The principal approaches to random-access file organization and addressing are reviewed in this paper. The review is general, in the sense that it is relatively independent of specific equipment. In the case of a number of unsettled questions, the author's evaluations of alternatives are included.

The relation between sorting and random-access file addressing is clarified by viewing both as belonging to a common class of ordering operations. Basic considerations of both sequential and random-access approaches, arithmetical key-to-address transformation methods with their overflow problems, and table lookup methods are discussed.

Results of an experimental analysis of key transformation techniques are presented.

File organization and addressing

by Werner Buchholz

The first section of this paper is a general discussion of file storage and storage devices. Key transformation is the subject of the next section which is followed by one on the overflow problem. The use of tables is discussed in the remaining section. An experimental analysis of key transformation techniques appears in an appendix. Finally, a fairly comprehensive bibliography is included.

The more extensive treatment of key transformation and overflow as compared to that of tables reflects the results of a recent investigation and not necessarily their relative importance.

Table 1
Part number list

601
602
606
610
6135X3–64 1–4
620
6206SKF
6215X1
6220X1
6225X1
6225X2
6254
630
6305RST
6309RST
631
631C
633
634
635
635C
6350X3
6369D5022
637
638
638C
64X113–17 7–16

Files and their storage

A *file* consists of a group of related records. Each *record*, in turn, is comprised of a number of related data fields and an *identifier* field which distinguishes that record from others in the same file.[1] Sometimes several fields in a given sequence constitute the identifier. The identifier is the means of selecting and retrieving a desired record from a file. On occasion, an identifier field is unnecessary when the position of the record in the file alone may identify it.

Typical identifiers are: names, part numbers, or chronologically assigned serial numbers. Identifiers are frequently chosen according to some classification scheme that is intended to characterize the nature of each item, so that similar items have portions of the identifier in common and will be assigned to a common group of storage locations. Such classification schemes are also intended to

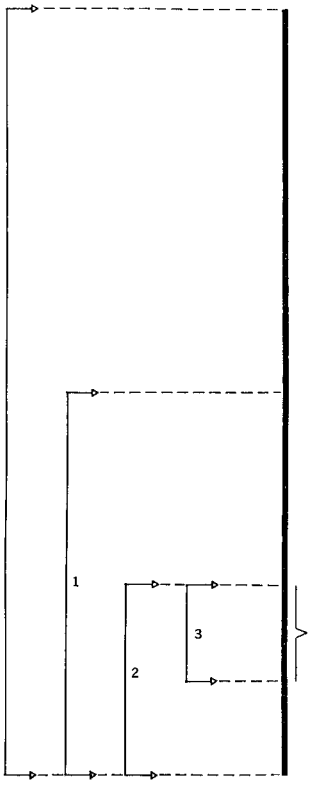
help find unknown items given only some of the common characteristics, as in the familiar example of cataloging library books. But we will not be concerned here with the difficult problems of information retrieval: developing useful classification schemes and assigning unique identifiers in a meaningful way. We will assume that each item already carries its identifier and that when an item is desired its complete identifier is already known.

An item may have primary and secondary identifiers. The primary identifier, or *key*, is chosen to be the one by which an item will be retrieved most often. For inventory parts this is the part number. Secondary identifiers are needed when items are to be collected in a different grouping. Thus inventory parts might be called out by vendor or by assembly in which they are used. The primary identifiers are usually unique. Secondary identifiers are usually non-unique; a search by secondary identifier can be expected to yield more than one answer. A segment of an actual part number list is shown in Table 1 illustrating the variable nature of keys found in some key sets.

scanning and addressing

Retrieval of a record from file storage may be by scanning or by addressing.[2] When the exact location of the desired record in the file storage device is not known, it is necessary to *scan* part or all of the file to search for the record. This involves comparing its key with the key of one record after another until matched. With N records in random order, an average of $(N + 1)/2$ records will have to be scanned. However, except in trivial cases, sequential scanning of a file to find a single record takes much too long.

Figure 1 Binary search



Scanning can be speeded up by first *sorting* all file records to rearrange them into the sequence of their keys. If the location of any one item is known, the direction of search is determined by whether the desired key is higher or lower in sequence. It is often possible to estimate the approximate location of a record from its key and then to find it by a short sequential search from there. The *binary search* technique may be applied: the size of the file area in which to search for the desired record is successively halved by comparing the record sought with a record at the mid-point of the current search area to determine whether the next search should be in the upper or lower half (Figure 1 shows an instance of three such comparisons). Binary searching locates a record within at most $\log_2 N$ tries. Another way of reducing the search time per item is to collect and sort a batch of items for processing in one scanning pass through the file (see batch processing). Since sorting as such is a well-documented procedure, we will not discuss it further.[3,4]

Insertion of new records into a sorted file usually requires the relocation of many existing records to maintain the proper sequence. Thus records kept physically in sequence cannot occupy fixed locations. The alternative approach is to assign each record to a fixed location and to retrieve a record by specifying the address of its location. This requires that each key be associated with an address.[5]

An *address* is usually a number compounded of two or more coordinates that physically select the location. For example, on a disk file various digit groups of an address might specify position on a track, track, disk side, and module (group) of disks. Similarly in a core memory the address might be made up of the x and y coordinates that determine a core in a plane. For reasons of economy an addressable location in a high-capacity file storage unit is made large enough to hold a block of many words, corresponding to one or more records. To select a portion of a block it is necessary to transfer the block to the lower-capacity internal memory of a computer whose finer address structure permits the selection of single words or parts of words. File storage addressing is generally restricted to handling entire blocks of data.

Although the addressing system of a specific file storage device is tailored to the needs of that device and not of the data to be stored there, it is entirely possible to identify each record by its physical storage address. With this simple *direct addressing* method, the key is itself the address. However, the use of direct addressing is limited to applications where the key set may be freely chosen to conform to the restrictions of the available set of addresses (e.g., the storage of relatively small, isolated files or of temporary data such as arrays of numbers for mathematical computations). Otherwise, two problems occur: first, addresses are artificial numbers that are difficult to remember and transcribe correctly; second, most files are subject to occasional or frequent reorganization either to take care of expansion or to maintain a desired sequence. Reorganization of the file means change in addresses. If the addresses are also the record identifiers, this means change external to the machine system: catalogs must be changed, clerks, vendors, and customers notified, and so on. Frequent changes of this kind are not feasible in an application of any size. The cost of just a one-time initial conversion from an existing set of identifiers to one suited for direct addressing is often prohibitive.

Hence keys much longer than addresses are usually chosen to provide for mnemonic symbols or a classification scheme indicative of some physical characteristic. Symbolic keys may survive a file reorganization when addresses cannot. The number of distinct keys possible is much larger than the number actually assigned to file records at any one time. With such a sparsely populated set one can generally find an unused block of keys to assign to a new block of data without unduly violating the rules of classification. Conversely, such keys are unusable as random-access storage addresses. It is clearly uneconomical to provide a separate storage location for each of the possible identifiers made up of, say, 10 alphanumeric characters when the number of different items in the file is only, say, one million and not likely to grow much.

There is no unique and simple way of transforming a long key to a shorter address. One can use a cross-reference table (index), containing every key with the address of its record, and find a desired address by programming a table lookup. Although table

lookup has advantages, the basic problem, which will be discussed later in some detail, is merely transferred from the file to its table. The problem of associating key and address may occasionally be side-stepped by punching both key and address into transaction cards that originated as output of the system, such as punched-card bills being returned with payments. In reality, a cross-reference table has been incorporated in the transaction cards. The system still has to be able to deal with exceptions when transactions arrive without the card or when the address has been changed while the card was in circulation. Of course, the address may be looked up manually in catalogs, index card files, or tub files. But because of the extra clerical labor in an otherwise automatic system and because of the difficulty of keeping manual cross-indexes up to date, it is clearly more desirable to be able to mechanize the conversion from key to address.[6]

Thus there is a choice of looking in the file or in a table and of scanning or computing an address from the key. The four possibilities are actually interrelated. If a separate table is not used, the file serves as its own table, so that any method of access is a form of table lookup. Computing an address does not avoid all scanning because, as we shall see, the address obtained is often not that of the item but of the starting point for a short search.

