UNIFIED DYNAMIC HASHING

James K. Mullin

The University of Western Ontario

ABSTRACT

This paper attempts to unify a variety of dynamic hashing methods. Spiral storage, linear hashing and - to a certain extent, linear hashing with partial expansions can be seen as particular cases of a more general technique. The approach is closest to spiral storage in concept. A new instantiation of the general method is offered which permits an adjustment to the dynamic growth rate during expansion. In addition, "optimal" performance results if a sufficiently accurate estimate of the file size is possible.

Introduction

Since 1979, a number of dynamic hashing methods have been discovered. Dynamic hashing methods are hash based, direct access storage methods which maintain good performance while the storage space is kept proportional to storage demand. Thus large underestimates of the space required do not result in poor performance or the need for a complete reorganization.

We assume a conventional environment where records are mapped into physical device buckets. Each bucket has space for a number of records. The operations of fetch, insert and delete are performed on records -- each with a unique identifying key.

A number of these new methods can be unified with one theory. Litwin's linear hashing [2], Larson's linear hashing with partial expansions [1], and Martin's spiral storage [3] can all be viewed as instantiations of a general approach. This general approach also suggests other useful, simple approaches. In particular, a new method is presented where:

1. The growth rate can be dynamically adjusted so as to react to observation of the actual growth demand;

2. Optimal performance will be achieved if the file size estimate is accurate. Optimal performance is attained when each bucket has the same expected number of records.

The general method is actually on adaptation of spiral storage without the exponential spiral. Many of the requirements are similar. First, a hash function which hashes the record key into (0...1) is assumed. A sawtooth function:

$$X = \lceil C\text{-hash(key)} \rceil + \text{hash(key)}$$

is employed. The parameter C is fixed by the file size. C increases as the file size increases. The range of X is always one unit from C to C+1. During file growth or contraction, the variable C is incrementally readjusted. Thus the value of X will change for only a small number of keys if the change in C is small. The sawtooth function may be seen graphically in Figure 1. A logical address is computed using a growth function:

$$\text{logical\_address} = \lfloor \text{grow}(X) \rfloor$$

In spiral storage, this growth function is an exponential b^X. "b" is some small constant greater than one. The mapping from the X space to logical bucket addresses may be seen in figure 3.

The storage space may be dynamically readjusted so as to keep storage used in proportion to storage demand. This is conveniently done by monitoring storage utilization or packing factor after update operations. It is possible to either reduce or expand the storage space. The remainder of this paper will only consider storage expansion. When it is determined that the storage space should be increased, the value of C is readjusted to eliminate the first bucket.

$$C = grow^{-1}(first+1).$$

All records in the old first bucket (left) are remapped into a new larger space on the right. In spiral storage, the inverse of the growth function is log_b (address). The reader should note that both the left and right boundaries of the file move. This is shown in Figure 2. Many operating systems would have difficulty with a file where both boundaries move. Fortunately, a simple method is available to map logical addresses to physical addresses where only the right physical file boundary moves. In summary, a physical bucket address is determined by the sequence:

key → hash(key) → X → logical address → physical address

Before describing this physical mapping, we discuss the general constraints on growth functions.

A growth function must be:

1. Continuous and 1 to 1.
2. The first derivative (slope) must exist at all but a finite number of points. Such points may only occur at a bucket boundary.
3. The slope, if it exists, must be greater at X+1 than at X for all X.
4. The inverse of the growth function must be easily computed.

The first point simply ensures a unique mapping between keys and buckets. The second point ensures that there is a computable slope for a bucket. The third point is perhaps the least obvious. This condition is in place because when one remaps a bucket, the remapping in the X space is from X to X+1. This condition ensures that the mapping is to a new larger space. The fourth point is a practical matter to guarantee reasonable speed of processing. The entire growth function need not be known in advance. It may be dynamically adjusted for sections above the last allocated bucket.

Physical Address Mapping

Martin [3] presents a method to map logical addresses to physical addresses. The general method employs an extension of his method. The general method requires that when a bucket is released, that space is immediately reused before any newly allocated space is employed.

