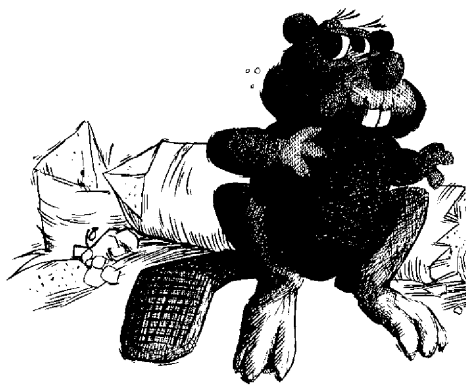
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Linear Hashing with Separators – A Dynamic Hashing Scheme Achieving One-Access Retrieval

Per-Åke Larson

CS-84-23

November, 1984

Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval*

Per-Åke Larson

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Report # CS-84-23

ABSTRACT

A new dynamic hashing scheme is presented. Its most outstanding feature is that any record can be retrieved in (exactly) one disk access. This is achieved by using a small amount of supplemental internal storage which stores enough information to uniquely determine the current location of any record. The amount of internal storage required is small: typically one byte or less for each page of the file. The necessary address computation, insertion and expansion algorithms are presented and the performance is studied by means of simulation. The new method is the first practical method offering one-access retrieval for large dynamic files.

* This work was supported by Natural Sciences and Engineering Research Council of Canada, Grant A2460.

Electronic mail: (UUCP) {allegra,decvax}!watmath!watdaisy!palarson
(CSNET) palarson%watdaisy@waterloo
(ARPA) palarson%watdaisy%watmath@csnet-relay

Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval*

Per-Åke Larson

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Report # CS-84-23

1. Introduction

A new class of file structures, commonly referred to as dynamic hashing schemes or extendible hashing schemes, has been developed over the last few years. These hash-based file structures are designed to accommodate files that grow and shrink dynamically. The most efficient schemes in this class are linear hashing [LI80], improved versions thereof [LA80, RL82] and spiral storage [MA78].

This paper introduces a new dynamic hashing scheme which has the additional advantage that any record can be retrieved in (exactly) one disk access. This is achieved by using a small amount of additional internal storage where enough information is stored to uniquely determine on which page a record is stored. The amount of additional internal storage needed is small; typically one byte or less for each page (bucket) of the file is sufficient. The new method is a combination of two earlier schemes: linear hashing with partial expansions [LA80, LA83b] and external hashing using fixed-length separators [GL82, LK83]. It inherits all the advantages of linear hashing with partial expansions: it can handle dynamically growing and shrinking files, the storage utilization is controlled by the user, and insertions and deletions are quite fast.

The new method can, of course, be used as a fast general-purpose access method, but it appears to be of most interest for applications with tight bounds on the retrieval speed. Among the methods guaranteeing retrieval in one access the new method is, by far, the most space efficient. Extendible hashing [FN79], dynamic hashing [LA78] and virtual hashing [LI78] can all achieve the same retrieval speed, but the amount of internal space needed is much larger. Furthermore, none of them have a user-controllable storage utilization.

The rest of the paper is organized as follows. Section 2 explains the basic ideas of external hashing with fixed-length separators and how it guarantees that any record can be retrieved in one access. Section 3 discusses a modification of linear hashing with partial expansions that uses linear probing to handle overflow

records. Combining these two methods directly gives the new scheme. The necessary address computation, insertion and expansion algorithms are presented in section 4. Performance results, obtained by simulation, are summarized in section 5.

2. External hashing using separators

Gonnet and Larson [GL82] introduced several techniques for speeding up external hashing. The most practical one makes use of fixed-length signatures and separators [LK83]. It is applicable to any external hashing scheme that handles overflow records by open addressing, that is, without using links or pointers. The two most well-known open addressing schemes are linear probing and double hashing [KN73].

Assume that we have an external hash file consisting of m pages (buckets) where each page has a capacity of b records. In addition to the external file an internally stored *separator table* is required. The table contains m separators each one of length k bits. Separator i, $i = 0,1,..., m-1$, corresponds to page i in the file.

We assume that given a record with key K, we can compute its *probe sequence*, $H(K) = (h_1(K), h_2(K),..., h_m(K))$. The probe sequence is uniquely determined by the key, and defines the order in which pages will be checked when inserting or retrieving the record. For each record we will also need a *signature sequence*, $S(K) = (s_1(K), s_2(K),..., s_m(K))$. Each signature is a k-bit integer. The signature sequence is also uniquely determined by the key of the record. When the record with key K probes page $h_i(K)$, signature $s_i(K)$ is used, $i = 1,2,..., m$. Implementation of $H(K)$ and $S(K)$ will be discussed further below.

The separator table is used in the following way. Consider a page to which r, $r > b$, records hash. The page can store only b records, so at least $r-b$ records must be forced out, each one trying the next page in its probe sequence. The r records are sorted on their current signatures. Records with low signatures are stored on the page while records with high signatures are forced out. A signature value that uniquely separates the two groups is stored in the separator table. This value is the lowest signature occurring among the records forced out. However, it may not be possible to find a separator that gives exactly the partitioning $(b, r-b)$ because the length of signatures (and separators) is limited to k bits. If so, we try $(b-1, r-b+1)$, $(b-2, r-b+2)$,..., $(0,r)$ until we find a partitioning where the highest signature in the first group is different from the lowest signature in the second group. This only means that a page having overflow records may actually contain less than b records.