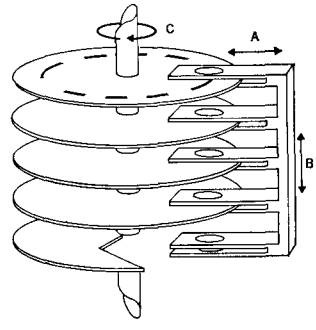
sequential and random storage

By *sequential storage* we mean a one-dimensional medium where the only record that is immediately accessible for reading or writing is the next one in the direction of travel of the medium. The classical example of sequential file storage, and the only important one, is magnetic tape.

Random-access storage devices are all those not restricted to sequential operation. Current examples are magnetic drums, magnetic or optical disks, magnetic sheets or cards, and photographic strips for document storage.

Random-access storage ideally means that access to any location in storage takes the same (presumably short) time regardless of when and where the last location was selected. This ideal is approached only by relatively low-capacity devices such as core memory. In practice, random access means that the longest time for switching between any pair of locations is very much shorter than the time to go sequentially through all intervening locations at normal reading or writing speeds. Random access usually takes significantly more time than access to a location near the last one; with a constantly rotating device the time also depends on where the device happens to be when the access is initiated.

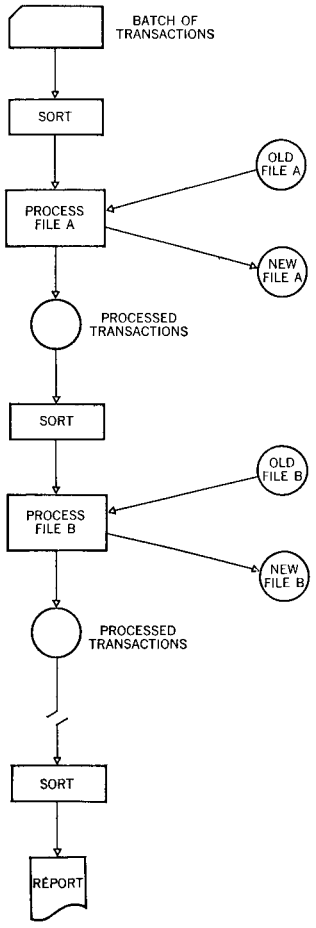
Figure 2 Three-dimensional access in a disk storage unit



Random-access storage devices are designed with modes of access in more than one dimension. One access may be to one of a number of tracks on a given recording surface (start-stop motion *A* in Figure 2), another access may be the selection of one of several surfaces (here electronic switching of recording heads, *B*), and a third may be to a desired point along a track of the physically moving medium (time selection, *C*).

Magnetic tape has one very important characteristic; it is

Figure 3 Batch processing



essentially continuous along its recording dimension. Except at the end of a reel, there are as a rule no physically predetermined starting and stopping points. Consequently there are no predetermined locations for writing data and there are no restrictions on the length of a block of data being written. The end of a block may simply be marked by a gap. The next time the tape is written the gaps may be somewhere else. On reading tape one can only ask for the *next* block. Except for the limited technique of counting gaps as they pass by, there is no way to select a specific record without inspecting each record in sequence.

A second and less obviously fundamental characteristic of tape is that it is not practical to change selectively a single block in the middle of a tape already written. To change anything on a tape, every block must be read, whether changed or not, and rewritten on another tape. This is an engineering limitation that follows from the continuous nature of tape and the great length of tape on a reel, which make *reliable* selective alteration impractical.

Random-access storage necessarily differs on both of these points. If we want to have direct access to a block of data anywhere, the data must have a definite, specifiable location to which we can return precisely. Selective alteration of the contents of any location is permitted, so that writing blocks in random sequence becomes possible.

These properties, so important to random processing, also have disadvantages. Discontinuity generally implies gaps and gaps waste space if the records are short.[7] As soon as a record is altered in place, the old information is gone, whereas with tape one is forced to generate a new tape so that the old information remains as a back-up in case of mishap. Hence restart procedures present special problems with random processing. The time spent with tape in rewriting unchanged records may be used to advantage for a complete review of the entire file. This takes extra time and effort with random storage. Random storage thus removes some restrictions of sequential storage by introducing others.

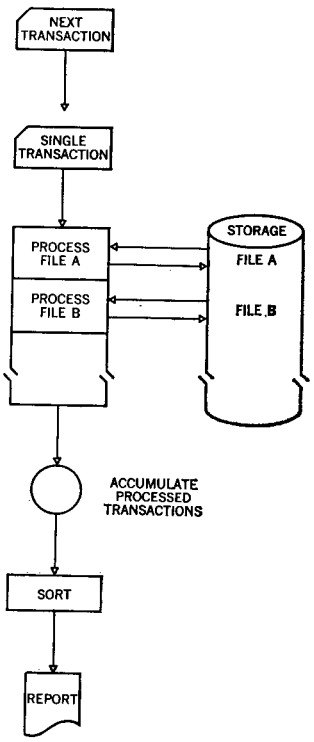
batch and random processing

The quite different characteristics of sequential and random-access storage give rise to quite different modes of processing. With a sequential file storage medium the only efficient method is *batch processing*. A substantial batch of transactions is accumulated before processing can begin. The transactions are then sorted in the sequence in which the records are stored in the file and processed in a single pass through the file. If transactions are to be processed against more than one set of file records, the batch must be sorted again before each pass through a different file (Figure 3).

Random (or *in-line*) *processing*, made possible by random-access storage, permits each transaction to be processed without delay as soon as it arrives. Even if small batches are allowed to accumulate, they may be processed in any order without prior sorting. Smaller batches mean shorter processing cycles and, accordingly, more current information in the file. Input transcription

(keypunching), processing, and output printing may be overlapped as long as the sequence remains the same; so output may begin before input ends. If all pertinent files are contained in the random-access storage, all records affected by a transaction can be updated in a single pass. This cuts down delays and possibly manual intervention, and it simplifies the correlation of data contained in different files (Figure 4).

Figure 4 Random processing



There are reasons for wanting a random-access storage device that also has good sequential properties. For example, it may be advantageous to batch-process transactions even with a random-access file storage device. The transactions are sorted in the sequence in which the corresponding file records are stored, which may be the sequence of the keys (sorted file) or of addresses computed from the keys (key transformation). The access mechanism then needs to move in only one direction for the entire pass, which clearly takes much less time than random back-and-forth motion over the whole range of addresses. This speed advantage is offset to some extent by the time taken to sort the transactions once for each file. Only urgent transactions and exception cases, usually few in number, need to be processed in random sequence. The real advantage of random-access storage is that it *can* be operated economically in random sequence when desired. It does not follow that it should always be used this way.

If we simply stored records in a file in some arbitrary sequence such as the sequence of their arrival, without regard to the retrieval problem, we would be forced to make a long, exhaustive search to retrieve a specific record. To avoid this we must plan ahead and arrange the file in an order that aids retrieval, whether in the sequence of the keys (sorted) or of addresses derived from a key transformation formula. Also, order is especially important in finding that a record is absent or duplicated. Absence of a record is revealed by merely looking at the place where it would have gone, instead of having to search every item in storage. Duplicate items, whose presence may not otherwise have been suspected, are brought together by ordering. Order helps one to find related items or items that almost but not quite fit the request as given, perhaps because of differences in spelling or other errors.

order and the speed of retrieval

It is sometimes suggested that random-access storage files have made ordering—specifically sorting—unnecessary, meaning that input transactions need not be sorted, each item being posted on the master record as it comes. However, this posting is just a type of sorting by distribution, except that the processing may take place on the same pass. Moreover, sorting is still necessary to prepare output records. Thus we may vary the approach to take advantage of new equipment and procedures, but sort we must.

Key transformation

The process of transforming a record key, or external address, to the corresponding internal address, giving the location in the

file where the record should be found, generally consists of two parts.

First, the key is changed to a format best suited to the arithmetical capability of a specific computer. An alphanumeric key may have to be converted to an octal number by splitting each 6-bit character code into two 6-bit codes each containing 3 of the original 6 bits, so that decimal arithmetic can be performed on the result. The length of the key may have to be reduced to fit the registers of a fixed-word-length computer, perhaps by folding and adding one portion of the key to another.

