UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

File Organization: Implementation of a method guaranteeing retrieval in one access

Per-Ake Larson
Ajay Kajla

CS-83-4

April, 1983

File organization: implementation of a method guaranteeing retrieval in one access.

Per-Ake Larson and Ajay Kajla
Data Structuring Group

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

ABSTRACT

A new file organization guaranteeing retrieval of any record in one access was recently introduced by Gonnet and Larson. This paper gives a detailed description of a test implementation of the new method, including the algorithms used for implementing the hashing and signature functions required by the method. The design has been tested on two existing files and the empirical results agree closely with the theoretical predictions. The method makes use of a small in-core table. As predicted and confirmed by the test results the table can be kept very small: one or less than one bit per record stored in the file is sufficient. A load factor of 80 percent in the file itself can be achieved without difficulty.

Categories and Subject Descriptions:

D.4.3 [Operating Systems]: File Systems Management - *Access Methods*

H.2.2. [Database Management]: Physical Design - *Access Methods*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Hashing, searching, file organization.

File organization: implementation of a method guaranteeing retrieval in one access.

Per-Ake Larson and Ajay Kajla

Data Structuring Group

Department of Computer Science

University of Waterloo

Waterloo, Ontario, Canada

N2L 3G1

1. Introduction

In a recent paper Gonnet and Larson [1] introduced a new file organization that guarantees retrieval of any record in one access. The new method is based on hashing, and makes use of a small in-core table to direct the search. The table contains enough information to uniquely determine on which page a record is stored. The table is of fixed size and typically very small: one or less than one bit per record stored in the file.

This paper is intended as an introduction to the new method and its implementation. The emphasis is on implementation; theoretical results have been reported elsewhere [1,3]. The basic ideas and algorithms for insertion, retrieval and deletion are given in section two. The next two sections contain a detailed description of a test implementation. The hashing and signature functions used are described in section three. The design has been tested on two existing files. In section four the results from these tests are reported and compared with the corresponding theoretical results. In section five the problem of selecting the target load factor is discussed.

2. Basic ideas and algorithms

The new method can briefly be characterized as a modification of hashing (with open addressing), where a small, fixed amount of internal storage is used for directing the search, so that any record can be retrieved in one disk access. A more detailed description of the method is given in this section.

Assume that we have n records that are to be stored in an external file consisting of m pages (buckets), where each page has a capacity of b records. The load factor is denoted by α and defined as $\alpha = n/(mb)$. In addition to the external storage space, an internally stored separator table is required. This table stores m separators, each of length k bits. Separator i, $0 \leq i \leq m-1$, in the table corresponds to page i in the external file. The use of this table will be explained further below.

The method is based on hashing where overflow records are handled by open addressing (no links or pointers are used). The two most well-known open addressing schemes are linear probing and double hashing [2]. 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 of a record defines the order in which the pages will be checked when inserting or retrieving the record. It is uniquely determined by the key, and every probe sequence is a permutation of the set of page addresses $\{0,1, ..., m-1\}$.

For each record we will also need a *signature sequence*, $S(K) = (s_1(K), s_2(K),...,s_m(K))$. Each signature $s_i(K)$ 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 $h_i(K)$ the signature $s_i(K)$ is used, $i = 1,2, ..., m$. Implementation of $H(K)$ and $S(K)$ will be discussed in section three.

The separator table is used in the following way. Consider a page to which r, $r > b$, records hash. Because the page can store only b records, at least $r - b$ of them must become overflow records, each one trying the next page in its probe sequence. The r records are sorted on their current signatures, and records with low signatures are stored on the page, while records with high signatures become overflow records. A signature value that uniquely separates the two

groups is stored in the separator table. The value stored is the lowest signature occuring among the overflow records. Because the length of a signature (and a separator) is limited to k bits, we may not be able to find a separator that gives exactly the partitioning $(b, r-b)$. In that case 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 contain less than b records.

