RETRIEVAL OF RECORDS IN KEY SEQUENCE
FROM DIRECT ACCESS HASH FILES

by: Matthew Morgenstern

DCS-TR-#83

9/79

Dept. of Computer Science
Rutgers, The State Univ.
New Brunswick, New Jersey
08903
RETRIEVAL OF RECORDS IN KEY SEQUENCE FROM DIRECT ACCESS HASH FILES

Matthew Morgenstern
Department of Computer Science
Rutgers University
September 1979

ABSTRACT

The desirable random access characteristics of hash tables until now have been considered incompatible with the ability to retrieve the record next in key sequence. In this paper, algorithms are developed for such Get Next retrieval from a specific category of hash tables. It is shown that the random access performance is unchanged, and that the overhead for the Get Next operations can be quite satisfactory. These characteristics can be particularly advantageous for file and database system access. Implications for choice of blocking factors on secondary storage devices are assessed.

Extensions of the basic technique serve to expand the scope of applications, allow for distribution dependent hashing schemes, and describe a spectrum of schemes in which tradeoffs can be made between the efficiencies of random access and Get Next retrievals.

Index Terms: Hash tables, scatter storage, data base access, sequential retrieval, random access, searching, index structure, file organization, distribution dependent hashing, bucket size.

CR Categories: 3.7, 3.72, 3.74, 4.33, 4.34
RETRIEVAL OF RECORDS IN KEY SEQUENCE
FROM DIRECT ACCESS HASH FILES

Matthew Morgenstern

I. OVERVIEW

Files which are organized and accessed using hash table methods provide rapid average insertion and retrieval times. The performance benefits can be quite significant for very large files stored on disks or similar secondary storage media.

Until now it had been thought that the "Get Next record in key sequence" operation could not be implemented in any feasible manner for hash table organization [6], [10], [16]. This belief has been clearly stated in the literature. "No concept of serial access exists within the direct file which uses randomizing for its key-to-address translation. ... If the key for the next record is not known, no practical method exists to retrieve that record" [16]. The basic cause for this assumption was the apparent random scattering of the keys over the table by the key to address transformation. Thus applications which might require retrieval of the next record in key sequence even occasionally, could not make use of the desirable direct access capabilities of hash table organization.

In this paper, we shall develop simple techniques for storing the records in a hash table so that we can achieve the Get Next record capability with limited effort while retaining the random access properties. We will find that in appropriate circumstances there is virtually no overhead in space, time, nor algorithmic complexity for random insertion, deletion, and retrieval. In fact, there is a potential for a small net savings in space. The basic technique does not change the distribution of keys over the table, so that no additional overflow records are created.
The limited overhead is incurred only for the Get Next operation itself. The amount of this overhead depends upon how well the parameters of the given situation fit the category prescribed in this paper. Deviations from this optimal category can be dealt with in several ways.

After presenting and analyzing the basic algorithm, including overflow handling, we shall:

1. extend the basic approach to a larger range of situations;
2. develop additional techniques that can retrieve a group of \( N \) records in key sequence significantly faster than \( N \) times the overhead for separate Get Next operations — this would make possible sequential processing of part, or even all, of the hash table file in key sequence;
3. show how partial knowledge about the distribution of keys in the key space can be used to reduce overflows and improve performance; and finally
4. describe a file organization model encompassing these techniques, and such that as the parameters of this model are varied, we obtain a spectrum of file organizations which range from sequential storage on the one hand, to the standard direct access hash table organization at the other end of the spectrum.

While the basic Get Next procedure is almost "free" under the appropriate conditions, these additional techniques offer advantages with respect to certain tradeoffs, which we will assess. They can be used to create various performance alternatives based upon the relative frequencies of direct access, retrieval of next individual record in key sequence, and retrieval of a group of records in key order.

II. BACKGROUND

We start with a set of \( N \) records, each of which is uniquely identified by a key, \( K_i \), drawn from the key space \( W \) of all potentially constructible keys. Generally the size of the key space \( W \) is very much larger than the number of records \( N \). A table of size \( M \) is used to store these records with a relatively small percentage of unused slots.
A key-to-address transformation (or hashing function) interprets each key Ki as a bit string or extended integer and transforms it into a relative address in the table. Ideally each actual key should be mapped to a unique table address. This is often expressed in a weaker form by seeking transformations which distribute the keys uniformly over the table regardless of any "reasonable" clustering or correlation of the keys in the key space [15]. Of course, given prior knowledge of the transformation to be used (with \( N << \text{size}(W) \)), it will always be possible to artificially construct a set of distinct keys which will be distributed extremely poorly over the table, no matter how good the transformation is in general.

Much analysis of key transformations by various authors [3, 4, 6, 7, 8, 15] has reinforced the conclusion that the "division method" is preferred. This view is well expressed by Lam [7]: "Faced with an arbitrary key set, the selection of a transformation technique is obvious: the division method is preferred. While other techniques may sometimes perform better, one also risks obtaining inferior results more often."

To review the division method briefly, a divisor \( D \) is chosen so as to be less than or equal to the table size \( M \). Integer division of the key \( K \) by \( D \) yields a remainder \( R \) which is used as the address into the table. The choice of the divisor \( D \) significantly affects the number of distinct actual keys which transform to the same table address — such coincidences are called "collisions". This address may refer to a fixed size bucket which can hold several records, or to a storage slot for a single record (effectively a bucket of size 1). Collisions in excess of the bucket size are dealt with by an overflow handling strategy. The presence of overflow records prevents the hash table access method from fully achieving the ideal performance of a single probe (I/O event if the file is on disk) to retrieve any record based on its key.

Thus it is important to choose a divisor which is close to the desired table size and which minimizes collisions and overflows. Several suggestions appear in the literature [3, 6]. A relatively simple rule for generally good performance is advanced by Severance [15]: choose the divisor \( D \) so as not to contain any prime factor below, say, 20.
III. THE GET NEXT RETRIEVAL CAPABILITY

If we have just retrieved a record from the hash table based upon a given key, and wish to obtain the next record in key sequence, what are our alternatives? We could sort the records, or the pairs (key, record-location), into key sequence if we expect to need successive records frequently. But keeping this sorted copy up to date in the presence of insertions and deletions could be a significant problem. Alternatively, given just the hash table file, we could search the whole file once through for the smallest key value which is greater than the current key. But this is infeasible in very large files. Even for medium size files, this would be too costly if done more than very infrequently.

It is important to notice that if we did try to search through the table for the next larger key, we could not stop our search until we had inspected the entire table. Thus even if we encountered what turned out to be the desired record relatively close to our starting point, we would not know that we had in fact found it. The reason is that we were looking across the whole file for the smallest key which is larger than the current key. Thus the whole file must be inspected without exception.

A. The Algorithm

The essence of the proposed Get Next capability is to:

(a) Ensure that the desired next record will be close to its predecessor, on the average;

(b) Mark each record when it is first inserted into the table, so that in any subsequent retrieval, we can identify the next record and key as soon as we find it. (We will call this identification the Level Number of the record, and will derive it momentarily.);

(c) Not introduce overhead for random insertion, retrieval, and deletion. The limited overhead will be encountered only during the Get Next operation.
To accomplish this objective we will require that the division method of key transformation be used — it is the most desirable of all the key transformation methods for hash tables in general, as discussed above.