Second, the digits or bits of the (modified) key are reduced to the length and range of numbers which can be used as file addresses. A simple and, as we shall see, good way is to divide the key by a suitable number and add the remainder to the starting address of the file.

No method of transforming keys to addresses is known that completely avoids the problem of overflow caused by too many keys transforming to the same addresses. The management of time-consuming overflow (to be discussed later) must be considered together with the choice of a transformation formula.

key distribution

In the external set all keys are distinct, but the distribution of keys over the entire range is usually far from uniform. Clusters and gaps occur from the way in which keys are assigned and modified over a period of time. As a rule, gaps arise because only a small fraction of the character combinations possible in a key set are ever assigned as actual keys, leaving most of the set empty. When new keys are assigned, they are often entered as a group. For example, the parts belonging to a newly designed assembly may be coded as a block carrying consecutive numbers with common prefixes or suffixes or both. Similarly, deletions may remove groups of keys. Classification schemes, suffixes, or prefixes all create a nonuniform distribution. The most common type of cluster is a short, uniform sequence of keys wherein successive keys differ by 1 in some position, which need not be the low-order (rightmost) one.

The ideal distribution of keys throughout the range of keys is a completely uniform one. Considering decimal keys, for instance, uniform distribution means that the difference between any pair of successive keys, taken in ascending order, is constant. Some digit positions will vary and others may never change. Extracting the varying digits from the key and using them for the address may produce a uniform set of decimal addresses with no duplication. Thus 5000, 5001, 5002, 5003, $\cdots$, 5019 is a uniform set of 20 keys, as is 5000, 5100, 5200, 5300, $\cdots$, 6900. A similar argument applies to non-decimal keys and addresses except that the radix must be changed appropriately. Picking out the wrong two digits in these examples would produce 20 addresses that are all the same. This represents the other extreme of the key transformation problem. The uniformity existing in a key set is, therefore, an important consideration.

Intermediate between these extremes is the case of a random distribution. A random distribution would result if, each time a key must be assigned to a new record, that key is chosen at random from the entire set of as yet unassigned keys; successive assignments would be completely independent of each other. When such keys are converted to addresses, by any of the usual techniques such as extracting selected digits, the resulting address set will tend to have a random distribution over the range of addresses. Unlike the keys, which by the nature of their selection are unique, some of the addresses will occur more than once and some may remain unused. Thus a random distribution is considerably worse than the ideal case considered above but very much better than the other extreme case. There is a distinct advantage of the random key distribution—it is less sensitive to the method used for conversion.

From another point of view, the worst key distribution is a random one. There is no way to transform random keys to addresses with better than random distribution, whereas one can take advantage of any uniformity in a key set to obtain fewer duplicates among the addresses. In practice, purely random key sets and completely uniform ones are both rare. A key set is likely to contain a series of clusters of irregular length and separation. Clusters introduce a degree of uniformity and their irregular length and separation imply a degree of randomness. A well-chosen conversion technique will produce an address set that reflects both elements and has a distribution intermediate between random and uniform.

division

We shall first discuss the division technique assuming that the keys are in a form suitable for arithmetic; later we shall return to the question of modifying keys that are not already in this form.

If the number of available addresses is A and the first address is F, then a key K may be transformed to an address by computing K/A and adding the remainder to F. (All numbers are treated as integers.) This produces A possible consecutive addresses from F to $F + A - 1$.

The radix must be considered. If both keys and addresses are decimal, the division should be decimal; if the computer to be used has no decimal arithmetic, radix conversion is needed before and after division. Alphanumeric (that is, non-decimal) keys may be treated as binary numbers; if the addresses desired are also binary, the division should be binary, and computers without binary arithmetic are then required to do radix conversion. Changing non-decimal keys to decimal addresses (or decimal keys to binary addresses) requires some form of conversion in any case. We shall return to this problem later on.

Besides compressing the addresses to any desired range A, division has another important property. Consecutive keys produce consecutive remainders after division by any integer A, considering zero to follow $A - 1$. As long as there are fewer than A consecutive keys in a run, this means that *all remainders resulting*

Table 2
Division by 17 (prime)

Dividend	*Remainder*
1000	14
1001	15
1002	16
1003	00
1004	01
1005	02
1006	03
1007	04
1008	05
1009	06
3000	08
3100	06
3200	04
3300	02
3400	00
3500	15
3600	13
3700	11
3800	09
3900	07

from such a run are different.[8] The same is true for a run of keys where successive keys are separated by a constant d (larger than 1) provided d and A are relatively prime (i.e., have no common factors greater than 1). Table 2 illustrates this property by showing remainders after dividing some 4-digit numbers by 17.

There is still the problem of two or more runs giving the same addresses. If the starting points of two runs differ by a constant D, their starting addresses will not coincide as long as D has no factor common to divisor A. Thus there will be at most a partial overlap between the addresses obtained from two runs of keys. Since overlap cannot be completely avoided, there will be some irregularly distributed duplicate addresses, but this can be further minimized by permitting each addressable location to hold several records and thus averaging out the irregularities in the now deliberate overlaps from different runs (see later section on *buckets*).

It follows from this reasoning that, normally, a prime number should be chosen for the divisor A. Then d and D can have no factor common to A unless it be A itself. In practice, it would be unlikely for a key set to have systematic separations between keys that are multiples of large prime numbers. Classification schemes, prefixes, and suffixes produce separations that are small multiples of some power of 10 or 2, and hence not multiples of any large prime. A logical choice is a prime number slightly less than the range of available addresses. If 10000 addresses are available, the divisor might be chosen to be the prime 9973. This divisor leaves 27 addresses unused.

It does not follow that a prime divisor is always the best choice for a given set of keys nor that all primes produce equally good results.[9] Primes do, however, avoid serious maldistribution and may be safely used without detailed analysis.

other conversion techniques

Truncating a decimal key and retaining the n lowest digits is equivalent to dividing by 10^n and keeping the remainder. If keys are mostly consecutive integers with only few gaps, truncation may be superior to real division because it is faster and simpler. The address ranges are, of course, limited to 100, 1000, 10000, etc. However, with a more irregular key set the divisor 10^n becomes a very poor choice for reasons already mentioned. Some key sets may be successfully treated by *extracting* a different set of n digits if it is known which digits change and which remain constant; since extraction differs from truncation only in that the digits are first rearranged, similar comments apply. Other arithmetical methods that have been proposed are *shifting-and-adding* techniques or *multiplication* by an arbitrary constant. These operations are usually followed by truncation or division to compress the range, so that one might as well divide by a prime number directly.[10]

A commonly stated objective for address conversion formulas is to *randomize* the key set. The usual meaning of this term is that the conversion technique is intended to disperse the clusters in the original key set and to achieve a nearly random distribution

of addresses over the address range. As we have seen, this is the wrong objective. The ideal key set is a uniform one and dispersing clusters destroys whatever uniformity already exists. To the extent that "randomizing" techniques really randomize, they are making things worse.

One such technique is *radix transformation* followed by truncation.[11] As applied to purely numerical keys, each decimal digit is interpreted as if it were a radix-11 digit which can assume any of the ten values 0 to 9 but never the value 10_{11}. The number so interpreted is converted back to radix 10 if decimal addresses are desired (or to radix 2 for binary addresses) by the usual methods. Finally the excess high-order digits are discarded to form an address of the desired length. A similar procedure may be used for nonnumerical keys by treating them as binary numbers and choosing appropriate radices.

The 6-digit key 400083, for example, would be converted to a 3-digit address by computing $4 \times 11^5 + 8 \times 11^1 + 3 = 644295$ and leaving 295 as the truncated address.

