 78% storage utilization using linear hashing, which result has also been verified by a simulation study. We also observe that the proposed incremental hashing schemes can have even better storage utilization than linear hashing, when the keys are not uniformly distributed. Moreover, the proposed approach can be generalized to set the growth of a file at rate of $((n+k-1)/n)$, where k is an integer larger than 1. As k is increased, the average number of overflow pages per home page is reduced, resulting in a decrease in the average number of disk accesses for data retrieval (while also decreases storage utilization). From our simulation study, when $k=3$ (or 4), the proposed approach can still have better storage utilization than linear hashing while also needing fewer disk accesses for data retrieval than all the other cases of k.

The rest of the paper is organized as follows. Section 2 describes the basic ideas of these two incremental hashing schemes. Section 3 gives formal descriptions

of the proposed schemes. Section 4 presents the performance analysis for the proposed schemes. Section 5 discusses the simulation results of the proposed schemes, and compares them with linear hashing. Section 6 extends the incremental approach to have the growth of a file set at a rate of $((n+k-1)/n)$. Finally, Section 7 contains a conclusion.

2. BASIC IDEAS

In this section, we describe the basic ideas of two incremental schemes: forward incremental hashing and backward incremental hashing.

2.1 Forward Incremental Hashing

In a dynamic hashing scheme without using an index, the data records are stored in chains of pages linked together [26]. A chain *split* occurs under certain conditions, for example, whenever the number of records exceeds the upper bound of a load control L. Given a key, this scheme addresses records by using a series of split functions, h_0, h_1, ..., h_i. In forward incremental hashing, the split functions are defined as follows.

Let each key be mapped into a string of binary bits first, i.e., $H(key) = (b_{q-1}, \ldots, b_1, b_0) = c$. Let $h_0(c) = 0$ be the function to load the file initially. The rest of the split functions, h_1, h_2, ..., h_i, are defined as follows:

for any record with $H(key) = c$:

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

that is, $h_{i+1}(c) = \Sigma_{k=0}^{i} b_k$ and $0 \leq h_{i+1}(c) \leq i+1$.

Let a split pointer *sp* point to the next page to be split and, initially, split pointer *sp* points to page 0. A *full expansion* occurs when a split occurs at a page next to which is a new added page [15]. A *level* is defined as the number of full expansions which have occurred so far. For each level d, h_d or h_{d+1} is used to locate a page depending on whether $h_d(c) \geq sp$ or not, and there are at most $(d+1)$ pages in level d. On each level d, the pages are split in order from page 0 to the maximum number of pages on that level. After all the pages in the current level d have been split, i.e., after a full expansion, the value of level d is increased by 1, and the splitting process starts again from page 0 (i.e., *sp* is reset to 0).

Consider an example in which each home page can contain three records. Let L, the load control, be 3. Let H(key) be the binary number representation of the key. Initially, data records with H(key) = "0001", "0010", "0011", are inserted into page 0 by h_0, as shown in Fig. 1-(a). When a data record with H(key) = "0100" is inserted into home page 0, the number of records has exceeded L; therefore, a split occurs. Since $h_1(0001) = h_1(0011) = b_0 = 1$ and $h_1(0010) =$

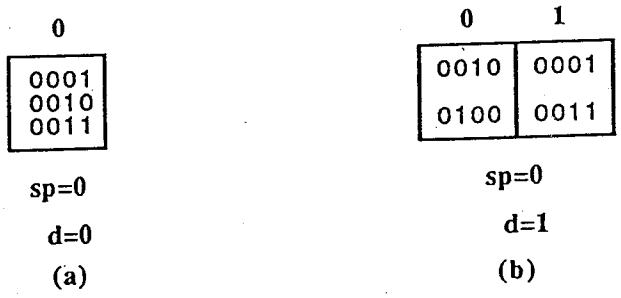


Fig. 1. An example of forward incremental hashing.

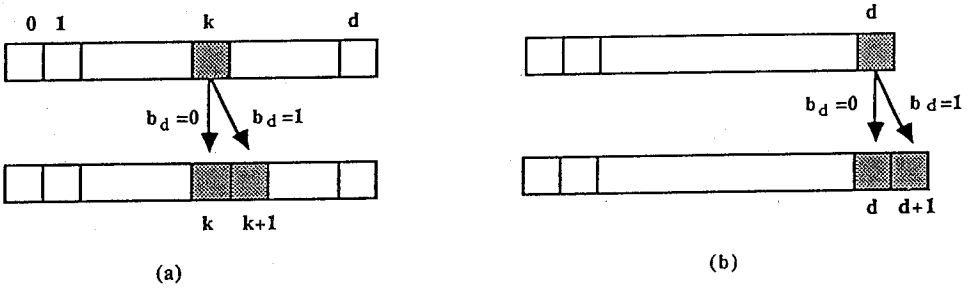


Fig. 2. A splitting operation in forward incremental hashing.

$h_1(0100) = b_0 = 0$, data records with H(key) = "0001" and "0011" are moved to the newly added page 1, and data records with H(key) = "0010" and "0100" still stay in page 0 as shown in Fig. 1 – (b). After the split occurs, *sp* is increased by 1, which exceeds the maximum index of pages in level 0, i.e., a full expansion occurs; therefore, *sp* is reset to page 0, and the value of *d* is increased by 1.

In general, when an insertion causes a splitting and $sp = k$, the data records in page k will be redistributed to page k or page $(k+1)$, according to whether the value of bit b_d is 0 or 1, respectively, i.e., according to the value of $h_{d+1}(c)$ as shown in Fig. 2-(a). When a split occurs and $sp = d$, i.e., *sp* has pointed to the maximum index of pages in level d, a new page $(d+1)$ is added at the end of the file, and the data records in page d are redistributed to page d or page $(d+1)$, according to whether the value of bit b_d is 0 or 1, respectively, as shown in Fig. 2-(b). (Note that forward incremental hashing always splits a page k into page k and page $(k+1)$; i.e., it splits forwards.)

2.2 Backward Incremental Hashing

The above definition of splitting functions has shown how forward incremental hashing adds only one more page per full expansion. However, using these hashing functions, the system may result in an uneven load distribution when the keys of input data records are uniformly distributed, i.e., more data records

Table 1. The variance of the load distribution in forward incremental hashing.

Level	Page number (X)								Mean	Variance
d	0	1	2	3	4	5	6	7		
1	1	1							1	0
2	1	2	1						1.3	0.2
3	1	3	3	1					2	1
4	1	4	6	4	1				3.2	3.8
5	1	5	10	10	5	1			5.3	13.6
6	1	6	15	20	15	6	1		9.1	48
7	1	7	21	35	35	21	7	1	16	173

$$\text{Mean} = \frac{1}{d+1}\sum_{i=0}^{d} Xi \qquad \text{Variance} = \frac{1}{d+1}\sum_{i=0}^{d} (Xi - \text{Mean})^2$$

are distributed in those pages which are near the central part of the file. This case can be explained as follows. Since the number of records stored in page k is equal to the total number of 1's in the binary representation of a key, there are C_k^d records in page k ($0 \le k \le d$) when the keys are uniformly distributed. Therefore, the variance of the load distribution can show how the uneven load distribution occurs in forward incremental hashing, as shown in Table 1, where the value shown in the intersection position of level d and page number X is the number of records stored in that case. From Table 1, we observe that the variance of the load distribution is nearly an exponential function of the level d. To make uniform the load distribution, let's consider backward incremental hashing, which is defined as follows.

For any record with H(key) = c:

$$h_0(c) = 0,$$

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

Consider the example shown in Fig. 3, where the size of a home page is 3, the size of an overflow page is 1, and the load control is 3. Initially, data records with H(key) = "0001", "0010" and "0011" are inserted into home page 0 by using $h_0(c) = 0$, and $sp = 0$ and $d = 0$, as shown in Fig. 3-(a). After the data record with H(key) = "0100" is inserted, the number of inserted records exceeds the load control. Therefore, data records in page 0 are split into page 0 and page 1 by using h_1, as shown in Fig. 3-(b). (Note that at this point, sp in level 0 also has pointed to the maximum index of pages in level 0; therefore, the value of level d should be increased by 1, and sp is reset to 0.)

