Linear Hashing with Overflow-Handling by Linear Probing

PER-ÅKE LARSON
University of Waterloo

Linear hashing is a file structure for dynamic files. In this paper, a new, simple method for handling overflow records in connection with linear hashing is proposed. The method is based on linear probing and does not rely on chaining. No dedicated overflow area is required. The expansion sequence of liner hashing is modified to improve the performance, which requires changes in the address computation. A new address computation algorithm and an expansion algorithm are given. The performance of the method is studied by simulation. The algorithms for the basic file operations are very simple, and the overall performance is competitive with that of other variants of linear hashing.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hashing, dynamic hashing schemes, extendible hashing, open addressing, linear probing

1. INTRODUCTION

Linear hashing is a file organization intended for files that grow and shrink dynamically. The technique was originally proposed by Litwin [5]. Larson [1, 2], and subsequently Ramamohanarao and Lloyd [7], developed more general schemes with substantially improved performance.

Linear hashing requires some method for handling overflow records. Not all of the methods developed for traditional hashing schemes are applicable because linear hashing requires that all records hashing to a page be locatable. The three schemes mentioned above were all developed assuming that separate chaining is used; that is, records overflowing from a page are placed somewhere in a separate overflow area and linked into an overflow chain emanating from the home page. In this paper, an overflow-handling method based on open addressing is presented. The technique used is linear probing, that is, if a record does not fit into its home page, the next higher pages are tried until the first nonfull page is found. There is no dedicated overflow area, and no pointers are used.

This work was supported by the National Sciences and Engineering Research Council of Canada under grant A-2460.
Author's address: Dept. of Computer Science, University of Waterloo, Waterloo, Ont. N2L 3G1, Canada.

Using linear probing for handling overflow records has a number of advantages. The retrieval and insertion algorithms are extremely simple. All problems associated with managing a separate overflow area vanish. Linear probing preserves locality of reference, thus (potentially) avoiding long disk seeks. Insertions and file expansions can easily be speeded up by allocating more buffer space, reading or writing several consecutive pages when accessing the disk.

Several other techniques for handling overflow records in connection with linear hashing have been studied. Mullin [6] investigated prime area chaining; that is, chaining is used but overflow records are stored in nonfull primary pages, not in a dedicated overflow area. Larson devised [3] and analyzed [4] a technique for combining the overflow area and the primary pages in the same file. Ramamohanarao and Sacks-Davis [8] proposed recursive linear hashing where overflow records from a linear hash file are stored in another (smaller) linear hash file, overflow records from the second level file go into a third level linear hash file, and so on.

The new method presented here is a modification of Larson's scheme [1], but the same idea can be applied to that of Ramamohanarao and Lloyd [7]. However, the latter method requires substantially more buffer space during expansions. In all other respects its performance is very close to that of Larson's scheme.

2. LINEAR HASHING WITH PARTIAL EXPANSIONS

Linear hashing is a technique for gradually expanding (or contracting) the storage area of a hash file. The file is expanded by adding a new page at the end of the file and relocating a number of records to the new page. The basic ideas of the method are briefly outlined in this section.

The original scheme proposed by Litwin [5] proceeds by first splitting page 0, then page 1, and so on. Consider a file consisting of N pages with addresses 0, 1, ..., $N-1$. When page j, $j = 0, 1, \ldots, N-1$ is split, the file is extended by one page with address $N+j$. Approximately half of the records whose current address is j are moved to the new page. A pointer p keeps track of which page is the next one to be split. When all the N original pages have been split, the file size has doubled. The pointer p is reset to zero and the process starts over again. A doubling of the file is called a *full expansion.*

The key idea is to split the pages in a predetermined sequence. After splitting a page, it should be possible to locate the records moved to the new page without having to access the old page. The essence of the problem is to design an algorithm which, based on the record key, determines whether a record remains on the old page or is moved to a new page, and which also ensures that approximately half of the records are moved. There are several solutions to this problem [1, 5, 7].

The development of linear hashing with partial expansions was motivated by the observation that linear hashing creates a very uneven distribution of the load over the file. This slows down retrieval and insertions by creating a large number of overflow records. The load of a page that has already been split is expected to be only half of the load of a page that has not yet been split. To achieve a more even load distribution, the doubling of the file (a full expansion) is carried out by a series of partial expansions. If two partial expansions are used, the first one

increases the file size to 1.5 times the original size and the second one to twice the original size.

