Algorithmica (1996) 16: 215–232

Algorithmica

Efficiently Extendible Mappings for Balanced Data Distribution

D. M. Choy,[1] R. Fagin,[1] and L. Stockmeyer[1]

Abstract. In data storage applications, a large collection of consecutively numbered data "buckets" are often mapped to a relatively small collection of consecutively numbered storage "bins." For example, in parallel database applications, buckets correspond to hash buckets of data and bins correspond to database nodes. In disk array applications, buckets correspond to logical tracks and bins correspond to physical disks in an array. Measures of the "goodness" of a mapping method include:

(1) The *time* (number of operations) needed to compute the mapping.
(2) The *storage* needed to store a representation of the mapping.
(3) The *balance* of the mapping, i.e., the extent to which all bins receive the same number of buckets.
(4) The cost of *relocation*, that is, the number of buckets that must be relocated to a new bin if a new mapping is needed due to an expansion of the number of bins or the number of buckets.

One contribution of this paper is to give a new mapping method, the *Interval-Round-Robin* (*IRR*) method. The IRR method has optimal balance and relocation cost, and its time complexity and storage requirements compare favorably with known methods. Specifically, if m is the number of times that the number of bins and/or buckets has increased, then the time complexity is $O(\log m)$ and the storage is $O(m^2)$. Another contribution of the paper is to identify the concept of a *history-independent* mapping, meaning informally that the mapping does not "remember" the past history of expansions to the number of buckets and bins, but only the current number of buckets and bins. Thus, such mappings require very little information to be stored. Assuming that balance and relocation are optimal, we prove that history-independent mappings are possible if the number of buckets is fixed (so only the number of bins can increase), but not possible if the number of bins and buckets can both increase.

Key Words. Database management, Parallel database, Disk array, Extendible storage system, Data striping, History independence, Data mapping, Balanced distribution.

1. Introduction.

A common way to store a large number of data objects is to distribute the objects across multiple storage locations. This approach:

- Enables parallel processing when each storage location is supported by a separate piece of hardware (e.g., storage device, CPU).
- Reduces the search scope when the storage location(s) of the target object(s) of a selective retrieval can be deduced from the query.
- Allows horizontal expansion to accommodate capacity or performance growth by adding storage locations incrementally.
- Can be easily scaled to handle a very large volume of data.

As an example, in a database management system (DBMS), a relation can be "horizon-

[1] IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, USA. {dmc, fagin, stock}@almaden.ibm.com.

Received March 22, 1994; revised October 16, 1994. Communicated by C. K. Wang.

tally partitioned" into multiple subsets of tuples, with each subset stored at a separate DBMS node or on a separate storage device [3], [7]. The partitioning can be done, for instance, by hashing on a designated "partition key" of the relation to produce a node number. To do so, a two-step mapping is often used: the tuples are hashed to a relatively large, but fixed, range of hash bucket id's, which in turn are mapped to storage "bins" (i.e., DBMS nodes or storage devices). This indirect mapping simplifies the remapping of bins (e.g., to switch data accesses to a redundant storage device when a device fails, or to support a bulk migration of data from one node/device to another). It also facilitates the addition of (one or more) bins to accommodate database growth or to increase parallelism. Since hashing and other data partitioning methods have been well investigated, we hereby focus on the bucket-to-bin mapping and assume that the tuples are distributed fairly evenly into the buckets using a suitable partitioning method. We are particularly interested in how well a bucket-to-bin mapping is able to support database growth, i.e., to accommodate the addition of new bins.

Another example of object distribution across multiple storage locations is the case of data striping for a disk array [6]. An attractive way to use a disk array is to treat the entire array as a single, logical disk so that existing system and/or application software does not have to be changed to understand the physical configuration and operation of an array. With this approach, we take the "buckets" to be logical tracks in the logical disk, with as many logical tracks as there are physical tracks on the disks in the array. It is then the responsibility of the array controller to map the logical tracks one-to-one onto the physical tracks. This approach is followed, for example, in the IBM 3514 High Availability Disk Array [4]. We think of each disk in the array as a "bin," so once again we are assigning buckets to bins. When (one or more) new disks are added, the number of buckets and bins both increase. This is because when new disks are added, there are more physical tracks, and hence more logical tracks, that is, buckets. This is in contrast to our previous example of a partitioned database, where the number of bins, but not buckets, may increase. A simple way to handle the addition of new disks is to map the new logical tracks onto the new disks, while keeping the mapping from the old logical tracks to the old disks unchanged. A disadvantage of this approach is that the new logical tracks are initially unused, so the new disks will be underutilized until the new logical tracks are written with data. As an alternative to alleviate this disadvantage, a two-step expansion procedure could be used: first the number of disks (bins) is increased, and the old logical tracks are remapped evenly to all the disks; then the number of logical tracks (buckets) is increased, and the new logical tracks are mapped evenly to all the disks.

Note that in both applications of bucket-to-bin mappings described above, an important measure of the goodness of a mapping is its *balance*, that is, the extent to which all bins receive the same number of buckets. Ideally, the mapping is *optimally balanced*, that is, if there are B buckets and n bins, then each bin receives either $\lfloor B/n \rfloor$ or $\lceil B/n \rceil$ buckets.

A typical method to map buckets to bins in an optimally balanced way is a simple Round-Robin (RR) assignment. The bin number assigned to a given bucket can be easily calculated using modular arithmetic:

$$\text{bin-number} = \text{bucket-number} \mod n,$$

where n is the number of bins used to store the set of objects. (We assume here and subsequently that buckets are numbered $0, 1, 2, \ldots, B-1$ and that bins are numbered $0, 1, 2, \ldots, n-1$.) While this method is simple and efficient, there is unfortunately a large relocation cost when new bins are added. For example, if a new bin is added to n existing bins, approximately the fraction $n/(n+1)$ of the data must be moved from one bin to another before the data can be accessed using the new mapping. For a large database, this leads to a long period of unavailability of data, which is not acceptable to many applications. In contrast, an optimal relocation would move only $1/(n+1)$ of the data, which is to populate the new bin so that the expected workload is balanced over all $n+1$ bins, without bucket redistribution among the old bins.

A different approach is to maintain a stored bucket-to-bin mapping. In this case, a directory of B entries is maintained in which the ith entry contains the bin number assigned to bucket i, where B is the total number of buckets and is usually a fairly large integer. Thus, each bucket can be individually assigned or reassigned to any bin. When new bins are added, individual buckets are selected for relocation to the new bins so that only the minimum amount of data is moved. However, with this approach, a directory has to be maintained for each relation (or other partitioned set of objects), or for each group of relations that share the same mapping. Furthermore, these directories must be resident in the main memory to assure a good performance for data access. For a large database, they consume a significant amount of memory.