Given a logical address y, we must determine how that address was instantiated. If the address was a newly allocated bucket, then the physical address is simply the logical address of the last bucket at the time y was instantiated, minus the logical address of the first bucket at this time. This is simply the count of existent logical buckets. If the bucket "y" was instantiated with a recycled bucket, one determines the logical address of the recycled bucket and recurs. The third possibility is that y was one of the original allocation, then physical address equals logical address. The problem is now reduced to determining how the logical address is instantiated. This is almost as simple in the general case as with spiral storage. Refer to Figure 3.

After a bucket (other than one of the initial allocation) is instantiated, if it receives remapped keys from more than ancestor, it must have been instantiated with newly allocated space. If it receives keys from only one ancestor, it must have been the first bucket needed when the ancestor was deallocated and was instantiated from its deallocated ancestor. This is true because of the condition that a bucket always maps to a larger space and the discipline of immediately reusing a deallocated bucket.

The process of determining the ancestor range is as follows:

1. Determine the X space range which maps to the bucket.

   $low\_x = grow^{-1}(logical\_address)$

   $high\_x = grow^{-1}(logical\_address+1)$

2. Since the X space between a bucket and its remapped ancestor differs by 1, the X space range of the ancestor is

   low_x-1 to high_x-1

3. The ancestor bucket address range is then

   grow(low_x-1) to grow(high_x-1).

4. If $\lfloor grow(low\_x-1) \rfloor$ and $\lfloor grow(high\_x-1) \rfloor$ are the same, then the bucket was instantiated from the logical bucket at $\lfloor grow(low\_x-1) \rfloor$. Otherwise, it was instantiated from newly

allocated space.

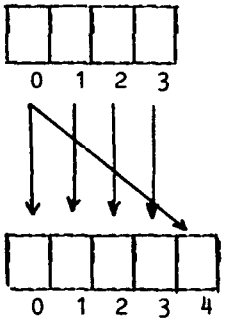
The algorithm is:

```
function physical (logical):integer;
  .
begin
 if logical <= initial_max
  then physical := logical
  else
    begin
      low_x := grow^-1 (logical)
      high_x := grow^-1 (logical+1)
      anclow := grow(low_x-1)
      anchigh := grow(high_x-1)
      if ⌊anclow⌋ = ⌊anchigh⌋
       Then
         physical := physical(⌊anclow⌋)
       else
         physical := logical-⌊anclow⌋
    end
end physical
```

A Linear Hashing Instantiation

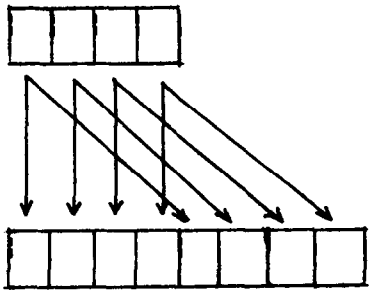
Litwin's linear hashing can be developed from the general method by the choice of a growth function as shown in Figure 4. This function is piecewise linear. "ORIG" is the initial number of buckets. Each linear segment covers a unit on the X axis. The slope of the first segment is ORIG , and the slope of the kth segment is K·ORIG . In the figure ORIG=4.

Consider the first adjustment. Bucket 0 is removed. The records will be remapped to logical buckets 4 and 5. In terms of physical mapping, since bucket 0 is removed, logical bucket 4 is recycled physical bucket 0 and logical bucket 5 is the newly allocated physical bucket 4.



In effect, the old physical block 0 is split between physical block 0 and physical block 4.

At the conclusion of what Litwin would term a "full expansion", the physical mappings will be exactly the same



as linear hashing. In addition, the section of the growth curve used to map active buckets is the second line segment. In effect, the choice of this piecewise linear growth function gives exactly the same performance as linear hashing and exactly the same physical splitting.

An approximation to linear hashing with partial expansions

Larson [1] has an important modification to linear hashing. The essence of his strategy is to map two blocks into three. This will mean that after splitting, one can expect the newly created and newly split blocks to have two-thirds of the former contents instead of half their former contents, as is the case in linear hashing. Though the bucket mappings are not exactly the same, one can achieve the same general effect with a different piecewise linear growth function. Again refer to figure 4. In this case, there is also a line segment covering each unit of the X axis. The slope of the k-th line segment is 1.5*slope of the k-1st line segment. The slope of the first line segment is "ORIG".

