LBL-19373
c2

Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

RECEIVED
LAWRENCE
BERKELEY LABORATORY

APR 25 1985

LIBRARY AND
DOCUMENTS SECTION

Computing Division

To be presented at the SIGMOD 1985 Conference,
Austin, TX, May 28-31, 1985

MODIFIED DYNAMIC HASHING

K. Kawagoe

March 1985

TWO-WEEK LOAN COPY

This is a Library Circulating Copy which may be borrowed for two weeks.

LBL-19373
c2

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

LBL-19373

Modified Dynamic Hashing

Kyoji Kawagoe

Computer Science Research Department
University of California
Lawrence Berkeley Laboratory
Berkeley, California 94720

March, 1985

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, United States Department of Energy under contract DE-AC03-76SF00098.

Modified Dynamic Hashing

Kyoji Kawagoe

C&C Systems Research Laboratories
NEC Corporation
Miyamae,Kawasaki,Kanagawa 213,Japan

On leave from NEC Corporation until July 1985
Current address: Computer Science Research Department
Lawrence Berkeley Laboratory
University of California

[**Abstract**]

This paper describes a modification for the unified dynamic hashing method presented by J. K. Mullin. The main advantage of this modified dynamic hashing method is that it provides a single file access to a record, while the unified dynamic hashing method may require several accesses for records in buckets that overflowed. This method is spatially efficient because it does not use indexes or tables commonly used by other dynamic hashing methods.

1. Introduction

Since the late seventies, there have been much work on dynamic hashing methods. Dynamic hashing methods make it possible to expand the hashing space dynamically by changing the hashing function. The hashing functions are chosen such that the effect on data locations is minimized.

The features of dynamic hashing methods are:
1) The file space can dynamically be changed as the number of records increases. That is, even if the estimate on the needed file space is incorrect, the file never overflows. Furthermore, dynamic hashing methods provide fairly dense file structures even in the case of high updating activities.

*This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, United States Department of Energy under contract DE-AC03-76SF00098.

2) Dynamic hashing methods access a record in a file more efficiently than a B-tree, because they do not require the additional index information that is necessary in B-tree methods. In practical applications, a record can be usually retrieved in a single logical access when using dynamic hashing methods.

Because of these features, dynamic hashing methods maintain good access performance even if the space required is largely underestimated. [Mull84]

Kjellberg and Zahle [Kjel84] have categorized many existing dynamic hashing methods into two classes. The first class of dynamic hashing has a file space expansion operation when an overflow occurs. This class includes Extendible Hashing [Fagi79], Expandible Hashing [Knot71] , Virtual Hashing [Litw78] and Dynamic Hashing [Lars78]. To maintain the relationship between split buckets and the remaining buckets, either tree structures or tables are usually used.

The second class of dynamic hashing avoid splitting until all overflow space is used. Therefore, an overflowed bucket is not split, but the records are stored in another bucket which has available space. Then, in the case that there is no more space in a file, some buckets are split into a larger address space. No index information is necessary for this class of hashing methods. This class of hashing includes Spiral Storage [in Mull84], Linear Hashing [Litw80], Linear Hashing with Partial Expansions [Lars80] and Unified Dynamic Hashing [Mull84]. The unified dynamic hashing employs a piecewise linear function for the file space growth function.

In this paper, we present a modified dynamic hashing method that combines the advantages of the first and the second classes described above. The first class has the advantage of providing a single file access to every record, but at the cost of using an additional index. The second class does not use an index, but requires more than one access for buckets that overflowed. In the method described here, we achieve always a single file access without using an index. This is achieved by modifying the unified dynamic hashing method [Mull84], as will be explained in the following section. In addition, we show that this method that space utilization is at least as good as the other dynamic hashing methods.

Section 3 describes the data organization and access algorithm of the proposed method. In section 4, the simulation results for the file load factor of the method are presented and then discusses the performance of the method and some issues on the model.

2. Background information on the unified dynamic hashing

The unified dynamic hashing method that Mullin proposed can represent all the methods in the second class of dynamic hashing. According to his representation, dynamic hashing methods are expressed by the following:

Key -->Hash(Key)-->X-->Logical bucket address-->Physical bucket address,

where Hash(key) is an arbitrary hashing function of keys and whose domain is [0, 1). The value of X is derived from the function: X=⌈c-Hash(Key)⌉+Hash(Key), where c is a parameter dependent on the particular method used. The mapping from the X values to the logical space is represented by a function called 'growth function'.

Fig.1(a) shows the behavior of the X-function for different values of c. Note that as the value of c changes, the interval of X values stays, but the starting and ending values are c and c+1, respectively. Therefore, as the value of c increases, the region of X has larger values shown in Fig.1(b). An example of a growth function is shown in Fig.1(c). As can be seen in Fig.1(c), the growth function permits the range of X to grow as the value of the parameter c increases. The effect of this function is therefore to increase the logical space dynamically. A growth function should meet the following requirements: a) it should be continuous and one to one, b) the first derivative at all points should exist except in a finite number of points which occurs at bucket boundaries, c) the slope has to be greater at x+1 than at x, where x and x+1 are elements of the region X, and d) inverse value calculation should be simple [Mull84]. The main reason for the above requirements is that the growth function is used not only for the mapping of X-value-to-logical address but also for the expansion of a logical file space.

Mullen pointed out in his paper that the physical address is uniquely determined from the corresponding logical bucket address. The ease of mapping between logical and physical addresses is an important feature of the second class of dynamic hashing. He also makes it clear that the difference among the dynamic hashing methods in the second class lies in the selection of the growth function. Spiral storage [in Mull84] uses a form such as b^X as its growth function. Linear hashing [Litw80] uses a set of linear functions such as aX+b.

