¡le

Chapter conversions: Fig. –¿ Fig. change references outside of chapter. Chap. –¿ Chap. change references outside of chapter. Sec. –¿ Sec. change references outside of chapter.

DATABASE DESIGN: FILE ORGANIZATION, Gio Wiederhold

material for new CHAPTER 6,7,8, file CHAP4.TEX June 22, 2000

Chapter 4

Hybrid File Organizations

Even the question whether one did right to let WESCACthus rule him, only WESCACcould reasonably be asked. It was at once the life and death of studentdom; its food was the entire wealth of the college, the whole larder of accumulated lore; in return it disgorged masses of new matter – more alas than its subjects ever could digest . . .

John Barth
Giles Goat-Boy, vol.1, first reel

6-0 INTRODUCTION

In this chapter we illustrate some of the varieties of file designs that can be obtained by combining the basic methods described in the previous chapter. Such combinations can satisfy requirements that are not fulfilled by the basic methods.

We will show in Sec. 4-1 some simple structures which save space or retrieval time when the data or problems have a well-defined structure. In Sec. 4-2 index construction techniques will be presented, leading to an example of a complex indexed-sequential file in Sec. 4-3. In Sec. 4-4 tree-structure files with data within an indexlike structure are presented, leading to hierarchical files with an example in Sec. 4-5. Sections 4-6 and 4-7 discuss combination-based or direct access and ring structures. The final two sections discuss files based on virtual storage and phantom files.

The information presented in this chapter is more diverse than the contents of Chap. 3. Techniques described in earlier sections will often be applied again in combination with other file methods. Many techniques improve the data-retrieval capability by adding redundancy but this increases the amount of space required and makes updates more difficult. The aggregate performance of a system depends on the balance of update and retrieval usage, and will be presented in Chap. 5.

SUMMARY OF PREVIOUS RESULTS We will review briefly the six basic file designs which were discussed in detail in Chap. 3.

The *pile file* provides a basic unstructured organization which is flexible, uses space well when the stored data vary in size and structured, is easy to update, quite awkward for fetching specific records, and amenable to exhaustive searches.

The *sequential file*, containing a collection of ordered fixed records, is inflexible, efficient for the storage of well-structured data, difficult to update, awkward for finding single records, and very suitable for efficient exhaustive processing.

The *indexed-sequential file* adds a single index to the sequential organization. The result is a file which can be efficiently searched and updated according to one attribute, but is otherwise still fairly inflexible. The file is reasonably efficient for the storage of well-structured data, and suitable for exhaustive processing.

The *multi-indexed file* removes the constraint on sequentiality and allows many search attributes. It is quite flexible, reasonably efficient for data storage, permits very complex updates at some cost, allows convenient access to specific records but is awkward for exhaustive processes. Much storage may be occupied by the indexes.

The *direct file* is quite inflexible because of its mapping requirements, has some storage overhead, allows updates at some cost in complexity but provides efficient record retrieval according to a single dimension. Serial and exhaustive searches are, in general, impossible.

The *multiring file* provides for a number of record types with many interconnections. Redundant and empty fields are reduced, but some space is required for the linkage pointers. Updates may be performed at a moderate cost. The ring organization provides flexible, but not always fast, access to single records and allows great flexibility for exhaustive or subset searches.

Table 4-1 Grades of Performance for the Six Basic File Methods

	Space		Update		Retrieval		
	Attributes		Record size				
File method	Variable	Fixed	Equal	Greater	Single record	Subset	Exhaustive
Pile	*A*	*E*	*A*	*A*	*E*	*D*	*B*
Sequential	*F*	*A*	*D*	*F*	*F*	*D*	*A*
Indexed-sequential	*F*	*B*	*B*	*D*	*B*	*D*	*B*
Multi-indexed	*B*	*C*	*C*	*C*	*A*	*B*	*D*
Direct	*F*	*B*	*B*	*F*	*A*	*F*	*E*
Multiring	*C*	*B*	*D*	*D*	*C*	*A*	*B*

A = excellent, well suited to this purpose
B = good
C = adequate
D = requires some extra effort
E = possible with extreme effort
F = not reasonable for this purpose

Only the multi-indexed file and the multiring files provide the capability for access to data according to more than one dimension. These two methods provide the basic building blocks for file designs providing multiattribute access whenever there are many records. Table 4-1 above lists these conclusions in a summary form.

REASONS FOR COMBINING THE BASIC METHODS It is obvious that there are frequently requirements that are not well supported by any of the organization types described up to this point. Typical requirements may include the need to collect variable-length records while preserving sequentiality, or the desire to take advantage of the speed of direct access while requiring retrieval according to more than one attribute.

Not all possible solutions will or can be listed. Many of the methods described here have been implemented in practice; others are shown in order to provide a logical continuity or to suggest alternate approaches. For most of these methods, no comparative formulas will be provided. We hope that the reader, using the approaches of Chap. 3, will be able to develop formulas to estimate any file organization method that is encountered in this chapter or in actual practice.

REVIEW OF TERMS In this section we frequently refer to the *search argument* and to the *key* and *goal* portion of the record. These terms are used particularly when files are used for data retrieval. The term search argument denotes the attribute type and value known when a record is to be retrieved. The *key portion of the record* is the field to be matched to the search argument. The fields of the record to be retrieved by the fetch process are known by the term *goal*. In an example, the search argument may be "`social security number=134-51-4717`"; the key in the specified attribute field of the record contains the matching value "`134-51-4717`", and the goal portion of that record contains the `name` and `salary` for the employee.

The key is generally intended to identify one record uniquely, so that there is a simple functional dependence from key to record. A key may comprise multiple attribute fields of the record in order to achieve a unique identification. An example of the need for multiple fields might be the title of a book, which may not identify a book uniquely without specification of author or publisher and date of publication.

It should be understood that another instance of a retrieval request can have a different composition. Then the search argument may comprise a different set of attributes; these may have been previously part of the goal, and the goal can include the former key values.

4-1 SIMPLE FILES

We will take a step backward in this section and present some file mechanisms which are simpler than any of the six presented in Chap. 3. Such simple files are sometimes the only ones provided on small computers or on systems which were not designed to provide data-processing services. The simple files presented are in general not adequate for databases. A better file organization can be constructed on top of some of them if a database is needed and no alternative file support exists. The extra software layer will typically diminish performance but is required to provide database software without excessive code to deal with file inadequacies.

We deal first with some simple index types. If the attribute values used for indexing are restricted to integers, a simplified indexing scheme can be used.

4-1-1 Immediate Access

No structure to assist access was provided in the earliest direct files. An index number given by the user specifies a relative record position. Translation of a key to an index number, density of the file, and avoidance of collisions is left to the user of the file. Index numbers are integers of a limited range. The specifications for many FORTRAN implementations of *direct* access input and output limit themselves to this level of service. FORTRAN statements defined for this purpose include

```
DEFINE FILE 17(n,R)
...
WRITE (17 'key) data
```

The number 17 following the keywords `DEFINE FILE` is the *name* for the file; FORTRAN restricts even the file names to integers. The parameter `n` specifies the maximum size of the file and `R` the maximum record length. The values given for `n`, `R`, and `key` must be integers, with `key`≤`n`. If the `data` field exceeds `R`, the record written will be truncated; if it is less, the unused space will be filled with blanks, or its contents left unpredictable.

In IBM FORTRAN systems, two additional parameters are provided:

```
DEFINE FILE 17(n,R,code,nextkey)
```

The `code` specifies usage; `U` is for an unformatted file to be stored while a `code` of `E` indicates formatting for input-output. The variable `nextkey` specifies an output parameter, which is set to the key value of the next existing record in the file in order to simplify serial processing.

To be expanded to 243 pt for Figure space
HERE

Figure 4-1 File using an integer key with all record spaces allocated.

PREALLOCATED SPACE If, in the implementation, the entire space for all the expected records has to be allocated in advance, the simplest direct organization can be used. The only transformation τ required by the system to locate a record given a key is the following:

```
record_address = filebegin + (key-1) * R
```

as indicated in Fig. 4-1.

If data are of variable lengths, then the user has also to program blocking routines. In essence, the file system provides the user with a software-sized block, calling it a *record*. Some systems add the further restriction that the size of this record has to be equal to the system block size. Now most user records will require blocking routines if space utilization is a concern. Such systems frequently are found in microcomputers, and in simple timesharing systems.

The required system software is of course minimal, and also quite reliable. A subsystem developer may not want more than these basics from the system. Users may choose to add the logic described for indexed-sequential, B-trees, or direct files to their programs if their requirements make this addition necessary. On larger systems immediate access may be available as a optional building block for file designers unhappy with standard file organization methods.

4-1-2 Simple Indexes

The addition of an index to such a file provides flexibility of space allocation, so that the file can be incrementally or sparsely filled. Used record spaces can be identified by a marker in the index to provide the capability to omit, insert, and update records. Figure 4-2 shows such an arrangement.

A distinction now can be made between `WRITE` and `REWRITE` operations to prevent loss of previously stored data due to errors in the calculation of the key value. A function `FREE` can be provided to allow the user to test the status of a record space.

```
      IF ( FREE(key) )  GO TO 20
C     A record with this key exists already in the file
```

Space for records is taken out of blocks as required. The entire index is predefined, and allocated at least to an extent that the highest existing record is included. Where records do not exist, the index entry value is `Null`. The actual attribute value or key does not appear in the index since the proper index entry is found by the computation

```
index_address = indexbegin + (key-1) * P
```

which is similar to the one used for the immediate access to the record without indexes. In fact, we have for the index a structure identical to that for the file in Fig. 4-1. The storage overhead is reduced for nondense files, since we expect that the index entry, only a pointer of size P, is much smaller than the actual record.

In this organization the indexes are relatively simple to maintain. Data records are easily read and written. Updating of records, however, can involve the invalidation of record spaces so that a list of the record spaces that are available for reuse has to be maintained.

It is easy to permit the storage of variable-length records, since the access algorithm does not depend on a fixed value of R. A record-marking scheme as described in Sec. 2-2-4 will be employed. However, if deletion or updating of records of varying size is to be allowed, space management becomes more complex. Compression and expansion of the file has to be provided, or a periodic reorganization will be required. Since only one index is maintained, few problems occur when a record has to be moved to another position during an update or a reorganization. In practice most systems of this type do not allow for variable-length records nor do they warn the user that the effects are unpredictable.

4-1-3 Indexing of Buckets

The index organization shown above also can be used to support more complex environments. Since this mechanism can provide facilities for variable-length records, it also can be used to store sets or *buckets* of small records. The concept of using buckets allows the use of variable-length records in most file-organization methods.

Since file storage is organized in terms of blocks, these buckets are defined again so that they match block sizes as effectively as possible. Not all the records in a key range for a bucket need to be present. Records can be packed and searched sequentially in the block or area assigned to the bucket. For example, the records numbered 1 to 60 could be assigned to a bucket using the first index entry, 61 to 120 to the second entry, and so on. The buckets accommodate defined partitions of the index range. If many long records are placed into a bucket, it may overflow. This type of file is hence a hybrid of features found previously in the direct and the indexed file organizations.

In these fixed indexes the anchor points always have the same source key values, while in indexed files the anchor keys were those of entries appearing at the file system partitions. The lowest level index needs to contain only one entry per bucket, so that the index size is again reduced. The proper allocation to buckets depends on the expected average record length, the block length, and the desired performance ratios between file density, sequential access, random access, and updating of the file.

An indexed bucket organization can be very useful for files consisting of program or data text. The records here are small and of variable length, but the variation between typical pages of text is not very great. Both fast sequential access, for processing and searching, as well as reasonable access to specific lines to be edited are required. Keeping the density low provides all these features.

A bucket which contains a page of text can be expected to fill several blocks. These blocks are exclusively assigned to an index entry. Multiple blocks for one index entry can be chained sequentially from the initial block referenced by the index entry. Figure 4-3 sketches this organization. This file now can be read sequentially with a low overhead.

Space will remain unused in the last block of a sequence of blocks containing data for one bucket. Relatively fewer blocks will be filled incompletely if the buckets use chains consisting of multiple blocks. Better use of space causes increased fetch processing time, since the number of accesses to locate a record increases for many of the records. An insertion into a long chain can be particularly expensive. Careful allocation of continuation blocks to increase locality will reduce the frequency of seeks and also the expected rotational latency incurred when fetching or inserting.

To be expanded to 234 pt for Figure space

HERE

Figure 4-3 Bucket indexes with linked overflow blocks.

If we denote the chain length in terms of blocks as bch we expect that

$$T_F = (x + \frac{bch + 1}{2})(s + r + btt) \qquad \text{4-1}$$

Here x is again the depth of the index and $s + r + btt$ the components of the random access time. If record sizes and record quantities per bucket are random, half a block per chain will be wasted. Thus, the waste per record in the chain is

$$W = \frac{\frac{1}{2}B}{bch - \frac{1}{2}} \frac{1}{Bfr} = \frac{R}{2bch - 1} \qquad \text{4-2}$$

The index space SI is preallocated to contain m bucket pointers of size P. The value of $m = n_{max}/(bch\,Bfr)$, typically $m < n$. The space used per record R' is then

$$R' = \frac{SI}{n} + R + \frac{R}{2bch - 1} \approx P + R + \frac{R}{2bch - 1} \qquad \text{4-3}$$

where R is the actual record size. The storage required for the index, SI, is reduced when the bucket size and hence the value of bch increases.

When such files are used for the storage of textual data, the line numbers can provide the basis for block address computation. The index entry is found by computing the quotient of the line number and the estimate of lines that can be stored in the block or blocks expected per bucket, as shown in the example below:

Given is $B = 1000$ and $bch = 2$. The expected bucket loading capacity is then 1500 bytes. The text averages 75 characters per line; hence one bucket should be assigned for each set of 20 lines.

4-1-4 Transposed Files

In some applications there is a frequent requirement to access all values of a given attribute. An example is the selection of the *average person*, where first the sum of all values of a given attribute has to be computed before a specific record can be selected.

If the file is relatively small and storage is affordable, the creation of a duplicate data file, in transposed form, may be the easiest solution to the problem of minimizing exhaustive search times for specific attributes. Only one record is fetched per attribute from a transposed file, and this record contains the values of this attribute for all entities, e.g., persons, in the file.

A transposed file contains one record per transposed attribute type from the main file, and each record has a number of data values which is equal to the number of records in the main file as shown in Fig. 4-4. If all attributes are transposed, the transposed file will be equal in size to the main file.

A transposed file is oriented to aid selection or evaluation of data according to a dimension which is orthogonal to the data entry or update dimension. Use of transposition presents opportunities for the discovery of trends or functional dependencies among attributes which are otherwise not obvious. Many research-oriented data processing tasks will take transposed data and group, select, and correlate the values found. When transposed files are used for statistical purposes, the updating or re-creation of the transposed file may be done only periodically. This process is analogous to a file reorganization.

The potentially very long records, $R = nV$ for one attribute, can easily exceed the limits of available analysis programs in terms of storage space. The processing time required for such long records also will become more significant. The analysis of very large quantities of data will also require consideration of the numerical stability or accuracy of the analysis algorithm used. The assumption made in most other examples that the computational time can be neglected is not valid when processing records from transposed files.

A record of a transposed file shows much similarity with the lowest level of an exhaustive index. A transposed file, however, stands more on its own; it does not need to contain pointers from the transposed file to the main file. It may be convenient to use a transposed file as input to a program which generates indexes to the main file. If the record positions of the main file cannot be computed, the transposed file can contain one record with values of all the original source record positions.

TRANSPOSITION OF A FILE The effort needed to transpose a large file with many attributes is large. The most obvious procedure is to construct one transposed record during one exhaustive read of the file. To transpose all of a file in this manner will require a time equal to $a \cdot T_X$ for the a passes over the file. If blocks for multiple transposed records (buf) can be built up in core at a time, only $\lceil a/buf \rceil$ passes over the file are required.

Floyd[74] describes two transposition algorithms which take on the order of $\lceil \log_2 w \rceil$ passes where w is the greater of n or a. The simpler algorithm generates records increasing gradually in size to n; the second algorithm is given as Example 4-1 and generates each transposed record as a sequence of pieces of size a and requires only three buffers of that size.

TRANSPOSITION PROGRAM The program shown as Example 4-1 on pages 176 and 177 transposes a source file using $\lceil n/a \rceil (2 + \lceil \log_2 a \rceil) a$ fetches and as many insertions. Most of the operations are not sequential and will require a seek. The benefit of this procedure will become substantial when $a/buf \gg 10$.

