¡le

DATABASE DESIGN, Gio Wiederhold.

Updated CHAPTER 3 part 4, file i3dbd-hash.tex, April 4, 1996

Chapter 3 part 4

Hashed Files

3(4)-0 INTRODUCTION

The two final file organization types to be presented in this chapter are the hashed file organization and the ring file organization. The two methods are quite different, actually so different that they complement each other, so that we will see them used together in Chapter 7. Hashed access uses a computation to determine the file address for a specific record. The ring organization uses pointers to locate successor records.

However, these two file organizations are both distinguished from the file organizations covered in Chaps. 3 and 4 in that they do not derive from any of the previous methods, and also that they have no counterparts in common, manual usage. The access methods used with these files are such that a computer is required to manage their complexity. Otherwise the two methods are quite distinct, and not structurally related, so that either pair of sections can be read without cross reference to the other sections.

3(4)-1 DESCRIPTION AND USE OF THE HASHED FILE

The hashed file exploits the capability, provided by disk units and similar devices, to access directly any block of a known address. To achieve direct addressing, the key of the record is used to locate the record in the file. Hashed access is diagrammed in Fig. 3(4)-1.

There are two components to a hashed file:

1 The file space, organized into m slots. Each slot has the capacity to hold one of the n records of the file.

2 The computation τ which computes the slot address for a record given its key.

We will present the basic organization of hashed files in Sec. 3(4)-1-1. After the general description we will deal with two specific issues: in Sec. 3(4)-1-2 we consider how to compute a suitable address for the record from a key and in Sec. 3(4)-1-3 we consider what to do when the computation leads to a conflict, or *collision*.

In Sec. 3(4)-1-4 we show methods that accomodate growth of a hashed file. In Sec. 3(4)-1-5 alternate algorithms for the key-to-address computation are surveyed and a method for evaluating their effectiveness is provided. In Sec. 3(4)-1-6 shows applications which can use a hashed file organization profitably. Section. 3(4)-2 then evaluates the performance parameters.

The earliest direct-access disk files were used by electromechanical accounting machines which would use a key number, punched on a card, to determine where the remainder of the card contents was to be filed. That number provided the address for *direct* access to the file. Direct access is fast but inflexible. Hashing transforms the key with a computational algorithm before it is used as an address. Hashed access is still fast, since it avoids intermediate file operations. A disadvantage is that the method forces the data to be located according to a single key attribute.

A COMPARISION We can compare hashed access with an indexed-sequential file in that access is provided according to a single attribute; however, records of a direct file are not related to their predecessor or successor records. The hashed file methods use a computation to provide the record address for a key, whereas indexed file organizations search indexes to determine the record corresponding to a given key. Index B-trees used extra space to reduce the effort of insertion. The hashed file uses extra space in the main file to simplify insertion of records into the file.

To be expanded to 288 pt for Figure space
was Fig.3-17

Figure 3(4)-1 Record accessing in a hashed file.

3(4)-1-1 An Overview of Hashed Files

Hashed files are based on direct access to a file, using a relative address as described in Chap. 2-3-3. In immediate implementations of direct access, identification numbers are assigned to the data records that provide a relative address into the file. Thus, employee `Joe` is designated as `257`, and that tells us directly that his payroll record is to be found as record 257. We will list a number of problems with the use of direct access locate data records, and then present the common methods used currently.

PROBLEMS WITH USE OF IMMEDIATE FILE ADDRESSES

- Identification numbers for a person or item may be needed in more than one file. This requires one object to carry a variety of numbers.
- To reuse space, the identification number has to be reassigned when a record has been deleted. This causes confusion when processing past and present data together.
- Natural keys are names, social security numbers, or inventory numbers where groups of successive digits have meaning. Such keys are long; longer than is needed give a unique address to each record. In general, the number of people or items which may be referred to by *natural* keys is much larger than the number of records to be kept on the file. In other words, the key address space is much larger than the file space, and direct use of such a key would fill the file very sparsely.

DEFINITIONS FOR HASHED FILES We define now the concept of a *key-to-address-transformation* or KAT, also denoted by the function τ. The argument for $\tau$is the key field from the record. When a record is to be stored the key is given with the record and τcomputes the slot address for the insertion function of the hashed access method. When a record is to be retrieved a search key must be provided. It will be transformed by τinto a slot address, and the retrieval function of the method will fetch the record from the slot.

The file space is partioned into m *slots*: each slot can hold one record. We need at least one slot per record expected, so that $m \geq n$. In practice the *density* of a hashed file $\frac{n}{m} \approx .0.8 - - - 0.5$.

An actual access to the file obtains a *bucket* of slots at a time, often the size of bucket y is determined by the blocksize so that $y = B/R$. We can expect to find $y_{effective} = \frac{n}{m}y$ records in a bucket.

KEY-TO-ADDRESS TRANSFORMATIONS The problems identified with the immediate use of keys as file addresses are solved by interposing a computational procedure. Such a *key-to-address transformation* translates the key attribute values into relative addresses within the file space.

The incoming keys may be strings of characters or numbers, as defined as the key for the data file. Each relative address to be produced identifies a *slot* into which a record may be placed and ranges from 0 to $m - 1$.

Figure 3(4)-1 shows the access to a hashed file for a new record with key "`John`". The key-to-address transformation algorithm τapplied to the string "`John`" has generated the relative record address "`314`". The same key-to-address transformation is applied to a search key when a record is to be fetched.

Most files will have their own key type and slot range. Even with the same key the procedure may differ. The KAT $\tau$to compute the relative addresses from social-security-number key of a person in the 1000 slot `Employees` file differs from the KAT for the same key in the 150 slot `Sales_personnel` file. The 9-digit social-security-number provides the capability to address $10^9 = 1\,000\,000\,000$ slots.

REQUIREMENTS FOR KATS An ideal key-to-address translation should satisfy two requirements:

1 The source key must be reduced in size to match the slot range.

$2^?$ The slot addresses generated should be unique

these objectives conflict. Although the source keys are presumbly unique, it is impossible to compute a smaller numbers which are unique for arbitrary source keys. We settle hence for a lesser criterion

$2^!$ The slot addresses generated should be as unique as possible

The degree of uniqueness of slot addresses is high if the addresses are uniformely distributed. A uniform distribution of addresses will locate an equal fraction of the keys to each and every slot. Techniques to generate uniformely-distributed random numbers form the basis for such transformations.

