CASCADE HASHING

Peter Kjellberg and Torben U. Zahle

Computer Science Department, University of Copenhagen
Sigurdsgade 41, DK-2200 Copenhagen N, Denmark

Abstract

Cascade Hashing is a new dynamic hashing scheme which is based on Spiral Storage.

The purpose of this paper is first to give a unified exposition of Linear Hashing, Spiral Storage and other dynamic hashing schemes, and second to describe a new method for storing overflow records. The method stores the overflow records in the main file itself and clusters overflow records from each primary bucket in one or very few overflow buckets.

Calculations on the load of the file promises search lengths very close to one even for a storage utilization above 90%, which makes the method appear better than any present dynamic hashing scheme.

1. Introduction.

Since the early seventies, a number of techniques have been described which allow a hash file to change its size dynamically along with the changing number of stored records.

The first proposals, named Expandable Hashing [Knot71], Dynamic Hashing [Lars78] and Extendible Hashing [Fagi79], are all based on the same idea; when a bucket overflows, instead of creating an overflow record, the file is extended with a new bucket and the records of the overflowing bucket are split between the old and the new bucket. To keep track of the splitting, an index or directory is created that grows and shrinks with the size of the file.

Different kinds of tree structures are used for the index, but Virtual Hashing [Litw78] reduces the index to a bit-table with one bit for each bucket by introducing a series of hash functions, one for each "level" of splitting. At any stage the primary bucket of a record can be reached without accessing other buckets.

Virtual Hashing also allows for a certain amount of overflow to defer splitting. Versions of Extendible Hashing with overflow are described in [Scho81] and [Tamm82]. A rather complex version with an index of limited size is found in [Lome83].

In the late seventies methods were invented which avoided indexes completely: Spiral Storage [Mart79] and Linear Hashing [Litw80]. While Spiral Storage has remained rather unnoticed, Linear Hashing has drawn much attention. The basis of these methods is still the splitting of buckets, but instead of splitting the buckets that overflow, the buckets are split in linear order 0,1,2,..., thus reducing the index to a single pointer showing which bucket is to be split next.

Linear Hashing with Partial Expansions [Lars80] is a generalization of Linear Hashing which gives better performance. A further generalization of Linear Hashing is given by Ramamohanarao & Lloyd [Rama82].

[Mull81] and [Lars82] describe versions of Linear Hashing and Linear Hashing with Partial Expansions respectively, where the overflow is stored in the main file, as opposed to the earlier versions where overflow was stored in a separate overflow area.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

The schemes without indices will be discussed in more detail later in the paper.

The number of records which can be stored in a bucket is called the blok factor and denoted b; b' is the blok factor of a possible separate overflow storage. If the main file and the separate overflow storage contains M and M' buckets respectively, the file can at most hold bM+b'M' records. When the current number of records is x, then x/(bM+b'M') is called the storage utilization, while x/bM is called the load factor, denoted f. It may be noted, that when a separate overflow storage is used, the load factor can become greater than one. When overflow is stored in the main file, the load factor equals the storage utilization.

In the present paper we will only consider the so called "controlled split", where a split is performed only when the file has been filled to a certain limit, thus making the file size grow linearly with the number of stored records.

2. Conceptual Linear Hashing.

First we describe Linear Hashing briefly, introducing terms used in the following descriptions.

Linear Hashing expands the file by splitting the buckets one by one in linear order 0,1,2,..

When a bucket is split, approximately half of its records are removed and restored in a new bucket at the end of the file. No index is needed, only a single pointer p showing which bucket to split next. When all buckets in the file are split, the pointer is reset to the first bucket (0) and the process starts over again, but now with a file twice as large. The overflow records are taken care of by explicit pointers to a separate overflow area.

We will only consider the case where the initial file size is 1.

After L completed doublings or expansions of the file, its size is 2^L. L is called the file level.

Using a series of split functions $h_L(k)$, one for each file level, where each $h_L(k)$ hashes randomly on the interval $[0,2^L[$, we get the hash function

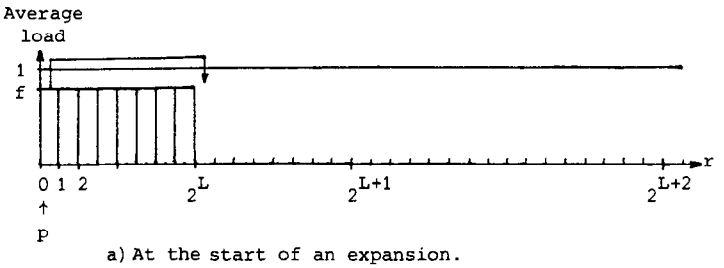
$$h(k) = \begin{cases} h_L(k) & \text{if } h_L(k) \geq p \\ h_{L+1}(k) & \text{otherwise.} \end{cases} \qquad (1)$$

h(k) is a real number; to get the integer bucket number we use

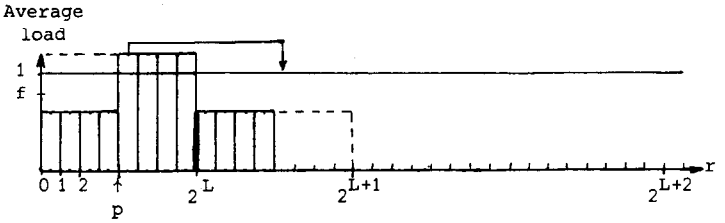
$$H(k) = \lfloor h(k) \rfloor. \qquad (2)$$

In the following discussion we use the real bucket number, and assume truncation to get the integer bucket number.

Let D(k) distribute uniformly over as large a number of buckets as needed. Then an example of a Linear Hashing function is shown in fig. 1.



a) At the start of an expansion.



b) During an expansion.

$$r = h^M(k) = \begin{cases} D(k) \bmod 2^L & \text{if } D(k) \bmod 2^L \geq p \\ D(k) \bmod 2^{L+1} & \text{otherwise} \end{cases}$$

where $L = \lfloor \log_2 M \rfloor$ and $p = M - 2^L$

Fig.1. Linear Hashing.

