University of Toronto
Scarborough Campus

Computer Science B28
23 January 1984

ADDITIONAL MATERIAL ON HASHING†

A SIMPLE TABLE-ASSISTED HASHING SCHEMES‡

The goal of the technique described here is to ensure that we can retrieve a record given its **key** with ***a single access*** to secondary storage. To achieve this we are willing to **sacrifice** a bit on the efficiency of the algorithm for inserting records into the file.

We shall assume that we can spare $M \cdot l$ bits of main memory, where M is the number of buckets in the file and l is the length of the key (in bits). This main memory space will be devoted to the storage of an array (table) $maxKey[0..M-1]$. The table has one entry for each bucket of the file. In general, $maxKey[b]$ will contain the maximum key presently stored in bucket b.

We shall use a hash function to generate a probing sequence of buckets $p(K,1), p(K,2), \cdots, p(K,M)$ for each key K. Any of the open addressing techniques for generating such sequences can be used. However, double hashing will probably give the best results because it avoids clustering more than linear probing and pseudo-random probing do — and clustering will make insertions less efficient, especially when the file starts getting full. As in **open** addressing, there is no overflow **area**: all records will be stored in one of the M buckets that were allocated for the file.

We want to arrange things so that when we are looking for a record with key K, we can use the following search procedure: Find the smallest i so that $K \le maxKey[p(K,i)]$. The record with key K will be in bucket $p(K,i)$, if it is in the file at all. Note that the outlined search procedure involves only one access to secondary storage — namely the access to bucket $p(K,i)$. The calculations needed to determine i only need information in the $maxKey$ table which resides in main memory.

To make the above search procedure possible, however, considerable care must be exercised when inserting new records to the file. Suppose we want to insert record R with key K. We first look in bucket $b = p(K,1)$. If there is room in that bucket, we insert the record in bucket b, update $maxKey[b]$ (if necessary), and we are done. If there is no room in bucket b, we proceed in one of two ways:

Case (a): K is greater than the keys of all records presently in bucket b. Then (according to our search criterion), R does not "belong" to bucket b. Thus, we try to insert R to bucket $p(K,2)$. If that bucket is also full and K is greater than all keys stored there, we try bucket $p(K,3)$ and so on.

Case (b): There is a record in b whose key is greater than K. Then (according to our search criterion) R "belongs" to bucket b. Since there is no room to insert R in b, we'll have to purge another record from b. In particular, we must purge the record with the maximum key presently in b, say record R'. We replace R' by R in b, we update $maxKey[b]$, and finally we re-insert R' to the file, using the same procedure.

We now give (in pseudo-code) the algorithms for inserting a record and for finding a record given its key. M denotes the number of buckets in the file and BF the blocking factor of the file, i.e. the number of records that can fit into one bucket.

† Notes by Vassos Hadzilacos.

‡ Read this section as an introduction to the more sophisticated technique described in Section 5.3 of Smith and Barnes.

```
INSERT(R)
if there are M*BF records in the file already then error (no room to insert R)
else
 i := 1
 loop
  K := key of record R
  b := p(K, i)
  fetch bucket b into main memory
  exit when b has an empty slot or b has a record with key K
  if K > maxKey[b] then i := i+1
  else
   let R' be the record in b with key = maxKey[b]
   replace R' by R in b
   maxKey[b] := maximum of all keys of records now in b
   write bucket b back into secondary storage
   R := R'
   i := 1
  end if
 end loop
 if b has a record with key K then error (attempt to insert duplicate key)
 else % bucket b has an empty slot
  put R in bucket b
  if K > maxKey[b] then maxKey[b] := K end if
  write b back into secondary storage
 end if
end if
```

```
SEARCH(K)
i := 1
loop
 exit when i > M or K ≤ maxKey[p(K, i)]
 i := i + 1
end loop
if i > M then failure (no record with key K in file)
else
 fetch bucket p(K, i) into main memory
 if the bucket contains a record with key K then success (record found)
 else failure (no record with key K in file) end if
end if
```

As a test of your understanding of the insertion algorithm explain why it is impossible for a situation to arise, where the insertion of record ***R*** ***causes*** record R' to be re-inserted, and the re-insertion of ***R'*** ***causes*** ***R*** to be re-inserted again.

Discussion: Even though searches are pretty efficient, the insertion of a record may require the insertion of another record and so on. This "domino effect" may cause long sequences of insertions to be generated as a result of a single insertion. This will be especially pronounced as the file gets full — why? This makes this technique inappropriate if insertions to the file occur frequently. On the other hand, if search operations significantly outnumber insertions (as is the case in some applications), this is a good technique to use.