There is a wide variety of choices for key-to-address transformations. We will in this initial section examine a single choice:

A randomizing KAT implemented by computing the remainder-of- division

In Sec. 3(4)-1-5 other methods will be considered and evaluated, but all the required principles can be developed using a single type of KAT.

Figure 3(4)-2 shows a small file where record are placed into slots using as KAT the remainder-of-division by 500.

A randomizing transformation for a personnel file uses the social security number as the key. We can assume that the value of the low-order digits of these numbers is evenly distributed and hence there is a high probability of deriving out of these digits a unique number for every employee. If one wants to allow space for up to $n = 500$employees, the value of the key may be divided by 500, leaving a remainder with values between 0 and 499.

```
Al     or   322-43(4)-6178   giving  178
Joe    or   123-43(4)-6284           284
Mary   or   036-23-0373            373
Pete   or   901-23-4784            284
```

Still by chance, two identical slot addresses were generated: `Joe` and `Pete` both received slot number `284`; the records for `Joe` and `Pete` will collide if both are placed directly into a file.

Figure 3(4)-2 Key-to-Address Transformation

The problem with randomizing transformations is that they will generate some identical addresses from different source keys, so that more than one record may be directed to the same place in storage. Such an occurrence is called a *collision*.

Consider using the KAT from Fig. 3(4)-2. The algorithm computed for `Pete`, with `social security number = 901-23-4784`, a slot number of `284` but this did not provide a space for `Pete`'s record. Befire storing a new record the content of the slot itself has to be tested. The slot can be empty, it can contain an older record for `Pete`, or it can already contain a colliding record for `Joe` or someone else.

INSERTION PROCEDURE To determine on insertion whether a collision is occurring we check if the slot is empty. Insertion into an empty slot is straightforward. If the slot is full, there is a collision. Now we compare the key value of the record found in the slot with the key value of the new record. If key fields match, then perhaps the record found should be replaced; the implied operation is an `update` of all attributes but the key.

If the new and old key fields do not match, it is a true collision. Then the rules for collision resolution, as set for the particular file organization, must be followed. The most common choice is *open addressing*, where we search through successor slots untill an empty slot is found. That slot is used for the colliding record.

FETCH PROCEDURE When a record is to be fetched a similar process is followed. An empty slot indicates that there is no record matching the search If there is a record, then the search key is compared with the key of the record in the slot. If they match, the record is retrieved from the slot.

If the new and old key fields do not match, then there had been a true collision earlier and we search further to try to locate a matching record. The retrieval process depends on the method used for collisions resolution on insert. For the scheme used above we will check the successor slots. If the slot is empty we give up, if it is not, the key is checked and a matching record will be fetched. If the record still does not match the search is continued through all slots.

Deletion of records from a hashed file must be handled with care, since just setting slots "empty" can confuse the retrieval process. This problem is further addressed in Sec. 3(4)-1-3.

TERMINATION Both insert and retrieval are sure to terminate as long as the number of slots m is greater than the number of records n. If m is sufficiently greater than n the probability of finding an empty slot is good, and it will only be rarely neccessary to go to slots in a successor block. These concepts will be expanded on in the performance evaluation of Sec. 3(4)-2.

We summarize both the insert and retrieval algorithm as a decision rule below, but first consider what happens if source keys are not unique.

NON-UNIQUE RECORD KEYS Typically records of hashed files are defined to be *unique*. If in this file application multiple records may have the same key, the record to be inserted can still be stored according to the collision resolution scheme used.

If nonunique records, i.e., records with duplicate keys, are to be retrieved, a count is needed to find all matches to the search argument. For deletion either all records with that key must be deleted, or other fields of the record must be inspected.

We will not consider duplicate source keys further in this chapter.

THE DECISION RULE The corresponding outcomes are now:

Condition	Insert	Fetch
`slot_key=NULL`	`ok to insert`	`no record found`
`keys match`	`replace record if permitted`	`record found`
`keys dont match`	`look for another free slot`	`go to successor slots`

The precise handling of collisions is elaborated in Sec. 3(4)-1-3.

INITIALIZATION When a hashed file is created it must be intialized to "empty". This means that for all slots the field to be used for the key is set to some tombstone. In our examples we use Λ to indicate a `NULL` key field.

3(4)-1-2 Randomizing Key-to-address Transformations

The KATs we consider use using a randomizing transformations when translating the key value into relative file addresses.

KEY AND FILE ADDRESS SPACE A key value may range over a large number of possible values, limited only by the maximum size of the key field V. The number of legal keys is up to 10^V for a numeric key, 26^V for a simple alphabetic key. For a social security number the key space size 10^9, for a 20-character name field it is nearly $2 \cdot 10^{28}$

The number of records n to be kept, and the record space of the file, will be much less for all users, even for the Social Security Administration itself. The available file address space is defined in terms of the record capacity or number of slots m of the file. The number of actual records put in the file cannot exceed the number of available slots; hence, if the key uses *base* distinct symbols,

$$base^V \gg m \geq n$$

Since many of the key-to-address algorithms depend on a numeric key value, it may be desirable to convert alphabetic keys into integers. If a value from `0` to `25` (`lettervalue`) can be assigned to the 26 letters used, a dense numeric representation can be obtained without any loss of information, using a polynomial conversion as

```
/* Letters to Integers; applied to the key given in array 'letter'*/
numeric_value = 0;
DO i = 1 TO number_of_letters;
numeric_value = numeric_value * 26 + lettervalue(letter(i));
END;
```

Numbers and blanks occurring in keys require appropriate expansion of this routine, and computer-word overflow has to be considered.

Now the actual key-to-address translation can take place, using the remainder-of-division algorithm described below and assigning $m \approx 1.5 \cdot n \rightarrow 751$ slots for the 500 employees

```
/* Slot number for the key; uses Remainder of Division by a prime */
rel_address = MOD(numeric_value, 751);
```

The file system uses relative addressing, as detailed in Sec. 2-3-3, to translate the sequential slot numbers given in `rel_address` from their range of 0 to $m-1$ into the physical addresses of the devices, probably disk units, used for the storage of the data. Sequentiality of the storage blocks is immaterial for hashed access.

For files it is desirable to compute block addresses instead of addresses of specific record slots. Records with the same block address will be placed sequentially into the block. The *Bfr* slots available in such block comprise a *bucket*. The effect of using buckets is considered below when collisions are discussed.