The height of the buckets in the figure illustrates the load, where f is the load factor of the file. Buckets taller than 1 are overflowing; the quantity above 1 is actually stored elsewhere.

Note, that any split function $h_{L+1}(k)$ stores k either in the same bucket as the previous function $h_L(k)$ or in the bucket into which $h_L(k)$ was split.

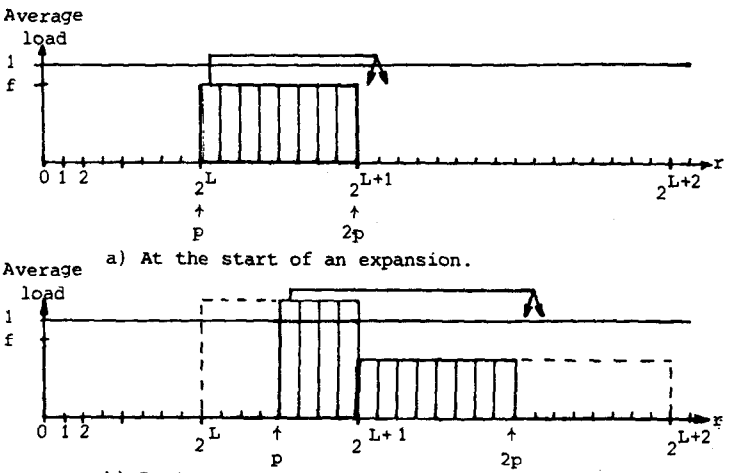
As the file grows to a certain size (M), it is possible to determine the level L and the split position p. In the case of Linear Hashing we have $L = \lfloor \log_2 M \rfloor$ and $p = M - 2^L$.

In the following we will mark the hash function h(k) with a superscript M to indicate how many buckets the file contains: $h^M(k)$.

The split in Linear Hashing may be thought of as a process where the first bucket in the file is removed and its records restored in two new buckets appended to the end of the file. Fig. 2

shows this process, called Conceptual Linear Hashing, which is the way we will consider Linear Hashing in the rest of the article.

d(k) is a function distributing the records uniformly on the interval [0,1[.



a) At the start of an expansion.

b) During an expansion.

$$r=h^M(k)=\begin{cases}2^L(1+d(k)) & \text{if } 2^L(1+d(k))\geq p\\ 2^{L+1}(1+d(k)) & \text{otherwise}\end{cases}$$

where p=M and $L=\lfloor \log_2 p \rfloor$

Fig.2. Conceptual Linear Hashing

When the first bucket (bucket 1) is split, the file has doubled and consists of the buckets {2,3}. When both of these are split, the file has doubled again and now consists of the buckets {4,5,6,7}. This process may be continued, and after L completed expansions the file consists of the buckets $\{2^L,\ldots,2^{L+1}-1\}$.

At any time, the bucket to split next, p, is the first bucket in the file. The number of buckets in the file, M, is p as well, so generally the file consists of the buckets {p,..,2p-1}. When bucket p is split, its records are restored in the buckets 2p and 2p+1. The file level L is a function of p: $L = \lfloor \log_2 p \rfloor$.

Computer systems normally only allow data areas to grow and shrink at one end. Anyway, this problem is easy to overcome, since a simple algorithm exists, that transforms the (conceptual) bucket numbers above to physical bucket numbers or page numbers 0,1,2,...

In fig. 3a the numbers of the existing conceptual buckets are listed for the first few M's.

An algorithm transforming these to page numbers must fullfil two criteria:

	Conceptual bucket number: n												
	1	2	3	4	5	6	7	8	9	10	11	12	13
M 1	1												
2		2	3										
3			3	4	5								
4				4	5	6	7						
5					5	6	7	8	9				
6						6	7	8	9	10	11		
7							7	8	9	10	11	12	13

(a) Existing conceptual bucket numbers

	Actual bucket number: P(n)						
	0	1	2	3	4	5	6
M 1	1						
2	2	3					
3	4	3	5				
4	4	6	5	7			
5	8	6	5	7	9		
6	8	6	10	7	9	11	
7	8	12	10	7	9	11	13

(b) Placement of conceptual buckets

Fig. 3. Conceptual to actual bucket number transformation

1) All page numbers 0 to M-1 must be in use.

2) When a bucket is stored on a page, it must remain on this page throughout its entire lifetime.

These demands can easily be met: When the file is expanded the first of the new buckets is placed on the same page as the bucket being removed, while the second of the new buckets is placed on the first free page (fig. 3b).

A glance at the coloumns of fig. 3b immediately unveils a pattern, that is exploited in the following algorithm, transforming a conceptual bucket number n to page number P(n).

```
function P(n);                          (Alg. 1a)
integer n;
begin
  if n mod 2 = 0 then P(n) := P(n/2)
  else P(n) := (n-1)/2;
end;
```

Written without recursion the algorithm is:

```
function P(n);                          (Alg. 1b)
integer n;
begin
  integer m;
  m := n;
  while m mod 2 = 0 do m := m/2;
  P(n) := (m-1)/2;
end;
```

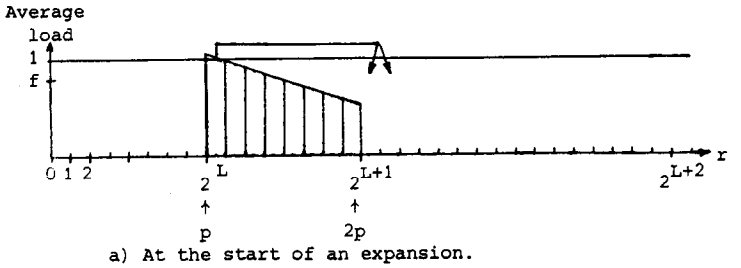
3. Exponential Distribution Method.

Fig. 2 also illustrates the main problem of Linear Hashing: Since each split function $h_L(k)$ distributes the records uniformly over its range of buckets, then during an expansion of the file the still unsplit buckets will contain twice as many records as those already split. Keeping the load factor constant (and preferably high) a changing amount of overflow records result from the unsplit buckets and give rise to changing performance.