Figure 3-(c) shows the system state after two more records with H(key) = "0101" and "0110" inserted into page 1 and page 0 by using $h_1(0101) = 1$ and $h_1(0110) = 0$, respectively. After a data record with H(key) = "0111" is inserted, the number of inserted records exceeds the load control again; therefore, data

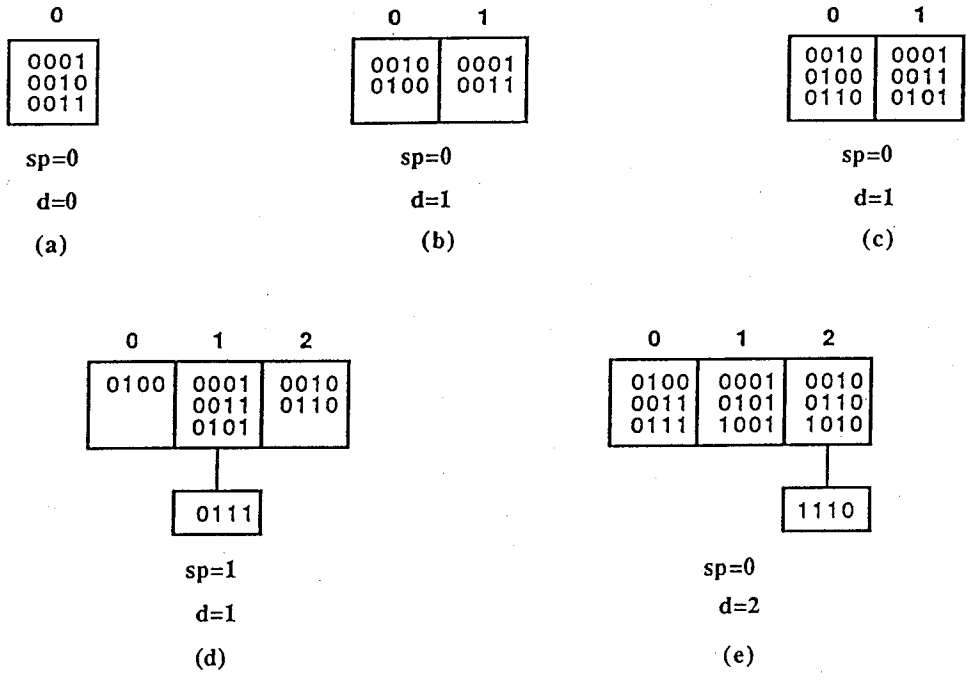


Fig. 3. An example of backward incremental hashing.

records in page 0 have to be redistributed to page 0 and page 2 by using the split function $h_{d+1}=h_2$, and then *sp* is increased by 1 as shown in Figure 3-(d). After three more records with H(key)= "1001", "1010" and "1110" are inserted, a split occurs again. Since $sp=1$, data records in page 1 have to be redistributed to page 1 and page 0 by using the split function $h_{d+1}=h_2$. After *sp* is increased by 1, *sp* is greater than *d* (i.e., the maximum index of pages in current level *d*); therefore, *sp* is reset to 0, and *d* is increased by 1, as shown in Fig. 3-(e).

In general, backward incremental hashing always splits a page k ($k>0$) into page k and page $(k-1)$; i.e., it splits backwards. Note that when k is equal to 0, it splits page 0 into page 0 and a new added page $(d+1)$, where d is the maximum index of pages in level d.

The load distribution for backward incremental hashing can also be described in terms of the variance, as shown in Table 2. (Note that when we let X_i^d be the number of records in page i of level d when the keys are uniformly distributed and $X_0^2=2$, $X_1^2=1$, $X_2^2=1$, the value of X_i^d is $(X_i^{d-1}+X_{i+1}^{d-1})$ when $d>2$ and $0\leq i<(d-1)$, and the value of X_{d-1}^d is X_{d-1}^{d-1} and the value of X_d^d is X_0^{d-1} when $d>2$.) Compared with forward incremental hashing, backward incremental hashing has much smaller variance of load distribution than does forward incremental hashing, which shows that backward incremental hashing has more even load distribution than forward incremental hashing when the keys are uniformly distriubted.

Table 2. The variance of the load distribution in backward incremental hashing.

Level	Page number (X)								Mean	Variance
d	0	1	2	3	4	5	6	7		
1	1	1							1	0
2	2	1	1						1.3	0.2
3	3	2	1	2					2	0.5
4	5	3	3	2	3				3.2	0.9
5	8	6	5	5	3	5			5.3	2.2
6	14	11	10	8	8	5	8		9.1	6.9
7	25	21	18	16	13	13	8	14	16	24.5

$$\text{Mean} = \frac{1}{d+1}\sum_{i=0}^{d} Xi \qquad \text{Variance} = \frac{1}{d+1}\sum_{i=0}^{d} (Xi - \text{Mean})^2$$

3. THE ALGORITHMS

In this section, we give a formal description of address computation algorithms for forward incremental hashing and backward incremental hashing, respectively. We also describe retrieval, insertion and file split algorithms used in both schemes. In these algorithms, the following variables are used globally: (1) b: the size of a home page in terms of the number of records; (2) w: the size of an overflow page in terms of the number of records; (3) *sp*: the split pointer 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 for Forward Incremental Hashing

Let function H(key) map a key into random binary bit patterns of length q, for q sufficiently large. Let function $b_i(c)$ return the value of the *i*th bit of the binary patterns, which is denoted by c (= H(key)). To compute the final home page number after *d* full expansions, the function home_address is defined as follows:

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

In this function, initially, all the data records are mapped into page 0 by $h_0(c)=0$ and, hence, *address*$=0$. Then, the for-loop statement traces the home page number (denoted as *address*) through d full expansions. Because the result of the above for-loop statement is equal to the summation of the binary bit patterns from b_0 to b_{d-1}, we actually need not trace the loop for d times, and we only take a sum operation. For the unfinished $(d+1)$th full expansion, a page may have been split or not. Depending on whether or not *address*$<sp$, the final home page number is determined. If *address*$\geq sp$, the page determined by the key after d full expansions has not yet been split; therefore, the page number is the final home page number. Otherwise, *address* must be added with the value of bit b_d to determine the final home page number.

3.2 Address Computation for Backward Incremental Hashing

Based on the same definitions for function H(key) and $b_i(c)$, the home page computation function for backward incremental hashing is given as follows:

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

In this function, initially, all the data records are also mapped into page 0 by $h_0(c)=0$ and, hence, *address*$=0$. Then, the for-loop statement traces the page number through d full expansions according to the hash function defined in Section 2.2; i.e., it traces the 'history' of the record as the file goes through d full expansions. The *address* after the for-loop statement represents the home page number after the dth full expansion. As in forward incremental hashing, we should compare *address* with *sp* to determine the final home page number. In this address computation algorithm, the for-loop statement needs $O(d)$ to determine the home page number after d full expansions. To speed up this computation, we have given another address computation algorithm, which needs $O(\log d)$ in the worst case and may need only $O(1)$ in the best case [14].

3.3 Overflow Handling and Retrieval

In [13], Larson applied *separators* [11] for home pages to linear hashing to guarantee that any data record can be retrieved in one disk access, where overflow pages are eliminated and overflow recores are distributed among the home pages. This method 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. 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*, $p(K) = (p_1(K), p_2(K), \ldots, p_n(K))$, $(n \geq 1)$, defines the order in which the pages will be checked when inserting or retrieving the records. For each data record with key = K, its *signature sequence*, $s(K) = (s_1(K), s_2(K), \ldots, s_n(K))$, is a k-bit integer. When a data record with key = K probes page $p_i(K)$, the *signature* $s_i(K)$ is used, and $1 \leq i \leq n$. Implementation of p(K) and s(K) has been discussed in detail in [11]. 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. Since in [13], 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, incremental hashing also applies *separators* but only for overflow pages.

To apply *separators* to handle overflow pages in both of the proposed incremental hashing schemes, 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 those m pages has a capacity of w records. For each overflow record of home page i with key = K, let its *probe sequence* be $p_i(K) = (p_{i1}(K), p_{i2}(K), \ldots, p_{im}(K)) = (1, 2, \ldots, m)$, $(m \geq 1)$. (Note that to increase storage utilization, we probe overflow page j until overflow page $(j-1)$ is full when a data record is inserted.) For each overflow record of home page i with key = K, let its *signature sequence* be $s_i(K) = (s_{i1}(K), s_{i2}(K), \ldots, s_{im}(K))$. When an overflow record of home page i with key = K probes page $p_{ij}(K)$, the *signature* $s_{ij}(K)$ is used, and $1 \leq j \leq m$. By using *separators* and the above modification, any data record can be found in at most two disk accesses. (Note that a *separator* table has two entries: one is a *separator* value, and the other one is a pointer to an overflow page. Moreover, to reduce the overhead used to store all the separator tables in the main memory at the same time, in physical implementation, we can reserve some space in each home page for the related separator table. When a home page is read in, the related separator is stored into main memory at the same time.) The following function current_address(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 jth overflow page of home page i:

```
function current_address(key) : pointer;
  var i : integer;              /* home page of the record */
      j : integer;              /* an index */
      m : integer;
          /* the number of overflow pages owned by home page i */
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 j = 1 to m do
            begin
                  if s_ij(K) < 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;
```

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 a *separator*. 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.