Assume that two partial expansions per full expansion are used. We start from a file of $2N$ pages, logically divided into N groups of two pages each. Group j consists of pages $(j, N + j)$, $j = 0, 1, \ldots, N - 1$. To expand the file, group 0 is first expanded by one page, then group 1 by one page, and so on. When expanding group j, $j = 0, 1, \ldots, N - 1$, approximately one-third of the records from page j and $N + j$ are relocated to the new page $2N + j$. When the last group $(N - 1, 2N - 1)$ has been expanded, the file has increased to $3N$ pages. The second partial expansion starts, the only difference being that now groups of three pages $(j, N + j, 2N + j)$ are expanded to four pages. When the second partial expansion is completed, the file size has doubled from $2N$ to $4N$ pages. The next partial expansion reverts to expanding groups of size two $(j, 2N + j)$, $j = 0, 1, \ldots, 2N - 1$. The one after that expands groups of size three, and so forth. This approach can immediately be generalized to any number of partial expansions per full expansion.

To implement linear hashing with partial expansions, an address computation algorithm is required. Such an algorithm is given in [1]. It computes the (current) home address of a record at any time, given its key and the current file size. It is designed for any number of partial expansions per full expansion.

The scheme outlined above gives a method for expanding the file one page at a time. In addition, rules for determining *when* to expand (or contract) the file are needed. Several alternatives are possible; but here we consider only the rule of constant storage utilization. According to this rule, the file is expanded whenever the overall storage utilization rises above a threshold α', $0 < \alpha' < 1$, selected by the user. Storage utilization is defined as $\alpha = k/(bm)$, where k records are stored in m pages, each having a capacity of b records; m includes pages allocated for overflow records, if any.

The results reported in [2] show that increasing the number of partial expansions improves the retrieval performance, as expected. On the other hand, insertion costs tend to increase. Two partial expansions per full expansion seems to be a good compromise.

3. HANDLING OVERFLOW BY LINEAR PROBING

An overflow-handling method based on open addressing must satisfy certain requirements to be applicable to linear hashing. As will become clear from the discussion in this section, linear probing satisfies these requirements. It also offers a number of other advantages.

The cost of expanding a linear hash file is directly affected by the cost of locating all records hashing to a given page. An expansion involves locating all records hashing to a number of existing pages and relocating some of them to the new page. A method generating a large number of possible probe sequences, double hashing, for example, would make this too costly. Each probe sequence emanating from a page participating in the expansion must be checked. Linear probing generates only one probe sequence from a page. Furthermore, the probe sequence is the same as the physical address sequence. Provided that sufficient buffer space is available, a (physical) read or write operation can be made to

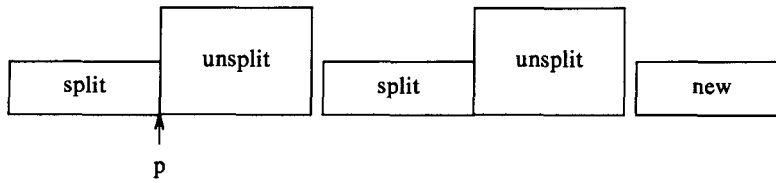

Fig. 1. Illustration of the load distribution in the middle of an expansion.

transfer several consecutive pages between disk and main memory. This will reduce the number of accesses to the disk and speed up insertions and expansions.

When expanding the file, all existing probe sequences must somehow be extended to include the new page created. It is desirable that this be done without any actual relocation of records. This is easily achieved by modifying linear probing so as not to wraparound to the first page when reaching the (currently) last page of the file. If there are overflow records from the last page in the current address space, they are allowed to go into the first unused page at the end of the file. In effect the page has, prematurely, been taken into use by receiving overflow records before receiving any "native" records. The next time the file is expanded it will be within the address space of the file. It is unlikely that more than one such overflow page would be needed. The effect of this modification is that each record has, in principle, an infinite probe sequence.

Linear hashing with partial expansion extends the file by one page by increasing the size of one group. The expansion sequence discussed in the previous section was group 0, group 1, and so on. This particular sequence is not crucial for the method; it merely simplifies the address computation.

The expansion sequence 0, 1, ... has a serious drawback when overflow records are handled by linear probing. As illustrated in Figure 1 for the case of two partial expansions, it creates blocks of consecutive pages with a high load factor (the unsplit pages). In these areas, long islands of full pages are more likely to arise. Long islands of full pages slow down insertions, subsequent retrieval, and expansions. They can be avoided to some extent by changing the expansion sequence in such a way that the split pages with a lower load are spread more evenly over the file. The expansion sequence 0, 1, ... uses a step length of one. We can instead use a larger step length and make a number of sweeps over the groups. If a step length of s, $s \geq 1$ is used, the first sweep would expand groups $0, s, 2s, \ldots$, the second sweep groups $1, s + 1, 2s + 1, \ldots$, and the last sweep would be $s - 1, 2s - 1, 3s - 1, \ldots$. This achieves the desired effect of spreading the split pages more evenly over the file.