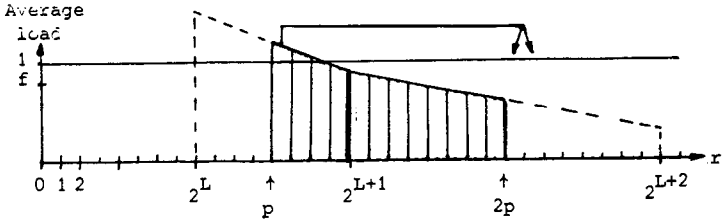
The following is based on two fundamental ideas. First, that it might create less overflow to have a skew distribution of records, so that the buckets which split late are not so full. Second, that it would be nice to choose a distribution which let the performance remain stable.

Even though the average performance of Linear Hashing is good, in many applications the number of records in a file stabilizes for shorter or longer periods, and the point of stabilization may be where the performance is the worst.

To remove performance oscillations we must construct a hash function, which gives the file the same load pattern for any number of buckets. The function must bias the record distribution so that the last bucket in the file contains approximately half the records of the first bucket. Then the two new buckets resulting from a split will fit nicely into the biased load.



a) At the start of an expansion.



b) During an expansion.

Fig.4. Biased Linear Hashing.

With such a biased load a constant load factor of 75% may be kept up with hardly any overflow records, bringing search lengths down to practically one.

The construction of a hash function which creates the same skew distribution of records for any number of buckets, is not trivial though. If the most straightforward biasing, a gradually decreasing distribution, is used, the performance will still oscillate.

However, the method named Spiral Storage described in a university report by G.N.N.Martin [Mart79] actually is such a biased linear hashing scheme. And furthermore the method turns out to be quite simple!

Since Martin's report in our opinion is difficult to understand, and since the method is not explained in terms similar to other literature on hashing, we will in the following summarize Martin's results using the terminology introduced in section 2.

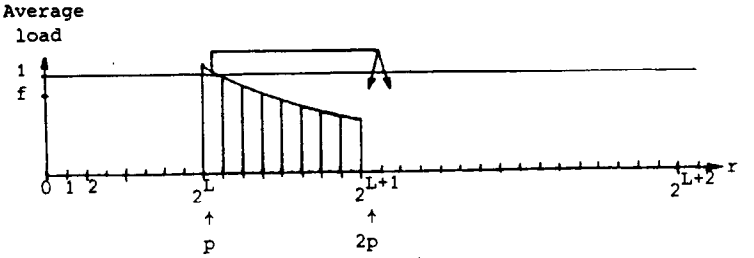
The essence of Spiral Storage is, that instead of the uniform distribution resulting from the split functions $2^L(1+d(k))$, a biased distribution is obtained using the split functions

$$h_L(k) = 2^L \cdot 2^{d(k)}, \qquad (3)$$

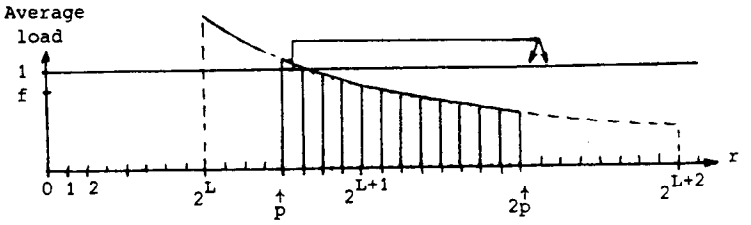
i.e.

$$h^M(k) = \begin{cases} 2^L \cdot 2^{d(k)} & \text{if } 2^L \cdot 2^{d(k)} \geq p \\ 2^{L+1} \cdot 2^{d(k)} & \text{otherwise} \end{cases} \qquad (4)$$

where $p = M$ and $L = \lfloor \log_2 p \rfloor$.



a) At the start of an expansion.



b) During an expansion.

$$r = h^M(k) = \begin{cases} 2^L\, 2^{d(k)} & \text{if } 2^L\, 2^{d(k)} \geq p \\ 2^{L+1}\, 2^{d(k)} & \text{otherwise} \end{cases}$$

where $p = M$ and $L = \lfloor \log_2 p \rfloor$

Fig. 5. Spiral Storage.

In an earlier report [Kjel84] we have shown that the average load for a given hash value r is:

$f/\ln 2 \cdot p/r$

so that for a given M and thus p, the average load follows the inverse function 1/r. This is the case for <u>both</u> split functions, thus we have managed to get a smooth load distribution with no discontinuity at the intersection of the two split functions.

Further it can be seen, that the load at the start of the file (r=p) is f/ln2 and the load at the end of the file (r=2p) is 1/2 * f/ln2.

It is interesting to note that (4) actually produces a file with the sought properties. The load of the last bucket is (a little more than) half the load of the first bucket and the relative distribution of the records within the file does not change.

If the load at the start of the file is set to 1 and the equation then solved for f, then the maximum load factor of a file which idealy has no overflow at all can be found:

$f/\ln 2 = 1 \iff f = \ln 2 = \underline{0.6931}.$

Spiral Storage produces a constant amount of overflow, the relative size of which is derived in [Kjel84]. As well, the average and maximum relative overflow sizes using Linear Hashing are derived, using the same assumptions as for Spiral Storage. The results are shown in table 1. It is seen that Spiral Storage not only removes the oscillations in overflow size but also produces less overflow than the average for Linear Hashing!

f	S.S.	LH av.	LH max
0.5000	0.0000	0.0000	0.0000
0.6931	0.0000	1.6848	4.5408
0.7500	0.4367	2.9871	6.7347
0.8000	1.4143	4.3741	8.7722
0.8500	2.8077	5.9421	10.8611
0.9000	4.5179	7.6523	12.9687
0.9500	6.4702	9.4712	15.0727
1.0000	8.6071	11.3706	17.1573

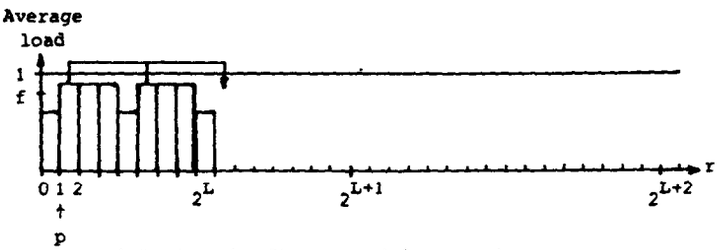
<u>Table 1.</u> Relative amount of overflow using Spiral Storage and Linear Hashing (average and maximum) for different load factors (f). In pct.

