21 HASHING

Hashing (Fig. 19.5) has been used for addressing random-access storages since they first came into existence in the mid-1950s, but nobody had the temerity to use the word *hashing* until 1968. The word *randomizing* was used until it was pointed out that not only did the key conversion process fail to produce a *random* number, but, contrary to early belief, it was undesirable that it should.

Many systems analysts have avoided the use of hashing in the suspicion that it is complicated. In fact it is simple to use and has two important advantages over indexing. First, it finds most records with only one seek, and, second, insertions and deletions can be handled without added complexity. Indexing, however, can be used with a file which is sequential by prime key, and this is an overriding advantage for some batch-processing applications.

There are many variations in the techniques available for hashing. They have been compared in many different studies [1,2,3,], and from these studies we will draw certain guidelines about which are good techniques and which are best avoided.

FACTORS AFFECTING EFFICIENCY

The factors which the systems analyst can vary when using hash addressing are as follows:

1. The bucket size.
2. The packing density, i.e., the number of buckets for a file of a given size.
3. The hashing key-to-address transform.
4. The method of handling overflows.

Optimal decisions concerning these factors have a substantial effect on the efficiency of the file organization. We will review them in the above sequence.

BUCKET SIZE

A certain number of address spaces are made available, called *home buckets*. A bucket can hold one or more records, and the systems analyst can select the bucket capacity. As shown in Fig. 19.5, the hashing routine scatters records into the home buckets somewhat like a roulette wheel tossing balls into its compartments.

Let us suppose that a roulette wheel has 100 balls which it will distribute to its compartments. Each ball represents a record, and each compartment represents a bucket. The wheel's compartments can hold 100 balls in total; however, we can vary the size of the compartments. If a ball is sent to a compartment which is full, it must be removed from the roulette wheel and placed in an overflow area.

If we have 100 compartments which can hold only one ball each, the wheel will often send a ball to a compartment which is already full. There will be a high proportion of overflows. If we have 10 compartments which

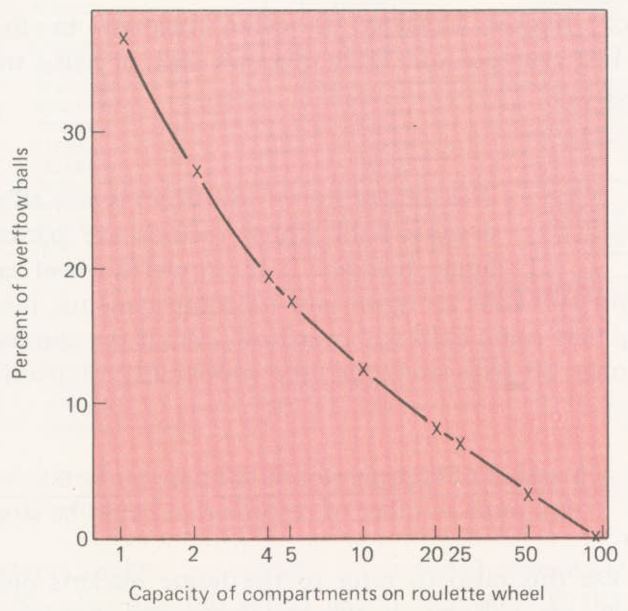


Figure 21.1

can hold 10 balls each, there will be far fewer overflows. It is an exercise in basic statistics to calculate the expected number overflows, and Fig. 21.1 shows the result.

If a systems analyst chooses a small bucket size, he will have a relatively high proportion of overflows, which will necessitate additional bucket reads and possibly additional seeks. A larger bucket capacity will incur fewer overflows, but the larger bucket will have to be read into main memory and searched for the required record.

Figures 21.2 and 21.3 illustrate a simple hashing process with a bucket capacity of 2.

If a direct-access device is used which has a long access time, it is desirable to minimize the numbers of overflows. A bucket capacity of 10 or more should be used to achieve this end.

On the other hand, if the file is stored in a solid-state, or core, storage, the overflow accesses can be carried out as rapidly as the read operations used when searching a bucket. In this case it is desirable to minimize the bucket-searching operation at the expense of more overflows. In such a case a bucket size of 1 is economical. Later we will discuss systems using *paging* in which a page containing many items is read into solid-state storage and hashing is used for finding an item on the page as quickly as possible. A bucket size of 1 is used.

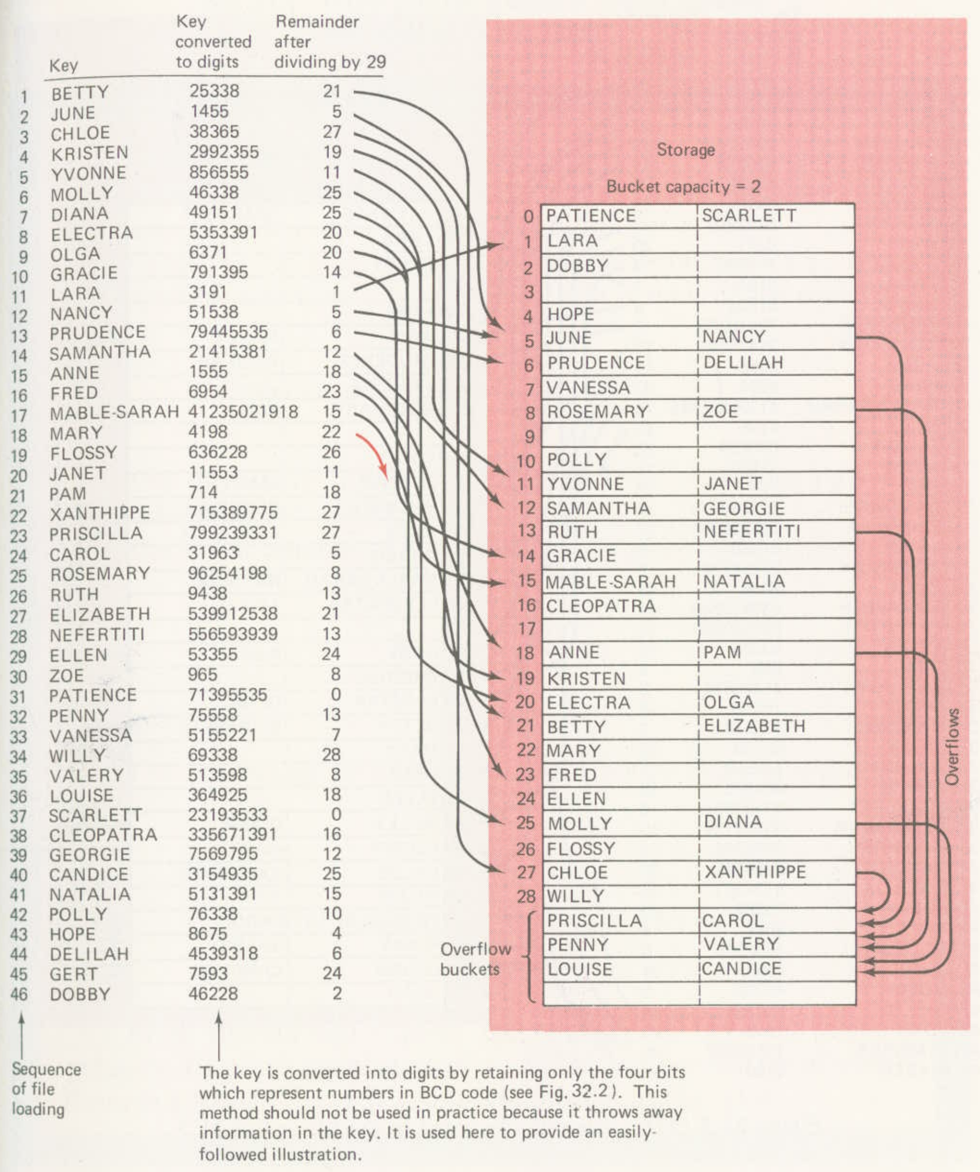
In practice, bucket capacity is often tailored to the hardware characteristics. For example, one track of a disk unit, or half a track, may be made one bucket.