One further modification of the expansion sequence is proposed: to have each sweep go backwards instead of forwards. If the file consists of N groups, the first sweep would be $N - 1, N - 1 - s, N - 1 - 2s, \ldots$, and correspondingly for the other sweeps. Going backwards is not more expensive than going forwards, and it has some beneficial effects. The average cost of an expansion is slightly reduced. The cost will also be more uniform. Pages in the beginning of the address space receive overflow records from fewer pages than those with high addresses. They are therefore less likely to overflow, which in turn means that long islands of full pages are less likely to form. Consequently, expansions involving pages with low

addresses are expected to be less expensive than expansions involving pages with high addresses. Expansions immediately before the end of a sweep are the most expensive. By going backwards they involve pages with low addresses, which will slightly reduce the cost. To see why the expansion costs will be more uniform, consider the situation when the expansion sequence encounters a long island of full pages. If we are proceeding forwards, the first expansion to reach the island will be quite expensive. All pages up to the end of the island must be checked. This first expansion is likely to create a "hole" in the island, thereby reducing the cost of the next few expansions. This "hole" may occur anywhere on the island. When going backwards there will be less variation. Every expansion is likely to shorten the island by a few pages, thereby reducing the cost of the next expansion. However, the island cannot be shortened by more than the step length.

Example. When using 3 sweeps, the expansion sequence for a file consisting of 8 groups is as follows. The first sweep expands groups 7, 4, and 1 (in that order), the second sweep groups 6, 3, 0, and the last one groups 5 and 2.

It is suggested that the expansion rate is controlled by the following rule: the file is expanded whenever the overall load factor increases over a user-selected threshold α', $0 < \alpha' < 1$. Because there is no separate overflow area, this rule will result in a storage utilization that, for all practical purposes, is constant and equal to α'. The performance analysis in Section 5 assumes the use of this rule.

4. ALGORITHMS

In this section we discuss algorithms needed to implement linear hashing with overflow handling by linear probing. Only two algorithms are given in detail: one for address computation and one for file expansion. Insertion, retrieval, and deletion need be discussed only briefly. A linear hash file with linear probing is defined by a number of parameters, and its current state by a number of state variables:

Parameters

b	page size in number of records
N	original number of groups
n_0	number of partial expansions per full expansion
s	number of sweeps per partial expansion (or step length used)
α'	target load factor

State variables

cpx	current partial expansion, $cpx \geq 1$. Initial value: $cpx = 1$
sw	current sweep, $1 \leq sw \leq s$. Initial value: $sw = 1$
p	next group to be expanded, $0 \leq p < N \times 2^k$, where $k = (cpx - 1)$ **div** n_0. Initial value: $p = N - 1$
maxadr	highest address in current address space. Initial value: $maxadr = n_0 N - 1$
lstpg	highest page in use, $lstpg \geq maxadr$. Initial value: $lstpg = maxadr$

The address computation algorithm given below makes use of two hashing functions. The first one, denoted by h, is a normal hashing function, $0 \leq h(K) \leq n_0 N - 1$, and is used for distributing the records over the original file (of size n_0

N pages). The second one, denoted by D, returns a sequence of values $D(K) = (d_1(K), d_2(K), \ldots)$, where the values d_i are uniformly distributed in [0, 1). The value $d_i(K)$ is used to determine whether to relocate the record with key K to the new page during the ith partial expansion. Assume that the ith partial expansion expands each group from n to $n + 1$ pages. To achieve a uniform distribution of the load over the file (at the end of the expansion), approximately $1/(n + 1)$ of the records hashing to the n "old" pages should be relocated to the new page. This can be achieved by relocating a record if $d_i(K) \leq 1/(n + 1)$ (changing its address), otherwise not. The group size during the ith partial expansion is $n = n_0 + (i - 1)$ **mod** n_0. The address computation algorithm given below is based on this idea. The hashing function D can easily be implemented by a random number generator to which the key is provided as the seed.

```
procedure address (K : key): integer;
begin
  ha, fsz, ngrps, i, k, swp, swpl, fsw, npg: integer;
  ha: = h(K);
  fsz: = n0 × N;
  ngrps: = N;
  for i: = 1 to cpx do begin
    if di(K) ≤ 1/(n0 + 1 + (i − 1) mod n0) then begin
      k: = ngrps − 1 − (ha mod ngrps);
      swp: = k mod s;
      swpl: = ngrps div s;
      fsw: = swp × swpl + min (swp, ngrps mod s);
      npg: = fsz + fsw + (k div s);
      if npg ≤ maxadr then ha: = npg;
    end;
    fsz: = fsz + ngrps;
    if(i)mod n0 = 0 then ngrps: = 2 × ngrps;
  end;
  return (ha);
end;
```