The unified dynamic hashing is a way of viewing dynamic hashing, which gives a useful insight on selecting growth functions. Moreover, it provides new usage of dynamic hashing.

3. The Modified Dynamic Hashing

Unlike the unified dynamic hashing method which splits several buckets when the file is full, The modified dynamic hashing method splits only the bucket that overflowed. The idea of the modified dynamic hashing is illustrated in the Fig. 2. Note that the buckets which are on the right side of the split bucket have a larger logical address after splitting. However, the above structure is not practical. The reason is that it is impossible to determine a new hashing function without

changing the physical address of the buckets on the right. An alternative to this ideal bucket splitting is presented in Fig 2(b). First, all the buckets on the left side of the bucket that overflows are logically moved toward higher addresses in a logical space without changing their physical addresses. They are placed logically at the end of the current logical space. Then, the bucket which overflows is split into the two. One of the two uses the same physical area where the bucket overflows. The other is physically created. Both are logically placed at the end of the logical space.

A more detailed description of this method is presented in the example shown in Fig.3. Fig.3(a) shows the set of records which have to be stored in a file. Initially, it is assumed that the file has no records. The file is used for a secondary access of an employee master file using the salary values. Fig.3(b) explains the symbol and notation for the next figures. Fig.3(c) shows a sequence of buckets in the file according to the record insertion operation. The records are stored in the order shown in Fig.3(a). First, two records are stored in the bucket 1 which is the only bucket in this file (Fig.3(c)-1 to -3). Then, we assumed that this bucket is full after these two records were inserted. The new logical address space is created by changing the value of the parameter c mentioned in the previous section. Therefore, before the third record is stored, a new bucket is generated. After generating the new bucket, the two records in the overflow bucket are split (Fig.3(c)-4). The physical bucket address is determined by the X-value of the record in the splitting operation. In this case, the new bucket generation causes the value of c to change from zero to one. Then, the X value at the bucket boundary becomes 1.5 which is the intermediate value of the X value region in the overflow bucket. According to the X value of each record in the overflow bucket, the record is stored in the appropriate bucket. The logical addresses of the buckets which are generated by splitting are 2 and 3. Note that the physical address of the split bucket stayed the same (i.e. 1), while its logical address is changed from 1 to 2. The additional bucket has therefore physical address 2 and logical address 3. After that, new record can be stored in one of the buckets (Fig.3(c)-5). Fig.3(c)-6 shows the second splitting, since the bucket with logical address 3 overflows. This splitting causes a change in the logical addresses of the buckets whose addresses are lower than the overflow bucket address. Therefore, the address of bucket 2 is changed into the previous maximum plus one, that is 4. The new logical address space becomes [4, 6] after splitting, because the overflow bucket is split into two bucket whose addresses are 5 and 6 similar to the first splitting. The value c is also changed into 2.0 because the region of the overflow bucket X-values is [1.5, 2.0] and c corresponds to a boundary point between the old logical address maximum and the generated address minimum. The record insertion operation continues in this manner. The sequence of the logical addresses whose buckets overflow are kept as control data. In this example, this split sequence is { 1, 3 }. By keeping the trace of the split history, it is possible to calculate the physical address from an X-value which is obtained from a record key (a salary value in this example). Meanwhile, the direct representation of the growth function as shown in Fig.3(d), can be avoided. An example of calculating the physical address from the X-value is given in the section 3.5 below.

Basically, this way of splitting uses the idea of the unified dynamic hashing proposed by Mullen [Mull84]. The piecewise linear functions used in [Mull84] are also used here as a growth function and the same saw-tooth function is used for deriving the X value. However, the main differences between the method presented here and the unified dynamic hashing method are: 1) the representation of the growth function is more spatially efficient because it does not use a table but only a split sequence, 2) splitting is done dynamically for one bucket at a time, rather than many buckets when the file is full.

In the unified dynamic hashing, the set of pairs (X, logical address) where the first derivative is not continuous, have to be stored in some table form. The number of pairs depends on the storage utilization. The above points of (X, logical address) correspond to the break

points of the space expansion rate. Therefore, the less space is available for the table storage, the less dynamic the storage expansion is, thus defeating the feature of dynamic hashing. This situation occurs because a file has a large amount of unused area immediately after space expansion. On the other hand, in the modified dynamic hashing presented in this paper, This phenomena is avoided, because buckets are split dynamically one at a time. The logical addresses of split buckets are stored in a sequence of which is used for determining a physical address from key values. The length of the sequence increases linearly with splitting requirements. However, the values of each element in the sequence are all integers and monotonically increasing. Therefore, some compression technique can be employed to reduce more the storage space for the sequence.

The other difference is the storage utilization. The unified dynamic hashing belongs in the second class of dynamic hashing. So, file space expansion is made when the file becomes full. The overflow records are stored in another bucket when the bucket has an area available for storing these records. However, in the modified dynamic hashing, a bucket which overflows is split at the time. Therefore, file control mechanism becomes simple and easy to implement. It is worth noticing that the splitting method is similar to B-tree techniques. This means that the behavior of splitting is also similar to that of a B-tree except that one splitting operation does not affect the splitting of another bucket.

Next, the process of the modified dynamic hashing is described in more detail. First, some terms are introduced more precisely.

[bucket]

All records are stored in a set of buckets. Therefore, a bucket is a unit of logical space. This term is used to avoid the confusion with a usual term "page". A "bucket" does not refer to a physical area but to a logical area. However, the entire the logical space is not used, but only parts of it are effectively used. The effective logical space changes as buckets are split, and its range is denoted by $[L_{min}, L_{max}]$. Each "bucket" has also a unique physical address corresponding to its logical address.

[bucket size]

The bucket size is a number of records which can be stored in a single bucket. While we assume that records are fixed length in the diagrams and examples for simplicity, this assumption is not necessary and not made in the actual procedures described later. We require that the hashing function is chosen so that the number of records which have the same hash value is always less than the bucket size. Thus, the hashing function should be chosen carefully by the file designer.