An example may be helpful at this point. Assume that a page is probed by 5 records with signatures 0001, 0011, 0100, 0100 and 1000. If the page size is 4, we can 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 partitioning is (2,3). The two records with signatures 0001 and 0011 are stored on the page and the separator is 0100.

The separators are easily maintained during insertion. If a record probes a page with a separator that is greater than the signature of the record, the record "belongs" to that page. The record is inserted into the page. If the page was completely filled already, this will force out one or more records (those with the highest signatures) and the separator is updated accordingly. Note that this means that a record may be relocated after it has been inserted.

One small problem remains: how do we initialize the separator table or, in other words, how are pages that have not yet overflowed marked? The subsequent algorithms assume that the initial separator values are strictly greater than all signature values. Using k bits the largest value possible is $2^k - 1$, meaning that this value cannot be allowed as a signature. This must be taken into account in the implementation of the signature function.

An algorithm for computing the current page address and the corresponding signature of a record with key K is given below. The separator table is called sep. Note that the procedure returns two values: the page address and the corresponding signature.

```
procedure address (K);

 for i = 1 to m do
  adr: = h_i (K) ;
  sign : = s_i (K) ;
  if sign < sep [adr] then return (adr, sign) ; endif ;
 endloop ;

 {no such record can exist in the file}
  return (-1, -1) ;
end address ;
```

Given the address computation algorithm the retrieval algorithm is trivial. First the address is computed. If (-1,-1) is returned there is no such record in the file. Otherwise the indicted page is read in and the records on the page are checked. If the desired record is not found on that page the algorithm terminates unsuccessfully. At most one disk access is required, provided that the separator table is available in main memory.

The insertion algorithm is more complicated. The complications arise from the fact that insertions may cascade, and it is not known in advance whether or not this process will terminate. By insertions cascading we mean that inserting a record into a (full) page will force out one or more records from the page, each one of these records has to be reinserted into some other page, which in term may force out other records, etc.

The insertion algorithm is given at the end of this section. Records which have been forced out from some page are stored on a stack, and the algorithms keeps inserting records from the stack until it is empty. All the pages that are changed during insertion are kept in a page pool and written out only if the insertion process terminates successfully. The insertion fails if we run out of space for the record stack or (more likely) the page pool. The separator table is updated during insertion, but the old values are saved, and restored if the insertion fails. The main data structures used by the algorithm are defined below in a Pascal-like notation:

```
page-pool : array 1..maxpages of page ;

page : array 1..b of                    {b record slots}
       record
         status : (empty, full);
         key    ;                       {record key}
         info   ;                       {additional fields}
       end ;
page-tbl :
       array 1..maxpages of             {keeps track of what}
       record                           {pages are in page-pool}
           page-no : integer ;          {external page address}
           pntr    : integer ;          {points to a page in page-pool}
           old-sep : bit(k) ;           {saves the old separator}
       end ;

rcrd-stk : array 1..maxstk of
           record
              pg : integer ;            {points to a page and}
              slot : integer ;          {record slot in page-pool}
           end ;

buffer :   record                       {holds the record currently}
             status : (empty, full) ;   {being inserted}
             key ;
             info ;
           end ;
```

One tricky detail of the algorithm may require clarification. To reduce the size of rcrd-stk, only pointers to records forced out from a page are stored on the stack. The actual records remain in their former home pages, until they are picked up, one by one, to be reinserted. Let us call such a record a guest. For each record slot containing a guest there will be an entry in rcrd-stk. In all other respects guests are indistinguishable from "normal" tenants. During processing we may have a situation where several entries on the stack point to the same record slot. This occurs when a record is inserted into a full page where some of the slots still contain guests. Because the guests have the highest signatures they will be forced out, their page addresses and record slot numbers being placed on the stack again. The same entries deeper down in the stack hence become superfluous. Because this occurs very rarely, the algorithm was designed to make processing of superfluous stack entries harmless, instead of checking the whole stack each time a new entry is added. A superfluous stack entry points either to an empty slot (the record has already been processed) or to a record that is already on the correct page. The first case is taken care of when an element is popped from the stack by checking whether the indicated slot actually

contains a record. In the second case the record will be processed, but it will immediately be inserted into the same page again.