PACKING DENSITY

The proportion of overflows is also affected by the density with which records are packed into the home buckets. If the roulette wheel can hold 100 balls in total and 100 balls are spun into its compartments, there will be a high probability that some balls will overflow—even if the compartments are quite large. If only 80 balls are spun, the probability of overflow will be much lower.

$$\text{Packing density} = \frac{\text{Number of records stored in home buckets}}{\text{Maximum number of records that could be stored in them}}$$

When we use this ratio to refer to the home buckets only, ignoring overflow records, we will call it the *prime packing density*. The above roulette wheel spinning 80 balls is used with a prime packing density of 80%.



	Key	Key converted to digits	Remainder after dividing by 29
1	BETTY	25338	21
2	JUNE	1455	5
3	CHLOE	38365	27
4	KRISTEN	2992355	19
5	YVONNE	856555	11
6	MOLLY	46338	25
7	DIANA	49151	25
8	ELECTRA	5353391	20
9	OLGA	6371	20
10	GRACIE	791395	14
11	LARA	3191	1
12	NANCY	51538	5
13	PRUDENCE	79445535	6
14	SAMANTHA	21415381	12
15	ANNE	1555	18
16	FRED	6954	23
17	MABLE-SARAH	41235021918	15
18	MARY	4198	22
19	FLOSSY	636228	26
20	JANET	11553	11
21	PAM	714	18
22	XANTHIPPE	715389775	27
23	PRISCILLA	799239331	27
24	CAROL	31963	5
25	ROSEMARY	96254198	8
26	RUTH	9438	13
27	ELIZABETH	539912538	21
28	NEFERTITI	556593939	13
29	ELLEN	53355	24
30	ZOE	965	8
31	PATIENCE	71395535	0
32	PENNY	75558	13
33	VANESSA	5155221	7
34	WILLY	69338	28
35	VALERY	513598	8
36	LOUISE	364925	18
37	SCARLETT	23193533	0
38	CLEOPATRA	335671391	16
39	GEORGIE	7569795	12
40	CANDICE	3154935	25
41	NATALIA	5131391	15
42	POLLY	76338	10
43	HOPE	8675	4
44	DELILAH	4539318	6
45	GERT	7593	24
46	DOBBY	46228	2

Storage

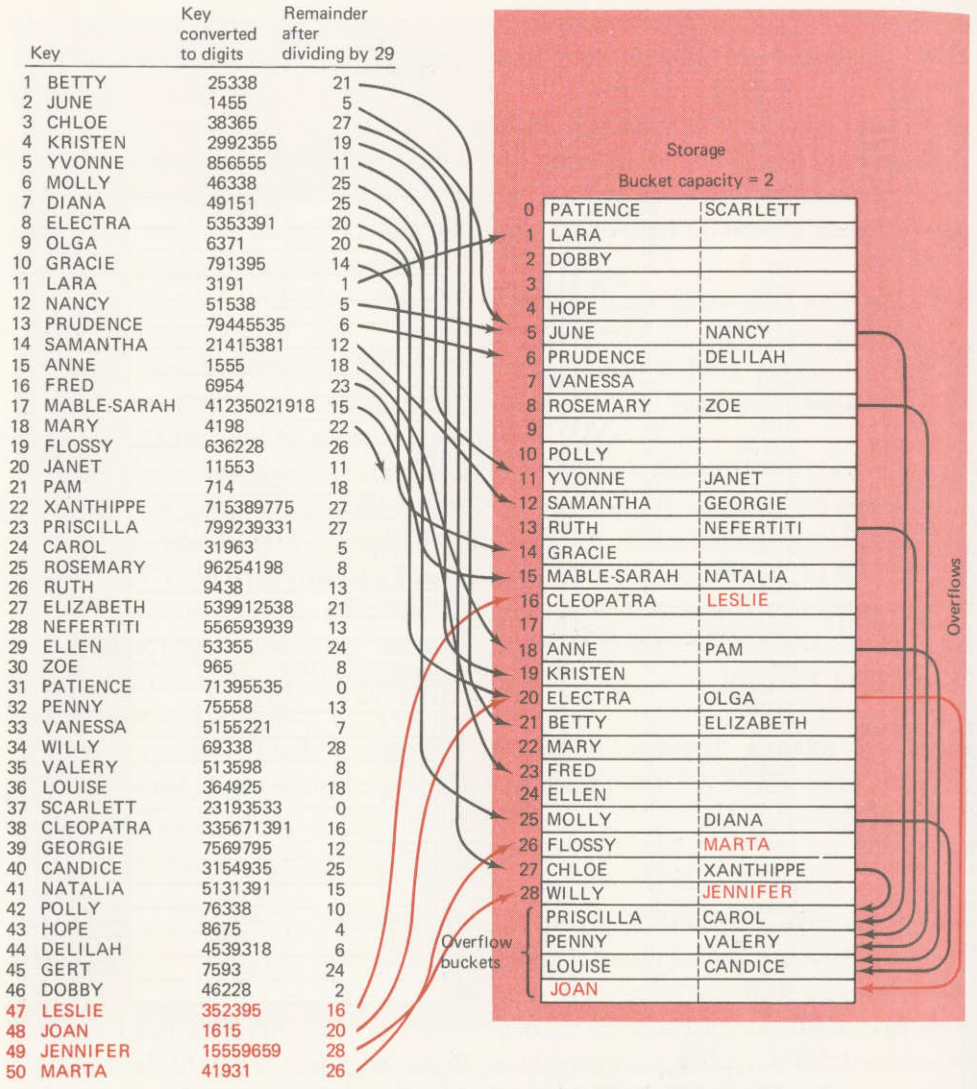
Bucket capacity = 2

0	PATIENCE	SCARLETT
1	LARA	
2	DOBBY	
3		
4	HOPE	
5	JUNE	NANCY
6	PRUDENCE	DELILAH
7	VANESSA	
8	ROSEMARY	ZOE
9		
10	POLLY	
11	YVONNE	JANET
12	SAMANTHA	GEORGIE
13	RUTH	NEFERTITI
14	GRACIE	
15	MABLE-SARAH	NATALIA
16	CLEOPATRA	
17		
18	ANNE	PAM
19	KRISTEN	
20	ELECTRA	OLGA
21	BETTY	ELIZABETH
22	MARY	
23	FRED	
24	ELLEN	
25	MOLLY	DIANA
26	FLOSSY	
27	CHLOE	XANTHIPPE
28	WILLY	
Overflow buckets	PRISCILLA	CAROL
	PENNY	VALERY
	LOUISE	CANDICE

The key is converted into digits by retaining only the four bits which represent numbers in BCD code (see Fig. 32.2). This method should not be used in practice because it throws away information in the key. It is used here to provide an easily-followed illustration.