The current address of a record is computed by tracing all its address changes from the first to the current partial expansion. If the record was moved during the ith expansion (or would have been moved, had it existed in the file), the address of the new page must be computed. This is done by adding the file size when the ith expansion started (*fsz*), the number of pages created by fully completed sweeps (*fsw*), and by the last, partially completed, sweep (the term (k **div** s) + 1). If $npg \leq maxadr$, the new address is within the current address range. This test can fail only when $i = cpx$, indicating that the current partial expansion has not yet reached the page in question.

Once the home address has been computed, insertion or retrieval of a record is done in the same way as for traditional linear probing. The only difference is that the probe sequence of a record does not wraparound when it reaches the last page of the file. If the last page is reached during an insertion and it is full, then the next page is taken into use and *lstpg* is increased by one.

We expand the file as soon as the overall load factor rises above the threshold α'. An algorithm for expanding the file by one page is given below. File expansion

necessitates some (local) rearrangement of records. When records are moved to the new page, space will be freed up in the old part of the file. To retain searchability these holes must, whenever possible, be filled by moving overflow records back to, or at least closer to, their home pages.

Consider a page participating in an expansion and denote its address by *pg*. All records whose home address is *pg* must be checked because some of them will be moved to the new page. To find these records, page *pg* is first checked, then $pg + 1$, and so on, up to and including the first nonfull page. Let us call this area the search area. The expansion algorithm scans over the search area twice. The first scan collects every record within the search area that is not stored on its home page. The set of collected records will include two types of records: those that are to be moved to the new page and those that will remain within the search area, but which may be moved closer to their home pages. The collected records are temporarily stored in an area called the record pool. In addition to the record itself, its home address is also stored. To avoid writing, pages are not modified during the first scan.

The second scan goes over the same area as the first scan, restoring records collected during the first scan. Only records that are to remain within the search area are restored; those that are to be moved to the new page are left in the pool. Whenever there is an empty slot on a page being scanned, the algorithm attempts to fill that slot with a record from the record pool. The record selected is the one with the lowest home address.

The above process is repeated for every old page participating in the current expansion. There are $np = n_0 + (cpx - 1)$ **mod** n_0 such pages. After this stage, the only records remaining in the record pool are those that are to be moved to the new page. Their home address is exactly equal to the address of the new page. They are inserted in the last part of the algorithm.

Assume that there are *np* old pages involved in an expansion and that there were x records hashing to those pages. To achieve a uniform distribution, approximately $x/(np + 1)$ of those records should be relocated to the new page created. The address computation algorithm is designed so that the probability of a record being moved is $1/(np + 1)$. This ensures that the *expected* number of records being relocated is $x/(np + 1)$, but for any given expansion the actual number may be different.

A page in the file consists of b record slots and each slot consists of three parts: a status field, the key of the record, and the rest of the record. The status field indicates whether the slot is empty or full. To keep the algorithm simple, it was written so as to use only one buffer page. No particular ordering of the records on a page is assumed. Modifying the algorithm to use several buffer pages or a more efficient organization within a page requires only local changes.

