

Original citation:

Martin, G. N. N. (1979) Spiral storage: incrementally augmentable hash addressed storage. Coventry, UK: Department of Computer Science. (Theory of Computation Report). CS-RR-027

Permanent WRAP url:

<http://wrap.warwick.ac.uk/46327>

Copyright and reuse:

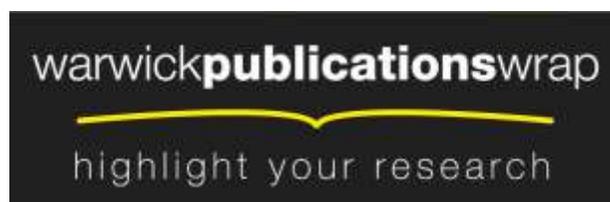
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:

The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.

For more information, please contact the WRAP Team at: publications@warwick.ac.uk



<http://wrap.warwick.ac.uk/>

The University of Warwick

THEORY OF COMPUTATION

REPORT . NO.27

SPIRAL STORAGE: INCREMENTALLY AUGMENTABLE
HASH ADDRESSED STORAGE

BY

G. N. N. MARTIN

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

March 1979

SPIRAL STORAGE: INCREMENTALLY AUGMENTABLE HASH ADDRESSED STORAGE

Several well known techniques for organising data so that they may be retrieved on some key attribute divide the data space into k equal parts and use a function that maps all possible values of the key attribute onto the integers 0 to $k-1$. When storing a data item whose key maps to n , we attempt to place the item in or near the $n+1$ 'th part of the data space.

However, as the number of data items that we wish to store increases, we must increase the size of our data space, modify the mapping function to suit the new value of k , and reorganise the items already stored in the data space. Traditionally this has been an expensive operation, and this has limited the use of hashing to special situations. This paper describes a simple mapping function that minimises the impact of reorganisation. The technique is assessed by calculating how many parts of the data space must be examined in order to read or insert a data item. Depending on the detailed implementation, we find we usually need only access the part that holds or is to hold the data item.

Errata

P8 line 2, for $(b-1)/b$
read $b/(b-1)$

P12 line 9, for $D=Cd-\frac{a}{d}C(d+1)$
read $D=Cd-\frac{d}{a}C(d+1)$

P12 line 13, for $\frac{D}{a} \frac{dr}{\ln b}$
read $\frac{D}{a} \frac{da}{\ln b}$

This paper has been submitted to VLDB Conference, March 1979.

OBJECTIVES AND REQUIREMENTS.

Let us list the possible effects of the reorganisation that we wish to promote or to avoid.

1. We should preserve the real time performance of processes accessing the data by allowing access to most of the data during reorganisation.
2. We should not require large quantities of temporary storage during reorganisation.
3. We should be able to reduce or increase the size of the data space.
4. The cost of a small increase in size of the data space should be in proportion to the cost of a larger increase.
5. The cost of accessing the data space should remain as constant as possible, provided the number of data items stored and the size of the data space are increased in the same proportion.

If we can meet these objectives, we can express the storage size of the data space as so many bytes per data item. We may then change the size of the space to match a change in the number of data items, confident that the cost of accessing the data space will remain constant. Alternatively we may increase the size of the data space in advance of an expected increase in data volume.

THE BASIC TECHNIQUE.

We map the data items into the data space so that they tend to be more dense at one end than at the other. To expand the space we extend it at the less dense end, freeing (less) space at the more dense end. Items that used to occupy the space that is freed are spread more sparsely over the new space instead. This process is illustrated in figure 1.

We shall now describe an algorithm designed to achieve this biased mapping, and draw a geometrical analogy. To start with we assume that it is always possible to store a data item in the location suggested by the mapping algorithm.