An alternative solution, called Cascaded-Round-Robin (CRR) mapping [2], has recently been proposed. Similar to the simple RR scheme, a CRR mapping can easily be computed in $O(m)$ operations, using a stored sequence of about m integers, where m is the number of times bins are added and is usually a small integer. In particular, m will be smaller than the number of added bins if several bins are added at each expansion. (In contrast, RR needs only a single stored integer, namely, n.) While a CRR mapping can be readily defined that is optimal with respect to relocation, it might not be optimally balanced.

In this paper we propose an Interval-Round-Robin (IRR) mapping to map buckets to bins. It is also easily computable and it guarantees optimality in both relocation (minimum data movement) and balanced bucket distribution. While IRR needs to maintain more data than CRR ($O(m^2)$ integers rather than m where, as before, m is the number of times that bins have been added), it is still far less than the full directory mapping (B integers). Moreover, the IRR mapping can be computed in $O(\log m)$ operations.

For the various mapping methods mentioned above, there is a wide variance in the amount of storage needed to hold a representation of a particular mapping, where the "representation" is the information used to distinguish one mapping from another in the same class of mappings. Call this storage the "mapping storage." In the case of the simple RR method, for example, the number n of buckets is sufficient to represent a particular RR mapping, so the mapping storage is that needed to hold a single number. For the Stored-Directory (SD) method, on the other hand, the representation is a list of B integers, so the mapping storage is large if B is large. The mapping storage of the CRR and IRR methods lies between these two extremes if the number of expansions is not too large. How small can mapping storage be for mappings that are optimal in both balance and relocation? (Recall that the RR method is not optimal in relocation.) The representation must keep track at least of the number of bins, for the following simple

reason. Suppose that the same mapping f were used for two different numbers, n and n', of bins. Say that $n < n'$. Then either f maps a bucket to a nonexistent bin when used for n bins, or f leaves some bin empty when used for n' bins and therefore cannot have optimal balance. So an optimal mapping (optimal in both balance and relocation) must "know" the number of bins. Is just the number of bins sufficient? This is less clear: on the face of it, this seems possible. However, we prove (Theorem 4.2) that an optimal mapping cannot depend just on the number of bins; some additional information is necessary. How much additional information is needed? In particular, is the number of buckets enough? We say that a mapping is *history-independent* if it depends *only* on the number of buckets and bins; so, in particular, it does not depend on the history of expansions which led to the current number of buckets and bins. We show (Theorem 4.1) that if the number of buckets is fixed (so only the number of bins can increase), then there is a history-independent mapping that is optimal. Therefore, in the case where the number of buckets is fixed, mapping storage need be no more than the storage to record the number of buckets and bins. What about the case where the number of buckets is not fixed? We show (Theorem 4.3) that in this case, where the number of bins and buckets can both increase, there is no history-independent mapping that is optimal. It is an open question in this case what the minimal mapping storage is.

The rest of the paper is organized as follows. Section 2 contains definitions, including descriptions of the measures of "goodness" of a mapping method that are of interest to us. In Section 3 we describe our new IRR method. Section 4 contains our results on the existence and impossibility of history-independent mappings. Section 5 contains some concluding remarks and open questions.

2. Definitions. Given a positive integer B, the number of buckets, and a positive integer n, the number of bins, the problem is to construct a mapping f from the set of bucket id's $\{0, 1, 2, \ldots, B-1\}$ to the set of bin id's $\{0, 1, 2, \ldots, n-1\}$. Typically, B is much larger than n. The term "bucket x" is sometimes used as shorthand for "the bucket with id x," and similarly for "bin y." Another part of the problem is to handle an *expansion* when the number of bins increases from n to some $n' > n$, and possibly also the number of buckets increases from B to some $B' > B$. Thus, we also have to explain how to construct a new mapping f' from B buckets to n' bins, or, more generally, from B' buckets to n' bins, when an expansion occurs. In effect, there is not just a single mapping, but rather a family of mappings, i.e., a general "mapping method." We can view a mapping method as a function $M(x, \rho)$ which takes a bucket id x and a *representation* ρ of a particular mapping, and returns a bin id. That is, $f(x) = M(x, \rho)$ where ρ is the representation of f. For example, for the RR method mentioned in the Introduction, the representation ρ is simply n, and $M(x, \rho) = x \bmod n$. For the SD method, ρ is a list $(y_0, y_1, \ldots, y_{B-1})$ of integers, and $M(x, \rho) = y_x$.

Measures of the goodness of a solution include the following:

1. *Balance.* The *size* of bin y under the mapping f is the number of buckets that f maps to bin y. A mapping f from B buckets onto n bins is said to be *balanced* if the size of every bin is either $\lceil B/n \rceil$ or $\lfloor B/n \rfloor$. Equivalently, a mapping is balanced if no two bins differ in size by more than 1. If the mapping is balanced and $r = B \bmod n$, then

r bins have size $\lceil B/n \rceil$ and $n - r$ bins have size $\lfloor B/n \rfloor$. Define a *standard balanced mapping* to be one where the first r bins have size $\lceil B/n \rceil$ and the last $n - r$ bins have size $\lfloor B/n \rfloor$.

2. *Mapping Complexity.* This is the number of operations needed to compute $f(x)$, given a bucket id x.
3. *Mapping Storage.* This is the amount of storage needed to store a representation of the mapping. In placing upper bounds on the mapping storage of a particular mapping method $M(x, \rho)$, we bound only the storage needed for the representation ρ (which can, in general, depend on n, B, and the number of expansions), and we ignore the (constant) storage needed to hold an algorithm for computing M. The algorithms used in our particular mapping methods are not complicated, and are described in the text.
4. *Bucket Relocation.* When a mapping f is replaced with another mapping f' as the result of an expansion, the *bucket relocation* of the expansion is the number of buckets that are assigned to different bins by f and f', i.e., the number of bucket id's x such that $f(x) \neq f'(x)$ and $0 \leq x < B$.
5. *Expansion Complexity.* This is the amount of computation required to construct the representation of the new mapping when an expansion occurs.

These measures could depend on the entire past history of expansions, in particular, on the number of expansions. The expansion complexity is less of a concern than the other measures, since the time required to compute a representation of the new mapping is typically dominated by the time required actually to move the data residing in buckets that are relocated to a different bin. Since the remaining four measures (balance, complexity, storage, and relocation) are difficult to juggle at one time, we have chosen in this paper to focus on mappings that are optimal in both balance and relocation, and to study the time complexity and storage requirements of such mappings.

