A SINGLE-FILE VERSION OF LINEAR HASHING WITH PARTIAL EXPANSIONS

Per-Åke Larson

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

Abstract

Linear hashing with partial expansions is a file organization intended for dynamic files. In this paper an improved version of the method is presented. Instead of having a separate overflow area, the storage area for overflow records is incorporated into the main file. Hence, all the records are stored in one file which grows and shrinks according to the number of records stored. By using several overflow chains per page, the retrieval performance is further improved. The expected performance of the new scheme is studied by means of a few examples.

1. INTRODUCTION

Linear hashing with partial expansion is a new file organization primarily intended for files which grow and shrink dynamically [4]. It can accomodate any number of insertions and/or deletions, retaining good storage utilization and retrieval performance without periodic reorganization. It is a generalization of linear virtual hashing developed by Litwin [3]. For the sake of brevity, we will in the sequel use the term "linear hashing" as a generic term covering both methods mentioned above.

Linear hashing, as presented in [4,8], is a two-area technique: in addition to the prime storage area, there is a separate storage area for overflow records. Hence, we have two files which grow and shrink according to the number of records stored. In this paper a new version of linear hashing is presented where the storage area for overflow records is combined with the prime storage area in the same file. Consequently there is only one file that expands and contracts dynamically. The lack of a separate overflow area significantly simplifies implementation of the scheme, especially on systems where a large floating storage pool is not provided. An additional change in the way overflow records are handled reduces the amount of storage required for overflow records and significantly improves the retrieval performance.

Linear virtual hashing is presented in [4] and its generalization, linear hashing with partial expansion, in [8]. A performance analysis of linear hashing with partial expansion is reported in [5]. Similar, but distinct, methods are described in [1, 3, 6, 9]. A version of linear virtual hashing where overflow records are chained in the primary storage area has recently been developed by J. Mullin [10]. His scheme is similar to ours in the sense that all records are stored in one file, but retrieval is slower. K. Karlsson developed a version of linear virtual hashing which handles overflow records by open addressing using no pointers [2]. The performance is rather poor, however, (insertions are particularly slow) and it does not seem possible to achieve significantly better performance by methods based on open addressing.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the basic ideas of linear hashing. In section 3 a modification that speeds up retrieval by using several overflow chains per page is discussed. Thereafter (section 4), the technique for incorporating the overflow area into the main file is presented. The performance of the new method is studied in section 5 by means of a few examples. Some remaining open problems are discussed in the last section.

2. AN OUTLINE OF LINEAR HASHING

Because the new method is a modification of linear hashing, the basic ideas of this technique are first outlined. More detailed presentations and discussions can be found in [4, 5, 7, 8].

The starting point for linear hashing is a traditional hash file where overflow records are handled by separate chaining. In other words, overflow records are stored by linking one or more overflow pages from a separate storage area to an overflowing primary page. Each overflowing primary page has its own, completely separate, chain of overflow pages. The size of an overflow page may be one or larger than one

record.

Linear hashing is a technique for incrementally expanding (and contracting) the primary storage area of a hash file. Linear virtual hashing accomplishes this by splitting the primary pages in an orderly fashion: first page 0, then page 1 etc. Consider a file consisting of N pages, pages 0, 1, ..., N-1. When splitting of page j takes place, j = 0, 1, ..., N-1, the file is extended by one page, page N + j, and approximately half of the records from page j (and its overflow pages, if any) are moved to the new page, cf fig. 1. A pointer p keeps track of which page is the next one to be split. When all the original pages have been split and the size of the primary area has doubled, the pointer is reset to zero and the splitting process starts over again. A doubling of the file is called a full expansion.