Deletion of a record can be performed in two accesses. The page containing the record to be deleted is read in and the record slot is declared empty (or deleted). No separators are changed. Note, however, that this is not a "true" deletion in the sense that it would leave the file in the same state as if the record never had existed in the file. A "true" deletion would involve changing the separator table, possible relocating some records. To find out what changes are necessary, every page in the file may have to be checked. This is obviously too slow so we must resort to the above marking solution.

```
procedure insert (Rcrd) ;
  page-cnt : = 0 ;              {number of pages in page-pool and entries in page-tbl}
  stkp : = 0 ;                  {stack pointer for rcrd-stk}
  done : = false ;
  buffer : = Rcrd ;
  buffer.status : = full ;
  while not done do

    (h, sign) : = address (buffer.key) ;
    if h < 0 then goto failure ;

    {check if page h is in core, if not bring it in}

    cp : = 0 ;
    for i = 1 to page-cnt do
       if page-tbl [i].page-no = h
       then cp : = page-tbl [i].pntr ; exitloop ;
       endif ;
    endloop ;

    if cp = 0
      then
        if page-cnt = maxpages then goto failure ; endif ;
        page-cnt : = page-cnt + 1 ;
        read page h into page-pool [page-cnt] ;
        page-tbl [page-cnt] : = (h, page-cnt, sep [h]) ;
        cp : = page-cnt ;
    endif ;
```

```
rcrd-cnt : = {number of full record slots on page cp} ;

case

(rcrd-cnt < b) :     {there is room on the page}

  t : = {any empty slot on page cp} ;

  page-pool [cp].page [t] : = buffer ;
  page-pool [cp].page [t].status : = full ;

  {get next record from the stack}

  repeat
   if stkp = 0 then done : = true ; exitloop ;

      else
       (r, s) : = rcrd-stk [stkp] ;
       stkp :   = stkp - 1 ;
       buffer : = page-pool [r].page [s] ;
       clear page-pool [r].page [s] ;
       page-pool [r].page [s] : = empty ;
   endif ;
  until buffer.status = full ;

(rcrd-cnt = b) :  {the page is already full}

 maxsgn : =  {the highest signature among the records on page cp} ;

 if maxsgn < sign
   then sep[h] : = sign ;

   else
     for i : = 1 to b do
       (adr, sg) : = address (page-pool[cp].page[i].key) ;
       if sg = maxsgn
       then if stkp > maxstk then goto failure ; endif ;
            stkp : = stkp + 1 ;
            rcrd-stk [stkp] : = (cp, i) ;
       endif ;
     endloop ;
     sep [h] : = maxsgn ;
     if sign < maxsgn
        then
        (cp, s) : = rcrd-stk [stkp] ;
        stkp : = stkp - 1 ;
        exchange buffer and page-pool [cp].page [s] ;
     endif ;
   endif ;
 endcase ;
endloop ;
```

```
    {the insertion succeeded, write out all pages}
    for i : = 1 to page-cnt do
      h : = page-tbl [i].page-no ;
      cp : = page-tbl [i].pntr ;
      write page-pool[cp].page into page h ;
    endloop ;

    return ("done") ;

  failure : {the insertion failed, restore the separator table}
    for i : = 1 to page-cnt do
      h : = page-tbl [i].page-no ;
      sep [h] : = page-tbl [i].old-sep ;
    endloop ;

    return ("failed") ;

  end insert ;
```

3. Implementation of the hashing and signature functions

The overall performance of the new method is largely determined by the performance of the hashing function and signature function used. This section presents an implementation designed for files where the key is a string of ASCII characters. The design has been tested on two existing files, and it was found to work reasonably well.