The structure of the record pool is intentionally left unspecified. The actual implementation will (mainly) depend on the amount of internal storage available. If sufficient main memory space is available, all records in the record pool can reside in main memory, otherwise some or all of them will have to be temporarily stored on disk. The size of the record pool is one of the performance variables studied in the next section.

```
procedure expand (p : integer)

{type declarations}

slot = record
          status: (empty, full);
          key; {record key}
          rest; {additional fields}
       end;
page = array 1..b of slot;

{local variables}
buffer : page;
{declarations for the record pool go here}
i, j, gr, np, lvl, ngr, n, pg, cp, lmdf, cnt, adr : integer;

gr: = p;
np: = n0 + (cpx − 1) mod n0;
lvl: = (cpx − 1) div n0;
ngr: = N × 2 ↑ lvl;

{update state variables}
maxadr: = maxadr + 1;
p: = p − s;
if p < 0 then begin {next sweep}
  sw: = sw + 1; p: = ngr − sw;
  if sw > s then begin {next partial expansion}
    cpx: = cpx + 1;
    sw: = 1; p: = ngr − 1;
    if (cpx − 1) mod n0 = 0 then {begin next full expansion}
      p: = 2 × ngr − 1;
  end;
end;
for i: = 1 to np do begin
  pg: = gr + (i − 1) × ngr;

  {first scan: from page pg to the first nonfull page; collect every record not located on its
  home page and store them in the record pool}
  cp: = pg − 1; lmdf: = pg − 1;
  repeat
    cp:= cp + 1;
    read page cp into buffer;
    cnt: = 0;
    for j: = 1 to b do
      if buffer[j].status=full then begin
        cnt: = cnt + 1;
        adr: = address(buffer[j].key);
        if adr ≠ cp then begin
          lmdf: = cp;
          insert (buffer[j], adr) into rcrdpool;
        end;
      end;
  until cnt < b or cp = lstpg;

  {second scan: reinsert all records from the record pool whose address is ≤ lmdf}
```

```
  for cp: = pg to lmdf do begin
    read page cp into buffer;
    for j: = 1 to b do begin
      {if the slot contains a record collected during the first scan, declare it empty}
      if buffer[j].status=full and address(buffer [j].key) ≠ cp
      then buffer[j].status=empty;
      {attempt to fill empty slots}
      if buffer[j].status=empty and not empty(rcrdpool)
      then begin
        {note: the records in rcrdpool are assumed to be sorted
        in ascending order on home address}
        adr:=[home address of first record in rcrdpool];
        if adr ≤ cp then begin
          buffer[j]:=[first record in rcrdpool];
          buffer[j].status:=full;
          delete first record from rcrdpool;
        end;
      end;
    end {j-loop};
    write buffer into page cp;
  end {cp-loop};
end{i-loop};
```

{at this point all records remaining in the record pool have home address = *maxadr*, so insert them into the new page}

```
cp: = maxadr − 1;
repeat
  cp: = cp + 1;
  if cp ≤ lstpg
    then read page cp into buffer
    else for j: = 1 to b do buffer[j].status:=empty;
  j: = 0;
  while j < b and not empty (rcrdpool) do begin
    j:= j + 1;
    if buffer[j].status=empty then begin
      buffer[j]:=[first record in rcrdpool];
      buffer[j].status:=full;
      delete first record from rcrdpool;
    end;
  end;
  if lstpg < cp then begin
    lstpg : = cp;
    expand the file space up to lstpg;
  end;
  write buffer into page cp;
until empty (rcrdpool);
end {expand};
```

Deletions can be handled in one of two ways: by marking or by "true" deletion. Deletion by marking is extremely simple. The record to be deleted is somehow marked as deleted, and no attempt to fill the hole immediately is made. However, later insertions may fill the hole with a new record. Holes may also disappear as a result of the local reorganization done during an expansion operation. During retrieval, records marked for deletion are treated the same way as unmarked records, except that a marked record is never returned. Deletion by marking is

simple to implement, but it has the effect that the performance of the file will slowly deteriorate unless the deletion rate is very low compared with the insertion rate. This effect of marking is well known for linear probing and most other overflow handling schemes.

A "true" deletion means that we immediately try to fill the hole left by the deleted record by moving an overflow record closer to its home page, then the new hole must be filled, and so on. The process stops when we reach the first, nonfull page (there are no more overflow records that need be checked). This method is more expensive than marking, but it ensures that the performance of the file does not deteriorate. The algorithm is straightforward and exactly the same as for traditional linear probing. If two buffer pages are used, the number of disk accesses required is at most twice that of an unsuccessful search.

5. PERFORMANCE

In order to study the performance of the new method and the effects of parameter changes, a simulation model was built. Results obtained by a series of simulation experiments are presented and discussed in this section. The following performance measures are considered: average number of disk accesses for successful and unsuccessful searches, average number of disk accesses for insertion of a record, and average size of the record pool used during an expansion. Internal processing time is ignored because the total elapsed time for an operation is dominated by the time for disk accesses.

The performance will vary cyclically, with a cycle corresponding to a full expansion. The performance behavior over a full expansion is illustrated in Figures 2 to 5. The parameters for the example file are: page size 20, storage utilization 0.8, and 2 partial expansions per full expansion. The results plotted are the averages from 100 simulated file loadings. Two different cases are shown: 2 sweeps (solid line) and 4 sweeps (dotted line) per partial expansion. Note that the insertion costs in Figure 4 include both the accesses required to insert a record and the accesses required for file expansion.

The full expansion from 1000 to 2000 pages depicted in Figures 2 to 5 consists of two partial expansions: the first one from 1000 to 1500 pages and the second from 1500 to 2000 pages. Both retrieval and insertion costs are lower during the second partial expansion. Each "hump" in the graphs represents one sweep. A local maximum occurs close to the end of each sweep. Increasing the number of sweeps, in this case from 2 to 4, clearly improves the performance. It reduces the overall average and the variations.