To conclude this section we mention that the hash function for Spiral Storage actually may be written as a single expression:

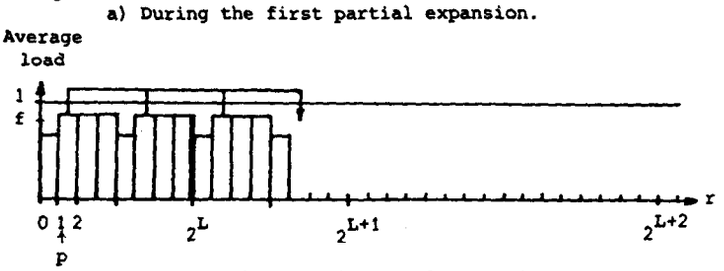
$$H(k) = \left\lfloor 2^{\lceil \log_2 p - d(k) \rceil} + d(k) \right\rfloor \qquad (5)$$

4. Splitting from several buckets.

P.-Å.Larson has shown in [Lars80], that compared with Linear Hashing, a better performance can be achieved using a generalized Linear Hashing where an expansion is done in a series of partial expansions. Using two partial expansions (which according to Larson is a good compromise in many cases) the method works as follows: In the first partial expansion groups of two buckets are split, moving a part (approx. 1/3) of their records to a new bucket (fig. 6a).



a) During the first partial expansion.



b) During the second partial expansion.

<u>Fig.6.</u> Linear Hashing with 2 partial expansions.

When during this first partial expansion all the original buckets have been split, the file has been expanded to 1.5 times its original size. Thereafter the second partial expansion starts, now splitting groups of three buckets, moving approx. 1/4 of their records to the new bucket (fig. 6b). When the second partial expansion is ended, the file has doubled, and the method starts again with the first partial expansion.

Ramamohanarao & Lloyd [Rama82] have considered this idea and give another method where the growth consists of single expansions in which always groups of s buckets are split (fig. 7a).

When all buckets are split, a new expansion starts, again splitting groups of s buckets. Since the file size is not always divisible by s, a number of empty "round-up buckets" are added to the file (fig 7b).

It is obvious, that both methods give better <u>search</u> performance than Linear Hashing since less overflow occurs. In Linear Hashing the still un-

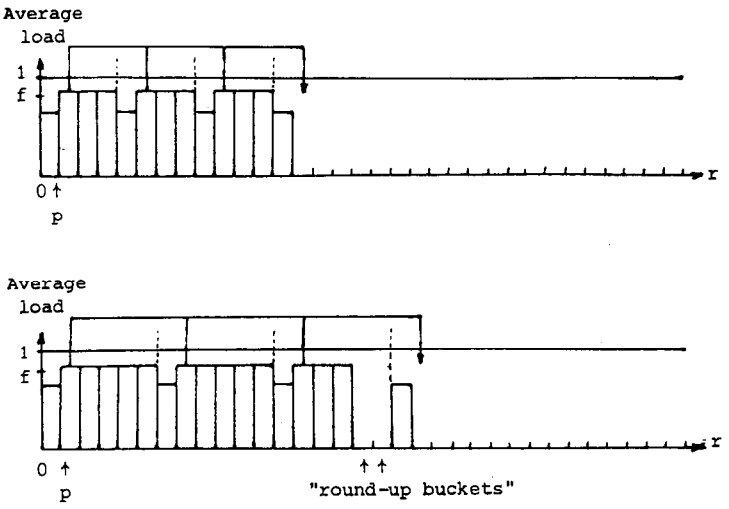


Fig.7. Ramamohanarao and Lloyd's Linear Hashing (s=3) at two consecutive expansions.

split buckets contain twice as many records as the split buckets. With 2 partial expansions the ratio is 3:2 during the first partial expansion and 4:3 during the second. With Ramamohanarao & Lloyd's method the ratio is always (s+1):s.

It is more surprising, that when the number of partial expansions (respectively the value of s) is reasonable, also insertions are faster than with Linear Hashing, despite the fact that a split is more expensive! The reason is, that the increased cost of splitting is more than outweighed by the decrease in overflow chain lengths.

But still the performance oscillates, and in both cases the hash function is more complicated than with Linear Hashing.

It can be expected that splitting from several buckets in Spiral Storage (in the following called Generalized Spiral Storage) also will show an improved performance for both search and insertion times. Furthermore it does not show any of the inconveniences of the generalized Linear Hashing schemes above.

Generalized Spiral Storage resembles the method of Ramamohanarao & Lloyd (but is actually even more general). The performance is the same for any number of buckets, the hash function is no more complicated than that of simple Spiral Storage and it does not need any "round-up buckets".

The idea of the generalized Spiral Storage is very straightforward: Each expansion is performed by 1) adding a number of buckets, say t, to the high end of the file and 2) removing some number, say s, from the low end of the file, distributing the records of the s removed buckets on the the t new buckets! We say, that s buckets split into t, or equivalently that 1 bucket splits into t/s. When t=2 and s=1 this is the simple Spiral Storage.

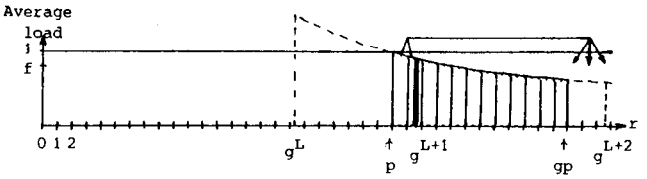
Let g = t/s. In the general case the "file level" is defined as $L = \lfloor \log_g p \rfloor$. Note, that now L is not the number of doublings of the file. (Actually, increasing L by one is equivalent to an expansion of the file by a factor g, which generally is not an integral number of buckets!)