Figure 21.2 A simple illustration of hashing to a storage with 29 prime buckets, each of capacity 2.

The systems analyst can exercise a trade-off between saving storage and saving time by adjusting the prime packing density. If the key-to-address conversion algorithm scatters the records into buckets at random like the roulette wheel, we could calculate statistically the percentage of overflows



Key	Key converted to digits	Remainder after dividing by 29
1 BETTY	25338	21
2 JUNE	1455	5
3 CHLOE	38365	27
4 KRISTEN	2992355	19
5 YVONNE	856555	11
6 MOLLY	46338	25
7 DIANA	49151	25
8 ELECTRA	5353391	20
9 OLGA	6371	20
10 GRACIE	791395	14
11 LARA	3191	1
12 NANCY	51538	5
13 PRUDENCE	79445535	6
14 SAMANTHA	21415381	12
15 ANNE	1555	18
16 FRED	6954	23
17 MABLE-SARAH	41235021918	15
18 MARY	4198	22
19 FLOSSY	636228	26
20 JANET	11553	11
21 PAM	714	18
22 XANTHIPPE	715389775	27
23 PRISCILLA	799239331	27
24 CAROL	31963	5
25 ROSEMARY	96254198	8
26 RUTH	9438	13
27 ELIZABETH	539912538	21
28 NEFERTITI	556593939	13
29 ELLEN	53355	24
30 ZOE	965	8
31 PATIENCE	71395535	0
32 PENNY	75558	13
33 VANESSA	5155221	7
34 WILLY	69338	28
35 VALERY	513598	8
36 LOUISE	364925	18
37 SCARLETT	23193533	0
38 CLEOPATRA	335671391	16
39 GEORGIE	7569795	12
40 CANDICE	3154935	25
41 NATALIA	5131391	15
42 POLLY	76338	10
43 HOPE	8675	4
44 DELILAH	4539318	6
45 GERT	7593	24
46 DOBBY	46228	2
47 LESLIE	352395	16
48 JOAN	1615	20
49 JENNIFER	15559659	28
50 MARTA	41931	26

0	PATIENCE	SCARLETT
1	LARA	
2	DOBBY	
3		
4	HOPE	
5	JUNE	NANCY
6	PRUDENCE	DELILAH
7	VANESSA	
8	ROSEMARY	ZOE
9		
10	POLLY	
11	YVONNE	JANET
12	SAMANTHA	GEORGIE
13	RUTH	NEFERTITI
14	GRACIE	
15	MABLE-SARAH	NATALIA
16	CLEOPATRA	LESLIE
17		
18	ANNE	PAM
19	KRISTEN	
20	ELECTRA	OLGA
21	BETTY	ELIZABETH
22	MARY	
23	FRED	
24	ELLEN	
25	MOLLY	DIANA
26	FLOSSY	MARTA
27	CHLOE	XANTHIPPE
28	WILLY	JENNIFER
	PRISCILLA	CAROL
	PENNY	VALERY
	LOUISE	CANDICE
	JOAN	

Figure 21.3 Four new records added to the file in Fig. 21.2.

for different prime packing densities. Box 21.1 shows the calculation, and the curves in Fig. 21.4 give the results of this calculation.

The systems analyst ought to be able to find key-to-address conversion algorithms that do better than the roulette wheel, as we will see. Many are worse. The equations in Box 21.1 and the curves in Fig. 21.4 give a useful guideline to the trade-off among prime packing density, bucket size, and the number of overflows.

Box 21.1 Prime packaging density, bucket capacity and numbers of overflows

Let N = the total number of balls, M = the total number of compartments in the roulette wheel, and C = the capacity of a compartment. Then the prime packing density = N/CM.

Using the binomial distribution, the probability that a given compartment will have x balls sent to it in N spins of the roulette wheel is

$$\text{Prob}\,(x) = \frac{N!}{x!(N-x)!}\left(\frac{1}{M}\right)^{x}\left(1-\frac{1}{M}\right)^{n-x}$$

The probability that there will be Y overflows from a given compartment is $P(C + Y)$.

The mean number of overflows from a given compartment is

$$\sum_{Y=1}^{\infty} \text{Prob}\,(C+Y)\cdot Y$$

The percentage of overflows for the entire roulette wheel is therefore

$$100 \times \frac{M}{N}\cdot \sum_{Y=1}^{\infty} \text{Prob}\,(C+Y)\cdot Y$$

From this we can explore the relationship between bucket capacity, C, prime packing density, N/CM, and percentage of overflows in a hashed file. The results are plotted in Fig. 21.4.

If the file is on an electromechanical storage with a long access time, the primary concern may be to cut down the number of accesses. The systems analyst may decide to hold the overflow percentage to 1%. As seen in Fig. 21.4, he may do so by having a prime packing density of 70% and a bucket capacity of 20 or more. On the other hand, access time may be of less concern than the efficient use of storage space. He may decide to use a

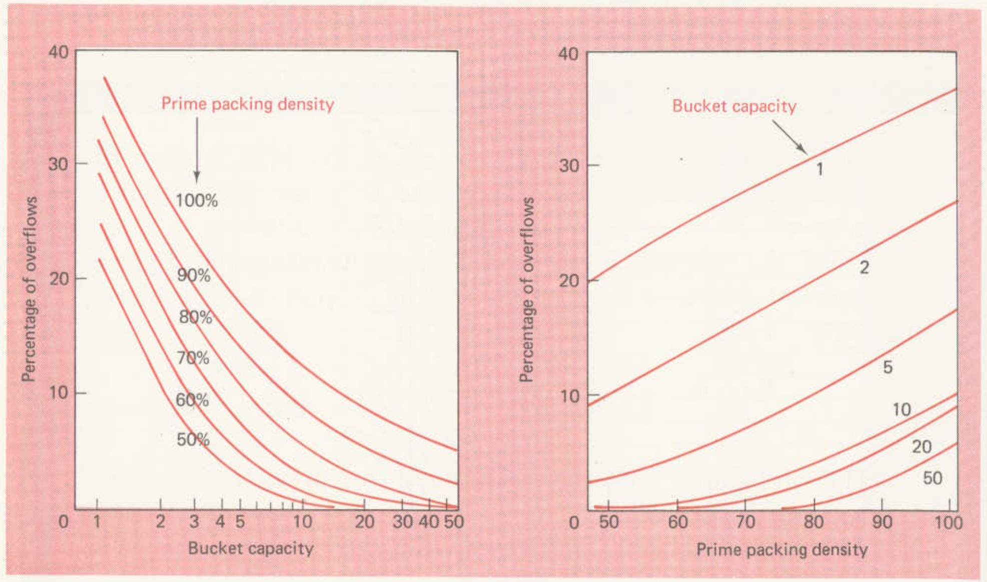


Figure 21.4 The systems analyst can exercise a trade-off between prime packing density, bucket capacity, and percentage of overflows. These curves are drawn from a key-to-address transform which perfectly randomizes the key set, like a roulette wheel. Compare with Fig. 21.7.

prime packing density of 95% and again use a large bucket size to reduce overflows. Conversely, he may want to avoid the computer time needed to search a large bucket. If the file is in solid-state memory so that overflow access is not time-consuming, he may use a bucket capacity of 1 and use a high packing density because this storage is expensive.