Table 4-1 Floyd's second transposition algorithm.

```
/* Program to transpose a source file of 'n'records of length 'a'.
    The input and output files are sequential.
    The program contains  three distinct sections:
        1. Copy groups of 'a'records to a work file, resequencing
      all but the first record of a group in inverse order.
      Values in these records are circularly shifted so that the
      first column elements form a diagonal.
      The last group is padded with zeroes.
        2. Record pairs within groups are selected, and within each
      pair alternate entries are transferred to rotate columns by
      1,2,4,8,.. using powers of two in log2(a) passes per group.
        3. When all groups are rearranged, the work file is copied,
      shifting each record to the final postion, to the output file.
  file_transpose: PROCEDURE(source,transposed,n,a);                    */
          DECLARE(source,transposed,work) FILE;
          DECLARE(spacein,spaceout,save)(a);
          DEFAULT RANGE(*) FIXED BINARY;  DECLARE(MOD,MIN)BUILTIN;
       /* The transposition proceeds in groups of "a" records */
          groups = (n-1)/a+1;  amin1 = a-1;  record_in = 0;
       /* Perform sections 1 and 2 for each group and leave re-  */
  g_loop: DO group = 1 TO groups;      /* sults on the work file */
       /* Copy and rearrange a group for permutation */
  loop_1:  DO out_rec_pos  =  a TO 1 BY -1;
           IF record_in<n  THEN READ FILE(source) INTO(spacein);
                           ELSE spacein = 0;
           record_in = record_in+1;  base_out = a*(group-1)+1;
       /* Shift records to align columns so that transposition
                                   can proceed in parallel blocks */
            CALL movearound(out_rec_pos);
            record_out = MOD(out_rec_pos, a) + base_out;
            WRITE FILE(work) KEYFROM(record_out) FROM(spaceout);
           END loop_1;
       /* Set up number of passes to permute one group */
           order = 1;  order2 = 2;
  loop_2:  DO WHILE( order<a );
       /* Permute records by order = 1,2,4,8, ... */
            CALL rearrange(base_out);
            order = order2;  order2 = order*2;
            END loop_2;
          END g_loop;
       /* Now copy the result from each transposed group to the */
  loop_3: DO shift = 0 TO amin1;                /* output file */
       /* Each record contains "groups" segments of length "a"*/
    a_loop: DO rec = shift+1 BY a TO base_out-1;
              READ FILE(work) KEY(rec) INTO(spacein);
              CALL movearound(shift);
              WRITE FILE(transposed) FROM(spaceout);
              END a_loop;
            END loop_3;

/*     Dick Karpinski helped with checkout and structuring */
  /* Subprocedures */
    /* Procedure to copy records, elements are circularly shifted */
   movearound: PROCEDURE(move);
            DO i = 1 TO a;    /* note that last (a) precedes 1 */
              j = MOD(i-move-1, a)+1;  spaceout(j) = spacein(i);
            END;
        END movearound;
```

```
 /* Subprocedure to pair alternate records */
rearrange: PROCEDURE(beg_rec);
 /* The record pairs exchanging data will be:
     pass=1: (last,1),(last-1,last),(last-2,last-1),...;
     pass=2: (last,2),(last-2,last),(last-4,last-2),...;
     pass=3: (last,4),(last-4,last),(last-8,last-4),...;etc. */
       rec_in, rec_save = beg_rec + order;
    /* Loop through all "a" records in the group */
 t_loop: DO transfer_count = 1 TO a;
           IF rec_in=rec_save
              THEN DO;  rec_out = rec_in-1;
    /* This record is saved to provide data for its partner */
                  READ FILE(work) KEY(rec_out) INTO(spaceout);
                  save = spaceout;  rec_save = rec_out;
                  END;
              ELSE DO;  rec_out = rec_in;
                  spaceout = spacein;
                  END;
           rec_in = MOD(rec_out+amin1-order, a) + beg_rec;
    /* Use saved record when it comes */
         IF rec_in=rec_save THEN spacein = save;
                ELSE READ FILE(work) KEY(rec_in) INTO(spacein);
    /* Shift data values from "spacein" to "spaceout" */
         CALL transfer;
         WRITE FILE(work) KEYFROM(rec_out) FROM(spaceout);
    /* Terminate when all records have been processed */
         END t_loop;
     END rearrange;
 /* Subprocedure to move alternate elements in records selected
                                     by the rearrange procedure */
transfer: PROCEDURE;
    /* Elements to be moved from "spacein" to "spaceout" are for:
        pass=1, order=1: 2,4,6,8,etc.
        pass=2, order=2: (3,4),(7,8),(11,12),etc.
        pass=3, order=4: (5,6,7,8),(13,14,15,16),etc. */
         DO i = order BY order2 TO amin1;
          limit = MIN(i+order, a);
         DO j = i+1 TO limit;  spaceout(j) = spacein(j);
         END;
     END transfer;
/* This completes the file transposition program */
   END file_transpose;
```

4-2 MULTILEVEL INDEX STRUCTURES

Some issues associated with indexed files were not considered in the previous chapter. One of these is due to the complexity associated with long and variable-length attribute values in the key. The next section will deal with abbreviation of index keys, and Sec. 4-2-2 will deal with alternate index structures which can also ameliorate the problems of long keys. Serial processing by means of an index is considered in Sec. 4-2-3. There is an interaction of the processing of keys and the optimal block size; the trade-off is evaluated in Sec. 4-2-4. Finally, Sec. 4-2-5 presents the interesting issues of retrieval of records by multiple attributes using indexes.

4-2-1 Key Abbreviation

The keys used to identify data records can be very long. Examples of long keys which can be encountered are names of individuals, technical terms, names of diseases, and addresses. Long keys, when kept in various levels of an index as well as in the goal records themselves, not only waste space but reduce the number of entries that can be kept in one index block. The reduction of fanout will require more levels of indexing and hence increase the processing time significantly.

With long key fields and a large index, there will be long sequences with identical high-order parts of the key field. On the other hand, the low-order parts of the keys may not aid at all in the discrimination process between records. The key field can be shortened by judiciously abbreviating the keys. Abbreviation of long keys may be made *externally* before records are put into a system, or may be performed *internally*, completely within the file system. Some key abbreviation algorithms are designed to be convenient in either case.

EXTERNAL KEY ABBREVIATION In order to introduce the topic of key abbreviation, we will present a scheme oriented toward manual use. This scheme also randomizes the key values, so that buckets used to store the records are evenly filled. The abbreviation has hence also a hashing objective.

Hashing techniques are difficult for human interaction; the computed addresses tend to be unintelligible. In Fig. 4-5 we have the example of a mailing label, with a code which is intended to be understandable by clerks who have to deal with subscription renewals and delivery complaints of newspapers and magazines. The file is accessed based on the coded file address in the top line. We see that the zip code, the characteristics of the name, and the street address are used to obtain an abbreviated key. This method is similar to a sequence-maintaining hashing technique but can also be carried out manually. The method used does not guarantee uniqueness unless a sequence digit is added.

A mailing label code. The underlined parts of the address are included in the code. Counts are given for the dotted characters. A sequence digit 1 is appended.

```
60282HGS5155POT31

Edgar R. B. Hagstrohm
            - --.....
155 Proteus Park
--- - --...
Chicago, Ill 60282
             -----
```

1pt Figure space

Figure 4-5 Abbreviated address key.

check fig height with remark

In many European countries identifiers for individuals are constructed using similar techniques. When such an abbreviated key is constructed at a later time, the sequence digit is not known, so that no unique transformation can be made. In that case multiple records have to be retrieved and checked using further information, in a manner similar to the handling of collisions in a direct file organization. A trade-off will have to be made between the length of the abbreviated key and the number of key collisions.

INTERNAL KEY ABBREVIATION Abbreviation techniques are often essential within the file system. Adequate fanout requires that we deal well with potentially long keys. For character-string fields abbreviation by deletion of segments of low-order blanks is common. Keys can be much more abbreviated, however, since keys should not need more bits than needed to discriminate among the n items in the file. A theoretical minimum is a length of $\log_2 n$ bits, and within one index block only $\log_2 y$ bits should be needed for the key.

When processing data automatically, it is necessary that only one record be selected in response to a fetch request. In the technique used externally a sequence digit was used to assure uniqueness. A file system has to abbreviate the key so that any record, given a unique key, say the full `name and address` of a subscriber, can be uniquely identified by that key alone. This will also avoid collisions. Automatic abbreviation is feasible when the keys appear in a sorted sequence. The portions of the key which do not discriminate between adjoining keys can then be deleted.

In order to benefit from internal abbreviation the index has to be able to handle variable-length keys. The record marks which are needed to indicate key and record boundaries reduce the effect of the abbreviation.

VARIABLE-LENGTH KEYS Variable-length keys can occur naturally or be due to an abbreviation algorithm. Naturally occurring variable-length keys, for instance, people's names, should in any case be allocated to variable-length fields, since a fixed allocation of adequate space for the longest key may be prohibitive. Key abbreviation is hence not costlier and is always desirable for variable-length keys. Figure 4-6 shows a problem due to a fixed-length abbreviation: in which volume would you find a "Canoe"?

To be expanded to 468 pt for Figure space
HERE

Figure 4-7 Part-number index with block-based key abbreviation.

Figure 4-7 illustrates opportunities for abbreviation of long keys. The index shown uses the highest value of the key as the block anchor point. The keys in our examples are numeric and letters indicate block addresses. We will show how to delete redundant high-order and low-order segments of keys.

HIGH-ORDER KEY ABBREVIATION If it is known from a higher level index that all index entries in a block refer to keys within a certain range, the high-order digits will be identical and this *high-order segment* can be deleted. In Fig. 4-7 the two leading digits of entries in block C of the second-level index were redundant, and were deleted from the individual entries. In block D only one digit could be deleted, but in both cases the number of entries per block and the fanout ratio y has increased.

It is desirable to keep in a header portion of each index block the value of the high-order segment to allow reconstruction of the key from data within the index block. Key abbreviations may be also applied to individual entries within blocks, as is shown in Example 4-2 for blocks B and D. Now the entries are also variable within a block.

¡¡UNDERFULL ¿¿

Table 4-2 High-order key abbreviations

B :	000+13=a,+29=b,00+407=c, +704=d,0+2050=e,10+094=f ,+151=g;	D :	118+00=p,+10=q,11900=r,1 2+130=s,1220+1=t,+5=u,+9 =v,12+210=w,+384=x;

The stored segments are separated by markers {+ = , ;} as indicated. The high-order segment is repeated in front of a + symbol whenever the truncation changes. Some analysis is needed to make the truncation optimal for the entire block.

If the key is of known length, the value and length of high-order segments omitted may be implied by the length of the remainder, as shown in Example 4-3.

Table 4-3 High-order abbreviation for keys of known length

B :	00013=a,29=b,407=c,704=d ,2050=e,10094=f,151=g;	D :	11800=p,10=q,900=r,2130= s,201=t,5=u,9=v,10=w,384 =x;

LOW-ORDER KEY ABBREVIATION Similarly, the right-hand side of keys may frequently be shortened. Indexes where the attribute is a potentially long string of characters, such as a name, sometimes arbitrarily limit fields to a fixed number of characters. When keys are poorly distributed, records may no longer have unique keys in the index.

This method can be seen in the volume headings of an encyclopedia as shown in Fig. 4-6. Three characters here did not provide enough discrimination between Volumes 1 and 2, 2 and 3, 4 and 5, and 15 and 16.

Instead of arbitrary limits, the contents of an index block can be inspected to determine how many low-order digits can be safely deleted. In the earlier example (Fig. 4-7), the two low-order digits of the first index level (block A) can be deleted without loss of discrimination. For insertion, omitted digits are taken to be nines. ¡¡??¿¿ The third character is required only because of the density of entries in the range between 13 000 and 14 000.

The alternate version of block D of Fig. 4-7 is shown in Example 4-4 with low-order abbreviation applied to the entries. The dash (-) indicates the boundary for the low-order segment.

Low order key abbreviation has the disadvantage that the actual data file has to be accessed in order to determine whether a record corresponding to the full key exists. A search for a nonexistent record, via a record-anchored but abbreviated index, will require one more access. Without the abbreviation, the lack of the record is recognized when the index entries do not yield a match. When using block anchors, the file has to be accessed in either case.

Both types of abbreviation can be applied at the same time. The result is not easy to to decode, but combined abbreviation is effective. When there is likely to be little high-order abbreviation – there are few records relative to the key space – the low-order abbreviation is effective, and vice versa.

In Example 4-4.5, also based on block D of Fig. 4-7 both abbreviations are applied to the entries. The point (.) symbol defines segments; the dash (-) indicates an omitted segment. In these small examples the effect is less obvious than in practice, where index blocks are long. Also, instead of using marker characters as `= , - . ;` in practice count fields are being used. An actual implementation is described in Sec. 4-3.

Table 4-4.5 Combined high- and low-order abbreviation for keys

```
D : 118.0=p,-1=q,119=r,121=s
    1220.1=t,-5=u,-9=v,12.21
    =w,-3=x;
```

REPEATING KEYS In indexes where the keys are not *unique*, so that more than one record for a given key can exist, it can be profitable to avoid restating the key. With the flexible key organization obtained with abbreviated keys, such a facility is easy to implement, as shown in Example 4-5.

Table 4-5 Abbreviation of repeating keys

```
E : 1241-1=y&z,-2=...
```

Two records, stored at `y` and `z`, both have a key value of `12411`.

ABBREVIATING POINTERS If the goal records occupy only a part of the file space, then the pointers to the goal record may be similarly abbreviated. The high-order segment of the pointer in any block can be significantly shortened, especially in the case where the indexed entries are used to point to a sequential file in index order. Similar benefits of increase in fanout ratio and reduction of levels of indexing can be expected.

USE OF ABBREVIATED INDEX ENTRIES Abbreviation of keys is effective when keys are relatively long and files are big. Often one level of index can be saved. When there is no need to access data serially, a key-to-address translation algorithm can perform the same service without the complexity and computational overhead of variable-length keys. Since the keys are denser in the record-anchored indexes associated with multi-indexed files, key abbreviations can play an important role there. The abbreviation of pointers complements abbreviation of the keys and can lead to further savings.

4-2-2 Index Blocks Structured as Trees

The abbreviated keys require considerably more processing time c than would be required if a simpler index entry format were used. Measurements have allocated 14% of total database CPU usage to abbreviation and expansion of keys. If the indexes are kept on high speed disks, the computation time may exceed the time available because of the access delays. Since index blocks are not accessed sequentially this does not affect the assumptions made for the bulk transfer rate made in Sec. 2-3-4, but the heavy demand on CPU resources is significant.

In a multiprogrammed environment the CPU is used at the same time for other processes and may be too valuable to spend on excessive index abbreviation. Processing all the keys in sequential order requires accessing about $cix = \frac{1}{2}y$ entries for comparison with the search argument. The approaches described in this section reduce cix by replacing the sequential key structure of the entries in the current index block with a tree. The use of rapid search in an index block regains computational time lost because of abbreviation. We recall that the motivation for abbreviation is to increase the fanout ratio, which reduces the number of seek operations in the tree as well as the index space.

This section presents three alternate tree structures for indexes. When entries are of fixed size and not abbreviated, a conventional binary search or probing, as shown in Sec. 3-2-3, also provide rapid access to entries within an index block.

JUMP INDEX SEARCH A simple scheme groups the y index entries within one block into a number of sections to provide a second level. The number of sections is chosen to be about equal to $\sqrt{y}$. An initial search pass *jumps* from section to section and compares one entry in each section. A subsequent pass searches within the proper section through the $y/\sqrt{y} = \sqrt{y}$ entries found in a section.

The expected number of comparisons of sections and entries is $\frac{1}{2}\sqrt{y}$ each, so that the total number of comparisons cix expected is

$$cix = 2\,(\tfrac{1}{2}\sqrt{y}) = \sqrt{y} \qquad \langle\text{jump search}\rangle\ 4\text{-}4$$

which is the minimum obtainable with two passes and linear searching. The number of comparisons $\sqrt{y}$ to find an entry is within a factor of 2 of the optimum, a binary search, for typical values of y. This additional pseudo-level does not require redundant entries of the key or pointers, only a means to locate the sections.

If the records are of fixed length the location of the sections in the index block is computable. Otherwise $\sqrt{y}$ section pointers can be used in each index block to define a two-level tree. These section pointers have to be distinguishable so that they will not interfere with sequential processing of index entries. An implementation is shown in Sec. 4-3-3 and Fig. 4-18.

A BINARY ARRANGED INDEX TREE We presented the algorithm for binary search within a sequential file in Sec. 3-2-1, Fig. 3-5. Since entries in an index block are always sequential, a binary search can be used here too if the entries are of fixed size. For entries of arbitrary length a method which uses explicit successor pointers can permit a binary search to be used.

Figure 4-8 shows block D from the earlier example in Fig. 4-7. The entries are arranged by level, so that the search begins at the middle value. Two fields per entry direct the search algorithm to the next index entry to effect a binary partitioning. In this example, the maximum number of comparisons is four, and the expected number is $25/9 = 2.78$. In a sequential search, $(1+9)/2 = 5$ comparisons would be expected. In general, the tree search will require on the order of

$$cix = \log_2 y \qquad \langle\text{binary search}\rangle\ 4\text{-}5$$

comparisons. This is significant when there are many entries per index block.

The index shown has grown in size because of the pointers which relate the index entries to the tree structure. Of these pointer fields, $y+1$ will be empty. In the example the comparison is based on the whole value of the key and some opportunities of high-order key abbreviation are lost.

```
          Example:
find match to argument = 1 2205
  begin at entry 1:
    argument ¿ key  entry 6;
  continue at entry 6;
    argument ¡ key  entry 8;
  continue at entry 8;
    argument = key  Found.
```

A TRIE An alternate arrangement into a binary tree can be achieved by basing the partitioning decision on individual digits of the key. A base comparison value partitions the key digit value into two subsets. The method produces also an abbreviation of the key, since entries are represented by only one digit. The digits of the key to be used are chosen to provide the best partition of the search space, but the number of comparisons will generally be greater than the number for a binary search, since we cannot guarantee that the space will be optimally partitioned. Exact matches, which would lead directly to data, are also excluded now. The destination after a comparison can be either a goal record or another comparison entry. Files based on this idea are named *tries* (pronounced "try-s").

Figure 4-9 shows an index trie for the same index block D. The first digit (which anyhow could be omitted from the block) will of course not play a role, but even the second digit never provided a useful discrimination. The expected number of comparisons in this example is $29/9 = 3.22$, and in large indexes the improvement over sequential searching will again be significant. In this method we can also make very effective use of the pointer fields by allowing both index entry numbers (digits) and goal block addresses (letters).

These tries, because of the binary decisions made, are best implemented in practice by basing the comparison on a single bit, rather than on a digit. In this case no column with base comparison values has to be provided, since the conditions are always only "=0" or "=1".

The number of entries in a block of a trie leading to y records n_e is determined by the number of decision nodes $n_d = \lceil y/2 \rceil$ and their ancestors in the tree $n_d - 1$, and hence $n_e \approx y$. The space required for the key segment of a binary trie entry is composed of the bit position of the key and two internal references. An entry for a five-character (40-bit) key and up to 500 entries per block will require only $\lceil \log_2 40 \rceil + 2\lceil \log_2 500 \rceil = 24$ bits or 3 bytes, plus extra space for the external pointers. Trie entries are hence apt to be smaller than index entries, and there will be a corresponding increase in fanout.

To be expanded to 387 pt for Figure space
HERE

Figure 4-9 Index arranged as a trie.

Indexes based on trie structures as shown are suitable only for record-anchored indexes, with entries pointing to each individual record, since an exact match is implied. Intermediate search arguments will not provide a sensible result. If the digits used for the branching decisions are chosen only from left to right, the unpredictability of the match is avoided; and then the method might be used with an index containing entries which point to block-anchor points in the primary file or refer to lower-level index blocks.

4-2-3 Range Queries

The ability to perform a serial search has to be considered in the design of indexed files. A query may specify a *range* of values, say all `Employees having salaries between $20 000 and $30 000`, so that records musr be serially accessed. Serial access is also needed to access values that do not match exactly, say `An Employee with a salary > $50 000`.

File structures that deal well with Get-Next operations, as indexes, can range queries well. An index provides an ordering for the data records, so that it is possible to find pointers to successor records by scanning through the index blocks at level 1. Making sure that serial access works well affects the management of indexes.

Serial access is required for `Get_Next` and `Read_Exhaustive` operations. Long sequences of repeated `Get_Next` operations occur when a subset is to be retrieved. If the attribute that is indexed can have nonunique values, many entries may have to be retrieved for a single search argument, say,

`FIND Houses with 3 bedrooms`

An important requirement in many data-analysis tasks is to retrieve subsets specified by a *range*. An example of retrieval by a range of values is a query such as

`FIND Houses priced` $31\,000 \rightarrow 33\,000$

These are the types of queries for which serial access capability through an index is essential. We will first consider access within a block, then access between blocks, and finally the use of the pointers obtained to fetch goal data.

The pointers we obtain from the index entries are commonly referred to as *tuple identifiers* or TIDs; the source for this term is found in Chap. 7; see Table 7-1. We assume that the number and size of the TIDs for one query is such that we can collect them in core storage and sort them, prior to accessing any of the data blocks containing the desired results.