Figure 1 Illustrating the operation of a biased mapping.

```

|:|:|:|. |.:| |.|. |:|:|. | | | | |
|:|:|. |:|:|. |:|:|. |:|:|. |
|.:| |:|:|. |:|:|. |:|:|. | |.:|
|.:| |:|:|. |:|:|. |:|:|. | |:|:|. |
|.:| |:|:|. |:|:|. |:|:|. | |:|:|. | |.
|.:| |:|:|. |:|:|. |:|:|. | |:|:|. | |.:|. |
|.:| |:|:|. |:|:|. |:|:|. | |:|:|. | |.:|. | |.
|.:| |:|:|. |:|:|. |:|:|. | |:|:|. | |.:|. | |.:| |
|.:| |:|:|. |:|:|. | |:|:|. | |.:|. | |.:| | |.

```

The top line of the figure illustrates twenty four data items mapped into a data space of eleven locations using an algorithm that tends to place twice as many items in the leading locations as in the trailing locations. The items are remapped into twelve locations by removing the leading location and splitting the contents between two new trailing locations; this process is repeated until the data space consists of nineteen locations.

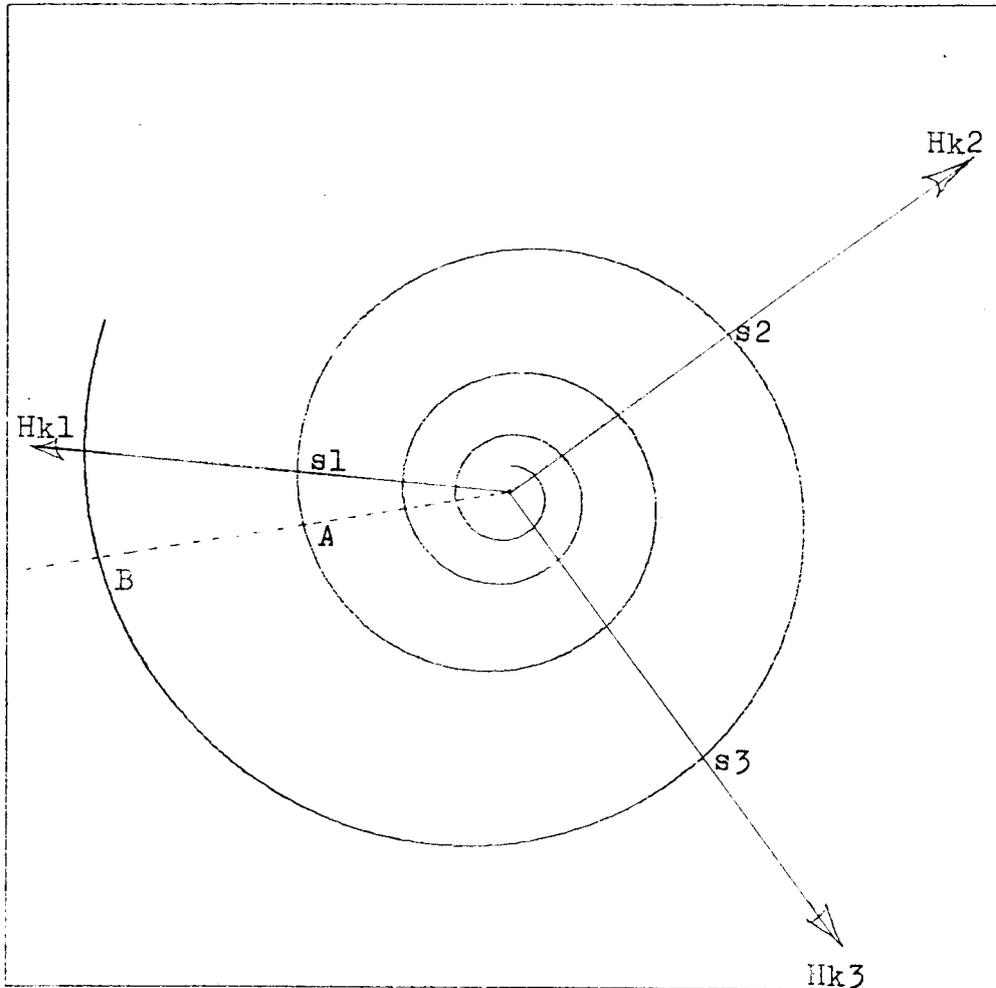
Let us define some symbols.