Example: Consider a page being probed by 5 records with signatures 0001, 0011, 0100, 0100, and 1000, respectively. If the page size is 4 we obtain a perfect partitioning: the first 4 records are stored on the page and the separator is 1000. However, if the page size is 3, a perfect partitioning cannot be obtained. The best one is (2, 3), that is, the two records with signatures 0001 and 0011 are stored on the page and the separator is 0100.

Insertion of a record may require relocation of other records and updating of

separators. If the record to be inserted probes a page having a separator greater than the (current) signature of the record, the record "belongs" to that page and it must be inserted on that page. If the page is completely filled already, one or more records (those with the highest signatures) must be forced out and the separator updated accordingly. The records forced out must then be reinserted into some other pages, which may in turn force out other records, etc. Eventually this cascading of records will stop and the insertion process terminates.

One final detail: What should the separator of a page that has not yet overflowed be set to? The necessary algorithms will be simpler and more uniform if the initial separators are strictly greater than all signature values. Using k bits the largest possible value is 2^k-1, which is used as the initial separator. The range for signatures must consequently be restricted to 0,1, . . . , 2^k-2.

It is easily seen that a record can be retrieved in one disk access. Given a search key, we follow its probe sequence, at each step comparing the current signature with the appropriate separator from the in-core separator table. As soon as we encounter a separator that is strictly greater than the signature, the probing process stops. The corresponding page is read in and its records checked. If the desired record is not found on that page, it does not exist in the file. There is no need to check any other page. In essence, external probing has been replaced by internal probing and a final read is done only when the address of the desired record has been uniquely determined.

One consequence of using separators must be pointed out: a storage utilization of 100% cannot be achieved, not even theoretically. The usable capacity of a page depends on the value of its separator and there is no guarantee that a page can be completely filled. There is an upper bound on the achievable storage utilization, which depends on the separator length, the page size and the way probe sequences are generated. This has been analysed under the assumption of random probing (a theoretical approximation of double hashing) [GL82]. However, the upper bound is, per se, of limited practical consequence. Normally we will have to set the target storage utilization significantly lower. The reason for this is simple: when the storage utilization increases, the cost of inserting a record also increases (more records will be relocated). Near the bound the insertion costs increase dramatically [LA83a]. As we will see later on, in practice a storage utilization of up to 80% is a realistic goal.

3. Linear hashing with open addressing

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 original idea is due to Litwin [LI80]. Linear hashing with partial expansions, developed by Larson [LA80], is a generalization of linear hashing that achieves better performance. A slightly different generalization was introduced by Ramamohanarao and Lloyd [RL82]. It is assumed that the reader is familiar with the basic ideas of linear hashing with partial expansion.

Linear hashing, and its variants, require some method for handling overflow records. Several methods based on chaining have been proposed for this. However, overflow chaining cannot be combined with the signature-separator

approach explained in the previous section. A method satisfying the following two requirements is needed:

1. The full probe sequence of a record must be computable without accessing the file.
2. The maximum number of different probe sequences emanating from a page must be small.

The first requirement is imposed by the signature-separator approach. When using separators, external probing is replaced by internal probing. If computation of the next address in the probe sequence requires access to the externally stored file, as in chaining, nothing has been gained.

The second requirement is dictated by linear hashing. An expansion of the file by one page involves locating all records hashing to a (predetermined) set of existing pages and relocating some of them to the new page. Each possible probe sequence emanating from a page participating in the expansion must be checked. A method generating a large number of probe sequences, double hashing, for example, would make this too costly.

Linear probing satisfies both the above requirements. Let h be the home address of a record. Linear probing is the technique that uses probe sequences of the form h, $h+1$, $h+2$,... It is simple and widely used in practice. However, in order to combine it with linear hashing some slight modifications are necessary. Using linear probing to handle overflow records in connection with linear hashing was first suggested in [LA83b], where the necessary modifications were also discussed.

When adding a new page at the end of the file all probe sequences should be extended to include the new page. It is desirable that this can be done without actually relocating records. This can be achieved by modifying linear probing so as not to wrap around when reaching the (currently) last page of the file. If there are records overflowing from the last page in the current address space, they are allowed to go into the first unused page(s) at the end of the file. The page has been taken into use by receiving overflow records before receiving any "native" records. When the file is expanded the next time the page will be within the address space of the file.

To improve performance it is also necessary to change the expansion sequence. Linear hashing with partial expansion extends the file by increasing the size of one group of pages. The expansion sequence originally proposed was group 0, group 1, etc. This particular sequence has a very serious drawback when overflow records are handled by linear probing. As illustrated in Fig. 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 overflowing pages are more likely to form. Such islands will slow down insertions (more records must be relocated) and expansions (more pages must be checked).

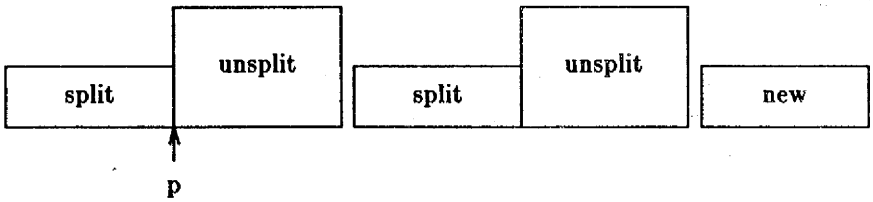


Figure 1: Load distribution of linear hashing with two partial expansions

The risk of creating long islands of overflowing pages can be reduced by changing the expansion sequence in such a way that the split pages (with a lower load factor) are spread more evenly over the file. The modified expansion sequence explained below is a straightforward implementation of this idea.