To illustrate the definitions and to set the stage for a comparison of our method with known methods, we first review two of the known methods that were mentioned in the Introduction. Both methods produce standard balanced mappings.

The RR mapping is given simply by

$$f(x) = x \bmod n,$$

where n is the number of bins. Mapping complexity is small (one modular operation), as is mapping storage (it is sufficient to store n). The problem with this scheme is that bucket relocation is very large. For example, if the number of bins increases from n to $n + 1$, then the data in approximately the fraction $n/(n + 1)$ of the buckets must be relocated. In contrast, the minimum fraction that must be relocated to obtain a balanced mapping is approximately $1/(n + 1)$.

It is not difficult conceptually to minimize bucket relocation, given that mappings should always be balanced. Initially (before any expansion has occurred) the RR mapping is used. Say that the present mapping is a standard balanced mapping f to n bins, and say that an expansion from n bins to n' bins occurs. (To simplify the description, we assume that B does not increase, although the method generalizes easily to the case where B increases.) For each bin y with $0 \leq y < n$, calculate the number of buckets that must be moved from bin y to the new set of bins (n through $n' - 1$) to produce a

new standard balanced mapping f' to n' bins. (As noted above, it is straightforward to compute, for a given bin y and a given B and n, the number of buckets in bin y under any standard balanced mapping from B buckets to n bins.) Then choose a specific set of buckets to be moved from bin y. It is easy to see that this minimizes bucket relocation, given that mappings should always be balanced, since it moves the minimum possible number of buckets required to populate the new set of bins. (To see this, it is enough to recall that the mapping is a standard balanced one, where the higher-numbered bins contain the smaller number $\lfloor B/n \rfloor$ of buckets.) The difficulty with this approach is that the definition of the mapping becomes complicated. One solution, the SD method, is to store the values of the mapping explicitly, i.e., for each bucket id x, the directory contains the value of $f(x)$. Mapping complexity is small (one directory access). The problem with this approach is that mapping storage is the number B of buckets, which could be large.

3. The Interval-Round-Robin Method. Our new mapping method, the IRR method, is related to the SD method in that it always produces a (standard) balanced mapping and minimizes bucket relocation at each expansion. The advantage over the SD method is that the mapping storage is bounded above by cm^2, where m is the number of expansions that have occurred and c is a small constant. This amount of storage could be significantly smaller than B if m is not too large. To achieve this smaller storage, a price is paid in mapping complexity. Two implementations of our method are suggested; these are, in fact, two variations of the basic IRR method, with slightly different representations ρ and mapping functions M. The first, using a table representation of the mapping, has mapping complexity $O(\log m)$ (where, again, m is the number of expansions). The second, using a tree representation, has mapping complexity $O(m)$ (this can be reduced to $O(\log m)$ by rebalancing the tree), and the constant factor c in the upper bound cm^2 on mapping storage is larger for the tree representation than for the table representation; however, the tree representation is more convenient to update when an expansion occurs.

The first four sections, 3.1–3.4, give a basic "implementation-independent" description of the IRR method. Sections 3.5 and 3.6 suggest two possible implementations. To simplify the description, these first six sections concern the case that the number B of buckets is fixed. In Section 3.7 we describe how the method can be modified to handle an increasing B. In Section 3.8 we compare the performance of the IRR method with that of the RR, SD, and CRR [2] methods.

3.1. *Representation of the Mapping.* Suppose that we are in a situation where m expansions have occurred. Part of the representation of the mapping is the sequence $n_0, n_1, n_2, \ldots, n_m$ where $n_0 > 0$ is the number of bins initially, and n_j is the total number of bins after the jth expansion. It is convenient to define $n_{-1} = 0$. Let $d_j = n_j - n_{j-1}$ for $0 \leq j \leq m$. Thus, at the jth expansion, d_j bins are added to the existing n_{j-1} bins to create a new total of n_j bins. Note that $d_j > 0$ for $0 \leq j \leq m$, since $n_{j-1} < n_j$. In what follows, we assume that the numbers d_j are also stored, although an alternative is to recompute a particular d_j whenever it is needed. Define the jth *block* (of bins), for $0 \leq j \leq m$, to be bins with id's in the interval $[n_{j-1}, n_j)$. (For integers z_1 and z_2 with

Algorithm 1: Mapping Computation
Input: A bucket id x
Find i such that x is in $[a_{i-1}, a_i)$
$j = b_i$
Return $n_{j-1} + [(x - c_i) \bmod d_j]$

Fig. 1. Algorithm for mapping computation.

$z_1 < z_2$, the interval $[z_1, z_2)$ contains all integers z with $z_1 \leq z < z_2$.) Thus, d_j is the number of bins in the jth block.

The basic idea of the mapping is to divide the space $[0, B)$ of bucket id's into intervals. All buckets in the same interval are mapped to bins that belong to the same block. A bin can contain buckets from several different intervals. If bucket x is mapped to a bin in block j, then the bin to which bucket x belongs is given by $n_{j-1} + [rank(x) \bmod d_j]$, where $rank(x)$ is the rank of x among the set of bucket id's that are mapped to block j, where the smallest bucket id in this set has rank 0, the second smallest has rank 1, etc. For example, if the jth block consists of bins 7, 8, and 9 (so $n_{j-1} = 7$ and $d_j = 3$) and if buckets 10, 11, 15, 16, 17, 20, 25, 26 are mapped to this block, then $rank(10) = 0$, $rank(11) = 1$, $rank(15) = 2$, etc. So buckets 10, 16, 25 are mapped to bin 7, buckets 11, 17, 26 are mapped to bin 8, and buckets 15, 20 are mapped to bin 9.

In addition to m, the n_j's, and the d_j's, the rest of the representation of the mapping consists of the following:

1. An integer $k \geq 1$, the number of intervals.
2. Integers a_i for $0 \leq i \leq k$ where

$$0 = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = B.$$

   The ith interval is $[a_{i-1}, a_i)$, for $1 \leq i \leq k$. We imagine that the intervals are ordered from left to right, and we say that the ith interval is to the left of the jth interval (and that the jth is to the right of the ith) if $i < j$.
3. Nonnegative integers b_i for $1 \leq i \leq k$. For the ith interval $[a_{i-1}, a_i)$, the number b_i is the block number associated with this interval. Thus, $0 \leq b_i \leq m$. All buckets in $[a_{i-1}, a_i)$ are mapped to bins in block b_i. Define $block(x) = b_i$ for all x in $[a_{i-1}, a_i)$. In general, several intervals can be mapped to the same block; that is, we can have $b_i = b_j$ for different i and j.
4. Nonnegative integers c_i for $1 \leq i \leq k$. For each i, the number c_i is the number of buckets x in intervals to the left of the ith interval (i.e., $x < a_{i-1}$) such that x is mapped to a block other than b_i (i.e., $block(x) \neq b_i$). The c_i's are helpful in computing the mapping. Note that $a_{i-1} - c_i$ is the number of buckets x in intervals to the left of the ith such that x is mapped to block b_i. So $a_{i-1} - c_i = rank(a_{i-1})$ and, in general, $x - c_i = rank(x)$ for all x in $[a_{i-1}, a_i)$. We call c_i the *rank adjustment* of the ith interval.