[split sequence]

The logical addresses of the buckets which have been split are stored in a sequence which is called the split sequence. A split sequence L is represented by the ordered set of $\{L_i\}$, where L_i i=1, 2,..n are the logical split bucket addresses.

[bucket number]

The bucket number, which is initially N_o, represents the total number of buckets used at any points in time. This value is increased one at a time as splitting occurs.

Next, we describe the procedures for record insertion, logical address calculation, physical address calculation and splitting buckets. In the remarks section (3.5) we give an explanation for these procedures, as well as an example that shows how they are used. It is suggested that the reader refers to the example when reading the procedures below.

3.1 Record Insertion Procedure

step1: Determine the physical address from the key value of a record which needs to be inserted as follows:

1) key --> hash (key) (with user selected hash function)
2) hash(key) --> X (the saw tooth function $X=\lceil c\text{-hash(key)}\rceil+\text{hash(key)}$ is used)

3) X -> logical address (shown in section 3.2 below)

4) logical address -> physical address (shown in section 3.3 below)

step2: If the bucket with the calculated physical address is full then perform the split procedure described later.

step3: Insert the record at the appropriate location in the bucket.

3.2 Logical address calculation from X value

Proc: Logical(X)

X: Given X value

step1: Calculation of the base X value (X_2) and the base logical address (L_b)

Decompose X into X_1 and X_2 such that

$X=X_1+X_2$, X_1:Integer, $0 \le X_2 \le 1$

then calculate L_b and X_b by the following:

$L_b=[X_2/a]$, $X_b=L_b*a$

where $a = 1 / N_o$

let $l=0$, $i=0$, $L_{max}^0=N_o$, $L_{min}^0=1$

(see definition below)

step2: execute $L(L_{max}^0, L_{min}^0, L_b, X_b, X_2, l, i)$

step3: Logical=L, return

Proc: $L(L_{max}^i, L_{min}^i, L_t, X_t, X_{2t}, l, i)$

L_{max}^i and L_{min}^i: The max. and min. logical addresses after i-th splitting

L_t: Temporary logical address used for calculation

X_t: The maximum X-value of the bucket with logical address L_t

X_{2t}: Temporary X-value with the same decimal part as that of the given X-value

l: The number of splittings

i: Indicates the i-th splitting

step1: Return condition

if $L_{min} \le L_t \le L_{max}$

then, $L=L_t$ return

step2: case1: $L_t = L_{i+1}$ (case that the bucket L_t is split)

if $X_t-(1/2)^{l+1}*a \le X_{2t} \le X_t$

then

$L_t=L_{max}^i-L_i+L_{i+1}+1$

$X_t=X_t + 1.0$

else

$L_t=L_{max}^i-L_i+L_{i+1}$

$X_t=X_t-(1/2)^{l+1}*a + 1.0$

endif

$X_{2t}=X_{2t}+1.0$

execute $L(L_{max}^i-L_i+L_{i+1}+1,\ L_{i+1}+1,\ L_t,\ X_t,\ X_{2t},\ l+1,\ i+1)$

(See Remarks 1-(c) and 1-(d) in section 3.5)

case2: $L_t < L_{i+1}$ (case that the bucket L_t is in the left of the split bucket L_{i+1})

$L_t=L_t+L_{max}^i-L_{min}^i+1$

$X_t=X_t+1.0$

$X_{2t}=X_{2t}+1.0$

execute $L(L_{max}^i+L_{i+1}-L_i+1,\ L_{i+1}+1,\ L_t,\ X_t,\ X_{2t},\ l,\ i+1)$

(See Remarks 1-(a) 1-(b) and 1-(c))

others:

execute $L(L_{max}^i+L_{i+1}-L_i+1,\ L_{i+1}+1,\ L_t,\ X_t,\ X_{2t},\ l,\ i+1)$

(See Remarks 1-(a) 1-(b) and 1-(c))

3.3 Physical address calculation from Logical address

Proc: Physical(L)

L: Given logical address

step1: Initialization

let k=n (n is the split sequence size)

$L_{max}^k = L_{max}$, $L_{min}^k = L_{min}$

where L_{max} and L_{min} mean the current maximum and minimum logical addresses.

step2: calculate $P(L, L_{max}^k, L_{min}^k, k)$

step3: Physical = P, return

Proc: $P(L, L_{max}^i, L_{min}^i, i)$

L: Given logical address

L_{max}^i: The maximum logical address after i-th splitting

L_{min}^i: The minimum logical address after i-th splitting

i: Indicates the i-th splitting

L_i: Logical address of the bucket of the i-th splitting

step1: Case that the bucket has been generated due to splitting

if $L = L_{max}^i$ then $P = L_{max}^i - L_{min}^i + 1$, return

step2: Case that the bucket has existed from the beginning

if $L < N_o$ then P=L, return

step3: Case that the bucket is in the left of the split bucket

if $L \geq L_{max}^i - (L_i - L_{i-1})$

then $P(L + L_i - L_{max}^i + 1, L_{max}^i - (L_i - L_{i-1} + 1), L_{i-1} + 1, i-1)$

< See Remarks 1-(b) >

else $P(L, L_{max}^i - (L_i - L_{i-1} + 1), L_{i-1} + 1, i-1)$

< See Remarks 1-(c) >

endif

3.4 Split procedure

step1: Create a new bucket whose physical address equals to the current maximum physical value plus one.

step2: Calculate min/max values(X_{min} and X_{max}) of X which map the logical address of the bucket which overflows to the X value space.