Now when logical bucket 0 is freed and remapped, the records are mapped to logical bucket 4 and the "bottom half" of logical bucket 5. When logical bucket 1 is freed, it remaps to the "top half" of logical bucket 5 and bucket 6. Logical buckets 4 and 6 will be the recycled physical buckets 0 and 1. Logical buckets 5 and 7 will be newly allocated physical buckets 4 and 5. Thus a 2:3 expansion is achieved. The "partial expansion" completes when all addressing is covered by the second line segment.

Spiral storage

Spiral storage is an instance of the general method directly. The growth curve is b^x . "b" is some small constant greater than one. The larger this constant, the more rapidly the space

expands. There is a simplification possible in the physical mapping routine which shortens the code needed. The lower and upper key boundaries mapping to a given address "a" are in general.

$$high\_ancestor = grow(grow^{-1}(a+1)-1)$$

$$low\_ancestor = grow(grow^{-1}(a)-1)$$

but simplify for spiral storage to:

$$high\_ancestor = (a+1)/b$$

$$low\_ancestor = a/b$$

A new dynamic hashing method

All three dynamic hashing methods presented: linear hashing, linear hashing with partial expansions and spiral storage, offer excellent performance. Successful search probes may be done with reasonable packing densities (say .75) and modest blocking of 10:1 in under 1.2 accesses on average. Thus, from this point of view, there is little to choose between them. Each has its own advantages.

Spiral storage offers constant average performance and a choice of growth rate. The choice of growth rate must be made at file initialization time. Often one needs to observe how the demand for space is proceeding before chosing a growth rate.

It can be shown that constant expected performance is achieved with spiral storage. This results from a property of the exponential which keeps the distribution of records across buckets fixed during expansion when the packing factor is kept fixed. It is an open problem whether other methods can maintain constant expected performance during file growth. One might investigate methods where only some of the performance measures such as successful search time are kept constant.

Linear hashing and linear hashing with partial expansions offer optimal performance if the initial estimation of storage space is a good one. Linear hashing grows at a rate of two blocks for each block released while linear hashing with partial expansions grows at three blocks for each two blocks released. There is thus less reorganizational activity with linear hashing as compared to linear hashing with partial expansions but poorer search performance.

In considering the general method, it is possible to remap a bucket into a new larger space without determining the growth factor at file initialization. In other words, the growth curve need only be known for those buckets currently being mapped. The unused section of the growth curve may be modified during growth. Information about experienced past growth may be used to modify the subsequent expansion rate. The proposed new method piecewise linear hashing uses this fact.

Piecewise Linear Hashing (PLH)

Piecewise linear hashing offers a dynamically controlled growth rate. It also offers optimal performance when the estimate of file size is close to the actual file size. Optimal performance is attained when all buckets have the same expected occupancy. This will be true with a linear mapping from X to bucket address which maps to only one line segment in figure 4.

PLH starts out initially with a linear mapping in exactly the same manner as linear hashing. The initial slope of the line is "ORIG". At the point when the first bucket is to be remapped, a choice of expansion rate must be made. This choice determines the slope of the next segment in the X space. A choice of slope must be be made before remapping the first bucket mapped to a line segment. A table would be kept recording the initial allocation and the slopes chosen. This information is sufficient to reconstruct the growth curve and its inverse. Such a table would typically be quite small. In the case of linear hashing, the size of the table is the number of expansions plus one.

Implementation

A simple way to implement PLH is to maintain a table containing information describing the piecewise linear growth curve. Figure 5 shows a curve and its corresponding table. Note that the slope does not need to be monotonically increasing as it is not in the figure. The only important considerations are:

The slope always increases between X and X+1 for all X, so that buckets are remapped to a larger space during file growth.

The curve be defined at every point which will be required for mapping.

Rather than have a user specify a curve, it would be preferable to request a growth rate over a range. One could then guarantee increasing slope between X and X+1. The growth function and its inverse are simply done by table lookup and interpolation.