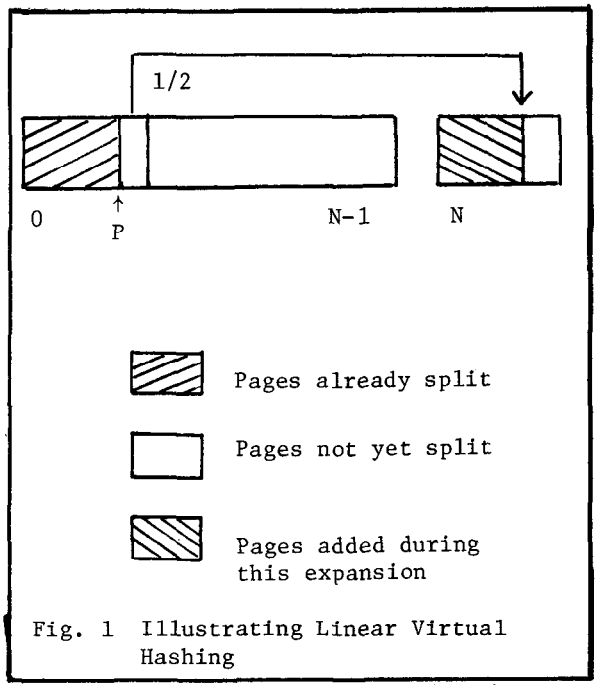

Fig. 1 Illustrating Linear Virtual Hashing

After splitting a page, it should be possible to locate the records which were moved to the new page without having to access the old page; otherwise, nothing would be gained by splitting. The essence of the problem is to devise an algorithm to determine which records are to remain on the old page and which records are to be moved to the new page. There are several solutions to this problem. We cannot here go into details of the solution, the interested reader is referred to [8]. The important point is that, given the key of a record, it is always possible to locate the current "home page" of the record without accessing any other page. If the record is not in the home page, it must be on the overflow chain emanating from the home page.

The development of linear hashing with partial expansions was prompted by the observation that linear virtual hashing creates an extremely uneven distribution of the load on the file. 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, the doubling of the file (a full expansion) is carried out in a series of partial expansions. If two partial expansions are used, the first one increases the file to 1.5 times the original size, while the second one increases it to twice the original size, cf. Fig. 2.

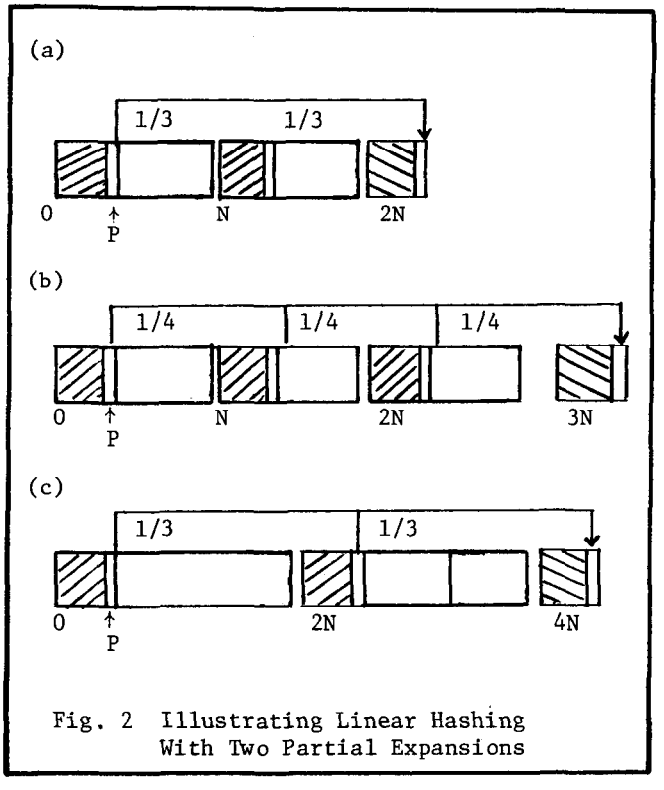

Fig. 2 Illustrating Linear Hashing With Two Partial Expansions

When two partial expansions per full expansions are used, we start from a file of 2N pages, logically subdivided into N pairs (groups of size two) of pages, where the pairs are (j, N + j), j = 0, 1, ..., N-1. When the file is to be expanded, this is done by first expanding group 0, then group 1, etc. by one page. When expanding group j, approximately 1/3 of the records from page j and j + N are moved to a new page j + 2N. When the last pair, (N-1, 2N-1), has been expanded, the file has increased from 2N to 3N pages. Thereafter, the second partial expansion starts, the only difference being that now groups of three pages (j, j + N, j + 2N), j = 0, 1, ..., N-1, are expanded to four pages, cf. Fig. 2b. When the second partial expansion has been completed, the file size has doubled, from 2N to 4N. The next partial expansion reverts to expanding groups of size two, (j, j + 2N), j = 0, 1, ..., 2N.