Radix conversion is superior to truncation alone because troublesome runs of keys differing by some power of 10, say 10^k, are converted so that successive keys now differ by 11^k, which is a number that is prime relative to the divisor 10^n implied by truncating to n final digits. The radix conversion technique is primarily aimed at a simple implementation of file addressing with special-purpose equipment. Since $11 = 10 + 1$, the conversion from radix 11 to radix 10 may be reduced to a series of decimal additions and shifts so that multiplication is avoided. Truncation avoids division. For programming on a computer that has adequate division facilities, division by a prime is superior to radix conversion. Division is simpler to program, it is not limited to powers of 10 as address ranges, and it preserves runs of consecutive keys whereas this radix conversion method introduces gaps corresponding to the missing digit value 10_{11}. The gaps tend to disperse addresses more widely and produce appreciably more overflow for key sets containing many such runs.[12]

Another randomizing scheme that is sometimes suggested is to multiply the key by itself (presumably stemming from an erroneous analogy to the center-squaring method of generating random numbers). Squaring keys can produce an excessive number of zeros and has no merit. Other techniques intended to randomize the addresses use coding methods originally developed for error correction[13,14] or statistical work.

folding

On some computers division can be applied directly to numerical keys of virtually any length. Other computers limit the size of the dividend to the length of the arithmetical registers, so that multiple-precision division would have to be programmed for longer keys. In either case, division of longer numbers takes considerably more time.

Truncation may be used to limit the dividend. Only a portion of the key is used, the rest being discarded. A better technique is

Table 3 Folding

748 629	→ 377
758 629	→ 387
758 729	→ 487
759 728	→ 487

to split the key into two or more parts which are then added together; this process is referred to as *folding*. Examples of 6-digit keys folded to form 3-digit numbers are shown in Table 3. Folding has the advantage that variability in any portion of the key is reflected in the result. Specifically a sequence in any position is retained through both the processes of folding and dividing by a prime. It should be noted, however, that folding is not as good as multiple-precision division. The previous discussion of division assumed that all the keys in a key set are unique. Clearly the folded keys cannot be expected to remain unique.

To the extent that folding produces duplicate dividends, the remainders after subsequent division will also be duplicated. It is unlikely, however, for keys that will produce duplication to be related to each other by ordinary classification schemes. Thus folding introduces a little more randomness and correspondingly increases the chance for overflow, but it is not likely to cause a maldistribution. The same cannot be said of truncation.

alphanumeric keys

Many key sets are made up in whole or in part of characters other than decimal digits. Names consist of alphabetic characters, blanks and some punctuation marks. Part numbers frequently contain a mixture of numerical and non-numerical characters. The treatment of such keys depends on their coding and on the nature of the equipment used.

When alphanumeric characters are expressed in a binary code (such as the 6-bit code used in most IBM® computers), the keys may be treated as binary numbers for the purpose of key transformation. In a computer with binary arithmetic, division of such a key (after folding by binary addition, if necessary) by a prime divisor (in binary) is a simple matter. Another simple case is presented by computers (such as the IBM 7070) where two decimal digits are used to encode each alphanumeric character, since division can be carried out in decimal form as for numerical keys.

Many computers employ binary coding for alphanumeric characters but have no facilities for binary arithmetic. For 6-bit character coding, an effective technique is to split each 6-bit group (*byte*) into two smaller bytes of 3 bits and to convert each 3-bit byte to the 6-bit code for one of the digits 0 to 7. Thus code 111 001 (letter I in the IBM code) becomes decimal 71 and 001 001 (digit 9) becomes decimal 11. In effect, each character is replaced by two octal digits. The resultant octal number is treated as if it were a decimal number. Any folding is done by decimal addition, and division is, of course, decimal. Any other conversion of a 6-bit code to a unique pair of decimal digits, such as the previously mentioned 7070 code, would be equally effective.

A commonly used but inferior technique is to convert each 6-bit code to a single decimal digit. This technique makes use of the fact that in the IBM code the alphabetic characters differ from valid numerical digits in only two bit positions, the *zone* bits. All zone bits (11, 10, or 01) are converted to the numerical form (00), usually as a by-product of passing the key through

the adder. Thus 111 001 (I), 101 001 (R), and 011 001 (Z) all convert to 001 001 (9). The trouble with this simple *zone suppression* method is that it is not a unique conversion; it necessarily creates much duplication among the keys, which may far outweigh the duplication introduced by subsequently converting the modified keys to addresses. Zone suppression cannot be recommended when a substantial fraction of the characters in a key are non-numerical.

Because the character codes that may occur in a key set are often considerably fewer than the maximum number of combinations allowed by the code (64 with 6 bits), the key set viewed as a set of binary numbers has gaps introduced by the code itself independently of the key assignment. These gaps tend to break up sequences in the key set and to produce a distribution of remainders close to that of a random set. Hence keys that are known to be purely numerical, even though encoded in 6-bit form, should *not* be converted to the double-digit format (octal or decimal). If a key set is largely numerical with only very few non-numerical characters scattered through it, the zone suppression technique should be considered because the extra address duplication thus introduced may be offset by the fact that decimal sequences are preserved in the single-digit format. But, when in doubt, the safe approach is the double-digit format; division by a prime number will then give a distribution no worse than random.

key transformation summary

In summary, the following points should be kept in mind when selecting a key transformation scheme:

1. Division by a prime number is (in the author's judgment) the best general method known for transforming *unique* numerical keys to addresses of a smaller range.
2. Any information thrown away when preparing keys for division will probably destroy the uniqueness of the key set and thus worsen the distribution.
3. As far as practical, decimal or near-decimal keys should be treated as decimal numbers and keys containing a substantial fraction of non-numerical characters should be treated as binary numbers.

Overflow handling

Because the transformation of long keys to shorter addresses by any of the formulas discussed above will not give a completely uniform distribution, a procedure must be provided for storing elsewhere those *overflow* records whose keys convert to an address that is already fully occupied.[15] An addressable location, or *bucket*, may have a capacity for one or several records. Whenever that capacity is exceeded and overflow occurs, an empty slot must be found for the overflow record. Assuming that the entire storage area is not 100% loaded, there will be other buckets that are not yet filled and the overflow record may be stored in one of them. The alternative is to set aside a separate overflow storage area.

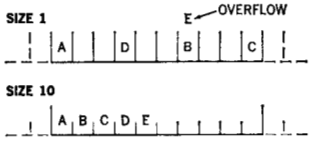
The problem in any case is, first to find an empty slot, and second to find it again later on with minimal searching. (The analogous problem of overflow with a sorted file will be discussed briefly later on.)

Before discussing ways of handling overflow we will discuss how to minimize it further by the use of multiple-record buckets.[16]

buckets

Consider a disk file with 10,000 tracks, each track capable of storing ten records. We may number each record location from 00000 to 99999 and devise a key transformation formula giving 5-digit addresses. The bucket size is one record. Overflow occurs whenever more than one key converts to the same address. There is a probability of 21% that this will happen for a random distribution in a file that is half full. This is illustrated in the upper part of Figure 5 where one of 5 records A to E in a file half full is typically an overflow record.

Figure 5 Multiple-record buckets



Now assume we number only the tracks, from 0000 to 9999, and we rearrange the conversion formula to produce a 4-digit address from each key. The ten record locations in a track are all considered to have the same 4-digit address as the track. Each track is now a bucket of size ten. Overflow will occur only if more than ten keys convert to one address, for which the probability is merely 0.4%, again with the file half full and random distribution, a reduction of 50 : 1. (The advantage goes down as the file fills up, but it is still a significant 3 : 1 with the file almost full.) As illustrated in the lower part of Figure 5, where the same key transformation formula is assumed except for truncation of the low-order digit, all records are stored in the order of arrival starting at one end of the bucket until it is full; in this case overflow is quite unlikely.

The reason for the advantage of a multiple-record bucket arrangement is that adjacent minor overflows and underflows tend to cancel out. When each address can hold only one record, there will usually be one or more empty addresses next to addresses that have overflows. With a bucket size of 10, we group ten such locations together under one address and assume that a given record with that address can be at any available one of the ten locations. Thus, overflow records of the previous example that now fall into one bucket will first occupy the previously empty locations in that bucket, and the whole bucket must contain ten records before it could overflow. This greatly reduces the probability of any record overflowing a bucket address.

We must note, however, that it may be necessary to search the entire bucket to find an empty slot for a new record or to find a record previously stored. Buckets, therefore, provide an advantage in search time only if the access time to successive records within a bucket is much shorter than the access time to the bucket itself. This situation is typical of rotary storage devices such as disks, where access to a track is much longer than the sequential access to records within a track. A disk track, therefore, makes a good bucket, especially when there are facilities for rapidly

scanning the keys of every record on a track (the IBM 1301 disk storage unit offers a limited facility of this type).[17] When the access time to a record is independent of its location, as in core storage, a bucket size greater than one record would ordinarily be a hindrance; a search through several records in a bucket would usually take much longer than a search through one or more randomly located overflow locations. A rotating storage device with multiple, electronically switchable heads presents an intermediate situation where access to one of a group ("cylinder") of tracks takes less time than access to a different group of tracks.