Choosing the Growth Rate

A major advantage of this method over competing dynamic hashing methods is the possibility to adjust the rate of storage expansion dynamically. Determining the best expansion rate is not a simple task. There is a conflict between retrieval efficiency and the reorganization overhead incurred during insertion. Optimum retrieval performance is obtained when all buckets have the same expected occupancy. Bucket occupancy is inversely proportional to the slope of the growth curve at the bucket. Thus all buckets must be mapped from the same straight line segment on the growth curve. Unfortunately, expansion is only possible when a bucket maps to a larger space. Thus the slope of the new segment of the growth curve must be higher than slope of the original segment for space growth. A small expansion factor (small difference in slopes) will keep all bucket occupancies close together and result in good retrieval performance.

Opposed to this is the reorganization overhead. Space expands at the ratio of the slopes between the new section of the growth curve at a bucket and the old section. To expand space rapidly one wants a high ratio between the two slopes. Otherwise there will be bucket reorganization overhead for only a small increase in space.

The ideal situation is to use a high expansion ratio when a large number of new records are being added where the process ends with most of the buckets mapped from the same straight line segment of the growth curve. A complete analysis of how to choose the expansion rate must consider retrieval frequency, update frequency, space cost, and the order of occurrence of retrieval and update operations. Information on the time order and volume of future operations must either be known or predicted well.

Analysis

This section sketches a mathematical analysis of the general method. Numerical results are not presented as they vary with the growth curve selected. Interested readers are referred to Martin [3] or Mullin [4] for an analysis of spiral storage or Litwin [2] for linear hashing. Given that all of the dynamic hashing methods offer good search performance, growth curves which approximate one of the previous methods will also offer good search performance. The main difference in performance will lie in control over the reorganizational effort connected with bucket remapping. If one calculates a growth factor as the ratio of new file space to old file space when a bucket is remapped as r, then each time the file space increases by $r-1$, each bucket must be read and re-written to $\lceil r \rceil$ new buckets. Note that the physical address mapping ensures that one of these new buckets is, in fact, the bucket read.

The remaining analysis investigates the relationship between packing factor and expected probes to access a record. A general method will be derived to determine the expected number of accesses for a successful search and for a failed search. This is done given a packing factor, growth function and number of buckets in the file. The general method will be illustrated using PLH. The simplest conflict handling method is assumed: link chaining from the prime bucket. The overflow chains are assumed unblocked. In searching the prime bucket and each overflow record, one access is counted. Many other conflict handling strategies are possible. See, for example, Mullin [5].

The performance analysis is in three parts.

1. Given the expectations of records in the buckets, determine the probability distribution of length of overflow chains across the space.
2. Given the overflow chain length distribution, calculate the expected number of accesses for a successful search and a failed search. These are the performance criteria sought.
3. Since packing factor is commonly used as a measure of space utilization, rather than bucket expectations, calculate the bucket expectations used in step 1 from the packing factor.

The records arrive randomly and independently. Thus within any bucket the conditional probability of actually finding K records in the bucket when the expectation is E is given by Poisson's law:

$$P[Nr=K/E] = E^K e^{-E} / K!$$
$$K=0 \ldots \text{records}$$

The expectation E varies across the data space "s". The probability of a bucket containing K records across the data space is:

$$P[Nr=K] = \int_{sb}^{se} P[nr=K/E]ds$$

"sb" is the beginning of the space and "se" is the end of the space.

To actually perform the above integration, we need a relationship between storage address s and the expectation E. The expectation is inversely proportional to the slope of the growth

function s=grow(X) evaluated at the particular storage address s. This is because the expectation in a bucket is directly proportional to the fraction of the key range mapping to the bucket. The precise relationship depends on the growth function.