These values can easily be obtained from the procedure of calculating logical address described before.

step3: For each record in the bucket, do the following:

if an X value of hash(key of record) is between $[X_{min}, (X_{min} + X_{max})/2]$

then leave the record in the bucket

else insert the record in the new bucket

step4: Update the following values:

The constant c of the saw-tooth function

Split sequence L

Current logical space $[L_{min}\ L_{max}]$

as $c = X_{max}$, $L_{min} = L_{n+1} + 1$, $L_{max} = L_{max} + L_{n+1} - L_n + 1$, and $n = n+1$.

3.5 Remarks

1) Explanation

The above procedures can be explained using the following equations.

Let a split sequence be $\{L_1, \cdots, L_k\}$ after k splittings, the effective logical space at the time $[L_{min}^k, L_{max}^k]$, and the initial bucket size N_o. Then the following equations hold. Given a logical address L,

a) Physical(Logical(X-1))=Physical(Logical(X))

for X such that $X_{min}(L) \leq X \leq X_{max}(L)$ and L is not a split bucket

b) $\text{Physical}(L + L_k - L_{max}^k + 1) = \text{Physical}(L)$

for L such that $L \geq L_{k-1}$

c) $L_{max}^{k-1} = L_{max}^k - L_k + L_{k-1} - 1$, $L_{min}^{k-1} = L_k - 1$

d) $X_{max}(L) - X_{min}(L) = (1/2)^{l(L)} * a$

where

-Physical(L) means Physical address corresponding with Logical address L,

-Logical(X) means Logical address mapped from a value X,

-$X_{min}(L)$ and $X_{max}(L)$ mean minimum and maximum

-X values with logical address L,

-$a=1.0/N_o$,

-l(L) means the split number of the bucket with logical address L.

2) Integration of the above procedures

It is easy to combine the logical address calculation and the physical address calculation shown above, into a single procedure, because both procedures use the history of splitting. The combined procedure makes it possible to calculate the physical address from the X-value directly [Kawa84].

3) Example

The procedures stated before are easy to understand with the example shown in Fig.4.

First, a file is assumed to contain five buckets as shown in Fig.4(a). Each bucket has a logical address which is the same value as its physical address in the range of [1, 5]. After insertion of some records, bucket 3 is assumed to overflow. Therefore, buckets 1 and 2 which have lower logical addresses than bucket 3, are logically moved toward higher addresses. Then, bucket 3 is split into two buckets. The result is shown in Fig.4(b) (disregard the shading of bucket 4, which will overflow in the next step). Buckets 6 and 7 have the same physical address as the previous buckets 1 and 2, respectively, and buckets 8 and 9 are generated by splitting bucket 3. Physically, only one bucket is created in this process and some records in the buckets are moved to the new bucket 9, whose physical address is 6. When bucket 4 overflows, it is split as shown in Fig.4(c). Then Fig.4(d) shows the result of overflowing bucket 6 and Fig.4(e) shows the result of overflowing bucket 9. The resulting split sequence is , therefore, { 3, 4, 6, 9 }.

Instead of the split sequence, the original growth function can also be used for address calculation, which is shown in Fig.4(f). However, it is obvious from Fig.4(f) that seven pairs of (X-value, logical address) are necessary to store this function, which requires more space than the split sequence. Therefore, we use only the split sequence in the procedures shown in the previous sections.

Next, we show an example of computing the physical bucket address from a given key. From the key value, the X-value is obtained by simple calculation. Assume that this value from the given value is 2.54. The value c after the first step is .6 because c is changed to X_{max} as shown in the split procedure in section 3.4. Similarly, c changes in the next steps and the current c value after 5 record insertion is 1.6. According to the above procedure, the result of step 1 is $X_2 = 0.54$, $a = 0.2$, $L_b = 3$, $X_b = 0.6$ $L_{max}^0 = 5$, $L_{min}^0 = 1$, $L_{max} = 18$ and $L_{min} = 10$. Then at step 2, the Proc L (5, 1, 3, 0.6, 0.54, 0, 0) is used. First, because L_t is not between L_{max} and L_{min} and also because L_t equals to L_1 and new X_t, and L_t are calculated. The result is that $L_t = 9$ and $X_t = 1.6$. Next, the procedure L (9, 4, 9, 1.6, 1.54, 1, 1) is executed. Because both L_t is not between L_{max} and L_{min} and L_t not $\leq L_1$, L is executed again with the change of L_{min}^0 and L_{max}^0. At the second and third iterations, the results are L (11, 5, 9, 1.6, 1.54, 1, 2) and L (14, 7, 9, 1.6, 1.54, 1, 3), respectively. At the forth iteration, $L_t = L_4$. Then, it is obtained for $X_t = 2.55$ $L_t = 17$. Finally, when L (18, 10, 17, 2.55, 2.54, 2, 4) is executed, L_t is in $[L_{min}, L_{max}]$ and the procedure terminates. Therefore, the logical address $L_t = 17$ is the result.

After obtaining the logical address, the physical address can be calculated as follows. From the logical address = 17, the current $L_{max} = 18$ and the current $L_{min} = 10$, the procedure P (17, 18, 10, 4) is performed. The condition of the step 3 holds because L (=17) does not equal to L_{max} (=18) and L is not less than N_o (=5) but $L \geq 15$ which is $L_{max}-(L_4-L_3)$. Therefore, the procedure P (9, 14, 7, 3) is executed. At the second iteration, the result is to execute P (9, 11, 5,

2) because no if-conditions in the procedure hold. At the third iteration, the result is to execute P (9, 9, 4, 1) because of the same reason. Finally, the value L (=9) becomes the same as L_{max}^{1}. The physical address P = L_{max}^{1}-L_{min}^{1}+1 = 6. Therefore, a record with X-value 2.54 is in the bucket with physical address 6. This access is, thus, guaranteed to succeed only a single access is necessary.