The size of the record pool was computed as follows. Consider an expansion operation adding one page to the file. During the expansion the number of records in the pool varies. The space occupied by the pool is proportional to the maximum number of records in the pool during that expansion. The maximum number of records in the pool was recorded for every expansion operation, and the results plotted in Figure 5 are averages of over 100 simulated file loadings.

Table I shows how increasing the number of sweeps affects the performance. The figures are averages over a full expansion. As seen from the table, the performance first improves and then slowly deteriorates again. Around 5 sweeps per partial expansion appears to be optimal.

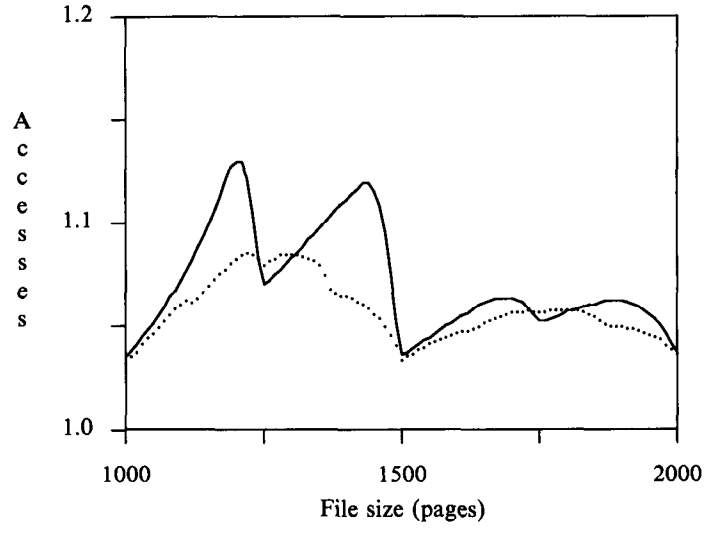

Fig. 2. Average number of disk accesses for a successful search.

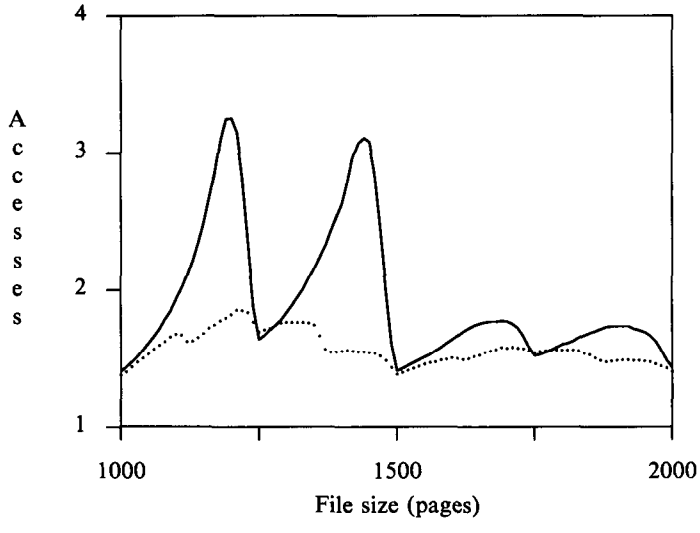

Fig. 3. Average number of disk accesses for an unsuccessful search.

Table II shows the (overall) average performance for a few combinations of page size, storage utilization, and number of partial expansions. The number of sweeps is 5 in all cases. The overall behavior is typical for linear probing: increasing the page size improves the performance significantly, and the performance starts deteriorating rapidly when the storage utilization is pushed over the 0.80–0.85 range (earlier for small pages, later for large pages).

Several other methods for handling overflow records in connection with linear hashing have been proposed [1, 3, 6, 7, 8]. The method described in [6] will not

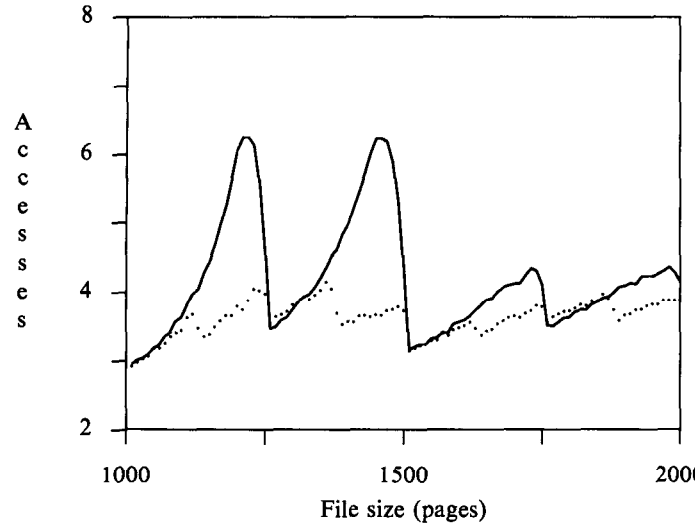

Fig. 4. Total insertion costs, including expansion costs, in average number of disk accesses.