If, again, d(k) is a function distributing all records uniformly on the interval [0,1[, then

$$h^M(k) = \begin{cases} g^L \cdot g^{d(k)} & \text{if } g^L \cdot g^{d(k)} \geq p \\ g^{L+1} \cdot g^{d(k)} & \text{otherwise} \end{cases} \qquad (6)$$

where p = M/(g-1) and $L = \lfloor \log_g p \rfloor$,

is the hash function for generalized Spiral Storage.



a) During an expansion.

$$r=h^M(k) = \begin{cases} g^L\, g^{d(k)} & \text{if } g^L\, g^{d(k)} \geq p \\ g^{L+1}\, g^{d(k)} & \text{otherwise} \end{cases}$$

where $p=\frac{M}{g-1}$ and $L=\lfloor \log_g p \rfloor$

Fig.8. Generalized Spiral Storage with $g=\frac{3}{2}$.

The average load for a given hash value r is ([Kjel84]):

$$f(g-1)/\ln(g) \cdot p/r$$

which means that we get a smooth load distribution in the general case too.

The load at the start and at the end of the file is $f(g-1)/\ln(g)$ and $1/g \cdot f(g-1)/\ln(g)$ respectively. This means that when e.g. g = 3/2 the load of the last bucket of the file is (a little more than) two thirds the load of the first bucket.

What we have achived is a more even record distribution over the file, reducing the number of overflow records; the price is a greater number of accesses to perform a split.

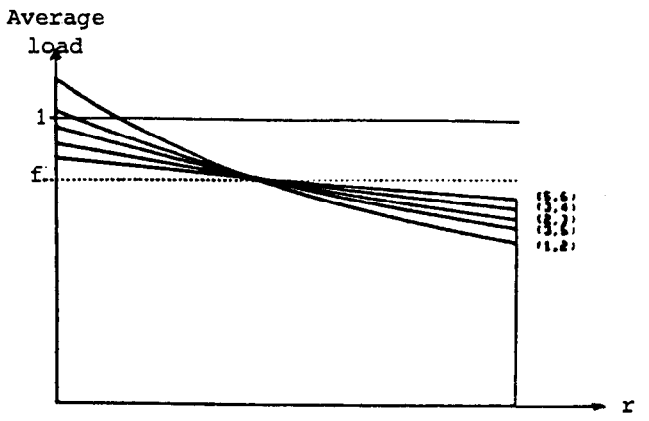


Fig.9. The average load of a generalized Spiral Storage file, for a selection of (s,t)-pairs.

The cases where t = s+1 are equivalent to those of Ramamohanarao & Lloyd: Each split produces one extra bucket. In other cases more than one bucket results from each split (thus provoking a split more seldom) but are in all other respects no more complicated. E.g. the case (s,t) = (3,5) produces two extra buckets by splitting three (which may be thought of as splitting 1 bucket into 5/3 bucket three times) and actually in all respects it acts as a compromise between (1,2) and (2,3) (two times 1 into 3/2).

Again, we can find the maximum load factor of a file which ideally has no overflow by equalling the load at the start of the file (shown above) with 1 and solving for f:

$f = \ln(g)/(g-1)$.

s,t	f_{max}
1,2	0.6931
3,5	0.7662
2,3	0.8109
5,7	0.8412
3,4	0.8630
4,5	0.8926
5,6	0.9116

Table 2. Maximum load factor for generalized Spiral Storage without overflow for a selection of s,t-pairs.

As with simple Spiral Storage the relative amount of overflow is deriven in [Kjel84]:

f	1,2	3,5	2,3	5,7	3,4	4,5	5,6
0.70	0.01						
0.75	0.44						
0.80	1.41	0.18					
0.85	2.81	1.02	0.27	0.02			
0.90	4.52	2.40	1.29	0.66	0.30	0.02	
0.95	6.47	4.22	2.93	2.11	1.55	0.85	0.46
1.00	8.61	6.36	5.06	4.20	3.59	2.79	2.28

Table 3. Relative amount of overflow using generalized Spiral Storage for a selection of s,t-pairs and different load factors (f). In pct.

The only problem left in making the generalized Spiral Storage work is a generalization of algorithm 1, transforming the conceptual bucket numbers to (physical) page numbers. The algorithm works exactly as alg.1 and is given here without further comments.

The algorithm transforms the conceptual bucket number n to page number P(n) in the case where s buckets are split into t:

```
function P(n);                          (Alg. 2a)
integer n;
begin
  if n mod t < s
    then P(n) :=
      P(⌊n/t⌋·s + n mod t)
    else P(n) :=
      ⌊n/t⌋·(t-s) + (n-s) mod t;
end;
```

Written without recursion the algorithm is:

```
function P(n);                          (Alg. 2b)
integer n;
begin
  integer m;
  m := n;
  while m mod t < s do
    m := ⌊m/t⌋·s + m mod t;
  P(n) := ⌊m/t⌋·(t-s) + (m-s) mod t;
end;
```

It may be noticed that in the cases where t = s+1 alg.2 reduces to:

```
function P(n);                          (Alg. 3)
integer n;
begin
  if n mod t ≠ s
  then P(n) := P(⌊n/t⌋·s + n mod t)
  else P(n) := (n-s)/t;
end;
```

Finally we mention that the hash function for generalized Spiral Storage may also be written as a single expression:

$$H^M(k) = \left\lfloor g^{\lceil \log_g p - d(k) \rceil + d(k)} \right\rfloor \qquad (7)$$

where $p = M/(g-1)$.

5. Overflow handling.

The easiest way to treat the overflow problem is of course simply to avoid any overflow at all. Because of the biased distribution of records in the file, we must expect an overflowing bucket to be located at the beginning of the file, and we may simply split buckets up to and including the one that overflows. This scheme, however, has two major disadvantages which both stem from the fact that a rather long sequence of splits can result from a single insertion:

- The load factor of the file may drop heavily.
- Very long insertion times may occur now and then.

For this reason, and in order to increase the load factor beyond the values listed in table 2, a new scheme for handling overflow records is described in the following.