The key is assumed to be a string (of fixed or variable length) of ASCII characters. The first task is to convert this string to an integer. The conversion algorithm takes five low-order bits from each character, concatenates 6 such bitstrings to a string of 30 bits, and XORs these longer strings together. All blanks are discarded. The algorithm is given below in a Pascal-like notation. AND is the bitwise and - operation, XOR is bitwise exclusive or, and ord is Pascal's ord function. The function returns a 30 bit integer.

```
procedure convert (key, keylng) ;
  key : array 1..keylng of character ;
  keylng : integer ;
begin
  cnt, v, t, i, c : integer ;
  cnt : = v : = t : = 0 ;
  for i : = 1 to keylng do
      c : = ord (key [i]) ;
      if c ≠ ord (' ')
      then c : = AND (c, 37₈) ;
           t : = shiftleft (t, 5) + c ;
           cnt : = cnt + 1 ;
           if cnt = 6
           then v : = XOR (v, t) ; t : = cnt : = 0 ; endif ;
      endif ;
  endloop ;
  return (XOR (v, t)) ;
end convert ;
```

To implement the hashing function H discussed earlier any method that generates a sequence of valid addresses could be used. However, for performance reasons it is recommended that double hashing [2] be used. For double hashing two values must be computed from the (converted) key: an initial address, *fa* and a steplength, *sl*. Given these two values the probe sequence $h_1, h_2, \cdots, h_m$ is computed as

$$h_1 = fa$$
$$h_i = (h_{i-1} + sl) \ \textbf{mod} \ m, \quad 2 \leq i \leq m \ ,$$

where m is the file size. To guarantee that a probe sequence will trace through every valid address, the step length must be relative prime to the file size m. The easiest way to ensure this is to choose m as a prime.

To compute *fa* and *sl* the procedure based on the division - remainder method, and suggested by Knuth [2] was selected. This procedure has been tested on several occasions and usually found to perform well. According to this procedure *fa* and *sl* are computed as

$$fa := \text{ckey} \ \textbf{mod} \ m$$
$$sl := ((\text{ckey} \ \textbf{dlv} \ m) \textbf{mod} \ (m-2)) + 1$$

where ckey is the converted key (see above), and **dlv** means integer division.

The method chosen for computing the sequence of signatures makes use of a random number generator, to which the converted key is supplied as the seed. Again a conservative

approach was taken, and a well-tested mixed congruential generator was selected. Successive random numbers are computed as

$$r_{i+1} = (a * r_i + d) \bmod 2^{32}$$

where r_0 = ckey and the constants are $a = 3141592653$ and $d = 2718281829$. This generator has been extensively tested by Knuth [2].

From each random number generated we need only a few bits for the signature. It is well-known that the low order bits of numbers computed by a generator of the above type are relatively "non-random". To obtain a signature which depends on all 32 bits, it was decided to first divide the number by a suitable constant, c, and extract the required number of bits from the remainder. The choice of the constant is not completely arbitrary. Assume that the remainders are uniformly distributed in [0, c-1]. In order for the numbers obtained by extracting k low order bits to be uniformly distributed, as well, c should be close to 2^i for some sufficently large i. It should not, however, be exactly equal to 2^i, because then the remainder is simply the i low order bits. The constant chosen was $c = 2^{13} - 1 = 8191$, which satisfies the requirements and also happens to be a prime. If signatures of more than 10 bits are needed, which we consider very unlikely, this constant should be increased.

The above procedure gives signatures uniformly distributed in $[0, 2^k - 1]$. However, because all separators are initialized to $2^k - 1$, signatures having the value $2^k - 1$ cannot be allowed. If this value is obtained the signature is set at 0. This will result in 0 occurring as a signature twice as often as any other value. But 0 is the last value to be used as a separator for a page, and consequently it is very unlikely that 0 will ever occur as a separator. This was considered to be a better solution than computing signatures uniformly distributed in $[0, 2^k - 2]$.