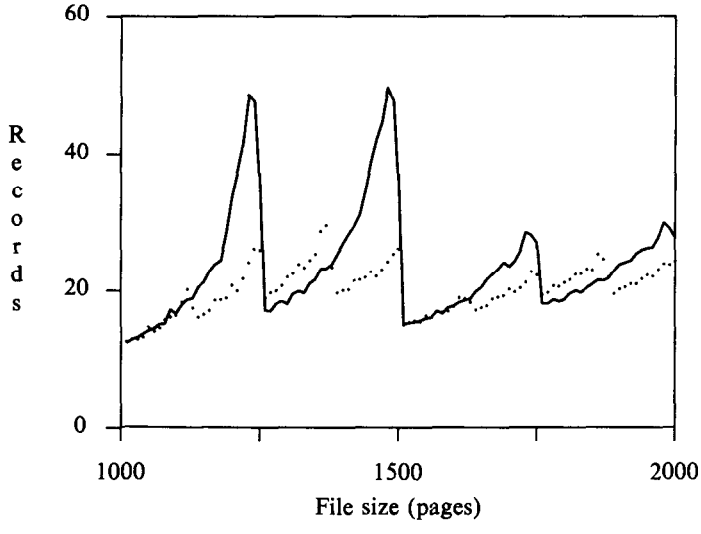

Fig. 5. Average size of record pool (in number of records).

be considered further; its performance is consistently worse than that reported in [1, 3]. A direct comparison with the methods suggested in [7, 8] is not possible because the expansion rate is controlled differently. In those two methods the storage utilization is not kept constant. However, it is known that the performance of the method in [7] is very close to that of the method in [1].

Let us first consider the retrieval performance. Compared with [1, 3], the number of accesses required for a successful search is essentially the same for a bucket size of 20 or 40 records. For bucket size 10, the current method is slower for high storage utilization (0.9). With respect to unsuccessful searches the

Table I. Average Performance as a Function of the Number of Sweeps

Sweeps	Number of disk accesses			Size of record pool
	Successful search	Unsuccessful search	Insertion (total)	
1	1.48	9.66	16.43	91.6
2	1.07	1.92	4.19	23.6
3	1.06	1.65	3.77	21.1
4	1.06	1.59	3.67	20.7
5	1.06	1.59	3.67	20.7
6	1.06	1.61	3.69	21.0
8	1.07	1.66	3.77	21.7
10	1.07	1.70	3.82	22.1

$b = 20$, $\alpha' = 0.8$, $n_0 = 2$

Table II. Average Performance Over a Full Expansion

n_0	α'	Successful search (disk accesses)			Unsuccessful search (disk accesses)		
		$b = 10$	$b = 20$	$b = 40$	$b = 10$	$b = 20$	$b = 40$
2	0.7	1.06	1.02	1.01	1.40	1.17	1.07
	0.8	1.14	1.06	1.03	2.22	1.60	1.32
	0.9	1.51	1.25	1.13	9.93	5.49	3.42
3	0.7	1.05	1.01	1.00	1.36	1.13	1.03
	0.8	1.12	1.05	1.02	2.10	1.49	1.22
	0.9	1.40	1.18	1.08	6.81	3.85	2.38

n_0	α'	Insertion cost (total) (disk accesses)			Size of record pool (number of records)		
		$b = 10$	$b = 20$	$b = 40$	$b = 10$	$b = 20$	$b = 40$
2	0.7	4.30	2.94	2.41	8.7	14.3	25.9
	0.8	6.13	3.67	2.77	14.6	20.7	34.8
	0.9	22.0	9.87	5.59	53.2	55.2	70.2
3	0.7	4.90	3.12	2.44	9.3	14.7	26.1
	0.8	6.89	3.84	2.75	15.5	21.2	34.7
	0.9	18.9	8.20	4.50	45.3	46.4	59.0

$s = 5$, buffer space: 1 page

Table III. Effects of Using Additional Buffer Space During Insertions and Expansions