KEY-TO-ADDRESS CONVERSION ALGORITHMS

The key-to-address conversion algorithm generally has three steps:

1. If the key is not numeric, it is converted into a numeric form ready for manipulation. The conversion should be done *without losing information* in the key. For example, an alphabetical character should not be converted into *one* digit (as in Fig. 21.2) but into two. Alphanumeric data may be manipulated in the form of binary strings.
2. The keys are operated on by an algorithm which converts them into a spread of numbers of the order of magnitude of the address numbers required. The key set should be distributed *as evenly as possible across this range of addresses.*

3. The resulting numbers are multiplied by a constant which compresses them to the precise range of addresses. The second step may, for example, give four digits when 7000 buckets are to be used. The four-digit number is multiplied by 0.7 to put it in the range 0000 to 6999. This *relative bucket number* is then converted into a machine address of the bucket.

For the second step many transforms have been proposed and tested. It is desirable that the transform distribute the keys as evenly as possible between the available buckets. Realistic transforms distribute the keys very imperfectly, and so overflows result. The following are some of the more useful candidates:

1. Mid-square method

The key is multiplied by itself, and the central digits of the square are taken and adjusted to fit the range of addresses.

Thus, if the records have 6-digit keys and 7000 buckets are used, the key may be squared to form a 12-digit field of which digits 5 to 8 are used. Thus, if the key is 172148, the square is 029634933904. The central four digits are multiplied by 0.7: 3493 $\times$ 0.7 = 2445. 2445 is used as the bucket address.

This is close to roulette-wheel randomization, and the results are usually found to be close to the theoretical results of Fig. 21.4.

2. Dividing

It is possible to find a method which gives better results than a random number generator. A simple division method is such. The key is divided by a number approximately equal to the number of available addresses, and the remainder is taken as the relative bucket address, as in Fig. 21.2. A prime number or number with no small factors is used.

Thus, if the key is 172148 again and there are 7000 buckets, 172148 might be divided by 6997. The remainder is 4220, and this is taken as the relative bucket address.

One reason division tends to give fewer overflows than a randomizing algorithm is that many key sets have runs of consecutive numbers. The remainder after dividing by, say, 6997 also tends to contain runs of consecutive numbers, thereby distributing the keys to different buckets.

3. Shifting

The outer digits of the key at both ends are shifted inward to overlap by an amount equal to the address length, as shown in Fig. 21.5. The digits

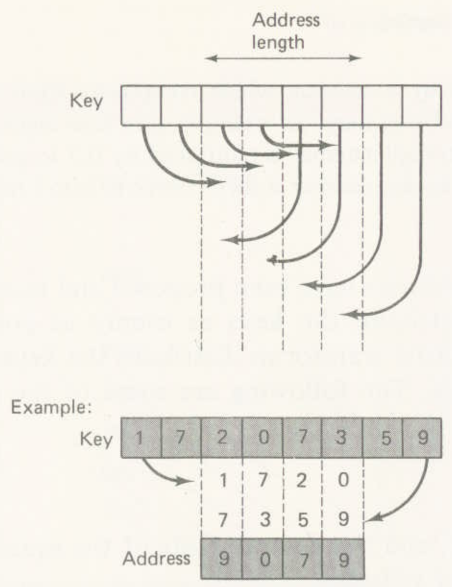


Figure 21.5 Key-to-address conversion by shifting.

are then added, and the result is adjusted to fit the range of bucket addresses.

4. Folding

Digits in the key are folded inward like folding paper, as shown in Fig. 21.6. The digits are then added and adjusted as before. Folding tends to be more appropriate for large keys.

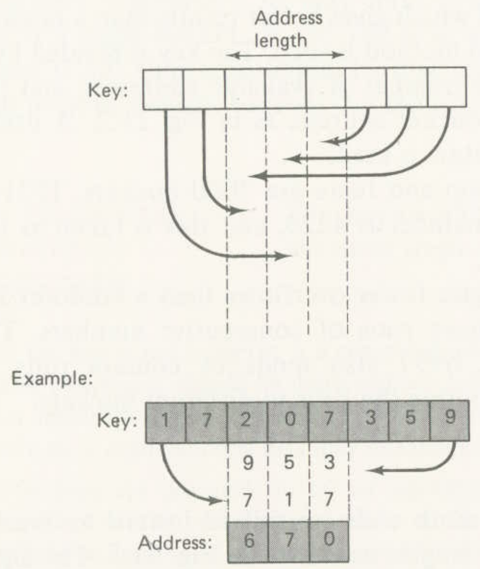


Figure 21.6 Key-to-address conversion by folding.

5. Digit Analysis

Some attempts at achieving an even spread of bucket addresses have analyzed the distribution of values of each digit or character in the key. Those positions having the most skewed distributions are deleted from the key in the hope that any transform applied to the other digits will have a better chance of giving a uniform spread.

6. Radix Conversion

The radix of a number may be converted, for example, to radix 11. The excess high-order digits may then be truncated.

The key 172148 is converted to

$$1 \times 11^5 + 7 \times 11^4 + 2 \times 11^3 + 1 \times 11^2 + 4 \times 11^1 + 8 = 266373$$

and the digits 6373 are multiplied by 0.7 to give the relative bucket address 4461.

Radix 11 conversion can be performed more quickly in a computer by a series of shifts and additions.

7. Lin's Method [9]

In this method a key is expressed in radix p, and the result is taken modulo q^m, where p and q are prime numbers (or numbers without small prime factors) and m is a positive integer.

The key 172148 would first be written as a binary string: 0001 0111 0010 0001 0100 1000. Grouping the string into groups of three bits, we obtain 000 101 110 010 000 101 001 000 = 05620510. This is expressed as a decimal number and divided by a constant q^m. The remainder is used to obtain the relative bucket address.

8. Polynomial Division

Each digit of the key is regarded as a polynomial coefficient; thus, the key 172148 is regarded as $x^5 + 7x^4 + 2x^3 + x^2 + 4x + 8$. The polynomial so obtained is divided by another unchanging polynomial. The coefficient in the remainder forms the basis of the relative bucket address.

CHOICE OF TRANSFORM

The best way to choose a transform is to take the key set for the file in question and simulate the behavior of many possible transforms. For each transform, all the keys in the file will be converted and the numbers of records in each bucket counted. The percentage of overflows for given bucket sizes will be evaluated.

Several researchers have conducted experiments on typical key sets searching for the ideal transform [3]. Their overall conclusion is that the simple method of *division* seems to the best general transform. Buchholz [1] recommends dividing by a prime slightly smaller than the number of buckets. Lum et al. [3] say that the divisor does not have to be a prime; a nonprime number with no prime factors less than 20 will work as well.