4. Discussion

The performance of a file organization and its access method can be evaluated with the following measures.

- The load factor
- The number of storage accesses for searching
- The number of storage accesses for insertion

The last two values equal exactly to one in the case of the modified dynamic hashing, because the file organization with the modified dynamic hashing does not use overflow area, since a bucket which overflows is split immediately. Therefore, the modified dynamic hashing only requires one logical access. However, the logical access may not require a physical I/O in case that the bucket is already in memory.

The fact that the number of logical accesses always equals to one is an important advantage. It is achieved by using more CPU time in calculating the physical address than the conventional dynamic hashing methods. However, this is a worthwhile trade-off because CPU costs are continuously decreasing and unnecessary physical I/O should be avoided.

To estimate the load factor, we can use analysis methods similar to that of B-tree. The splitting causes the decomposition of the records in the overflow bucket into the two buckets. The result of B-tree analysis, which has been done by Yao [Yao78], has showed that the average load factor is as follows in the case that records in buckets are sufficiently large,

LOAD = ln 2 * (m+1/m), where m is the bucket size.

When the value of m is also sufficiently large, LOAD is approximately ln 2. Therefore, in the modified dynamic hashing, the load factor is expected ln 2.

To investigate the above result, a simulation was performed. Fig.5 shows the load factor for several bucket sizes and bucket numbers. The bucket sizes chosen are 20, 50, 100 and 150. For each bucket size, the load factors are calculated after n record insertions, where n is set to 500 in the simulation. Records to be inserted are assumed to have random keys. From the simulation results in Fig.5, it can be observed that the load factor is close to ln 2.

We also measured the number of bucket splits. For the case that ten thousand records are inserted, the number of splittings is approximately between 100 and 650 depending on the size of a bucket. This means that in practice the split sequence can be stored in main memory.

In summary, the advantages of the modified dynamic hashing are:

1) It provides direct access to the desired bucket without additional bucket or index
2) It provides ln 2 load factor with less variation, similar to B-tree
3) It is easy to implement due to the simple access method.
4) No index tree is required. The necessary space size is proportional to the number of splits.

In order to make this hashing technique more practical, there are a couple of problems related with the modified dynamic hashing that have to be worked out in detail. We indicate below possible techniques for their solutions.

The three problems are: 1) record deletion and bucket concatenation, and 2) the reorganization of logical space, and 3) address calculation cost reduction.

The deletion problem can be supported as follows. One possible method is to delete records logically, not physically. However, this method causes the load factors to degenerate. As in the case of a B-tree, bucket

concatenation should be used. By doing so, it is possible to store records more efficiently. In our method, it is possible to use the split sequence including concatenated bucket addresses in order to achieve the effect of concatenation.

The problem of organizing the logical space arises as more and more splitting cause the logical space to assume larger and larger values, and the split sequence increases. The computation time therefore increases as well. The reorganization of the logical space can be achieved by exchanging the physical addresses of buckets. It amounts to finding the simplest method for the bucket context permutation, which is similar to matrix permutation.

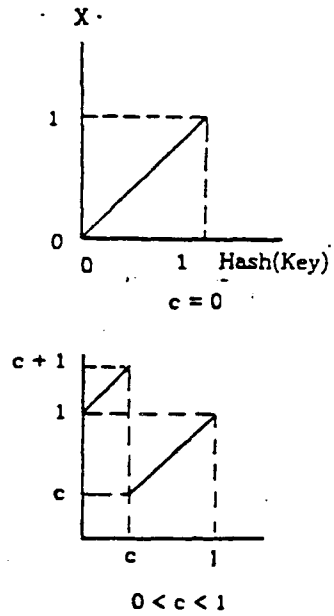
The cost problem of address calculation comes from the split sequence for address mapping which needs a linear order access O(n), where n means the number of buckets. If a tree structure can be constructed, the access cost could be reduced to O(log n). In order to change a split sequence into a form of tree structure, it is required to include the set of X-values related with the split buckets as well as pointers for tree representation. Therefore, the necessary memory size for this split tree structure increases. This means that designers should select an appropriate structure, considering trade-off between the memory cost and the access cost.

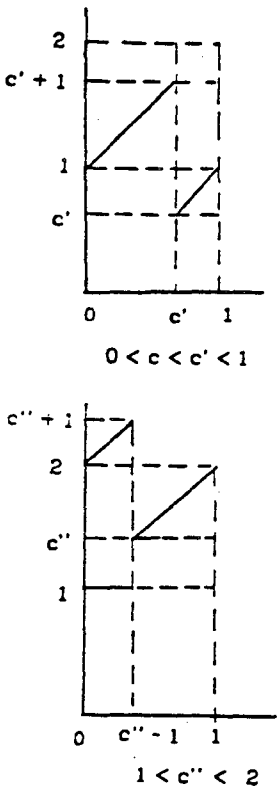
5. Conclusion

A new dynamic hashing method called Modified Dynamic Hashing has been described in this paper. This method is based on the unified hashing method presented by Mullin. As a growth function, the modified dynamic hashing includes piecewise linear functions, which the same as suggested by Mullin. However, his method requires a table of pairs for the representation of the function. Instead, the method presented in this paper has split sequence which is an ordered set of split bucket logical addresses. The number of integers in split sequence is the order of the number of buckets, and therefore can usually be stored in main memory. Moreover, only one access is required using this method, while other methods may require additional bucket accesses for overflowed bucket. Therefore, this method has better performance as a secondary index file access method.

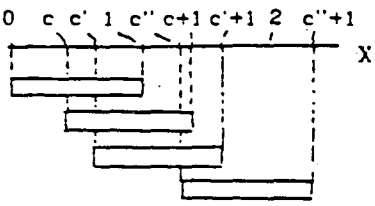
Acknowledgement

I would like to thank Dr. Kato, Dr. Mikami, Dr. Naniwada and Mr. Managaki for giving me a chance to study in the US and also to thank Dr. Shoshani for his helpful and constructive comments on a draft of my paper and his helping me with my English.