3.2. *Computation of the Mapping.* The way to compute the mapping given a bucket id x is first to find the interval $[a_{i-1}, a_i)$ to which x belongs, and then to compute the mapping using b_i, c_i, n_{b_i-1}, and d_{b_i}. Pseudocode for the mapping computation is given

by Algorithm 1 in Figure 1. As described below, the Find operation can be implemented either by binary search if the a_i's are stored in a table, or by binary tree search if the a_i's are stored in a binary search tree.

In certain applications of bucket-to-bin mappings, we need to know the "position" of each bucket in its bin. For example, in the disk array example, if logical track (bucket) x is mapped to physical disk (bin) y, then to read or write logical track x we need to know the physical track (position) on disk y which corresponds to logical track x. Formally, a *position function* is any function p defined on the bucket id's such that, for every bin id y, if U_y is the set of bucket id's mapped to bin y and if z_y is the size of bin y, then p maps U_y one-to-one onto $\{0, 1, 2, \ldots, z_y - 1\}$. A natural position function for the IRR mapping is easily computed as

$$p(x) = \left\lfloor \frac{x - c_i}{d_j} \right\rfloor$$

using the i and j computed by Algorithm 1.

3.3. *The Initial Representation.* Initially, when there are n_0 bins and no expansions have occurred, the representation is given by $m = 0$, $k = 1$, $a_0 = 0$, $a_1 = B$, $b_1 = c_1 = 0$, and $d_0 = n_0$. Thus, the mapping is exactly the RR mapping, $y = x \bmod n_0$.

3.4. *Expansion.* This section contains a description of how the representation of the mapping should be modified when the number of bins is increased. Assume that we are in a situation where m expansions have occurred previously (for some $m \geq 0$) and that we have a representation of the mapping, from B buckets to n_m bins, as described above; call this mapping the *old mapping*. Suppose that the number of bins is increased to n_{m+1}. This creates a new block $m + 1$, consisting of bins in $[n_m, n_{m+1})$. The basic idea is, for each block j with $0 \leq j \leq m$, to move the proper number of buckets from block j to block $m + 1$ so as to produce a new standard balanced mapping from B buckets to n_{m+1} bins. Among the buckets in block j, the ones with a higher id number are moved. This has the effect that if a bucket stays in the same block, then it remains mapped to the same bin. So for each block j with $0 \leq j \leq m$, there will be a *splitting point* s_j such that, for each bucket x mapped to block j in the old mapping, if $x < s_j$, then bucket x remains in block j in the new mapping, and if $x \geq s_j$, then bucket x is moved to the new block $m + 1$ in the new mapping. If $a_{i-1} < s_j < a_i$ for some interval $[a_{i-1}, a_i)$ with $b_i = j$ in the representation of the old mapping, then this interval will be split into two intervals, $[a_{i-1}, s_j)$ that remains mapped to block j, and $[s_j, a_i)$ that is mapped to block $m + 1$.

To make the following description of mapping expansion independent of implementation, the result is given as a set of *actions* to be performed. There is an action A_i associated with each interval $[a_{i-1}, a_i)$ in the representation of the old mapping. There are three types of actions:

1. If $A_i = Null$, then buckets in the interval $[a_{i-1}, a_i)$ do not move. The block number and the rank adjustment of the interval do not change.

Algorithm 2: Computation of Expansion Actions
Input: A new number n_{m+1} of bins
$r = B \bmod n_{m+1}$
$ceil = \lceil B/n_{m+1} \rceil$
$floor = \lfloor B/n_{m+1} \rfloor$
$z = 0$
while $r \geq n_z$
  $z = z + 1$
end /* $n_{z-1} \leq r < n_z$ */
do $j = 0$ **to** m
  if $j < z$ **then** $t_j = d_j * ceil$
  else if $j = z$ **then** $t_j = (r - n_{j-1}) * ceil + (n_j - r) * floor$
  else $t_j = d_j * floor$
end
$w = 0$
do $i = 1$ **to** k
  $j = b_i$
  if $t_j \geq a_i - c_i$ **then**
    $A_i = Null$
  else if $t_j \leq a_{i-1} - c_i$ **then**
    $A_i = Move(a_{i-1} - w)$
    $w = w + a_i - a_{i-1}$
  else /* $a_{i-1} - c_i < t_j < a_i - c_i$ */
    $s = t_j + c_i$
    $A_i = Split(s, s - w)$
    $w = w + a_i - s$
end

Fig. 2. Algorithm for computing expansion actions.

2. If $A_i = Move(c)$, then all buckets in the interval $[a_{i-1}, a_i)$ are moved to block $m + 1$. The block number of the interval is changed to $m + 1$, and c becomes the new rank adjustment of the interval.
3. If $A_i = Split(s, c)$, then the interval $[a_{i-1}, a_i)$ is split into two intervals, $[a_{i-1}, s)$ and $[s, a_i)$. Buckets in $[s, a_i)$ are moved to block $m + 1$, and c is the rank adjustment of the interval $[s, a_i)$. Buckets in $[a_{i-1}, s)$ do not move; the block number and rank adjustment of $[a_{i-1}, s)$ are identical to those of $[a_{i-1}, a_i)$ in the old mapping.

Pseudocode for computing the appropriate actions is given by Algorithm 2 in Figure 2. This code has two parts. The first part computes, for each block j with $0 \leq j \leq m$, the quantity

t_j = the total number of buckets that should remain in block j after the expansion.

The second part scans through the intervals and finds the appropriate action for each one. The variable w holds a running total of the number of buckets that the algorithm has so far determined should be moved to block $m + 1$; this total is useful in computing new rank adjustments. The a_i, b_i, and c_i used in Algorithm 2 are those of the old mapping. Some explanation of the second part of the algorithm may be helpful. Suppose we are finding the action for the ith interval, $[a_{i-1}, a_i)$. Let $j = b_i$ be the block to which this interval is mapped in the old mapping. There are three cases.