To understand the motivation for the algorithm, consider the division method itself for a moment. The transformation from the key space $W$ to the hash table addresses is a multi-valued mapping. We can let $L$ describe the "level" or surface number of the mapping, and $H$ be the location within that level. $L$ is the Level Number referred to above. In terms of this multi-level mapping, it can be seen that the next larger key will be the first key following the current key on the same level, or else the first key on the next level of the mapping after wrap-around.

Also, just to simplify the discussion initially, we will assume that overflow records are stored in a separate overflow area rather than in the main table, and are accessible via a chain from their home address (the address to which their key is transformed). Naturally we need to examine the entries in the overflow chain for one table address before considering the next table address. Other overflow techniques can be used, and later we will see that maintaining the overflow entries within the table using the linear probing scheme has important advantages.

**Algorithm 1: Basic Get Next Retrieval**

The basic Get Next algorithm, then, is to start from the location of the current key, and to examine successive records in the table until one of three conditions arise indicating that we have found the next record in key sequence and thus may stop. If we reach the end of the table (e.g. if the current key is located near the physical end of the table) then we wrap around to the front of the table.

We will have found the desired next record as soon as one of these three conditions is satisfied:

1. Before reaching the end of the table and wrapping around, we find a record with the same Level Number as in our current record. The first such
record is the desired next one in key sequence.

(2) If we reach the end of the table and need to wrap around to the front, our criterion is to look for a record with a Level Number one greater than in our current record. The first such record is the desired next one in key sequence.

(3) There is a small possibility that one may return to the starting point, i.e. the current record, without satisfying either criterion above. (This unlikely occurrence can be prevented with the augmented approach presented below.) For this case, we need to have kept track of the record containing the smallest Level Number greater than the current record’s Level Number as we went through the table. If there was more than one such occurrence of this next largest Level Number, we want that record with the smallest home address in the table. In this case, the selected record is the desired next one in key sequence.

# # # #

We will show that in appropriate circumstances only a small fraction of the table need be examined on the average. To understand why this algorithm works, consider the Division method of key transformation, which we are using. The home address, \( H \), of the record with key \( K \) is the remainder after integer division by the divisor \( D \). Thus \( H = K \mod D \). Let \( L \) be the quotient of that integer division, \( K/D \). Then \( K = L \times D + H \).

Consider the progression of values for \((L, H)\) as some hypothetical key (variable) takes on successive key values from the key space \( W \). \( L \) will start and stay at zero as \( H \) sequences through the integers from zero to \( D - 1 \). Then \( L \) will increment by one and the process will repeat.

As indicated above, the next larger key which we want will be the "closest" key following the current key on either the same level, or else on the next level of the mapping. By "closest" we mean the next larger home address \( H \), modulo the table size. This explains cases 1 and 2 in the algorithm. Case 3, though unlikely to be needed, clearly holds since it finds the smallest \((L, H)\), and thus the smallest key, greater than the key of the current entry.
If our table addresses refer to buckets rather than to separate locations for individual records, our same algorithm can be used without modification. In particular, we may stop at the first record satisfying criteria (1) or (2) without searching to the end of the bucket. Although this may seem surprising at first, the reason is that all records stored in a bucket must have different Level Numbers -- so long as overflow is stored outside the table. Stated differently, using the division algorithm, collisions can occur only between key values which are on different surfaces of the multi-valued mapping to the table. All keys on a given surface (i.e. with the same Level Number) will map to distinct table addresses.

B. Evaluation

An interesting consequence of this latter interpretation of the division method is that keys differing from one another by less than the number of addresses, D, in the table will not collide. Thus keys which form a run of consecutive key values, as often occurs in real files, will not collide with one another if the length of the run is less than D. Such cannot be said for a theoretical hashing function which distributes each key with equal probability to the addresses in the table. So in this sense, the division method is a better transformation than a "perfect randomization" method. (The comparison of division and "perfect randomization" is analyzed in depth by Ghosh [4] and commented upon by Berstiss [2]).

We see that if we store the Level Number L we do not have to store the key K at all, since the key always can be reconstructed by $K = L \times D + H$. Furthermore, storing the Level Number L requires floor($\log_2 D$) fewer bits than does storing the full key K. (We will use "ceil" and "floor" functions to denote rounding up and rounding down, respectively, to obtain an integer result.) If each of the D home addresses refers to a bucket having b slots for b records, then we have the potential of saving $[D \times b \times \text{floor}(\log_2 D)]$ bits over the whole table -- assuming that overflow records are stored in a separate overflow area. This small savings in
space is due to the general observation that we need not store the entire key if we can reconstruct the key from the value that we do store and the address of the record. (This small space savings is applicable to any hash table method in which the key can be reconstructed [6, pp.543-4].) Of course, our pointer to the overflow chain may use some or all of these saved bits per record.

Notice that we have not reduced the efficiency of the direct access capabilities of the hash table organization. We will now evaluate the Get Next operation. For future reference we will state here the definition of the primary parameters we will use:

- \( K \) = key
- \( D \) = divisor, and number of addressable buckets in table
- \( H \) = home address of record = \( K \mod D \)
- \( L \) = Level Number = \( \text{floor}(K/D) \); thus \( K = L \times D + H \)
- \( N \) = number of records stored in table
- \( b \) = number of record slots in each bucket of the table
- \( M \) = table size = number of slots in the table = \( D \times b \)
- \( W \) = key space of all possible key values
- \( S \) = size of \( W \)
- \( a \) = \( N/M \) = load factor of table
- \( P_{	ext{nk}} \) = distance in hash table between two consecutive key values expressed as a fraction of the table size

Consider the Key Space \( W \), having \( S \) consecutive values for possible keys, of which only \( N \) keys are actually present. We seek the average distance in this Key Space between the keys of two consecutive records (the end of the key space is considered to wrap-around to the beginning of the key space). Clearly the sum of such inter-key distances must equal the size \( S \) of the Key Space for any set of \( N \) keys. Thus the average inter-key distance in the Key Space is just this sum \( S \), divided by the number of keys \( N \), i.e. \( S/N \). This is the reciprocal of the Key Space density.

From the characteristics of the division method, as discussed above, we can see that the distance in the Key Space \( W \) between two consecutive keys will map into the same distance in the hash table -- so long as this distance is less than the number of addresses, \( D \), in the table. In the
unlikely event that the distance in the Key Space between two specific consecutive keys is greater than D (the number of addresses in the table), the search in the table will stop when the whole table has been inspected. This event is unlikely because we only will be interested in cases for which \( S/N \ll D \) (i.e. the expected inter-key-space distance is much smaller than the table size).