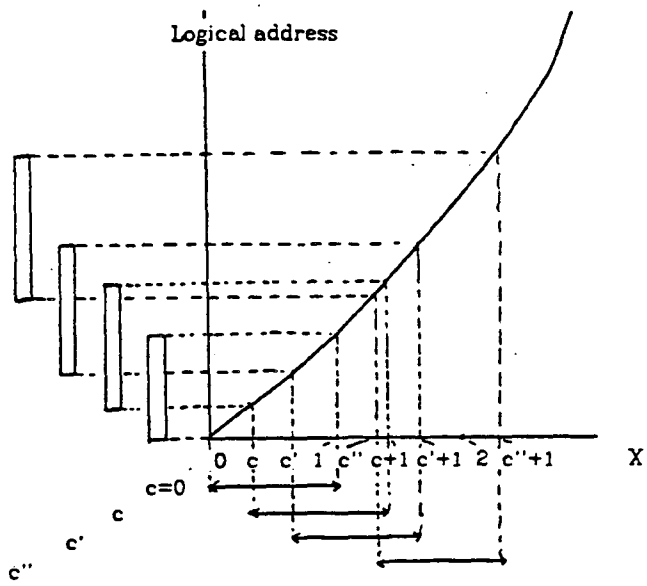




(a) X function

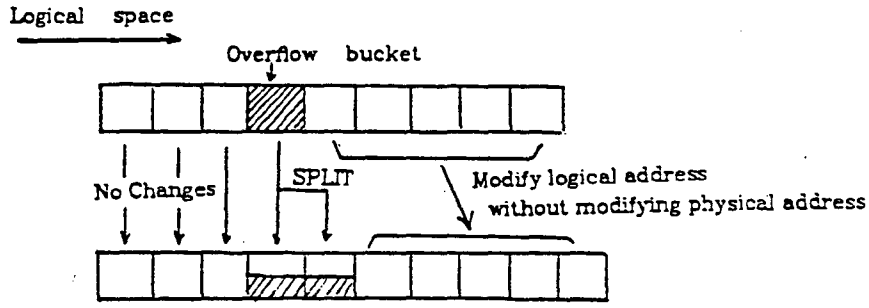


(b) X function region

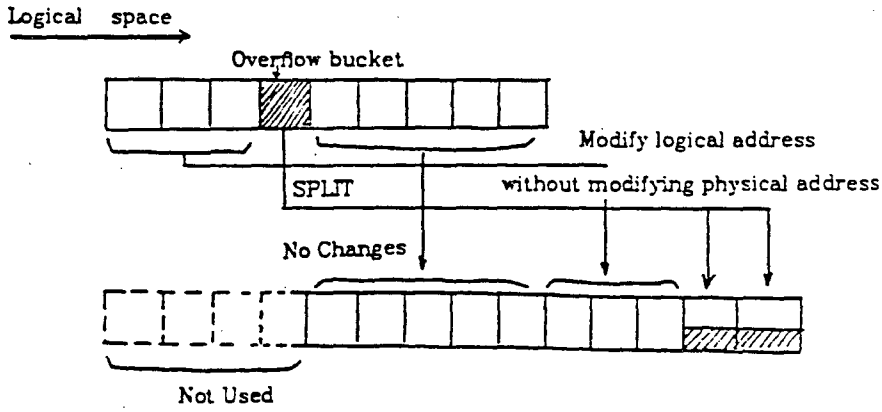


(c) Growth function

Fig.1 X function and Growth function



(a) Ideal splitting



(b) Modified dynamic hashing splitting

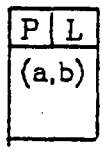
Fig.2 Bucket Splitting

salary	hash value	record-id
10	0.1	1
100	0.9	2
20	0.4	3
120	0.6	4
130	0.7	5

(a) Example records

L[x,y] : Logical space domain

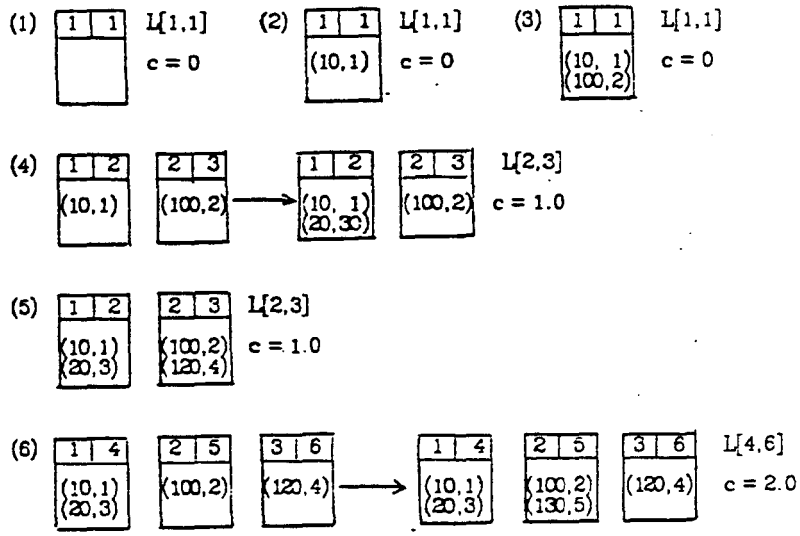
c : a parameter used for X-value calculation