Given these probabilities of bucket contents, one can solve for the probabilities of overflow chains of various sizes:

$$P[\text{overflow}=K] = P[Nr=K+bc] \; ; \quad K > 0$$

where bc is the prime bucket capacity. Since the probability of various sizes of overflow sums to 1, one can compute

$$P[\text{overflow}=0] = 1 - \sum_{k=1}^{\infty} P[\text{overflow}=K]$$

The summation may be stopped at some reasonable point. Given the various probabilities of overflow, one can solve for expected accesses. The expected number of accesses for a failed search (one where the key is not in the file) is given by:

$$\text{failed\_search} = 1 + \sum_{K=1}^{\infty} K \cdot P[\text{overflow}=K]$$

The expected number of accesses for a successful search is given by:

$$\text{successful\_search} = P[\text{overflow}=0] +$$
$$P[\text{overflow}=1]\ (bc/(bc+1) + 2/(bc+1) +$$
$$P[\text{overflow}=K]\ (bc+2+3+\ldots K+1)/(bc+K)$$

We now have a means of computing expected performance in terms of bucket expectations. However, practitioners usually think in terms of packing factor over the whole area rather than individual bucket expectation. Given a packing factor, one wishes to compute expectations.

Packing factor pf is defined as:

$$pf = NREC / (bc \cdot number\_of\_buckets)$$

where NREC is the total number of records and bc the capacity of the prime bucket.

$$NREC = \int_{sb}^{se} E \; ds.$$

The number of buckets is: se-sb+1.

Thus

$$pf = \int_{sb}^{se} E \; ds \; / \; [(se-sb+1)*bc].$$

The solution of this equation for E at a bucket varies with the growth function. For PLH or linear hashing where the growth function is composed of line segments, and these slopes are constants, the solution is very simple. The expectation in a bucket is inversely proportional to the slope of the line segment at a bucket. Choose E1 to be the expectation of the leftmost bucket being considered.

$$NREC = \int E1 \; ds + \ldots + \int E1 \; S_K \; ds.$$

S_K is the ratio of slope in the leftmost bucket to the bucket considered. There is a separate integration across each line segment. The expectation within a line segment is E1 S. Thus

$$NREC = E1\ (s1-sb) + S_1 E1\ (s2-s1) + \ldots$$
$$S_K E1\ (se-sk)$$

and the E's may be computed from the packing factor.

Conclusions

A unified approach to a variety of dynamic hashing methods has been presented in terms of general growth functions. Spiral storage and varieties of linear hashing are instantiations of the general approach. A variety of growth functions are exhibited. PLH offers the ability to dynamically control the growth rate and provides a growth function with a trivially computed inverse. The main advantage of a dynamically controlled growth rate lies in the control over the tradeoff between search performance and bucket remapping work. .A choice may be made after file growth has been observed. High growth rates lead to less remapping overhead but poorer search performance. Using reasonably small line segments one can approximate any growth curve desired. Many other growth functions are possible.

REFERENCES

1. Larson, P. Linear Hashing with Partial Expansions. Proc. 6th Intern. Conf. on Very Large Databases, Montreal, 1980, 224-232.

2. Litwin, W. Linear Hashing: A new tool for file and table addressing. Proc. 6th. Intern. Conf. on Very Large Databases, Montreal, 1980, 212-223.

3. Martin, D.N. Spiral Storage: Incrementally Augmentable Hash Addressed Storage. U. of Warwick. Technical Report 27, Coventry, UK, March 1979.

4. Mullin, J.K. Spiral Storage: Efficient Constant Performance Extensible Hashing. Submitted Computer Journal.

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

Figure 1

Hash to X Mapping Using the Sawtooth Function

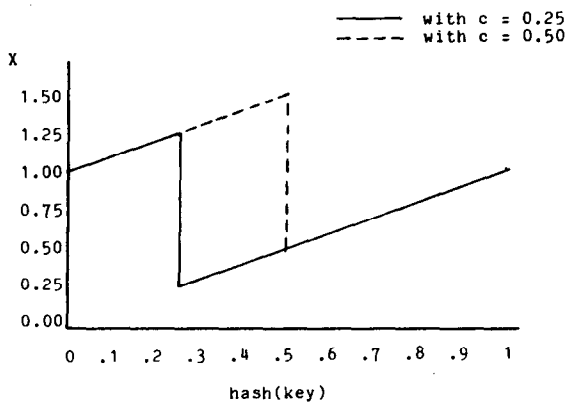
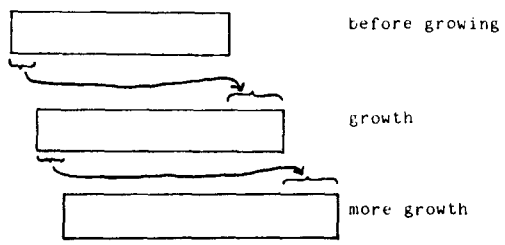


Figure 2

File Growth with a Growth Function



Notice that records move to a larger space.