Another problem with this method is the need for the main memory resident table. If the keys are long (and there are many buckets in the file), $M \cdot l$ bits of main memory might be a bit much to be asking for. There is a way around this problem. The main idea is to use the key of a record to generate (in a pseudo-random manner) a short bit string, called the record's "signature". Rather than using the keys to order the records that are hashed into the same bucket, we use their signatures. Thus, ***maxKey [b]*** contains

the maximum signature of all records stored in the bucket (which, consists of just a few bits). However, there are some problems that must be resolved: while **keys** are guaranteed to be unique, signatures are not (i.e. different records' signatures might collide). This creates problems because the signature stored in *maxKey [b]* must clearly differentiate between those records with greater signatures (and which cannot possibly be in the bucket) and those with smaller (or equal) signatures (which will be in the bucket, if they are in the file at all). The details of how this issue can be resolved are spelled out in Section 5.3 of Smith and Barnes, which you are expected to read.

HASH FILE REORGANISATION

As slots get filled, collisions increase and the performance of hashing deteriorates, because an increasing number of accesses are needed to located a record.

The traditional approach to this problem is to reorganise the file when the load factor reaches a certain threshold value. This reorganisation, called rehashing, involves the allocation of a larger number of buckets to store the file and the re-insertion of the file's records into the new, larger, set of buckets. Because the number of buckets in the file has changed, the hash function used to decide where to store the records of the file must also change.

Rehashing is a big job. Therefore (a) we should arrange things so that it doesn't have to be done too often and (b) when it becomes unavoidable, we should try to do it as efficiently as possible. To achieve the first goal, we must be able to anticipate the future growth of a file, so that sufficient storage can be devoted to it when the file is rehashed The second goal can sometimes be achieved by using the following method for rehashing.

In our subsequent discussion we shall assume that we are using the division method for hashing. Suppose that the old file contained M buckets, so that the hash function used was $h(K) = K \bmod M$.† When we wish to reorganise the hash file, we allocate an additional M buckets, so that the new tile will contain $M' = 2M$ buckets. The new file will consist of the old M buckets (which will be buckers 0 through M -1 of the new file) and the M newly allocated ones (buckets M through 2M -1 of the new file). In the new file we shall use the hash function $h'(K) = K \bmod M'$ to determine the position of a record with key K. *Notice* that

Theorem: For any natural number K, $K \bmod 2M$ is equal to either $K \bmod M$ or to $(K \bmod M) + M$.

Proof: By the fundamental theorem of arithmetic, we have that for some numbers q and r, $K = 2Mq + r$, where $0 \le r < 2M$, and for some numbers q' and r', $K = Mq' + r'$, where $0 \le r' < M$. *Thus,* $2Mq + r = Mq' + r'$ and therefore $r = M(q' - 2q) + r'$. But since $0 \le r$ c 2M and $0 \le r' < M$, it must be that $q' - 2q$ *is* either 0 or 1; and therefore, $r = r'$ or $r = r' + M$, *as* wanted □

The significance of this for the issue at hand is that when we rehash, the records that were in bucket i in the old file, will be distributed between exactly two buckets in the new file, namely buckets i and $i+M$; *in* other words, some records that were in bucket i need not be moved at all, and those that have to be moved must all be moved to the same place. This makes it possible to do rehashing fairly efficiently, keeping at most two buckets in main memory at any time and not needing to bring to main memory any bucket of the new file more than once. We could express the rehashing algorithm in pseudo-code as follows.

Suppose that the file to be reorganised presently has M buckets, $0, 1, \cdots, M-1$.

† $K \bmod M$ denotes the remainder in the division of K by M.

```
REHASH
 Allocate M new buckets for the file (buckets M, M+1, . . ., 2M - 1)
 for i: 0..M-1 do
  fetch buckets i and i+M to main memory
  for each record R in bucket i (and any overflow records that belong to i) do
   K := key of record R
   if K mod 2M = K mod M then leave R in bucket i
   else move R to bucket i+M end if
  end for
  write buckets i and i+M back to secondary storage
 end for
 M := 2 * M
```

LINEAR HASHING

One of the disadvantages of the rehashing technique described above is that it is very inflexible regarding the size of the new file: it *must* have twice as many buckets as the old file. This is a fairly drastic change in the size of the file. *Linear* hashing (not to be confused with the collision resolution technique we called "linear probing") is a method that avoids this problem by providing a smoother growth in the size of the file: the file now grows a bucket at a time. In addition, this growth is incorporated into the insertion algorithm — so that linear hashing completely eliminates the need for rehashing, at the expense of slightly slower insertions. As in the previous section we are assuming that we use the division method for hashing, that is, $h(K) = K \bmod M$.

The overall organisation of the hash file is illustrated in Figure 1. The file consists of a primary area and an overflow area The primary area consists of a number, say ***M,*** of buckets called the home buckets and some additional, up to ***M,*** buckets, called the *new* buckets. The overflow area consists of a number of slots. Each bucket in the primary can hold up to ***BF*** records and a pointer to an overflow area slot; each overflow area slot can hold one record and a pointer to another overflow area slot.