The one after that again expands groups of size three, etc.

To implement linear hashing with partial expansion an address computation algorithm is required. The algorithm must be able to compute the address of the current home page of a record at any time. Such an algorithm is given in [4]. It works for any number of partial expansions per full expansion.

So far, nothing has been said about controlling the expansion (or contraction) rate. The scheme explained above gives a method for expanding the file by one page, but we also need rules for determining when to expand the file. A set of such rules is called a control function. Several alternatives are possible. cf. [4, 8], but we will here consider only the rule of constant storage utilization. According to this rule, the file is expanded whenever insertion of a record causes the storage utilization of the file to rise above a threshold α, $0 < \alpha < 1$, selected by the user. When computing the storage utilization, the overflow pages in use are also taken into account. This rule is optimal in a certain sense [4]. To implement the rule, it is necessary to keep track of the number of primary pages, the number of overflow pages and the number of records stored in the file.

The results reported in [4, 5] show that increasing the number of partial expansions per full expansion significantly improves the retrieval performance. On the other hand, the costs for inserting a record (which include the costs for expansions) tend to increase. Two partial expansions seem to be a good compromise in many situations.

3. SEVERAL OVERFLOW CHAINS PER PAGE

One of the features of linear hashing causing problems is the tendency to create a relatively larger number of overflow records. Especially towards the end of a partial expansion, the remaining unsplit pages will have many overflow records, which adversely affects performance. To overcome this problem, it was proposed in [4] and [8] to use relatively large overflow pages, thereby reducing the length of each overflow chain.

This solution has one drawback, however. When the overflow page size is larger than one, some of the overflow pages will have empty slots. Because of this internal fragmentation, more overflow space than minimally needed must be allocated. The larger the page size, the worse will the internal fragmentation be. In order to achieve a fixed overall storage utilization, the file must be more heavily loaded. But this in turn creates more overflow records. This less desirable effect is clearly visible in Table 1.

Size of Overflow Page	Load Factor	Overflow Records Per Primary Page
1	0.883	0.79
2	0.892	1.00
3	0.903	1.25
4	0.916	1.56
5	0.993	1.96
10	1.282	10.2
15	2.024	26.5

Primary page size: b = 20
Storage Utilization: α = 85

Required Load Factor and Amount of Overflow Records at the Beginning of an Expansion

TABLE I

An alternative solution is to use several overflow chains per primary page. When an overflow record occurs, it is placed somewhere in the overflow area and linked to one of the chains emanating from its home page. The chain is selected by a hashing function which depends only on the key of the record. To retrieve a record, we first check the records in the home page, and then the records on the appropriate chain.

With this solution, there seems to be no reason for using overflow pages larger than one. We simply chain records. (The physical page size in the overflow area would probably be more than one record, but the point is that the resolution of the chains is one record.) Naturally, the number of overflow records per page is not affected by the number of chains, but by using several chains, each chain is kept shorter. The cost, in terms of storage, of having several chains is quite low: one pointer per chain in each primary page.

Number of Overflow Chains	Expected Search Lengths Successful	Unsuccessful
1	1.125	2.350
2	1.138	1.675
3	1.108	1.450
4	1.094	1.337
5	1.085	1.270
10	1.067	1.135
15	1.062	1.090
20	1.059	1.067
∞	1.050	1.000

Primary page size: $b = 20$

Storage utilization: $\alpha = 0.85$

No. of partial expansions: $n_0 = 2$

Expected Search Lengths For Different Number of Overflow Chains Per Primary Page

TABLE 2

Table 2 shows the effects on the retrieval performance of increasing the number of overflow chains for an example file using linear hashing. Already a moderate number of chains (3 - 5) gives a significant performance improvement. Increasing it beyond five does not seem worthwhile.

The case of an infinite number of chains was included to show that the length of successful searches cannot be forced to one by increasing the number of chains. Note that increasing the number of chains does not change the amount of overflow records. It merely decreases the length of the chains, to the point where any overflow record can be retrieved in one extra access. The length of unsuccessful searches, on the other hand, approaches one because the probability of hitting an empty chain approaches one.

This lower bound on successful searches is of interest also for another reason. Assuming that the physical page size in the overflow area is larger than one record, one can envision an overflow storage scheme that attempts to keep overflow records from the same primary page on the same physical page. For successful searches, the case of an infinite number of chains is also a lower bound on the performance attainable by such schemes.