Consider a file consisting of N groups of pages. The expansion sequence 0,1,..., $N-1$ uses a step length of one. To spread out pages with a low load factor more evenly we can increase the step length and make a number of sweeps over the groups. If the step length is s, $s \geq 1$, the first sweep would expand groups 0, s, $2s$,..., the second sweep groups 1, $s+1$, $2s+1$,..., and the last sweep would be $s-1$, $2s-1$, $3s-1$,... One further modification will slightly improve the performance: having each sweep go backwards instead of forwards. The first sweep would then be $N-1$, $N-1-s$, $N-1-2s$,..., and correspondingly for the other sweeps. Going backwards is not more expensive than going forwards, and it has some beneficial effects on the performance [LA83b].

Example: Consider a file consisting of 10 groups, number 0,1,..., 9, and assume a step length of 3. The first sweep expands groups 9, 6, 3, 0, in that order. The second sweep expands 8, 5, 2 and the third, and last, sweep expands 7, 4, 1.

Linear hashing also requires a a set of rules for determining *when* to expand (or contract) the file by one page. There are many alternatives , but we will here consider only the rule of constant storage utilization. According to this rule the file is expanded whenever the overall load factor rises above a user-selected threshold α, $0<\alpha<1$. No separate overflow area is used so this rule will lead to a storage utilization that, for all practical purposes, is constant and equal to α.

4. Algorithms

In this section algorithms needed to implement the basic file operations are presented and discussed. Three algorithms will be given in detail: address computation, insertion of a record and file expansion. Deletions will be discussed briefly, but a full algorithm is not given.

The details of the file structure are determined by a number of parameters and the current state of the file is defined by a number of state variables:

Parameters

b	page size in number of records
N	original number of groups
n_0	number of partial expansion per full expansion
s	step length used in computing the expansion sequence
α	desired storage utilization
k	separator length in bits

State variables

cpx	current partial expansion. Initial value: $cpx = 1$.
sw	current sweep, $1 \leq sw \leq s$. Initial value: $sw = 1$.
p	next group to be expanded, $0 \leq p < N2^c$ where c = (*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: $maxadr = n_0\,N-1$.

Computation of the current location of a record is done in two steps: first its home address is computed and then its actual location is determined by comparing signatures and separators. The computation of the home address has been written as separate routine. It is also needed in the expanson algorithm.

The routine for computing the home address makes use of two hashing functions. The first one, denoted by h, is a traditional hashing function returning values in the range $[0, n_0\,N-1]$. It 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, $D(K) = (d_1(K), d_2(K),...)$, where the values d_i are independent and uniformly distributed in [0,1). The value $d_i(K)$ is used for determining whether to relocate the record with key K to a newly created 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 partial expansion) a fraction of $1/(n+1)$ of the records should be relocated to the new part of the file. This is achieved by relocating a record if $d_i(k) \leq 1/(n+1)$, otherwise not. The group size during the ith partial expansion is $n = n_0 + (i-1)$ **mod** n_0. The 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 of the record is supplied as the seed.

```
procedure home_address ( K : key): integer ;
begin
    ha, fsz, ngrps, i, lc, 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
            lc : = ngrps - 1 - (ha mod ngrps) ;
            swp : = lc mod s ;
            swpl : = ngrps div s ;
            fsw : = swp × swpl + min (swp, ngrps mod s) ;
            npg : = fsz + fsw + (lc divs )+ 1 ;
            if npg ≤ maxadr then ha : = npg ;
        end ;
        fsz : = fsz + ngrps ;
        if (i) mod n0 = 0 then ngrps = 2 × ngrps ;
    end ;
    return (ha) ;
end {home_address} ;
```

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

The actual location of a record depends on its home address, its signature sequence and the separators in the separator table. An algorithm for computing the current address of a record with key K is given below. The algorithm makes use of two functions with the following declarations:

procedure signature (K : **key** ; i : **integer**) : **bit** (k)
procedure separator (j : **integer**) : **bit** (k).

The first function returns the signature of key K to be used when probing the ith page, i ≥ 1, in its probe sequence. The separator table grows and shrinks dynamically. It may be implemented in several ways. The second function returns the separator of page j, j ≥ 0.

```
procedure current_address (K : key) : integer ;
begin
    ha, i, cp : integer ;
    ha : = home_address (K) ;
    i : = 1 ;
    for cp : = ha to lstpg do
        if signature (K, i) < separator (cp)
        then return (cp)
        else i : = i+1 ;
end ;
```

Retrieval is extremely simple. Given a search key K the address is computed using the algorithm above. The corresponding page is read in and the records on the page are checked. If the desired record is not found on the page, it does not exist in the file.

Insertion of a record may involve relocation of other records. If the record to be inserted happens to hit a full page, some of the records stored on that page will be forced out. They must be inserted into the next page, which, in turn, may force out other records, etc. Eventually this cascading of records will stop and the insertion terminates.