The effect of this maximum on the search distance is to reduce the expected value of the inter-key distance in the table below the corresponding value in the Key Space. For the cases in which we will be interested, this effect is negligible. Thus, the expected value for the distance between consecutive keys in the table will be closely upper bounded by the corresponding average distance in the Key Space, namely \( S/N \). We will use this convenient result. Thus the fraction of the table, \( P_{nk} \), that would need to be inspected, on the average, is upper bounded by \( P_{nk} = S/(N D) \).

C. Range of Applicability

To help put this method in perspective, let us consider a simple example, and assume that table addresses reference individual record slots rather than buckets. If we were using a nine digit key (such as the social security number) to reference one million records stored in a hash table file which was 80\% loaded, then \( S = 10^{**9} \), \( N = 10^{**6} \), and the load factor \( a = .8 \). The table size \( D \) would need to be \( N/a = 1.25 \times 10^{**6} \). Hence \( P_{nk} = S/(N D) = 10^{**9}/(10^{**6} \times 1.25 \times 10^{**6}) = .0008 = 0.08\% \). Thus less than one-tenth of one percent of the table would need to be inspected on the average to find the next record in key sequence in this example.

To understand the general range of situations which are well suited to the basic Get Next method, assume that we can accept, on the average, inspection of some fraction \( f \) of the table (e.g. \( f = .001 \)) to find the next record in key sequence. Then we want \( P_{nk} = S/(N D) < f \). The number of records \( N \) is determined by the application. The size \( S \) of the key space is primarily determined by the application, though it can be reduced, as will
be discussed. We have direct control over the number of addresses \( D \) in the hash table, and thus can control \( \text{Fmk} \).

To consider buckets of size \( b \), notice that \( N = a \cdot M = a \cdot b \cdot D \), or \( D = N/ab \). Thus \( \text{Fmk} = S/(N \cdot D) = ab \cdot S/N \cdot b^2 \). This formula indicates what may be slightly surprising at first. Namely, that the larger we make the bucket size, \( b \), the worse the Get Next performance will be (not considering overflow). Actually the reason is simple: the larger the buckets, the fewer the number of addresses, \( D \), in a table which has the same total capacity. Alternatively stated, the average number of home addresses to be inspected to obtain the next record in key sequence is simply \( S/N \), regardless of the bucket size \( b \).

Since \( S \) and \( N \) are determined by the application, let us focus on the design parameters of the loading factor \( a \) and the bucket size \( b \) in the formula for \( \text{Fmk} \). The factor \( a \cdot b \) in the expression for \( \text{Fmk} \) is related to the random access behavior, in that the smaller the value of \( a \) and the larger the value of \( b \) the better the random access performance, in general (see [15] for detailed guidance). This criterion is in agreement with our goals here with respect to the choice of the loading factor \( a \), but seemingly is at odds with our goals for the choice of the bucket size \( b \), which we would like to be as small as possible for the reasons just explained.

It is useful to explore the choice of the bucket size \( b \) further. The larger the bucket size, the more keys can transform to that bucket address without producing an overflow. However, since the total table size is limited, there will be fewer bucket addresses in the table when the buckets are larger, and thus more records will have to map to each bucket. Nevertheless, previous literature indicates that larger buckets do tend to reduce the average number of overflow entries per bucket (averaged over the table) and the average number of buckets that need to be accessed. Severence [15] clearly explains this as: "intuitively, we find that as [the number of slots in a bucket] increases, local excesses and shortages cancel one another, and buckets overflow less frequently". The average number of overflows and the average buckets accessed have been analyzed in the literature and used as the basis for selecting the bucket size of files.
organized on secondary storage disks.

However, some fundamental assumptions of bucket size selection seem open to question when considering files stored on disks. The average number of overflow entries per bucket, or the average number of buckets accessed, are relevant parameters only to the extent that overflows cause additional I/O activity — which usually is the most costly of all the factors (the other factors being data transfer cost, and storage costs).

The unstated assumption is that an I/O event accesses only one bucket. This assumption ignores two simple possibilities, the most important of which is that several buckets could be stored as a single physical block for retrieval in a single I/O event. The second possibility is the use of chained channel I/O commands to retrieve several blocks in a single I/O event.

For previous uses of hash table files, there was not much reason to differentiate between bucket size and block size. But for our purposes of key sequence retrieval, there is a good reason: the smaller the buckets the smaller the fraction of the table that needs to be inspected to find the next record in key sequence. The same I/O efficiency of large buckets and blocks still can be obtained by having several small buckets in a single large block. Block sizes then can be chosen independently of bucket sizes and thus made relatively large based upon the tradeoffs of I/O event costs, transfer costs, main storage costs, and secondary storage costs; see Severance [15] for further guidance.

In choosing the actual bucket sizes, we also need to consider whether there would be an effect on the handling of overflow entries. In particular, large buckets have a "smoothing" effect on the overflows from neighboring separately addressed locations, as referred to above. However, this same smoothing effect can be achieved by the linear probing method of overflow handling, combined with the above choice of small buckets and a large block size. Linear probing also is desirable for our purposes of key sequence retrieval from files, as will be discussed shortly.
The conclusion is that bucket sizes can be kept rather small for the benefit of key sequence retrieval. And block sizes can be large, so that there would be no increase in the random access overhead (average number of I/O events) due to overflow entries. (As a side note, if buckets had been chosen large for some reason in a specific case, a similar effect of small bucket size for key sequence retrieval can be obtained by first scaling the key by the number of smaller "logical sub-buckets" that otherwise would have been used in the same space occupied by one large bucket.)

Returning to the selection of the design parameters as they affect \( F_{nk} \), we have seen that the bucket size \( b \) can be kept small. Also, reducing the loading factor \( a \) is of benefit to both random access and \textit{Get Next} performance, but may be constrained by the cost of external storage. For the given application's values of \( N \) and \( S \), and our choice of parameters \( a \) and \( b \), the resulting expression \( F_{nk} = (a \times b) \times S/N^{2} \) may represent an acceptable cost for the expected frequency of \textit{Get Next} operations. If not, or if further improvement in this cost is still sought, the effective size \( S \) of the key space can be reduced, but at some cost in direct access retrieval efficiency. One might use either scaling or truncation of the low order portion of the key (which is just scaling by a power of the radix). A new and potentially useful method for this purpose will be described in section 5.

For the moment, consider the auxiliary use of scaling to reduce the effective key space size. If used as the sole transformation, scaling would produce too many collisions. But here we would be using it to transform keys to a very much larger virtual address space (\( S' > N \)) than the final table size. Such partial scaling can be effective, though it still would create collisions when there is a "run" of consecutive key values. The division transformation then would be applied to this virtual scaled space to bring the reduced key into the range of the final table, and our \textit{Get Next} strategy could be applied with favorable behavior.

So far, we have indicated the scope of applicability of the basic technique as a function of the average overhead, \( F_{nk} \), which the user can accept for his frequency of using \textit{Get Next} operations. We also have indicated minor variations that are useful in selected circumstances. In a
later section, we will develop a compression technique which further extends the range of applications, and also can provide for retrieving a group of the next several records in key sequence much more efficiently than record at a time retrieval.