1. $t_j \geq a_i - c_i$. Recall that $a_{i-1} - c_i$ is the number of buckets that are mapped to block j and that belong to intervals to the left of the ith interval. Therefore, the number of buckets that are mapped to block j and that belong to intervals 1 through i inclusive is $(a_{i-1} - c_i) + (a_i - a_{i-1}) = a_i - c_i$. Since this number $a_i - c_i$ is at most t_j (the number of buckets in block j after the expansion), the entire ith interval should remain mapped to block j. That is, the action A_i should be *Null*.
2. $t_j \leq a_{i-1} - c_i$. In this case the buckets that are mapped to block j and that belong to intervals to the left of the ith interval are sufficient to fill block j with at least t_j buckets. Therefore, the entire ith interval should be moved to block $m + 1$. To compute the new rank adjustment (call it c_i'), note that $a_{i-1} - c_i'$ should be the number of buckets that are mapped to block $m + 1$ and that are in intervals to the left of the ith interval. Since this number is precisely w, we have $c_i' = a_{i-1} - w$. Then w is incremented by $a_i - a_{i-1}$, since this many additional buckets are mapped to block $m + 1$.
3. Otherwise, $a_{i-1} - c_i < t_j < a_i - c_i$. In this case the ith interval must be split. To compute the splitting point s, note that we want the buckets mapped previously to block j, together with the buckets in the interval $[a_{i-1}, s)$, to fill block j with exactly t_j buckets. That is, $(a_{i-1} - c_i) + (s - a_{i-1}) = t_j$, which gives $s = t_j + c_i$. The argument that the new rank adjustment of the interval $[s, a_i)$ should be $s - w$ is identical to the argument for the previous case, except that the left endpoint of the interval is now s instead of a_{i-1}. Similarly, w is incremented by $a_i - s$.

Since both mapping complexity and mapping storage depend on the number k of intervals, it is useful to have an upper bound on k as a function of m. The following gives such a bound.

LEMMA 3.1. *If k intervals are produced as the result of m expansions to the number of bins, then*

$$k \leq \tfrac{1}{2}m(m + 1) + 1.$$

PROOF. The proof is by induction on m. Initially (when $m = 0$) there is one interval. Assuming that the bound holds for m expansions, we prove it for $m + 1$ expansions. Just before the $(m + 1)$st expansion, there are $m + 1$ blocks, 0 through m. For each of these blocks, there will be at most one interval that is mapped to the block and that is split during the $(m + 1)$st expansion. So the $(m + 1)$st expansion causes at most $m + 1$ intervals to be split, thus creating at most $m + 1$ new intervals. Therefore, using the induction hypothesis, the total number of intervals after $m + 1$ expansions is at most

$$\tfrac{1}{2}m(m + 1) + 1 + (m + 1) = \tfrac{1}{2}(m + 1)(m + 2) + 1. \qquad \square$$

3.5. *A Table Implementation.* In the table implementation, the numbers a_i, b_i, c_i, d_j, and n_j are stored in random-access tables (or arrays). The Find operation in Algorithm 1 is done by binary search in the table of the a_i's. Obviously, mapping complexity is $O(\log k)$ and mapping storage is $O(k)$. By Lemma 3.1, mapping complexity is $O(\log m)$ and mapping storage is $O(m^2)$.

When an expansion occurs, Algorithm 2 is first applied to produce a sequence of actions. To reduce the amount of data that is inaccessible at any one time during an expansion, it is advantageous to handle the actions one at a time. Then the maximum unit of data that is inaccessible at any one time is the data belonging to buckets that are in the same interval and that are moved to the new block $m + 1$. A straightforward way to update the tables incrementally is to maintain two sets of tables, Old-Tables and New-Tables, with Old-Tables initialized to the tables that represent the mapping before the expansion. The non-*Null* actions are processed one at a time. To process a particular action, it is applied to Old-Tables to produce New-Tables, and any data that must be moved to the new block $m + 1$ as a result of the action is moved at this time. After each action is processed, Old-Tables is replaced by New-Tables.

3.6. *A Tree Implementation.* In the tree implementation, the numbers a_i, b_i, c_i are stored in a binary search tree [5, Section 6.2.2]. The numbers n_j and d_j are stored in random-access tables as above. The tree consists of a set of *nodes*. Each node is either an *internal node* or a *leaf*. Each internal node contains one of the numbers a_i for some $1 \leq i \leq k - 1$, and also contains two pointers to the node's left child and right child. Each leaf contains a pair (b_i, c_i) for some $1 \leq i \leq k$. If the leaves are numbered from left to right, then the ith leaf contains (b_i, c_i) for the ith interval. To compute the mapping given a bucket id x, starting at the root of the tree, compare x with the number a stored at the node. If $x < a$, then follow the pointer to the left child; if $x \geq a$, then follow the pointer to the right child. Continue until a leaf is reached, and use the b_i and c_i stored there to compute the bin id as in the last two lines of Algorithm 1. Initially, when $m = 0$ and no expansions have occurred, the tree consists of a single leaf containing $(0, 0)$.

An advantage of the tree representation over the table representation is that, when expanding the mapping, new copies of the representation are not created. The existing (tree) representation is simply modified by adding nodes and pointers. In particular, if the action A_i is *Split*(s, c), then the ith leaf (containing (b_i, c_i)) becomes an internal node whose children are leaves. After the action is processed, this internal node contains s, its left child contains (b_i, c_i), and its right child contains $(m + 1, c)$.

Mapping complexity is proportional to the depth of the tree. An upper bound on the depth is m, since each expansion can increase depth by at most one. The depth (and, therefore, the mapping complexity) can be reduced to $O(\log m)$ by rebalancing the tree, using the tree transformations of AVL trees [1] (see also Section 6.2.3 of [5]), whenever a *Split* action is processed. This requires keeping an additional two bits of balance information in each internal node. Mapping storage is again $O(m^2)$, although the constant factor here is larger than for the table representation, due to the pointers and balance information.

3.7. *Handling Expansion in the Number of Buckets.* The purpose of this section is to describe how the basic expansion procedure of Section 3.4 can be modified to handle the

case where the number of buckets increases from B to some $B' > B$. We suppose that the number of bins increases as before, from n_m to n_{m+1}, although the case where the number of bins remains unchanged can be handled as a special case of the description below. We omit certain details such as the computation of new rank adjustments, since the computations are similar to those of Section 3.4 and can easily be filled in by the reader.