An insertion algorithm is given below. Records waiting to be inserted are assumed to be kept in an area called the record pool. The structure of a record entry, of an external page and of an entry in the record pool is specified below in a Pascal-like notation.

```
record_slot = record
                status : (empty, full) ;
                key ;   {record key}
                info ;  {additional fields}
              end ;

page = array 1 .. b of record_slot ;

pool_entry = record
                nprb : integer;      {next page to try}
                sign : bit (k);      {signature when probing page nprb}
                ha :  integer;       {home address}
                rcrd :  record_slot; {the actual record}
             end ;
```

The actual implementation of the record pool is left unspecified. However, the algorithm below assumes that extraction of the first record from the pool always gives the one with the lowest value of the fields nprb and sign. The algorithm also makes use of two procedures:

```
procedure  add_to_pool (nprb : integer ; sign : bit (k) ;
                                ha : integer ; rcrd : record_slot)
procedure set_separator (j : integer ; value : bit (k))
```

The first procedure adds a record to the record pool and the second one sets the separator of page j to value.

The logic of the insertion algorithm is straightforward. It consists of two nested loops: an inner loop that attempts to insert every record in the record pool into the current page and an outer loop that advances forward one page at a time in the file. The process is started from the page into which the new record is to be inserted, and terminates as soon as there are no more records in the record pool.

The procedure process-current-record called by insert takes the current record (stored in cr) and attempts to store it on the current page (in buffer). If its signature is greater than the separator of the page, it is returned to the pool and will eventually proceed to the next page. Otherwise the record logically belongs to the current page. If so, there are two cases to consider: whether the page is full or not. If the page is not full, the record is inserted into an empty slot. If the page is full, a number of records are forced out and placed in the record pool. The separator is updated accordingly. Once there is room on the page, we try to store the current record again, taking into account the fact that the separator has changed.

```
procedure insert (R) ;

begin
    buffer : page ;  {holds the page being modified}
    cr : pool_entry ; {holds the record being processed}
    {set up the structure for the record pool here}
    poolcnt, cp, nr : integer ;
    maxsign : bit (k) ;
    done : boolean ;

    {copy the new record into cr}
    cr.rcrd.key : = R.key ;
    cr.rcrd.info : = R.info ;
    cr.rcrd.status : = full ;
    cr.ha : = home_address (R.key) ;
    cr.nprb : = current_address (R.key) ;
    cr.sign : = signature (R.key, cr.nprb - cr.ha + 1) ;

    poolcnt : = 0 ;
    done : = false ;

    {consider one page at a time}
    repeat
        cp : = cr.nprb ;
        read page cp into buffer ;
        nr : = {no. of records stored on the page read in} ;
        repeat
            process_current_record (cp, nr, poolcnt, cr, buffer) ;

            {cr is now empty, get the next one from the record pool}
            if poolcnt = 0 then begin
                done : = true ; exitloop ;
            end
            else begin
                move the first record from the record pool into cr ;
                poolcnt : = poolcnt - 1 ;
            end ;
        do until cr.nprb > cp ;

        write buffer into page cp ;
        it cr.nprb  > lstpg then begin
            lstpg : = lstpg + 1 ;
            expand the file space and the separator table up to lstpg ;
            set_separator (lstpg, 2^k -1) ;
        end ;
    do until done ;

end {insert} ;
```

```
procedure process_current_record (cp, nr, poolcnt : integer ;
                                  cr : pool_entry ; buffer : page) ;
    {a highly specialized procedure that takes the record in cr and tries
    to store it on the page currently in buffer}
    begin
        sg, maxsign : bit (k) ;
        haddr, i : integer ;

        if cr.sign ≥ separator (cp) then begin
            sg : = signature (cr.rcrd.key, cr.nprb - cr.ha + 2) ;
            add_to_pool (cr.nprb + 1, sg, cr.ha, cr.rcrd) ;
            poolcnt : = poolcnt + 1 ;
        end
        else if nr < b then begin
            insert the record in cr into any empty slot in buffer ;
            nr : = nr + 1 ;
        end
        else begin
            {the page is full, make room by forcing out
            the records with the highest signature}
            maxsign : = {highest signature among the records in buffer} ;

            for i : = 1 to b do
                if buffer [i].status = full then begin
                    haddr : = home_address (buffer [i].key) ;
                    if signature (buffer [i].key, cp - haddr + 1)
                       = maxsign then begin
                          sg : = signature (buffer [i].key, cp - haddr + 2) ;
                          add_to_pool (cp + 1, sg, haddr, buffer [i]) ;
                          poolcnt : = poolcnt + 1 ;
                          nr : = nr - 1 ;
                    end ;
                end ;

            set_separator (cp, maxsign) ;
            {now there is room on the page, try again}

            if cr.sign < maxsign then begin
                insert the record in cr into any empty slot in buffer ;
                nr : = nr + 1 ;
            end
            else begin
                sg : = signature (cr.rcrd.key, cp - cr.ha + 2) ;
                add_to_pool (cp+1, sg, cr.ha, cr.rcrd) ;
                poolcnt : = poolcnt + 1 ;
            end ;
        end
    end {process_current_record} ;
```

Insertion of a record increases the overall load factor. When the load factor exceeds the user-selected threshold α the file space is expanded. This is done by adding a new page at the end of the file and relocating some records to the new page. The separator table must also be expanded. When records are moved to the new page, space will be freed up in the old part of the file. This means that it may be possible to move some overflow records back to, or at least closer to, their home pages. An expansion algorithm that achieves this is given below.