3.4 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 tried according to its *probe sequences*. In the case where insertion of the data record causes relocations of some other records in the overflow page, related *separators* may also have to be updated [11]. The informal description of procedure insert(key) is given as follows:

```
procedure insert(key);
  var pageno : integer;              /* home page of the record */
```

```
begin
    pageno = home__address(key);
    if (the number of records on page pageno < b) then
        insert the record into page pageno;
    else
    begin
        if the current overflow pages are full then
            append a new page at the end;
        insert the record into one of the overflow
            pages of page pageno by using separators;
    end;
end;
```

In the main program used to implement incremental hashing, we use a global counter to record the number of records newly inserted. That is, the split control (by using the load control L) is not implemented inside the procedure insert(key) but in the main program. When a split should occur, which can be detected by testing the value of the global counter, the main program will call the following procedure file__split. In this procedure, data records in page *sp* (including its overflow pages) have to be redistributed to page *sp* or $(sp+1)$ in forward incremental hashing (or page *sp* or page $(sp-1)$ in backward incremental hashing), depending on whether the value of b_d is 0 or 1. If $sp=d$ when a split occurs, data records in page *d* are redistributed to page *d* or page $(d+1)$ (or page *d* or page $(d-1)$ in backward incremental hashing) by using h_{d+1}. When a full expansion occurs, *d* is increased by 1, and *sp* is reset to 0. The results of the above actions are equal to updating *sp* (and *d*) first and then re-inserting (i.e., by calling procedure insert) those data records which are in the page where the old *sp* points by using the new hashing function h_{d+1}. The informal description of procedure file__split is given as follows:

```
procedure file__split();
  var B : buffer;
begin
    read home page sp and its overflow pages into buffer B;
    set home page sp and its overflow pages to empty;
    sp = sp+1;
    if sp>d then
    begin
        sp = 0;
        d = d+1;
    end;
    for all the records in buffer B do insert(key);
end;
```

4. PERFORMANCE ANALYSIS

In this section, we present the performance analysis of both incremental hashing schemes under the split control of the load control L. In this performance

analysis model, we assume that the keys for data records are distributed uniformly and independently to each other, and that the page size is measured in terms of the number of record slots. The size of a home page is denoted by b, and the size of an overflow page is denoted by w. The overhead for updating *separator* tables for home pages is ignored. We also assume that the number of overflow pages for each home page is a minimum. In other words, if a home page has k, $k \geq 0$, overflow records, then there will be $\lceil k/\mathrm{w} \rceil$ 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, *sp*, the average retrieval cost and the storage utilization, that is, 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

$$\begin{aligned} \mathrm{ns(N)} &= 0, && \text{for } 0 \leq \mathrm{N} \leq (s_0 * \mathrm{L}) \text{ or} \\ \mathrm{ns(N)} &= \lceil (\mathrm{N} - s_0 * \mathrm{L})/\mathrm{L} \rceil, && \text{for } \mathrm{N} > (s_0 * \mathrm{L}). \end{aligned}$$

(Note that to reduce the number of splits, we assume that the first split is delayed until the first s_0 pages are filled with $s_0 * \mathrm{L}$ records in this performance analysis.) Since, in incremental hashing, the growth rate of a file is $((n+1)/n)$, the number of home pages expanded (denoted by m) is given by

$$s_0 + (s_0 + 1) + \ldots, + (s_0 + (m-1)) \leq \mathrm{ns(N)} < s_0 + (s_0 + 1) + \ldots, + (s_0 + m).$$

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

The load distributions (i.e., the number of records distributed into a certain page) for home pages are different in these two proposed incremental hashing schemes as mentioned in Section 2. Let $P(sp, i, s)$ be the probability that a data record will be hashed into home page i after s full expansions when the split pointer points to page *sp*. In forward incremental hashing, the probability $P(sp, i, s)$ is $(C_i^s / 2^s)$ when *sp* is reset to 0 since, when *sp* is reset to 0, there are already 2^s data records inserted. Moreover, at this time, the number of records stored in page i is equal to the total number of 1's in the binary representation of a key, i.e., there are C_i^s records in page i $(0 \leq i \leq s)$, when the keys are uniformly distributed. Therefore, when $sp = 0$, among those 2^s data records, there are C_i^s data records in page i as shown in Table 1. During the $(s+1)$th full expansion, the probability $P(sp, i, s)$ needs to be modified to $((C_{i-1}^s + C_i^s)/(2^s + \sum_{k=0}^{sp-1} C_k^s))$, when $0 < sp \leq s$ and $i \leq sp$. (Note that, since forward incremental hashing always split forwards, the load distribution during the $(s+1)$th full expansion for home page i

should be the one for home page i plus the one for home page $(i-1)$ after s full expansions.) When $0<sp\leq s$ and $i>sp$, the probability $P(sp, i, s)$ is $(C_i^s/(2^s+\sum_{k=0}^{sp-1} C_k^s))$, where the home page i has not been split yet. For example, for the load distribution shown in Table 1, if we let s be 3 and sp be 2, then the probability $P(2, 0, 3)$ is $((C_{-1}^3+C_0^3)/(2^3+C_0^3+C_1^3))$ $(=(1/12)$, for $C_{-1}^3=0)$, the probability $P(2, 1, 3)$ is $((C_0^3+C_1^3)/(2^3+C_0^3+C_1^3))$ $(=(4/12))$, the probability $P(2, 2, 3)$ is $((C_1^3+C_2^3)/(2^3+C_0^3+C_1^3))$ $(=(6/12))$ and the probability $P(2, 3, 3)$ is $((C_3^3)/(2^3+C_0^3+C_1^3))$ $(=(1/12))$.

For similar reasons in backward incremental hashing, after s full expansions and $sp=0$, the probability $P(0, i, s)$ for home page i $(0\leq i<s)$ is $((P(0, i, s-1)+P(0, i+1, s-1))/2)$, and the probability $P(0, s, s)$ for home page s is $(P(0, 0, s-1)/2)$. During the $(s+1)$th full expansion, after a split occurs in home page 0 (i.e., $sp=1$), and all the data records of home page 0 have been redistributed to home page 0 and a new added home page (i.e., page $(s+1)$), the probability $P(1, i, s)$ $(0\leq i<s)$ is $(P(0, i, s)/(1+P(0, 0, s)))$ and the probability $P(1, s+1, s)$ for the new added home page $(s+1)$ is $(P(0, 0, s)/(1+P(0, 0, s)))$. Moreover, when $1<sp\leq s$, the probability $P(sp, i, s)$ of the page left of $(sp-1)$ (i.e., $0\leq i<(sp-1)$) is $((P(0, i, s)+P(0, i+1, s))/(1+\sum_{k=0}^{sp-1} P(0, k, s)))$, while the probability $P(sp, i, s)$ of the page right of $(sp-1)$, including page $(sp-1)$, (i.e., $(sp-1)\leq i\leq s$) is $(P(0, i, s)/(1+\sum_{k=0}^{sp-1} P(0, k, s)))$, and the probability $P(sp, s+1, s)$ for the new added home page $(s+1)$ is $(P(0, 0, s)/(1+\sum_{k=0}^{sp-1} P(0, k, s)))$.