- K is all possible key values, $k \in K$.
- R is the real numbers, $r \in R$.
- J is $[0, 1)$, $j \in J$. In other words, $0 \leq j < 1$.
- $\{r\} = r - \lfloor r \rfloor$.
- H is a hash function, mapping K onto J 'at random'. As usual, if we can choose an H that maps K onto J more evenly than 'random', we may profit by it.
- Juxtaposition binds functions to their arguments, and we use the lambda notation thus $H = \lambda k.Hk$.

In our analogy we interpret Hk as a fraction of one revolution, and call Hk the direction of the data item with key k . To map directions to locations, we imagine the storage medium laid out along a spiral: the data space occupies one complete revolution of the spiral, and so each direction maps to one (and only one) location in the data space. This is illustrated in figure 2.

Figure 2

Illustrating spiral storage.



The storage medium is considered to lie along an exponential spiral, unit length of the spiral representing one storage location. The data space occupies one complete revolution of the storage, from point A to point B, thus every direction maps to exactly one location in the data space. Three data items have been shown, with keys k_1 , k_2 , and k_3 ; their directions are Hk_1 , Hk_2 , and Hk_3 , and the locations at which they are stored are labelled s_1 , s_2 , and s_3 .

Choosing arbitrary origins, we say that our data space starts after c revolutions, $c \in \mathbb{R}$, and that the first location of the data space is $[b^c]$, $b \in \mathbb{R}$, $b > 1$. We shall call c the angle of the data space. We shall call b the growth factor, because the data space grows b times as large each time the angle increases by one. Given c , the angle at which we must store a data item with direction j is given by $G = [c-j] + j$, and so the location at which that item is stored is $[b^G]$. The address of this item relative to the start of the data space is $[b^G] - [b^c]$.

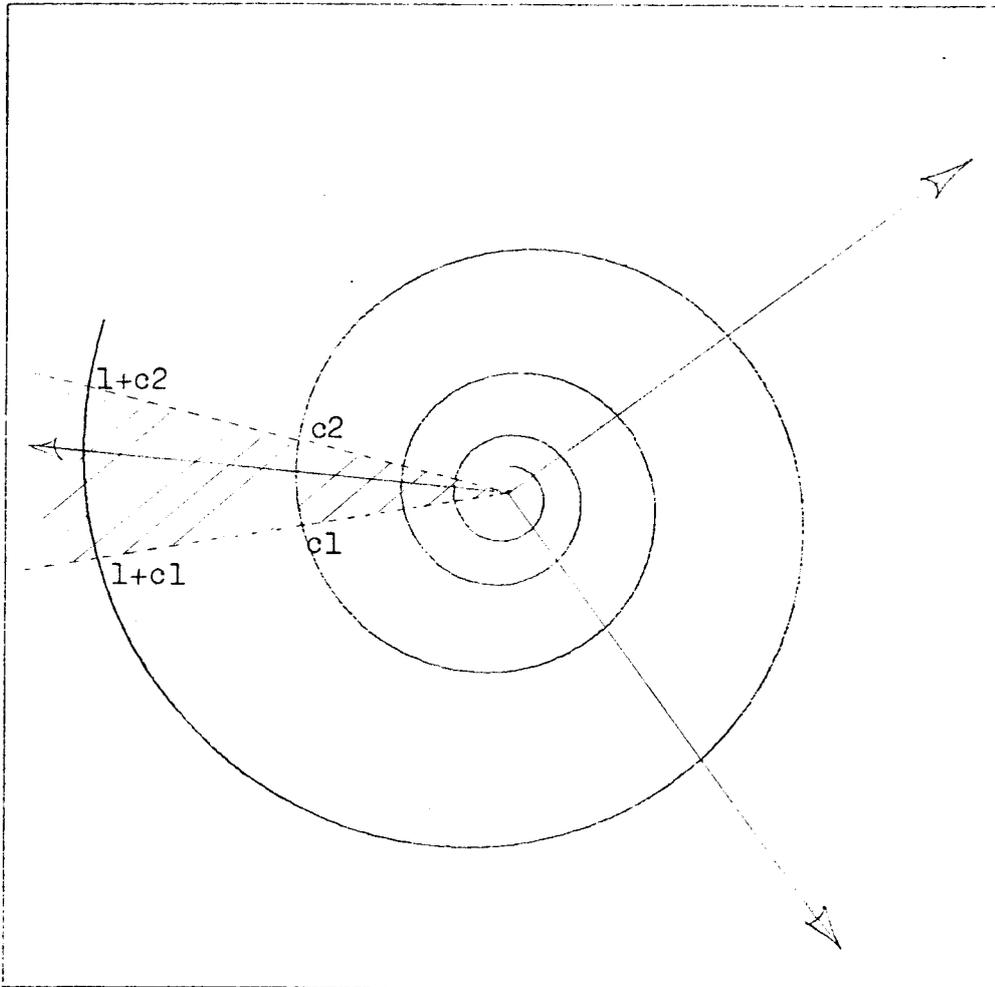
The data space contains $[b^{c+1}] - [b^c] = (b-1)b^c$ locations. If we wish to expand our data space, we increase c : if we wish to contract our data space we decrease c . If we increase c from c_1 to c_2 the data items whose directions lie in the sector $[c_1, c_2)$ will be remapped into an area b times as large as they were formerly allocated. This is illustrated in figure 3.