IV. OVERFLOW HANDLING

Earlier we simplified the discussion by assuming that the overflow would not be stored in the same table, but rather would be chained into a separate overflow area. This assumption is not necessary for our technique. (For discussions of overflow handling strategies see [6], [15].) In particular, the linear probing (open overflow) method often is used because it does not require additional space outside the table. With linear probing, when the home address for a new record is occupied, we still can insert it in the table by probing successive locations following the home address until we find an empty slot for this overflow record. A set of consecutive table slots which are all filled is referred to as a "cluster." In effect, blank table entries serve to delimit the cluster.

Retrieval using linear probing compares the desired key to each entry, beginning with the home address and continuing until the end of the cluster, wrapping around to the beginning of the table if needed. For this to work as intended, deletion must not change the extent of the cluster. Thus a newly freed entry can be flagged as "empty but once used" so that a new record can be inserted into it but such that retrieval will continue beyond it to the real blank entry and the end of the cluster.

When we seek one or several records next in key sequence using our previous algorithm, we must inspect the overflow records from one home address before being concerned with the next home address. If overflow records were maintained in a chain, we might need a separate I/O event for each record. On the other hand, if we use linear probing, such overflow entries will be stored close to each other and to their home address. We expect to store a number of locations from the table into a single physical
block, which is retrievable with a single I/O event. Thus the number of I/O events needed to find overflow records is significantly reduced using linear probing. This is desirable for both direct and Get Next retrieval. Linear probing, then, is the preferred method of overflow handling for our purposes.

We now present the Get Next record in key sequence algorithm for linear probing using buckets of size \( \geq 1 \). Note that the Level Number, as well as the original home address, must be stored in the record, thereby removing the small space savings mentioned previously. Naturally, the original key can be recomputed from these two parameters.

Consider the pairs (Level Number, Home address). We will establish the "less than" ordering between pairs:

\[(L_1, H_1) < (L_2, H_2) \text{ if and only if either}
\]

\[L_1 = L_2 \text{ and } H_1 < H_2, \text{ or else}
\]

\[L_1 < L_2.
\]

Thus, for an example with \( L_1 = L_2, H_1 < H_2, \) and \( L_2 < L_3 \), we would have the ordering \( (L_1, H_1) < (L_2, H_2) < (L_3, H_3) \).

ALGORITHM 2: Get Next Retrieval with Linear Probing

The Get Next operation begins at the home address \( H_0 \) of the current record, which has Level Number \( L_0 \). As before, we treat the table as circular, so that the last address is followed by the first address of the table.

As the algorithm inspects successive entries, it remembers which has the smallest \( (L_j, H_j) \) greater than \( (L_0, H_0) \). When we come to the end of a cluster, if this value of \( (L_j, H_j) \) is such that either (1) \( L_j = L_0 \) and \( H_j > H_0 \), or else (2) \( L_j = L_0 + 1 \) and \( H_j < H_0 \), then we have found the desired next record in key sequence and may stop. Note that a key satisfying (1) is smaller than a key satisfying (2). In the rare event that we return to the original home address \( H_0 \), we stop there, and the record having the thus far smallest \( (L_j, H_j) > (L_0, H_0) \) is the desired
It can be seen that this criterion is a restatement of the three conditions originally presented in Algorithm 1, and generalized to recognize the presence of clusters.

V. EXTENSIONS FOR KEY SEQUENCE RETRIEVAL

The basic algorithm described above for retrieval of the next record in key sequence created no additional overhead for random retrieval, insertion, and deletion under the appropriate range of circumstances. This was so because we do not modify the distribution of keys and records in the table when we use the basic algorithm. In this section, we will discuss techniques that do modify this distribution so as to further increase the range of applicability and/or to provide for even greater efficiency when retrieving a group of records in key sequence from the random access hash table. We begin by motivating and explaining the basic idea of what we shall call "compression" of the key space. Several different ways of utilizing this concept then will be explored.

For the moment, reconsider the idea behind simple scaling of the key space $S$ by some scaling factor $Y$. The resulting address space will have size $\text{ceil}(S/Y)$. Each consecutive set of $Y$ key values would be mapped into the same resulting address. Since typical distributions of actual keys often have runs of several consecutive key values [4], scaling will cause multiple collisions. For a run of length $r$, $[r - \text{ceil}(r/Y)]$ of these records will collide since the run will be mapped into $\text{ceil}(r/Y)$ addresses (ignoring the specific value of the starting key).
On the other hand, the division method of key transformation prevents such collisions of keys for a run of length \( r \leq d \), where \( d \) is the divisor being utilized, as noted above. We would like to preserve this property while reducing the effective size of the key space. To this end, imagine breaking the original key space of size \( S \) into \( VI \) intervals of equal size. Map each such interval of the original key space into a separate interval of an intermediate "Virtual Address Space". Do so by using the division transformation with divisor \( d \).

This Virtual Address Space then will be composed of \( VI \) intervals each of length \( d \), and its size will be \( A = d \times VI \). (If \( d \) does not evenly divide \( S \), we may think of extending the key space \( S \) to be a multiple of \( d \), rather than needing to treat the last interval specially.)

We thus have mapped a key space of size \( S \) into an address space of size \( A \). In effect we have "compressed" the key space by a compression factor \( C = S/A \). Just as there are \( VI \) intervals in the Virtual Address Space, so too there are \( VI \) intervals of the key space. These key space intervals are each of size \( C \times d = S/VI \), and each has been compressed into an interval of the address space, which is of size \( d = A/VI \). The mapping of a Key Space interval into a Virtual Address Space interval is itself a multi-level mapping, with the number of such levels being the compression ratio \( C \).

Notice that this compression ratio \( S/A \) is the same as if we had scaled the keys by this factor \( C = S/A \). But if there were runs of consecutive keys values, scaling definitely would have produced multiple collisions as shown above. Our compression method will cause zero collisions among keys of the same run, for runs of length \( r \leq d \). This Virtual Address Space then can be mapped to the actual hash table in any of several ways. We now will consider several types of compression.

It is useful to summarize here the primary additional parameters that will be utilized in the development of these extended techniques. These supplement the parameters summarized previously in section 3.2.
Summary Of Parameters For The Extended Techniques

d = size of interval in Virtual Address Space, and

number of addresses in one interval of the final
hash table, using the compression technique

VI = number of intervals in Virtual Address Space, and

in Key Space

A = size of Virtual Address Space = d * VI

I = number of physical intervals in hash table, using

compression = D/d

C = compression ratio = S/A

VIj = Virtual Interval Number in Virtual Address Space

for key Kj

Lj = Level number of key Kj

hj = number of the interval in hash table containing home

address of key Kj; also called home address

interval number

Hj = home address in hash table of key Kj (in interval hj)

Aj = address in hash table of start of interval hj

BI = number of physical device blocks per interval of table

For Variable Compression:

KVI = lowest key value possible in i-th group of intervals

all having same compression ratio.