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

$$W(\mathrm{t}) = 0, \quad \text{for } 0 \leq t \leq \mathrm{b} \quad \text{or}$$
$$W(\mathrm{t}) = \mathrm{j}, \quad \text{for } (\mathrm{b}+(\mathrm{j}-1)*\mathrm{w}+1)<t\leq(\mathrm{b}+\mathrm{j}*\mathrm{w}).$$

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

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

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

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

and the storage utilization can be obtained as follows:

$$UTI(\mathrm{N}) = \mathrm{N}/((n+1)*(\mathrm{b}+\mathrm{w}*OP(\mathrm{N}))).$$

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

$$US_i = 1, \quad \text{for } OP_i = 0 \quad \text{or}$$
$$US_i = 2, \quad \text{for } OP_i > 0.$$

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

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

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

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

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

$$SS(\mathrm{N}) = (\Sigma_{i=0}^{n} RA_i(\mathrm{N}))/\mathrm{N}.$$

Table 3 shows the results derived from the above formulas, where $s_0 = 1$, $\mathrm{N} = 1000$, $\mathrm{b} = 10$, 20, 40, and 80, $\mathrm{w} = 0.5 * \mathrm{b}$ and $\mathrm{L} = 0.8 * \mathrm{b}$, $\mathrm{L} = \mathrm{b}$ and $\mathrm{L} = 1.2 * \mathrm{b}$

Table 3. Analysis results.

Parameters			Forward			Backward		
b	w	L	ss	us	uti	ss	us	uti
10	5	8	1.893	2.0	0.931	1.830	2.0	0.967
10	5	10	1.908	2.0	0.939	1.850	2.0	0.970
10	5	12	1.907	2.0	0.949	1.860	2.0	0.972
20	10	16	1.848	2.0	0.909	1.760	2.0	0.948
20	10	20	1.856	2.0	0.922	1.780	2.0	0.952
20	10	24	1.861	2.0	0.936	1.800	2.0	0.956
40	20	32	1.756	2.0	0.889	1.641	2.0	0.921
40	20	40	1.783	2.0	0.901	1.680	2.0	0.929
40	20	48	1.753	1.969	0.946	1.720	2.0	0.937
80	40	64	1.619	2.0	0.852	1.520	2.0	0.889
80	40	80	1.685	2.0	0.874	1.522	2.0	0.887
80	40	96	1.642	2.0	0.908	1.600	2.0	0.914

b: the size of a home page
w: the size of an overflow page
L: load control
ss: successful search
us: unsuccessful search
uti: storage utilization

in forward incremental hashing and backward incremental hashing, respectively. From this table, we observe that the storage utilization can be up to nearly 95% in forward incremental hashing and 97% in backward incremental hashing.

5. SIMULATION RESULTS

In this section, we show the simulation results of both incremental hashing schemes by using the load control strategy, and we compare them with linear hashing [15]. Moreover, we compare them with the results of performance analysis as presented in Section 4.

In this simulation study, 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. Since the overflow pages are handled by *separators*, at most two disk accesses are required for a successful or an unsuccessful search.

Table 4 shows the simulation results of forward incremental hashing and backward incremental hashing, where $N=1000$, $w=0.5*b$ and $L=0.8*b$, $L=b$ and $L=1.2*b$, respectively. Compared with the analysis results shown in Table 3, the simulation results shown in Table 4 are very close to those shown in Table 3.

Figures 4-(a), (b), (c) show the relationship between the storage utilization and b (i.e., the size of a home page) in forward incremental hashing, backward incremental hashing and linear hashing, where $N=1000$, $W=0.5*b$, and $L=0.8*b$,

Table 4. Simulation results.

Parameters			Forward			Backward		
b	w	L	ss	us	uti	ss	us	uti
10	5	8	1.914	1.958	0.921	1.890	1.847	0.917
10	5	10	1.914	1.958	0.939	1.890	1.847	0.935
10	5	12	1.914	1.958	0.948	1.890	1.847	0.943
20	10	16	1.844	1.989	0.917	1.780	1.847	0.925
20	10	20	1.855	1.969	0.917	1.780	2.0	0.943
20	10	24	1.857	1.975	0.934	1.800	2.0	0.952
40	20	32	1.757	1.955	0.893	1.640	2.0	0.926
40	20	40	1.777	1.976	0.901	1.680	2.0	0.926
40	20	48	1.784	1.984	0.943	1.720	2.0	0.944
80	40	64	1.617	1.938	0.893	1.519	2.0	0.926
80	40	80	1.666	1.906	0.893	1.520	2.0	0.893
80	40	96	1.635	1.874	0.893	1.600	2.0	0.926

b: the size of a home page
w: the size of an overflow page
L: load control
ss: successful search
us: unsuccessful search
uti: storage utilization

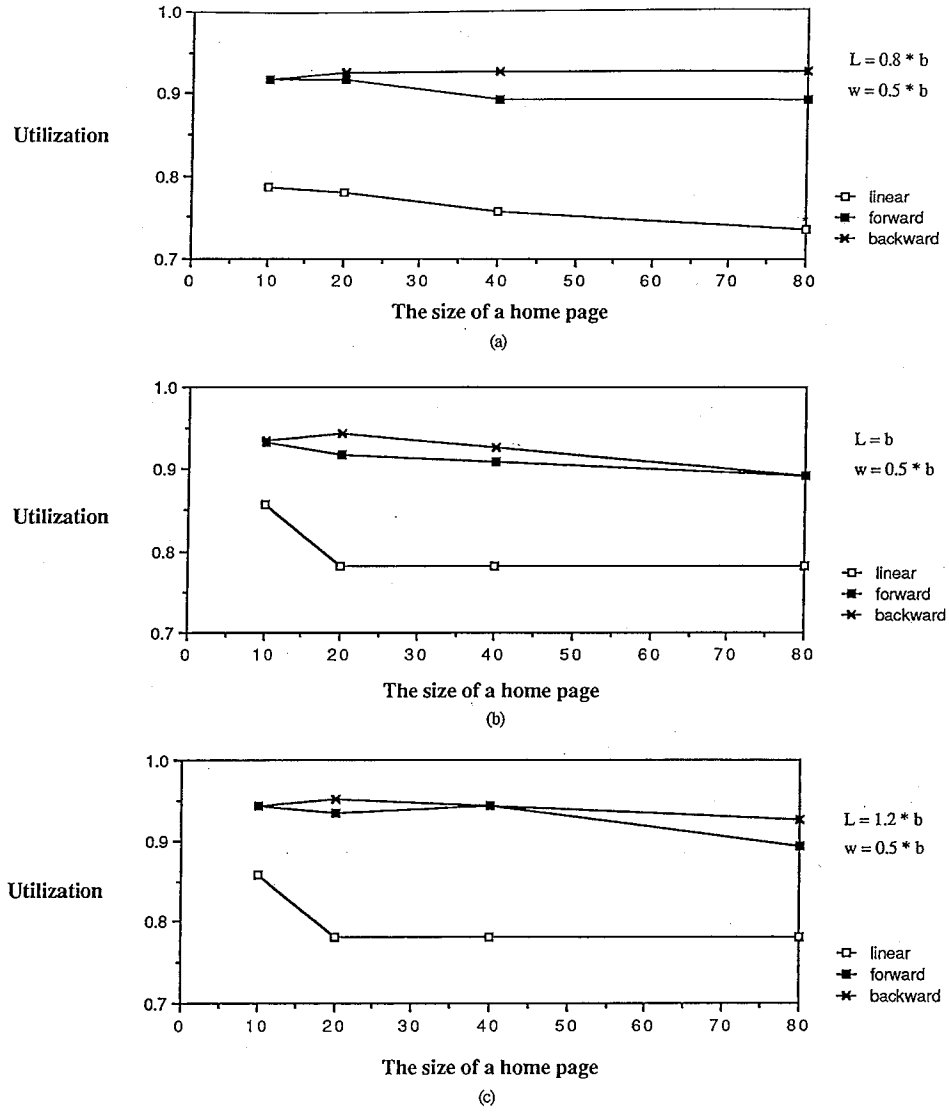


Fig. 4. The relationship between the storage utilization and b: (a) $L=0.8*b$; (b) $L=b$; (c) $L=1.2*b$.

b, and $1.2*b$, respectively. In all of these three cases, whether $L<b$, $L=b$ or $L > b$, both versions of incremental hashing have higher storage utilization than does linear hashing. When $b=40$, $w=20$, and $L=48$, as shown in Figure 4-(c), forward incremental hashing can achieve 94% storage utilization, as compared to 78% storage utilization for linear hashing under the same conditions. Moreover, backward incremental hashing may achieve higher storage utilization than forward incremental hashing. This is because that the latter would let data records be

distributed in those pages which are near the central part of the file, as explained in Section 2, resulting in more overflow pages.