3(4)-1-3 Collision Management

COLLISIONS When a randomizing key-to-address transformation produces identical addresses for distinct records, we have a collision. In Fig. 3-20 a new entry `Shostakovich` collides with an earlier entry `John`. The resolution of the collisions will occupy us in this section.

The more slots a file has relative to the number of records to be stored, the lower the probability of collisions will be. While we will evaluate such probabilities in detail subsequently, we note that at typical B-tree density ($m/n = 1.45$), we incur a rate of collisions that is quite bearable; even without buckets, we expect collisions less than 35% of the times we access the file.

¡¡ ¿¿

We will now discuss methods of handling collisions, first from the point of view of inserting a new record to a hashed file.

COLLISION RESOLUTION Three strategies are in use to resolve collisions, sometimes in combination. Two methods, *linear search* and *rerandomization*, which use the main file for storage of the records which have collided, the *overflow*, are referred to as *open addressing* techniques. Figure 3-20 shows the third alternative, the use of a separate *overflow area* for such records. The performance of all three methods is significantly improved if first a search through the current block or *bucket* is made.

To be expanded to 396 pt for Figure space
was Fig.3-20

Figure 3(4)-20 Hashed file with collision.

Search in the Bucket An important aspect in the management of overflows from collisions is the grouping of slots into buckets. A typical bucket is equal to one block, as shown in Fig. 3-21. A search through successive slots in the bucket is used to locate space for the new record. Only computational time is required to access the records within a bucket. Only when the bucket itself is filled will one of the three alternative overflow methods require a disk access. If the bucket is of block size B, there will be space for *ovpb* collisions in the block, where

$$ovpb = \left(1 - \frac{n}{m}\right) \frac{B}{R} \qquad \text{3(4)-69}$$

namely, the fraction of free space times the number of slots per block. We can use the blocking factor *Bfr* for B/R if we are careful to remain aware that we are talking here about slots, and not records. The cost of most collisions is now drastically reduced. An analysis of the effect of the use of buckets is unfortunately quite complex. Figure 3-23 gives results for the use of buckets with a linear search in the file. We find for the earlier example ($m/n = 1.45$) and $Bfr = 8$ a reduction in collision cost from 35% to 5%.

Bucket Addresses When buckets are used, the entire block is made available when a record is required. Now a shorter address can be used. If there are *Bfr* slots in a bucket, the address space reduces from m to m/Bfr and the size of an address pointer will be $\log_2 Bfr$ bits less.

To be expanded to 178 pt for Figure space
was Fig.3-21

Figure 3(4)-21 Buckets.

Linear Search in the File When an overflow out of the bucket does occur, the search can continue to the next sequential block. This avoids the costs of seek times to new areas but tends to cluster overflows together. The linear search method for overflow slots is predominant for hashed files, since it is simple and, one hopes, infrequently invoked. As long as the number of entries is less than the number of slots available ($n \leq m$), a record space will eventually be found. Since the availability of a free slot can be determined a priori, no alternate termination method is required.

The problem of clustering is that, when an area of the storage space is densely used, fetches need many search steps. Additional insertions tend to increase the size of the dense area disproportionally. Clustering has been shown to be quite detrimental even for moderately loaded hashed files; an example how clusters develop is shown in Fig. 3-22.

Rerandomization in the File To avoid or to distribute the clusters one may assign the bucket for the overflow at a random position in the file. A new randomizing key-to-address computation, taking other identifying data or using an alternate algorithm, computes a new slot address in the same file space. Rerandomization, when applied to devices such as disk files, causes a high overhead, since generally a new seek is required. Further collisions can occur, and the procedure may have to be repeated a number of times. A randomizing procedure cannot easily guarantee that it will ever locate an empty space. Rerandomization techniques are used commonly when hashing in primary memory, but for files they appear less beneficial. They probably should also be evaluated carefully when it appears that memory is used, but the operating system uses paging to storage for memory management.

Use of an Overflow Area The third method of handling collisions is to put all records which cause a bucket overflow into a separate file with a linkage from the primary record, similar to the overflow chains used in indexed-sequential files. If such a separate overflow area is established, no clustering will take place. An overflow still causes a new seek but avoids the repeated block fetches which can occur with the previous two methods. A difficulty here lies in the allocation of sufficient overflow space, since the number of collisions is not absolutely predictable. The overflow area may fill while the main area has much free space.

The cost of overflows out of the main file can be lowered again by allocating overflow areas on each cylinder. This requires modification of the key-to-address transformation to account for the gaps in the primary addressing space. If buckets are already used, it may be better to use this space to reduce the density per bucket.

Linkage to Colliding Records Once the block for an overflow is located, the record is yet to be found. Overflow records in successor blocks can be found either by simply matching all successive records in the block to the search argument, or by locating the candidate records by use of a chain before doing the comparison.

Without a chain the search for a record or for a free slot for an insertion terminates when an empty slot is reached. This approach constrains the deletion of records, as discussed below. In a densely loaded file it may take many accesses to find an empty slot, because many intervening slots may be filled by records belonging to the set associated with a different primary bucket address. To identify an arbitrary stored record as a member of the set of colliding records for a specific prime bucket, the key of stored record accessed has to be transformed to the corresponding address, which is then matched to the initial bucket address. If the addresses are equal, this record belongs to this *collision set*.

The use of a pointer chain avoids accessing records which do not have the same computed bucket address. Now fetches can be limited to the set of colliding records. The use of chain also improves fetches where the record is not in the file, since the search terminates when the chain pointer is `NULL`, rather than going on to a free slot.

For insertions it is best to use any free slot in the nearest bucket. If chains are used, the inserted record must be linked into the chain. The linkage of records into the chain at the current block improves locality. This is probably more effective than trying to keep records in sequential order by key or by access frequency.

DELETION FROM HASHED FILES Deletion of a record involves some additional considerations. We have to assure that the freed slot does not hinder subsequent fetch usage, and we also would like to have the slot available for reuse to avoid excessive reorganizations.

We will first consider the cases where a chain is used to link the records of a collision set. The use of a chain requires that deletions mark the slot `Empty` and also that the chain is reset so that any successor records can still be retrieved. If there is more than one record in the collision set, the predecessor record has to be rewritten with the pointer value from the deleted record. Subsequent insertions should use free slots close to the primary bucket, so that the search path does not become longer and longer.