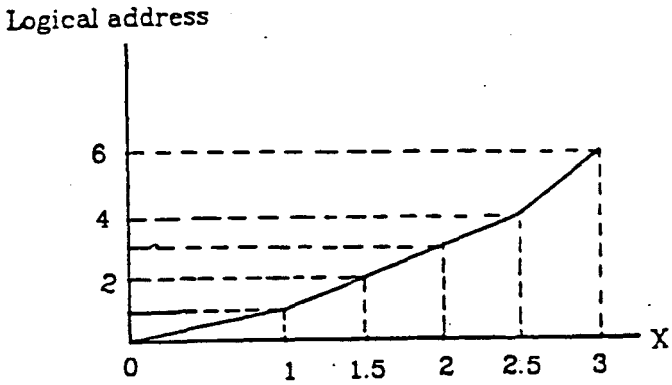
P Physical address
L Logical address
a salary (key value)
b record-id

Bucket size = 2

(b) Notation and assumption



(c) Buckets history as records insert

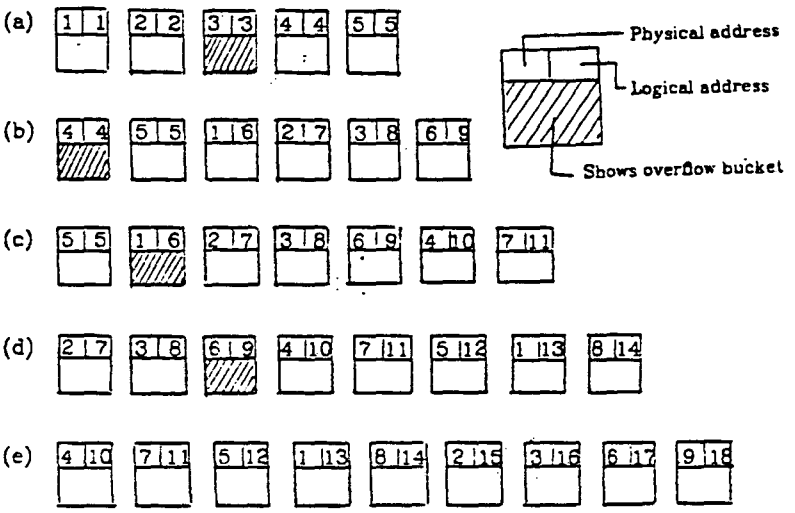


(d) Growth function

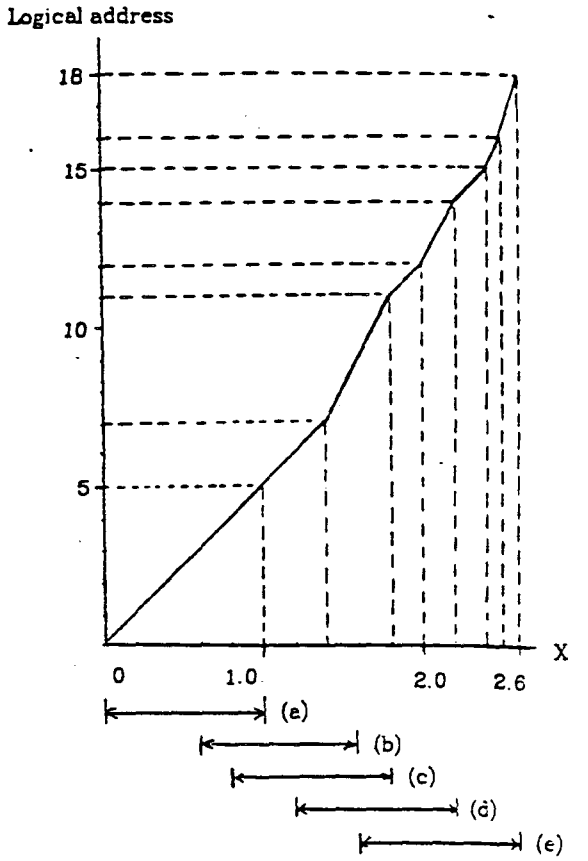
X	L
1	1
2.5	4
3	6

(e) Table of growth function

Fig.3 Example



(a)-(e) Bucket history



(f) Growth function

Fig.4 Example of modified dynamic hashing

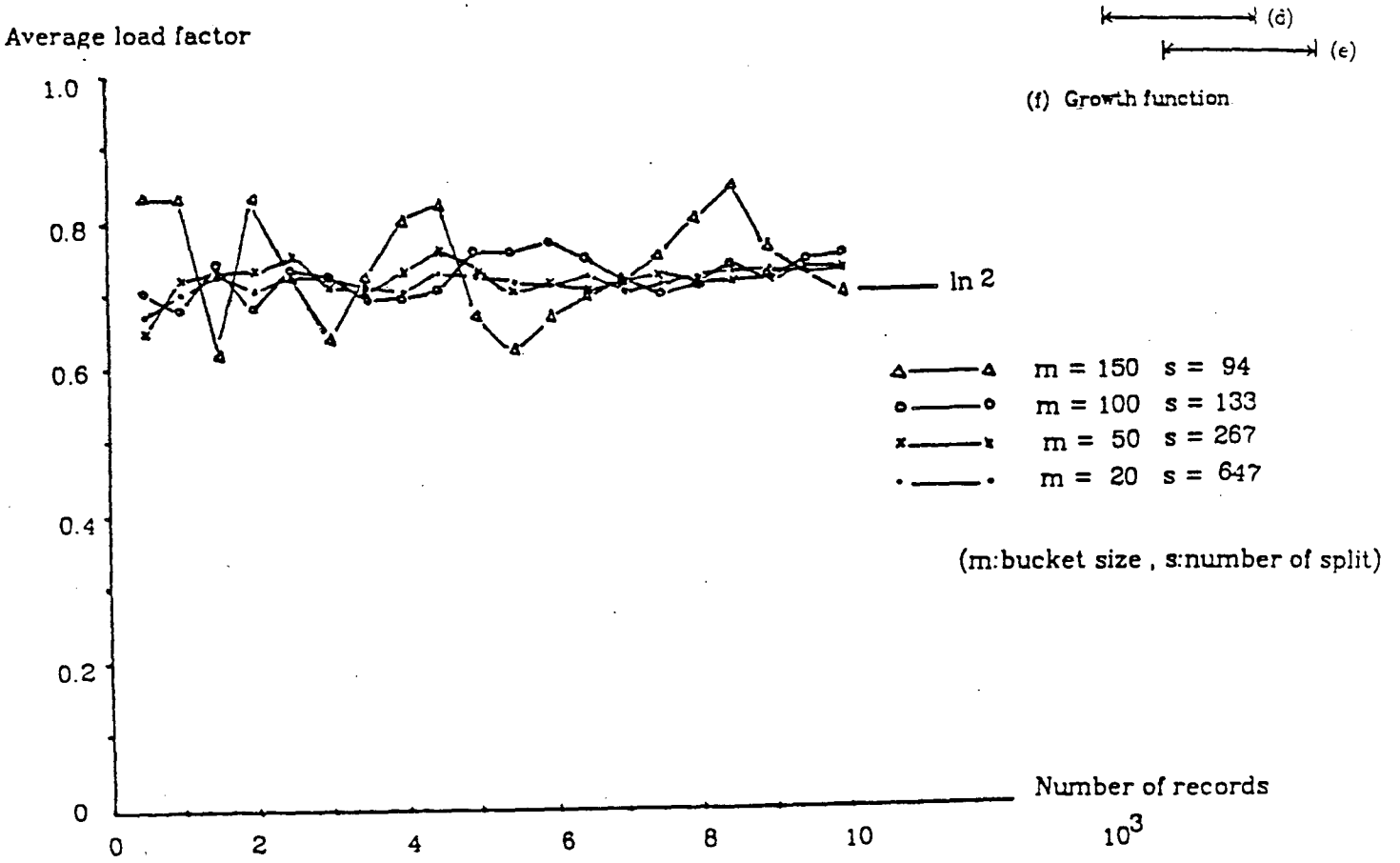


Fig.5 Average load factor vs. Number of records with Modified dynamic hashing

References

[FaNP79] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R., Extendible Hashing: A Fast Access Method for Dynamic Files. ACM. TODS., 4, 3, pp315-344, Sep., 1979

[Kawa84] Kawagoe, Kyoji, Modified Dynamic Hashing, Internal Memo., Oct, 1984

[KjeZ84] Kjellberg, P. and Zahle, T. U., Cascade Hashing, Proc of 10th VLDB, Singapole , pp481-492, Aug., 1984

[Knot71] Knott, G. D., Expandable Open Addressing Hash Storages and Retrieval, Proc. 1971 ACM-SIGFIDET W. S. on Data Description Access and Control, pp187-206, 1971

[Knut73] Knuth, D. E., The Art of Computer Programming, Vol 3/Sorting and Searching, Addison-Wesley, 1973

[Lars78] Larson, P., Dynamic Hashing, BIT 18, 2, pp184-201, 1978

[Lars80] Larson, P., Linear Hashing with Partial Expansions, Proc.of 6th VLDB , pp224-232, 1980

[Litw78] Litwin, W., Virtual Hashing:Dynamically Changing Hashing, Proc. of 4th VLDB, West-Berlin, Sep, pp517-523, 1978

[Litw80] Litwin, W., Linear Hashing:A New Tool for File and Table Addressing, Proc of 6th VLDB, pp212-223, 1980

[Lome83] Lomet, D. B., Bounded Index Exponential Hashing, ACM TODS, 8, 1, pp136-165, 1983

[Mull84] Mullen, J. K., Unified Dynamic Hashing, Proc.of 10th VLDB, pp.473-480, AUg., 1984