SERIAL ACCESS WITHIN AN INDEX BLOCK In the B-tree structure the entries on level 1 are sequential in a block and provide for easy collection of TIDs. Sequential entries with faster random fetch access are provided by the jump search method shown in Sec. 4-2-2; internal pointers are used to permit variable-length entries.

Neither the binary tree nor the trie presented in Sec. 4-2-2 provides for serial searching within the index blocks. A further linkage is required within each block if these indexes have to provide a set of TIDs in serial order by entry value.

SERIAL ACCESS BETWEEN INDEX BLOCKS When the entries in a block have been exhausted, the next index entry has to be found in a successor block. Since the next key value is not known, no fetch using a search argument equal to the next key is possible; the successor has to be found by navigation through the tree.

At the end of a lowest-level index block, a new search from the second level is required to find the next block. At each level the search requires an entry from the superior level to find the successor block when the end of a block is reached. This search requires a stack of information about the current status of index blocks at all levels.

A linkage of index blocks on the same level of the tree can avoid this pain. The solution using lower-level linkage of index blocks is shown in Fig. 4-10. The linkage pointers will have to be maintained with care. For a B-tree the pointers are reset when a lower level block is split. The pointers also identify the partner block to be checked when deletions reduce the number of entries in the block below a certain minimum.

ACCESSING DATA BLOCKS The pointers we obtain from processing of index entries can be effectively used to reduce the number of block accesses to the data file. By putting the pointers in order by block number we can assure that each block will be accessed only once; if possible, we will fetch multiple goal records out of one block. This technique becomes especially important when processing multi-attribute queries, as will be shown in Sec. 4-2-5.

We need to know the number of block fetches b_G required to retrieve n_G goal records in TID order from a large file of n_{file} records with Bfr records per block. The file uses hence $b_{file} = n_{file}/Bfr$ blocks. We will first consider the extreme cases. For small results ($n_G \ll n_{file}$) the value of $b_G = n_G$ since one block fetch will fetch one result record. For large numbers of n_G the value of $b_G \rightarrow b_{file}$, if the retrieval is carried out in TID order, since nearly every block will be accessed once; at $n_G > n_{file} - Bfr$ this is absolutely true.

In the important intermediate region, where many but much less than all records are selected from the goal file, some blocks will yield multiple records and others will not be needed at all. An estimate of the number of block fetches required to retrieve n_G records in TID order from a file of n records with Bfr records per block, for the cases $n_G \leq n_{file} - Bfr$, is computable as [Whang[81]]

$$b_G = \frac{n}{Bfr}\left(1 - \left(1 - \frac{Bfr}{n}\right)^{n_G} + \frac{Bfr}{n_G^2}\frac{n_G(n_G-1)}{2}\left(1 - \frac{Bfr}{n}\right)^{n_G-1} + \frac{1.5}{n_G^3 Bfr}\frac{n_G(n_G-1)(2n_G-1)}{6}\left(1 - \frac{Bfr}{n}\right)^{n_G-1}\right) \qquad \text{4-6}$$

This formula is a good approximation in the range of values seen in databases of the computationally difficult function required to produce an exact result [Yao[77]]. To a retrieval time computed on the basis of this technique the computation time c, required for sorting of the TIDs in core prior to file access, has to be added.

SERIAL LINKAGES IN ALTERNATE FILE ORGANIZATIONS In the case of the indexed-sequential file, which we examined in Sec. 3-4, serial access according to a single attribute was established through the location of the data records themselves and through a linkage for any overflow records. Files with multiple indexes establish multiple serial access paths within the indexes themselves. The optional extra linkage between blocks at the same level, presented above, increases update costs when new blocks are allocated but simplifies serial access. Serial access remains indirect, and data records are fetched according to the pointers found in index entries. The alternative, multiple direct linkages of data records themselves, is used in the design of the multiring file.

Since indexes are small, it is easier to achieve a high degree of locality for indexes than for data records. If few records are to be obtained per serial retrieval, multiple indexes with linkages will perform better than multiring structures. For retrieval of subsets the indirection of access makes use of an index costlier than following a ring.

4-2-4 The Interaction of Block Size and Index Processing

Up to this point we have used the block size as a given parameter in the design of indexes. In many cases this is a valid assumption, but it is instructive to see how changes in block size affect the performance of an index. The size of an index entry will be denoted below as Rix, and the number of index entries by nix. The positive effect of large block sizes is a high fanout ratio ($y = B/Rix$, from Eq. 3-26), which reduces the number of levels required. The number of levels x is again determined by $x = \log_y nix$ (Eq. 3-27).

The detrimental effects of large block sizes are the increased block transfer time $btt = B/t$, the increased computational effort c to locate the appropriate index entry in a block, and the cost of core-storage occupancy. The computation is a function of ther number of records which have to be searched in core cix and the time for a unit access and comparison c_1

Figure 4-11 shows the combined effect of these factors except the core-storage occupancy for a fairly large index. The fetch time within the index is computed as

$$T_{Fix} = \lceil \log_y n \rceil (cix\, c_1 + s + r + y\frac{Rix}{t}) \qquad 4\text{-}7$$

where $cix = \frac{1}{2}y$ ⟨linear search⟩
and $cix = \log_2 y$ ⟨binary search (Eq. 4-5)⟩
Fig. 4-11 does not show $cix = \sqrt{y}$ ⟨jump search (Eq. 4-4)⟩
which is in between, but close to the binary search.

The optimum length for index blocks in the case shown, a 2314-type disk, is 4000 bytes. The steps in the function are an effect of the discrete assignment of index levels. The lower, continuous curve represents use of optimal index processing; approaches were described in Sec. 4-2-2. The time needed to inspect one index entry c_1 is taken to be 40 microseconds(μs). Abbreviated entries will take longer to process and will raise the left side of the curve, especially if linear searches are made through the index blocks, but their use will also decrease the size of an average index entry, so that the same fanout will correspond to a smaller block size. Devices with a shorter seek time will favor shorter block sizes, while devices with a higher transfer rate will cause minimal values of the fetch time to occur with larger blocks.

To be expanded to 342 pt for Figure space
HERE

Figure 4-11 Fetch time versus fanout.

4-2-5 Partial-Match Retrieval

Indexes provide multiattribute retrieval, that is, access via any of several attributes. We also have to consider queries where several attributes are combined. Such a query is termed a *partial-match query*, since several of many (a) attributes of the record are to be matched. A partial match query with four attributes terms is shown in Example 4-6.

An important use of indexes is the reduction of search effort for partial-match queries. For each attribute {`style, price, bedrooms, location, ...`} indexes are available; each key value found will point to many records.

Table 4-6 Partial-Match Query

```
LIST houses WITH style     = "Colonial",
             AND  price     = 31 000 → 33 000,
             AND  bedrooms  = 3,
             AND  location  = "Wuthering Heights";
```

MERGING OF MULTIPLE INDEXES To restrict the retrieval to one or a few goal records, all arguments in the query are first processed against their indexes. Each index produces a list of identifiers (TIDs) for the candidate records. Example 4-5 showed an efficient encoding for multiple records per key.

These lists of TIDs are then merged according to the expression given by the query, using boolean functions to select the TIDs for those goal records which satisfy all query arguments. Fig. 4-12 sketches how 4 houses are found for Example 4-6.

Only the goal records needed for the response are eventually retrieved. The cost of processing a partial-match query can be estimated by expanding Eq. 3-55. We use a_Q indexes, merge the lists, and retrieve n_G goal records using b_G (Eq. 4-6) block accesses.

$$T_F(a_Q, n_G) = a_Q x(s + r + btt) + c + b_G(s + r + btt) \qquad 4\text{-}8$$

We made here the assumption that each subset of TIDs of the a_Q selected subsets was fetched from one block. If more index blocks are required some additional serial access costs are incurred; the linkage scheme for index blocks presented in Sec. 4-2-3 will minimize incremental costs.

The merging of TIDs is aided by sorting the TIDs initially, so that the retrieval of the blocks containing goal records is conveniently carried out in TID order. The value of b_G has an upper bound of n_G, but no block has to be accessed more than once. A good estimate of b_G as a function of n_G, n, and Bfr is given in Sec. 4-2-3 as Eq. 4-6.

In the next section we will discuss schemes that reduce the factor a_Q of Eq. 4-8 by creating additional indexes. If two or more attributes occur together in partial-match queries, it is possible to construct a single index for those specific attributes. Partial-match retrieval using other schemes is shown in Secs. 4-1-4, 4-6-4, and 4-7-5. Other aspects of the process of information retrieval are discussed in Chap. 10.

PARTIAL-MATCH INDEXES An index is not necessarily based on a single attribute, but can be based on entries that contain the catenation of several attribute values of a record. The number and size of the keys and hence the size of a combined index will be larger, so that the fanout may be less than the size of a single attribute index; but with effective index-abbreviation schemes, the increase will be moderate. In Fig. 4-12 it may be wise to combine `Style` and `Bedrooms`, since `Bedrooms` has a very low partitioning power. This combination is denoted `St&Be`. If it is desirable to have access capability via one index for *all* queries which specify the logical intersection of attributes, then, in the same example another eleven combinations are required in addition to the four single attribute indexes (Fig. 4-13a).

The number of combinations for a indexes will be $2^a - 1$. The number of indexes can be much less, since an index ordered on a catenation of multiple attribute values $v_1, \ldots, v_a$ is also ordered for attributes $v_1, \ldots, v_{a-1}$. An index for a attributes can hence also serve to answer queries for one of the combinations of $a-1$ attributes, one of the combinations of $a-2$ attributes, etc. An index for another combination of $a-1$ attributes can similarly serve simpler queries. An assignment of a minimal set of indexes is shown in Fig. 4-13b.

```
     St   St&Be   St&Be&Pr   St&Be&Pr&Lo
     Be   St&Pr   St&Be&Lo
     Pr   St&Lo   St&Pr&Lo
     Lo   Be&Pr   Be&Pr&Lo
          Be&Lo
            Pr&Lo
Null
(1)   4     6          4           1
          15 combinations
                 (a)
```

```
St   St&Be   St&Be&Pr   St&Be&Pr&Lo
Be   Be&Lo   Be&Lo&St
Pr   Pr&St   Pr&St&Lo
Lo   Lo&Pr   Lo&Pr&Be
       St&Lo
         Be&Pr
 0     2           3          1
        6 indexes
           (b)
```

Figure 4-13 Partial-match indexes. *(a)* Combinations of indexes for four attributes {St,Be,Pr,Lo}. *(b)* A minimal set of indexes for the four combinations.

In the minimal set the first index serves four combinations; three indexes serve three combinations each, and two combinations of two attributes remain. Only six indexes are required now. In general

$$nic(a) = \binom{a}{\lceil \frac{1}{2}a \rceil} = \frac{a!}{\lceil \frac{1}{2}a \rceil ! \lfloor \frac{1}{2}a \rfloor !} \qquad \text{4-9}$$

indexes will be required. This number is still quite large even for a modest set of attributes a, e.g., $nic(9) = 126$. It is possible to reduce the number of indexes further if some merge operations among indexes are acceptable.

Fewer combinations of attributes need be stored if for some queries a merge of two or more distinct TID lists is permissible. Then the a attributes will be partitioned into d sets with a_i attributes so that $\sum_{i=1}^{d} a_i = a$. The attributes will be partitioned to let frequent queries require only one set. The number of needed combinations reduces drastically. For $d = 2, a = 9, a_1 = 5, a_2 = 4$ the number of indexes becomes $nic(5) + nic(4) = 16$; for $d = 3, a = 9, a_1{=}a_2{=}a_3 = 3$ the number of indexes is $3\,nic(3) = 9$, the original number, although each index will carry more keys. The increased cost of updates will still discourage extensive use of combinatorial approaches for partial-match queries.

4-3 AN IMPLEMENTATION OF AN INDEXED-SEQUENTIAL FILE

A practical example of a comprehensive indexed-sequential file system is provided by IBM in their Virtual Storage Access Method (VSAM). The term *virtual* refers only to the fact that the programs used to implement VSAM depend on the virtual addressing provided by IBM's 370 type of computers. The approach is a hybrid of techniques presented in Secs. 3-3, 3-4, and 4-2.

We can describe the file design, using the terminology developed in the previous section, as follows:

> VSAM is an indexed-sequential file organization. The single index and the data file uses a B-tree algorithm. The index has a fixed number of levels, thus limiting the file size, albeit to a very large number of records. Anchor points are the keys with the highest value in a train of blocks. There is high- and low-order abbreviation of keys as well as abbreviation of pointers. The variable-length elements of the index are described by count fields which specify the original key length and the amount of low-order abbreviation. The lowest-level index is linked to maintain seriality.

In addition, a number of practical considerations have been given much attention in the design, and we will discuss some of these in detail.

It also is possible to use the structure of VSAM files while bypassing the processing procedures in order to provide record retrieval by *relative byte addressing* as presented in Sec. 2-3-3. The conversion of record keys to relative byte addresses is a user responsibility. Of interest here is the use of VSAM which corresponds to the normal use of full VSAM facilities to obtain indexed-sequential access to variable-length records.

When evaluating the performance of a VSAM file, the fact that the unit moved from disk to core consists of an entire train rather than just one block has to be considered. First the cost of moving a train of sequential blocks has to be estimated; then this result can be used with care in the relations based on variable-length unspanned blocking; Eqs. 2-20 and 2-21 provide the basic parameters.

4-3-1 Space Allocation

When a file is established, a large *portion* out of the available disk space is allocated to the file based on estimated future file-space requirements. The intent of such a *bulk allocation* is to keep the records closely together, so that the locality within a portion of a file is high.

When all the trains in a portion have been used, but space for a new train is required. a new portion will be allocated and an entry for this portion inserted at the top level of the VSAM B-tree; this level is called the *directory*. The first (low-order) half of the trains will be copied to the newly allocated portion, and the space from the trains moved will be put in a list (see Sec. 4-3-5) of free trains for reuse in the original portion.

A new portion is also obtained by a bulk allocation, but it may be smaller. It will not necessarily be close to the old portion. Benefits of locality are important for sequential processing of the data. Benefits of locality will of course be lost when other users intersperse file requests on the same device. A disadvantage of bulk allocation is that substantial storage space may be unused if the file size cannot be accurately estimated.

To be expanded to 432 pt for Figure space
HERE

Figure 4-14 VSAM space allocation.

Directory A directory to the allocated portions is kept as part of an entry for every VSAM file in a VSAM file catalog. The directory is limited to 256 entries. A possible allocation is shown in Fig. 4-14. The size of the allocation to the initial and to any further portions is determined by the expected maximum file size, as estimated from parameters given at the original file definition time. These parameters also are kept in the file catalog entry. The entire catalog contains many file descriptions and is itself a VSAM file.

The allocation within the portions is also done using the B-tree algorithm. The blocking and insertion of records into trains will be presented below.

Blocking VSAM uses the *train* concept presented in Sec. 2-3-1 to achieve blocking. Each train has a fixed number of blocks, also determined at the time of initial allocation, and is treated as one logical unit. Within a portion there will be space for a number of trains. Some of these may be in use; others may be free. Trains in use contain a variable number of records.

The records may be of variable length and may span blocks within a train. A partially filled train is shown in Fig. 4-15. The actual data records are packed into the forward area of the train; the three-byte markers giving the length of these records are stored in reverse order in the rear of the train. Adjoining records of equal size are controlled by a single, double-sized marker. A control field for the entire train gives the amounts of space used by the markers and left free.

RECORD INSERTION The free space in a train can be used to insert records and position information without requiring index manipulation. The new record is merged into its proper position. Some shifting within the train will nearly always take place to achieve this. The markers may have to be shifted too. Since they contain only the length, old markers retain their values.

If there is insufficient space in the train, a free train will be requisitioned. The B-tree algorithm is carried out in three steps, performed in a sequence which reduces vulnerability of the data in case of system malfunction.

1 The records comprising the first half of the full train are copied into the new train, and the new train is written out to disk.

2 The index is updated to include the new train at the appropriate point; the last record provides the key value for the index entry.

3 The old train is adjusted by shifting the latter half of the records to the front, and this train is rewritten.

During this sequence, the record which was to be added is put into the correct position, and the length markers are set for the old and the new train.

We already have discussed the measures taken when there is no free train in the portion, so that a new portion has to be allocated. Both overflow handling algorithms are B-tree based; they differ mainly in terms of the size of the units that are being manipulated.

LOADING DENSITY To allow updating of the files without immediately forcing the splitting of trains and portions, the initial file density is specified at < 1. Here the B-tree algorithm is used for an indexed sequential file, trading space in order to avoid deterioration of performance due to insertions. Reorganization is not normally needed.

A file which undergoes much updating will eventually achieve an equilibrium similar to the equilibrium conditions described in Eq. 3-47. We used the result that space utilization will approach 69% so that we expect the trains eventually to be 69% occupied.

In addition, there will be some storage loss from unused trains in portions. The directory to the trains in one portion provides for the allocation of initial free trains. Trains which are freed are made available for reuse only within the same portion, so that also here a space utilization of 69% is expected when equilibrium conditions are reached. It will take a very long time to achieve this equilibrium, since portions will not be split frequently if the initial loading density and train allocation was generous. However, a specific portion can overflow while there is yet much space in all but one of the trains.

If trains are initially filled to 60% and portions to 90%, the initial storage utilization will be $0.60 \cdot 0.90 = 54\%$. After a number of updates equal to 40% of the initial loading, the storage utilization will be $1.40 \cdot 0.54 = 75\%$. This is the expected density for the trains under random insertions only. For this case we can expect that now many trains are filled, but also that little portion overflow has yet occurred. Beyond this point portions will overflow more frequently, and the portions will also achieve this density. Eventually the total storage density will approach the limit of $0.75 \cdot 0.75 \rightarrow 0.5625$, or 56%. If insertions and deletions both occur at high rates, the eventual equilibrium density may become as low as $0.69 \cdot 0.69 \rightarrow 48\%$.

If better utilization of space in a highly dynamic environment is desired, a reorganization can be used to reload the file to a desired density.

4-3-2 Addressing

A relative address space is set up for every file. The address begins with the initial portion and continues through all further portions allocated to the file. The components used to construct the address are shown Fig. 4-16. The size of some of the components is established for the life of the file, using estimates of future file size which are provided by the user or estimated by the system at file-creation time.

If no records are inserted, deleted, or replaced by records of differing size, the relative address of a record will remain the same. Such addresses then can be used to access data records in a VSAM file directly. The addresses are used in an extension of VSAM to provide multiple indexes. A primary file will contain the actual data records, and secondary files have lower-level entries which refer to the records of the primary file.

Relative addresses can be obtained by a program as a by-product of any operation on a record of the file to enable such usage. With relative addressing sequential access can be carried out more rapidly. Any change of record arrangements due to updates or insertions will, however, invalidate such addresses.