Figures 5-(a), (b) show the relationship between the storage utilization and the size of an overflow page with three different L in these two incremental hashing schemes, where b = 10. From these two figures, we observe that in both schemes, the larger the load control is, the higher the storage utilization is, when the size of an overflow page is fixed. This is because the larger the load control is, the lower is the frequency of occurrence of a split, which results in a decrease in the number of needed home pages (i.e., an increase in the storage utilization). Moreover, as the size of an overflow page is increased, the storage utilization is decreased. This is because the larger the size of an overflow page is, the more empty slots the overflow pages have, which results in a decrease in storage utilization.

Figures 6-(a), (b) show the relationship between the average insertion cost in terms of the number of disk accesses and the size of a home page (b) with

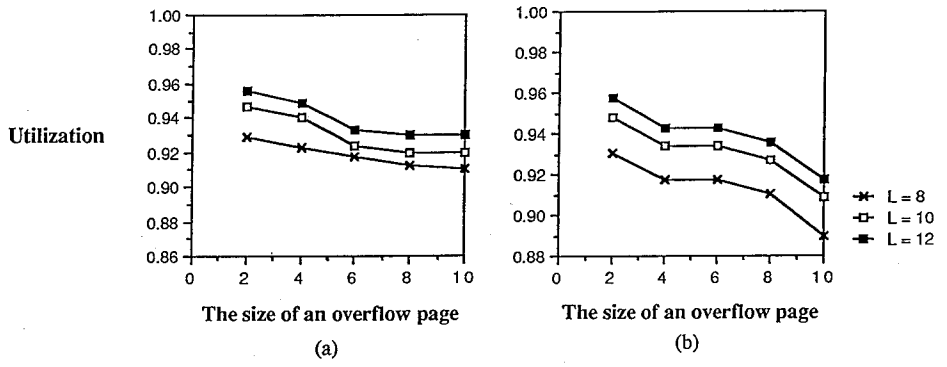


Fig. 5. The relationship between the storage utilization and the size of an overflow page: (a) forward incremental hashing, (b) backward incremental hashing.

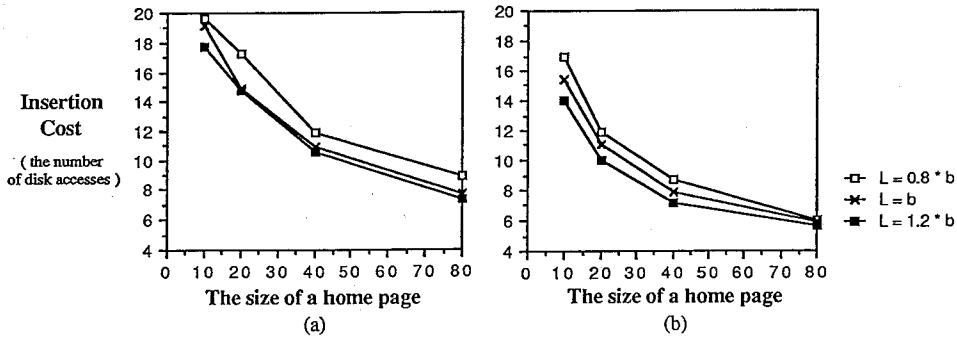


Fig. 6. The relationship between the average insertion cost and the size of a home page: (a) forward incremental hashing, (b) backward incremental hashing.

different L, where the size of an overflow page is $0.5*b$ and $N=1000$ in both schemes. From these two figures, we observe that as the value of b is increased, the average insertion cost is decreased. This is because that the larger the value of b is, the more data records can be put in a home page, instead of in an overflow page, which results in a decrease in the insertion cost. (Note that the insertion cost includes the number of disk accesses to add a data record into a certain page and the number of disk accesses to relocate data records when this insertion causes a split.) Obviously, since storage utilization and insertion cost are always a trade-off, both proposed schemes need a higher insertion cost than does linear hashing. However, in the next section, we will extend the proposed schemes such that a lower insertion cost than that for linear hashing is achievable at the cost of a decrease in storage utilization.

The above simulation results are based on the assumption that the input data records are uniformly distributed and show how incremental hashing is better than linear hashing. Now, let's examine some more interesting results when the input data records are not uniformly distributed. Consider a special case in which almost all of the data records are hashed into the same home page. Assume that k splits occur in linear hashing and that the file contains one page initially; in this case, there are k more pages are added. Under the same number of splits, there are s more pages added in incremental hashing, where $(1+2+\ldots+s)\leq k<(1+2+\ldots+(s+1))$, as explained in Section 4. Therefore, $(((s+1)*s)/2)\leq k$ and $s=\lfloor(\sqrt{8*k+1}-1)/2\rfloor$, i.e., after k splits occur, and about $\lfloor(\sqrt{8*k+1}-1)/2\rfloor$ pages are added in incremental hashing as compared to k pages in linear hashing. When $L=b$ and $w=1$, the storage utilization is $(((k+1)*b)/((\lfloor(\sqrt{8*k+1}-1)/2\rfloor*b)+(k*b)))$ in incremental hashing as compared to $(((k+1)*b)/((k+1)*b+k*b))$ in linear hashing, where there are $(k*b)$ overflow records. As k is increased, the storage utilization approaches 1 in incremental hashing while it is about (1/2) in linear hashing.

Figure 7 shows a comparison of the storage utilization between these two incremental hashing schemes and linear hashing when the keys of input data records are not uniformly distributed, where $N=500$, $b=10$, $w=4$, $L=10$ and the keys of data records are multiplied by 2^1, 2^2, ..., 2^9 and 2^{10} to simulate the case in which

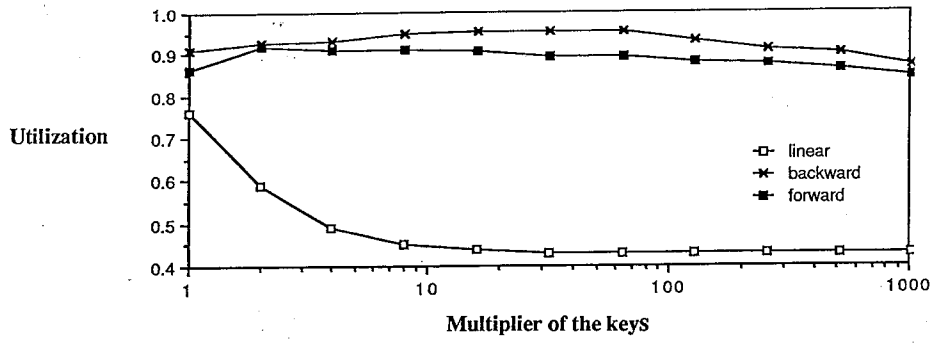


Fig. 7. The relationship between the storage utilization and the non-uniform key distribution.

almost every record is hashed into the same page. (Note that in Fig. 7, the X-axis has been replaced by the logarithmic function of x with base 2.) In this case, incremental hashing schemes can provide even better storage utilization than can linear hashing as the multiplier is increased. When the multiplier is increased (i.e., more data records are hashed into the same home page), storage utilization in linear hashing even drops below 50% while both incremental hashing schemes still keep storage utilization above 85%.

6 EXTENSION

Both incremental hashing schemes defined in Section 2 restrict the growth of a file at a rate $((n+1)/n)$ per full expansion, where n is the number of pages of the file; i.e., only one new page is added to the file after a full expansion. Under the split control by the load control L, $(n*L)$ more data records are disitributed into $(n+1)$ home pages in incremental hashing, instead of $(2*n)$ home pages in linear hashing, which results in better storage utilization in incremental hashing than in linear hashing as has been proved by both analysis and simulation results. However, the 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. In this section, we extend the proposed approach to have a growth rate of $((n+k-1)/n)$ per full expansion $(k \geq 2)$; i.e., $(k-1)$ more pages are added per full expansion, such that the number of disk accesses for data retrieval and insertion operations can be reduced. We describe the basic ideas, present address computation algorithms and discuss the simulation results.

6.1 Basic Ideas and Formal Definitions

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

For any record with $H_k(\text{key}) = c$:

(1) Extended forward incremental hashing:

$$h_0(c) = m_0, \quad \text{where } 0 \leq m_0 \leq (m-1)$$
$$h_{i+1}(c) = h_i(c) + c_i, \quad \text{for } i \geq 0,$$

where c_i is the value of the ith digit of c;

that is, $h_{i+1}(c) = m_0 + \sum_{j=0}^{i} c_j$, and $0 \leq h_{i+1}(c) \leq m_0 + (i+1)*(k-1)$.