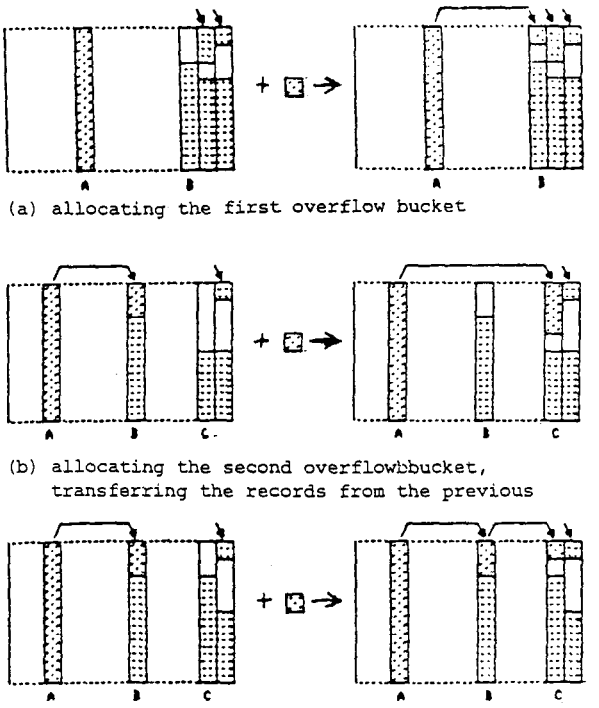
The main purpose of the method is to incorporate the overflow in the main file, thus reducing the data area to one single file growing linearly in size with the number of stored records.

As mentioned in the introduction two Linear Hashing schemes with overflow in the main file have been described by Mullin [Mull81] and Larson [Lars82]. Both methods chain the overflow records to the primary bucket. The difference between the two is, that while Mullin uses the empty space in other buckets for the overflow, Larson sets every k'th bucket aside entirely for overflow.

Larson concludes his article by advising, that overflow records be clustered in one or a few overflow buckets. This is the second purpose of the present method. To achive this goal the method uses, like Mullin, the empty space in other buckets for overflow, but instead of chaining single overflow records, it chains overflow buckets, i.e. the empty space in a bucket is used entirely for overflow from one other bucket.

The success of this idea is due to the biased record distribution. The overflow is gathered at the "low" end of the file, while the empty space is gathered at the "high" end.

The method works as follows:



(a) allocating the first overflow bucket

(b) allocating the second overflowbbucket, transferring the records from the previous

(c) allocating the second overflow bucket, chaining to the previous

Fig. 10. Chaining overflow buckets in Cascade Hashing.

When insertion of a record is attempted (fig. 10a) into a full primary bucket (A), the highest numbered bucket with an uncluttered overflow space (B) is allocated, the record is inserted herein, and this new overflow bucket is chained to the primary bucket. Additional records with primary bucket A are inserted into the overflow bucket B as long as space is available.

When there is no more room in B, a new overflow bucket (C) is selected the same way: The highest numbered bucket with an uncluttered overflow space.

Since C usually is higher numbered than B (the file has presumably been split a number of times because of the insertions), then it may be expected that the overflow space in C is larger than that in B, and then usually all overflow records of A can be stored in C, thus keeping the length of A's overflow chain to 1. Therefore, if possible, the overflow records in B are transferred to C and the chain is relinked (fig. 10b), we say that the overflow records cascade from B to C. Further overflow records are inserted using the same mechanism.

If C cannot contain all overflow records, then only the new record is inserted in C, and C is chained to B (fig. 10c).

The purpose is to keep the overflow chains short. As the main operation on a hash file is searching, it is profitable to reorganize the overflow chains during insertion and deletion in order to reduce search time - as long as the reorganizations do not seriously affect the insertion- and deletion-times.

There are two other important ways to keep the chains short. First, during insertion, no record shall be an overflow record unless its primary bucket is full of its own records. If the primary bucket is full, but contains overflow records from another bucket, the "strangers" are pushed out and inserted elsewere.

Second, all buckets in a chain, except for the last, must be full. When a record is deleted, some other record from the end of the chain is brought back to fill out the hole.

Algorithms for insertion, deletion, splitting and grouping are included as appendix A.

We still need to verify, that this method works for all cases. In [Kjel84] we show, that the relative number of overflowing buckets, assuming a perfect distribution, is

$f/\ln(g)-1/(g-1)$.

This is illustrated in table 4. The table shows, that there will be at least one free bucket for each overflowing bucket, and usually many more.

f	1,2	3,5	2,3	5,7	3,4	4,5
0.70	1.0					
0.75	8.2					
0.80	15.4	6.6				
0.85	22.6	16.4	9.6	2.6		
0.90	29.8	26.2	22.0	17.5	12.8	3.3
0.95	37.1	36.0	34.3	32.3	30.2	25.7
1.00	44.3	45.8	46.6	47.2	47.6	48.1

Table 4. Relative number of overflowing buckets for a selection of (s,t)-pairs and different load factors (f). In pct.

Of course, it may still happen, that the actual records are not distributed uniformly with d(k), so there is a risk that no free overflow bucket can be found when needed. This problem can be solved by allowing the overflow chains to coalesce, i.e. that overflow chains from several primary buckets use the same overflow bucket [Vitt82,Knut73]. But this is probably not worth the trouble; a much easier solution is to perform one or more splits. Eventually some free space will result.

This solution may cause the load factor to drop below the wanted level, but that ought not to occur. Actually, if it does, it is merely a warning that either the load factor is set too high or the used distribution function d(k) is not uniform enough! (The extra splitting can be seen as "graceful degradation".)

A typical situation is shown in fig. 11.

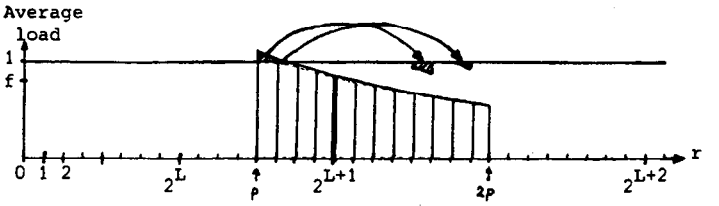


Fig. 11. A typical situation with f = 0,8 and g = 2.