Combining all the bits and pieces discussed above we finally arrive at the following implementation of the address-signature algorithm. All undeclared variables are assumed to be unsigned integers.

```
procedure address (key, keylng, m, k) ;
   key : array 1..keylng of character ;
   keylng : integer ;  {key length}
   m : integer ;       {file size, must be prime}
   k : integer ;       {separator length}
begin
   constants  a = 3141592653 ;
              d = 2718281829 ;
              c = 8191 ;

   ckey : = convert (key, keylng) ;

   fa : = ckey mod m ;
   sl : = ((ckey div m) mod (m-2))+ 1 ;

   adr : = fa - sl ;
   r   : = ckey ;

  for i : = 1 to m do
    adr : = (adr + sl) mod m ;
    r   : = (r*a+ d) mod 2^32 ;
    sign: = (r mod c) mod 2^k ;
    lt sign = 2^k-1 then sign : = 0 endif ;

    lt sign < sep [adr] then return (adr, sign) endif ;
  endloop ;
  return (-1, -1) ;
end address ;
```

4. Test results

The impetus to implement the new method came from a large time-sharing installation with a cluster of CPUs running in full load-sharing mode. All CPUs access a common userid file, currently containing information about approximately 11000 users. This file is accessed frequently, especially during a few peak hours. To speed up retrieval each CPU has an in-core index of approximately 7K bytes. The index is fixed in real core, not paged. Even so, retrieval normally requires 2 accesses. By using the new method the size of the in-core table can be decreased to 2-3 K bytes, and retrieval will require only one access.

At the time of writing the design has been tested on two files. Test file A is the userid file discussed above containing 10802 records. The key is a character string of at most 8 characters. Test file B is another userid file of 2129 records from a different installation. The key is at most 12 characters long. Most of the keys in file B actually consist of names in the form: <initials>

<surname>. Test results will be given below and compared with theoretically predicted results, whenever the theoretical results are known.

The first tests concentrated on the performance of the hashing and signature functions. The procedure for computing the first address in the probe sequence (the home page) was tested by checking the observed distribution of the number of records hashing to a page. If the hashing function gives addresses uniformly distributed over the file, the number of pages receiving 0, 1, 2, ... records will follow a Poisson distribution with parameter $\lambda = n/m$. Results from one of the tests of file A and the corresponding theoretical results are plotted in Fig. 1. The parameter is $\lambda = 10802/2621 = 4.121$. The observed and theoretical distributions are in close agreement; a chi-square test does not reveal any significant deviation. Further tests of both file A and B gave similar results.

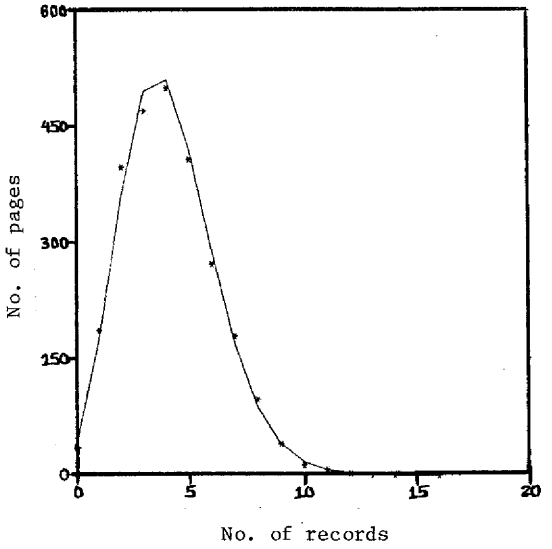


Figure 1: Observed and theoretically predicted number of pages receiving i records, i=0, 1, 2, ... (Test file A, $\lambda = 10802/2621 = 4.121$)

The distribution of the step length was tested in the same way. The results were not quite satisfactory, however. All tests performed on File A showed a statistically significant deviation.