Figure 21.7 shows some typical results. The red curve shows the theoretical behavior of a perfectly randomizing transform like a roulette wheel. The points plotted show the average overflow percentages given by three common transforms on eight widely differing but typical key sets. The mid-square method is close to a theoretical randomizing transform. The division method performs consistently better than the randomizing transform. The folding method is erratic in its performance and so is the shifting method, probably because of the uneven distribution of characters in the key sets. Shifting and folding almost always perform less well than division. The more complex methods such as radix transformation, Lin's method, and polynomial division also perform less well, often because their behavior is close to that of an imperfect random number generator.

The ideal transform is not one which distributes the key set *randomly* but one which distributes it uniformly across the address space.

DESIGN RECOMMENDATION

The behavior of the good transforms on actual files is usually somewhat better than that of a perfectly randomizing transform but is fairly close to it. A systems analyst who is designing file layouts would therefore be employing a prudently conservative assumption if he used the roulette wheel calculation of Box 21.1 or the curves in Fig. 21.4 for making estimates of file packing density and percentage of overflows. He should use these curves along with knowledge of the hardware characteristics to select appropriate bucket sizes.

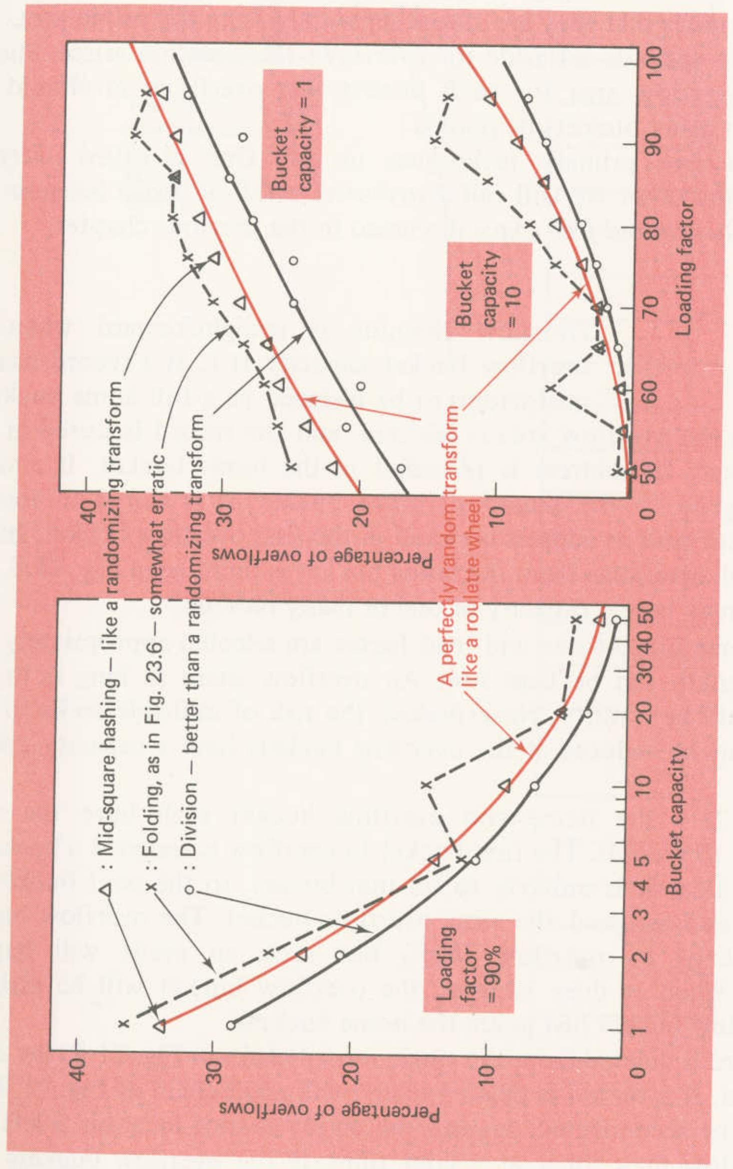


Figure 21.7 A comparison of three popular hashing algorithms with a perfectly random transform. *(Plotted from data averaging the results from eight different files, in Reference 3.)*

WHAT SHOULD WE DO WITH OVERFLOWS?

There are two main alternative places to store overflows. They may be stored in a separate overflow area or in the prime area. The calculation of Fig. 21.4 assumes that they are stored separately from the prime area.

If separate space is set aside for overflows the question arises: Should there be an overflow area for each bucket that overflows or should the overflows from many buckets be pooled?

There are two primary techniques in use. One is called *overflow chaining*, and the other we will call *distributed overflow space* because it is similar to the *distributed free space* discussed in the previous chapter.

OVERFLOW CHAINING

Overflow chaining is straightforward when the overflow bucket capacity is 1. If a record has the misfortune to be assigned to a full home bucket, a free bucket in the overflow area is selected, and the record is stored in that overflow bucket. Its address is recorded in the home bucket. If another record is assigned to the same full home bucket, it is stored in another overflow bucket, and its address is stored in the first overflow bucket. In this way a chain of overflows from the home bucket is built, as in Fig. 21.8. The home bucket may have a capacity of one or many records.

If the home bucket size and load factor are selected appropriately, the mean chain length can be kept low. An overflow chain as long as that in Fig. 21.8 should be a rarity. Nevertheless, the risk of multiple seeks to find one record can be reduced if the overflow buckets have a capacity greater than 1.

In Fig. 21.9 the home and overflow bucket each have the same capacity, say, 10 records. The first bucket to overflow is assigned a bucket in the overflow area. It is unlikely to fill that bucket, so the next buckets to overflow are also assigned the same overflow bucket. The overflow bucket will be unlikely to overflow itself, but such an event will happen occasionally. When it does happen, the overflow bucket will be assigned another overflow bucket just as are the home buckets.

If a record is deleted from the single-record chain in Fig. 21.8, the chain is reconnected. If a record is deleted from the bucket chain of Fig. 21.9, the chain cannot be reconnected. Instead the empty record location is left, and another overflow may fill it at a later time. If the overflow buckets have many deletions, it may be desirable to reorganize the overflow area periodically.