Define an *old block* to be any block j with $0 \leq j \leq m$. First, for each old block j, compute the quantity t_j^{before} = the number of buckets in block j before the expansion, and t_j^{after} = the number of buckets in block j after the expansion. The numbers t_j^{after} are computed as in the first part of Algorithm 2 using n_{m+1} and B', and the t_j^{before} are computed similarly using n_m and B. Let $\delta_j = t_j^{\text{after}} - t_j^{\text{before}}$. Thus, if $\delta_j < 0$, then buckets must be removed from block j; if $\delta_j > 0$, then buckets must be added to block j; and if $\delta_j = 0$, then the size of block j does not change. Since mappings are standard balanced mappings, it is easy to see that it is impossible for some old block to decrease in size while another old block increases in size. Therefore, exactly one of the following two cases must occur:

Case 1. $\delta_j \geq 0$ for all j.

Case 2. $\delta_j \leq 0$ for all j, and there exists at least one j with $\delta_j < 0$.

In Case 1 the "interval" $[B, B')$ of new buckets is split into several new intervals. For each j with $\delta_j > 0$, there will be a new interval mapped to block j, with the size of the new interval equal to δ_j so that block j reaches its correct size t_j^{after}. The remaining new interval is mapped to the new block $m+1$. (This must cause block $m+1$ to reach its correct size for the following reason. Each block j with $0 \leq j \leq m$ reaches its correct size t_j^{after}, which is the size of block j under any standard balanced mapping from B' buckets to n' bins. Since the sum of the sizes of all blocks $0 \leq j \leq m+1$ is B', block $m+1$ must reach its correct size as well.) The order in which new intervals are assigned to blocks can be arbitrary, except for the following requirement. Let $R = [a_{k-1}, a_k) = [a_{k-1}, B)$ be the rightmost interval before the expansion. If R is mapped to a block j with $\delta_j > 0$ in the old mapping, then map a new interval of the form $[B, s)$ to block j in the new mapping. Then combine the two intervals $[a_{k-1}, B)$ and $[B, s)$ into a single interval $[a_{k-1}, s)$.

In Case 2 the second part of Algorithm 2 is used to find the appropriate action for each of the old intervals, where t_j in the algorithm is t_j^{after}. The new interval $[B, B')$ is mapped to block $m+1$. If the interval R is either moved or split, then combine the interval of the form $[s, B)$ (mapped to the new block $m+1$ after the expansion) with the interval $[B, B')$ to obtain a single interval $[s, B')$. (In both cases the combination of two intervals into one is done to allow us to obtain the same bound on the number of intervals as was obtained in Lemma 3.1.)

Notice that, in many applications, the new buckets (in the interval $[B, B')$) are initially empty. When this is the case, no physical movement of data is needed for the new buckets.

We now show that the bound of Lemma 3.1 still holds, so that all the previous complexity and storage bounds still hold for the case where the number of buckets can also increase.

Table 1. Comparison of efficiency.

	Mapping complexity	Mapping storage	Bucket relocation
Round-Robin	1	1	Large
Stored-Directory	1	B	Minimum
Interval-Round-Robin	$O(\log m)$	$O(m^2)$	Minimum

LEMMA 3.2. *If k intervals are produced as the result of m expansions to the number of bins and the number of buckets, then*

$$k \leq \tfrac{1}{2}m(m+1)+1.$$

PROOF. The proof proceeds by induction as in the proof of Lemma 3.1. We just have to show that the $(m+1)$st expansion creates at most $m+1$ new intervals. As in the previous proof, just before the $(m+1)$st expansion there are at most $m+1$ blocks. Consider separately the two cases above.

In Case 1 an obvious bound is that the "interval" $[B, B')$ is split into at most $m+2$ new intervals, one for each of the old blocks $0, 1, \ldots, m$, and one for the new block $m+1$. However, it is possible to reduce this bound by 1. Consider two subcases. First, if the rightmost interval R is mapped to a block j with $\delta_j = 0$, then $[B, B')$ is split into at most $m+1$ new intervals, since a new interval is not needed for block j. Second, if R is mapped to a block j with $\delta_j > 0$, then $[B, B')$ could initially be split into $m+2$ new intervals, but then one of these new intervals is combined with an old interval into a single interval.

In Case 2 an obvious bound is again $m+2$: as in the proof of Lemma 3.1, at most $m+1$ old intervals are split (one for each old block), and together with the new interval $[B, B')$ this gives a total of at most $m+2$. We again improve this bound by 1, considering two subcases. First, suppose that the rightmost interval R is not moved or split. Let j be the block to which R is mapped under the old mapping. Since R is the rightmost interval under the old mapping, it follows that no interval mapped to block j is moved or split (i.e., $\delta_j = 0$), so at most m old intervals are split, rather than $m+1$. Second, if R is moved or split, then $m+2$ new intervals could be created initially, but then two intervals are combined into one. □

3.8. *Comparison of Efficiency.* Properties of the RR method, the SD method, and the IRR method are summarized in Table 1. All three methods produce optimally balanced mappings.

Another mapping method which has been proposed is the Cascaded-Round-Robin (CRR) method [2]. In the basic version of the method, mapping complexity is $O(m)$ and mapping storage is $m+2$ after m expansions; there is no bucket movement among existing bins during an expansion, although the mapping can become somewhat imbalanced after expansions. An extension to the basic version does rebalancing when the imbalance exceeds a user-defined threshold, although this might cause extra bucket relocations. (A side benefit of the rebalancing is that mapping complexity and storage decrease to

small constants, independent of m.) Both the CRR and the RR methods could perform better than IRR if access to buckets is not uniform, but higher numbered buckets are "hotter," i.e., receive more accesses than lower numbered buckets. CRR and RR spread hot buckets fairly evenly across the bins, whereas IRR will concentrate hot buckets in newly added bins.

4. History Independence. In this paper we consider various mappings f from the set of bucket id's to the set of bin id's. Thus, if x is a bucket id, then $y = f(x)$ is the corresponding bin id. On the face of it, f has only one parameter, namely, the bucket id x. There are, however, other parameters that are implicit. That is, there are really four parameters:

1. The bucket id x.
2. The number B of buckets.
3. The number n of bins.
4. The history H that describes how the number of buckets and bins has varied over time.

For definiteness, we take H to be a vector $\langle (B_0, n_0), \ldots, (B_m, n_m) \rangle$ of ordered pairs. Thus, B_0 is the initial number of buckets and n_0 is the initial number of bins, and so on. We are concerned in this paper only with situations where the number of bins and the number of buckets never decrease. Therefore, we restrict attention to histories where $n_i \leq n_{i+1}$ and $B_i \leq B_{i+1}$ for $0 \leq i < m$. In order that m truly reflect the number of expansions, we require that $(B_i, n_i) \neq (B_{i+1}, n_{i+1})$ for $0 \leq i < m$. We say that the parameters are *consistent* if

1. $0 \leq x < B$.
2. $0 < n \leq B$.
3. $B = B_m$.
4. $n = n_m$.