Table 4 gives a good idea of the effect of both bucket size and the degree of file loading on the amount of overflow to be expected. The *initial overflow* tabulated is the number of records that overflow their assigned buckets on initial loading (before the overflow is stored) as a percentage of the actual number of records. It is given as a function of bucket size (maximum number of records per address) and load factor (actual number of records divided by maximum record capacity of storage area) assuming a random (Poisson) distribution of transformed keys.[18] A load factor greater than 1.0 means that the overflow must be stored in a separate area, the main area being too small to hold all the records. The initial overflow figures shown are to be taken only as figures of merit because the actual overflow depends also on the method of

Table 4 Percent initial overflow for random distribution

Bucket Size	*Load Factor* 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2
1	4.84	9.37	13.61	17.58	21.31	24.80	28.08	31.17	34.06	36.79	39.35	41.77
2	0.60	2.19	4.49	7.27	10.36	13.65	17.03	20.43	23.79	27.07	30.24	33.30
3	0.09	0.63	1.80	3.61	5.99	8.82	11.99	15.37	18.87	22.40	25.91	29.33
4	0.02	0.20	0.79	1.96	3.76	6.15	9.05	12.32	15.86	19.54	23.25	26.93
5	0.00	0.07	0.37	1.12	2.48	4.49	7.11	10.26	13.78	17.55	21.42	25.30
6	0.00	0.02	0.18	0.67	1.69	3.38	5.75	8.75	12.24	16.06	20.06	24.11
7	0.00	0.01	0.09	0.41	1.18	2.60	4.74	7.60	11.04	14.90	19.00	23.19
8	0.00	0.00	0.05	0.25	0.84	2.03	3.97	6.68	10.07	13.96	18.15	22.46
9	0.00	0.00	0.02	0.16	0.61	1.61	3.36	5.94	9.27	13.18	17.44	21.86
10	0.00	0.00	0.01	0.10	0.44	1.29	2.88	5.32	8.59	12.51	16.85	21.36
11	0.00	0.00	0.01	0.07	0.33	1.04	2.48	4.80	8.01	11.94	16.34	20.94
12	0.00	0.00	0.00	0.04	0.24	0.85	2.15	4.36	7.51	11.44	15.89	20.58
14	0.00	0.00	0.00	0.02	0.14	0.57	1.65	3.64	6.67	10.60	15.15	19.99
16	0.00	0.00	0.00	0.01	0.08	0.39	1.28	3.09	6.00	9.92	14.56	19.53
18	0.00	0.00	0.00	0.00	0.05	0.28	1.01	2.65	5.45	9.36	14.07	19.16
20	0.00	0.00	0.00	0.00	0.03	0.20	0.81	2.30	4.99	8.88	13.66	18.86
25	0.00	0.00	0.00	0.00	0.01	0.09	0.48	1.65	4.10	7.95	12.87	18.31
30	0.00	0.00	0.00	0.00	0.00	0.04	0.29	1.23	3.47	7.26	12.31	17.93
35	0.00	0.00	0.00	0.00	0.00	0.02	0.18	0.94	2.98	6.73	11.87	17.66
40	0.00	0.00	0.00	0.00	0.00	0.01	0.12	0.73	2.60	6.29	11.53	17.47
50	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.45	2.04	5.63	11.03	17.20
60	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.30	1.65	5.14	10.68	17.03
70	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.20	1.37	4.76	10.41	16.93
80	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.13	1.14	4.46	10.21	16.86
90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.09	0.97	4.20	10.05	16.80
100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.83	3.99	9.92	16.77

insertion, deletion, and overflow storage.[19] The table shows that increased bucket size is very effective in cutting down on overflow compared to the rather expensive method of reducing the load factor by allocating more storage area.

separate overflow storage

Overflow records may be stored at addresses that are not part of the main file and are excluded from the range of addresses permitted by the key conversion formula. How big should the overflow area be? One might calculate the overflow for a random distribution and add a margin of safety, or one might determine the overflow experimentally. If there are several overflow areas, one would provide a common secondary overflow area in case one of the others filled up.

It seems wise to anticipate even the highly unlikely though generally not impossible case of extreme maldistribution where most record keys happen to convert to the same address. To avoid a sudden and completely unexpected collapse of the application, the overflow areas should be monitored continuously. If a predetermined overflow level is exceeded, there would then be time to study and adjust the conversion scheme while allowing the operation to continue.

Since a separate overflow area duplicates the corresponding record locations left vacant in the main storage area, it is desirable to be able to use empty main storage locations for overflow storage whenever storage space is a limiting factor. We will assume no separate overflow area and proceed to the question of where to put and later find an overflow record.

chaining

Chaining may be used to specify a sequence of records other than the sequence of consecutively numbered record locations. Space is left in each record for a *chain address* which gives the address of the next record in the logical sequence, and that record may in turn specify its successor. Thus each record forms a link in a chain, which may be as long as desired. The last link in a chain must be determined by counting or marked by a suitable end-of-chain code. A blank chain address may be this end code, or it may indicate that the successor is the record at the next address. A second chain address in each record has occasionally been used to link it directly back to its predecessor.

A record may take part in several chains, each serving a different purpose and requiring a separate chain address. Of the many applications for chaining, the present discussion is concerned only with the linking of any number of overflow records to their *home location*, whose address is that computed from their keys by the specified key transformation formula.[20,21]

For single-record buckets a new record is stored at its computed home location if that is free; the record is marked with the end-of-chain code, thus forming a single-record chain. If the home location is already full (overflow), a search is made along any existing overflow chain to find its end. The end-of-chain code is removed from that location, the address of a not-yet-filled location is inserted to add another link to the chain, and the record is

stored at the new location which is then marked as the end of the chain. This requires finding an empty space somewhere in storage. As long as the storage area is not too full, a sequential search of locations will soon find a slot. The search becomes very long, however, when the storage area is nearly full. It may be better then to search the entire area once and make a table of all available space. Henceforth this availability table will be consulted whenever a new record is to be stored, either to find an overflow location or to remove a newly filled home location from the table. (An availability table may be too long to be practical if used when the area is not very full.)

To find an already existing record one computes the address of its home location from the incoming key and compares that key with the key of the record stored at that location. If they do not match, one proceeds to the next address in the chain until one of the keys matches (that is, the desired record is found) or the end of the chain is reached (that is, the desired record is not in storage).

When a new record arrives for insertion, it is possible that its home location is already occupied by a record which overflowed from a different home location and may have still other records chained to it. The simplest thing is to let the new record go to the end of the existing chain, thus allowing two independent chains to merge. To save subsequent retrieval time, however, it is better to keep the chains separate by placing the new record in its proper home location, and moving the intruder to another empty location while relinking its chain appropriately. Keeping the chains separate also helps in the deletion of obsolete records.

Multiple-record buckets may also be chained by providing a chain address in each bucket. Whenever a bucket is filled, it is chained to another bucket that is not yet full and additional records are stored there together with home records for that bucket. Eventually that bucket may also fill up and overflows from either bucket are chained to yet another bucket. It is clear that the above-mentioned merging of chains cannot be avoided with multiple-record buckets. Whenever that bucket overflow becomes significant, it is quite likely that no bucket is available for overflow from one chain that does not already contain some records from another chain.

When the equipment requires that records be moved to memory for examination of keys, blocking of individually chained records may be used to reduce accesses to the file. Blocking provides an effect similar to multiple-record buckets, without having to scan the records in a bucket, but at the expense of space for a chain address in each record. Each record is given a separate address to be computed from its key. A record is stored at its own home location when possible. Overflow records are stored within the same block as long as space is available. Once the block is full, overflow must go to some other location. Different records in the same block may be chained to separate blocks. The search time

is reduced since most of the overflow searching will be restricted to one block and thus can be done in memory at high speed. Moreover, the block size for a given file need not remain fixed.