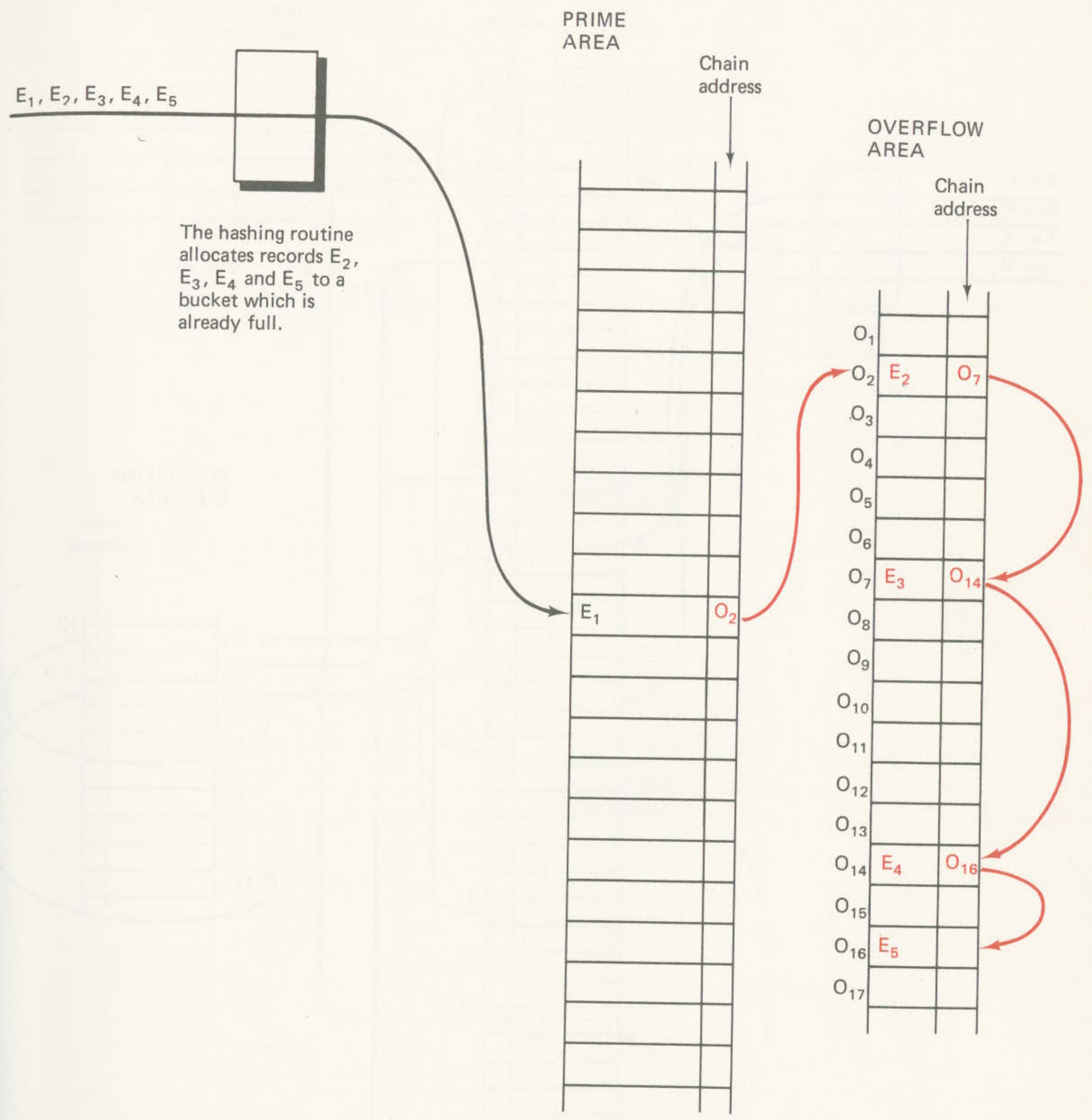


Figure 21.8 Overflow chain with overflow bucket capacities of 1 record.

DISTRIBUTED OVERFLOW SPACE

In Fig. 21.10 there is no chain. Instead overflow buckets are distributed at regular intervals among the home buckets. If a home bucket overflows, the

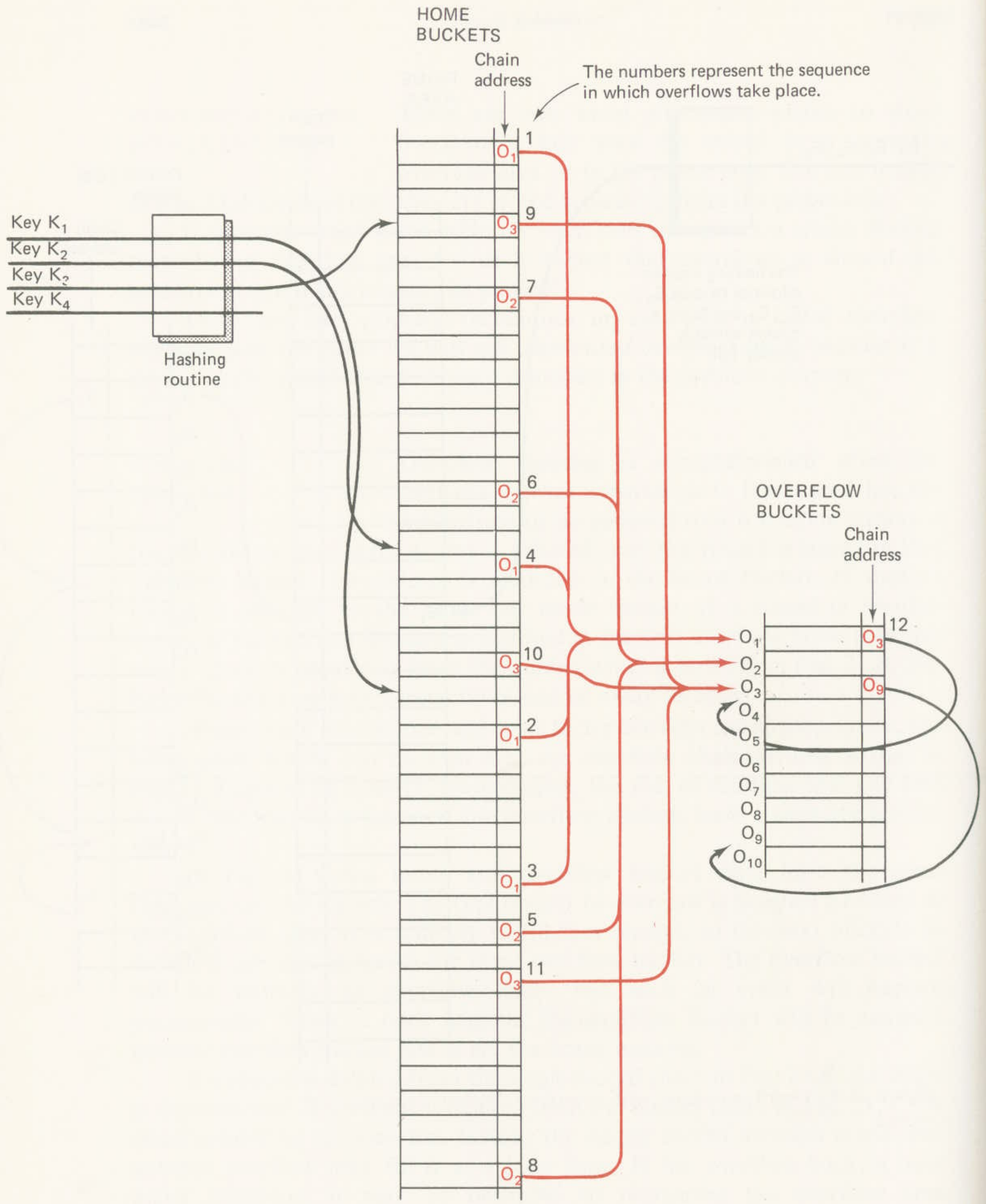


Figure 21.9 Overflow chaining with overflow bucket capacities of many records.