(2) Extended backward incremental hashing:

$$h_0(c) = m_0, \quad \text{where } 0 \le m_0 \le (m-1)$$
$$h_{i+1}(c) = h_i(c) - c_i, \quad \text{if } h_i(c) \ge c_i \quad \text{or}$$
$$h_{i+1}(c) = (i+1)*(k-1) + (h_i(c) - c_i) + (m-1) + 1, \quad \text{otherwise;}$$

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

In general, in extended forward incremental hashing, when an insertion causes a split on page i (which is pointed to by sp and $0 \le i \le (n-k)$), where n is the number of pages of the file after the dth full expansion, the data records in page i will be redistributed among those k pages which are page i, $(i+1)$, ..., $(i+k-2)$ and $(i+k-1)$, according to the value of c_d. When a split occurs in page $(n-k+1)$, a new page n is added to the file and the data records on page $(n-k+1)$ will be redistributed among the current page and the next $(k-1)$ pages, where page n is newly added. Similarly, a split occurs in page i, where $(n-k+2) \le i \le (n-1)$ causing a new page, $(i+k-1)$, to be added to file. When a split occurs in page $(n-1)$, i.e., after a full expansion, page n to page $(n+k-3)$ are added in the previous splits, and page $(n+k-2)$ is added at this time.

By applying the concept similarly to extend backward incremental hashing, it is found that when a split occurs in page 0 during the $(d+1)$th full expansion, $(k-1)$ new pages are added to the end of the file; then, the data records in page 0 will be redistributed among page 0 and those $(k-1)$ new added pages according to the value of c_d When a split occurs in page i, where $1 \le i \le (k-2)$, the data records in page i will be redistributed among those k pages, which are page i, page $(i-1)$, ..., page 0, page $(n+k-2)$, page $(n+k-3)$, ..., page $(n+i)$. Otherwise, when $(k-1) \le i \le (n-1)$, the data records in page i will be redistributed into page i, page $(i-1)$, ..., page $(i-k+1)$.

6.2 Performance

In this subsection, we study the performance of these extended incremental hashing schemes by using a simulation technique and compare them with the original forward and backward incremental hashing schemes, respectively.

Tables 5 and 6 show the simulation results of extended forward incremental hashing and backward incremental hashing, respectively, and the results support our claims: As k is increased, the growth rate is increased, resulting in a decrease in storage utilization and the cost (in terms of the number of disk accesses) of data retrieval and insertion operations. From both of the simulation results, we observe that when k is 3 (or 4), both extended incremental hashing schemes can have better utilization than can linear hashing while having a lower cost of data retrieval and insertion operations than do all the other cases of k at the same time.

Figures 8-(a), (b), (c) show the relationship between the value of k and the cost of inserting a data record, and successfully and unsuccessfully searching for a data record, respectively, in extended forward hashing, where $N = 1000$, $b = 80$, $w = 40$ and $L = 96$. As k is increased, all of these costs are decreased. Moreover, these costs in extended forward incremental hashing can even drop below the costs of linear hashing at the cost of decreasing storage utilization. For example, when k is larger than 14, the average insertion cost will be lower than that in

Table 5. The simulation results in extended forward incremental hashing schemes.

Hashing Scheme	Parameters		L=0.8*b				L=b				L=1.2*b			
	b	w	I	ss	us	uti	I	ss	us	uti	I	ss	us	uti
forward	20	10	17	1.84	1.98	0.92	14	1.85	1.96	0.92	14	1.85	1.97	0.93
k3	20	10	11	1.81	1.93	0.86	11	1.81	1.93	0.86	10	1.81	1.99	0.89
k4	20	10	10	1.78	1.91	0.83	9.6	1.78	1.91	0.87	9.3	1.78	1.96	0.87
k5	20	10	8.6	1.76	1.88	0.80	7.9	1.76	1.88	0.80	9.0	1.76	1.88	0.85
k6	20	10	7.5	1.74	1.89	0.76	7.9	1.74	1.89	0.82	7.4	1.74	1.91	0.82
k7	20	10	7.1	1.72	1.83	0.76	7.5	1.72	1.83	0.76	6.9	1.72	1.83	0.76
k8	20	10	7.5	1.69	1.90	0.74	6.3	1.69	1.90	0.74	7.3	1.69	1.90	0.82
k9	20	10	6.7	1.67	1.93	0.70	6.1	1.67	1.93	0.80	6.9	1.67	1.97	0.80
k10	20	10	5.2	1.62	1.93	0.68	6.0	1.62	1.93	0.78	5.9	1.62	1.96	0.78
linear	20	10	2.6	1.00	1.00	0.78	2.7	1.14	1.40	0.78	2.8	1.23	1.66	0.78
forward	80	40	8.9	1.61	1.93	0.89	7.4	1.66	1.90	0.89	7.4	1.66	1.90	0.89
k3	80	40	7.4	1.50	1.90	0.86	6.4	1.61	1.93	0.78	5.8	1.61	1.93	0.80
k4	80	40	5.5	1.53	1.85	0.69	5.6	1.45	1.93	0.73	5.9	1.45	1.93	0.89
k5	80	40	5.2	1.35	1.83	0.65	5.5	1.35	1.83	0.83	4.8	1.47	1.80	0.78
k6	80	40	5.2	1.26	1.72	0.78	4.6	1.39	1.70	0.71	4.1	1.52	1.84	0.69
k7	80	40	4.9	1.24	1.63	0.71	4.0	1.46	1.78	0.62	3.7	1.47	1.95	0.62
k8	80	40	4.2	1.33	1.67	0.59	3.6	1.43	1.90	0.58	3.5	1.34	1.98	0.56
k9	80	40	3.8	1.33	1.81	0.56	3.4	1.31	1.97	0.53	3.3	1.19	2.00	0.56
k10	80	40	3.5	1.32	1.88	0.49	3.2	1.19	1.91	0.53	3.1	1.11	1.91	0.83
k12	80	40	3.0	1.10	1.79	0.45	2.9	1.01	1.25	0.50	2.8	1.08	1.68	0.83
k14	80	40	2.8	1.00	1.00	0.46	2.8	1.05	1.13	0.53	2.6	1.13	1.61	0.73
k16	80	40	2.8	1.00	1.00	0.49	2.5	1.10	1.30	0.63	2.3	1.12	1.50	0.67
linear	80	40	2.2	1.00	1.00	0.73	2.3	1.08	1.25	0.78	2.4	1.18	1.50	0.78

b: the size of a home page
w: the size of an overflow page
L: load control
k: base system
ss: successful search
us: unsuccessful search
uti: storage utilization
I: insertion cost

Table 6. The simulation results in extended backward incremental hashing schemes.

Hashing Scheme	Parameters		L=0.8*b				L=b				L=1.2*b			
	b	w	I	ss	us	uti	I	ss	us	uti	I	ss	us	uti
backward	20	10	12	1.78	1.84	0.93	10	1.80	2.00	0.95	8.7	1.64	2.00	0.93
k3	20	10	8.6	1.72	1.93	0.88	7.7	1.72	1.93	0.92	7.6	1.72	1.93	0.92
k4	20	10	7.4	1.67	1.89	0.84	6.4	1.67	1.89	0.89	6.5	1.67	1.89	0.89
k5	20	10	6.5	1.64	1.81	0.82	5.9	1.64	1.81	0.82	5.6	1.65	1.88	0.89
k6	20	10	6.0	1.60	1.89	0.77	5.4	1.60	1.88	0.84	5.2	1.60	1.88	0.84
k7	20	10	5.2	1.58	1.80	0.76	5.0	1.58	1.80	0.76	4.7	1.58	1.80	0.76
k8	20	10	4.9	1.54	1.84	0.72	4.6	1.55	1.84	0.73	4.0	1.55	1.86	0.81
k9	20	10	4.7	1.50	1.88	0.70	4.0	1.50	1.88	0.79	4.2	1.49	1.92	0.80
k10	20	10	4.4	1.46	1.87	0.67	3.9	1.46	1.90	0.76	3.9	1.46	1.88	0.76
linear	20	10	2.6	1.00	1.00	0.78	2.7	1.14	1.40	0.78	2.8	1.23	1.66	0.78
backward	80	40	5.9	1.52	2.00	0.93	5.9	1.52	2.00	0.89	5.6	1.60	2.00	0.93
k3	80	40	4.1	1.34	1.82	0.83	4.3	1.32	1.80	0.86	4.2	1.30	1.70	0.93
k4	80	40	3.6	1.15	1.63	0.76	3.5	1.25	1.81	0.86	3.4	1.34	1.82	0.80
k5	80	40	3.2	1.10	1.50	0.78	3.0	1.21	1.69	0.73	3.2	1.20	1.60	0.78
k6	80	40	2.7	1.13	1.59	0.64	2.9	1.12	1.59	0.64	3.1	1.09	1.41	0.67
k7	80	40	2.6	1.06	1.38	0.58	2.8	1.06	1.22	0.61	2.8	1.08	1.40	0.58
k8	80	40	2.7	1.04	1.20	0.54	2.7	1.04	1.20	0.54	2.6	1.08	1.40	0.52
k9	80	40	2.6	1.02	1.18	0.48	2.5	1.02	1.27	0.47	2.4	1.07	1.40	0.64
k10	80	40	2.5	1.00	1.09	0.44	2.4	1.01	1.18	0.43	2.3	1.04	1.28	0.61
k12	80	40	2.4	1.00	1.00	0.37	2.2	1.00	1.08	0.53	2.1	1.01	1.24	0.51
k14	80	40	2.2	1.00	1.00	0.46	2.0	1.00	1.00	0.46	1.9	1.00	1.00	0.46
k16	80	40	2.0	1.00	1.00	0.40	1.9	1.00	1.00	0.40	1.8	1.00	1.00	0.40
linear	80	40	2.2	1.00	1.00	0.73	2.3	1.08	1.25	0.78	2.4	1.18	1.50	0.78