The page address and the chain number can be computed by the same hashing function. If there are m pages and r chains per page, we need a hashing function that hashes over the range 0 to mr-1. If the value returned from the hashing function is h, compute the page address as $\lfloor h/r \rfloor$ and the chain number as $h \bmod r$. In case of linear hashing, this can be accomplished by having the hashing function h_0, cf. [4], to hash over the range 0 to n_0Nr-1, and computing the (initial) page address and chain number from the value returned by h_0. The chain number (within a page) of a record would not be changed by expansions.

4. OVERFLOW STORAGE IN THE PRIME AREA

To get rid of the separate overflow area, we apply an old idea used in some implementations of hashing: every kth page in the primary storage area is reserved for overflow records. In connection with traditional hash files, an additional separate overflow area may still be required. However, in connection with linear hashing this is not necessary. If all the overflow pages are full, we simply expand the file. This will normally decrease the number of overflow records, thus freeing some overflow space. In any case, after expanding the file by at most k-1 pages, a new overflow page will eventually be created.

The address computation must be modified in order to ensure that we hash directly to non-overflow pages only. Assume that the file consists of m directly addressable pages with (logical) addresses 0, 1, ..., m - 1. Every kth physical page will be an overflow page, requiring a total of $\lfloor m/(k-1) \rfloor$ overflow pages. The "real" pages k-1, 2k-1, 3k-1, ... are designated as overflow pages. The "real" address h' of a directly addressable page with logical address h is computed as $h' = k\lfloor h/(k-1) \rfloor + h \bmod (k-1)$. The complete algorithm for computing the "real" address and chain number of a record is given below.

Let n_0 , $n_0 \geq 1$, denote the number of partial expansions per full expansion and r , $r \geq 1$, the number of overflow chains per (primary) page. Every kth page is an overflow page, $k \geq 2$. We start from a file consisting of n_0N directly addressable pages and $\lfloor n_0N/(k-1) \rfloor$ overflow pages. The address computation algorithm below makes use of an initial hash function $h_0(K)$ which hashes uniformly over $\{0, 1, \ldots, n_0Nr - 1\}$ and a sequence of hashing functions $D(K) = (d_1(K), d_2(K), \ldots)$, where each function $d_i(K)$ hashes uniformly over $\{0, 1, \ldots, 2n_0 - 1\}$. It also requires knowledge of the current state of the file as

defined by the following three variables:

- ℓ: the level of the file, that is, the number of completed full expansions, $\ell \geq 0$, initially $\ell = 0$,
- n: the number of buckets per group of groups not yet expanded during this partial expansion, $n_0 \leq n < 2n_0$, initially $n = n_0$,
- p: a pointer indicating the next group to be expanded from n to n + 1 buckets, $0 \leq p < N2^{\ell}$, initially $p = 0$.

These three variables must be updated when expanding or contracting the file, see [4]. The number of directly addressable pages is $m = nN2^{\ell} + p$ and the number of overflow pages $\lfloor m/(k-1) \rfloor$. The pages are assumed to have the real page addresses 0, 1, ..., $m + \lfloor m/(k-1) \rfloor - 1$.

The output of the algorithm is the real page address h' and chain number c of the record with key K. The variable h is the address when counting only directly addressable pages, s keeps track of the number of groups on each level, q is the size of the group hit on the last level, while j, u, and v are auxiliary variables.

Algorithm A

1.1 $v \leftarrow h_0(K)$, $s \leftarrow N$

1.2 $h \leftarrow \lfloor v/r \rfloor$, $c \leftarrow v$ mod r

2.1 for $j \leftarrow 1$ to ℓ do

2.2 if $d_j(K) \geq n_0$ then $h \leftarrow (h \bmod s) + sd_j(K)$ fi ,

2.3 $s \leftarrow 2s$,

2.4 od

3. if h mod s < p then $q \leftarrow n + 1$ else $q \leftarrow n$ fi

4.1 $j \leftarrow \ell + 1$, $u \leftarrow d_j(K)$

4.2 while $u \geq q$ do $j \leftarrow j + 1$, $u \leftarrow d_j(K)$ od

4.3 if $u \geq n_0$ then $h \leftarrow (h \bmod s) + su$ fi

5. $h' \leftarrow \lfloor h/(k-1) \rfloor k + h \bmod (k-1)$,

6. end

We will assume that the rate of expansion is governed by the rule of constant storage utilization. To compute the storage utilization, we need only keep track of the file size (number of pages) and the number of records stored in the file. If there is sufficient overflow space, the file grows linearly, that is, the file size is a linear function of the number of records stored. In this case, the storage utilization is constant and equal to the threshold α. However, if there is an insufficient amount of overflow space, the file must be expanded prematurely and the storage utilization drops below the threshold. This is discussed further in the subsequent section.