Figure 3

Physical Address Mapping

Determining Ancestor Buckets

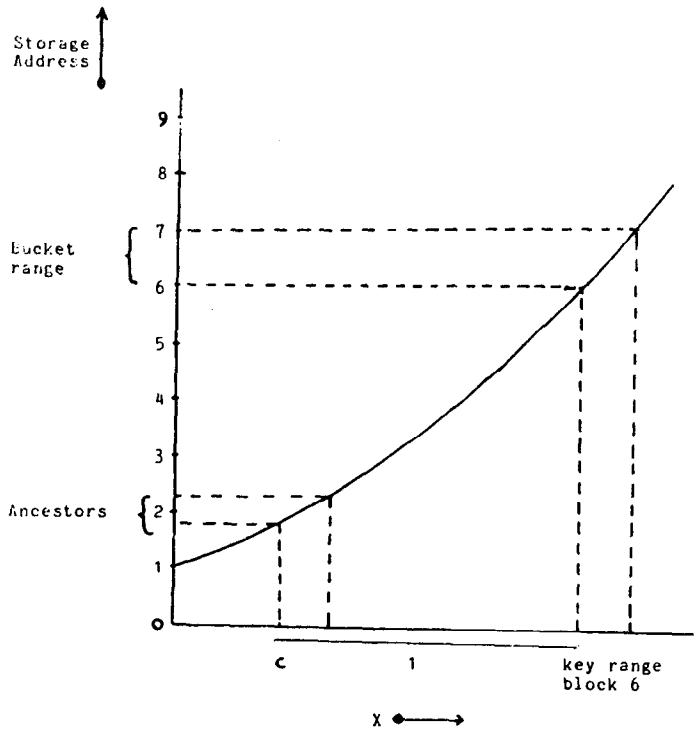


FIGURE 4

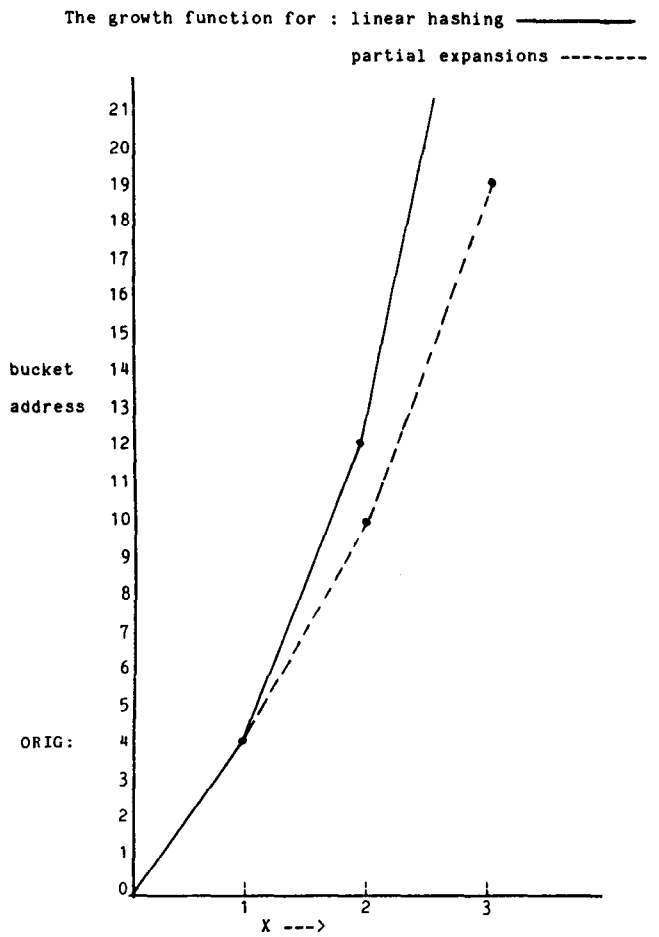


FIGURE 5