b: the size of a home page
w: the size of an overflow page
L: load control
k: base system
ss: successful search
us: unsuccessful search
uti: storage utilization
I: insertion cost

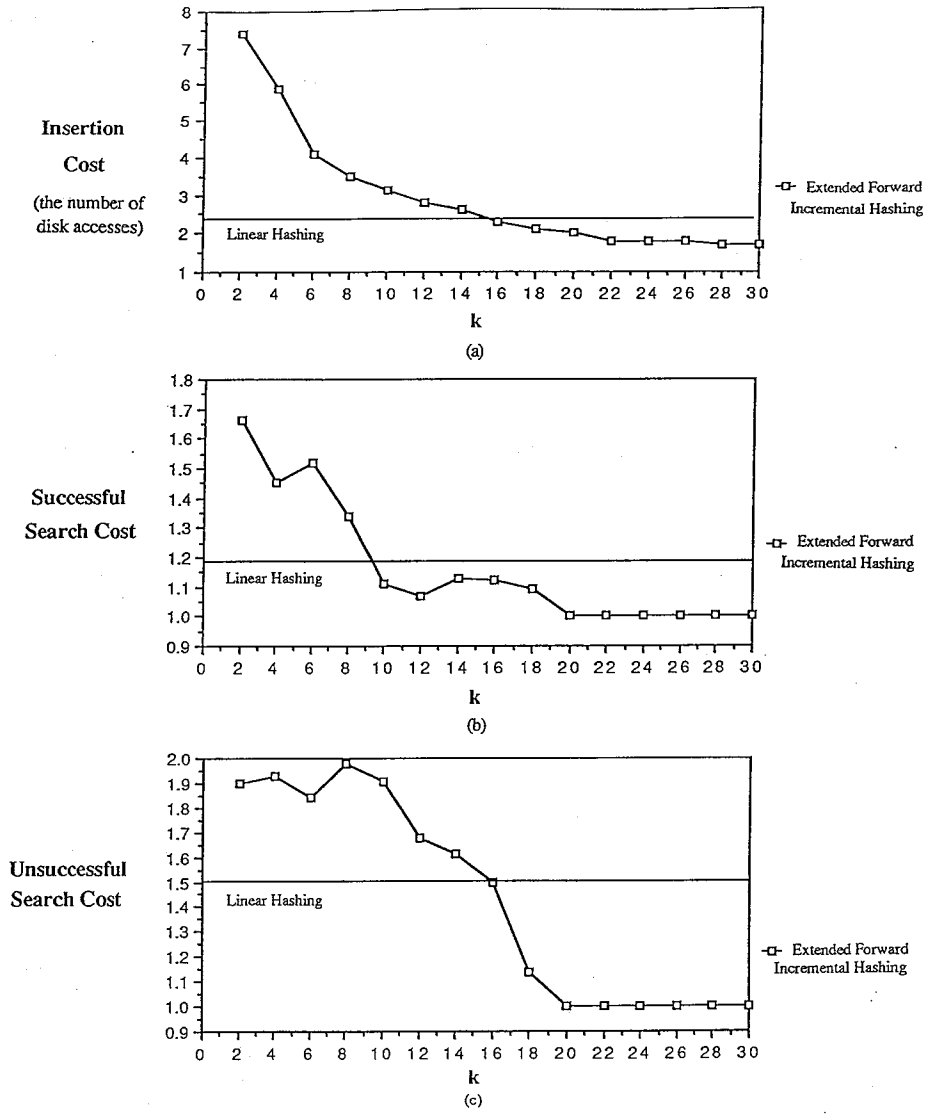


Fig. 8. The relationship between the value of k in forward incremental hashing and: (a) insertion cost; (b) successful search cost; (c) unsuccessful search cost.

linear hashing as shown in Fig. 8-(a). When k is larger than 8, the average successful search cost will be lower than that in linear hashing as shown in Fig. 8-(b). When k is larger than 16, the average unsuccessful search cost will be lower than that in linear hashing as shown in Fig. 8-(c).

Similarly, in extended backward incremental hashing, the average insertion cost will be lower than that in linear hashing. When k is larger than 8 as shown in Fig. 9-(a), the average successful search cost will be lower than that in linear

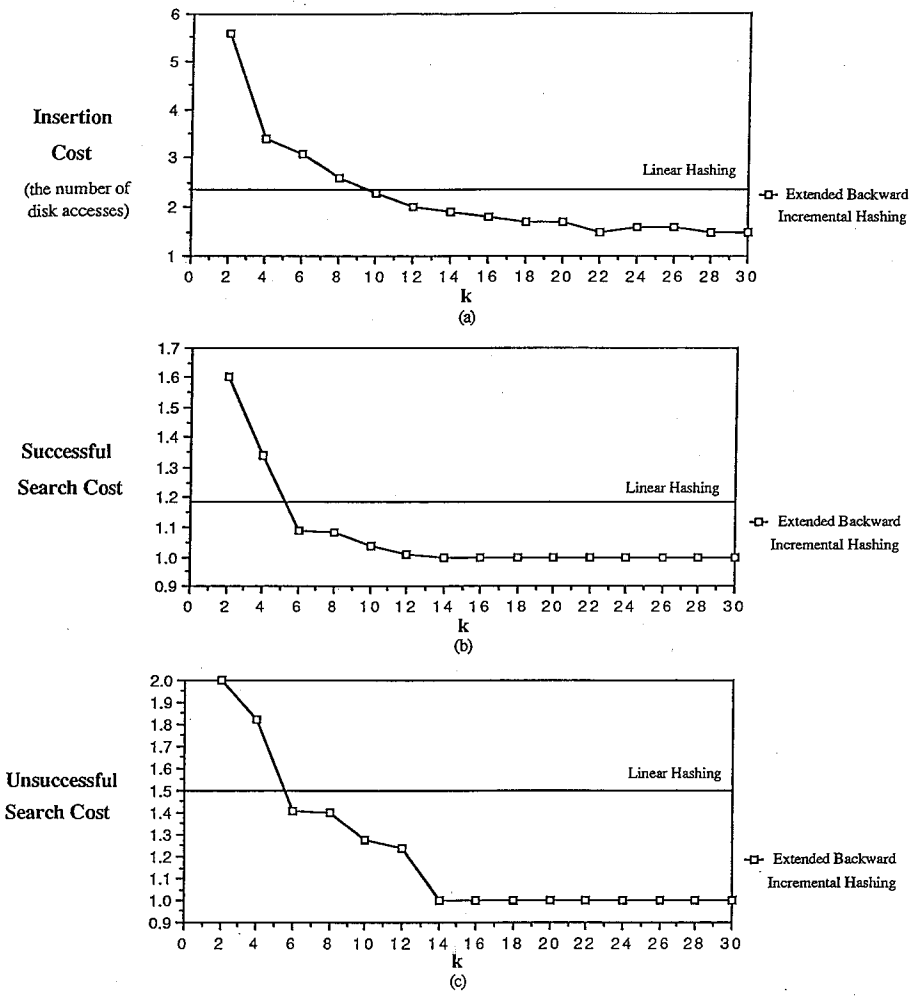


Fig. 9. The relationship between the value of k in backward incremental hashing and: (a) insertion cost; (b) successful search cost; (c) unsuccessful search cost.

hashing when k is larger than 5 as shown in Fig. 9-(b), and the average unsuccessful search cost will be lower than that in linear hashing when k is larger than 5 as shown in Fig. 9-(c).