To make it easier to locate empty space for overflow records, we can use a bitmap with one bit per overflow page. A bit indicates whether the corresponding page is full or not. The bitmap can be stored in a parameter page in the beginning of the file and fetched into main storage when the file is opened. If the bitmap must be expanded, additional portions can be stored in one or more overflow pages. This should normally not be necessary, except for very large files. The procedure for locating empty space affects the performance. Whenever possible, overflow records from the same home page should be stored on the same overflow page in order to reduce the number of accesses required to scan the chains.

A problem may occur if we allow all the available overflow pages to become completely filled before expanding the file. An expansion of the file by one page should normally result in fewer overflow records. This cannot be guaranteed, however. The number of overflow records may actually increase, even though it is extremely unlikely. This could occur, for example, if all records from a group of pages are moved to the new page during an expansion. To avoid problems with such unlikely but possible cases, we should start expanding the file before completely running out of overflow space. The safety margin need not be large, one or two pages should be enough.

5. EXPECTED PERFORMANCE

A mathematical model of the scheme presented above has been developed. Because of space limitations, it could not be included in this paper. We will here study a few numerical examples to gain some understanding of the expected performance.

A few words on the assumptions of the model are necessary. The model is asymptotic, cf. [5]. It is assumed that several overflow chains per page are used and that every kth page is designated as an overflow page, giving an overflow storage ratio of $f = 1/k$. All pages, including overflow pages, are assumed to have the same capacity. This is slightly unrealistic because there are more pointers on an overflow page than on a directly addressable page. The analysis is pessimistic in the following sense: all overflow records from the same home page are assumed to reside on different overflow pages.

This means that, when traversing an overflow chain, fetching the next record always requires one read access. The impact of this assumption is greatest on the cost of expanding the file: every overflow record belonging to a page must be independently read and written.

All the necessary file parameters (parameters required for address computation, number of records stored, etc.) and the bitmap are assumed to be available and updated in main storage. Hence, free overflow space can be located without accessing the file.

For the first example, we choose the following parameter combination: page size 20 records, 5 overflow chains per page, required storage utilization 0.85, overflow storage factor 0.0625 (every 16th page) and 2 partial expansions. The graphs of figures 3 -7 show the expected performance of this file over a full expansion.