[RamL82] Ramamohanaran, K. and LlOyd, J. W., Dynamic Hashing Schemes, The Computer Journal, 25, 4, pp478-485, Nov., 1982

[Tamm82] Tamminan, M., Extendible Hashing with Overflow, Inf. Proc. Letters, 15, 5, pp 227-232, Dec., 1982

[Yao78] Yao, A. C. C., On Random 2-3 Trees, Acta Inf., 9, pp159-180, 1978

Appendix

Proof of the equations in 3.5-1)

a) It is obvious from file expansion way to hold that both the logical address of X-1 and the logical address of X have the same hashing value, when a bucket with logical address of X-1 is not a split bucket. Then, the bucket with logical address of X-1 and the bucket with logical address L are the same bucket, physically. Therefore, the physical address of the bucket with the logical address L equals to the physical address of the bucket with the logical address whose X value is X-1 , where $X_{min}(L) \leq X \leq X_{max}(L)$.

b) Let L' be the logical address of L before splitting. Then, by splitting of L_k, L' bucket on the $(L_k-L'+1)$ lower of the L_k bucket is logically moved to the L bucket on the $(L_{max}{}^k- L)$ lower of the $L_{max}{}^k$ bucket. Then, $L'=L+L_k-L_{max}{}^k+1$ holds.

c) The length of logical space after k-1 splitting is $L_{max}{}^{k-1}-L_{k-1}$. That of logical space after k splitting is also $L_{max}{}^k-L_k$. The difference of them equals to one because only one bucket is generated by splitting. Then, $L_{max}{}^{k-1}=L_{max}{}^k-L_k+L_{k-1}-1$ holds. It is easy to show $L_{min}{}^{k-1} = L_k -1$.

d) By splitting, the difference between $X_{max}{}^k$ and $X_{min}{}^k$ is half of that between $X_{max}{}^{k-1}$ and $X_{min}{}^{k-1}$. Then, $X_{max}-X_{min}=(1/2)^l*a$ is easily obtained.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720