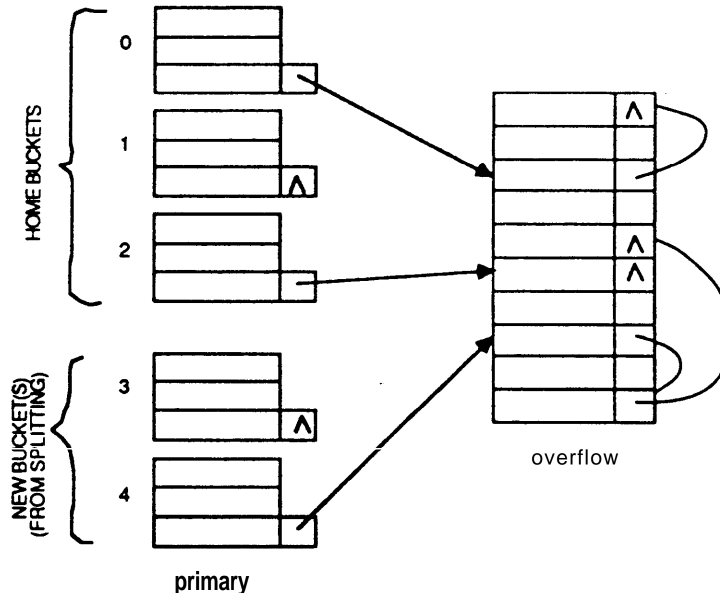

Figure 1

Initially, there are only the ***M*** home buckets in the primary area (no new buckets exist, yet). When a bucket overflows, the records that cannot be accommodated in the bucket are stored in the overflow area, forming a chain whose head is the pointer of the bucket So far, what we have described is simply hashing

with record chaining for collision resolution. Now comes the extra twist of linear hashing. Whenever a bucket overflows for the first time, a new bucket is added to the end of the primary area of the file. Suppose this is the *i*th new bucket to be added (i.e. there are i buckets that have overflowed); thus, it is bucket $M+i$ of the file. This new bucket is used to split the *i*th home bucket. Let's call buckets ***i and M + i buddies*** of each other. Splitting a home bucket involves the distribution of its records between itself and its buddy as in the rehashing technique described previously. Namely, a record with key K stays in the home bucket if $K \bmod 2M = K \bmod M$ and goes to the home bucket's buddy otherwise (i.e. if $K \bmod 2M = (K \bmod M) + M$).

Note that a new bucket is created every time a bucket overflows. However, the home bucket chosen to be split might ***not*** be the same as the one whose overflow ***caused the*** split. This is because the order in which buckets overflow is not predetermined, while the order in which they are split is. Thus, a split might not help alleviate the problem that caused it — in particular, it may result in the splitting of a bucket that is way **underfilled.** The question then arises: Why bother splitting such a bucket? The answer is that it helps, in the long run. The overflow of a bucket indicates that the load factor of the hash table starts getting high. Thus, more buckets must be allocated to reduce the load factor. This is exactly what linear hashing does. By the way, the name "**linear** hashing** comes from the **(linear)** ordering according to which the home buckets are split: **first** bucket 0, then bucket 1 and so on.

Suppose now we **are** looking for a record with key K. If the record's home bucket $h(K) = K \bmod M$ has been split, the record might be in bucket K mod M or in its buddy, bucket $(K \bmod M) + M$. How do we know where to look? Of course we could look in both places, but it turns out there is no need to do so. A simple **calculation will** tell us in exactly which of the two buckets a record with key K **will** he (if it is in the file at **all**): **Specifically,** if $K \bmod M = K \bmod 2M$, then the record is in the home bucket K mod ***M; otherwise,*** the record must be in the home bucket's buddy, i.e. bucket (K mod M) + M. For this to work properly we must take similar action when inserting a record whose home bucket has been split. Suppose K is the record's key, so that its home bucket is $K \bmod M$. If this has been split, we must decide whether the record should go to the home bucket or its buddy. We use the same criterion: If $K \bmod 2M = K \bmod M$ ***the*** record stays in the home bucket (or its overflow **area,** in case that bucket has overflowed); **otherwise,** the record goes to the home bucket's buddy, (K mod M) + M, (or the buddy's overflow area).

After all M home buckets have been split, we have a file with $2M$ buckets. We now start over the cycle of splitting, considering the 2M buckets as the home buckets, and adding new ones at the end. That is, we split buckets $0, 1, \ldots, 2M-1$ in that order. When bucket $2M-1$ is split we have a file with 4M buckets; the whole cycle is then repeated again. In effect, every time the number of buckets doubles with respect to the "original" M home buckets, we take the resulting $2M$ buckets as the home buckets, and start over again.