LIMITS Any implementation of a file has limits. Many limits here are related to the assignment of components of the relative address used for pointing, as shown in Fig. 4-16. The size of the entire address is limited to 32 bits, so that the size of a file is limited to $2^{32} \approx 4.29 \cdot 10^9$ bytes. A portion is limited to one device. Up to 256 portions may be allocated, so that a file has to grow by substantial increments if it is not to run out of portions. Block may be {512, 1024, 2048, or 4096} bytes long, and a train is limited to a total of 32 768 bytes. The size of a record is limited to the space available in one train. There are some other implementation limits to allow simple fixed sizes for the marker fields seen in Fig. 4-15.

To be expanded to 432 pt for Figure space
HERE

Figure 4-17 Components of an index record.

4-3-3 Index Manipulation

A multilevel index is used to access the data records. The index is considered to be a separate but related file. The index entries at the lowest level point to the beginning of the train, but the value associated with each index entry is the highest or last value in the train. Every train in a portion has an index entry, but entries for free trains do not have a key value and are collected at the end of the index. The assignment of fields within a single index train is given in Fig. 4-17. The index entries in a train are grouped into section to accelerate the search within a block. In order to appreciate the issues addressed in a specific B-tree implementation, we will describe the VSAM index in more detail.

To be expanded to 270 pt for Figure space
HERE

Figure 4-18 Index levels in VSAM.

LEVELS We indicated earlier that the number of index levels x is fixed for a VSAM file. Three levels are easily identifiable; the grouping of index entries into sections provides intermediate levels. It is convenient to regard $x = 3$.

If the file uses multiple portions, the level 1 index will be split into corresponding pieces. In the example sketched in Fig. 4-18 these pieces have been located adjacent to the corresponding data portions, which can help to obtain good performance. Locating the pieces of the index on separate or on devices with a smaller access time than the data can increase the fetch performance even more.

The next higher index level (2) provides at the same time information regarding the location of the allocated disk portions and the indexes. It encompasses the portion directory discussed in Sec. 4-3-1.

If a portion is split, a new level 1 index is built for the new portion, the old index is changed to indicate the trains that are now free, and one new entry is made in the second-level index. Since there is a limit of 256 portions, index level 2 is limited to 256 entries, and one third (x) level block provides the root. If the root block is frequently referenced, the paging scheme should keep this block in memory, providing fast access.

BLOCKING OF THE INDEX The index file uses trains of a size appropriate to its own needs. One index record appears to fill one train completely, but within each record there are three types of fields: a header, entries for free trains, and index entries. There may be unused space between the two variable-length areas to allow for changes in the contents of the index record. Figure 4-17 shows the various components of an index record.

The index entries are again stored from right to left within an index record. They are grouped into a number of sections to reduce the index search time. The significance of the various fields is indicated through the matching Greek-letter symbols. In order to read or write programs to interface with the file structure at this level of detail, the reference manuals should be used.

KEY ABBREVIATION Since VSAM deals with general keys, often long names, effective abbrevation of keys is neccessary in order to have a fanout which can address all records in a file using only the fixed three levels. Table 4-2 shows a set of keys and how they would appear in abbreviated form.

The key abbreviation algorithm uses two count fields, each of fixed size, to describe the extent of abbreviation. The first field (F) indicates the number of characters omitted from the high-order part of the key, and the second field (L) indicates the number of characters remaining after both high- and low-order abbreviation. The high-order part of a key can be reconstructed by using the preceding entries. The initial entry establishes the range for the first data block. The F value of the first entry and the L value of the last entry will always be zero.

The low-order part is reconstructed with highest possible values in the character collating sequence. For clarity we assume this character to be "Z". All records to be inserted subsequently with keys from the begin value to the reconstructed boundary value for the first train (BEZ... in the figure) will be added to this first train. The boundaries for all subsequent trains are similarily established once the abbreviation has been done. Note that in the entries, when fewer characters are required to define the next entry, the size of the actual key entry will be zero.

Table 4-2 VSAM Key Abbreviation

Source Key	Key	F	L	Remarks

ninepoint

BADE	BA	0	2	Begin value			
BERRINGER	E	1	1	first train:	BAA.....	to	BEZ.....■
BIGLEY	I	1	1	second train:	BFA.....	to	BIZ.....■
BRESLOW	RES	1	3	third train:	BJA.....	to	BRESZ...■
BRETSHNEV	T	3	1	fourth train:	BRETA...	to	BRETZ...■
BRODY	O	2	1	fifth train:	BREUA...	to	BROZ....■
BRUCKNER	none	2	0	sixth train:	BRPA....	to	BRZ.....■
BUHLER	U	1	1	seventh train:	BSA.....	to	BUZ.....■
CALHOUN	CALH	0	4	eighth train:	BVA.....	to	CALHZ...■
CALL	none	3	0	ninth train:	CALIA...	to	CALZ....■
CROSS	none	1	0	tenth train:	CAMA....	to	CAZ.....■

This example shows the abbreviations if there were only one record per data train. In practice one extreme record per train is chosen as anchor. The anchor in VSAM is the immediate successor record, that is, the first record in the next train. Otherwise it could not be determined if a reference to "BIGLOW" would be an existing entry in the third train or a new entry in the second train.

When the index is used, the entries have to be searched serially. The number of matching high-order characters is counted. If the number of characters that are provided in the index is equal to the number matched, the proper index entry has been found. A proper match occurs also when the search-argument character in the current position is smaller than the matching index-entry key character, or if the index key entry is abbreviated so that no further key character is available.

INDEX SEARCH The number of index entries within a train, y, can be substantial, and the basic procedure to reconstruct the keys for matching is very time-consuming. In order to reduce the search time, a jump search, as described in Sec. 4-2-2 is implemented, leading to $cix = \sqrt{y}$ (Eq. 4-4). The index entries within one train are grouped initially into $\sqrt{y}$ sections. After insertions and deletions the sizes of the sections may differ from this optimum.

The key for the last index entry in each section, the section key $c^*_{s_j}$ of Fig. 4-17, is abbreviated less, to permit reconstruction for jumping from the preceding section key $c^*_{s_{j-1}}$. The first search pass uses this section key and skips from one to the next section using the section length indicator σ_j to find the proper section. When the section has been found, a search pass for the appropriate entry is restricted within that section. This additional pseudo-level does not require repeated entries of the key or pointers, only a reduced key abbreviation for the section keys.

POINTERS Pointers are also abbreviated. The index pointers on level 1 to the data trains include only the displacement within a portion in terms of trains. For instance, if there are 300 trains in one portion, $\gamma = \lceil 300/2^8 \rceil = 2$ 8-bit bytes suffice for the pointer entry. To calculate the byte address of the train, the contents of the pointer field is multiplied by the train size and the beginning address of the portion δ is added.

4-3-4 Sequential Access

In VSAM the index has to be read to process the data trains serially. To obtain high-performance serial access, the use of higher levels of the index is avoided. The linkage pointer (ϵ) is used to locate successive lowest-level index trains, and the data trains are fetched using the pointers of the index entries.

After many dynamic changes of a VSAM file the trains will no longer be in a physical sequence. Within each train, however, the record sequence has been maintained. For large trains fewer seeks is required, but train size will be limited because of buffer capacity and the limits imposed by the relative addressing scheme. A file with portions covering multiple cylinders will have its trains distributed over them, so that seeks during serial access still can add substantially to the time required to read the file.

LOCALITY CONTROL There are some optional features to increase search performance through improved locality. One of these features allows replication of the index on a track of a rotating device so that the average rotational latency for reading of the index can be reduced. Other options provide for the close location of index file components and the data file itself to minimize seek times. Index and data can also be placed on separate devices to provide overlap of access. This is especially useful when processing the file serially.

4-3-5 Programming

Programmers who use a file-access method can operate on one of three levels:

- They can write instructions to manipulate the file on the basis of understanding the logical functions to be executed.
- They can attempt to also understand the underlying physical structure in order to produce programs that are efficient within the perceived system environment.
- They can use attractive features of the file system and attempt to bypass or disable parts of the file-system support which do not suit them, and replace these parts with programs of their own invention.

Each of these choices has associated costs and benefits, and the decisions should be made based on documented evaluations of the trade-offs which were considered. The cost of the first alternative may be inadequate performance and the cost of the third alternative may be intolerable inflexibility.

A philosophy in structured programming, *information hiding*, advocates the imposition of the first approach by not revealing the underlying structure of system programs. This is sometimes unintentionally aided by the lack of clear system documentation. The author prefers to work with a good knowledge of the available tools and use them to a maximal extent. When programming tools are inadequate and have to be rejected, good documention of the reasons, not only of the resulting system, will help successors maintain the programs and may also give guidance to the developers of newer tools.

STATEMENTS The operations that are basic to the use of VSAM and similar methods include the following statements.

An `OPEN` statement is to be issued before the file can be processed. The `OPEN` statement is used to specify the file name, the type of operations to be performed, and optionally, a *password* to be verified. The `OPEN` statement also specifies the area or areas where the parameters for further processing requests will be found. The catalog, the file listing all files, will be searched to locate the file. The device or devices which contain index and data will be accessed and directory information extracted. A check to ensure the use of the right versions of the index files and the data file is made by matching the contents of a field in each file directory which contains the most recent update times. Buffer storage for file processing also is obtained and initialized.

A `CLOSE` statement is used to indicate that processing of the file is completed. The `CLOSE` statement empties any outstanding buffers and updates the file directory.

A `GET` statement is used to read data from the file. Before its execution the search argument and other parameters are placed into a parameter area which specifies what is to be read. The parameters to be set for indexed access include the key value and indicate whether an exact or a less-than-or-equal match is desired. The position and length of the area into which the record is to be placed have to be specified. An area for receiving error messages should also be provided. A further parameter states whether the process will wait until the reading is completed. Otherwise the process can continue computation while the system executes the `GET` process asynchronously. The calling process can check later if the system process is complete. The length and address of the record read will be placed into the parameter list upon completion.

The advantage of using a separate fixed area for the parameter list is that only parameters that change have to be respecified when the next `GET` is issued. To provide more flexibility of operation, it is possible to generate new or to modify existing request parameter areas during program execution by means of `GENCB` and `MODCB` statements.

The `PUT` statement initiates the writing of a record, again using parameters contained in the specified parameter area. When writing, the length of the record has to be set into the parameter list prior to execution of the statement. The parameter area can be separate since its address is specified as part of a `GET` or `PUT` operation.

The `ERASE` statement causes deletion of the indicated record.

The `POINT` statement provides a capability to reset the current-record pointer for serial `GET` and `PUT` statements.

The `CHECK` statement is used to determine whether a previous asynchronous operation has been completed.

The `ENDREQ` statement can be used to terminate an outstanding request. This will cancel an operation which is no longer desirable, perhaps because of a detected error or because a parallel operation has delivered the desired data already.

ESCAPE PROCEDURES Optional parameters allow the specification of program segments which are to be executed when the file system performs certain operations or senses certain conditions in the file. One use of these escape procedures is to compress or expand data records in order to save file space. Others can be used when `End-of-file` conditions occur. Considerable modification of the basic operations is possible when these escape hatches or exits are used.

These facilities are also exploited to create multi-indexed files within VSAM. Upon insertion of a record into the data or primary file an escape procedure will initiate updating of all secondary indexes.

The programmer who uses escape procedures has access to all the options and the underlying code at the same structural level. Now the distinction between the three levels of programming described above is not very explicit, and good intentions can easily come to naught. Programs which are invoked as escape procedures may introduce errors which are not visible when the program which invokes the file services is checked out. These escape hatches can also easily be misused to violate system security.

4-4 TREE-STRUCTURED FILES

Up to this point we have discussed indexed organizations where the index was *distinct* from the data portion of the file. In this section, we will describe file organizations where the goal portions of records are lifted into the index structure itself. A block of such a file is sketched in Fig. 4-19 and can be compared with the block of an indexed file shown in Fig. 3-15. The position of the entire record is determined by the position of its key. The tree-structured file organization is sketched in Fig. 4-20.

level
x = 3

To be expanded to 162 pt for Figure space
HERE

Figure 4-20 A tree-structured file.

Since the goal records are placed into a single tree, this organization is akin to the indexed-sequential file. Seriality is established according to only one attribute. The process of processing the records of the file serially is a generalization of *inorder traversal* for binary trees [Knuth[73F]]. Example 4-7 sketches the algorithm.

Table 4-7 Obtaining Records Serially from a Tree

```
proceed: go to the left subtree,
            if not terminal block
                 then  proceed,
                 else process the block,
         go up to the root for this subtree,
            if there is a record,
                 then do; process the record,
                      go to the next subtree,
                       proceed, end;
                 else do;
                      if this is the root for the entire file,
                           then  done,
                           else  proceed, end;
done:      ...
```

A tree-structured file will be *balanced* if the number of levels is equal or nearly equal for all branches of the tree. A balanced tree will require on the average fewer block accesses to locate a record, given that record requests are uniformly distributed.

4-4-1 Evaluation of a Tree-Structured File

The major advantage of this structure is the nonredundancy of storage. The goal record is directly associated with its key. Key fields appear only once, they are not replicated at other levels.

In the evaluation which follows we assume that the records are densely packed into blocks, although not spanned. Update performance would be better if a non-dense B-tree scheme were used, but the reduction in density reduces the effectiveness of the tree structure.

We now find goal records throughout the levels of this file and at different seek distances from the root index. A disadvantage of the tree-structured file is that the fanout can be significantly reduced versus the fanout of an index so that many more seeks may be required to retrieve an average record of a large file.

The increase in the expected number of block retrievals depends on the ratio of key sizes to goal sizes, which controls the reduction of the fanout factor. Some access benefit is gained because some records (7 out of 26 in Fig. 4-20) are closer to the root and hence easier to retrieve. This factor will only outweigh the reduction in fanout if the goal records are extremely small. This happens when the tree is used as an index tree but then another access is required to access the data file. An operational benefit of trees is that keys are stored in only one place, and hence are easier to update.

Records can be of variable length, since the access algorithms do not depend on a specific number of records per block. In order to simplify the analysis, a fixed value for R will be used below, which will provide adequate results if there are many records per block ($R \ll B$).

The fanout ratio is given by

$$y = \left\lfloor \frac{B - P}{R + P} \right\rfloor + 1 \qquad \text{4-10}$$

found by inspection of Fig. 4-19. Even though the lowest level does not require further pointers, space for such pointers is frequently allocated, since this simplifies further growth of the file. Then the fanout ratio y is the same at level 1.

We assume now that the file is filled at the top levels $(x, \ldots, 2)$, rather than at level 1 as shown in Fig. 4-20, and hence is balanced. This will provide the shortest access paths to the data. The number of records at each level is again determined by the number of blocks on a level and by the fanout ratio. Note that here, as with the original B-tree, the number of records in a block is one less than the fanout.

$$n_{level} = (y-1)b_{level} = (y-1)y^{x-level}$$

If the tree is balanced, only the bottom level (1) will be partially filled; it will contain the remaining records, computed by subtracting the records on levels $x \rightarrow 2$ from the total file size n

$$n_1 = n - (y-1)\sum_{i=x}^{2} y^{x-i}$$

We assume at least a two level tree here, i.e., $n > y - 1$. To find the number of levels in the tree, we follow Eq. 3-48. For $y \gg 1$ an estimate for x is

$$x_{est} = \lceil \log_y n \rceil \qquad \text{4-11}$$

The exact height of the tree can be obtained by summing the number of records at the successive levels $n_{level=x}, n_{x-1}, \ldots$, and stopping when all records are stored.

The expected, i.e., average, fetch time can be computed by taking the fetch time for the n_{level} records at each level and allocating this over the n records in the file. The access path increases from 1 for $level = x$ to x for the bottom level, and this accounts for the first factor in the summation below

$$T_F = \frac{s + r + btt}{n}\left((y-1)\sum_{i=x}^{2}(x-i+1)\,y^{x-i} \;+ x\,n_1\right) \qquad \text{4-12}$$

The fetch time in a tree structure is varies by level, so that the average is not determined by the integer height of the tree, as were the indexes shown earlier. Also, if the tree itself is used as an index, the pointers in the tree to the remote goal data can exist at various levels. The smooth curve in Fig. 4-11 is hence appropriate to describe the fetch time to fanout relationship seen here.

To be expanded to 333 pt for Figure space
HERE

Figure 4-21 Contents of blocks in the SPIRES system.

Table 4-8 Comparison of performance of two tree-structured files and similar B-tree indexed and indexed-sequential files

We will evaluate a tree-structured file and two alternative indexed files each for two cases, **a** short and **b** long records. We will use similar parameters as were used in the example of a personnel file given as Example 3-9 of Sec. 3-4.

In case **a** we will assume that an `employee` goal record contains only a `social security number` of nine characters (V_K) and a `skill code` of six characters (V_G), so that $R = 15$; in case **b** we use a still modest record length of $R = 180$ characters.

The remaining parameters required are

$n = 20\,000$ records, $B = 1\,000$ characters, $P = 4$ characters.

1 Tree-structured file

For the tree-structured file we assume a dense packing of records, and use Eqs. 4-9, 4-10, and 4-11. To verify that x_{est} is indeed equal to x, we also present the contents of each level of the files.

Case a

$y = \left\lfloor \frac{1000-4}{15+4} \right\rfloor + 1 = 53$

$x_{est} = \lceil \log_{53} 20\,000 \rceil = \lceil 2.49 \rceil = 3$

Case b

$y = \left\lfloor \frac{1000-4}{180+4} \right\rfloor + 1 = 6$

$x_{est} = \lceil \log_{6} 20\,000 \rceil = \lceil 5.21 \rceil = 6$

The files are then constructed as follows:

Level	Blocks	Records	Pointers	Blocks	Records	Pointers
6				$= x : 1$	5	6
5				6	30	36
4				36	180	216
3	$= x : 1$	52	53	216	1 080	1 296
2	53	2 756	$\leq 2\,809$	1 296	6 480	$\leq 7\,776$
1	331	17 192		2 445	12 225	
Total	385	20 000		3 964	20 000	

Hence $T_F =$

$\frac{1 \cdot 52 + 2 \cdot 2756 + 3 \cdot 17192}{20\,000}(s + r + btt)$

$= 2.86\,(s + r + btt)$

$\frac{1 \cdot 5 + 2 \cdot 30 + \ldots + 5 \cdot 6480 + 6 \cdot 12225}{20\,000}(s + r + btt)$

$= 5.53\,(s + r + btt)$

2 B-tree index to a data file

One of the alternatives to a tree-structured file is a file accessed via a B-tree index. Here we can consider two further cases, one where the file is record-anchored, as required for multi-indexed files, and one where the file is block-anchored, which is adequate for indexed-sequential files as VSAM. When the file is record anchored the record size does not affect the fetch time, but in the block-anchored case the short records of case **a** require only $\lceil n/Bfr \rceil = \lceil 20\,000/\lfloor 1000/15 \rfloor \rceil = 304$ blocks and corresponding pointers, whereas the longer records require $\lceil 20\,000/\lfloor 1000/180 \rfloor \rceil = 4000$ pointers for the data blocks. The fanout in the index is determined by the density and the entry size $V_K + P = 13$

Record-anchored	**Block-anchored**	
Cases a and **b**	**Case a**	**Case b**
$y = \left\lfloor \frac{0.69 \cdot 1000}{13} \right\rfloor = 52$	also $y = 52$	$y = 52$
$x = \lceil \log_{52} 20\,000 \rceil = \lceil 2.51 \rceil = 3$	$x = \lceil \log_{52} 304 \rceil = 2$	$x = \lceil \log_{52} 4\,000 \rceil = 3$
Fetching a record requires one more access to the file; hence $T_F =$ $(x+1)(s+r+btt)$=$4(s+r+btt)$	$3(s+r+btt)$	$4(s+r+btt)$

3 A dense index to a data file

The tree-structured file presented as alternative **1** of this example uses blocks densely and will hence be much harder to update than the B-tree file of alternative **2**. As a third alternative we will evaluate a dense index, one that is also hard to update. The same considerations of record anchors versus block anchors apply, but the fanout is greater. With block anchoring this case is the indexed-sequential file design presented in Sec. 3-3.