To be expanded to 315 pt for Figure space
was Fig.3-22

Figure 3(4)-22 Operations on a file without chaining and with open addressing.

Open addressing methods can be used without a chain. In that case a search for a record to be fetched or for a free slot for an insertion terminates when an empty slot is reached. This means that deleted records cannot just have their slot set to `Empty`, but have to adjust the file to avoid breaking a subsequent search.

Either a tombstone has to be set which indicates that this record, although deleted, still is on the path to successor overflow records, or the last record of the collision set has to be moved to fill this freed slot. In this latter case the slot can still not be marked `Empty`, because it may be crossed by some other collision set. Its tombstone, however, can indicate `Deleted`, and the slot is available for reuse.

Figure 3-22 illustrates what can happen in these clusters of collision sets when we use open addressing, no chains, and sequential search for slots for overflows. Some deleted record spaces can be reused, but the path length for fetches and inserts includes all spaces not explicitly marked empty. After many deletions and insertions there will be many slots which hold markers and no data. This increases the number of slots to be scanned when searching. The effective density of the file is hence greater than n/m, and this can considerably reduce the performance of a hashed file.

A partial, block-oriented, reorganization can collect sufficient information about the collision to set deleted slots which are not within any collision set to `Empty` in order to break these clusters. A complete reorganization would rehash all the keys in the block and place the records in order by computed address.

3(4)-1-4 Accomodating Growth of Hashed Files

¡¡Def problem¿¿ A method, termed *linear hashing* [Litwin[80]], can avoid the reduction of the fetch performance of a hashed file which occurs if the number of records n grows substantially during the existence of the file. The problem of file growth is that an increased density $dens = n/m$ leads to higher rates of collisions, as quantified by Eqs. 3-69, 3-73, etc.

The hashed file grows through insertion of records, and an inserted record may cause a collision. To reduce the probability of ever-increasing collisions we let the hashing space grow also, by appending blocks linearily to the end of the hash space. For collisions which still occur overflow blocks are used. We will use m_0 to indicate the initial, basic allocation, and let m_0 be a power of 2. The current allocation m_1 begins with m_0. THe simplest growth policy is to increase the allocated space m_1 by one slot for every collision; this will ensure that in the long term the collision rate is constant. The actual space allocation in a file is by buckets which have *Bfr* slots, so that the file is actually made to grow from $m_1 \Rightarrow m_1 + Bfr$ every *Bfr* collisions.

Whenever a new block is assigned it is populated by splitting a block, but the split is assigned differently from a split seen in B-trees. The block to be split is **not** the block where the collision occurred, but in linear hashing the block to be split is the *lowest numbered* block not yet split. This scheme avoids auxiliary tables to indicate which blocks have been split, only the current limit m_1 has to be known.

The method for addressing and the method for splitting cooperate. The block to be split has `rel_address = m1 - m0` so that the first block to be split corresponds to `rel_address = 0`. The address computation uses the same type of key-to-address transformations but uses for the space parameter m twice the initial allocation m_0, so that an extra high-order bit is available. The generated address is reduced to the current range $0, \ldots, m_1$ by an additional statement

```
IF rel_address > m1 THEN rel_address = rel_address - m0
```

This corrects the addresses of all blocks not yet split: $(m_1 - m_0), \ldots, m_0$.

A collision creates an overflow but can also indicate the need to append a new block. In that case a new block is allocated and the current `m1` is incremented. The block at $m_1 - m_0$ is now to be split. The hash addresses of the records in that block and any overflows attached to that block are recomputed using the incremented value of m_1. All those records will either have the original address (the high-order bit was zero and their `rel_address = m1-m0`) or will have the new address `m1`, referring to the new block! The records which now belong into the new block are moved and both the split block and the new block are rewritten.

When the file has grown to twice its size ($m_1 = m$) the hashing function is adjusted by another bit: m and m_0 are doubled, and the process can continue with $m_1 = m_0$. Methods to actually allocate these blocks are presented in Sec. ¡11-5¿.

3(4)-1-5 A Survey of Key-to-Address Transformation Methods

Of the many techniques which have been proposed and evaluated, only some are used for files.

¡¡ Two categories of these computations may be distinguished, *deterministic procedures* which translate identification fields into unique addresses, and *randomizing techniques* which translate the keys into addresses which are as unique as possible but do not guarantee uniqueness.¿¿

A *deterministic procedure* takes the set of all key values and computes a unique corresponding relative address. Algorithms for such transformations become difficult to construct if the number of file entries is larger than a few dozen. Adding a new entry would require a new algorithm, since the algorithm is dependent on the distribution of the source keys; hence only static files can be handled. The replacement of a computational algorithm with a table makes the problem of transformation more tractable: we have invented again the indexed file! We will not discuss deterministic hashed access further.

¡¡ Randomizing in intro ¿¿

The randomizations may be designed to try to preserve the order of the records, or it may be designed to maximize the degree of uniqueness of the resulting address. Most transformations take the latter as their only goal. The first class of transformations are called *sequence maintaining*; the other algorithms are referred to as *hashing techniques*. A family tree of key-to-address transformations is displayed in Fig. 3-18. A simple technique is illustrated in Fig. 3(4)-2.

To be expanded to 180 pt for Figure space
was Fig.3-18

Figure 3(4)-18 Key-to-address transformation types.

Distribution-dependent methods depend on some knowledge of the distribution of keys for the expected records. If we have, for instance, assigned random numbers to identify 1000 samples of some experiment being recorded, we can expect that any three digit positions of the sample identification will be randomly distributed, and suitable as direct-access keys. The liabilities of distribution-dependent transformations are major, since a change in key distribution can cause these methods to generate many more collisions than previously. A benefit of some distribution dependent key-to-address transformations is that they can allow maintenance of sequentiality. Such a *sequence-maintaining key-to-address transformation* achieves this goal by letting the addresses produced increase with increasing keys. Serial access is made possible in this case. Otherwise basic hashed files do not have the capability for serial access. Two distribution-dependent methods are *digit analysis* and *sequence maintaining transformations.*

The three hashing methods presented here (*remainder-of-division*, *exclusive-OR*, *folding-and-adding*) randomize the properties of the digits of the key to a uniform distribution. Operations such as arithmetic multiplication and addition, which tend to produce normally distributed random values (see Sec. 6-1), are undesirable when hashing.