We may now see that we have met our objectives closely. Taking them in order:

1. If we expand the data space by a small amount, say by a single location, we need only halt access to those locations in the sector being expanded.
2. While expanding the data space by a small amount, our intermediate storage requirements are only slightly larger than our final storage requirements.
3. We can expand or contract the data space by increasing or decreasing the angle c .
4. If we wish to expand the space by a given amount (less than one revolution), we can do so in small increments at no greater cost.
5. If we increase the size of the data space and the number of data items by the same percentage, then we can expect the distribution of data items to be similar in both cases (this is a property of the exponential spiral). It follows that the average number of accesses for any given operation will be the same.

Figure 3

Illustrating spiral storage.



The data space is located in the storage medium by the angle c at which it starts. If c increases from c_1 to c_2 the data space is expanded, the arc from c_1 up to c_2 being replaced by the arc from $l+c_1$ to $l+c_2$. This involves moving all and only those data items whose directions lie in the shaded sector.

SPACE ALLOCATION

In some environments it will be inconvenient to allocate space in units as small as a single location. We may prefer to allocate increments of (say) one eighth the size of the active space, so ensuring that a data space with a growth factor of two could be described as about twelve lumps of contiguous storage. We waste space if we do so of course, but we can still spread the cost of the actual reorganisation, and we can still proceed with a smaller allocation of space if that is all that is available.

More significantly, our algorithm requires that both boundaries of the data space be able to move: while space is being allocated at one end it is being freed at the other. This may be inconvenient, since many existing storage management schemes only permit one boundary to move. We may avoid this if we choose an appropriate mapping from locations to addresses. Instead of freeing the storage associated with a location as soon as it is not active, we reuse it to hold a new location as soon as we safely can.