Record-anchored	**Block-anchored**	
Cases a and **b**	**Case a**	**Case b**
$y = \left\lfloor \frac{1000}{13} \right\rfloor = 76$	also $y = 76$	$y = 76$
$x = \lceil \log_{76} 20\,000 \rceil = \lceil 2.29 \rceil = 3$	$x = \lceil \log_{76} 304 \rceil = 2$	$x = \lceil \log_{76} 4\,000 \rceil = 2$
Fetching a record requires also here one more access to the file; hence $T_F =$ $(x+1)(s+r+btt) = 4(s+r+btt)$	$3(s+r+btt)$	$3(s+r+btt)$

if no overflows have occurred.

Summary In comparing these three methods applied to this problem, we see that the best performance is obtained for short records with the tree structure, but that this method is the worst for long records. The indexing methods discussed in Chap. 3 are more robust but cannot match the instances where the tree structure is optimal.

To complete Example 4-8, we will list for the six cases the fetch time and file space.

Record size	**Short** $R = 15$	**Long** $R = 180$
Tree-structured file	$T_F = 2.86, b = 385$	$T_F = 5.53, b = 3964$
B-tree index on one attribute	$T_F = 4, b = 698$	$T_F = 4, b = 4269$
block-anchored	$T_F = 4, b = 311$	$T_F = 3, b = 4080$
Indexed-sequential	$T_F = 4, b = 573$	$T_F = 4, b = 4269$
block-anchored	$T_F = 3, b = 310$	$T_F = 3, b = 4054$

If the records are long, a tree structure still can be employed for the keys themselves by removing the nonkey portions of the records, the goal portions, into a separate file. Elements in this file are found by pointers so that some of the considerations developed for overflow of files are valid here. One extra access will be required to fetch the goal record; so there is an advantage only when operations that operate solely on the key are frequent. Such is the case when keys are used to produce counts or subset TID pointers.

To close this subsection, we show a block layout for a tree-structured file system used for information retrieval (Fig. 4-21), which attempts a compromise by splitting large records into a node segment (located within the tree), and a residual segment, kept in a separate file, to be accessed indirectly.

Fields of the record which remain in the node segment benefit from immediate access. The key, of course, will always be placed within the node segment and will not be replicated in the residual part. The record-marking scheme uses a position table as shown in Fig. 2-11.

4-4-2 Balancing Trees

In tree-structured files the performance of the file can degrade after insertions and deletions are made, since insertions are connected to their ancestors. For instance, insertions of records F1, F2, ... in Fig. 4-20 will increase the file height $x \to 4$, and the path length to these records will also be 4.

In order to retain the performance of indexed and tree-structured files, it is desirable that the tree be *balanced*. An optimally balanced tree has only path lengths to terminal nodes of size x or of size $x-1$. Inspection of Eq. 4-10, which predicted the search time, will show that any other distribution of the records will lead to higher expected search values.

The strict B-tree algorithm also leads to balanced files, although irregularity of insertions and deletions causes a reduction in density and thus longer path lengths, but the effect is bounded by the condition that $y_{eff} \geq y/2$. In tree-structured files balancing has to be explicitly done, either during update operations or by reorganization. Balancing has been extensively analyzed for *binary trees*.

BINARY TREES A tree where each record contains only one value and two branches ($y = 2$) is of interest for two reasons:

- The structure is appropriate for core storage because of its simplicity, and hence is also useful within a block of file storage.
- The behavior of dynamically changing binary trees has been investigated both theoretically and statistically.

A basic record in a tree is called a *node*. Figure 4-22 shows a binary tree equivalent to the file tree of Fig. 4-20. Analyses of conventional binary trees, based on core storage, typically assume equal access costs between linked nodes. The analysis of performance trees, when they are placed on files where multiple nodes fit into one block, has to consider that the access cost along sequential branches in a block is much less than the access cost along pointer branches. Only if locality is so poor that all nodes will reside on distinct blocks will access costs be equal. The equivalent analysis for binary file trees has to take differential access costs into account.

Algorithms to keep trees balanced are covered extensively in Knuth[73S], and their performance is presented by Karlton[76]. For certain file trees useful results can be derived from the analyses of conventional binary trees.

We will show in Sec. 4-1 some simple structures which save space or retrieval time when the data or problems have a well-defined structure. In Sec. 4-2 index construction techniques will be presented, leading to an example of a complex indexed-sequential file in Sec. 4-3. In Sec. 4-4 tree-structured files with data within an indexlike structure are presented, leading to hierarchical files with an example in Sec. 4-5. Sections 4-6 and 4-7 discuss combination-based or direct access and ring structures. The final two sections discuss files based on virtual storage and phantom files.

The information presented in this chapter is more diverse than the contents of Chap. 3. Techniques described in earlier sections will often be applied again in combination with other file methods. Many techniques improve the data retrieval capability by adding redundancy but this increases the amount of space required and makes updates more difficult. The aggregate performance of a system depends on the balance of update and retrieval usage, and will be presented in Chap. 5.

MAINTENANCE OF TREES Problems in tree balancing occur mainly when updating the file. Some of them have been discussed in Sec. 4-3-3. B-trees do not require rebalancing unless either a higher loading density than the equilibrium density is desired or when a simple deletion algorithm does not fill the nodes adequately.

If a file is reorganized, it can be more efficient to generate a new tree sequentially for each level instead of performing n insertions. After y_{eff} entries are written into a block for level 1, one entry is taken for level 2, etc. This procedure may also be advantageous for batch updating.

If a certain pattern of growth is expected, the reorganization algorithm can make decisions which will lessen the effect of unbalanced growth. This is typically done when most update activity is expected to occur at the end of the file. Extra index blocks at the end of each level may be preallocated.

When multiple indexes are to be created, the input to an index creation process can be the output of a transposition procedure as described in Sec. 4-1-3.

4-5 HIERARCHICALLY STRUCTURED DATA

In the previous section, we discussed the use of trees to store data that were sequentially indexed. All data records were of the same type and could be moved between levels as dictated by considerations of access efficiency. Treelike structures have found more appropriate use where data have a natural hierarchical organization. Figure 4-23 shows a data structure which is naturally hierarchical in concept.

To be expanded to 171 pt for Figure space
HERE

Figure 4-23 A hierarchical data structure.

We have already encountered hierarchical data structures when multiring files were presented in Sec. 3-6. In this section hierarchical structures based on trees will be described. A classical example of such data is a genealogical tree or an organizational hierarchy.

A list of assemblies, subassemblies, parts, and material, in a manufactured product, is an important hierarchical structure found in many database applications. This example, also called a bill-of-materials database, is shown in Fig. 4-24. The alternative, a translation of the hierarchy into a long, composite key, does not provide a convenient model to the user: the key for "dial" is

`Products&washing_machine&control&dial.`

A nomenclature which is common and nonsexist is also shown with the tree in Fig. 4-24.

We distinguish hierarchical structures that form proper trees from structures that form banyan-type trees or *networks*. The latter case arises when the goal records can be accessed via more than one path. In the bill-of-materials example, this condition would occur if a certain type of "`Switch`" is used in several of the subassemblies and only one record is to contain all the data for this switch. The record "`Switch`" would then have three parents. In a hierarchical file, every record has only one parent.

4-5-1 Criteria for Hierarchical Data Files

In tree-structured files access is optimized by generating a well-balanced tree. Since the records are similar throughout the file, the fanout ratio is made equal at every level. A hierarchical file is structured by the data; the number of levels and the fanout is controlled externally. We must distinguish now the fanout y, which is a data-structure-oriented value, from the number of entries per block, which is based on the mechanical blocking factor Bfr. In order to be able to process hierarchical data structures effectively, certain structural conditions are to be met.

> One requirement is that there will be a reasonable fanout at each level. The number of children for one record should not be so small that too many levels would be required to find level 1 data. Neither should there be so many children that the search for a specific child will be difficult.
>
> Another condition is that an equal level number implies an identical record type. This means that it is not desirable to change the hierarchy for the "`Spin Dryer`" in Fig. 4-24 by inserting a level below which distinguishes electric from gas dryers, if this is not to be done for all products.

Since this type of file structure is so closely related to the data structure for an application, we will present hierarchical files by means of a detailed example of a simple system.

4-5-2 MUMPS

As an example of an implementation of a hierarchical file we will describe the file system which supports the requirements of the MUMPS language. This language is oriented toward interactive computing on small systems. It was standardized in 1976 by the ANS institute [O'Neill76] and is finding increasing use. The syntax and user interface are close to those found in BASIC language systems. A unique feature is that a hierarchical file-management scheme is embedded in the language, so that it is frequently used for database implementations.

MUMPS operates in an interpretive mode, and hence no declaration of any variables is required. The MUMPS language allows great flexibility in the assignment of values to variable names. All data are represented by strings of up to 255 characters. Size limits are imposed on strings used in computations. A class of variables called *globals* are always fetched from and stored onto data files. To distinguish global variables in the language from program local variables, they are prefixed with the symbol "" or "^ ".

The names of MUMPS global variables are stored in a directory level of the user's file space, so that the name of a data file is permanently associated with the name of the global program variable. There is no control language to associate a file name from a system directory to a reference file name used within a program, but the language provides indirection so that any string can be used to address globals.

Integer subscripts are used to provide keys for the globals, they have a permissible range from 0 to 999 999 999.

If there is a single value associated with a global, as is the case in this write statement,

```
SET username="Penelope"
```

then the string `'Penelope'` will become the value of the global named `username`. Single globals of this type appear in the top level of the hierarchy. They provide a useful means for the communication of status variables, as `DONE` in Fig. 4-25, between program segments. To store one patient name with a key of `patno` into a `drug` global file which will have many patients, a subscript is used, as

```
SET drug(patno)='Manuel Cair'
```

To read the value of the name of patient identified by `hospno`, one writes

```
SET name=drug(hospno)
```

The number of subscripts ns used determines the level number in the hierarchy, $level = x - ns$. Note again that the subscripts are not used for address computation, but as search arguments and keys for the hierarchical global files.

The flexibility of MUMPS is seen in Fig. 4-25 where the field for drug frequency on level 1, item 2, has been set to a data element of a different type by

```
... SET icdno=49390
    SET drug(patno,dateno,icdno,1,3)="PRN"
```

using a string data value, indicating *as needed*, here rather than the usual numeric data element specifying a frequency.

An analysis of MUMPS files also provides an example of the capabilities which can be provided by systems that are considerably less massive than those that support the elaborate indexed-sequential files discussed earlier.

STORAGE STRUCTURE To begin with an example, we consider the following structure used for records in a clinic pharmacy. The description which follows will assume a B-tree data file approach followed in modern implementations. Since MUMPS does not need statements to define variables and storage, we will use a PL/1 notation to document the intended data.

Table 4-9 A Hierarchical Data Structure

```
DECLARE
   1 drugrecord,
     2 patients (n1),
       3 patient_number,
                                          } 2
       3 patient_name,
       3 visit_dates (n2),
         4 treated_problem (n3),
           5 drugs_prescribed (n4),
             6 drugname,
                                          } 3
             6 quantity,
             6 frequency;
```

This structure implies a hierarchy which provides storage for `n1` patients with *2* data values {`patient_number, patient_name`} each and a subtree for `n1·n2` visits, `n1·n2·n3` problems, and `n1·n2·n3·n4` drugs with *3* data elements per drug. Conventional languages and their files will have to allocate excessively large amounts of storage to accommodate the possible maximum array sizes.

If 500 patients may be seen in the clinic per day, if patients might have 100 visits and might have 20 diagnosed problems, and if treating a problem can require up to 10 drugs, then the array space required for the data items on each level is

$$1 + 500 \cdot \mathit{2} + 500 \cdot 100 + 500 \cdot 100 \cdot 20 + 500 \cdot 100 \cdot 20 \cdot 10 \cdot \mathit{3} = \quad 31\,051\,001$$

At some point in time there may actually be 450 patients seen in a day, the average number of visits is 8, the average number of problems per visit is 2.3, and the average number of drugs used for a problem is 1.2, so that only

$$1 + 450 \cdot \mathit{2} + 450 \cdot 8 + 450 \cdot 8 \cdot 2.3 + 450 \cdot 8 \cdot 2.3 \cdot 1.2 \cdot \mathit{3} = \qquad 42\,589$$

data fields would be occupied at that time, and more than 31 000 000 will be empty.

We wish to avoid these null entries because of the storage cost, and even more because of the processing cost incurred when fetching and rejecting empty fields. The hierarchical structure provides a method to achieve compact storage.

A MUMPS data file stores each data value in a distinct *segment*. The relationship of segments will be explained using the nomenclature of Fig. 4-24. A segment is identified through its position in the hierarchy of ancestors and by a key which distinguishes multiple children of one parent. The segments which are logically part of one record are not stored together. Segments of different records that are at the same level are stored together. Those segments on one level that belong to the same parent are arranged to form a continuous *segment sequence*. A logical record can be constructed by assembling from each level the segments that are related by a parent-child binding. Among multiple children the child with the desired identification can be chosen. For example, a record of Fig. 4-24 is made up of keys and data found at each level from `Product` to `Dial`.

A segment forms a triplet or quadruplet {level, key, value, and/or child}. The level is implied. Table 4-3 describes the stored components of a segment.

Table 4-3 Components of a MUMPS Data Segment

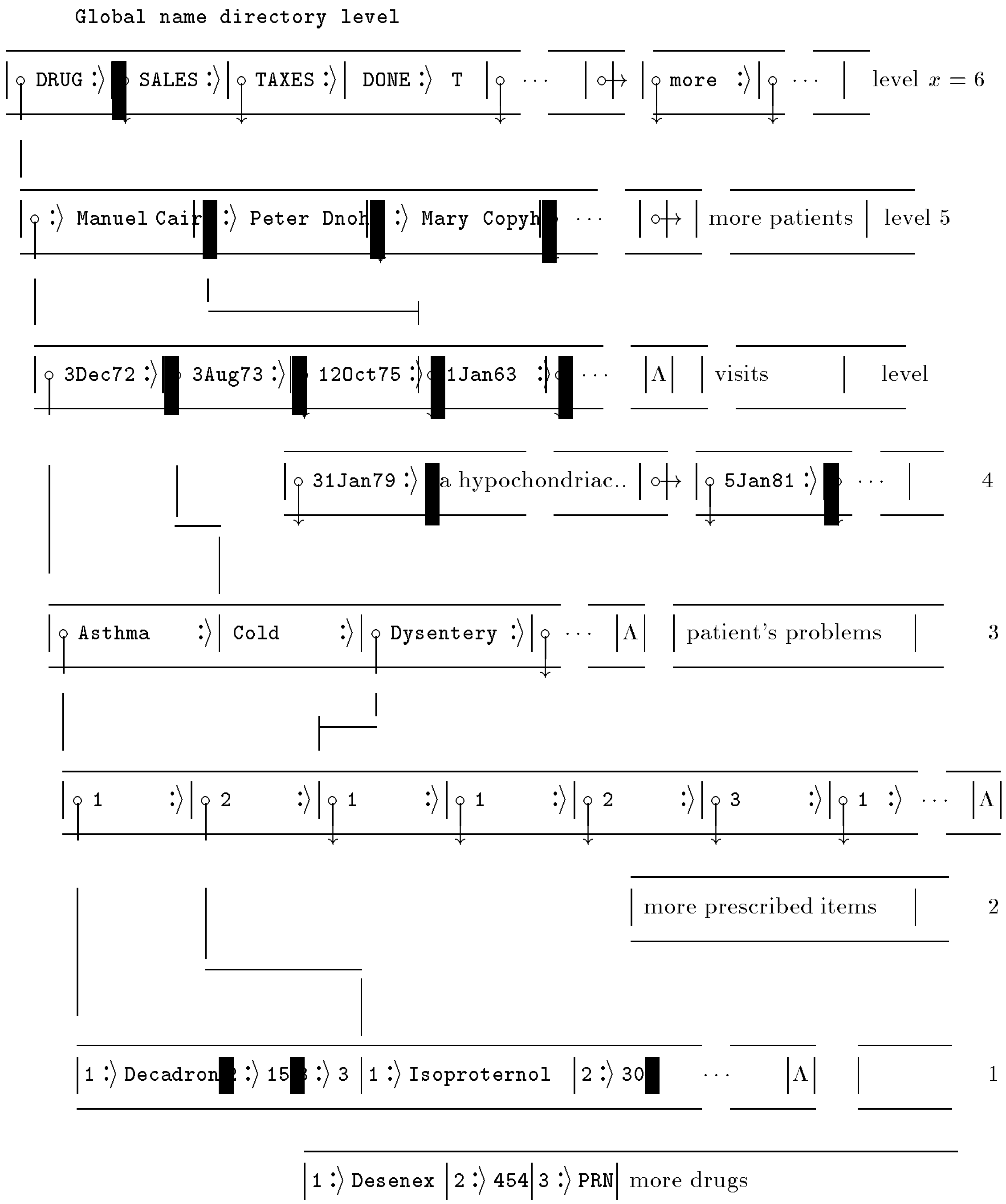
Function	Code	Typical value
Key (integer)	V_{key}	5 chars.
Type identification for the value field combination	Pid	1 char.
Data string value (if entered)	V_G	15 chars.
Pointer value to children (if entered)	P	2 chars.

Multiple children of the same parent, *siblings*, are appended to their particular segment sequence in order of arrival. Siblings which do not fit into the initial block referenced are put into successor blocks. Blocks are split as required. A linkage between blocks, as presented in Sec. 4-2-3, is maintained to assure rapid access to all segments of a sequence.