The *digit-analysis-method* attempts to capitalize on the existing distributions of key digits. An estimate or a tabulation is made for each of the successive digit positions of the keys of a sample of the records to be stored. The tabulation lists the frequency of distribution of zeros, ones, twos, and so on. The digit positions that show a reasonably uniform, even distribution are candidates for use in the slot address. A sufficient number of such digit positions has to be found to make up the full address; otherwise combinations of other digit positions can be tested. In Fig. 3(4)-2 the three low-order digits of a social security number were considered to be uniformly distributed. Similar use can be made of character-string keys.

A *sequence-maintaining-transformation* can be obtained by taking an inverse of the distribution of keys found. This inverse function is applied to the keys. The addresses generated will maintain sequentiality with respect to the source key. In a *piecewise-linear-transformation* the observed distribution is approximated by simple line segments. This approximation then is used to distribute the addresses in a complementary manner. An *exponential transformation* is presented in ¡¡Example 6-1 ¿¿ as an illustration of the evaluation of the uniformity of obtained addresses. These approaches are valid only if the source key distribution is stable.

The *remainder-of-division* of the key by a divisor equal to the number of record slots allocated m was used to obtain the desired address above. The remainder does not preserve sequentiality. The remainder of division is in some sense similar to taking the low-order digits, but when the divisor is *not* a multiple of the base (10 in Fig. 3(4)-2 above) of the number system used to represent the key, information from the high-order portions of the key will be included. This additional information scrambles the result further and increases the uniformity of the generated address.

Large prime numbers are generally used as divisors since their quotients exhibit a well-distributed behavior, even when parts of the key do not. In general, divisors that do not contain small primes (≤ 19) are adequate. If the address space m_0 is a power of two then $m = m_0 - 1$ is often suitable. Tests [Lum[71]] have shown that division tends to preserve, better than other methods, preexisting uniform distributions, especially uniformity due to sequences of low-order digits in assigned identification numbers, and performs quite well.

The need to use a division operation often does leads to programming problems. The key field to be transformed may be larger than the largest dividend the divide operation can accept, and some computers do not have divide instructions which provide a remainder. The remainder then has to be computed using the expression

```
rel_address = key - FLOOR(key/m) * m
```

The explicit `FLOOR` operation is included to prevent a smart optimizer from generating `address=0` for every `key` which would lead to all records colliding. Costly divisions can be avoided by replacing them by multiplication of the reciprocal of m, but the behavior of the result may differ.

The *exclusive-OR* (`x-or`) operation provides a very satisfactory randomization and can be used when key lengths are great or division is otherwise awkward. The `x-or` operation is available on most binary computers or can be obtained by statements as

```
DECLARE (rel_address,keypart1,keypart2) BIT(19);
x_or: PROCEDURE(arg1,arg2); DECLARE(arg1,arg2) BIT(*);
RETURN( (arg1 ∨arg2) ∧( ¬(arg1 ∧arg2)) );
END x_or;
...
rel_address = x_or(keypart1,keypart2);
```

As the example shows, the key is segmented into parts (here 2) which match the required address size. Using this operation results in random patterns for random binary inputs. If the segments contain characters, their boundaries should be avoided. For instance, `x_or('MA','RA')` will be equal to `x_or('RA','MA')`, so that "`MARA`" will collide with "`RAMA`". Care has to be taken also where the binary representation of decimal digits or characters is such that certain bit positions always will be zero or one. Both problems can be controlled by making the segment sizes such that they have no common divisor relative to character or word sizes. This operation is generally among the fastest computational alternatives provided by the hardware.

Folding and adding of the key digit string to give a shorter string for the address has been often used. Alternate segments are bit-reversed to destroy patterns which arithmetic addition might preserve. A carry from a numeric overflow may be added into the low-order position. This method is available in the hardware of some large HONEYWELL computers.

PROBABILITY OF SUCCESS WITH RANDOMIZATION METHODS We assess now the performance risks in randomization. A graphic or quantitative understanding of the processes of transformation can aid in developing a feeling for the effects of various methods to achieve random distributions [Peterson[57]]. The number of ways to transform n keys into m addresses is huge; there are in fact m^n possible functions. Even for the most trivial case, say, $n = 4$ records and $m = 5$ slots 625 transforms exist. Of these, however, only $m!/(m-n)! = 125$ would avoid any collisions while loading the file. It would obviously be most desirable to find one of these. On the other hand, we have $m = 5$ possibilities for a total collision; all records wind up in the same slot, and in those instances, the expected number of accesses for a fetch, using a chain, would be $(n+1)/2 = 2.5$; the remaining 495 transforms cause one or two collisions.

Since the randomization method is chosen without any prior knowledge of the keys, the selection of any reasonable method out of the m^n choices gives a probability of that method causing no collisions ($o_0 = 0$) of only $p_0 = (m!/(m-n)!)/m^n = 0.2$. On the other hand, the probability of selecting one of the five worst methods for this case, which would lead to $c = n - 1 = 3$ collisions and $o_{max} = 1 + 2 + 3$ extra accesses, is only $p_{max} = m/m^n = 0.008$. The remaining three cases of one collision, two collisions in two slots, and two collisions in the same slot can be analyzed similarly and are shown in Fig. 3-19. Given the assumption that we have just an average randomization, we find that we may expect an average of $p = \sum_c p_c o_c = 0.30$ additional accesses per record loaded.

To be expanded to 180 pt for Figure space
was Fig.3-19

Figure 3(4)-19 Fetch-length distribution.

We expect at the final point that $o = \sum_c p_c c = 1.05$ records collided and were not loaded within the main file; they will have been placed somewhere else. The main file now contains $n - o = 4 - 1.05 = 2.95$ records. The collision probability for the next insertion is $2.95/5 = 0.59$.

The distribution will have a similar shape for other values of m and n. The number of steps rapidly increases with n and m so that the distribution function becomes smooth. Estimates for the number of collisions in the general case are derived in Sec. 6-1, but the intent of this discussion is to show the relative unimportance of the specific choice of randomization method, as long as it is chosen to be outside of the obviously worst areas. It might be mentioned here that a change of keys, that is, a new set of data to be randomized, will completely reorder the position of the m^n functions in the figure, but the form of the distribution remains the same.