The algorithm begins by updating the state variables and then the necessary record relocation is done. Consider a page participating in the 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 locate these records page pg is first checked, then pg + 1, etc., up to and including the first page that has not overflowed. A page that has not overflowed has a separator of 2^k-1, (the maximum value). The algorithm scans over this area twice. The first scan collects every record that is not stored in its home page. The collected records are temporarily stored in an area called the record pool until reinserted during the second scan over the area. The set of collected records will include all those that are to be moved to the new page and also all overflow records that, possibly, will be moved closer to their home pages. To avoid writing, pages are not modified during the first scan. All the separators covering the area scanned are reset to 2^k-1.

The second scan goes over the same area as the first scan, reinserting the overflow records collected during the first scan. A page is read into the buffer and first "cleaned", that is, every slot containing an overflow record is declared empty. Then every record remaining in the record pool whose home address is less than or equal to the address of the current page is inserted into the page. If they cannot all be stored on the page, those with the highest signatures are returned to the record pool. This continues until the whole area has been covered.

The above proess is repeated for every page participating in the expansion. The records remaining in the record pool are those that are to be moved to the new page, that is, their home address equals *maxadr*. They are then inserted in the last part of the algorithm.

```
procedure expand (p : integer) ;

begin

    buffer : page ;   {holds the page being processed}
    cr : pool_entry ;  {holds the record being processed}
    {set up the structure for the record pool here}
    i, np, lvl, ngr, pg, cp, lmdf,
    hadr, poolcnt, nr, nprb : integer ;
    sg, oldsep, msign : bit (k) ;
    done : boolean ;

    msign : = 2^k - 1
    gr : = p ;
    np : = n_0 + (cpx - 1)  mod n_0
    lvl : = (cpx - 1) div n_0 ;
    ngr : = N × 2^lvl ;

    {update state variables}
    maxadr : = maxadr + 1 ;
    p : = p - s ;
    if p < 0 then begin
        sw : = sw + 1 ; p : = ngr - sw ;
        if sw > s then begin
            cpx : = cpx + 1 ;
            sw : = 1 ; p : = ngr - 1 ;
            if (cpx - 1) mod n_0 = 0 then
                p : = 2 × ngr - 1 ;
        end ;
    end ;

    poolcnt : = 0 ;
    for i : = to np do begin
        pg : = gr + (i-1) × ngr ;

        {collect all records to be relocated
        and store them in the record pool}

        cp : = pg - 1 ; lmdf : = pg - 1 ;
        repeat
            cp : = cp + 1 ;
            read page cp into buffer ;
            for j : = 1 to b do
                if buffer [j].status = full then begin
                    hadr : = home_address (buffer [j].key) ;
                    if hadr ≠ cp then begin
                        lmdf : = cp ;
                        nprb : = max (pg, hadr) ;
```

```
                    sg : = signature (buffer[j].key, nprb-hadr + 1) ;
                    add_to_pool (nprb, sg, hadr, buffer [j]) ;
                    poolcnt : = poolcnt + 1 ;
                end ;
            end ;
        oldsep : = separator (cp) ;
        set_separator (cp, msign) ;
    do until oldsep = msign ;

{reinsert all records whose home address ≤ lmdf}
    done : = false ;
    if poolcnt > 0 then
        if the first record in the record pool has ha ≤ lmdf
        then begin move it into cr ; poolcnt : = poolcnt - 1 ; end ;
    else done = true ;
    for cp : = pg to lmdf do begin
        read page cp into buffer ;
        nr : = 0 ;
        for j : = 1 to b do
            if buffer [j].status = full then begin
                hadr : = home_address [buffer [j].key) ;
                if hadr ≠ cp then
                    buffer [j].status = empty
                else nr : = nr + 1 ;
            end ;

        while cr.nprb = cp and not done do begin

            process_current_record (cp, nr, poolcnt, cr, buffer) ;

            {get next record}
            if poolcnt > 0 then
                if the first record in the record pool has ha ≤ lmdf
                then begin move it into cr ; poolcnt : = poolcnt - 1 ;
                end ;
            else done = true ;
        end ;

        write buffer into page cp ;
    end ;

end ;
```

```
{all records remaining in the record pool have home address = maxadr}

    cp : = maxadr ;
    if poolcnt > 0 then begin
        move first record in the record pool into cr ;
        poolcnt : = poolcnt + 1 ;
        done : = false ;
    end
    else done : = true ;

    repeat

        if cp > lstpg then begin
            lstpg : = lstpg + 1 ;
            expand the file space and the separator table up to lstpg ;
            set_separator (lstpg, msign) ;
        end ;

        if not done then begin
            read page cp into buffer ;
            nr : = {no. of records on the page just read in} ;

            while cr.nprb = cp do begin
                process_current_record (cp, nr, poolcnt, cr, buffer) ;
                if poolcnt > 0 then
                    if the first record in the record pool has ha < lmdf
                        then begin move it into cr ;
                        poolcnt : = poolcnt + 1 ;
                        end ;
                    else begin
                        done : = true ; exitloop ;
                    end ;
                end ;
                write buffer into page cp ;
            end
            cp : = cp + 1 ;
        do until done ;
    end {expand} ;
```

Both the expansion algorithm and the insertion algorithm read and write one page of a time. However, they can both be speeded up significantly by using more buffer space, transferring several consecutive pages between main memory and disk whenever performing a read or write operation. This reduces the number of file accesses at the cost of more main memory space. When accessing the old part of the file during an expansion, we can find out from the separator table exactly which pages will be affected and must be read in. When expanding a group from n to $n+1$ pages there are n islands of consecutive pages to check. If sufficient buffer space is available an entire island can be brought into memory

in the same read operation, the necessary record rearrangement done and the island written out in one write operation. (This is not possible in the last part of the algorithm where records are inserted into the new page.) The insertion algorithm can also be speeded up by using more buffer space, but not to the same extent as the expansion algorithm. When inserting a record we cannot, based on the separator table alone, predict how many pages will be affected. However, we can have every physical read and write operation transfer some fixed number of consecutive pages. The effects of multi-page reads and writes is studied in the next section.