To explain this second mapping, we will introduce two complementary functions. S is a function that maps angles to locations, thus

$$S = \lambda r. \lfloor b^r \rfloor$$

T is a function returning the lowest angle that maps to a location, thus

$$T = \lambda s. \log_b s$$

Now the lowest numbered location that contributes to location s when it is formed is $S(Ts-1)$. If the location $S(Ts-1)$ will be inactive by the time location $s+1$ is active then we can site location s at the same address as location $S(Ts-1)$, otherwise we must site location s at a newly allocated address. Since there will already be $s - S(Ts-1)$ active locations, that new address will be offset $s - S(Ts-1)$ within the data space. We can now construct a function mapping locations to address. Location s will map to offset A_s within the data space, where

$$A = \lambda s. S(T(1+s) - 1) > S(Ts - 1) \rightarrow AS(Ts - 1), s - S(Ts - 1)$$

$$= \lambda s. \left\lfloor \frac{1+s}{b} \right\rfloor > \left\lfloor \frac{s}{b} \right\rfloor \rightarrow A \left\lfloor \frac{s}{b} \right\rfloor, s - \left\lfloor \frac{s}{b} \right\rfloor$$

$$= \lambda s. \left\{ \frac{s}{b} \right\} > \frac{b-1}{b} \rightarrow A \left\lfloor \frac{s}{b} \right\rfloor, s - \left\lfloor \frac{s}{b} \right\rfloor$$

We may thus map locations to addresses in an average of $(b-1)/b$ divides and tests; table 3 illustrates this constant origin mapping. The constant origin mapping has one disadvantage, namely that the data items are no longer stored in the natural order of their directions. Thus (for example) if we increase the data space by several locations, then the blocks that must be read are not physically contiguous in the storage medium.

 Table 1 Mapping locations to constant offsets.

This illustrates the mapping of locations to addresses that allows us to alter the size of our data space by adjusting one boundary only. The addresses are expressed as offsets within the data space. At any one time only one location at any offset is active, except that the first and last active locations may map to the same offset. This does not mean that that offset can ever be overloaded, since in such cases the two locations cover the same directions.

The table is calculated for a growth factor of 1.4.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12
Location	2	3											
	4	5											
	6	8	7										
	9	12	11	10									
	13	18	16	15	14	17							
	19	26	23	22	20	25	21	24					
	27	37	33	32	29	36	30	34	28	31	35	38	
	39				41		43		40	44			42

Typical data spaces will consist of the following active locations:

Location 2	(one location only)
Locations 5 6 7	(three locations in all)
Locations 25 through 35	(eleven locations in all)

ESTIMATING PERFORMANCE

Collisions

A data item that will not fit into its preferred location is displaced. The discovery that an item must be displaced is called a collision. We shall need a collision strategy for storing and locating displaced items.

We shall assume the cost of any operation on the data space to be proportional to the number of locations within the data space that are accessed in performing that operation. By this criterion, the performance of our storage strategy depends on the cost of operating on displaced items. The number of displaced items will depend on the space utilisation, that is, the number of data items that are stored in the data space expressed as a fraction or a percentage of the maximum number of data items that can be fitted into that size of data space. Similarly, for a given space utilisation there will be more displaced items for larger growth factors, since the distribution of those items will be more biased. Thus when describing performance we must quote the space utilisation and the growth factor.

One common way of reducing collisions is to take a unit of storage large enough to hold several data items, and address it as a single location. Data is in any case generally blocked in this fashion for other reasons. The hash addressing algorithm identifies the location in which an item should be stored, and some other method is used to address the item within its location. The number of data items that will fit into a single location is called the blocking factor (d). With a blocking factor of the order of 100, a growth factor of about 2, and a utilisation of 50%, collisions will be so infrequent that we can assume all operations will take a single access. We illustrate our formulae with calculations based on a growth factor of 2, a blocking factor of 10, and a utilisation of about 70%. Under these circumstances operations average between 1.05 accesses and 1.35 accesses.

Assumptions

There are a number of implementaion options, of which we consider the most straightforward. Thus we assume a collision strategy that requires $H \text{ map } K \text{ onto } J$ so that $Hk=j, j', j'', \dots$: we attempt to store an item using direction j , and if there is no room in that location, under location j' , etc. It may be possible to improve on this by exploiting

the knowledge that there is more likely to be free space at the higher angles of the data space.

We assume that we read an item by following the strategy that was used to insert it. We could improve on this by (for example) keeping pointers to displaced items within the location from which they are displaced.