The tests on file B gave slightly better results: the deviation from the theoretical results was generally less, and for some test cases the deviation was not statistically significant on a 1 percent confidence level. A more detailed study revealed that this behaviour was caused by two factors: both files contained some very short keys (1 or 2 characters), and, in addition, file A contained many keys which differed only in the last character. Both these factors will have the same effect: certain step lengths will occur more frequently. However, double hashing is not very sensitive to the distribution of the computed step lengths, as long as there is enough variety. The clustering must be quite severe to have any noticeable effect on the performance. Consequently the implementation of the hashing functions was tentatively accepted, even though there was room for improving the computation of the step length.

The signature function was tested for signature lengths 4,6 and 8 bits, using both file A and B as input. For each key the first 10 signatures were computed and used in the test. Hence each test run gave 10 observed signature distributions, which were checked against the theoretical distribution. If the signature length is k, the signature values 1, 2, ..., $2^k - 2$ should occur with equal frequency, and the value 0 twice as often. Table 1 shows the number of cases for which the computer chi-square value exceeded the 5 percent limit.

	Signature length			Total
	4	6	8	
File A	1/10	2/10	3/10	6/30
File B	1/10	1/10	0/10	2/30

Table 1: Number of cases for which the chi-square value exceeded the 5 percent limit

For file A high chi-square values occur more often than statistically expected, but not alarmingly so. For file B the results are close to what is statistically expected. The performance of the signature function was considered to be acceptable, even though there seemed to be some need for improvement. A closer study of the observed distributions did not reveal any apparent, consistently occurring pattern.

Even if the hashing functions and signature function seemed to work reasonably well when studied in isolation, this does not guarantee good overall performance when combining them into

a complete system. Some theoretical results are available against which the observed overall performance can be tested. Gonnet and Larson [1] derived the probability that a page will contain j records, j=0,1,..., b. This probability is different for pages which have overflowed and pages which have not overflowed. From this model the expected number of pages containing j, j=0, 1,... b. records can be computed and compared with the observed number of pages. Table 2 contains such a comparison for one of several test cases studied. A chi-square test does not indicate any significant deviation. Other test cases gave similar results: none of them showed any statistically significant deviation from the predicted results,

	Pages without overflow rcds		Pages with overflow rcds	
# of records	Pred.	Obser.	Pred.	Obser.
0	19.2	26	0.0	0
1	94.4	94	0.0	0
2	232.1	212	2.1	4
3	380.3	374	23.0	26
4	467.4	496	170.2	171
5	459.6	424	771.7	794
Sum	1653.0	1626	967.0	995
Average	3.55	3.53	4.77	4.76

No. of pages: 2621 No. of records: 10484
Load factor: 0.8 $\chi^2 = 10.4$ df $= 8$
Page size: 5 Separator length: 4
Test file: A

Table 2: Predicted and observed number of pages containing 0, 1, ..., 5 records

Larson [3] has analysed the insertion costs and obtained a formula for the expected number of accesses required for an insertion. A comparison of the observed and expected behaviour during insertion gives a good indication of the overall performance. This was done for both test files, using several different page sizes and separator lengths. The results of two tests of file A are plotted in Fig. 2. The separator length was 4 bits in both tests, the page size is 5 records in Fig. 2(a) and 10 records in Fig. 2(b). The continuous line is the theoretically predicted average and the points scatter around the line are the observed values. Each point represents the average number of accesses for 120 insertions. The observations agree well with the predicted values for these two test cases. Other test runs gave similar results. Note that the number of accesses is twice the

number of pages changed, because each page must be read and written.

As indicated by the above results the overall performance for the two test files was close to what can be achieved by this method. The implementation of the hashing and signature functions was not "custom designed" for these files and we believe that they can be successfully used for other files as well.