Therefore, if we care about fast retrieval (and the low insertion cost) more than high storage utilization, we choose a k with a large value in extended incremental hashing. On the other hand, if we care about high storage utilization more than fast data retrieval (and low insertion cost), we choose a k with a small value. Since high storage utilization and fast data retrieval (and low insertion cost) are always a trade-off, the proposed extended incremental hashing provides a flexible choice between these two requirements.

7. CONCLUSION

In this paper, we have presented a new dynamic hashing approach, called incremental hashing. Incremental hashing requires no index and always adds only one more page after a full expansion; that is, the growth rate of a file per full expansion is $((n+1)/n)$ when n is the number of pages of the current size of the file. Two incremental hashing schemes have been presented, which are called forward incremental hashing and backward incremental hashing. Forward incremental hashing always splits forwards (i.e., redistribute data records in page i and page i and page $(i+1)$) while backward incremental hashing always splits backwards. By applying *separators* in both schemes to handle overflow pages, any data record is guaranteed to be accessed in at most two disk accesses. From our mathematical analysis and simulation study, both schemes have better storage utilization than does linear hashing, where forward incremental can achieve 94% storage utilization, and backward incremental hashing can achieve 95% storage utilization, as compared to 78% storage utilization in linear hashing when the keys are uniformly distributed. Moreover, both schemes can still achieve above 85% storage utilization while the storage utilization in linear hashing will drop below 50% when the keys are not uniformly distributed.

Moreover, since high storage utilization and fast data retrieval are a trade-off in all dynamic hashing schemes, we have extended both incremental hashing schemes to have a growth rate of a file of $((n+k-1)/n)$ in order to find a compromise between better storage utilization and fast data retrieval. From our simulation study, we can find that when k is 3 (or 4), both extended incremental hashing schemes can have better utilization that can linear hashing while having a lower cost of data retrieval and insertion operations than can all the other cases of k at the same time. If we care about fast retrieval (and low insertion cost) more than high storage utilization, we choose a k with a large value in extended incremental hashing. On the other hand, if we care about high storage utilization more than fast data retrieval (and low insertion cost), we choose a k with a small value. Therefore, the proposed extended incremental hashing provides a flexible choice between these two requirements.

Linear hashing with partial expansions [8] and the use of different split sequences are extensions to Litwin's basic algorithm of linear hashing to make uniform the load distribution [13], which, in turn, improves performance in storage utilization or insertion/retrieval operations. Since our incremental hashing is a new approach to dynamic hashing without an index, i.e., it has the same position as Litwin's linear hashing among its variants, extending incremental hashing with parital expansions (or with the use of different split sequences) is a possible future research direction.

REFERENCES

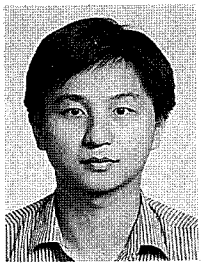
1. Bechtold, U. and Kuspert, K., "On the Use of Extendible Hashing without Hashing," *Information Processing Letters*, Vol. 19, No. 1, 1984, pp. 21-26.
2. Enbody, R.J. and Du, H.C., "Dynamic Hashing Schemes," *ACM Computing Surveys*, Vol. 20, No. 2, 1988, pp. 85-113.

3. Hachem, Nabil I. and Berra, P. Bruce, "Key-sequential Access Method for Very Large Files Derived from Linear Hashing," in *Proc. of the 5th International Conference on Data Engineering*, 1989, pp. 305-312.
4. Hachem, Nabil I. and Berra, P. Bruce, "New Order Preserving Access Method for Very Large Files Derived from Linear Hashing," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 1, 1992, pp. 68-82.
5. Hutflesz, A., Six, H-W. and Widmayer, P., "Globally Order Preserving Multidimensional Linear Hashing," in *Proc. of the 4th International Conference on Data Engineering*, 1988, pp. 572-579.
6. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R., "Extendible Hashing-A Fast Access Method for Dynamic Files," *ACM Trans. on Database Systems*, Vol. 4, No. 3, Sep. 1979, pp. 315-344.
7. Larson, P., "Dynamic Hashing," *BIT*, Vol. 18, 1978, pp. 184-201.
8. Larson, P., "Linear Hashing with Partial Expansions," in *Proc. of the 6th International Conference on Very Large Data Bases*, 1980, pp. 224-232.
9. Larson, P., "A Single-File Version of Linear Hashing with Partial Expansions," in *Proc. of the 8th International Conference on Very Large Data Bases*, 1982, pp. 300-309.
10. Larson, P., "Performance Analysis of Linear Hashing with Partial Expansions," *ACM Trans. on Database Systems*, Vol. 7, No. 4, 1982, pp. 566-587.
11. Larson, P., Kajla, Ajay, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access," *ACM Computing Practices*, Vol. 27, No. 7, 1984, pp. 670-677.
12. Larson, P., "Linear Hashing with Overflow-Handling by Linear Probing," *ACM Trans. on Database Systems*, Vol. 10, No. 1, 1985, pp. 75-89.
13. Larson, P., "Linear Hashing with Seperators-A Dynamic Hashing Scheme Achieving One-Access Retrieval," *ACM Trans. on Database Systems*, Vol. 13, No. 3, 1988, pp. 366-388.
14. Lee, Chien-I, *Design and Analysis of Dynamic Hashing Schemes Without Indexes*, Master Thesis, Dept. of Applied Mathematics, National Sun Yat-Sen University, R.O.C., June 1993.
15. Litwin, W., "Linear Hashing: A New Tool for Files and Tables Addressing," in *Proc. of the 6th International Conference on Very Large Data Bases*, 1980, pp. 212-223.
16. Lomet, David B., "Bounded Index Exponential Hashing," *ACM Trans. on Database Systems*, Vol. 8, No. 1, 1983, pp. 136-165.
17. Lomet, David B., "Partial Expansions for File Organizations with an Index," *ACM Trans. on Database Systems*, Vol. 12, No. 1, 1987, pp. 65-84.
18. Lum, V.Y., Yuen, P.S.T. and Dodd, M., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," *Communications of the ACM*, Vol. 14, No. 4, 1971, pp. 228-239.
19. Mendelson, Haim, "Analysis of Extendible Hashing," *IEEE Trans. on Software Engineering*, Vol. 8, No. 6, 1982, pp. 611-619.
20. Mullin, J.K., "Tightly Controlled Linear Hashing without Separate Overflow Storage," *BIT*, Vol. 21, No. 4, 1981, pp. 390-400.
21. Otto, E.J., "Linearizing the Directory Growth in Order Preserving Extendible Hashing," in *Proc. of the 4th International Conference on Data Engineering*,

1988, pp. 580-588.
22. Ramamohanarao, K. and Lloyd, John W., "Dynamic Hashing Schemes," *The Computer Journal*, Vol. 25, No. 4, 1982, pp. 478-485.
23. Ramamohanarao, K., "Recursive Linear Hashing," *ACM Trans. on Database Systems*, Vol. 9, No. 3, 1984, pp. 369-391.
24. Robin, J.T., "Order Preserving Linear Hashing Using Dynamic Key Statistics," in *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984, pp. 91-99.
25. Scholl, Michel, "New File Organizations Based on Dynamic Hashing," *ACM Trans. on Database Systems*, Vol. 6, No. 1, 1981, pp. 194-211.
26. Tharp, A.L., *File Organization and Processing*, Chap. 10, John Wiley and Sons, 1988, pp. 254-285.
27. Veklerov, Eugene, "Analysis of Dynamic Hashing with Deferred Splitting," *ACM Trans. on Database Systems*, Vol. 10, No. 1, 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.

She joined the Department of Applied Mathematics, National Sun Yat-Sen University, Kaohsiung, Taiwan, in 1991, where she is now an associate professor. Her research interests include database systems, distributed systems, knowledge-based systems and computer networks.



Chien-I Lee (李建億) was born in Taipei, Taiwan, R.O.C., on Aug. 13, 1965. He received the B.S. degree in computer science from Feng Chia University in 1987 and M.S. degree in applied mathematics from National Sun Yat-Sen University in 1993. He is now a Ph.D. student at the Computer and Information Science Department, National Chiao Tung University.