VIi = virtual interval number of first interval in i-th

group

Ej = relative interval number, counting from beginning

of group, which is the home interval hj for key Kj

A. COMPRESSION TO A VIRTUAL ADDRESS SPACE

There are several ways to use this compression technique to transform

a key space of some application into a smaller Virtual Address Space. The

next three sections consider these alternatives.
We begin here with the case that the Virtual Space as a whole is to be mapped into the direct access hash table -- using the division technique again, but now with the table size D as the divisor. Also, the Virtual Space intervals are each the same size. The effect of this process is that the original large key space will be "compressed" into the smaller Virtual Address space with far fewer collisions than with scaling. This compressed space now is to be mapped to the table as usual.

As a consequence of this approach, a much larger range of situations can make effective use of the Get Next capability. Furthermore, the cost of the Get Next operation will be much less than it would have been without this intermediate mapping. On the other hand, using this approach, random access will be somewhat sensitive to very skewed distributions of actual keys. This sensitivity decreases as the compression factor decreases, and also as the number of intervals VI decreases.

In the rest of this subsection, we consider in greater detail the modification to the Get Next algorithm that is necessitated by this mapping through the Virtual Space. (The casual reader may wish to proceed directly to the next subsection.) We assume throughout that the interval divisor d and the number of addresses in the table, D, are chosen so as to give good key transformation results (as discussed earlier). Our discussion is simplified by also assuming that d eventually divides D, though this is not a necessary part of the underlying principle.

There are two ways to view the resulting scheme as compared to the basic algorithm. One of these is a rule for "stopping within an interval". It may be useful when the interval size d is large. We defer it to the end of this subsection where it is presented as Rule 1.

A.1 General Retrieval

Here we consider the interpretation which is most useful when the number of addresses, d, in an interval of the table is not necessarily large. The interval then would be composed of a few blocks and could be
read using one to a few I/O events.

We define the Virtual Interval number \( V_Ij \) for key \( K_j \) to be the interval number in the Virtual Address Space. It is the same as the interval number in the key space. For key \( K_j \),

\[
V_Ij = \text{floor}(K_j/(d \times C)).
\]

In the final hash table the number of intervals is \( I = D/d \) (we assume \( d \) evenly divides \( D \)). Then define the Level Number \( L_j \), and the home address interval \( h_j \) as

\[
L_j = \text{floor}(V_Ij/I) \\
h_j = V_Ij \text{ mod } I.
\]

In each record store the pair \((L_j, h_j)\) and the key \( K_j \).

For overflow handling, we will use a modified form of linear probing. If the home address for key \( K_0 \) is in table interval \( h_0 \), then when we probe successive locations of that interval for overflow, we wrap around within that same table interval \( h_0 \), rather than continuing into the next interval's slots. Only if the current interval is completely filled, do we continue our probing into the next table interval to complete our inspection of the cluster.

**ALGORITHM 3: Get Next Retrieval Using Virtual Space Compression and Linear Probing**

The current key is \( K_0 \) with \((L_0, h_0)\). We search the current table interval \( h_0 \) — which is not very large — for the smallest key \( K_i \) greater than \( K_0 \). If it satisfies \((L_i, h_i) = (L_0, h_0)\) then it is the desired next larger key of the whole table. Otherwise we inspect successive table intervals in their entirety (unless Rule 1 below is utilized). We follow overflow clusters when they extend beyond an interval, and treat the table as circular.
In general, for each successive table interval hi, we seek the smallest \( K_1 > K_0 \) in that interval. If that key \( K_1 \) with \((L_i, h_i)\) satisfies either: (1) \( L_i = L_0 \) and \( h_i \geq h_0 \), or (2) \( L_i = L_0 + 1 \) and \( h_i < h_0 \), then we have found the desired next record in key sequence and may stop. In the rare event that we return to the original home interval \( h_0 \), we stop there, and the record having the thus far smallest key \( K_1 > K_0 \) is the desired record.

###

The Get Next strategies of Algorithms 1 and 2 inspect S/N table addresses on the average to find the next record in key sequence. Using Algorithm 3, the average number of table addresses until the next key is slightly less than \( \text{ceil}(S/(N \times d \times C)) \times d \). Roughly, we have reduced the search by the compression factor \( C \), from S/N to S/(N\ C) table addresses. The larger the compression \( C \), the better will be Get Next retrieval. Typically, this compression (which is the ratio of key space size \( S \) to Virtual Space size \( A \)) will range from a few to many orders of magnitude.

Consider further this average number of Table addresses that may need to be inspected in order to find the next record in key sequence using Algorithm 3. We will call this number \( Tnk3 \).

\[
\left[ \frac{S}{N} \right] = Tnk3 = \text{ceil} \left( \frac{S}{N \times d \times C} \right) \times d.
\]

It can be seen that \( Tnk3 \) is barely dependent on \( d \). Also, \( S/N \) is just the ratio of the key space to the number of actual records, both of which are determined by the application. Thus the dominant effect will be due to \( C \). The larger \( C \) is, the better will be the Get Next performance.

On the other hand, the larger \( C \) is, the greater the likelihood that random access will be sensitive to and will suffer from a non-uniform distribution of actual keys over the key space. It is suggested that \( C \) be selected as small as is consistent with the requirement that \( Tnk3 \) yields a desirable value for average Get Next retrieval. Trial runs with sample
data could be used to evaluate whether higher values of \( d \) would yield acceptable random access performance on the average for the actual set of keys in a file.

In selecting the parameter \( d \), recall that \( d \) is the maximum size of a run of consecutive key values that can be stored without creating any collisions within that run. Thus larger values of \( d \) will benefit random access. For \texttt{Get Next} retrieval using Algorithm 3, the whole interval of size \( d \) must be processed, thus suggesting that \( d \) not be too large. For external storage on disk, a reasonable heuristic would be to choose \( d \) equal to one to several buffers in length, so that not more than these number of I/O events would be needed to retrieve the whole interval from disk.

The remaining factor to choose is the loading factor of the whole table, \( a \). It is the percentage of storage locations in the table which actually are occupied. Existing guidelines for random access should be utilized; see Severance [15]. Typically \( a \) will range between 50\% and 95\%, with the smaller values giving better random access characteristics, and the larger values benefiting \texttt{Get Next} performance. (We will see that \( S = L / (a \cdot b) \) so the average number of record slots to inspect, \( T_{nk3} \), may be expressed as \( \text{ceil}[L/(a \cdot b)] \cdot (b \cdot d) \), which is inversely proportional to the loading factor.)

In making the tradeoffs between better random access behavior versus greater overhead for \texttt{Get Next} operations, the relative frequencies of these operations should guide the choices. Clearly if \texttt{Get Next} were to be used only occasionally, emphasis should be placed on random access characteristics.

A.2 Additional Rule for Retrieval from Large Intervals

Here we consider the additional rule for "stopping within an interval". This rule is valid for any size interval, but the probability that it will allow us to stop before inspecting the whole interval increases as the ratio of the table interval size to the key space size
increases, i.e. as \( d/S \) increases. Thus it will be useful primarily when
the interval size \( d \) is large. We present it here for completeness.