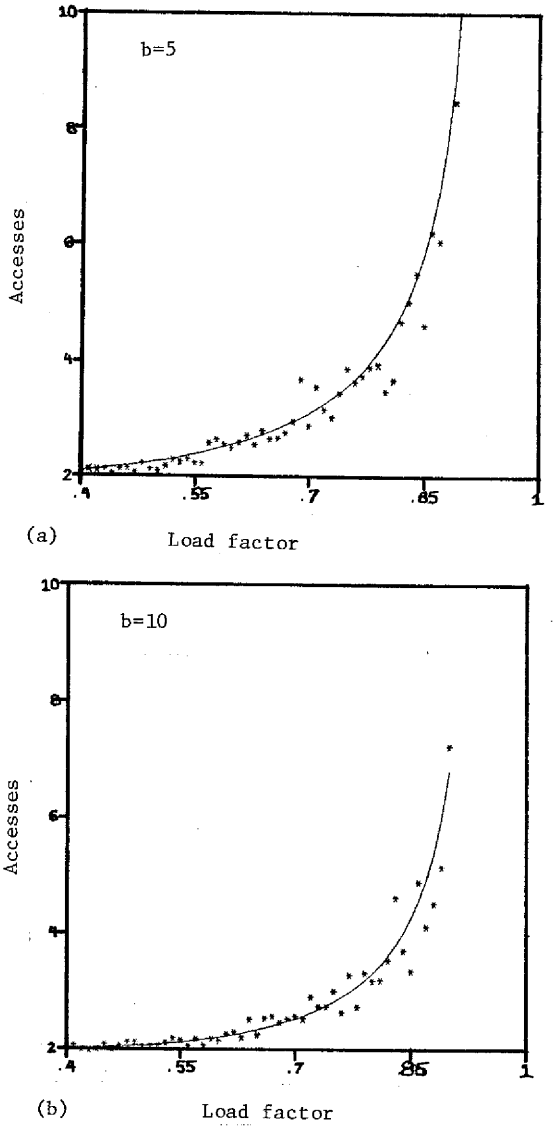


Fig. 2: Predicted and observed average number of accesses required for an insertion (k=4, b=5, 10, test file A)

5. Setting the target load factor

When using the new method a load factor of 100 percent 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 load factor which depends on the page size and the separator length. This was analysed by Gonnet and Larson [1]. The model is asymptotic and assumes that there is no restrictions on the amount of work allowed for an insertion. Numerical results are listed in Table 3.

	Separator length			
Page size	2	4	6	8
5	0.7922	0.9309	0.9793	0.9041
10	0.8238	0.9439	0.9838	0.9955
15	0.8374	0.9493	0.9856	0.9960
20	0.8453	0.9523	0.9866	0.9964
50	0.8637	0.9592	0.9889	0.9971

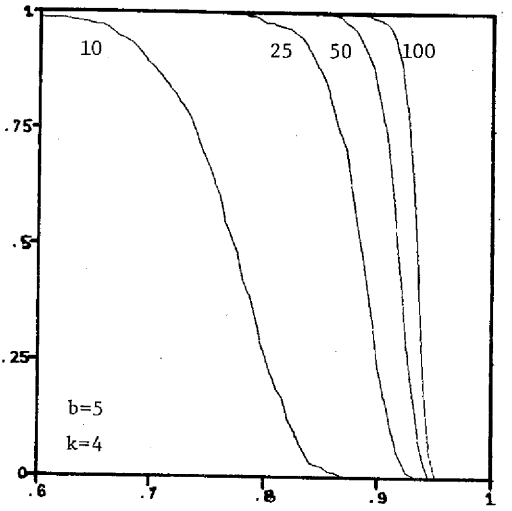
Table 3: Maximum achievable load factor

The crucial assumption is that there is no bound on the amount of work for an insertion. In practice we must set the target load factor lower because, during an insertion, all pages that have been changed are stored in a page pool of limited size. On average the number of pages in the pool will be small (cf Fig. 2), but occasionally an insertion may cascade badly causing many pages to be changed. If we run out of space for the page pool, the insertion fails. The problem is to dimension the page pool and set the target load factor at such a level that the risk of an insertion failing is minimal.

If possible, space for the page pool should be allocated dynamically. On average space for a few pages only is required, but occasionally space for several pages will be needed. If space is allocated statically most of it will be wasted during a "normal" insertion. If the

support dynamic memory allocation, the best alternative seems to be a two-level solution: a small in-core page pool and a larger one on disk. We have not worked out the details of this solution.