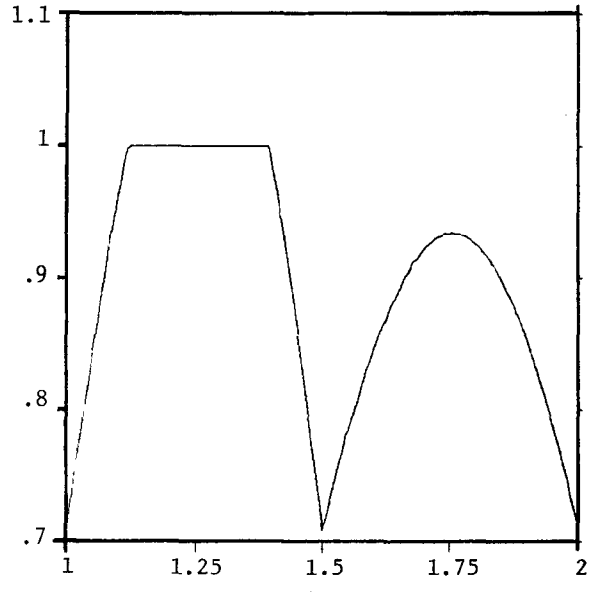

Fig. 3: Utilization of available overflow storage

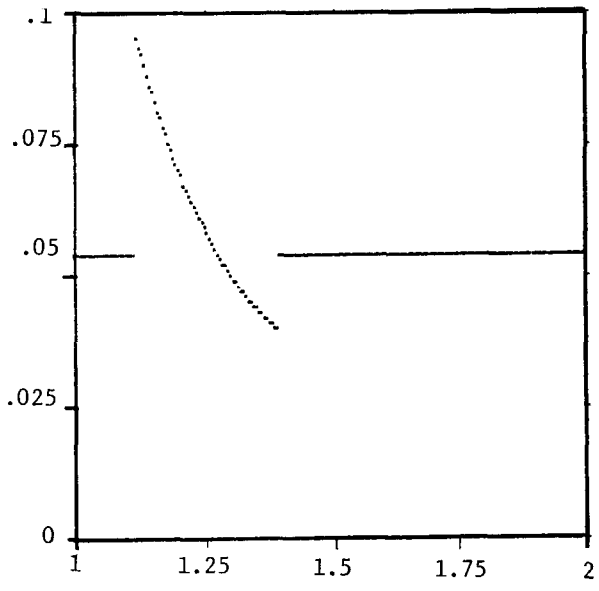

Fig. 4: Expansion rate (number of pages added per inserted record)

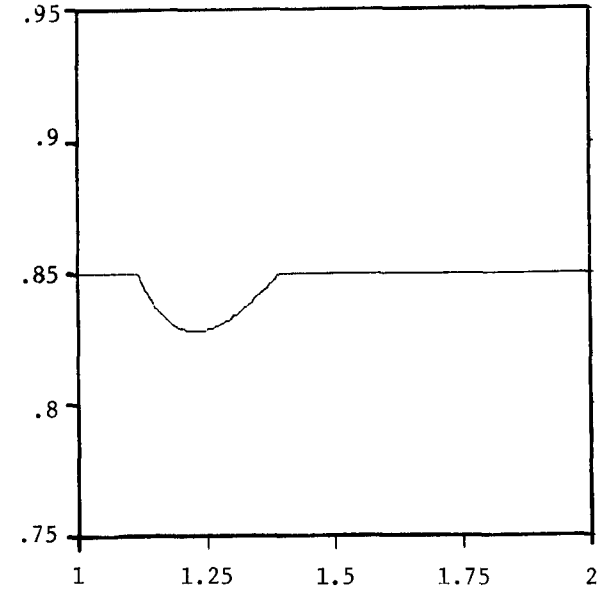

Fig. 5: Overall storage utilization

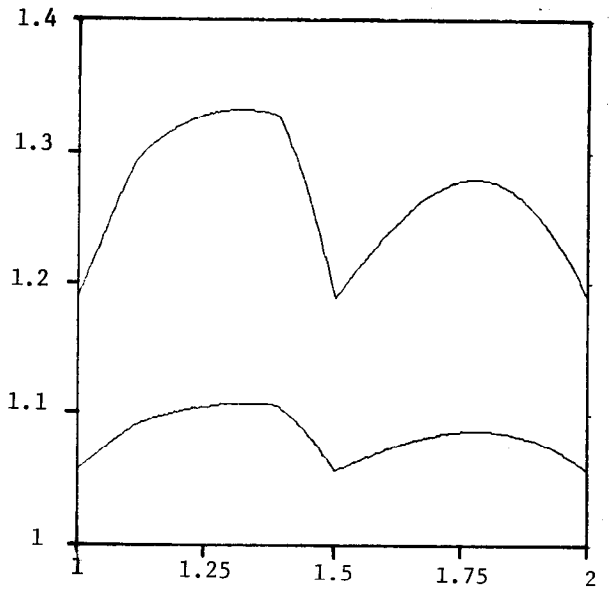

Fig. 6: Expected number of accesses for successful (lower line) and unsuccessful (upper line) searches