Within each interval, we define for key \( K_j \) the Cycle Number
\( C_j = \text{floor}\left(\frac{(K_j \mod (d * C))}{d}\right) \). In effect, this is a relative level
number, with respect to the local mapping of that key space interval.
Recall that the compression factor is \( C = S/A \), where \( A \) is the size of the
Virtual Address Space. It can be seen that the compression \( C \) is the
maximum number of cycles within any interval; i.e. \( 0 \leq C_j < C \). The
Virtual Interval number in the Virtual Address Space (same as in the key
space) was defined above for key \( K_j \) as \( V_{ij} = \text{floor}(K_j/(d * C)) \). We also
need to define the interval remainder \( R_j = K_j \mod d \).

Then the Home address for key \( K_j \) is \( H_j \):
\[ H_j = [V_{ij} * d + R_j] \mod D. \]
For each record we store the triple \((V_{ij}, C_j, R_j)\). We can reconstruct the
original key by:
\[ K_j = V_{ij} * C * d + C_j * d + R_j \]
if we wish, or we can store the key directly.

If we imagine generating consecutive key values, we see that the
corresponding triples \((V_{ij}, C_j, R_j)\) will increase most rapidly in \( R_j \) for
\( 0 \leq R_j < d \), next in \( C_j \) for \( 0 \leq C_j < C \), and least rapidly in \( V_{ij} \) for
\( 0 \leq V_{ij} < \text{floor}(S/(C * d)) \).

Within a Virtual Interval \( V_{ij} \), we can view \( C_j \) as a local level number,
and \( R_j \) as a local home address. This observation gives rise to the
following rule.

* * *

Rule 1: Stopping Within an Interval,
for Virtual Space Compression

Let \( C_j \) be the relative level number and \( R_j \) the relative home address
of an entry, both for a given Virtual Interval \( V_{ij} \). The current key is \( K_0 \)
with \((C_0, R_0)\). To retrieve the entry next in the key sequence, we inspect
successive locations of the table interval after R0, wrapping around within
an interval to follow an overflow cluster to its end. (If an interval is
completely filled, the overflow cluster will extend into the next
interval.)

The next larger key Ki will be found before inspecting the whole
interval if, at the end of a cluster, the thus far smallest Ki > R0 has
\( V_{i1} = V_{i0} \) and either (1) \( C_i = C_0 \) and \( R_i > R_0 \); OR (2) \( C_i = C_0 + 1 \) and
\( R_i \leq R_0 \). Else we seek the smallest \( (C_i, R_i) \geq (C_0, R_0) \) with the same \( V_i \)
value (i.e. smallest \( K_i > R_0 \)), in this whole interval. If unsuccessful in
this interval, apply Algorithm 3 beginning with the next table interval.
In general, when using Algorithm 3, we may (re)apply this rule for stopping
before the end of a given interval if we take \( C_0 \) and \( R_0 \) as zero in
intervals beyond the first.

###

3. COMPRESSION DIRECT TO THE TABLE

A different but related technique which we call Direct Compression
will be of significant benefit when retrieving several to many records that
are next in key sequence. The cost of retrieving such a group of records
in key sequence will require on the average little more effort than
retrieving a single record next in key order.

In terms of the general method of compression to a virtual address
space, as described above, consider the number of levels, L, to be one. In
effect there would not be any Virtual Address Space. Rather the mapping
would be direct from the fixed size intervals of the key space to the
corresponding fixed sized intervals of the table.

Using direct compression, each interval of the key space maps uniquely
into a corresponding interval of the table, except for overflows from a
completely full interval. As a consequence, when seeking a group of
records in order by key value, we will need to retrieve just the minimum
number of table intervals -- we proceed to the next interval only when the
current interval is exhausted. Thus the I/O activity for such retrieval will be almost as good as for a sequentially ordered file, yet random access will be possible and modestly efficient in many cases.

The rest of this subsection analyzes Direct Compression in greater detail. Basically, the formulas and algorithms above for Virtual Space Compression are applicable for Direct Compression, though they can be somewhat simplified since the Virtual Space is the same as the target table.

In particular, since there is only one Level, L = 1, the mod D operation is not needed. We see that the compression factor will be \( C = a \times b \times S/N \). The loading factor a and the bucket size b can be chosen as before. For key \( K_j \) the home address interval is \( h_j = \text{floor}(K_j/(d \times C)) \), and the home address is:

\[ h_j = h_j \times d + K_j \mod d. \]

When just a single record next in key sequence is desired, Algorithm 3 can be applied. The home address interval, \( h_j \), is stored in each record, and all Level Numbers are zero since the Virtual Space is the same as the table. Thus condition 2 of Algorithm 3 does not apply. Also, wrapping around to the beginning of the table does not occur as generally as before, being limited now to overflow from the last interval of the table. Overflow can be handled by the modified form of linear probing -- involving wraparound within an interval -- as described above.

We usually will expect to find the record next in key sequence in the same interval as the current record. In addition, Rule 1 for "stopping within an interval" could reduce the Get Next retrieval time somewhat if the interval size d is moderately large.

Now consider the process for retrieving a group of several records in key sequence, as will be described in Algorithm 4 which follows. Aside from overflows, the organization for Direct Compression ensures that all keys in a given interval will be less than all keys in the next interval. This algorithm assumes that we are at a record with key \( K_0 \) and that we wish to retrieve a group of records that are next in key sequence.
Part (a) of the algorithm may be applied when we require a group of up to X records that are next in key sequence. Aside from overflows, intervals may be processed one at a time until this number is satisfied. Independent processing of each interval is possible since keys in a subsequent interval are greater than those in the current interval. If the main memory buffers cannot accommodate a full interval, then part (b) of the algorithm would be useful.

This algorithm is intended for use with Direct Compression, but actually can be utilized with any compression method if we apply it to one level of the mapping at a time.

**ALGORITHM 4: Retrieving A Group Of Records**

**In Key Sequence**

(a) In order to find a group of up to X records that are next in key sequence after the current record of key K0, we need search the current interval for the X smallest keys greater than K0, following into the next interval for overflows if necessary. If we find fewer than X such keys and X is a requirement, then we repeat this process in the subsequent interval(s) for the remaining numbers of keys. Aside from overflows, we may process one interval at a time until the required number of records have been retrieved. Overflows from a completely filled interval must be followed into the subsequent interval. If we are short of main memory, we may use part (b) of the algorithm.

(b) If any interval is larger than the main memory buffers, we may read one block of the interval into main memory at a time. For each block, we select from it only those keys greater than the current key K0; we sort these keys and save them (on auxiliary storage if necessary, using a small block size). After all blocks of the interval have been processed, perform a merge to obtain in key sequence all the keys of the interval greater than the current key. If additional records are needed, repeat with successive intervals.
For the rest of this paper, we will assume that the whole interval can be accomodated in main memory, unless otherwise stated.

The number of keys next in sequence that can be found in the current interval without additional I/O events will depend on where the current key value lies with respect to the range of actual key values mapped into that interval. The number of intervals in the table will be \( I = D/d \), so that on the average \( N/I \) records will occur in an interval. If the current key has equal probability of being any of the keys in the interval, then on the average \((N/I - 1)/2\) records next in key sequence will be available in the current interval without additional I/O. In any specific situation, if more keys are needed, then the next interval will be processed also, and subsequent intervals as needed.