We assume optimistically that the distribution of displaced items within the data space is such as might be expected if the space were recreated by inserting all the items into an initially empty space of the same size. This is a common assumption when predicting the performance of hash addressed spaces, but can not safely be made unless collisions are very rare, or unless we take steps to migrate displaced items to their preferred location when space becomes available.

Finally, we assume that variables associated with a location can be described by continuous functions and integrated, which is acceptable if there are many locations. Thus we take $s=b^r$.

More symbols and identities.

If we expect 'a' occurrences¹ of a random event in any given interval, then the probability that we shall in fact find x occurrences is Bx , where

$$B = \lambda x \cdot \frac{a^x e^{-a}}{x!} \quad (\text{Poisson's law})$$

'a' is called the expectation in that interval. The probability that we shall have d or more occurrences is Cd , where

$$C = \lambda d \cdot \sum_{x=d}^{\infty} Bx$$

In most cases we are interested in the expectation of the number of items whose preferred directions will map to the location about a given angle, in which case we shall simply refer to this as 'the expectation'. So if 'a' is the expectation and d the blocking factor, then Cd is the probability that as many (or more) items will prefer a given location as it takes to fill that location. B and C have been tabulated [Molina] for $0.01 \leq a \leq 100$.

Over the data space, the expectation is proportional to the

¹The symbol 'a' is enclosed in quotes in the text for readability, to distinguish it from the indefinite article.

change of angle across a location, and hence inversely proportional to the location. In other words:

$$s=b^r \quad \text{and for } r \in [c, c+1), \quad a \propto \frac{dr}{ds}. \quad \text{Hence } a \propto \frac{1}{s}.$$

It follows that

$$\int_c^{c+1} \lambda dr = \int_{\frac{a_1}{b}}^{a_1} \frac{\lambda da}{a \ln b}$$

where a_1 is the expectation at the angle c (and hence the highest expectation in the data space). This helps us to integrate over all data items those values that are expressed for an individual item in terms of the expectation at the corresponding angle in the data space.

Space utilisation

We wish to quote the space utilisation at which our performance predictions will be achieved, but those figures will usually be calculated in terms of a_1 . Thus we must calculate the one in terms of the other.

The total number of items in the data space is the sum of the expectations in each location, while the maximum capacity of the space is the product of the number of locations in the space and the blocking factor.

Thus the utilisation is

$$U = \int_{b^c}^{b^{c+1}} a ds \div d(b-1)b^c$$

Since $a \propto \frac{1}{s}$ then $\frac{ds}{b^c} = -\frac{a_1 ds}{a^2}$ and so

$$U = \int_{\frac{a_1}{b}}^{a_1} \frac{a_1 da}{a(b-1)d} = \frac{a_1 \ln b}{(b-1)d}$$

Thus (for example) if the growth factor of the data space is $b=2$, and the highest expectation equals the blocking factor ($a_1=d$), then the space utilisation will be $\ln 2=69\%$. Note that this assumes that displaced items are stored in the data

space.

Percentage of items displaced.

If the expectation at a location is 'a', and the blocking factor d, then the expected displacement from that location is

$$\sum_{n=d}^{\infty} (n-d) B_n$$

$$= aCd - dC(d+1)$$

Thus the probability that an item mapping to that location is displaced is

$$D = Cd - \frac{a}{d}C(d+1)$$

Since items are mapped at random over $[c, c+1)$, the percentage of items that are displaced will be \bar{D}_a , where

$$\bar{D} = \lambda a_1 \cdot \int_c^{c+1} D \, dr$$

$$= \lambda a_1 \cdot \int_{\frac{a_1}{b}}^{\frac{a_1}{a}} \frac{D}{a \ln b} \, dr$$

Note that this assumes that we do not allow an item to be displaced unless its preferred location is full of undisplaced items, rather than just full of items some of which are themselves displaced.