Buffer pages	Average number of disk accesses		
	Insertion	Expansion	Total
1	3.27	2.86	6.13
2	2.52	1.70	4.22
3	2.28	1.32	3.60
4	2.16	1.15	3.31
5	2.11	1.08	3.19
6	2.08	1.00	3.08

$b = 10$, $s = 5$, $\alpha' = 0.8$, $n_0 = 2$

Table IV. Average Total Insertions Cost when Using 3 Buffer Pages During Insertions and Expansions

n_0	α'	Insertion cost (total) (disk accesses)		
		$b = 10$	$b = 20$	$b = 40$
2	0.7	3.14	2.52	2.25
	0.8	3.60	2.62	2.26
	0.9	9.29	4.64	3.04
3	0.7	3.57	2.73	2.33
	0.8	4.00	2.79	2.34
	0.9	7.92	4.07	2.78

$s = 5$

current method is consistently slower, in particular, for small pages and high storage utilization. Based on the characteristics of linear probing this was to be expected.

If only one buffer page is used, the (total) cost of inserting a record is higher for the current method than for the one in [1, 3]. However, the number of disk accesses can be significantly reduced by using more buffer space and having each access transfer several consecutive pages. This can be done both when inserting a record and when expanding the file. Table III shows how the average number of disk accesses required by these two operations are affected by the amount of buffer space used. The figures are averages over a full expansion. The cost of file expansions is given as accesses per record; that is, the cost of an expansion operation is amortized over the set of records inserted between expansion operations ($0.8 \times 20 = 16$ records in Table III).

The improvement in speed is considerable: using 3 buffer pages instead of 1 page reduces the average insertion cost by one access, and the expansion cost to less than half the original cost.

Table IV gives the total insertion costs for different parameter combinations when 3 buffer pages are used. They are lower than those of [1, 3], except when the load factor is 0.9. Using 3 partial expansions instead of 2 results in higher total insertion costs (except for page size 40 and storage utilization 0.9), and it reduces the retrieval costs only slightly.

In general we can draw the following conclusion: The performance of the current method is better or only slightly worse than that of competing methods when the page size is 20 records or more, and the target storage utilization is at most 0.80–0.85. To speed up insertions and expansions it is essential to use a larger buffer area and transfer several pages whenever accessing the disk.

6. CONCLUSIONS

A new method for handling overflow records in connection with linear hashing has been presented and its performance analyzed by means of simulation. The method is based on linear probing, and does not require a dedicated overflow area or chaining. The expansion sequence was modified from that originally proposed for linear hashing to avoid creating large clusters of full pages. This significantly improved the overall performance.

The algorithms needed to implement the basic file operations are very simple. An address computation algorithm and an algorithm for expanding the file by one page were given. Retrieval, insertion, and deletion can be done in the same way as for traditional linear probing.

The overall performance of the new method is better, or only slightly worse, than that of other variants of linear hashing, for a load factor up to 0.80–0.85 and a page size of 20 records or more. One of the advantages of using linear probing is that all the basic file operations can be speeded up simply by using more buffer space, and having every disk access transfer several consecutive pages. Linear probing also preserves locality of reference, thus (potentially) avoiding long seeks.

ACKNOWLEDGMENT

Constructive comments from the referees led to improvements in the presentation.

REFERENCES

1. LARSON, P.-Å. Linear hashing with partial expansions. In *Proceedings of the 6th Conference on Very Large Data Bases* (Montreal, Canada), ACM, New York, 1980, 224–232.
2. LARSON, P.-Å. Performance analysis of linear hashing with partial expansions. *ACM Trans. Database Syst. 7,* 4 (1982), 566–587.
3. LARSON, P.-Å. A single-file version of linear hashing with partial expansions. In *Proceedings of the 8th Conference on Very Large Data Bases* (Mexico City, Mexico), VLDB Endowment, 1982, 300–309.
4. LARSON, P.-Å. Performance analysis of a single-file version of linear hashing. *Comput. J.* (to appear).
5. LITWIN, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Data Bases* (Montreal, Canada), ACM, New York, 1980, 212–223.
6. MULLIN, J. K. Tightly controlled linear hashing without separate overflow storage. *BIT 21,* 4 (1981), 389–400.
7. RAMAMOHANARAO, K., AND LLOYD, J. K. Dynamic hashing schemes. *Comput. J. 25,* 4 (1981), 478–485.
8. RAMAMOHANARAO, K., AND SACKS-DAVIS, R. Recursive linear hashing. *ACM Trans. Database Syst. 9,* 3 (1984), 369–391.

Received February 1984; revised August 1984; accepted August 1984