SUMMARY OF KEY-TO-ADDRESS TRANSFORMATIONS It is obvious from the above that many choices exist for transforming keys of records to record addresses. We find also that the average performance of reasonable transformations is such that perfectly good results are obtained with less than perfect methods. Cute methods, as deterministic and sequence-maintaining schemes, carry in fact a high risk of failure when conditions change.

Since the placement of the records is determined by a computation, only one attribute of the record can be used for retrieval. Another inherent restriction due to the computation is that data records are expected to be of fixed length.

Records of randomized hashed files can be accessed only via one precise attribute value and not by range, and not serially. To locate the next record in key sequence requires knowing or guessing the value of the key, and this violates our definitions for `Get-Next` operations. Guessing a successor key would, in the great majority of fetches, result in a *record-not-found* condition. In practice this choice is excluded from consideration.

Some solutions to the problems listed here are shown in Sec. 4-6; they all carry additional access costs. All randomizing methods may create identical addresses, even when the source keys differ. Now we consider what may be done in that case.

3(4)-1-6 Use of Hashed Files

Hashed random access is an invention specific to computers and has been used to access data on the earliest disk files. Hashed files find frequent use for directories, pricing tables, schedules, name lists, and so forth. In such applications where the record sizes are small and fixed, where fast access is essential, and where the data are always simply accessed, the hashed file organization is uniquely suitable. Simple access means here use of a single key for retrieval, and no serial access.

Hashed files also play an important role as a component of more complex file organizations. We will find many uses of hashed files in Secs. 4-6 and 4-7.

3(4)-2 PERFORMANCE OF HASHED FILES

The performance of hashed files has been more thoroughly analyzed than the performance of any of the preceding methods. The initial parameter in these evaluations is the number of record spaces or slots m that are available for the storage of the n records. The number of records that still cause collisions is denoted here as o.

> We will first analyze a simple hashed-access structure with the main file having buckets that hold only a single record and with a separate overflow area to contain up to o records that caused collisions. The records are of fixed length, and contain a single pointer field to allow access to any successor records due to overflows. With the fixed record format the attribute names are not kept within the records, so that space is only needed for the a data values of size V.
>
> Access to the single overflow area will require a seek. There are no identical keys, i.e., all stored records are unique, so that all collisions are due to the randomization. The number of records stored n is still smaller than the number of slots m, so that a perfect distribution would not lead to any overflows and hence keep $o = 0$.
>
> We let overflow records be found via a chain of pointers starting from the primary area. The overflow chain is maintained so that blocks containing overflow records are accessed in sequence. This avoids accessing the same block more than once in a chain.

We will, when appropriate, also make comments which pertain to hashed files using open addressing and to the use of multirecord buckets, since these methods are used frequently.

3(4)-2-1 Record Size in a Hashed File

The space required for a file S_F as described above is

$$S_F = m\,(a\,V + P) + o\,(a\,V + P) \qquad 3(4)\text{-}70$$

or per record,

$$R_{effective} = \frac{m+o}{n}(a\,V + P) = \frac{m+o}{n}R \qquad 3(4)\text{-}71$$

The required overflow space is based on the fraction of collisions p, as discussed below, but should also consider the expected variability of that estimate, so that this area will in practice never overflow between reorganizations. For a file with many records the expected variability is small, and o can be safely dimensioned to be 25% to 100% greater than pn. The estimation of the needed overflow size o is elaborated in Sec. 6-1-5.

3(4)-2-2 Fetch Record in a Hashed File

In order to predict an average for time to locate a record in a hashed file, we must determine the probability of collision p, since the predicted fetch time is simply the time required for randomization, the time for the single hashed access, plus the average cost of the case of a collision

$$T_F = c + s + r + btt + p\,(s + r + btt) \qquad \langle\text{basic case}\rangle\ 3(4)\text{-}72$$

An analysis which estimates the collision cost p for this design is given in Sec. 6-1-5, leading to Eq. 6-15. The result of the analysis shows that for hashed files, single-slot buckets, and using separate overflow areas for collisions, the expected value of p is

$$p = \frac{1}{2}\frac{n}{m} \qquad \langle\text{with overflow area}\rangle\ 3(4)\text{-}73$$

To be expanded to 288 pt for Figure space
was Fig.3-23

Figure 3(4)-23 Overflow cost for multirecord buckets, open addressing, and linear search.

Effect of Open Addressing In open addressing with a linear search clusters of collisions can occur. The detrimental effect of clusters increases rapidly with the file density n/m, as seen in Fig. 3-23 for the bucket size $B/R = 1$. Knuth[73S] has derived, using the arguments of an "average" randomly selected randomization discussed above, that the number of additional accesses becomes

$$p = \frac{1}{2}\frac{n}{m-n} \qquad \langle\text{with open addressing}\rangle\ 3(4)\text{-}74$$

The result for bucket sizes $B/R > 1$, as derived by Knuth[73S], is complex and hence is presented in graphical form as Fig. 3-23. The overflow cost is based on the initial probability of an overflow plus all further overflows to be accessed, so that under poor conditions $p > 1$.

It should be noted that no overflow area is required when open addressing is used, so that for the same total amount of storage the value of m may be appropriately larger. Increasing m to $m + o$ will not compensate fully for the increased number of collisions, as demonstrated in Fig. 3(4)-10.

Measurements of hashed files have produced results which are in close agreement with Eqs. 3-72 and 3-73 [Lum[71]]. The values shown in Fig. 3-23 have been experimentally verified by Deutscher[75] using the remainder of division algorithm to compute the addresses. Even for poorly distributed keys the results became valid when bucket sizes $B/R \geq 5$.

We can now evaluate the effect of collisions. We will provide 50% more primary space than the amount required for the expected records themselves, i.e., $m = 1.5\,n$. For the basic hashed file – only one record space per bucket and a separate overflow area – we obtain a collision cost of

$$p = \frac{1}{2}\,\frac{n}{1.5\,n} = 0.3333 \qquad \text{or about 33\% extra accesses}$$

due to collisions when the file is fully loaded. A reasonable overflow area in this case may be based on $1.5\,p$, giving $o = 0.5\,n$.