The actual implementation of the conceptual hierarchy is shown in Fig. 4-25. The position of the segment sequence with all children is defined by a single pointer from the parent. The key for a data item is part of the segment, so that a search for a sibling is done at the same level used to keep sibling data. The number of pointers in a block is relatively high, since a segment contains only one or zero data elements, and hence is quite short. The actual format of the segments is described later. We will now consider the data found in this file structure.

CONTENTS OF THE FILE An entire hierarchical structure is known by one global variable name kept at level x, here `drug`. The variable names stored are limited in length to 8 characters to provide a high density of segments per block (Bfs_x) at the top level. There are no names associated with the variables at the lower levels, and extrinsic documentation is required to explain the meaning of levels in the hierarchy.

At level $x-1$, we find the patient number as key and the name as data. At level $x-2$, the date itself provides the key, and no actual data field is required. To permit use of the date as a key, the value of the date is represented in storage as an integer, say, `31271` for `3DEC71`. Patient `Cair` has had three visits recorded.



On level 1 are the triplets describing each drug prescribed

To be expanded to 350 pt for Figure space
HERE

Figure 4-25 MUMPS data storage.

The pointer field of the date segment allows retrieval of the problem list on level $x-3$. A problem name could be kept as data; here the problem is identified through a standard coding method. A suitable key might be the ICDA code (International Classification of Disease) which assigns numbers of the form xxx.xx to diseases. For a value of this code, 493.90, the key would be 49390. We have again a segment without an explicit goal field. Patient Cair had only one problem treated at his 3Dec72 visit.

Level $x-4$ contains as key a sequence number of each drug given in the prescription for this problem. For each drug there is a pointer to a segment sequence on level $x-5$. Cairs prescription for Asthma had two items; on a later visit for a Cold nothing was prescribed.

At this final level we find the actual data segments for a drug. In our example this segment sequence has always three segments with keys of 1,2,3 and associated data fields. These segments have no further pointers.

To retrieve drug data, we have available as keys the patient number, the visit date, the problem number, and the drug sequence number. The fetch statement to get the name of the drug would read

```
SET drugname=drug(patno,dateno,problemno,drugno,1)
```

and would require the 6 steps shown in Example 4-10 for execution. Intermediate level data, such as the patient name, can be found with fewer accesses.

Table 4-10 Navigating Down a Hierarchy

```
Access the initial block containing the global names,
   search for "drug",
   read through successor blocks until global="drug" found,
 Follow pointer to first level x-1 block ,
   search for patient segment  key=patno .
   /* If we have many patients, many blocks may have to be searched. */
 When found, follow pointer in segment to date level block,
   search for visit date  key=dateno .
 When found, follow pointer in segment to problem level block,
   search for problem number  key=problemno .
 When found, follow pointer in segment to prescription level block,
   search for drug number  key=drugno .
 When found, follow pointer in segment to drug data level block,
    and then search for drug data segment  key="1" .
 When found, pick up data field ("Decadron") out of segment.
```

As in all hierarchical files the natural structure of the data has a great effect on file performance. Within a single level the length of the segment sequence y is a very critical parameter. This length is determined by the number of siblings on one level and is not controlled by file-system procedures. If Bfs denotes the average number of segments in one block; a B-tree implementation will permit

$$Bfs = 0.69 \frac{B}{V_{seg}} \qquad \langle \text{MUMPS B-tree} \rangle \ 4\text{-}13$$

where V_{seg} is the segment size.

Sibling segments within a segment sequence, i.e., having the same parent, need not be kept in any particular order.

To locate all siblings in arbitrary order requires only sequential access to segments in a block and, if the sequence y extends beyond the block, serial access to the successor blocks on that level. To minimize the cost of accessing long sequences, most MUMPS systems do attempt to maintain locality of blocks when a block split occurs. The system may avoid spanning a segment sequence, which fits into one block, over two blocks. Such an optimization is possible since the segments are not sequential but the space versus time trade-off is quite complex.

4-5-3 Evaluation of a MUMPS File

Following the general description of the MUMPS file structure, we will evaluate a typical organization in greater detail so that some criteria for proper utilization of MUMPS capabilities become clear.

CHAINS The number of seeks or block fetches is determined by the number of levels and the length of the chains at each level. Figure 4-25 shows a structure where most segment sequences were short. Exceptions are the list of patients on level $x-1$ and the sequence of visits of a hypochondriac on level $x-2$.

The number of blocks to be accessed on one level depends on the number and size of sibling segments and on the size of the blocks used. In order to use little core storage, the implementers of MUMPS choose fairly small block sizes, from 128 to 512 characters in length. Figure 4-26 shows such a block. We will assume for our example a system with 512 bytes per block.

Two fields in each block are used for block management. One indicates the number of characters used in the block, and the other one contains the successor-block pointer. Binary integers are often used as pointers so that fields of two bytes in length can contain positive integer values up to 65 535. This value is not adequate for all blocks on a large disk unit, so that this block number is specific to one user. To each user, a number of cylinders are assigned, within which all the global variables are allocated, so that all the files of one user are limited to 65 535 blocks total.

SEGMENT FORMATS A segment consists of a type identification, the key, and as goal either or both pointer and data fields. Figure 4-27 shows some possible segments.

4	123 45 6789	↓	Manuel Phillipus Cair■	Patient data
3	3 12 72	↓		Date as key, no goal data in segment
1	161 00			Disease key for Common Cold, no drugs prescribed

| 2 | 1 | Aspirin■ | Drugname, no descendants

The *type identification* indicates what type of fields this segment contains. Two bits are required to encode the four possibilities:

{1: key only; 2: key, data; 3: key, pointer; 4: key, data, pointer; }.

The *key* field extends from the identification field and requires up to 9 digit positions. In binary form the value could be encoded in 33 bits.

A *pointer*, when present, occupies two character positions. Its presence indicates that this segment has a descendant subtree.

Data when present are of variable length, up to 255 characters, and are terminated with a special end-of-string marker.

EVALUATION OF THE HIERARCHICAL DATA STRUCTURE We now will evaluate the drug file level by level using the assumptions made earlier. We will assume segment sizes based on the values in Table 4-3. The average length of the data strings (15 characters) includes the end-of-string marker. Computation of the segment size

$$R' = Pid + V_{key} + V_G + P \qquad \text{4-14}$$

has to consider if a data or pointer value exists at the level. The example contains data at only two levels $\{1, 5\}$ and pointers are needed on all levels but the bottom one $\{1\}$. Denoting the size of a segment of a record at level i as R'_i, we find that the number of segments per block

$$Bfs_i = dens \left\lfloor \frac{B - 2P}{R'_i} \right\rfloor \qquad \langle\text{hierarchical file}\rangle \text{ 4-15}$$

The density for a B-tree data file implementation can again be estimated with Eq. 3-47 as $dens = 0.69\%$. Large segment sizes can lower the density further if spanning of blocks by single segments is avoided.

Since segments sequences can span blocks, a number of block accesses may be required to find a specific segment at one level. The expected number of blocks comprising an entire segment sequence of fanout y is

$$b_i(y_i) = \lceil \tfrac{1}{2} + y_i / Bfs_i \rceil$$

The extra $\frac{1}{2}$ block appears because a segment sequence may start at an arbitrary position in the first block.

If $b_i \gg 1$, the fanout y_i is too high at level i because the number of block accessed will be large on one level. There will also be fewer segments in the last block of the sequence, so that for arbitrary sequences the number of accesses can be precisely estimated using the procedure leading to Eq. 4-12, assuming an equal frequency of access to all segments of the sequence. The number of expected block accesses are the sum of accesses to the first block, accesses to reach intermediate blocks, if any, and accesses to reach the remaining segments in the final block, all divided by the number of segments in the sequence.

$$b_G = \frac{1}{y_i}\left(\tfrac{1}{2} Bfs_i + \sum_{j=2}^{b_i - 1} j\, Bfs_i + b_i\left(y_i - \tfrac{1}{2} Bfs_i - \sum_{j=2}^{b_i - 1} Bfs_i \right)\right) \approx \left\lceil \frac{b_i + 1}{2} \right\rceil \qquad \text{4-16}$$

The approximation applies when $y_i \gg Bfs_i$ so that the lesser number of segments in the first and last blocks does not affect the result greatly.

In MUMPS we find on level x the directory of file names or globals for a user. At this level the name strings are limited to 8 characters and no other key is needed.

If the average global-variable name is 5 characters, the segment size $R'_x = 9$ and $y_x = \lfloor(512-4)/9\rfloor = 56$. When there are many global files, including unsubscripted globals, it is likely that more than one block is required for the directory level of the user.

The initial segment for the `drug` file will be created when the first patient is stored by a statement as

```
SET drug(patno)='Manuel P. Cair'
```

The segment in level x will consist only of the key `drug` and a pointer, and a block on level $x-1$ will be allocated to store the key value from `patno` and the string with `Cair`'s name and the end-of-string mark. We assume for `patno` a 9 digit or five character encoding. The above patient segment will require $R'_{x-1} = 1+5+15 = 21$ characters, but as soon as lower-level segments are entered, a pointer field will be inserted, so that the level $x-1$ segments will actually become 23 characters in length. This will make $Bfs_{x-1} = \lfloor\frac{512-4}{23}\rfloor = 22$ so that there can be this many patient segments in a block. For the 450 patients seen in the clinic, the segment sequence will occupy 21 blocks. These have to be searched sequentially for every reference, requiring an average of 11 block accesses on this level. An approach to avoid this high cost is given at the end of this section.

We will summarize the remaining levels of the example rapidly since we find that all further segment sequences will fit into a block. On level $x-2$ the `visit dates` are stored as keys. This means that no `SET` statements with only two subscripts will be executed. This level will be filled indirectly when the `dateno` is used as a subscript to lower levels. On level $x-3$, we find the problem treated during the visit. Again, the problem is encoded as the key, `"Asthma"  icd=493.9  icdno = 49390`, and the key is set implicitly. If no drugs are prescribed for the problem the hierarchy terminates here with a segment of type 1.

Level $x-4$ provides pointers to the drug description. An alternative design could store the drug name as a data field on this level. It again contains segments having only a key and a pointer. Level $x-5$ contains the actual drug data as a sequence of three segments. This level would be loaded by statements such as

```
SET drug(patno,dateno,icdno,dno,1)="Decadron"
SET drug(patno,dateno,icdno,dno,2)=15
SET drug(patno,dateno,icdno,dno,3)=3
```

NAKED VARIABLES To simplify and speed execution of such sequences, an option is available that will allow reuse of a previously reached node, so that redundant searches down the tree can be eliminated. The absence of a variable name implies the reuse of the previous global variable and all but the last one of the subscripts. Further subscripts leading to lower levels can be appended. Using the *naked* variable notation, the last two statements become

```
SET (2)=15
SET (3)=3
```

The number of blocks accesses to reach this level $(x-5)$, assuming one block access for the directory level, is $T_F \approx (1+11+1+1+1+1)(s+r+btt) = 16(s+r+btt)$. Some seeks may be avoided, especially on level $x-1$, if locality is maintained. Further naked accesses to the same segment sequence are free if the buffers are retained, at least until the end of a block is encountered.

OPTIMIZATION BY CHANGE OF THE HIERARCHY Optimal fanout would avoid sibling overflow blocks. When the fanout $y \gg Bfs$, the hierarchy becomes a poor match for the file, as seen on the patient level $x-1$. A solution is to change the hierarchy originally envisaged for the file. We will apply this notion to the patient level of our example.

In order to obtain better performance, this level might be split into two levels, perhaps by using half of the patient number as the three-character key for each level. The name will appear only on the lower level $(x-2)$. Now $R'_{x-1} = 1+3+2 = 6$, $Bfs_{x-1} = 84$ and $R'_{x-2} = 1+3+15+2 = 21$, $Bfs_{x-2} = 24$ (Eq. 4-14). The original fanout $y = 450$ is distributed over two levels, so that $y_{x-1}, y_{x-2} = \sqrt{y} \approx 22$. Both $y_{x-1} < Bfs_{x-1}$ and $y_{x-2} < Bfs_{x-2}$, so that most segment sequences will fit into one block. The block fetches for the patient data have been reduced from 11 to slightly over 2. This file will have 6 levels, and $Tf \approx 6(s + r + btt)$.

If the file would have to hold information for all patients seen during a long period, say a year, then at least two and probably more patient levels will be needed to provide adequate performance.

4-5-4 Storage Allocation

Occasionally long chains of sibling data create long segment sequence, and the optimization process described above on the patient name. The search for children down the hierarchical tree also follows pointers. Optimization of locality for the serial blocks according to both dimensions is hence important. Most MUMPS implementations, therefore, attempt to allocate the next block in a series within the next accessible sector. A *sector* is defined to consist of all the blocks which can be read on one cylinder at the same rotational position.

Figure 4-28 shows a series chained on an unwrapped cylinder surface. Optimal placement has been achieved for the first four of the five blocks of chain C shown.

To be expanded to 330 pt for Figure space
HERE

Figure 4-28 Storage for a MUMPS cylinder.

Accessing blocks of C avoids latency r and seek time s. The example assumes that the next sequential sector can be read. On computer systems where buffering constraints prevent this, alternate block referencing is preferred (see Fig. 2-15).

In order to locate the best free block, a table entry for each sector is maintained which points to the first block of a chain of free blocks within the sector. If there is a free block in the sector, it is assigned to the chain which requested it, and the table is updated with the link value from this newly obtained block. If all of a cylinder is filled, so that the new block has to be placed on another cylinder, no rotational optimization is attempted. In fact, an extra sequence of file accesses may be required to store the old free space table and fetch one for the new cylinder.

When storage is released by the delete statement `KILL`, such as

```
KILL drug(patno)
```

all descendants of the deleted segment are also released. A lower-priority task collects all freed blocks and attaches them to the appropriate free space chains. These blocks can then be reused. Alternate methods for free space management are discussed in Sec. 6-5.

4-5-5 Conclusion

The fact that the method used by programs to reference data in hierarchical systems, such as MUMPS, is so closely interwoven with the file structure is both an advantage and a liability. In Chap. 7 we will discuss methods that try to maximize program and file structure independence. The interdependence of data and file structure could also be perceived earlier, in Sec. 3-6, When discussing ring-structured files, which also implement a hierarchical design.

When a hierarchical file structure is called for, careful data analysis and data dimensioning has to precede a file design. Intermediate indexes or direct accessed lists may be interspersed to provide adequate performance over the expected range of quantities of data on each level.

Hierarchical views of data may also be provided by more general database systems which can provide a data-independent system design. When the range of data and applications cannot be adequately assessed initially, the higher overhead costs of these more formal systems will be less than the cost of frequent rewriting of the database and database programs in order to match the data structure.

4-6 METHODS BASED ON DIRECT ACCESS

The direct-access method is often the fastest means to access records, but three features make the basic method often unsuitable:

- Its dependence on fixed-length records
- The fetch arguments that are limited to one key attribute
- The lack of serial access

Simple adaptations of direct access can be used to overcome these three limitations for many applications. A number of these will be illustrated in corresponding sections below. A fourth section will deal with partial-match retrieval.

To be expanded to 234 pt for Figure space
HERE

Figure 4-29 Bucket accessing to a direct file.

4-6-1 Variable-Length Records

The record length for directly accessed files is fixed because of the mapping from a record key to a slot number in the file area. If a direct-file organization is desired for its other features, either of two constructs will add variable-length record capability.

BUCKET ACCESSING When buckets are used, variable-length records can be accommodated in the buckets. The search within a bucket becomes more complex.

The use of a record mark as shown in Fig. 2-11 enables serial searches through the bucket. Some disk hardware allows the channel or controller to locate a specific record on a track; this can cost considerable gap space but reduces the software needed to read through blocks in order to identify the marks. If the bucket search time cix is a problem, one of the schemes used to structure index blocks (Sec. 4-2-2) may help.

If an overflow from the bucket occurs because of the insertion of many records that were longer than the expected average, the existing-collision management mechanism can be invoked. On the other hand, when the records in a block are shorter than average, overflow resulting from collisions due to hashing will be deferred. The use of buckets for variable-length records works best if $R \ll B$.

When pointers are used to mark records, as shown in Fig. 4-29, overflow chains can be maintained using these same pointers. In Sec. 3-5 overflow chains were used to manage clusters occurring because of insertion and deletion. Deletion of variable-length records is best accompanied by a rearrangement of the records remaining within the block to avoid space fragmentation.

To be expanded to 240 pt for Figure space
HERE

Figure 4-30 Indirect accessing to a direct file.

INDIRECT ACCESSING In this method, a small file containing a list of pointers is accessed directly; the selected entry in this list indicates the location of the actual record. The organization of the goal data file may be a pile or another organization type chosen according to other requirements placed on the file.

The *pointer-list* resembles the lowest level of an index, but the position of the entries is determined by the key-to-address transformation algorithm, so that the list is not suitable for serial searching. The benefit of the direct file, that only one block has to be fetched to locate a record, is lost, but two accesses will provide access to any record which is not in an overflow area.

The simple pointer-list can have a high fanout ratio, because no key is required if collisions are handled after accessing the main data records. The main file will use linkages to chain colliding records.

The alternative is to place the data keys also into the pointer-list entries. Now access efficiency is improved for colliding and missing records. The keys of overflow entries due to collisions will then also be placed in the pointer-list using open-addressing techniques. The associated record is itself placed in the main file as before.

If the pointer-list is kept using bucket techniques, a copy of the key is required in the pointer-list to identify the entry. These buckets will have many entries, so that techniques to increase the speed of searching through the bucket are desirable. A binary tree seems ideal for this purpose; its pointers can be used for locating child entries or goal records.

In each of these approaches, the collision rate is determined by the ratio of m/n of the address space. This means that only the pointer-list has to be given extra space and the main file can be kept dense. A high space ratio m/n and the many entries which fit into an index block *Bfr* will keep the occurrence of collisions minimal.

If some part of the goal record is required more frequently than the whole record, a mixed approach is feasible. A fixed-length record segment is constructed containing the most useful portions and placed so that it can be directly accessed. The variable segment which remains is retrieved when needed via a pointer from the fixed segment.

4-6-2 Multiattribute Direct Access

If the fetch requests are not restricted to one key attribute, the simple direct-access method fails. For each different attribute and its value in a goal record the particular transformation will produce its specific record address. If there is to be only one stored copy of each goal record, indirection via a pointer-list is again needed.

The pointer-list provides only pointers to the actual goal record; the goal is found at the cost of an additional access. Multiple pointer-lists will be used, similar to the multiplicity of indexes described for a multi-indexed file. Records can be found according to any attribute for which a transform and a pointer-list is provided. The pointer-lists are similar to the pointer-lists used above when provision for variable record length is made and the rate of collisions should again be low.

To be expanded to 288 pt for Figure space
HERE

Figure 4-31 Three-way access to a direct file.