Even if space for the page pool is allocated dynamically, some limit on its size must be imposed. Given such a limit, at what load factor level can we expect to exceed the limit for the first time? To gain some insight into this problem we turned to simulation; at the time of writing no theoretical results are available. To this end a series of simulation experiments was run. Loading of a file was simulated and the load factor at which the number of pages in the page pool for the first time exceeded 10, 25, 50 and 100 pages was recorded. In each experiments 500 file loadings were simulated and the results printed. In Fig. 3 the results from two experiments are plotted. The number of pages is 500. In Fig. 3(a) the page size is 5 and the separator length is 4 and in Fig. 3(b) they are 10 and 8, respectively. The graphs should be interpreted as follows (see Fig.3(a)): in approximately 90% of the cases, the limit of 10 pages had not been exceeded by any insertion when the load factor reached 0.7. When reaching a load factor of 0.8, less than 25% of the runs had not exceeded the 10 page limit.



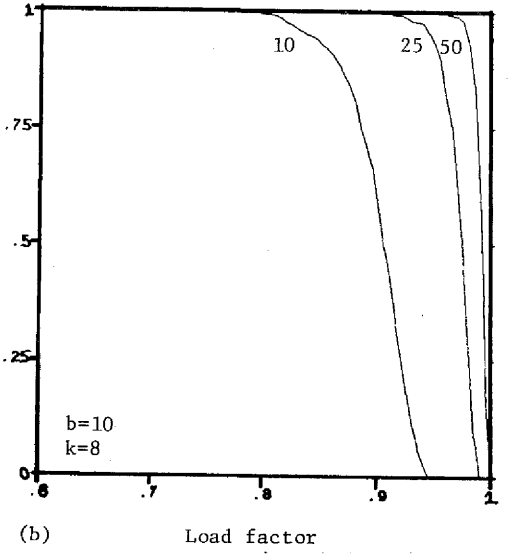


Fig. 3: Load factor at which the number of pages changed exceeds 10, 25, 50 and 100 pages.

What conclusions can we draw from Fig. 3? If the limit on the page pool is set to 25, it is

very unlikely that an insertion will fail before we reach a load factor of 0.8 for a file with b=5 and k=4 (0.9 for a file with b=10 and k=8). Note, however, that at a load factor of 0.8 the average number of pages changed is only 2.18 (1.51 for b=10 and k=8).

In general, it appears that if we set the limit at 25 pages, and the page size is 5 or more and the separator length 4 or more, then we should be able to achieve a load factor of at least 0.8 with very high probability.

6. Conclusion

An implementation of a new file organization guaranteeing retrieval of any record in one access has been presented. The design has been tested on two existing files and found to achieve the overall performance theoretically predicted. The method makes use of an in-core table. As predicted by a theoretical model and confirmed by the tests this table can be kept very small: one or less than one bit per record stored is sufficient. A load factor of 0.8 - 0.85 in the file itself seems to be a realistic goal. The page size has a profound effect on the performance: when the page size is increased, the in-core table can be made smaller and/or a higher load factor can be used.

The algorithms used for implementing the hashing and signature functions were presented in detail. As for all hashing schemes these are of critical importance for the overall performance. For the two test files they were found to perform reasonably well. They were not "custom designed" for the test files, and we believe that they can be successfully used for other files with a similar key structure as well.

Acknowledgement

The authors wish to thank the Department of Computing Services, University of Waterloo for their support, and Ron Hurdal for his interest in the project.

References

[1] Gonnet, G.H. and Larson, P.-A.: External hashing with limited internal storage, CS-82-38. University of Waterloo, 1982 (Extended Abstract published in Proc. ACM Symposium on Principles of Database Systems, (March 1982), Los Angeles, California).

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

[3] Larson, P.-A.: Further analysis of external hashing with fixed-length separators (in preparation).