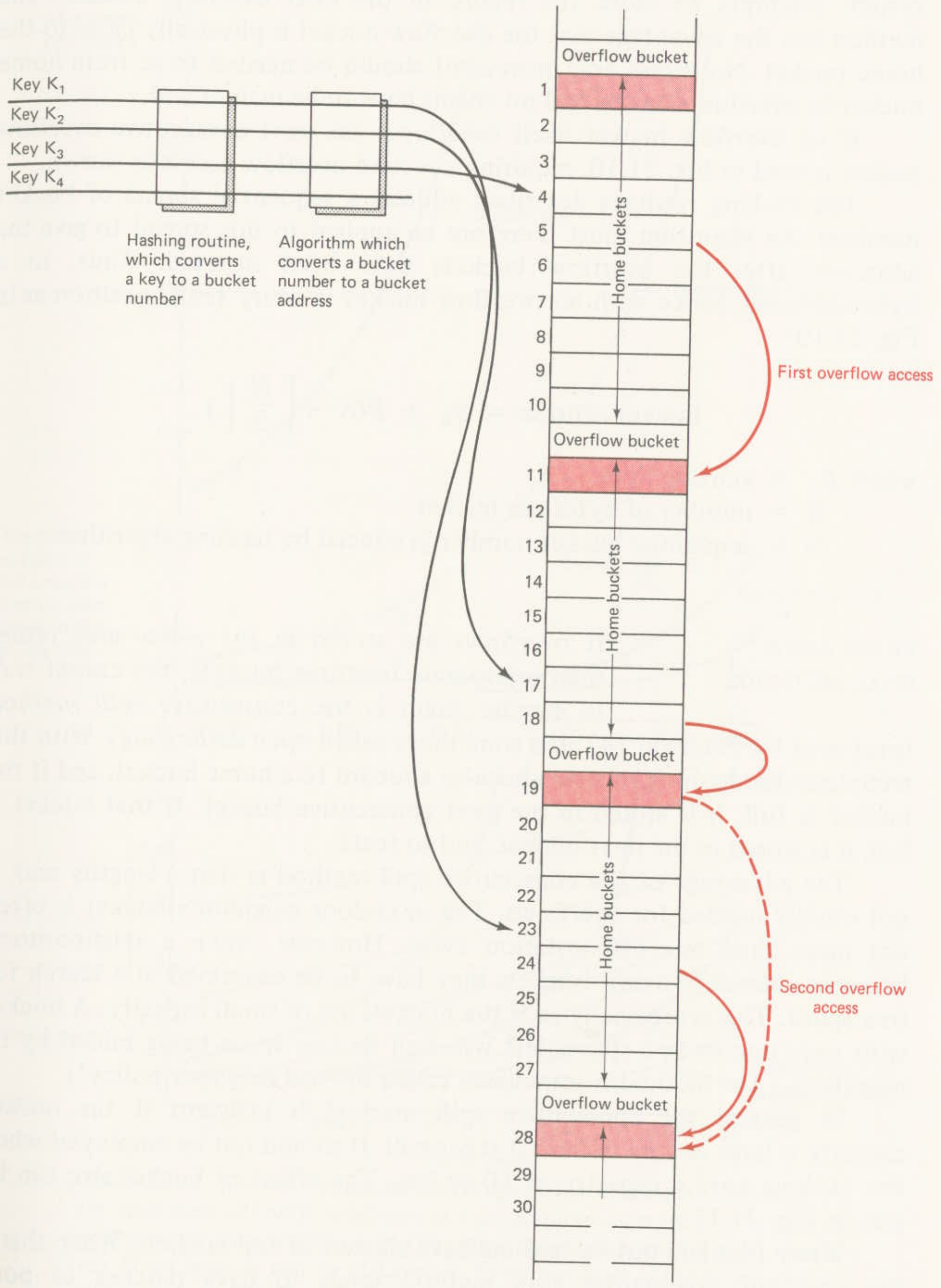


Figure 21.10 Distributed overflow space.

system attempts to store the record in the next overflow bucket. The method has the advantage that the overflow bucket is physically close to the home bucket. No access arm movement should be needed to go from home bucket to overflow bucket, and no chains have to be maintained.

If an overflow bucket itself overflows, the next consecutive overflow bucket is used in Fig. 21.10, requiring a second overflow access as shown.

The hashing routines described address a sequential spread of bucket numbers. An algorithm must therefore be applied to this spread to give the addresses after the overflow buckets have been inserted. Thus, in a byte-addressed device with an overflow bucket in every tenth position as in Fig. 21.10:

$$\text{Bucket address} = B_0 + B\left(N + \left\lceil \frac{N}{9} \right\rceil\right)$$

where B_0 = starting byte
B = number of bytes per bucket
N = sequential bucket number produced by hashing algorithm

PRIME AREA SPILL METHODS

If overflows are stored *in the prime area* rather than in separate overflow buckets, the easiest way to handle them is the *consecutive spill method* (proposed by Peterson [4] and sometimes called *open addressing*). With this technique the hashing routine allocates a record to a home bucket, and if the bucket is full, it is stored in the next consecutive bucket. If that bucket is full, it is stored in the next bucket, and so forth.

The advantage of the consecutive spill method is that a lengthy seek is not usually needed for overflows. The next-door neighbor's bucket is often not more than one disk rotation away. However, when a neighborhood becomes crowded, many buckets may have to be examined in a search for free space. This is especially so if the buckets are of small capacity. A bucket with only one or two spaces left will find its free space being raided by its neighbors. (The method is sometimes called the bad neighbor policy!)

In general the consecutive spill method is efficient if the bucket capacity is large and inefficient if it is small. It should not be employed when the buckets have a capacity of 10 or less. The effect of bucket size can be seen in Fig. 21.11.

Many files laid out by hashing have clusters of full buckets. When this is the case the consecutive spill method tends to have patches of poor performance in which strings of full buckets must be searched. A variation of the consecutive spill method is the *skip spill method* in which, when a