The records may still be located directly for the *primary* attribute and indirection employed for *secondary* keys. However, for symmetry, the system sketched in Fig. 4-31 uses indirect referencing throughout. Not all attributes need to have address spaces of equal length. The goal file itself may have a pile or a sequential organization. An insertion into a file using multiple pointer-lists will require changes to all pointer-lists in addition to the operation on the main file.

All entries for the same transformed value will have the same pointer-list address. This increases the collision rate for attributes which are not unique. When we dealt with simple direct files, whose records are stored by key values, we expected uniqueness. Many other attributes in records will not have unique values. The example of Fig. 4-31 is hence atypical; you may wish to consider the pointer-list for `sex = "male"`. Identical data attributes must lead to identical addresses in the the pointer-list space. This problem can also be dealt with through the collision-handling mechanism, but obviously attributes with poor partitioning power can overwhelm the mechanics of many solutions. The result of the key-to-address transformation can no longer be uniform. The use of overflow buckets rather than open addressing is now called for.

A popular database management system, ADABAS, uses the types of access presented here. The pointer-lists may be predefined and maintained, or created dynamically in response to transaction requests.

SHARED POINTER-LIST It also is possible to share a single pointer address space for several attributes by making this space appropriately larger and including the name, or another suitable identifier of the attribute itself, in the input to the address transformation function. Figure 4-32 sketches this design using the symbols (I_1, I_2, I_3) for attribute-type identification.

This method becomes especially interesting if the records have an irregular number of attributes, so that the individual pointer address spaces are of different sizes and possibly have dynamically varying length requirements. The fact that no individual pointer-list areas have to be dimensioned and allocated reduces the amount of unused space while keeping the probability of exceeding the pointer space low. A larger shared space can also reduce collisions due to clustering.

To verify that in case of a collision the correct pointer is picked, it is necessary that the pointer entry can be matched to its attribute name. If this check is not made, problems will arise if two key values of different attributes are both identical and produce the same result from the address transformation. An example would be a case where the `health_record_number` for `John` and the `employee_number` assigned to `Peter` are identical and both happen to transform to the same address by different algorithms. The author has come across a system where this case was not considered, but no access failure had ever been reported. Transformations could be designed to avoid this possibility, but this restricts the choices available and limits the adaptability of this approach.

A COMMON TRANSFORMATION PROCEDURE FOR MULTIPLE ATTRIBUTES■ A file using many attributes for access may need as many independent transformation routines. The definition and use of distinct routines can be avoided by transforming both the attribute name and value into an address within a single shared pointer-list. The sketch in Fig. 4-33 explains the catenation of the name and value strings prior to hashing.

A disadvantage of the simple scheme is that the pointer entry has to carry a long identifier for verification purposes. The attribute name is better carried in abbreviated form. The abbreviation might be a sequence number $I, I = 1 \rightarrow a$ in a dictionary or schema for possible attributes, and this solution is illustrated. The abbreviation has only to add enough information to the hashing algorithm to compensate for the increase in address space size. The combined input to the hashing algorithm has to be kept as a key to test for collisions.

Another solution to the collision problem is to defer the recognition of collisions to the goal records. The exact implementation will vary depending on the form of the goal records.

To be expanded to 396 pt for Figure space
HERE

Figure 4-32 A shared pointer-list for a direct file.

288pt Figure space

Figure 4-33 A common key transformation for a direct file.

remove 'file' from fig 4-33 ex 4-35

A COMMON KEY-TO-ADDRESS TRANSFORMATION FOR MULTIPLE FILES We considered in the previous subsections very large goal files, accessed using multiple attributes. As part of most databases many small files have to be maintained as well. Examples of such files are the lexicons which translate from a numeric code to a string for output, for instance, a disease code to a disease name. In some applications there may be several dozens of such lexicons.

These many small files are effectively accessed directly. A technique using a common access mechanism, as sketched in Fig. 4-34, can be used to provide a single access mechanism for multiple files.

The goal data are in separate files. They may be maintained simply as sequential files and periodically reorganized. A batch update of the direct pointer file can be combined with a purge of the previous pointers pertaining to a given goal file.

REPLICATED POINTER-LISTS TO DISTRIBUTED DATA Another form of this technique can be used in *distributed* systems. It may be effective to partition a single large file into fragments which will reside on the processor of a site where the data is used the most or where updates are executed. An example is the `employees` file of a company having multiple locations. A pointer-list to all records of the file can be replicated at all locations so that any record in any fragment can be fetched. All pointer-lists have to be updated when an employee is added or leaves, but remain unchanged when a pay or work status attribute is updated.

To be expanded to 315 pt for Figure space
HERE

Figure 4-34 A common key transformation for multiple files.

Here we have cases where the earlier definition of a file has been violated. It now is difficult to tell whether we are dealing with one, three, four, or even six files. The determination of number of files in these examples is left to the taste of the reader.

4-6-3 Serial File Access

Methods which use indirection by hashing to pointer-lists allow placing the data records sequentially according to one attribute. Serial processing of such a file by one attribute is now possible. Insertion, however, is now difficult, so that much of the flexibility of a direct-access file is lost. The use of a B-tree structured data file can keep updates convenient and provide seriality in one dimension.

In order to provide serial access according to multiple dimensions, the goal-record file may be organized as a ring structure. Flexibility of retrieval has been regained at some cost in complexity.

This combination of direct access and rings is found only in database systems, since procedural management of the complex structure is beyond the ambition of most programmers. Without pointer-lists direct access is provided for only one attribute, and this is a restriction of many CODASYL systems. If more general direct access is provided, the update process has to maintain simultaneously pointer-lists and change the ring pointers. The awkwardness of update may make the batch reorganization technique presented in Sec. 3-5, Eq. 3-89 appropriate here.

4-6-4 Partial-Match Retrieval with Direct Access

In basic direct files, access capability is restricted to one attribute. If intermediate pointer-lists are used, as described in Sec. 4-6-2, then access becomes possible by any of a variety of attributes. These pointer-lists can become the source for partial-match query processing.

The buckets that are created in the pointer-lists will contain TIDs for specific attribute-name and value combinations. If collision chains are maintained all identical entries will be on one chain; some irrelevant ones may be chained in as well and have to be removed based on their key value. The sets of TIDs obtained can be merged for partial-match retrieval, so that selection can be accomplished prior to accessing the goal file. The procedure can be visualized using Fig. 4-35 and compared with Fig. 4-12, where indexes were used.

We note that ranges of values are not supported unless the complications of additional serial access structures (Sec. 4-6-3) are accepted. The attribute values are hence made discrete; the `Price`, for instance, is given here as two distinct truncated values $\{31\,000, 32\,000\}$ rather than as $\{31\,000 \rightarrow 33\,000\}$ in the indexed case of Sec. 4-2-5. The information loss from truncation increases the number of collisions.

PARTIAL-MATCH HASHING The accessing and merging of many buckets can still be costly. An obvious approach to rapid direct partial-match access is to develop hash functions and pointer-lists which permit immediate retrieval of records in response to queries which specify multiple attributes in their search argument. Any or all combinations of more than one attribute, as were considered for indexes in Sec. 4-2-5, may be defined and added to the a pointer-lists for the single attributes.

To be expanded to 189 pt for Figure space
HERE

Figure 4-35 Multiattribute direct access lists for partial-match retrieval.

Providing for all possible $(2^a - 1)$ or even many partial-match attribute combinations increases the number and hence the redundancy in the access lists greatly. The reduction in number of indexes made possible by the sorted ordering within indexes does not apply here.

When multiattribute hashing techniques are being considered, the design decision has to balance redundancy in the files, leading to high update and storage costs versus the effort saved at retrieval time. This choice is actually equivalent to the familiar problem of binding-time selection, and we will abstract the alternatives using the binding paradigm.

Early Binding If fast access is paramount, all possible partial-match attribute combinations for each record can be hashed. At each hash address a set of pointers or TIDs will direct the fetch immediately to the goal records.

All these TIDs must be set when a record is inserted. If an average record has a' attributes of interest for retrieval, there will be $2^{a'} - 1$ entries per record.

The entries for the four `houses` retrieved for the query in example of Fig. 4-35 are shown in Fig. 4-36. These four `houses` have $4(2^a - 1) = 60$ TIDs distributed over the $2^a + 2^{a-1} - 1 = 23$ buckets for the $a = 4$ attributes and the fifth value for the extra `price`.

Entries for all records needed in response to a query will be found in the bucket; all records with identical key combinations cluster in the bucket. The bucket may contain further entries because of collisions from the randomization. Since the clustering of popular and few-attribute combinations leads to dense usage of some bucket addresses, a chained overflow to external buckets rather than open addressing is called for. Combinations which do not correspond to records in the file will not generate entries in the pointer-lists; there may be, for instance, no "`Condominiums`" in "`Wuth.Heights`".

Late Binding Rarely can we afford to allocate in advance pointers for every possible query. In Fig. 4-37 we show an alternative design where a predetermined number of structured pointers. Multiple of these pointers are needed for a given query, and the records to be fetched will be in multiple buckets. In the example the number matches approximatly the number of records in the file, so that the number of accesses needed is a bit more than the number of records being retrieved. Having a larger pointer space reduces collisions, but also increases the number of empty buckets being accessed.

The *hash address* is constructed by catenating the hash addresses for all attributes a. The size of the hash address is based on the number of buckets. If one bucket can hold one record, an appropriate length for the hash address is $\log_2 n$. This means that to each of the a attributes $\log_2 n/a$ bits can be allocated.

> For a file with a million records and 10 attributes
> the $\log_2(10^6) = 20$-bit address would allow $20/10 = 2$ bits per attribute.

Using this scheme, the attributes partition the search space only grossly. A retrieval based on only one argument will specify $1/a$ or two bits out of the twenty, and one could expect that one-fourth of the file is to be retrieved. The situation improves quickly as more attributes are specified in the query. When all attributes are specified, the expectation becomes close to one. Figure 4-37 illustrates the case for a nine-attribute query.

APPLICABILITY OF MULTIATTRIBUTE SCHEMES FOR PARTIAL-MATCH RETRIEVAL

The partial-match indexing and hashing schemes presented are not very effective at extreme ranges. We can take advantage of these conditions and modify these schemes, so that the update effort is greatly reduced while much of the benefit is retained.

Small Number of Arguments There seems to be little point in supporting rapid access for queries which lead to retrieval of a large fraction of the file. A heuristic which states: *Search sequentially when looking for more than 20% of the records* remains a valid guideline.

In the early binding cases with many access lists (Fig. 4-36) or combined indexes (Fig. 4-12), update time and storage space can be saved by only entering combinations of several, say, > 3 attributes into the access lists.

In the late binding case illustrated in Fig. 4-37 the retrieval algorithm might as well search sequentially when near exhaustive requests are made. In an interactive environment it is well to have the system respond to a poorly partitioned query with a counter question as: "*Do you really want* `13117` *references on the topic:* `system`?".

Large Number of Arguments The probability that a partial-match query will specify all attributes is slim. It is furthermore expected that, if many search arguments are specified, only a few or no records will respond to a query. There exists hence a further opportunity for reduction of file cost. Certain attribute keys can be placed in a *don't care* category, and not entered in combined lists. When queries are made, the retrieval from the file will fetch some unwanted records. The excess can then be pared down on the basis of the actual key values.

When early binding is used, this again reduces the number of entries in the pointer list; when late binding is used, it provides an opportunity for better use of the pointer list. Fewer combinations of attributes need be stored if for some queries a merge of two or more distinct TID lists is permissible. Then the attributes can be partitioned into these sets so that frequent queries require only one set. Some will need two or more, and an odd partial-match query may need one attribute from each set. The number of stored combinations reduces drastically, as was shown in Eq. 4-9.

Superimposed coding is another traditional technique used to reduce the number of distinct attribute fields. In applications where there are typically many attributes of binary (have or have-not) value, one column may be assigned to more than one attribute. It is hoped that the number of attributes will be such that a retrieval with many arguments will include few unwanted records (false drops). An attribute can also be assigned redundantly to a combination of several columns to avoid false drops due to a single other attribute. The percentage of *false drops* can, in fact, be statistically predicted, given data on attribute patterns in goal records and queries.

The schemes discussed above bring early and late binding multiattribute hashing schemes closer together. The handling of false drops is obviously a late-binding procedure; selection of attributes which can be omitted or superimposed is an early-binding process. There is still active research in this area. It is always difficult to balance rapid response to complex queries with efficient update and use of storage.

4-7 OPTIONS OF THE COMPLEX RING ORGANIZATION

In Sec. 3-6 we presented the basic ring-structure organization. Implementations of file systems based on this structure offer a number of options which allow better performance at some increase in complexity. Such options can be selected to apply to those rings which play a crucial role in the performance of the file. We will find language specifications for such options in Chaps. 8 and 9 when the CODASYL database schema language is presented.

4-7-1 Prior Pointers

In the serial chain and ring structures, pointers may be kept that refer to the preceding record. The availability of such a *prior pointer* simplifies certain search and update operations considerably. Prior pointers allow spacing backward through a ring, so that the predecessor record is found using one access rather than using $y-1$ accesses through the ring. If an open-chain structure is used, access to predecessors is impossible unless prior pointers or equivalent mechanisms are available.

An example of a query requiring such a predecessor search could be in a factory where the goal is a list of five machines adequate at a lower level of production for the task now assigned to machine `B`. The search finds machine `B` according to the `task` chain, and a predecessor search is needed now since the `machine` chain is sorted in order of increasing capability.

Update is greatly affected by the existence of prior pointers. Figure 4-38*(a)* shows a part of a ring structure ("`Equipment`") which has been entered from another ring sequence ("`Supplier=Earthworm`") at record `Cat`. At this point a successor record is to be inserted. In order to add the successor record `Cat2` into the ring, the pointer address to record `Dozer` from the original record `Cat` is inserted into `Cat2`, and subsequently a pointer to `Cat2` is inserted into `Cat`. No other record accesses are required in this case.

If, however, a new record `Grader2` is to be inserted in the *preceding* position (`Grader2` is to be put in service before `Grader`) and no prior pointer exists, the update becomes costly. All the records of the ring will have to be read, since record `Dozer` will have to be modified to point to the record being inserted. The availability of prior pointers, shown in Fig. 4-38*(b)*, speeds this update process but adds some complexity to all update processes. General use of prior pointers also affects storage requirements and reduces locality.

4-7-2 Pointers to the Master Record

Not all the needed data for a query may be found in a record of a ring structure. It may be neccessary to determine to which specific ring this record belongs, and then locate its header or master record for data pertaining to all members of the ring.

An example could be a case where a specific `personnel` record was found during a search for a certain `skill`. This record can be interpreted by establishing its category, but to determine in which `department` the `employee` belongs, the chain has to be followed to its header.

In order to accelerate such searches, pointers may be kept in the member records which refer back to the master record. When a record of a ring has been accessed, first the record category is determined in order to be able to interpret the data and pointer types stored in the record. This is achieved by identifying each record with a category code. All rings of the same type (e.g., `personnel` at various `locations`) will have records of the same record category.

If a master pointer exists, the master record can be quickly accessed. Use of master pointers allows efficient use of storage, since all master-dependent information can be stored only in the master record. The maintenance of master pointers is relatively easy, since both predecessor and successor records in the ring contain the same information. There is, of course, again an additional space requirement.

Typical master-record types seen are `Department`, `Supplier`, `Equipment`. If totals are maintained on a departmental level, relevant changes in a member `employee` record, such as a `salary` increase, can be reflected easily in the departmental total `payroll` field.

Some implementers let master pointers share space with prior pointers. The result has been called a *coral ring* and is illustrated also in Fig. 4-39. Alternate records have prior and master pointers. Locating the predecessor record now requires one or two accesses, and locating the header record also requires an average of 1.5 accesses.

4-7-3 Master Records with Indexes

If the need exists to quickly find individual member records in specific rings, indexes may be associated with master records. The resulting substructures have been discussed in the preceding sections. Since records of a a ring tend to be similar, and since rings are optimal if they are not very long, simple index structures will suffice. The relative benefits of indexes increases if individual rings have poor locality. Indexes become more interesting if they are not restricted to the ring hierarchy. In the personnel file example, good use could be made of an index by employee name to permit access to employee records without using department rings.

4-7-4 Rings and Direct Access

It is possible to combine a direct-access mechanism with a complex ring structure, since the ring structure makes no demands on the location (value of its own address) of a member record, whereas the direct-access methods are heavily dependent on them.

The combination allows rapid fetching of a record and subsequent collection of multiple records which satisfy a request for a data subset by following the chain for this specific response.

In order to reach the header record of a ring, a standard identifying term can be used. Figure 4-40 shows such an access. Instead of name of the individual, the standard term "`ALL`" is given and combined with the class description "`Fire`

Brigade". The pointer for this combination will lead to the header record for the Fire Brigade, so that the whole ring is available for processing.

Since the transformation controls the placement of the record, only one direct-access path can be provided per record. For instance, in Fig. 4-40 "John" can be found directly as Fire Brigade member #14, but he cannot also be accessed directly using his social security number. The ability to control placement of ring members to increase locality has also been lost.

to 333pt4-41A ring file with direct access to multiple rings via a common pointer-list.

RINGS WITH A POINTER-LIST AND DIRECT ACCESS The addition of indirection via pointers, as introduced in Sec. 4-6-2, can remove the record placement restrictions at additional access cost. The use of a pointer-list, as sketched in Fig. 4-33, is applied to the previous problem as shown in Fig. 4-41. Now multiple indirect paths to the goal records are possible.

There now also is freedom in record placement, which can be exploited to improve the locality of rings, so that retrieval of members will be fast.

The collision-handling mechanism can take several forms. Open addressing was preferred when records were not linked. In the ring structure, additional pointer chains can be considered but may require much maintenance.

A compromise using open addressing is to use the pointer-list for file-type verification and the goal record for record category and key identification. No key field is provided in the pointer-list. This becomes feasible when the pointer-list is shared by many rings. Most collisions will be recognized in the list, and can be resolved there. A collision which is detected when the goal record is accessed will require a continuation of the open addressing search through the pointer-list. The structures defined in such a system lend themselves well to some of the more formal information-retrieval methodologies. We will present a generalized view of such access in the next section.

4-7-5 Multiattribute Ring Access

In a ring structure with intersecting rings, as symbolically presented in Fig. 4-42, the records of the ring category "Available_skills" represent the *association* of "Employees" and "Skills". Let the number of Employees be ne and the total number of distinct Skills be ns. The associative level can be viewed either as ns rings, one for every skill, with ring sizes according to the number of employees which have that skill, $ne(\texttt{s})$; or as ne rings, one for every employee with ring sizes according to the number of skills which an employee has $ns(\texttt{e})$. The number of records in the association Available_skills na is then

$$na = \sum^{ns} ns(\texttt{e}) = \sum^{ne} ne(\texttt{s}) \qquad \text{4-17}$$

If rapid access to the individual members of the association is desired, direct access can be provided using as key the catenation of the two parent values.