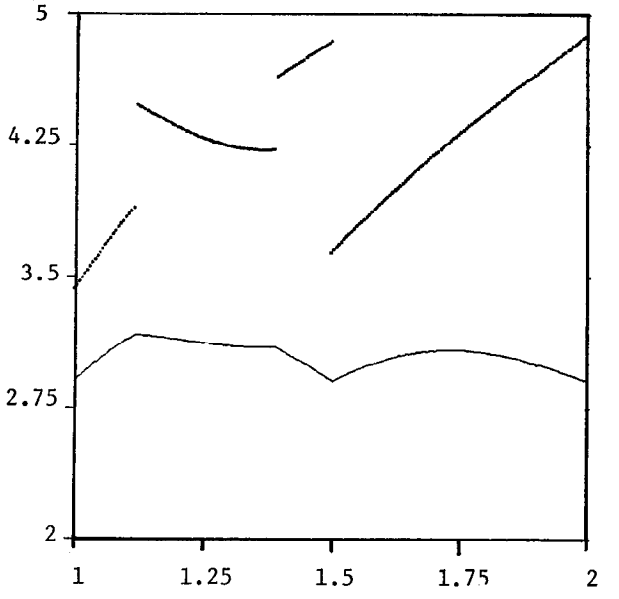

Fig. 7: Expected number of accesses for inserting a record. The lower line does not include costs for expanding the file.

The overflow storage factor was deliberately set low in order to show the behaviour of the file when running out of overflow space. Fig. 3 shows the utilization of the available overflow space. When the relative number of records in the file reaches 1.12, all the overflow space is in use. This causes a sudden increase in the expansion activity, see Fig. 4. From 1.12 to 1.40, the expansion rate is completely overflow-controlled. At 1.40, overflow storage ceases to be a bottleneck and the expansion rate returns to its normal level (0.059 pages/record or 17 records/page), see Fig. 4. During the whole second partial expansion, there is enough overflow storage available.

The increased expansion rate temporarily forces the overall storage utilization below 0.85, see Fig. 5. Because of the lower storage utilization, the expected search lengths grow more slowly during this period, see Fig. 6. There are three sudden changes in the cost of inserting a record, see Fig. 7. The changes at 1.12 and 1.40 are caused by the switch to/from overflow-controlled from/to load-controlled expansion. At 1.5 the first partial expansion ends and the second one begins.

For this file, the average performance over a full expansion is:

Successful Search	1.085
Unsuccessful Search	1.270
Insertion	4.305

Table 3 and 4 list the average performance of files with bucket size 20 and 40 records. The number of overflow chains is 5. In each case, the overflow storage factor is the lowest possible, such that at no point of an expansion lack of overflow storage occurs. In other words, the expansion rate is fully load-controlled. Further results will be given in a forthcoming paper dealing with the analysis of the scheme.

Expansions	Storage Utilization	Overflow Storage Ratio	Successful Search	Unsuccessful Search	Insertion
$n_0 = 1$	0.75	1/13	1.112	1.341	3.952
	0.80	1/9	1.177	1.527	4.500
	0.85	1/6	1.290	1.842	5.280
	0.90	1/4	1.507	2.422	6.383
$n_0 = 2$	0.75	1/33	1.031	1.101	3.374
	0.80	1/20	1.055	1.178	3.815
	0.85	1/13	1.096	1.304	4.419
	0.90	1/8	1.175	1.539	5.281
$n_0 = 3$	0.75	1/49	1.021	1.069	3.430
	0.80	1/29	1.038	1.125	3.827
	0.85	1/17	1.069	1.223	4.433
	0.90	1/10	1.129	1.407	5.313

Expected Performance For A File With Page Size 20 Records And 5 Overflow Chains Per Page

TABLE 3

Expansions	Utilization	Storage Ratio	Successful Search	Unsuccessful Search	Insertion
$n_0 = 1$	0.75	1/14	1.130	1.570	3.946
	0.80	1/9	1.229	1.949	4.737
	0.85	1/6	1.405	2.574	5.872
	0.90	1/4	1.761	3.746	7.657
$n_0 = 2$	0.75	1/54	1.018	1.096	2.835
	0.80	1/29	1.039	1.200	3.291
	0.85	1/17	1.080	1.394	3.998
	0.90	1/10	1.166	1.776	5.087
$n_0 = 3$	0.75	1/112	1.008	1.046	2.709
	0.80	1/55	1.019	1.105	3.051
	0.85	1/29	1.043	1.225	3.637
	0.90	1/15	1.096	1.483	4.628

Expected Performance For A File With Page Size 40 Records And 5 Overflow Chains Per Page

TABLE 4

A comparison of the results in table 3 and 4 with the corresponding results for the two-file version in [5] shows that for $n_0 = 1$ the expected search lengths of the current scheme are higher. For $n_0 > 1$ they are approximately the same or lower. However, it should be borne in mind, that the model here is quite pessimistic and that there is still room for improvements (see next section). As to insertions, the same comment applies and, furthermore, some of the costs of inserting a record were deliberately ignored in the analysis of the two-file version, cf. [5]. The main advantage offered by the single-file version is simplified storage management.