\bar{D}_a may be evaluated numerically. Table 2 works an example with $d=10$ and $b=2$, where we find that $\bar{D}_a = 4\%$. We have already shown that $b=2$ and $a_1=d$ implies a storage utilisation of 69% if displaced items are stored in the data space, we now see that the utilisation is 66% if displaced items are stored elsewhere (since $.69 \times .96 = .66$).

Accesses per successful read.

If we insert an item that maps to a full location, we must

attempt to insert that item (or another from that location) in another direction. If the alternate direction is random then all those directions that are used in attempts to insert an item will be spread at random over one revolution. Thus the expectation of such attempts in any given location (the access expectation) will be proportional to the expectation (of preferred attempts) at that location. The number of items inserted in the data space is the sum of the expectations at all locations, while the number of accesses needed to insert those items is the sum of the access expectations at all locations. Thus if the access expectation at angle c is a_2 then we require a_2/a_1 accesses per item to insert all the items, and if we follow the same strategy when reading as when inserting, then we will need on average a_2/a_1 accesses per successful read.

To calculate a_2 , consider that $\bar{D}a_2$ of the accesses will fail to read/insert an item, and so $1-\bar{D}a_2=a_1/a_2$. Table 2 shows that in our chosen example we can expect to make a successful read in an average of 1.05 accesses.

Accesses per unsuccessful read.

We must keep extra information in the data space if we are to limit the cost of attempting to read an item that is not in fact in the data space. This problem is not peculiar to spiral storage, so will not be considered here.

Accesses per insertion.

If displaced items are not stored in the data space, then the probability that a location is full is $\bar{C}a_1$, where

$$\begin{aligned}\bar{C} &= \lambda a_1 \cdot \int_c^{c+1} C d \, dr \\ &= \lambda a_1 \cdot \int_{\frac{a_1}{b}}^{a_1} \frac{C d}{a \ln b} \, da\end{aligned}$$

If displaced items are stored in the data space then the probability that a location is full is $\bar{C}a_2$. $\bar{C}a_1$ and $\bar{C}a_2$ may be calculated numerically (a_2 has already been calculated above).

Table 2 shows that at 69% utilisation, $d=10$, and $b=2$, $\bar{C}a_1=22\%$, so at least 22% of new items will be displaced, and $\bar{C}a_2=25\%$, so that if displaced items are stored in the data space, then a further 3% of new items can only be stored in their preferred locations by displacing an already displaced item from that location. Again, if displaced items are stored in the data space it will take around $(1-\bar{C}a_2)^{-1}$ accesses to insert a new item, and $\bar{C}a_1 - \bar{D}a_1$ of the items will move a second time to migrate to their preferred location. In the example in table 2, these figures work out at 1.33 accesses, and 18%.

Note that if our access method wishes to enforce that no two items in the data space should have the same key, then inserting an item must include an unsuccessful read. The cost of this has not been added in. Note also that if items are inserted into the data space and no items are deleted, the data space being expanded as necessary to keep the space utilisation constant, then we must also consider that each insert incurs ultimately the cost of adding $(dU)^{-1}$ of a location to the data space.

FURTHER CONSIDERATIONS.

Approximating exponentials

Spiral storage requires that we evaluate b^r , and (occasionally) $\log_b s$. This is easier and cheaper than we might expect, since we may use the approximation $b^r \approx F(\log_2(b)r)$ where

$$F = \lambda x \cdot 2^{\lfloor x \rfloor} \left(\frac{8.75}{3.5 - \{x\}} - 1.5 \right)$$

If we do so, then the expectation at any angle in the data space will be less than the expectation that would be achieved if we were to use an exact value for b^r with a 1% higher utilisation. Usually we start with a performance requirement, and this will lead us to choose a particular utilisation: if we plan to use the above approximation then we will meet our performance specification if we use a data space 1% larger than our formulae predicted. This rule of thumb would drop the utilisation in our example in table 2, from 69.3% to 68.6%.