Deletion of a record will free up one slot on a page. If that page has overflowed (its separator is less than $2^k - 1$) it may now be possible to return some of the overflow records to the page. If so, the separator must be updated accordingly. All the overflow records from the page must be checked to find those with the lowest signatures. This means that every page starting from the page where the record was deleted up to and including the first page that has not overflowed (its separator equals $2^k - 1$) must be checked. In essence a deletion necessitates the same type of local reorganization as a file expansion. The only difference is that only one island of full pages is affected and need to be reorganized. The reorganization part of the expansion algorithm can easily be modified to handle deletions.

5. Performance

In contrast to most hashing schemes, the new method has a fixed retrieval speed and the storage utilization can be selected by the user (within certain limits). The main variable cost of the new method is the total insertion cost. It has two components: the cost of inserting a record and the cost of expanding the file. The cost will here be measured in number of disk accesses. Internal processing is ignored. Another variable of interest is the amount of storage space needed for the record pool used when expanding the file by one page. The performance is affected by the following file parameters: page size, storage utilization, number of partial expansions, separator length, step length and the amount of buffer space used during insertions and expansions. The performance results presented in this section were obtained by simulation.

Let us first study the effects of varying the step length used in the computation of the expansion sequence. Figure 2 shows the development of the insertion costs over a full expansion. The file parameters are: page size 20 records, storage utilization 0.8, separator length 8 bits, 2 partial expansions and buffer space 1 page. In figure 2(a) the step length is 2 and in Figure 2(b) it is 5. The solid curve includes the cost of file expansions. The results plotted are averages from 100 simulated file loadings.

The graphs clearly show the cyclic behaviour. Two basic patterns are superimposed: longer cycles extending over a partial expansion and shorter cycles extending over a sweep. Each "hump" in the graphs correspond to one sweep. During a sweep the total insertion costs first increase slowly and then drop rapidly close to the end of the sweep.

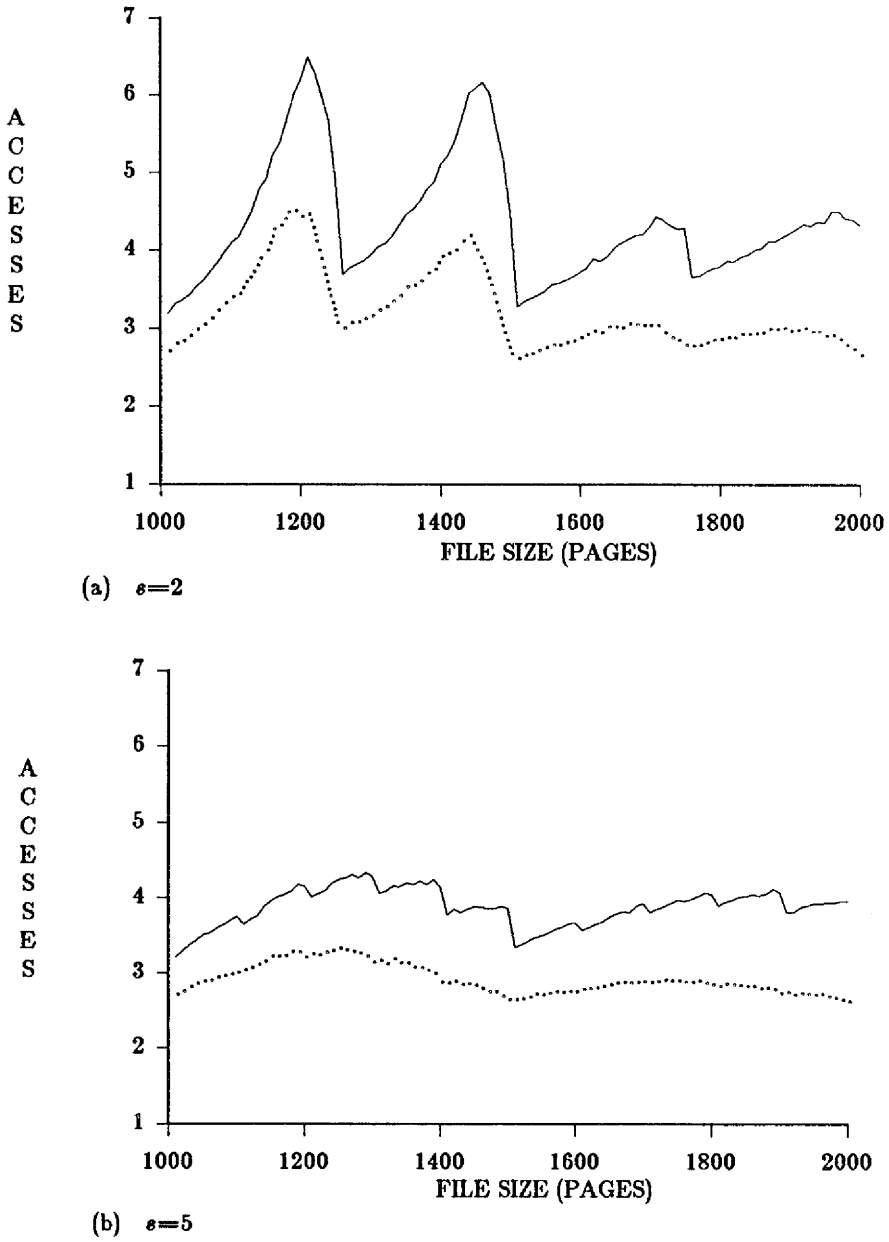


Figure 2: Total insertion costs (including expansions) and insertions costs.