Intuitively, the last two conditions simply say that according to the history, the number of current buckets is B and the number of current bins is n.

Each algorithm gives us a function F such that if the parameters x, B, n, H are consistent, then $F(x, B, n, H)$ is the bin id corresponding to bucket id x, if there are B buckets and n bins, and the history is H. A goal of this paper is to consider mappings that are optimal in both balance and relocation. We call such functions F *optimal*. In this section we consider the question of when there are optimal mappings that are independent of some of the four parameters listed above. In particular, we are interested in *history-independent* mappings, that is, functions F such that $F(x, B, n, H) = F(x, B, n, H')$ for every choice of H, H' (as long as the parameters x, B, n, H and the parameters x, B, n, H' are each consistent). Intuitively, if F is history-independent, then an algorithm for computing F does not need to "know" (or "remember") the history.

In this section we prove that, in general, there is no history-independent mapping that is optimal. If, however, the number B of buckets is held fixed, then there is a history-independent mapping that is optimal. This latter result says that, for each B, there is an optimal function F_B such that $F_B(x, B, n, H) = F_B(x, B, n, H')$ for every choice

of H, H' (assuming that the first component of every entry of the history H is B, and similarly for H'). Intuitively, F_B has only two parameters, namely the bucket id x and the number n of bins. Can we reduce the number of parameters even further, so that an optimal F_B is not only history-independent, but also independent of the number B of buckets? We show that this is not possible. Thus, it is necessary for an optimal mapping to "know" the number of buckets and the number of bins.

It is interesting how the question of the existence of history-independent mappings arose historically. We thought it was quite possible that there is some optimal mapping that is elegant and simple that we just had not thought of. By making the problem even more constrained (by imposing the condition of history independence), it seemed possible that this would cause a unique solution, which might well be that elegant mapping we could not find. As we show in this section, when we allow the number of buckets and bins to vary, then with this additional constraint of history independence not only is there at most one solution, there is no solution!

The first result in this section deals with the important special case where the number of buckets is held fixed. In this case we show that there is a history-independent mapping that is optimal.

THEOREM 4.1. *If the number of buckets is held fixed, then there is a history-independent mapping that is optimal.*

PROOF. Assume that there are B buckets. By induction on n, we define a mapping f_n of the B buckets into n bins that is a standard balanced mapping (as defined in Section 2). Thus, $f_n(x)$ is the bin id when x is the bucket id. Let f_1 be the mapping that maps every bucket into bin 0. Assume that $n > 1$ and that the mapping f_{n-1} has been defined; we now define f_n. Since f_{n-1} is a standard balanced mapping on $n-1$ bins, it is easy to see that in going from $n-1$ bins to n bins, there is a standard balanced mapping that is optimal, where the only buckets that are moved are ones that move from old bins (the first $n-1$ bins) to the new nth bin. Define f_n to be one such mapping.

By construction, for each $n > 1$, we know that in passing from $n-1$ bins (with mapping f_{n-1}) to n bins (with mapping f_n), the only buckets that are moved are ones that move from old bins to the new bin. It is not hard to see that it follows that for arbitrary n', n'' with $n' < n''$, in passing from n' bins (with mapping $f_{n'}$) to n'' bins (with mapping $f_{n''}$), the only buckets that are moved are ones that move from old bins (the first n' bins) to new bins (the last $n''-n'$ bins). Since both $f_{n'}$ and $f_{n''}$ are standard balanced mappings, it is not hard to see that a minimal number of buckets are relocated.

We now define $f(x, n)$ to be $f_n(x)$. From what we have said, the mapping f is a history-independent mapping that is optimal. □

REMARK. The complexity (number of operations) required to compute $f_n(x)$ appears to be large compared with that of the IRR mapping. A rough upper bound on the complexity of computing f_n by the obvious algorithm can be given. To compute $f_n(x)$, the obvious algorithm finds complete representations of $f_1, f_2, \ldots, f_n$ in sequence. Then $f_n(x)$ is found from the representation of f_n. The complexity thus depends on the method used to represent these functions. If we use the IRR method, then the complexity of computing a representation of f_{i+1} from a representation of f_i is easily seen to be $O(\min\{i^2, B\})$, since

there are at most this many intervals in the representation of f_i, and since the complexity of computing the new representation grows linearly in the number of intervals if the actions are processed in a single pass through the intervals. From this, an upper bound on the complexity of computing f_n is $O(\min\{n^3, nB\})$. In addition, storage $O(\min\{n^2, B\})$ is sufficient to compute the mapping.

We now show that the condition in Theorem 4.1 that the number of buckets is held fixed is essential. Thus, there is no single history-independent mapping that is optimal and that is also bucket-independent (that is, whose value does not depend on the number of buckets). Intuitively, this tells us that the minimal amount of information that must be "remembered" in order to define an optimal mapping is the number of buckets and the number of bins. (The restriction to optimal mappings is necessary to obtain this result, since the RR mapping, $f(x) = x \bmod n$, is both history-independent and bucket-independent, although it is not optimal since its bucket relocation is not minimal.)

THEOREM 4.2. *There is no history-independent, bucket-independent mapping that is optimal.*

PROOF. Assume that there is a history-independent, bucket-independent mapping that is optimal. Let $f(x, n)$ be the bin id when x is the bucket id and n is the number of bins (by assumption, we can take f to be a function of only the bucket id x and the number n of bins).

When there are exactly four buckets, we know that there is a permutation i_0, i_1, i_2, i_3 of 0, 1, 2, 3 such that $f(i_j, 4) = j$, for $j = 0, 1, 2, 3$ (this simply says that when there are exactly four buckets and four bins, then no bin is empty; this follows from the balance condition). By renumbering the buckets if necessary, we can assume without loss of generality that $f(j, 4) = j$, for $j = 0, 1, 2, 3$. Even though we obtained this under the assumption that there are exactly four buckets, we make use of the fact that it holds independent of the number of buckets.

We now claim that $f(j, 3) = j$ for $j = 0, 1, 2$. This is because if, say, $f(1, 3) = k$ and $k \neq 1$, then in going from three bins to four bins, there would be unnecessary additional movement of buckets, because bucket 1 would have to be moved from bin k to bin 1. This is a contradiction, because it is easy to see that when the number of bins is increased, each optimal mapping can only move buckets from old bins to new bins, and never move buckets from an old bin to a different old bin. By a similar argument, $f(j, 2) = j$ for $j = 0, 1$.