The function F is readily inverted. If $Fx = 2^{ny}$ where n is integral and $1 \leq y < 2$ then

$$x = n + 3.5 - \frac{8.75}{1.5 + y}$$

Table 2: Performance calculations for spiral storage.

This computes the cost in accesses of various operations on a space with growth factor $b=2$, utilisation $U=69\%$, blocking factor $d=10$.

Highest expectation $(a_1) = .69d/\ln 2 \approx 10$.

Lowest expectation $= a_1/b \approx 5$.

a	Cd	$C(d+1)$	D	D/a	$(Cd)/a$
5	.03183	.01370	.0044	.0009	.006
6	.08392	.04262	.0129	.0021	.014
7	.1695	.09852	.0288	.0041	.024
8	.2834	.1841	.0533	.0067	.035
9	.4126	.2940	.0859	.0095	.046
10	.5421	.4170	.1251	.0125	.054

$$\bar{D}a_1 = \int_{\frac{a_1}{b}}^{a_1} \frac{D}{a} da \approx 4\% \qquad \bar{C}a_1 = \frac{1}{\ln b} \int_{\frac{a_1}{b}}^{a_1} \frac{Cd}{a} da \approx 22\%$$

By estimating $\bar{D} \approx 10$ we find

$$a_2 = 10.47 \qquad \bar{D}a_2 \approx 4.5\% \qquad \bar{C}a_2 \approx 25\%$$

Thus a successful read will average $1/(1-\bar{D}a_2) = 1.05$ accesses, adding a new item will average $1/(1-\bar{C}a_2) = 1.33$ accesses, displaced items will average $\bar{D}a_1 = 4\%$, and migration will cause an average of $\bar{C}a_1 - \bar{D}a_1 = 18\%$ of inserted items to move from their initial positions.

The optimum values for the constants in F are less memorable, and only trivially more effective.

Migration

We have assumed that displaced items will be migrated to their preferred locations when space becomes available, and we have predicted the data movement implied by such migration. We shall now make crude estimates of the effect of not migrating

data items.

If the actual distribution of data items through a data space is similar with or without migration, then the probability of a new item mapping to a full location remains $\bar{C}a_2$, and the cost of insertions therefore remains $(1-\bar{C}a_2)^{-1}$ accesses. The probability that a new item is displaced is $\bar{C}a_2$, or less if we choose to move a displaced item from a full location when attempting to insert a new item. As the data space ages or grows the fraction of the items in the data space that are displaced will approach this probability, so the number of accesses required to read an item that is in the data space will rise to not more than the number of accesses required to insert an item.

In fact the actual distribution of the data items will not be exactly similar with and without migration. The data items will be spread slightly more evenly, and so the cost of operations will be slightly lower than predicted above.

CONCLUSION

While preparing this paper the author has received a paper describing another scheme for providing an extendable hash addressed space [EXHASH], and has so far failed to obtain papers describing two further schemes [Larson] & [Litwin]. The appearance of three other independently prepared papers reduces the need to justify our interest in the technique.

Spiral storage permits us to ensure that a hash addressed space does not become overloaded as long as the physical storage medium is not overloaded, and at very small cost. This should help us to exploit the natural superiority of hash addressing in general purpose access methods.

Acknowledgements

I am grateful to my employers, IBM, who have assigned me to Warwick University (England) in order that I might write this and other papers, and to the IBM Midland Marketing Centre for their word processing facilities. I am particularly grateful to my supervisor at Warwick University, Dr. M. Paterson, for his great help and persistent constructive criticism.

References

- EXHASH:Extendible hashing, R. Fagin, J. Nievergelt, N. Pippenger, H. Strong, IBM Research report number RJ2305(31047) July 1978
- Larson: Dynamic Hashing, P. Larson, BIT 18 (1978) 184-201
- Litwin: Virtual Hashing, Proceedings of Very Large Data Base conference, 1978.
- Molina: Poisson's Exponential Binomial Limit, E. C. Molina, Robert E. Krieger publishing company, 1973.