As seen in Figure 2, increasing the step length from 2 to 5 substantially lowered the insertion costs. However, if the step length is increased even further the total insertion costs will start increasing again. This is evident from Table 1, which shows the overall effect of increasing the step length. The figures are averages over a full expansion. A step length of 4 or 5 appears to be optimal. Experiments with other parameter combinations confirmed this conclusion.

Step length	No. of accesses required for Insertion	Expansion	Total
2	3.21	1.16	4.37
3	2.94	0.99	3.93
4	2.90	0.97	3.87
5	2.91	0.97	3.88
6	2.93	0.97	3.90
8	2.98	0.99	3.97
10	3.05	1.02	4.07

$b=20$, $n_0 = 2$, bufferspace: 1 page, $\alpha=0.8$, $k=8$

Table 1: Effect on average insertion costs of increasing the step length.

Tables 2 and 3 show the overall average insertion costs for different combinations of page size, storage utilization and number of partial expansions. In Table 2 the separator length is 5 bits and in Table 3 it is 8 bits. The step length used is 5 in all cases and the buffer space is 1 page. The figures in the first three columns are average number of disk accesses. The record pool size is measured in number of records.

Take, for example, the combination page size 10, storage utilization 0.70, 2 partial expansions and separator length 5. For this particular situation Table 1 indicates that inserting a record requires on average 2.60 accesses and that the added overhead due to file expansions is on average 1.69 accesses per record inserted, giving a total cost, on average, of 4.29 accesses to insert a record. Furthermore, the maximum number of records in the record pool during a single expansion step was on average 8.9 records.

n_0	α	Insertion cost			Expansion cost		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.70	2.60	2.26	2.10	1.69	0.71	0.32
	0.75	2.96	2.44	2.19	2.06	0.82	0.35
	0.80	*	2.80	2.37	*	1.06	0.43
	0.85	*	*	*	*	*	*
3	0.70	2.53	2.20	2.05	2.16	0.91	0.41
	0.75	2.86	2.36	2.13	2.64	1.03	0.44
	0.80	*	2.65	2.26	*	1.29	0.51
	0.85	*	*	2.56	*	*	0.70

n_0	α	Total insertion cost			Record pool size		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.70	4.29	2.97	2.42	8.9	13.9	25.4
	0.75	5.02	3.26	2.54	12.1	16.9	28.9
	0.80	*	3.85	2.80	*	23.4	35.0
	0.85	*	*	*	*	*	*
3	0.70	4.69	3.11	2.46	9.1	14.1	25.8
	0.75	5.50	3.39	2.56	12.4	16.7	28.7
	0.80	*	3.94	2.77	*	22.1	33.3
	0.85	*	*	3.26	*	*	44.7

Table 2: Average insertion costs ($k=5$, buffer space 1 page, step length 5).

n_0	α	Insertion cost			Expansion cost		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.70	2.63	2.30	2.13	1.62	0.70	0.32
	0.75	2.98	2.52	2.27	1.90	0.79	0.35
	0.80	3.60	2.91	2.53	2.49	0.97	0.41
	0.85	4.88	3.71	3.01	4.06	1.41	0.54
3	0.70	2.55	2.23	2.07	2.10	0.90	0.40
	0.75	2.87	2.41	2.17	2.47	1.00	0.43
	0.80	3.41	2.74	2.36	3.21	1.21	0.49
	0.85	4.44	3.35	2.73	4.94	1.66	0.63

n_0	α	Total insertion cost			Record pool size		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.70	4.25	3.00	2.45	8.4	13.6	25.1
	0.75	4.88	3.31	2.62	10.8	16.1	28.3
	0.80	6.10	3.88	2.94	15.6	20.7	33.5
	0.85	8.94	5.12	3.55	28.7	32.4	44.3
3	0.70	4.65	3.13	2.48	8.6	13.7	25.7
	0.75	5.33	3.41	2.60	11.0	16.0	28.3
	0.80	6.62	3.94	2.85	15.8	20.1	32.2
	0.85	9.38	5.01	3.36	26.9	28.6	39.7

Table 3: Average insertion costs ($k=8$, buffer space 1 page, step length 5)

When using the new method there are certain restrictions on the storage utilization. The exact limit depends on all file parameters, but the separator length is the most critical one. If the target storage utilization is set too high the file will "wander away". To explain this phenomenon, consider a situation where a new record to be inserted happens to hit a full page. A number of records will be forced out from the full page and they will all proceed to the next page. This may in turn force out even more records, etc. This creates a wave of records flowing towards the end of the file. Normally the wave will disappear quickly, but if it reaches a certain critical mass it will keep on growing, leaving behind it a string of poorly filled or completely empty pages. The set of records probing a page is divided in $2^k - 1$ subsets by the record signatures. If the subset of records corresponding to signature 0 is greater than the page size, the page will be left empty and the wave of records will increase. This is more likely to occur when signatures are short (fewer partitions) and the load factor is high (more records probing a page).