With open addressing, using the same primary space m and still bucket sizes equal to R, the value of p becomes

$$p = \frac{1}{2}\,\frac{1}{1.3(4)-1} = 1.0 \qquad \text{or 100\% extra accesses}$$

Adding the space no longer required for the overflow area o to the space for the main file m reduces this estimate to

$$p = \frac{1}{2}\,\frac{1}{1.5+0.5-1} = 0.5 \qquad \text{or 50\% extra accesses}$$

The use of multirecord buckets has a greater effect. If the record size $R = 200$ and the block size $B = 2000$, the bucket size can be made equal to $Bfr = 10$; this change increases only the computational overhead for a block retrieved into memory. We will *not* add the overflow space to the main file, so that $m = 1.5\,n$. We obtain from Fig. 3-23

$$p = 0.03 \;\text{ for }\; \frac{n}{m} = 0.66,\; Bfr = 10 \qquad \text{or about 3\% extra accesses}$$

Since now only 3% of the fetches will have to access another block, a linear, sequential scan for overflows is reasonable.

Figure 3(4)-10 Collision cost assessment

LOCATING THE SUCCESSOR BLOCK In linear searching, if the decision to read the next block can be made immediately, no revolution needs to be wasted. In general, however, one revolution is required to read the next block if the read decision is based on the contents of the predecessor block, so that the access time for an overflow is $T_{RW} + btt$ or $2r + btt$. Alternate-block accessing, as shown in Fig. 3-24, can reduce this delay to $2\,btt$, leaving a time $c < btt$ for the comparisons within a bucket. This technique does not require any general block renumbering, since the decision where to place overflows is specific to the procedure for hashed-file access.

We find overflow records in successor blocks by use of a chain. The search for a nonexistent record can be terminated at an end-of-chain flag. Multiple records may be found within the same block. In linear search we can expect, without clustering, $(m-n)/m\,Bfr$ free slots in a block, and these are available for insertion. A failure to find the record in a successor block p_2 is then

$$p_2 \approx p^{\frac{m-n}{n}Bfr} \qquad \langle\text{linear search}\rangle \; 3(4)\text{-}75$$

which makes the event quite rare. When needed, successive blocks are accessed using the chain of records extending from the primary bucket.

When a separate overflow area is used, the probability of not finding the successor block in the same bucket is higher and can be evaluated based on the number of blocks used for the overflow area. This value is

$$p_2 \approx o'/Bfr \qquad \langle\text{separate overflow area}\rangle \; 3(4)\text{-}76$$

We will ignore p_2 in the fetch evaluation below, since we will always choose methods with a low primary collision cost p.

In summary, the fetch time for the various cases becomes

$$T_F = c + s + r + btt + p\,t_{overflow} \qquad \langle\text{all cases}\rangle \; 3(4)\text{-}77$$

where the choices for p are given above and the values for the overflow cost are

$$\begin{aligned} t_{overflow} &= s + r + btt && \text{for a separate overflow area} \\ t_{overflow} &= r + btt && \text{for separate overflow areas on the same cylinder} \\ t_{overflow} &= 2r + btt && \text{for linear searching, sequential blocks} \\ t_{overflow} &= 2\,btt && \text{for linear searching, alternate blocks} \end{aligned}$$

and c depends on the complexity of the hashing algorithm and the bucket size.

FETCH FOR A RECORD NOT IN THE FILE In a fetch for a nonexistent record, we will be forced to follow the chain search until we find a `NULL` pointer. This time is greater than the time required for fetching an existing record, since we expect to find records on the average halfway into the chain. This condition occurs also when insertions are considered (T_{Iov}), since all the records of an overflow chain have to be tested to locate an empty slot for insertion.

The effect can be estimated by setting the term $p\,t_{overflow} \approx 2\,p\,t_{overflow}$ for the fraction of records(`not_in_file`) in Eq. 3-77 above.

To be expanded to 153 pt for Figure space
was Fig.3-24

Figure 3(4)-24 Alternate bucket overflow accessing.

3(4)-2-3 Get-Next Record of a Hashed File

No concept of serial access exists with the hashed file. If the key for the next record is known, a successor record can be found using the same fetch procedure used to find any record.

$$T_N = T_F \qquad 3(4)\text{-}78$$

If the key for the next record is not known, no practical method exists to retrieve that record.

For hashed files using linear key-to-address translations, which do maintain a serial ordering, the successor record is found by sequential reading, skipping over any unused slots. The probability of finding the record in the next slot can be estimated using the density of the main file n/m. A low density reduces performance here. If a separate overflow area is used, and the probability of overflow is significant, a case-analysis approach, as used for an indexed-sequential file (Eq. 3-40), may be neccessary.

3(4)-2-4 Insert into a Hashed File

To insert a record into the file, an empty slot has to be found which can be reached according to the key. The key-to-address transformation generates a prime bucket address. The bucket must be fetched and checked to determine whether there is a free slot for the record. We will consider now again buckets of only one slot and the use of a chain of records which collided. The probability of the initial slot being filled p_{1u} has also been derived in Sec. 6-1-5, Eq. 6-18 for the case of a separate overflow area:

$$p_{1u} = 1 - e^{n/m} \qquad \langle\text{overflow area}\rangle\ 3(4)\text{-}79$$

If open addressing is used all records are in the file and we can simply use the density:

$$p_{1u} = \frac{n}{m} \qquad \langle\text{open addressing }\rangle\ 3(4)\text{-}80$$

For multiple records per bucket the value of p_{1u} can again be best determined from Fig. 3-23.

To be expanded to 186 pt for Figure space
was Fig.3-25

Figure 3(4)-25 Overflow record linked as second chain member.

The insertion cost can be computed as the sum of the expected cost for the case where the primary slot is empty, which has the probability of $(1 - p_{1u})$, and the cost for the case where the primary slot is filled, a probability of p_{1u}. In both cases the primary block is fetched first. If the slot is found empty, the record is placed into the empty slot, and the block rewritten; otherwise the existing record is kept and one of the overflow procedures discussed has to be followed.