Below we give the insertion algorithm for linear hashing in pseudocode. We use a variable ***nextSplit*** to indicate the next bucket to be split Initially, ***nextSplit*** is set to 0. M indicates the number of home buckets in the file. Thus, the actual number of buckets in the file at any time is $M + nextSplit$. One detail that is not spelled out in the pseudo-code below is that when a bucket is fetched into main memory, we assume that all overflow records from the bucket are **also** fetched; similarly, when a bucket is written back into secondary storage, if there axe more records to be stored in the bucket than can be **accomodated,** the extra records are chained together in the overflow area with a pointer extended from the bucket to the head of that chain. With these conventions in mind we have:

```
INSERT(R)
K := key of record R
b := K mod 2*M
if b > M + nextSplit - 1 then b := b - M end if
if bucket b (or its overflow chain) contains a record with key K then
  error (attempt to insert duplicate key)
else
   if bucket b is not full then
     insert R into bucket b
   else
     put R in the overflow chain for bucket b
     if R was the first record to cause overflow of b then
     allocate a new bucket at the end of the file (this becomes bucket nextSplit + M)
     fetch buckets nextSplit (with its overflow records) and nextSplit + M into main memory
     for each record R' in bucket nextSplit do       % split bucket
       K' := key of R'
       if K mod M  =  K mod 2*M then leave R' in nextSplit
       else move R' to nextSplit + M end if
     end for
     write buckets nextSplit and nextSplit + M back to secondary storage
     nextSplit := nextSplit + 1
     if nextSplit = M then % all buckets of "original" file were split
       M := 2*M
       nextSplit := 0
     end if
   end if
  end if
end if
```

The algorithm to find a record given its key K is given in pseudo-code below:

```
SEARCH(K)
b := K mod 2*M
if b > M + nextSplit - 1 then b := b - M end if
fetch bucket b in main memory
if record with key K is found in b (or its overflow area) then success
else failure end if
```

BLOOM FILTERS

A Bloom filter is a nifty trick, based on hashing, used to avoid unnecessary searches in a differential file. Recall that differential files are used to (temporarily) store records that were recently "inserted" to a master file. If a differential file is used, to search for a record given its key,

1. We must search the differential file (sequentially). If the record is found, we are done; if the record is not found we must then
2. Search the master file (possibly using binary search). If the record is not found in the master file either, then we know that no record with the given key is stored in the file.

Step (1) is somewhat inefficient because it requires the sequential search of a file. A Bloom filter supplies a test which allows us to determine whether the differential file contains a record with a given key, without the need to search the file, or otherwise access secondary storage. The outcome of the test is either "No" (there is no record with the given key in the differential file) *or* "Maybe" (with very high probability, a record with the given key exists in the differential file). In the former case, the differential file need not be searched; in the latter case, it must be searched and most of the time a record will be found. However, there will (rarely) be "false alarms", i.e. situations in which the test says "Maybe" but in fact there

is no record with the given key in the differential file.

Let us now see how all this is achieved. The idea is to use some number of t different hashing functions

$$h_1, h_2, \cdots, h_t\colon KEYS \rightarrow \{0, 1, \cdots, M-1\}$$

where M is some number (any number will do, but the larger M, the more effective the Bloom filter will be, as we shall see). There is also a "bitmap" of size M, i.e. an array $B[0..M-1]$ of M bits, stored in main memory.

When the differential file is empty, all bits of B are 0. When a new record with key K is inserted to the differential file, we set bits $h_1(K), h_2(K), \cdots, h_t(K)$ of array B to 1. When the differential file is searched for a record with key K, we first test the values of bits $h_1(K), h_2(K), \ldots, h_t(K)$ of array B (doing so does not require access to secondary storage). If ***any*** of these bits is 0, then surely the record is ***not*** in the differential file (for if it were, all of these bits would have been set to 1) — so the outcome of the Bloom filter test is "No", in this case. If ***all*** of these bits are 1 then the record might be in the differential file — so the outcome if the Bloom filter is "Maybe'*, in this case. Such a record might not exist in the file, however, because there may exist t (not necessarily distinct) records with keys $K_1, K_2, \ldots, K_t$ in the file, such that $h_i(K) = h_i(K_i)$, for all $1 \le i$ It. This situation (which is a generalisation of a collision in hashing) is called a ***filtering error*** (or, as we called it above, a ***false alarm***).

An effective Bloom filter must have a very low probability of filtering errors. This probability depends on the following factors:

- the size M of the bitmap (the larger M, the smaller the probability of error);
- the number of hash functions (if there are too few or too many, the probability of error will increase — why?);
- the effectiveness of the hash functions themselves (they should produce as independently random values as possible).