We conclude this section with a few more results. In [Kjel84] we derive the number of the last bucket in the file (n_{last}) when a bucket n_{full} first overflows. Choosing n_{last} as overflow bucket for n_{full}, we then derive the highest load factor f_{max} at which this overflow bucket can hold the overflow of n_{full} throughout the entire lifetime of n_{full}. These load factors are tabellized in the first coloum of table 5. Thus with a load factor up to f any bucket only needs one overflow bucket, and this overflow bucket never needs to be moved!

In practice the records do not distribute ideally, and thus some buckets need more overflow space. But there is still room in the file for cascading the overflow records forward to another bucket.

For this reason the number of records stored in second (or higher) overflow bucket may be assumed to be negligible, and thus, by calculating the expected amount of overflow records and the expected number of overflowing buckets, we may state an expected length of successfull and unsuccessfull search. This is also done in table 5 - we will let the figures speak for themselves.

At present, work is in progress simulating the method with stochastic input.

s,t	f_{max}	O	ss	O_B	us
1,2	0.8568	3.0	1.030	23.6	1.236
3,5	0.8993	2.4	1.024	26.0	1.260
2,3	0.9224	2.0	1.020	27.5	1.275
5,7	0.9369	1.7	1.017	28.5	1.285
3,4	0.9469	1.4	1.014	29.2	1.292
4,5	0.9597	1.1	1.011	30.1	1.301
5,6	0.9675	1.0	1.010	30.7	1.307

Table 5. Maximum load factor for Cascade Hashing with one fixed overflow-bucket per primary bucket for a selection of s,t-pairs.
At these maximum load factors the table further shows:
O : Relative amount of overflow (in pct)
ss: Expected search length successfull search.
O_B: Relative number of overflowing buckets (in pct.)
us: Expected search length unsuccessfull search.

6. Conclusion

In this paper a new linear hashing scheme, called Cascade Hashing, has been described. It is based on two fundamental ideas:

1) That a biased distribution of records is used - like in Spiral Storage, and
2) That the overflow records from a bucket are clustered and stored in the main file.

Calculations seem to indicate that the performance with respect to successfull search, unsuccessfull search, insertion and deletion is superior to other known dynamic hashing schemes.

Acknowledgements

The authors would like to thank Per-Åke Larson, who first introduced us to the report of G.N.N.Martin.

Also thanks to Bent Pedersen at the University of Copenhagen for pointing out the similarities between Linear Hashing With Partial Expansions and generalized Spiral Storage.

References

Fagi79 Fagin, Ronald; Nievergelt, Jurg; Pippenger, Nicholas; Strong, H. Raymond - Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Trans. on Database Systems 4, 3, (Sep. 1979), 315-344

Kjel84 Kjellberg, Peter; Zahle, Torben U. - Cascade Hashing. DIKU rapport 84/6. Institute of Datalogy, University of Copenhagen, Copenhagen, Denmark, 1984

Knot71 Knott, Gary D. - Expandable Open Addressing Hash Table Storage and Retrieval. Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971, pp. 187-206

Knut73 Knuth, Donald E. - The Art of Computer Programming, Vol. 3/Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973

Lars78 Larson, Per-Åke - Dynamic hashing. BIT 18, 2, (1978), 184-201

Lars80 Larson, Per-Åke - Linear hashing with partial expansions. Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada, October 1-3, 1980, pp. 224-231

Lars82 Larson, Per-Åke - A single-file version of linear hashing with partial expansions. Proc. 8th Int. Conf. on Very Large Data Bases, Mexico City, September, 1982, pp. 300-309

Litw78 Litwin, Witold - Virtual Hashing: A Dynamically Changing Hashing. Proc. 4th Int. Conf. on Very Large Data Bases, West-Berlin, Germany, September 13-15, 1978, pp. 517-523

Litw80 Litwin, Witold - Linear hashing : A new tool for file and table addressing. Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada, October 1-3, 1980, pp. 212-223

Lome83 Lomet, David B. - Bounded Index Exponential Hashing. ACM Trans. on Database Systems 8, 1, (Mar. 1983), 136-165

Mart79 Martin, G. N. N. - Spiral storage: Incrementally augmentable hash addressed storage. Univ. of Warwick, Theory of Computation Report no.27. Coventry, England, March 1979

Mull81 Mullin, James K. - Tightly Controlled Linear Hashing Without Separate Overflow Storage. BIT 21, 4, (1981), 390-400

Rama82 Ramamohanarao, K.; Lloyd, John W. - Dynamic Hashing Schemes. The Computer Journal 25, 4, (Nov. 1982), 478-485

Scho81 Scholl, Michel - New File Organizations Based on Dynamic Hashing. ACM Trans. on Database Systems 6, 1, (Mar. 1981), 194-211

Tamm82 Tamminan, Markku - Extendible Hashing with Overflow. Inf. Proc. Letters 15, 5, (Dec. 1982), 227-232

Vitt82 Vitter, Jeffrey Scott - Implementations for Coalesced Hashing. Comm. ACM 25, 12, (Dec. 1982), 911-926

Appendix A

Below we give algorithms for insertion, deletion, splitting and grouping. In order to make the algorithms simple, we have assumed that a sufficiently large number of buckets can be in core simultaneously, and in a number of situations more sophisticated solutions to the treatment of special cases have been left out, although they could have saved accesses.

Some details in the implementation will be discussed briefly at the end of the appendix.

The algorithms presupposes the existence of the parameters s, t and p, as these are used earlier in this paper, i.e. the file contains the (g-1)p buckets with numbers p,..,gp-1 (where g=t/s), and a split is performed by removing s buckets and adding t buckets.

The algorithms refer to a number of "bucket types":

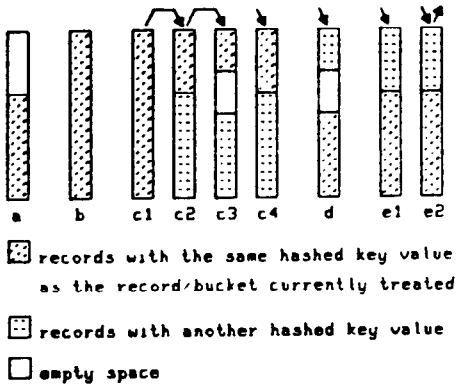


Fig 12. Bucket types.