It can be seen that the I/O per record accessed in key sequence will depend upon the number of desired records in key sequence obtained from an interval. This I/O per record will range from \( BI = \text{the number of blocks in an interval, if only one record is obtained; down to } BI/[N/I] \ll BI, \) if \( N/I \) records are obtained from the interval (the total contents of the interval). This assumes that all the desired records can be held in main memory.

This average I/O per desired record can be compared to the average number of I/O events to obtain a single record next in key sequence using the previous method of Virtual Space Compression. The I/O for the latter is the average number of table addresses to be inspected, \( Tm/k3 \), divided by the number of table addresses per block (i.e. addresses per interval divided by blocks per interval: \( d/BI \)):
\[
\lceil \frac{S/(N \cdot d \cdot C)}{d/BI} \rceil \cdot BI = \lceil \frac{S}{(N \cdot d \cdot C)} \rceil \cdot BI.
\]

Hence, Virtual Space Compression incurs I/O per record which is an integer multiple of the number of blocks per interval BI, while Direct Compression to the table incurs average I/O per desired record which is a fraction of BI. This is a significant advantage in reduced overhead when
processing groups of records that are in key sequence. The potential disadvantage is that this organization is more sensitive to the distribution of keys the smaller the interval size is. In particular, random access performance could be adversely effected, primarily in those cases where the actual distribution of keys is sufficiently skewed to cause a large number of overflows in a portion of the table.

When there is at least partial knowledge about this distribution of key values, perhaps from previous use of that file, then we can organize our table so as to alleviate this problem. In the next section we will show how to account for and eliminate those overflows due to known skewing of the key distribution.

C. VARIABLE COMPRESSION: Use of Partial Knowledge About The Key Distribution

It is sometimes the case that information is available about portions of the key distribution. We can make use of this knowledge to reduce the collisions and overflows in both the Virtual Space Compression method and the Direct Compression method. These previous compression methods can be sensitive to the distribution of actual key values. The method developed here, which is called Variable Compression, can be combined with these other techniques to reduce this sensitivity. Thus it should improve random access performance, as well as benefiting Get Next retrieval.

We also will see that Variable Compression allows the hash table to be stored in discontiguous sections. Each section can be at an arbitrary location. In fact, different sections can reside, temporarily or permanently, on different storage devices. This would allow additional access optimization based upon most recently used statistics, or other usage patterns.

Thus far, each fixed size interval of the key space mapped to a specific fixed size interval of the table. If an increased density of keys occurred in some interval of the key space, the capacity of the
corresponding table interval could be exceeded, and overflow records would be created. The basic idea of Variable Compression is that the intervals in the key space, and/or the intervals in the table, could be of different sizes based upon portions of the key distribution.

With Variable Compression, if we expect a disproportionate percentage of keys to fall in an interval of the key space, we could, for example, make the corresponding table interval(s) that much larger. The actual density of keys, then, will be more uniform across the table. Notice that, instead, we could decrease the interval size in the key space, rather than increasing the interval size in the table. That is, make each key space interval proportionately smaller where there is a higher density of keys and leave the table interval size unchanged. The result here too would be to distribute the keys more uniformly in the table, thereby reducing overflows.

Since we want to make maximum use of the main memory buffers, we will assume, for the rest of our discussion here, that the size of each interval in a given table will be fixed. Thus we will allow the key space intervals to have different sizes based upon the key distribution.

In the following part of this subsection we describe and analyze Variable Compression in greater detail. We show how it can be combined advantageously with Direct Compression and with Virtual Space compression. The ability to store the hash table in discontiguous sections, as noted above, also is described.

A set of consecutive key intervals which have the same compression ratio may be considered as a "group" of intervals. For the i-th such group, we need to retain in an index the lowest key value, KV_i, that could map into this group of intervals, and the interval number, VI_i, of the first interval in this group. VI_i represents the total number of Key Space or Virtual Address Space intervals that are utilized for key values lower than KV_i.
The compression factor for a group of intervals is selected based upon
the distribution or density of actual keys, as noted above. This
compression C is the ratio of the range of key values, (high key -
low key + 1), to be mapped into an interval of size d in the Virtual
Address Space, divided by d.

Vi is determined in the following manner as the index is being
established (or modified). Let X(i-1) represent the number of intervals
which (a) have consecutive key values, (b) have the same compression factor
and (c) are stored in a contiguous portion of the hash table beginning with
interval numbered Vi(i-1) -- these factors serve to define and delimit this
"group" of intervals. Then Vi = Vi(i-1) + X(i-1), with Vi = 0.

We will keep in main memory a static index which is a list containing
this mapping information (KV1, Vi) for each group of intervals in the hash
table. This list should be sorted on starting key value for rapid lookup
of the mapping function parameters. The following formulation of the
retrieval process is applicable to both Variable Direct Compression and
Variable Virtual Space Compression, with differences noted.

To retrieve a given key value Kj, we inspect the index of mapping
parameters to find an entry (KV1, Vi) such that KV1 <= Kj < KV(i+1). The
expression KV(i+1) is the next larger starting key value in the index, and
represents the i+1st entry.

We calculate the relative interval number for key Kj as

\[ k_j = \text{floor}\left(\frac{(K_j - KV_1) \times [(VI(i+1) - VI_i) / (KV(i+1) - KV_1)]}{1}\right) \]

Basically, Ej is the number, counted from the beginning of this group,
of the interval which contains the key value Kj. These intervals are with
respect to the key space and Virtual Address space. The quotient which
appears as the second factor could be stored for each group rather than
being recomputed each time.

As in the previous compression methods, the physical intervals in the
hash table are consecutively numbered beginning with zero. The interval
which contains the home address of a key Kj is referenced by its home
address interval number $h_j$. Both $h_j$ and the Level Number $L_j$ can be calculated as follows. (For Direct Compression, the Level Numbers will be zero as there is a single level of the mapping.) The factors $E_j$ and $V_i$ are as defined above, and $I$ is the total number of physical intervals in the hash table.

$$h_j = (V_i + E_j) \mod I$$
$$L_j = \text{floor}(\frac{V_i + E_j}{I})$$

If the hash table is stored contiguously, the physical Home Address in the table for key $K_j$ is

$$H_j = K_j \mod d + h_j \times d.$$  

Since table intervals are all of size $d$, the term $K_j \mod d$ represents the offset of the bucket for key $K_j$ with respect to the beginning of the home interval $h_j$

If the whole hash table is not stored in a single contiguous region, so that sections may be stored at arbitrarily chosen locations, then two cases arise. The first is when Direct (Variable) Compression is being utilized. A group of intervals then is delimited by a change in the compression factor and/or the end of a contiguously stored section. The physical starting address $A_i$ of group $i$ can be stored explicitly in the $i$-th mapping tuple. Then the home address for key $K_j$ is

$$H_j = K_j \mod d + E_j \times d + A_i$$

The second case for discontinuous storage of the table is more general, and in particular would be utilized for Virtual Space Variable Compression. Calculating the physical address of the interval directly, as for a simple table, is not possible due to the discontinuities. Storing the starting address $A_i$ of the group, as above, would require as many additional index entry tuples as there are logical groups which otherwise would cross from one section of the table to another. This could be many times the number of actual discontinuous sections.