6. OPEN PROBLEMS

As indicated by table 3 and 4, the basic operations of the proposed scheme are expected to be quite fast. Even so, further improvements are still possible. Two such ideas are briefly outlined below.

The performance model above assumes that all overflow records from the same home page reside in different overflow pages. Some scheme that attempts to cluster them on one or a few pages would significantly improve the performance. To improve retrieval performance it is enough to keep all records on the same overflow chain stored on the same page. This would also reduce the cost of inserting a record because fewer accesses are required to scan down a chain. Going further, attempting to keep all overflow records from the same home page on one overflow page, would not, reduce the search lengths further. It can only reduce the cost of expanding the file, but these savings may well be offset by additional accesses for rearranging records during insertion.

Designing an efficient clustering scheme for overflow records is an open problem. Nevertheless, we can still find a lower bound on the expected length of successful searches. Fetching an overflow record always requires at least one extra access, whatever the clustering scheme be. As mentioned in the section three, this situation can be modelled by having the number of overflow chains approach infinity.

No. of partial expansions	Storage Utilization	Bucket Size b = 20	b = 40
1	0.75	1.059	1.052
	0.80	1.087	1.082
	0.85	1.131	1.128
	0.90	1.203	1.204
2	0.75	1.020	1.010
	0.80	1.034	1.020
	0.85	1.056	1.039
	0.90	1.095	1.074
3	0.75	1.014	1.005
	0.80	1.024	1.011
	0.85	1.042	1.023
	0.90	1.075	1.048

Lower bound on the average length of successful searches

TABLE 5

The lower bound is tabulated in table 5 for bucket size 20 and 40 records. The overflow storage factor is the same as in table 3 and 4. A comparison with the results of table 3 and 4 shows that there is some room for improvement, expecially for $n_0 = 1$ and large buckets. Smaller buckets and several partial expansions per full expansion result in fewer overflow records per bucket, thereby reducing the margin for improvement. For insertions the margin for improvement should be greater, but on this problem we have no results.

In the present version of the scheme, additional overflow space is always allocated at the same rate, but the demand for overflow space is heaviest in the middle of an expansion, cf. fig. 3. If additional overflow space could be assigned at a variable rate (less in the beginning and end, and more in the middle of an expansion) the performance should be improved. The problem is to keep the address computation simple.

ACKNOWLEDGEMENT

Valuable assistance with the performance analysis was given by Kok-Weng Lee.

REFERENCES

1. Fagin R., Nievergelt J., Pippenger N., Strong H.R.: Extendible hashing - a fast access method for dynamic files, ACM Trans. Database Syst., 4, 3, 1979, 315-344.

2. Karlsson K.: Resolution de collisions du hachage virtuel lineaire par une methode du type addressage ouvert, Rap. D.E.A. Inf., Institut de Programmation, Univ. Paris VI, 1979.

3. Larson P.-Å.: Dynamic hashing, BIT, 18, 2, 1978, 184-201.

4. Larson P.-Å.: Linear hashing with partial expansions, Proc. 6th Conf. Very Large Data Bases, Montreal, 1980, 224-232.

5. Larson P.-Å.: Performance analysis of linear hashing with partial expansions, ACM Trans. Database Syst. (to appear).

6. Litwin W.: Virtual hashing: a dynamically changing hashing, Proc. 4th Conf. Very Large Data Bases, Berlin, 1978, 517-523.

7. Litwin W.: Hachage virtuel: une nouvelle technique d'adressage de memoires, These de Doctorat d'Etat, Univ. Paris VI, 1979.

8. Litwin W.: Linear hashing: a new tool for files and tables addressing, Proc. 6th Conf. Very Large Data Bases, Montreal, 1980, 212-223.

9. Martin G.N.N.: Spiral storage: incrementally augmentable hash addressed storage, University of Warwick, Theory of Computation Report No. 27, 1979.

10. Mullin, J.K.: Tightly controlled linear hashing without separate overflow storage, BIT 21, 4, 1981, 389-400.