Basically individual record chaining provides direct linkage of overflow records whereas bucket chaining merely provides a path along which to scan for the overflow records. When the file area is nearly full, bucket chaining requires more bucket searches on the average, but most of the time a file area will not be so full that the difference would be significant.

progressive overflow

Chaining requires a search for an empty slot whenever a new record produces an overflow. One way to search is to start at the point where the overflow occurred and continue through consecutive addresses. As noted before, an empty slot will probably soon be found as long as the storage area is not too full. By making the same short search every time that a record is referred to, one can dispense with the chain address.[2,22,23] We shall call this simple technique *progressive overflow* rather than "open addressing," a less descriptive term found in the literature

With progressive overflow a search for an already stored record starts at its home location and proceeds through consecutive addresses until it is found or until an empty position occurs (record absent). Normally these repeated searches do not consume much time, especially with a rotating storage device designed to permit continuous scanning from track to track with little or no lost time for track switching. The scheme also avoids the space and the bookkeeping required for chain addresses. Only when the number of records stored approaches the capacity of the storage area will the search time rapidly increase. The last record requires on the average the searching of half the records stored. Even the average search time for records chosen at random goes up rather sharply toward the end. Thus chaining may take significantly less time when the storage area is nearly full.

Progressive overflow is really a form of chaining where the links of the chain are made up of consecutive addresses. The end of a chain is marked by the first empty space encountered; a search for a non-existent record may be stopped at the first empty space because that is where it would be stored if it were entered as the next new record.

Progressive overflow tends to produce local overflow clusters because the overflow propagates from one location to the next. To put it another way, as soon as one location overflows, the probability is doubled that a new randomly chosen address will fall into the next location and perhaps cause it also to overflow. Hence the average search time is somewhat longer than with a purely random distribution. The same thing could happen with chaining unless some care is taken to pick out empty slots at random.

frequency loading

If the file storage area is full enough so that there is an appreciable amount of overflow, access time may be improved considerably by giving preference to records that are referred to most

frequently. If the frequency statistics, such as the number of references per month, are known for each record, the records may be sorted in the order of decreasing frequency; otherwise the records may be classified merely as relatively active or inactive according to general experience. During initial loading, or any subsequent reloading, the most active records are loaded first and the least active ones last.

Thus the most active records are most likely to go to their home locations and require only one access time. Most of the extra accesses to overflow locations will be incurred by the less active records arriving later.

two-pass loading

With chaining (but not with progressive overflow) a speed advantage may be gained by loading first only those records that can go into home locations. Records that would overflow are retained for a second pass so that all home locations are occupied before overflows are distributed. If a count of records in each bucket is kept during the first pass, further overflows during the second pass can be minimized by a procedure of chaining each overflowing bucket, as far as possible, to a complementary "underflowing" bucket, that is, a bucket which has space left for exactly the number of overflow records from the other bucket.[24] Not all buckets can be thus paired off because overflows and underflows are not symmetrical, so that some secondary overflow may still occur. This procedure may be expected to be most effective when the file storage area is nearly full.

deletions

So far we have discussed initial loading of a file and insertion of new records. Deletion of an obsolete record is no problem when it is an isolated record stored at its home location without overflow chaining. With individual record chaining, deletion of a record in a chain requires relinking of the chain to remove this address from the chain. The now vacant space is marked as available.

With bucket chaining, however, the possible merging of chains makes their relinking rather difficult. Instead it is simpler to tag a record as obsolete but not remove it until the next periodic file dump. The space and time possibly wasted by leaving obsolete records in the file is often trivial, and it may be a more desirable procedure to keep them in the file for a definite period of time. The greater difficulty of deleting records from chained buckets is, therefore, not necessarily a handicap.

duplicate keys

Primary key sets, as already noted, should not contain any duplicates. It is still possible, however, to have two different records with the same keys in the same file through some procedural or other error. Different items may inadvertently have been coded the same, or a master record for the same item may accidentally be entered twice into the system at different times. Unless precautions are taken to detect the remote possibility of duplicate addresses, one might happen to read and modify one record but write it on top of the other record.

Duplication may also occur when keys are abbreviated. The built-in scanning procedure for single-record operation of the IBM

1301 disk files, to save comparison circuits and space, uses 6-character "record addresses" instead of the full record keys. The record address is assigned to each record by programming. A record is selected for reading or writing by giving the bucket address (track number) and the 6-character record address. The intent is to select 6 characters out of every record key for the record address in such a way that duplication on any one track is extremely unlikely. Again, so long as duplication is not known to be impossible, its occurrence, though improbable, should be anticipated. This may be done by checking the entire track for duplicate record addresses while loading or inserting new records. Should a duplicate occur, it is placed on an overflow track even though there is space available on the home track. Overflow tracks must likewise be checked, of course.

Errors such as the accidental assignment of duplicate keys can be reduced by proper controls but they can never be completely prevented. Occasional errors could well be lumped together with record losses through catastrophic failures, for which back-up procedures are needed in any case. It might be reasonable then to conclude that these measures will also be sufficient to take care of the above problems without checking each new record separately, but this conclusion should be reached only after considering, not ignoring, the problem.

Tables

A desired record may be located in the random-access file either by searching among the records in the file or by looking up the address in a table. Such tables are really abbreviated files; each table record contains an identifier (key) and the corresponding address in the main file. The problem of looking up an entry in a table is basically the same as finding a record in a file.

For both tables and files a fundamental distinction is whether or not the entries are sorted in the order of the keys. Thus we may have the following six cases:

	1	2	3	4	5	6
Table	Sorted	Sorted	Unsorted	Unsorted	None	None
File	Sorted	Unsorted	Sorted	Unsorted	Sorted	Unsorted

There are also intermediate cases of partial sorting. "Unsorted file" here means: not sorted according to the key set of current interest; the file may actually be sorted according to a different key set.

sorted files

Keeping the records in a file sorted has the advantages, already referred to, of permitting the average access time to be reduced by accumulating and sorting a batch of transactions. Periodic processing of the entire file in sequence becomes practical without having to sort the file every time, which would often be prohibitive. Duplicate and absent records are easily found in a sorted file.

The obvious difficulty is that a new record can seldom be in-

serted into its proper sequential location without moving many records lying between that location and the nearest available one. Since some space will have to be left empty for future expansion, this space may be utilized as *overflow* areas to hold new records until the next reorganization of the file for the purpose of audit, review, and dumping for backup storage; at that time the new records are sorted and merged with the records in the main file while obsolete records are deleted. The inserted records in the overflow area may be chained to the main file records in their logical sequence, in a manner similar to the previously discussed overflow chaining for key transformation.[21] The insertion problem may be avoided, however, in applications where the setting up of new records in the file can be deferred until a subsequent reorganization period.

If the distribution of keys is sufficiently uniform, the location of a record in a sorted file can be predicted fairly accurately from one or more of the high-order characters of its key. Usually, however, large sorted files require lookup of auxiliary tables to reduce the search time.

table lookup

Tables are themselves files with short, fixed-length records that are less subject to change than the file records and hence more easily managed. They permit constraints on the main file organization to be removed: main file records may vary in length and be stored in any desired sequence without adversely affecting the access time or storage efficiency. The price for this flexibility is the overhead cost of storage space, access time, and maintenance of the table. Sorted files permit the use of partial tables, but otherwise the cost of the storage space for complete tables is substantial.

With sorted tables some form of scanning is used to find an entry or to determine its absence. Scanning may be speeded up by the binary search technique or by any other method of estimating the approximate location of the key to shorten the scan. Alternatively a hierarchy of tables may be used (see below).

The key transformation techniques previously discussed for direct access to files may also be employed to gain access to a complete table, thus reducing the problem of table maintenance as compared with sorted tables. The table entries must be arranged accordingly. The short item length of tables permits large bucket sizes, and the relatively small size of tables makes it practical to leave a fair amount of extra space, both factors combining to reduce the inherent address overflow to a negligible part of the overall access time.

multiple tables

Table lookup may or may not be used to find a record by its primary key, but tables are essential for efficient access to a file according to an identifier other than the primary key. The basic techniques are no different for these secondary tables.[25]

The addresses in the secondary tables may refer to the file or to the appropriate entry in the primary table, if any. Reference via the primary table is desirable if the file is reorganized frequently so that changes in the address of a record need not be

reflected in all the tables. Insertion of a record still requires modification of all the tables.