The general solution is to maintain a small auxiliary table which serves as a mapping from the table home interval number $h_j$ to the physical address of that home interval of the table. There need be only one entry
(h_i, A_i) per separate contiguous section of the table, with these entries maintained in sorted order by h_i. The pair would represent the first table interval number h_i in this contiguous section and its physical address. Then the home address for key K_j is

\[ H_j = K_j \mod d + (h_j - h_1) \ast d + A_i, \]

where \( h_i \leq h_j < h(i+1) \).

Overflow can be handled by using the modified form of linear probing described above: all locations of the home interval are cycled through, so that overflow from the end of the interval will be placed in the beginning of that interval. Only if an interval is totally filled would overflow continue into the next interval, or perhaps be stored in a separate area for overflow from intervals.

The procedure for Get Next record in key sequence is based upon Algorithm 3, and is applicable for both Variable Direct Compression and Variable Virtual Space Compression. The Level Number l_j and home interval number h_j required there have been defined above.

When using Algorithm 3 for Direct Compression with a variable mapping, all Level Numbers will be zero, so that condition 2 of the algorithm does not apply. Also, wraparound to the beginning of the table would not occur, except for overflow from the last interval of the table. For Virtual Space Compression with variable mapping, Algorithm 3 is used directly to obtain the next record in key sequence, just as it was utilized for regular Virtual Space Compression.

To retrieve a group of several to many records in key sequence from either form of Variable Compression, Algorithm 4 can be applied directly.

Variable compression provides at least two important advantages. First, when partial knowledge is available as to the distribution of actual keys, it can be applied to minimize clustering and overflows, thereby noticeably improving random access and also aiding Get Next retrieval. Second, it allows for discontinuous storage of the hash table, including placement on different levels of the storage hierarchy for access optimization.
VI. MODELLING THE SPECTRUM OF FILE ORGANIZATIONS:
FROM SEQUENTIAL TO RANDOM

Although the file organizations of sequential and random are usually considered as being incompatible with one another, we will show that they can be considered as the end points of a continuous spectrum of file organizations. The model of file organizations which accomplishes this is relatively straightforward, and is based upon the relationships which have been developed for the techniques presented in this paper. We begin by reviewing these relationships.

We started with a basic algorithm for Get Next retrieval of a record from a standard hash table organization. There we viewed the division method of address calculation as a multi-level mapping from the key space to the hash table. The Level Number served to identify the surface or level of the mapping.

This basic approach was extended in several ways in section 5. All the primary extensions involve some form of compression of intervals. If only a single level mapping (L = 1) were used, we had Direct Compression, which offered reasonable performance for retrieval of a group of records in key sequence from a randomly accessible hash table. This single level mapping directly to the table has the property that a set of consecutive key values is mapped into the same interval of the table, and that successive table intervals contain increasing key values. Thus Direct Compression (L = 1) is desirable for sequential-type operations, such as multiple consecutive key retrieval. With a many-leveled mapping (L > 1), we had Virtual Space Compression, which traded off reduced efficiency of group retrieval for better random access.

The other characterizing parameter is the interval size, d. To see its effect, for the moment let L = 1 for a single level mapping. When the interval size d is the same as the table size D, the table consists of a single interval. The whole range of key values is mapped to the entire table by the simple division method of key to address transformation. This obviously is nothing but the original unaltered hash table organization -- for which the basic Get Next algorithm can be applied under the appropriate
conditions.

On the other hand, we can continuously decrease the design parameter \( d \) for the interval size until we reach the other end of the spectrum. At that extreme of \( d = 1 \), there is a one to one correspondence between intervals and table addresses. Since we still are considering a single Level, it can be seen that each of these \( D \) successive table addresses will be the home address for keys of increasing value. We momentarily consider buckets of size one and ignore overflow (or assume that we are variably compressing the Key Space so that overflow does not occur). What we have is a sequential ordering of the keys in the hash table -- that is, we have a sequential file organization, with some free space preallocated in the file.

Thus as we vary the interval size \( d \) from \( D \) to one, we range from random organization to sequential organization. Notice that so far we have kept the number of Levels at one.

Now consider what happens if we vary the number of Levels \( L \) in the table for some value of the interval size \( d \). The result is a hybrid organization. The larger \( L \) is, the smaller the compression factor, and thus the better the random access characteristics will be. The smaller \( L \) is, the better the sequential processing becomes, in that we can obtain more records in key sequence from the same size table interval.

With this categorization, we can see that the interval size \( d \) can serve to describe the spectrum from random organization (\( d \) large) to sequential organization (\( d = 1 \)). The Level Number \( L \) can characterize the tradeoffs between random access to the table (generally better when \( L \) is large) versus key sequence retrieval (better when \( L \) small). In this characterization of file organizations, we can consider the pair \((d, L)\) as a vector whose magnitude indicates the range from sequential access for \((d, L) = (1, 1)\), to random access for \((d, L) \) large.
VI. CONCLUSION

On the surface, there is a seeming incompatibility between hash table organization and key sequence retrieval of entries. However, upon more careful analysis we have been able to develop algorithms which are relatively easy to apply, and which allow retrieval of one to many records in key sequence from a hash table organization. The basic technique is applicable in a selected set of circumstances, which we delimited. In these cases, it can be quite useful and entails no overhead in space nor time for its availability.

We then explored several additional techniques that provided tradeoffs between random access and key sequence retrieval. Virtual Space Compression allows the techniques for retrieval of the next record or the next group of records in key sequence to be applied to a much larger range of file design situations. Compression direct to the table takes this idea of compression to its logical conclusion, and provides the best key sequence retrieval, so long as the occurrence of overflow records does not increase noticeably.

An adaptive technique, which we called Variable Compression, then was developed. It provides the interesting and useful ability to structure and modify the table with respect to partial information about the distribution of keys in the key space. The greater our information about the key distribution, the smaller the number of overflow entries, and the better will be the performance of both key sequence and random access retrieval.

As we approach the limit of Direct Compression, by decreasing the Virtual Space size to the table size, we increase the feasibility of processing large groups of records in key sequence. In this limit, the whole file could be processed occasionally as groups of records in key sequence.

Finally, we cast our techniques into a new framework or model of file organizations. The parameters of this model characterize the spectrum of file organizations and their behavior with respect to sequential and random retrieval.
The techniques developed in this paper provide the designer of files and indices with new alternatives. The designer can choose from a range of organizations that offer rapid random access of hashed tables files, while also providing the ability to retrieve individual records or groups of records in key sequence.

ACKNOWLEDGEMENTS

The author wishes to thank Casimir Kulikowski and Naftaly Minsky for their comments regarding this paper.

REFERENCES