Entries in Table 2 marked by an asterisk indicate parameter combinations for which the file started "wandering away". These results indicate that a signature length of 5 bits is sufficient only up to a storage utilization of 0.75, and 8 bits is sufficient up to 0.85. If the target storage utilization is pushed much over 0.85, the cost of insertions and expansions becomes quite high (regardless of signature length), unless very large pages are used. It seems that in practice, a signature length of 8 bits is sufficient and the most convenient.

As explained earlier insertions and expansions can be speeded up by using more buffer space, transferring several consecutive pages whenever accessing the disk. Table 4 shows how the average number of accesses decreases when the buffer space is increased. The buffer size is measured in the number of file pages it can hold. The additional gain obtained by increasing the buffer size gradually decreases. The row labelled LB gives the lower bounds on the number of accesses. For the particular parameter combination covered by Table 4 it does not seem worthwhile to go beyond 3 pages; the total cost is already within 0.23 accesses of the minimum. This conclusion appears to hold for a wide range of parameter combinations.

Buffer size (pages)	Average no. of accesses		
	Insertion	Expansion	Total
1	2.91	0.97	3.88
2	2.36	0.61	2.97
3	2.16	0.51	2.67
4	2.08	0.47	2.55
5	2.04	0.46	2.50
LB	2.00	0.44	2.44

$b=20, \quad k=8, \quad \alpha=0.8, \quad n_0=2, \quad s=5$

Table 4: Effect on insertion and expansion costs of increasing buffer space

6. Conclusion and open problems

A new dynamic hashing scheme, called linear hashing with separators, has been presented. Its most outstanding feature is that any record can be retrieved in one disk access. To achieve this a small amount of supplemental internal storage is needed. The simulation experiments performed indicate that one byte per page (bucket) is sufficient. Compared with other schemes that can achieve retrieval in one access (dynamic hashing [LA78], extendible hashing [FN79]) the new method has two significant advantages: the amount of internal storage needed is much smaller and predictable, and the storage utilization is user-controllable (within certain limits). One noteworthy feature of the new method is that the number of disk accesses for insertions and expansions can be significantly decreased simply by using more buffer space during these operations. Overall the method is quite simple. It is the first practical method that can achieve one-access retrieval for large dynamic files.

There are a number of open problems and options that need to be explored. The first one is related to the expansion sequence. The one proposed in this paper is a straightforward implementation of the idea of spreading out pages with low load factor more evenly over the file. Are there other sequences that give better performance and/or simpler implementation? Does there exist an expansion sequence which is optimal in some sense?

The second problem is related to the probe sequence. Linear probing was used in this paper. It is well-known that linear probing suffers from secondary clustering. Are there better probe sequences?

A mathematical analysis of the scheme would be very useful. We need to be able to predict with more confidence how various parameters affect the performance, and in particular, when the file is expected to start wandering away. Based on the complexity of previous analyses of linear probing, this is expected to be a rather difficult task.

The idea of using signatures and separators can be combined with a number of other dynamic hashing schemes. Combining it with spiral storage appears

particularly appealing because of the stable (non-cyclic) performance of spiral storage [MA79].

References

[FN79] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R.: Extendible Hashing - a Fast Access Method for Dynamic Files, ACM Trans. Database Syst., 4, 3 (1979), 315-344.

[GL82] Gonnet, G.H. and Larson, P.-Å: External Hashing with Limited Internal Storage, Technical Report CS-82-38, University of Waterloo, 1982.

[KN73] Knuth, D.E.: The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[LA78] Larson, P.-Å.: Dynamic Hashing, BIT, 18, 2 (1978), 184-201.

[LA80] Larson, P.-Å.: Linear Hashing with Partial Expansions, In Proc. 6th Conf. Very Large Databases (Montreal, Canada), ACM, New York, 1980, 224-232.

[LA82] Larson, P.-Å.: Performance Analysis of Linear Hashing with Partial Expansions, ACM Trans. Database Systems, 7, 4 (1982), 566-587.

[LA83a] Larson, P.-Å.: Further Analysis of External Hashing with Fixed-Length Separators, Technical Report CS-83-18, University of Waterloo, 1983.

[LA83b] Larson, P.-Å.: Linear Hashing with Linear Probing, Technical Report CS-83-38, University of Waterloo, 1983 (to appear in ACM Trans. Database Systems).

[LK83] Larson, P.-Å. and Kajla, A.: File Organization: Implementation of a Method Guaranteeing Retrieval in One Access, Comm. of the ACM, 27, 7 (1984), 670-677.

[LI78] Litwin, W.: Virtual Hashing: A Dynamically Changing Hashing, In Proc. 4th Conf. Very Large Databases, (Berlin, West-Germany), 1978, 517-523.

[LI80] Litwin, W.: Linear Hashing: A New Tool for File and Table Addressing. In Proc. 6th Conf. Very Large Databases (Montreal, Canada), 1980, 212-223.

[MA79] Martin, G.N.N.: Spiral Storage: Incrementally Augmentable Hash Addressed Storage. Theory of Computation Rep. 27, University of Warwick, England, 1979.

[RL82] Ramamohanarao, K. and Lloyd, J.K.: Dynamic Hashing Schemes. The Computer J., 25, 4 (1981), 478-485.