When a *separate overflow area* is used, the procedure is to link the new record as the second member of the chain, as illustrated in Fig. 3-25. The block with the primary record is still rewritten, now with the existing record and the latest address in the overflow area in its pointer field. The new record is placed into the overflow area at position `last` with the pointer value obtained from the primary record. The two rewrites require

$$T_I = c + s + r + btt + T_{RW} + (1 - p_{1u})(s + r + btt + T_{RW}) \qquad \langle\text{simple}\rangle\ 3(4)\text{-}81$$

If duplicate records with the same key are not permitted, all in the bucket records have to be checked prior to insertion of the new one. This means that the entire chain must be followed. The cost considerations for the fetch of a nonexisting record apply $p(\texttt{not\_in\_file})$ and the search time for checking involves the same terms used in (Eq. 3-77). When the end of the chain is reached (pointer is `NULL`), a block with the selected prior record and the new inserted record is rewritten. If the block where the search terminated is full, an additional block fetch and rewrite is required. The probability of this event is only $1/Bfr$. The primary block is not rewritten unless it is the terminal block.

$$\begin{aligned} T_I = {} & c + s + r + btt \\ & + 2\,p_{1u}\,t_{overflow} + T_{RW} + \frac{1}{Bfr}(s + r + btt + T_{RW}) \end{aligned} \qquad \langle\text{with checking}\rangle\ 3(4)\text{-}82$$

The cost of following the chain is similar for overflow areas and for open addressing.

If buckets with many slots per bucket are used, the secondary overflow probabilities p_2 reduce rapidly and the chains will be short. We can consider here simply that insertion requires a fetch, and a rewrite. In open addressing there is a probability based on the density that the inserted record does not fit into the block found. Now

$$\begin{aligned} T_I = {} & c + s + r + btt \\ & + p_{1u}\,t_{overflow} + T_{RW} + \frac{n}{m}(s + r + btt + T_{RW}) \end{aligned} \qquad \langle\text{large buckets}\rangle\ 3(4)\text{-}83$$

Use of a separate overflow area would affect the last term, as seen in Eq. 3-82. The use of chaining affects mainly the value of c.

The considerations leading to Eqs. 3-81 and 3-82 apply also if we wish to optimize the record placement into blocks of the overflow area. Here the simple procedure of Eq. 3-80 is not adequate and the chain will have to be followed until a suitable block is encountered.

When the combination of open addressing, single-slot buckets, and no chaining is used, the entire set of colliding and all the associated clusters (see Fig. 3-22) have to be passed to find a free space. No satisfactory formulation for the linear searching required in this case is known to the author. A low density n/m is important to keep p_{1u} reasonable. Knuth[73S] suggests that

$$T_I \approx \frac{m}{m-n}(s + r + btt) \qquad \langle\text{open addressing, one slot buckets}\rangle\ 3(4)\text{-}84$$

when rerandomization instead of sequential search is used. A special technique for inserting a batch of insertions is given in the discussion leading to Eq. 3-89.

3(4)-2-5 Update Record in a Hashed File

The process of updating the record consists of finding the record and rewriting it into the original slot, so that

$$T_U = T_F + T_{RW} \qquad 3(4)\text{-}85$$

When the key changes, a deletion and a new write operation has to be performed.

Deletion requires the same effort when records are not chained. Depending on the method used for linkage, a tombstone may have to be set. If chaining is used, and it is desirable to set the slot to `Empty` to make it available for another collision set, then the preceding record has to be rewritten with the linkage pointer obtained from the deleted record.

3(4)-2-6 Read Entire Hashed File

An exhaustive search for a file using a randomizing key-to-address translation can be done only by searching through the entire space allocated to the file, since the discrete transformations from a sparse set of key values do not allow serial reading. The search is costlier, since the file space also contains empty and deleted slots. The area used by the overflows will also have to be checked. A separate overflow area will be dense, except for deletions, but

$$T_X \approx (m + o)\frac{R + W}{t'} \qquad 3(4)\text{-}86$$

In practice hashed files are rarely used when reading of the entire file is an expected operation.

3(4)-2-7 Reorganization of a Hashed File

Reorganizations are mainly required if the total number of records to be kept has grown and no provisions were made for extension of file space; linear hashing was presented in Sec. 3-1 as a method to accomodate growth. Growth causes the density n/m to exceed the design goals. Reorganization allocates a larger space to the primary area for the file; the randomizing procedure has to be rewritten or adjusted. All data have to be copied using an exhaustive search and reloaded into new slots. The terms T_X and T_{Load} defined below should be estimated using the old value of m for T_X and the new value of m for T_{Load}.

Reorganization is also beneficial when many deletions have occurred so that search paths have become tangled and long. For large collision sets which are not chained, this effect can reduce performance sufficiently to make a reorganization warranted.

If the file cannot be removed from service for a long time, i.e., a high *availability* is desired, an incremental reorganization can reorganize a single cluster (defined as the space between two empty slots) at a time. Otherwise a reorganization will read and reload the entire file

$$T_Y = T_X + T_{Load} \qquad 3(4)\text{-}87$$

where T_{Load} is the loading time discussed below.

LOADING A HASHED FILE The most obvious way to load or reload a hashed file is to rewrite the records one by one into the new space; then

$$T_{Load} = n\,T_I(n) \qquad 3(4)\text{-}88$$

with a continuously changing value of $T_I(n)$ as n/m increases. Initially there should be scarcely any collisions. The evaluation with Fig. 3-19 was based on loading the file. A first-order estimate to compute the average T_I can be based on a density of $\frac{3}{4}n/m$. Loading still requires much effort; every record will require at least one random block access, and if open addressing is used, the new file will develop clusters as it fills.

A two-pass procedure can be used to reduce the effect of clusters. In the first pass records are stored only if they have primary slots; any colliding records are placed into the remaining empty slots of the file during a second pass.

A major reduction in loading cost can be accomplished by sorting the file to be loaded into the hashed file sequence, i.e., by computed address. To do this, the key-to-address transformation is applied to all the record keys of the records to be loaded into the file, and the address obtained is attached to the records. The records are then sorted based on the attached address. Equation 3-11 indicated the speed of a sort operation. The sorted file is then copied without the addresses into the space for the hashed file; slots which do not have a matching address are skipped, and colliding records are placed into successor slots. This writing time is equal to the sequential exhaustive read time, so that

$$T_{Load} = c + T_{sort} + T_X \qquad 3(4)\text{-}89$$

The advantage of a substantially reduced loading time is that reorganization can become a valid alternative to more complex schemes used to combat clustering and to manage deletions in a dynamic environment.

Using the technique of sorting by address, leading to Eq. 3-89, to reload the hashed file space permits the entire reorganization to be performed at a cost of

$$T_Y = c + 2\,T_X + T_{sort} \qquad 3(4)\text{-}90$$

BATCH INSERTION The reloading method can also be used to add a batch of insertions into the file. The batch would have to be quite large. If we use n_I to denote the size of the batch, this insertion method would be profitable only if $n_I > (T_Y/T_I)$.