It is important to note that primary identifiers are normally chosen to be unique, but secondary identifiers may not be. Barring an error, one might expect no duplication in the primary table. All transactions with the same primary identifier presumably belong to the same record. It would be coincidental, however, if another independent method of classifying the same items would be entirely unique. In an employee file, for instance, all employee numbers should be unique, but names and education are not. Hence a lookup by secondary identifier must allow for multiple matches.

Tables may be useful for output processing as well as for input. Thus, instead of sorting the file in the order desired for output reports or summaries, it is possible to maintain a table in the output sequence. This is advantageous if the output uses only a small extract from the file and there are few changes in its make-up.

table hierarchy

When a storage device possesses a natural hierarchy of storage buckets, with two or more levels of increasing capacity and access time, multi-level tables are appropriate to find a record in a sorted file. Scanning starts with the top table in the hierarchy to find the proper table at the next lower level; this scanning proceeds downward from level to level until the file bucket is reached that contains the desired record.

Consider a two-level disk file, such as the IBM 1301, consisting of a number of major buckets (cylinders) each containing a number of minor buckets (tracks). The file records are first sorted in the ascending sequence of their keys. They are then stored in that sequence. Each time a track is filled, the key of the last record and the track address are entered in the track table for that cylinder. The track table itself may be the first track of the cylinder. When a cylinder is filled, the key of the highest record just stored, which is also the last entry of the track table, is entered in the common cylinder table and a new cylinder is begun. The cylinder table remains in core memory.

To find or to alter a single record previously stored, the cylinder table in memory is scanned first. The desired key may be compared successively with each table entry until the comparison changes from *high* to *low* or *equal*. The track address at that point is the address in the disk unit of the track table to be searched next in a similar manner. Again, the track table entry where the comparison changes gives the track where the record is stored. This track is finally scanned for an *equal* comparison, which locates the record if it exists. The track table search increases the basic random-access time by the relatively short time to read one extra track in the same cylinder.

For a large file the cylinder table may be too large to fit into available core memory and may take up several tracks if stored on the disk unit. This adds another level to the search, starting with a master table in core memory to index the cylinder table tracks.

In applications where it is not possible to defer record insertions and deletions until the next periodic file dumping and reloading time, an overflow procedure may be devised that preserves the basic table scanning method of retrieval. In one method, the new record is inserted at the proper point in the sequence and all subsequent records on that track are moved. The last record is moved to an overflow track. The track table entry is updated to show the new highest record on this track and an overflow entry is inserted behind it in the table.

Conclusion

Random-access file storage adds a new degree of freedom over sequential storage on tape, but sequential accessibility may still be an important measure of performance of a random-access storage device. High sequential speed improves file loading and dumping, periodic file review, and the scanning involved in many file addressing methods.

When records are always referred to at random, key transformation using division by a prime number is an efficient method of gaining access to files. The storage space and maintenance of cross-reference tables are avoided, and new records may easily be inserted in the file. Since the record sequence in such a file is not related to any natural processing sequence, routine record processing cannot take advantage of sequential access which is usually much faster than random.

Sorted files with table lookup should be considered in applications where random processing can be the exception rather than the rule and where record insertions are not too frequent. They greatly speed up sequential processing that may be needed periodically. For primary access to sorted files, scanning of partial tables reduces the cost of table storage and maintenance. When complete tables are needed, such as for reference by secondary identifiers, key transformation for addressing the tables is again an efficient technique.

The extra freedom of random-access storage should not be abused. Its real advantage over tape in most file processing applications is that the unusual and urgent transactions can be processed out of turn with little penalty in time and cost if the exceptions are few in number.

Appendix: Some results of an experimental address conversion analysis

Table A1 shows some selected results of an analysis, run on an IBM 7090, of different conversion techniques applied to artifically generated sets of numerical and alphabetic keys. The *initial overflow*, found by counting duplicate addresses generated from each key set by the methods indicated, is used as a figure of merit. All figures are for a bucket size of 10 records and an average of 10 records per address (load factor of 1.0). They correspond to a

Poisson overflow of 12.5% reproduced here from Table 4 for comparison. Finally, for comparison with the results from artificial key sets, two figures are shown for numerical keys taken from an actual insurance policy file. It should be remembered that all overflow figures will be reduced significantly if the loading is less than 100% (see Table 4).

These figures were selected as representative of the trends observed; they vary considerably as parameters are changed and should not be used in quantitative estimates. Some of these trends are:

1. Numerical (binary-coded decimal) keys should be divided decimally (or be converted to true binary numbers). Dividing such keys in binary (without radix conversion) produces more overflow.
2. Prime numbers such as 3001 (of the form $kR^n \pm 1$ for small k) *may* give trouble. The case quoted was extreme among those observed; but since it can happen, such divisors should be avoided.
3. Radix conversion for "randomizing" purposes generally produces more overflow than division by a prime, except when the distribution is close to random to begin with.
4. Truncation of long keys by discarding excess digits can lead to trouble as shown here. Folding by adding one set of digits to another part of the key is much less likely to do so.
5. Zone suppression for largely consecutive alphabetic keys significantly increases the overflow.
6. For alphabetic keys it makes little difference whether division is binary or decimal, or whether the code is binary (as in the IBM 1401, 7080, 7090) or decimal (as in the IBM 7070).

Table A1 Experimental results of address conversion analysis

	Percent Initial Overflow
Consecutive Numerical Keys	
Decimal division by 997 (prime)	0.3
Binary division by 997	6.8
Radix conversion using radix 11	6.4
Consecutive Alphabetic Keys	
Binary division by 997	8.5
Decimal division with zone suppression	25.5
Irregular Numerical Keys	
Decimal division by 997	10.0
Decimal division by 3001 (prime)	20.3
Folding and decimal division by 997	10.9
Truncation and decimal division by 997	24.9
Irregular Alphabetic Keys	
Binary division by 997	11.9
Decimal division by 997 with zone suppression	15.0
Decimal (7070) code, decimal division by 997	13.4
Random (Poisson) Distribution, for Comparison	12.5
100,000 Insurance Policy Numbers	
Decimal division by 9973 (prime)	6.3
Radix conversion using radix 11	7.7

ACKNOWLEDGMENT

Credit for techniques reviewed here belongs to authors cited below as well as the many authors of unpublished or anonymous papers and manuals.