Algorithm I (Insertion of record k)

I1 [Split] If the maximum load factor is exceeded, then perform a split.

I2 [Search] Search for key k. If the key is found insertion terminates unsuccessfully. (The search ends with the last bucket in the overflow chain for k - maybe the primary bucket - as current.) Let B <- current bucket. (B can be of any type but c1 and c2.)

I3 [Bucket type a or d, insert] If B is of type a or d (there is room for k in the primary bucket) then insert k in B and terminate.

I4 [Bucket type e1 or e2] If B is of type e1 or e2 (the primary bucket is full, but contains overflow from another bucket) then

I4.1 [Select an overflow record] Select one of the 'foreign' overflow records in B, say k', and remove it (temporarily!)

I4.2 [Insert] Insert k in B.

I4.3 [Reorg. and find end of chain for k']

I4.3.1 If B is of type e1 and k' was the only overflow record in B, then let B <- the previous bucket in the chain for k'.

I4.3.2 If B is of type e2 then

I4.3.2.1 If k' was the only overflow record in B then link B out of the chain for k'.

I4.3.2.2 Follow the chain for k' to the end and let B <- the last bucket in the chain.

I4.4 [Switch to insertion of k'] Let k <- k'. (B is now known to be of type b, c3 or c4.)

I5 [Bucket type c3, insert] If B is of type c3 (B has room for k) then insert k and terminate.

I6 [Select new overflow bucket] (B is now of type b or c4.) Select a new overflow bucket B'. If none is found then perform a split, write a warning to the maintenance staff and return to I2.

I7 [Insert] Insert k in B'.

I8 [Reorg. chain] If B is an overflow bucket and B' can contain all overflow records in B, then move the overflow records from B to B' and let B <- the predecessor to B in the chain. Repeat I8 as long as possible.

I9 [Link] Link B' to B.

Algorithm D (Deletion of record k)

D1 [Search] Search for key k. If the key is not found deletion terminates unsuccessfully. (the search ends with the bucket containing k as current.) Let B <- current bucket.

D2 [Delete] Delete k from B.

D3 [Simple deletion] If B is of type a, b, c3, c4, d or e1 (no chains need reorganization after the deletion of k) then go to D7. (In case c3, c4, d and e1 an action similar to I8 may be performed.)

D4 [Find end of chain] (B is now of type c1, c2 or e2. The overflow chain needs reorganization.) Follow the chain emanating from B to the end; let B' <- the last bucket in the chain.

D5 [Select a record] Select an overflow record in B', say k'. Remove k' from B'. If k' was the only overflow record in B' then link B' out of the chain.

D6 [Restore k'] Insert k' in B. (The 'hole' after k is filled.)

D7 [Group] If the load factor now is below minimum, then perform a grouping.

Algorithm S (Splitting)

S1 [Scan buckets to be split] Let B be bucket number p, p+1, ..., p+s-1 repeatedly. For each bucket do

S1.1 [Read] Read B. (It can be of any primary bucket type, i.e. all but c2, c3 and c4.)

S1.2 [Read own overflow] If B is of type c1 then all overflow buckets in its chain are read, the overflow records are removed (but saved for later!) and the buckets rewritten with the overflow space marked as empty.

S1.3 [Reorg. 'foreign' overflow chain] If B is of type d, e1 or e2 (this situation should be extremely rare) the B is linked out of the chain and the 'foreign' overflow records put aside for a while.

S2 [Increment] Let p <- p+s.

S3 [Rehash] Rehash all records, including own overflow records from S1.2 but excluding 'foreign' overflow records from S1.3. (The records hash to the new buckets gp-t,..,gp-1.) If some of the new buckets should overflow (this should be a rare situation) then the overflow is put aside for a while.

S4 [Rewrite] Rewrite the new buckets.

S5 [Insert overflow] Use algorithm I to insert overflow records put aside in S1.2 or S3 - if any.

Algorithm G (Grouping)

G1 [Scan buckets to be grouped] Let B be bucket number gp-t,..,gp-1 repeatedly. For each bucket do

G1.1 [Read] Read bucket B. (It can be of any type but c2, c3 and c4.)

G1.2 [Read own overflow] If B is of type c1 then all overflow buckets in the chain are read, the overflow records are removed (but saved for later!) and the buckets rewritten with the overflow space marked as empty. (This situation should be rare.)

G1.3 [Move 'foreign' overflow] If B is of type d, e1 or e2 then

G1.3.1 If B is of type e2 then link B out of the chain, follow the chain to the end and let B' <- last bucket in the chain. If B is of type d or e1 then let B' <- the predecessor to B in the chain.

G1.3.2 Select a new overflow bucket B_o. If none is found then recreate the initial situation, write a warning to the maintenance staff and terminate.

G1.3.3 Insert as many of the overflow records from B in B_o as possible, link B_o to B' and let B' <- B_o. Repeat G1.3.2-G1.3.3 until all overflow records in B are restored.

G2 [Decrement] Let p <- p-s.

G3 [Rehash] Rehash all records, including own overflow records from G1.2. (The records hash to the new buckets p,..,p+s-1.)

G4 [Rewrite] For each of the new buckets, let it be B', do

G4.1 [Rewrite primary bucket] Rewrite B'.

G4.2 [Rewrite overflow] Store overflow records from B' using the same procedure as G1.3.2-G1.3.3.

Implementation details

The overflow chain can be either a singly or a doubly linked list. Either structure has both advantages and disadvantages. The doubly linked list needs more accesses to update pointers while the singly linked list needs more accesses to find the predecessor in the list (but it is easily done, simply by hashing the key af any record and reading through the chain from the primary bucket).

The algorithms use simple solutions to the reorganisation of overflow chains. Since all buckets in a chain usually must be read anyway, a more refined solution may always minimize the number of used overflow buckets!

In order to select a new overflow bucket, all buckets with an empty overflow space can be linked together. This means fast selection but also accesses in other places to maintain the chain. If the buckets are not linked together the selection may be speeded up by maintaining a pointer to the highest bucket number known to possibly have a free overflow space.