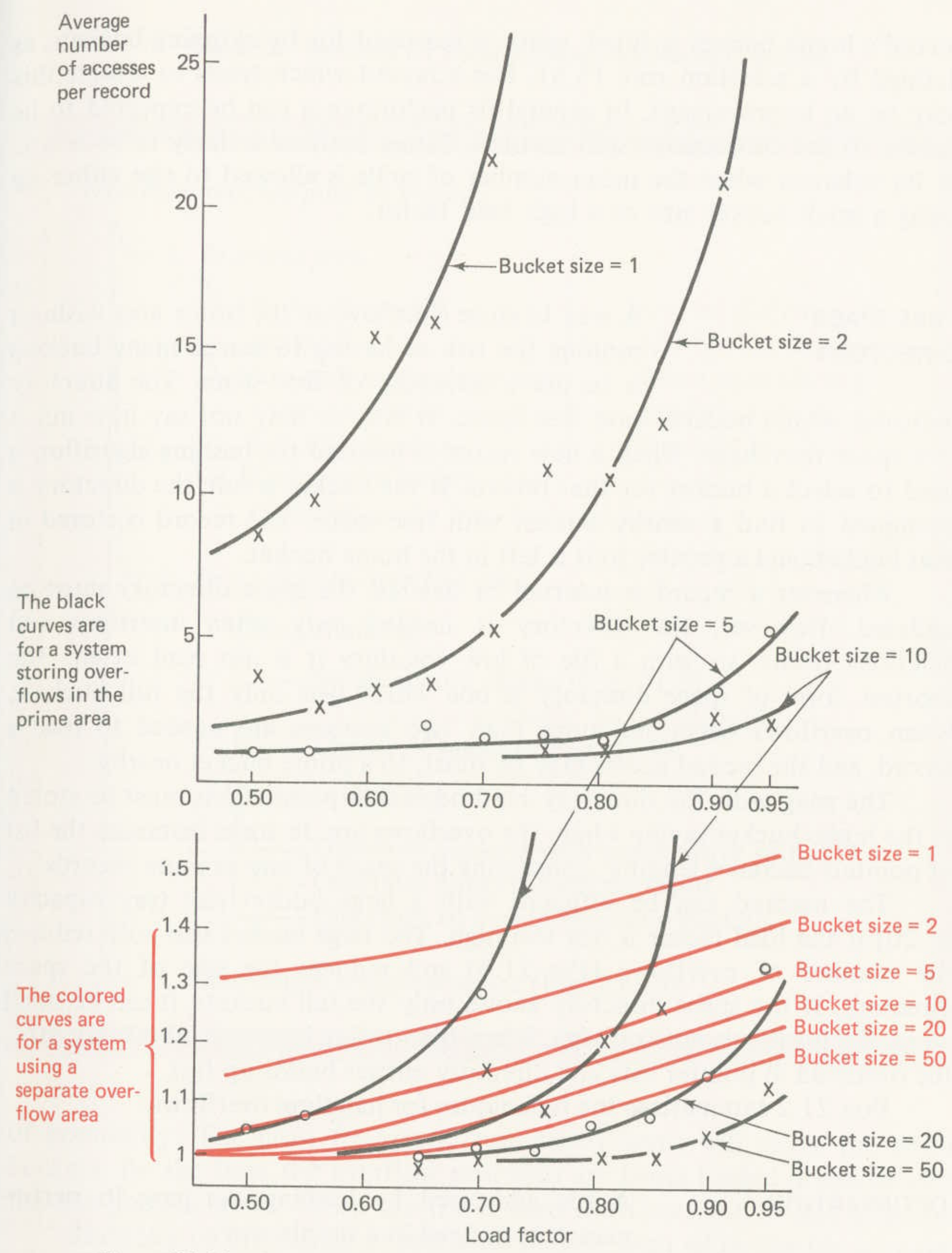


Figure 21.11 Curves comparing overflows in the prime area (consecutive spill method) with overflows in a separate area (in red). It is very inefficient to have overflows in the prime area with a small bucket size (say 10 or less). It can be efficient to have overflows in the prime area with a large bucket size (say 20 or greater). (Plotted from epirical data in reference [3].)

record's home bucket is filled, space is searched for by skipping buckets, as defined by a selection rule [5,6]. For a layout which tends to cluster this may be an improvement. In general its performance can be expected to be similar to the consecutive spill method. Either method is likely to be erratic in its behavior when the mean number of spills is allowed to rise either by using a small bucket size or a high load factor.

FREE-SPACE DIRECTORY

A way to store overflows in the prime area without running the risk of having to search many buckets is to use a directory of free space. The directory indicates which buckets have free space. It may or may not say how much free space they have. When a new record is inserted the hashing algorithm is used to select a bucket for that record. If the bucket is full, the directory is examined to find a nearby bucket with free space. The record is stored in that bucket, and a pointer to it is left in the home bucket.

Whenever a record is inserted or deleted the *space directory* must be updated. However, the directory is needed *only* when insertions and deletions occur, so with a file of low volatility it is not read often. The shortest form of space directory is one which lists only the full buckets. When overflows occur no more than two accesses are needed to read a record, and the second access may be short, to a prime bucket nearby.

The snag with the directory method is that pointer lists must be stored in the home bucket saying where the overflows are. In some instances the list of pointers becomes lengthy, consuming the space of one or more records.

The method can be efficient with a large bucket size (say capacity $\geqslant 20$) if the load factor is not too high. The large bucket size both reduces the number of overflows (Fig. 21.4) and reduces the size of the space directory. If the space directory shows only the full buckets, it can be small when the loading is not too high. When the loading increases, above say 90%, the overhead in pointer lists and directory entries builds up fast.

Box 21.2 summarizes the techniques for handling overflows.

OPTIMIZATION

A file addressed by hashing can have its performance optimized in a simple way.

A small proportion of the records will be stored as *overflows*. Because these records will take longer to access than others, they should be records which are accessed *infrequently*. To achieve this when the file is initially loaded, the frequently referenced items should be loaded first, and the infrequently referenced ones last. If possible, the loading should be in order

Box 21.2 Methods for Handling Hash Addressing Overflows

Overflow records may be stored:

- 1. *In prime area*
  - Consecutive spill method
  - Skip spill method (avoids some congestion caused by clustering)

    (Consecutive spill and skip spill: Very inefficient with low bucket capacities. Good with bucket capacities $>$ 20 (Fig. 21.11).)
  - Directory of empty space (poor when the load factor is high)
- 2. *In a separate overflow area*
  - Using chaining
    - Single-record chains (Fig. 21.8) (Chains occasionally become lengthy)
    - Chains of buckets (Fig. 21.9)
  - Without chaining
    - Overflow buckets with consecutive spill
    - Overflow buckets with skip spill
    - Directory of empty overflow buckets
- 3. *Distributed overflow space* (Fig. 21.10)

of popularity. The most frequently referenced items will go to the home buckets. By the time the overflow positions are being loaded it will be with the less popular items. When the file is used overflows will be less frequent.

Statistics on frequency of reference may be kept while the file is in use, and these statistics may be employed later when the file is reloaded. As with most physical data-base organizations, periodic reorganization can improve performance.

REFERENCES

1. W. Buchholz, "File Organization and Addressing," *IBM Systems J. 2*, June 1963, 86–111.
2. R. Morris, "Scatter Storage Techniques," *Comm. ACM 11*, No. 1, Jan. 1968, 38–44.
3. V. Y. Lum, P. S. T. Yuen, and M. Dodd, "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files." *Comm. ACM 14*, No. 4, April 1971, 228–259.
4. W. W. Peterson, "Addressing for Random-Access Storage," *IBM J. Res. Develop. 1*, No. 2, April 1957, 130–146.
5. C. E. Radke, "The Use of Quadratic Residue Research," *Comm. ACM*, Feb. 1970.
6. J. R. Bell, "The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering," *Comm. ACM*, Feb. 1970.
7. J. A. van der Pool, "Optimal Storage Allocation for Initial Loading of a File." *IBM J. Res. Develop. 16*, No. 6, 1972, 579.
8. J. A. van der Pool, "Optimal Storage Allocation for a File in Steady State," *IBM J. Res. Develop. 17*, No. 1, 1973, 27.
9. A. D. Lin, "Key Addressing of Random Access Memories by Radix Transformation," in *Proceedings of the Spring Joint Computer Conference*, 1963.
10. C. A. Olson, "Random Access File Organization for Indirectly Addressed Records," in *Proceedings of the ACM National Conference*, 1969.

BOOK STRUCTURE

Pointers are used in many of the techniques for representing associations between records or segments. Chapter 22 discusses pointers and the following three chapters give methods of representing the associations between data.