CITED REFERENCES AND FOOTNOTES

1. W. Buchholz (ed.), "Planning a Computer System," McGraw-Hill Book Co., New York, 1962, p. 39.
2. K. E. Iverson, "A Programming Language," John Wiley & Sons, New York, 1962, chapter 4.
3. *Iverson*, op. cit., chapter 6.
4. C. C. Gotlieb, "Sorting on Computers," Communs. ACM, vol. 6, no. 5, pp. 194–201, May, 1963. See also in that issue several other papers on sorting and a bibliography (p. 280).
5. An intermediate approach is to keep the *physical* location of existing records fixed but indicate the *logical* sequence of records by specifying in each record the address of the next record in sequence when it is other than the physically adjacent one. Such a file remains logically but not physically sorted after the insertion of new records. (See also Reference 21.) This method trades speed of random retrieval for ease of record insertion.
6. Using the key and a conversion formula for retrieval, when the address of a desired record cannot be uniquely derived from the key, has been called *indirect addressing*. Unique conversion formulas that change n keys to n different addresses such as multiplying a serial number by a constant and adding this to a base address, are included in direct addressing.
7. Even so the inter-record gaps required on a continuously moving medium such as magnetic disks may be much smaller than the mechanical start-stop gaps required on magnetic tape. Thus the IBM 1301 disk unit uses gaps of 38 characters (including addresses) as compared to the IBM 729 tape gaps of about 600 characters (at 800 bits per inch density). Gaps may not be apparent at all if the density is low enough to permit addressing and switching at any bit position, as on word-addressed drums.
8. W. P. Heising, "Note on random addressing techniques," this issue.
9. Prime divisors of the form $kR^n \pm 1$ should be avoided for keys that are numbers in radix R, where k is a small integer. As may be seen from the binomial expansion of $(R^n \pm 1)^{-1}$, the remainder after division is essentially a superposition of successive n-digit groups of the dividend, and this systematic superposition tendency is retained for small $k > 1$. Hence for decimal keys and in the range of 10^2 to 10^5, such primes as 101, 199, 401, 499, 599, 601, 701, 1999, 2999, 3001, 4001, 4999, 7001, 8999, 9001, 49 999, 59 999, 70 001, 79 999, 90 001 had better be avoided.
10. Multiplication by the reciprocal of a prime number (corrected for round-off error) may be considered as an alternative to division on computers not equipped with fast division.
11. A. D. Lin, "Key Addressing of Random Access Memories by Radix Transformation," AFIPS Conf. Proceedings Vol. 23, 1963 Spring Joint Comp. Conf., pp. 355–366.
12. The argument is due to W. P. Heising. Since there have been attempts to combine radix conversion with division rather than truncation, a word of warning may be in order. If the divisor happens to be divisible by 11 (or whatever radix is used to interpret the key), a very bad distribution of addresses may result. Even if the divisor is a prime, the additional radix conversion step generally worsens rather than improves the distribution of addresses.
13. G. Schay and N. Raver, "A Method for Key-to-Address Transformation," IBM Journal Res. and Dev., vol. 7, no. 2, pp. 121–126, April, 1963.
14. M. Hanan and F. P. Palermo, "An Application of Coding Theory to a File Addressing Problem," IBM Journal Res. and Dev., vol. 7, no. 2, pp. 127–129, April, 1963.
15. Records whose keys convert to the same address are often called *synonyms*.
16. The earliest papers known to the author, where the overflow problem resulting from key transformation including the concept of multiple-

record buckets are described, are unpublished reports by H. P. Luhn and A. D. Lin dated January–March 1953.

17. Without rapid key scanning facilities the whole bucket may be transferred to memory and scanned there. In that case a better choice of bucket size may be half a track, because this may leave enough time after reading the bucket for scanning and processing the record before rewriting the bucket during the next revolution. If memory space for several records is not available, a table of the keys in a bucket may be placed at the beginning of the bucket to be scanned in memory instead of the records themselves. This organization has been called an "indexed file," a term which is too easily confused with the more general use of cross-reference tables. Ref.: "Loading and Maintaining an Indexed File for the IBM 305," RAMAC 305 Bulletin, IBM Form J28-2042, July, 1959.
18. The initial overflow v shown in Table 4 for bucket size b and load factor f was computed by the following iterative procedure:

    Let $\mu = fb$, $T_0 = -1$, $V_0 = \mu$, $P_0 = e^{-\mu}$.

    $$\left.\begin{aligned} T_k &= T_{k-1} + P_{k-1} \\ V_k &= V_{k-1} + T_k \\ P_k &= (\mu/k)\, P_{k-1} \end{aligned}\right\} \text{for } k = 1, 2, \cdots, b$$

    Then $v = V_b/\mu$.

    (The function $P_k = \mu^k\, e^{-\mu}/k!$ is the Poisson probability, which gives the fraction of the addresses to which exactly k keys will be transformed for a random distribution with an average of μ keys per address.)
19. Another figure of merit often used is the *seek factor, or the average number* of reference cycles needed to retrieve any record. Overflow and seek factor are obviously related although not in any simple manner, and neither can be directly translated into actual performance figures. Clearly the smaller the overflow, the more closely the seek factor approaches 1.0.
20. "Disk File Organization Routines for the IBM Ramac 1401," IBM Form J24-1451, 1961. This manual refers to the IBM 1405 Disk Storage Unit.
21. "Disk File Organization Routines for IBM 1401/1311," IBM Form C24-1483, 1963. Ditto for IBM 1440/1311, IBM Form C24-3003.
22. W. W. Peterson, "Addressing for Random-Access Storage," IBM Journal Res. and Dev., vol. 1, no. 2, pp. 130–146, April, 1957.
23. G. Schay, Jr., and W. G. Spruth, "Analysis of a File Addressing Method," Communs. ACM, vol. 5, no. 8, pp. 459–462, August, 1962. Although it is claimed (p. 460) that a modification of the progressive overflow technique reduces the average number of accesses, any rearrangement of records should leave the average random-access time the same.
24. H. S. Samuels and T. L. Tarson, forthcoming issue.
25. L. R. Johnson has proposed an interesting technique for combining both primary and secondary tables with the file record. Key transformation and a chain address for overflow are used in lieu of the primary table. Two more addresses are used for each secondary table. The first of these gives the home location for the secondary reference (since it would only be by coincidence that the transformation for both the primary and the secondary identifiers of the same record would yield the same address, so that the start of the secondary chain would have to be somewhere else). The second of these addresses is the secondary chain address. Considering the awkwardness of maintaining such a file, it is not clear that telescoping several tables and the file presents any practical advantages. Ref.: L. R. Johnson, "An Indirect Chaining Method for Addressing on Secondary Keys," Communs. ACM, vol. 4, no. 5, May, 1961, pp. 218–222.

BIBLIOGRAPHY

References of greatest interest have already been cited and are not repeated in the following chronological list.

- G. Eisler, "Requirements for a Rapid Access Data File," Proc. 1956 Western Joint Comp. Conf., pp. 39–42.
- M. L. Lesser and J. W. Haanstra, "The RAMAC Data-Processing Machine: System Organization of the IBM 305," Proc. 1956 Eastern Joint Comp. Conf., pp. 139–146.
- A. I. Dumey, "Indexing for Rapid Access Memory Systems," Computers and Automation, vol. 5, no. 12, pp. 6–9, December, 1956.

- J. A. Postley, "Contrasts in Large File Memories for Large-Scale Computers," Proc. 1958 Western Joint Comp. Conf., pp. 193–194.
- W. P. Heising, "Methods of File Organization for Efficient Use of IBM RAMAC Files," Proc. 1958 Western Joint Comp. Conf., pp. 194–196.
- "RAMAC 305 Programmer's Guide," General Information Manual, IBM Form F26-2018, 1958, pp. 7–21.
- R. de la Briandais, "File Searching Using Variable Length Keys," Proc. 1959 Western Joint Comp. Conf., pp. 295–298.
- "Loading and Maintaining a Chained File for the RAMAC 305," RAMAC 305 Bulletin, IBM Form J28-2041, May, 1959. Also Addenda to this bulletin, IBM Form J28-2043, July, 1959.
- J. Jeenel, "Programming for Digital Computers," McGraw-Hill Book Co., New York, 1959, p. 280.
- E. Fredkin, "Trie Memory," Communs. ACM, vol. 3, no. 9, pp. 490–499, September, 1960.
- D. E. Ferguson, "Fibonaccian Search," Communs. ACM, vol. 3, no. 12, p. 648, December, 1960.
- "RAMAC 650 Programs: Utility-Scheduling-Chaining," Reference Manual, IBM Form C28-4046, 1960.
- "RAMAC 305," Reference Manual, IBM Form A26-3502–4, 1961, pp. 49–62.
- M. C. Yovits (editor), "Large-Capacity Memory Techniques for Computing Systems," Proceedings of ONR Symposium, May 1961. The Macmillan Co., New York, 1962.
- "IBM 1301 Disk Storage with IBM 7000 Series Data Processing System," General Information Manual, IBM Form D22-6576, 1962.
- "IBM 1410 Data Processing System—IBM 1301 Disk Storage," Reference Manual, IBM Form A22-6670, March, 1962.
- H. Hellerman, "Addressing Multidimensional Arrays," Communs. ACM, vol. 5, no. 4, pp. 105–207, April, 1962.
- "A General Approach to Automatic Programmed Address Conversion," 1400 Series Systems Bulletin, IBM Form J20-0235, 1962.
- M. D. McIlroy, "A Variant Method of File Searching," Communs. ACM, vol. 6, no. 3, p. 101, March, 1963.
- D. C. Johnson, "Dynamic Random Computer Processing," Data Proc. for Mgmt., vol. 5, no. 3, p. 11, March, 1963.
- Price Waterhouse & Co., "In-line Electronic Accounting, Internal Control and Audit Trail," General Information Manual, IBM Form F20-2019, undated.