Since $f(0, 2) = 0$ and $f(1, 2) = 1$, it follows that when there are exactly three buckets and two bins, then bucket 0 is in bin 0, and bucket 1 is in bin 1. Bucket 2 could be in either bin 0 or bin 1. Let us assume that bucket 2 is in bin 0 (that is, $f(2, 2) = 0$); a completely analogous argument will work if bucket 2 is in bin 1.

Since $f(0, 2) = 0$, $f(1, 2) = 1$, and $f(2, 2) = 0$, it follows that $f(3, 2) = 1$. This follows from the balance condition when there are exactly four buckets and two bins.

Now $f(4, 2) = 0$ or $f(4, 2) = 1$. To help decide which is the case, we consider the situation where there are exactly five buckets. If $f(4, 2) = 0$, then in passing from two bins to three bins, only one bucket needs to move to another bin, namely, bucket 2 from bin 0 to bin 2 (this bucket must move, since we already obtained the fact that

$f(2, 3) = 2$). If, however, $f(4, 2) = 1$, then in passing from two bins to three bins, two buckets need to move to another bin: this is because in addition to bucket 2 moving from bin 0 to bin 2, also bucket 3 or bucket 4 must move, since when there are just two bins, bin 1 contains three buckets (namely, buckets 1, 3, and 4). So if $f(4, 2) = 1$, then optimality would be violated. Therefore, $f(4, 2) = 0$.

We have shown that $f(0, 2) = f(2, 2) = f(4, 2) = 0$, and $f(1, 2) = f(3, 2) = 1$. By the balance condition when there are exactly six buckets and two bins, it follows that $f(5, 2) = 1$.

If there are exactly six buckets, then in going from two bins to four bins, we already know that bucket 2 moves from bin 0 to bin 2, and bucket 3 moves from bin 1 to bin 3. This is sufficient for balance, since this leaves the bins balanced (with buckets 0 and 4 in bin 0, buckets 1 and 5 in bin 1, bucket 2 in bin 2, and bucket 3 in bin 3). So by optimality, the only buckets that move in going from two bins to four bins are buckets 2 and 3, and so in particular bucket 5 does not move, that is, $f(5, 4) = f(5, 2) = 1$.

If there are exactly five buckets, then in going from two bins to three bins, we already know that bucket 2 moves from bin 0 to bin 2. This is sufficient for balance, since this leaves the bins balanced (with buckets 0 and 4 in bin 0, buckets 1 and 3 in bin 1, and bucket 2 in bin 2). So by optimality, buckets 3 and 4 do not move in going from two bins to three bins, and so $f(3, 3) = f(3, 2) = 1$, and $f(4, 3) = f(4, 2) = 0$. Since $f(0, 3) = f(4, 3) = 0$, $f(1, 3) = f(3, 3) = 1$, and $f(2, 3) = 2$, it follows that when there are exactly six buckets and three bins, then bucket 5 must go in bin 2 for balance, and so $f(5, 3) = 2$.

If there are exactly six buckets, then in going from three bins to four bins, we already know that bucket 3 moves from bin 1 to bin 3. This is sufficient for balance, since this leaves the bins balanced (with buckets 0 and 4 in bin 0, bucket 1 in bin 1, buckets 2 and 5 in bin 2, and bucket 3 in bin 3). So by optimality, the only bucket that moves in going from three bins to four bins is bucket 3, and so in particular bucket 5 does not move, that is, $f(5, 4) = f(5, 3) = 2$.

We have shown that $f(5, 4) = 1$ and $f(5, 4) = 2$. This contradiction proves the theorem. □

As we now show, it follows easily from Theorem 4.2 that in general (when the number of buckets and number of bins can both increase), there is no history-independent mapping that is optimal. This is in contrast to the situation in Theorem 4.1, which says that if only the number of bins is allowed to increase, then there is a history-independent mapping that is optimal.

THEOREM 4.3. *If the number of buckets and number of bins are both allowed to increase, then there is no history-independent mapping that is optimal.*

PROOF. Assume that there were such a mapping. Let $f(x, B, n)$ be the bin id when x is the bucket id, B is the number of buckets, and n is the number of bins (by assumption, we can take f to be history-independent). We now show that the mapping f is bucket-independent, which contradicts Theorem 4.2.

Assume that f is not bucket-independent. Then there are parameters x, B, B', n such that $f(x, B, n) \neq f(x, B', n)$. Since $B \neq B'$, we can assume without loss of generality

that $B < B'$. Consider a situation where there are B buckets and n bins, and then the number of buckets is increased to B' while the number of bins remains n. An optimal mapping would not move any of the old buckets, but simply add the new buckets into existing bins in a balanced manner. However, according to f, the old bucket x is moved from bin $f(x, B, n)$ to bin $f(x, B', n)$. This is a contradiction. □

5. Conclusion and Open Questions. If we restrict attention to optimal mapping methods, i.e., those that are optimal in both balance and relocation, the IRR method gives a new point on the "tradeoff" between the time complexity (number of operations) of computing the mapping, and the storage needed for a representation of the mapping. One extreme point of this tradeoff is the SD method, where the complexity is small (one directory access) but the storage is large (B numbers). In the case that the number of buckets is fixed, an opposite extreme point is provided by Theorem 4.1, where the storage is small (two numbers, n and B) but the complexity appears to be large (growing as $\min\{n^3, nB\}$). An obvious question from a practical viewpoint is to give new and improved points on the tradeoff. A question of theoretical interest is to give *lower* bounds on the tradeoff between complexity and storage. Informally, such a result would show that if the storage is sufficiently "small," then a "large" number of operations are needed to compute the mapping. To prove such a result, an appropriate model of computation would first have to be defined where the allowed operations are precisely defined. The small storage given by Theorem 4.1 depends on the number of buckets remaining fixed. It is an open question as to what the minimal storage is when the number of buckets does not remain fixed. Although we have considered only optimal methods in this paper, it is also reasonable to ask whether small deviations from optimality in balance and/or relocation can lead to significant improvements in the other measures.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR*, **146** (1962), 263–266; English translation: *Soviet Math. Dokl.*, **3** (1962), 1259–1263.

[2] D. M. Choy, A growth-oriented scheme to distribute objects to multiple storage locations, to appear.

[3] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen, The Gamma database machine project, *IEEE Trans. Knowledge Data Engrg.*, **2** (1990), 44–62.

[4] IBM, *IBM* 3514 *Quick Reference Manual*, Publication SA21-9613, 1993.

[5] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, MA, 1973.

[6] D. A. Patterson, G. Gibson, and R. H. Katz, A case for redundant arrays of inexpensive disks (RAID), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109–116.

[7] Teradata, *DBC/*1012 *Database Computer System Manual Release* 2.0, Document C10-0001-02, Teradata Corp., Nov. 1985.