There can be multiple associations between two parent types; in the figure a second association is called Training. In general, there can be any number of associations between two sets of data elements. In order to generalize this aspect, a third parent, say Expertise_Level, can be considered to be the header of the set {Training, Available_Skill, Craftsman, Teacher, ...}, that is, all the associations made between Employees and Skills. The three-way symmetric concept is indicated in Fig. 4-43. The rings for which this parent provides the header records will link all records for Training, Available_Skill, Craftsman, Teacher, etc.

IMPLEMENTATION OF A FUNDAMENTAL DATA STRUCTURE The associative structure of three elements represents the basic unit to carry information. It also forms a minimal record as defined in Sec. 3-1. To identify a record in such an association, three arguments are required; in a programming language implementation of such associations (LEAP-type data structures) these three are called 4-6-4 231 ¡par.4, line 4¿ ... has to balanced ==¿ ... has to be balanced ¡¿

4-7-1 232+ ¡par.2, line 1¿ . Figure 4-38 ... ==¿ . Figure 4-38(a) ... ¡¿

4-7-1 232+ ¡par.3, line 4¿ ... prior pointers speeds ==¿ ... prior pointers, shown in Fig. 4-38(b), speeds ¡¿

Object, *Attribute*, and *Value*, as shown in Table 4-4.

Table 4-4 General Associations in LEAP

Name	or	Sym	Function	Example
Object	or	O	Key	`Employee`
Attribute	or	A	Goal-attribute name	`Skill_Type`
Value	or	V	Goal-attribute value	`Expertise_Level, Training, etc.`

108pt Figure space

Figure 4-43 Generalized associative structure.

Given this structure, seven type of questions can be asked as in Table 4-5.

Table 4-5 Fundamental Queries in Data Triplets

#	Format	Example based on Fig. 4-42
1	A (O)= V	`Does Jones have Welding as an Available_skill?`
2	A (?)= V	`Which Employees have Welding as an Available_skill?`
3	A (?)= ?	`Which Employees relate to Welding and how expertly?`
4	A (O)= ?	`Does Jones relate to Welding and how expertly?`
5	? (?)= V	`What Employees have which Skills Available?`
6	? (O)= V	`Jones has which Skills Available?`
7	? (O)= ?	`To which Skills is Jones related and how expertly?`

The first query can have only the response `true` or `false`. Multiple responses will be given, in general, for the other queries; for instance, the last query (**7**) may have as response three triplets:

```
Welding(Jones)   = Available_skill
Machining(Jones) = Training
Painting(Jones)  = Available_skill
```

If semantic considerations are disregarded, it can be seen that any of the three fields can take on any of the three roles.

These structures are used in applications where complex relationships have to be accessed rapidly, for instance, in graphics and in robot operation. When rings have to be followed in order to retrieve a subset, there is the possibility that the rings will cross many block boundaries, so that the objective of rapid access is not achieved. Good locality can be achieved, as presented in Sec. 3-6, for all rings relative to one parent by placing all member records of a ring within a block. If this is done in respect to "`Skill_type`", then queries of type **1**, **2**, **3**, and **4** can be answered rapidly. Direct access can identify the block for a specific argument value of `A`, and then the position within the block can be obtained by in-core direct accessing for `O` or `V` to find headers for these subrings, or by direct in-core accessing using the catenation of `O&V` to locate individual records.

Implementations of LEAP-type structures have used redundancy in order to provide equivalent access for queries with unknown values of `A`; a copy of the file is organized based on `O` or `V`. The hash for query type **1** does not have to be supported in the second file. These structures have not had goal attributes appended to their records, but this is, of course, possible. The goal parts may be best kept remote, so that the locality of the primary access structure remains high. Collisions also have to be resolved appropriately; this is achieved by use of additional chains from header and member records.

4-8 FILES USING VIRTUAL STORAGE

Most modern computers permit the use of *virtual addressing*, an address space that is much larger than the actual primary or core storage available. A mechanism consisting of both hardware and software components is provided to allow references to be made throughout the entire space.

This approach is referred to as a demand paging environment or a *virtual-storage* system. We presented the choices of operating systems in Sec. 1-7.

to 378pt 4-44 Accessing a virtual storage file.

There will be a division of the address space into working units called pages. A *page* is identified by the high-order digits of the address and will be a few thousand characters in length. A hardware table, possibly with a software extension, will record which pages from the entire address space are actually represented in core storage at any given moment. A page-not-present interrupt occurs when a referenced address falls into a page not currently in core memory. When this happens, a system software mechanism is invoked to find or to create free space in core memory, fetch the required page from drum or disk, update the page table, and resume execution of the interrupted task, as shown in Fig. 4-44. The page size of a virtual system will, in general, define the effective block size of the file system.

If enough address space exists to handle the size of expected data files, virtual addressing provides a very convenient mechanism for handling files. Systems have been written that perform key-to-address transformations into this data space, and use these system facilities for all further page transfers in and out of core memory. A collision-resolution mechanism remains necessary.

Two types of virtual systems can be found: those which have a single address space, and those which employ multiple named address spaces, called *segments*. The use of a single address space erases any formal distinction between the area used for programs, the area used for program data, and the area used for files.

Boundaries between these areas are set up by convention. Such a convention could be that a file will consist of the space between two address values of α_1 and α_2. Reliability can be compromised if the conventions are difficult to enforce.

More sophisticated virtual systems will provide the capability for several separately named address spaces, called segments. Figure 4-45 sketches this approach, but details of the paging mechanism are omitted here.

A sequence of high-order bits of the hardware virtual address is used to pick up an entry from a segment table. One or more segments can be viewed as files, and the segment entry may have associated with it information regarding protection and sharing of such a file. The allocation of segment pages to disk storage and their movement to core when needed is again an operating-system function. Such a file still presents only a linear address blocked into pages, and no assistance is provided by the operating system to map records into this space. In order to better manage the virtual-address system, some operating systems, such as TENEX and TOPS-20 for the DEC-10/20 series of computers map only those file pages that have been referenced into the address space. Multiple processes can share an available file page. When the process using the files terminates, the file pages will be rewritten, if they were modified, and then released.

The use pattern associated with files may not have been considered by the designers of the virtual-memory system, who were concerned mainly with the manipulation of program sections and data referenced directly by these programs. Paging systems are especially effective if memory references, during time intervals commensurate with the time required to move a page to core, cluster within a limited number of pages.

To use a virtual-access mechanism effectively, it remains desirable to be aware of the page size and page boundaries and utilize pages densely. Blocks will be aligned to page boundaries to avoid accessing two pages for a single block request. The storage density can be very low if the file is not carefully allocated to the virtual address space. No matter how little is stored in a page, the entire page will be kept in the backup storage on the disk. Completely empty pages may or may not occupy file space, depending on the sophistication of the support system. The techniques used for blocking remain valid, since the same density considerations apply.

to 351pt 4-45 Segmented virtual addressing.

Sequential files, with fixed records, are easily mapped densely to a virtual address space, since we do not expect dynamic update. The performance of files using explicit block read access and virtual direct access will be similar, except that there will be no buffering to overlap computation with block reading in virtual systems. Paging facilities do provide good buffering when writing blocks, so that rewrite delays can be neglected except in terms of total system load.

Indexing is very desirable when using virtual memories, since it reduces the number of pages that have to be read. The pointer obtained from an index entry is an address in the virtual space. Only the high-order bits are kept and extended for reference by catenating $\log_2 pagesize$ zeros.

The use of direct-file approaches is simplified by the large address space, although effective space utilization has to be considered. The file storage density can be very low if the file is not carefully allocated to the virtual address space. The performance will be identical for similar values of m and Bfr. Open addressing with linear searching can make good use of any pages which have been brought in due to prefetching in systems which attempt high-performance page scheduling for program usage.

In systems with a single address space, shared use of the space by multiple files using a common transformation simplifies control of storage allocation. A marking convention using computable positions is required to indicate empty versus assigned record spaces. With ring structures locality remains a concern if excessive paging load is to be avoided when rings are being traversed.

We find then that the use of virtual storage does not change the approach to file design greatly, except that it does provide an alternative for the lowest level of file management. A trivial file system, as described in Sec. 4-1-1, is completely replaceable by virtual-storage access. Some database systems have been built directly on top of such virtual storage system, but these systems have incorporated within themselves one or more of the file-organization techniques presented in order to control locality while paging. A number of database systems for demonstration and educational purposes have not considered how to use the paging mechanisms effectively, since the implementers did not expect to operate in a production environment.

An evaluation of the performance of files based on paging can be obtained by combining data on file behavior in terms of block reads and writes with measurements regarding paging-system performance and available workspace size. Some paging algorithms assign priorities according to computer time to paging rate ratios. Whether this will penalize or benefit file-oriented programs depends on the objectives of the system administrators who set the parameters.

4-9 PHANTOM FILES

The key information in an index duplicates information in the file records. If this information is not needed as goal data when the record is accessed via other paths, the indexed attributes do not have to be kept in the goal records themselves. This data may still be stored on an inactive backup file so that it remains available for periodic regeneration of the index. If many fields are handled in this manner, a large portion of the goal file will not exist in active storage. We will refer to data which is used only for accessing, but is not available for result generation as *phantom* data.

Requests for goal data using multiple attributes can be resolved completely within the indexes by matching addresses of the goal records. Such an application is shown in Fig. 4-46, which is an example taken from a file system designed for a casting agency selecting models for TV advertisements.

If all goal data are kept only in indexed form, the entire file has become a phantom file. Files approaching this design have been called fully inverted files in the literature.

Another case of a phantom file can be found in bibliographic indexing. Searches through indexes are made using logical combinations of request parameters which in the end produce a set of call numbers for books. The call numbers are equivalent to the TIDs of the goal data. The goals are the books on the shelves of the library. A library which would always be accessed in this manner could store its books according to considerations other than the sequentiality of subject codes. If, instead of call numbers, the system would provide the actual shelf number and position, then books could be stored according to size, to minimize space used, or according to frequency of access, to minimize retrieval time. Browsing in such a library would be less productive but more surprising. Currently such optimizations are, in practice, restricted to computer files.

To be expanded to 396 pt for Figure space
HERE

Figure 4-46 Indexes to a phantom or manual file.

BACKGROUND AND REFERENCES

Ingenious combinations of basic file methods have been part of most file-oriented applications. Descriptions of the techniques used can often be found in user-oriented system descriptions. The lack of a taxonomy of file organization methods has made analysis and comparison of complex approaches difficult. Techniques have been used, forgotten, reinvented, and sometimes applied in inappropriate environments, so that it becomes very difficult to trace the history of this subject area. A multi-index file system would be a useful tool for someone attempting such a review.

The simple file systems presented in Sec. 4-1 are commonly provided in systems not oriented toward data-processing as FORTRAN (Brainerd[78]), minicomputer, and small timeshared systems. Their inflexibility is sometimes hidden in footnotes: "Records are of variable-length, from 1 to 2048 characters (* Users are always charged for 2048 characters per record)". The reason for the surprising charging rate is that the system uses one block per record.

Descriptions of simple systems are not often found in the regular literature, but user manuals often provide enough detail that the structure and hence their performance in various situations becomes obvious. Access to textfiles is described in Reitman[69],Frey[71], and Lefkowitz[69] presents a variety of index-organization techniques. Index abbreviation is analyzed by Hu[71] and Wagner[73]. Transposed files are used by Easton[69], Selbman[74], Weyl[75] and analyzed by Batory[79].

Tries were described by Fredking[60]. Burkhard in Kerr[75] and Litwin[81T] look at tries and direct access. The tradeoffs in blocking factors are analyzed by March[77]. Yao[77] and Whang[81] evaluated formulas to estimate block accesses and found most wanting; they are important for optimizers in database systems. Combinations of operations are estimated by Demolombe[80].

An early system using multiple indexes, TDMS, is described by Bleier[68]; Bobrow in Rustin[72] develops this approach. O'Connell[71] describes use of multiple indexes. Optimization and selection of multiple indexes is considered by Lum[71], Stonebraker[74], Shneiderman[74], Schkolnick[75], Yao[77], Anderson[77], Comer[78], Hammer[79], and Whang[81].

Multiattribute access via indexes is evaluated by Lum[70], Mullin[71], and Schkolnick in Kerr[75]. Ray-Chaudhuri[68], Bose[69], Shneiderman[77], and Chang[81] develop strategies of combinatorial index organization. Ghosh[76] includes construction of optimal record sequences for known queries.

Detailed specifications regarding VSAM can be found in various IBM manuals. A description of its design is given by Wagner[73], and its storage allocation is discussed by Chin in Kerr[75] and optimized by Schkolnick[77]. Reorganization intervals are estimated by Maruyama[76]. This file system is becoming the basis for much database development where IBM equipment is used.

Landauer[63] and Sussengut[63] describe trees for file systems. Many B-tree papers cited in Chap. 3 describe their use for data files. McCreight[77] includes the blocking of variable length records.

Binary trees were described by Hibbard[62] and analyzed by Arora[69], and Overholt[73]. Bentley[75] extends a binary tree to deal with multiple attributes. The balancing of binary trees has had much attention. Adel'son-Vel'skiĭ and Landis presented a basic algorithm (AVL-trees) in 1962. An extensive analysis is in Knuth[73]. Baer[75], Nievergelt[74], and Bayer[76] consider access frequency. The SPIRES system is presented by Martin in Nance[74] and by Schroeder in Kerr[75].

Hierarchical structures are found in many applications. Examples are found in Connors[66] and Benner[67], include Davis[70] and Booth[60], and many more are cited in

Senko[77].

Descriptions of MUMPS can be found in Allen[66] and in Greenes[69]. The MUMPS system was developed at Massachusetts General Hospital to support computing applications in medicine. The general system design was based on one of the earliest and best time-sharing systems: JOSS. The file system was designed to match specifically the hierarchical organization of data perceived in the area of medical record keeping, as seen by the developers at MGH and General Electric in their FILECOMP system which preceded the MUMPS implementation. Modern implementations use B-trees. The standard (O'Neill[76]) is now available on many computers.

New hashing techniques are analyzed by Scholl[81]. Mullin[72] applies hashing to indexed sequential overflow. Chains were proposed and analyzed by Johnson[61] to provide multiattribute access to a direct file. Dimsdale[73] presents details of a system using direct access to pointer lists. Wong[71], Files[69], Burkhard[76,79], Chang[82], and Rivest[76] present multi-attribute hashing schemes from early to late binding. Before proposed techniques can be adopted, it is necessary to verify that they are applicable to the scale of the intended application. Some interesting techniques require exponentially increasing resources as the file size becomes greater.

Systems developed by Bachman[66], Dodd[66], and Kaiman[73] provided direct access to rings for various applications. Inglis[74] investigates maintenance of indexes using rings. Härder[78] uses both indexes and chains for a DBMS. The LEAP system is described in Feldman[69,72]. This system has spawned further interest (Ash[68], Symonds[68]).

Daley[65] presents the virtual systems approach to files; its application is described by Rothnie[74]. Denning[71] compares the objectives of storage and memory systems. Hatfield in Freiberger[72] evaluates locality in paged systems. Bibliographic systems as described by Lancaster[79] have phantom goal data.

EXERCISES

1 Determine the optimal bucket size for a text file structured as described in Sec. 4-1-3. The file is to be used for storing text lines of 70 characters and has a block size of 256 characters. Do not consider more than four blocks per key group. The lines are numbered sequentially 1, 2, 3, ...

a In terms of minimal file size.

b In terms of lowest block-access cost for the sum of 10 exhaustive reads of the file and 1000 random accesses of one line each. Assume a file of 1000 lines. Do not consider more than 4 blocks per bucket.

2 For the transposed file in Fig. 4-4 write a program to find the average man and compute the program's expected performance.

3 Apply the various techniques of key abbreviation discussed in Sec. 4-2-1 on all the blocks of Fig. 4-7.

4 Devise a method for implicit high-order key-segment deletion. Illustrate the effect on the example given in Fig. 4-6.

5 Write a program to produce full keys given the abbreviated text of Fig. 4-7, starting at the beginning of an index block.

6 Write a similar program, but allow it to start at any attribute pair of Example 4-3.

7 Estimate the average search time for a record in a multi-indexed file according to one search key using:

a Conventional multi-indexed organization

b Abbreviated keys

c Tree-structured files

Parameters for the evaluation are :

Block size	B	2000 characters	Full key size	V	20 characters
Record size	R	200 "	Pointer size	P	8 "
Number of records	n	100 000			
Abbreviation factor for keys		50%			
Time to seek and read one block	T_B	50 milliseconds			

Assume a uniform key distribution. Document all other assumptions.

8 Describe the difference between an indexed and a tree-structured file.

9 Which file structure provides the best description for the following structures found in our environment? Justify the decision in one or two sentences.

a Genealogical chart

b Telephone book, white pages (people)

c Telephone book, yellow pages (services)

d Telephone book, green pages (synonyms of service names)

e Organization chart

f Manual medical records

g Railroad classification yard

h Airline terminal

i Social security office central files

10 Samuel Blockdrawer has been assigned to a group which is designing a regional database application for health records. The known requirements are the following:

- A large population (3.5 million).
- Records of variable length.
- Retrieval of patient records by name as well as by medical registration number. The medical registration number can be assigned at the first patient contact.

Occasionally, the database will be used for research purposes; however, most of its function is the individual health record.

A patient identification record will occupy at least 150 bytes, at most 280 bytes, with an average of 220 bytes. It contains information such as name, age, patient registration number, and so on. The average person has about three contacts with the health care delivery system per year, distributed as follows:

- Two visits to a pharmacy, for an average of 1.2 drugs; each drug record is 20 bytes long.
- 0.72 visits to a physician, each for a single complaint; the average visit generates 50 bytes.
- 0.1 hospitalization which generates an average of 500 bytes.
- 0.01 major hospitalization which is summarized to 2000 bytes.

The proposed computer system will use five byte pointers, four bytes each for data attribute descriptions and for the data values (names will be compressed to this length). Blocks are 7000 bytes long and the interblock gaps are equivalent to 500 bytes. The average seek time is 30 ms. The rotational latency is 20 ms. The advertised data-transfer rate is 1 million bytes per second. There are six blocks per track and 50 tracks per cylinder.

Sam proposes a design as shown in Fig. 4-47.
to 333pt4-47Regional health record system.
Questions:
Provide the appropriate formula and its values at the end of the first year, second year, and fifth year for each of the seven aspects of the file design which are listed below. State any assumption which you have to make in order to obtain the results.

a The average record size in the system as well as for the total file size.

b The time required to locate one specific patient visit by name and date.

c Time to find the next visit of that patient.

d The time required to add a visit record.

e The time to change a data value in a patient's record, if the registration number is given.

f The time required to collect the entire patient's record.

g The time required to collect all usage data for one specific drug.

Give three suggestions how this design can be improved and mention for each suggestion both the benefit and the cost considerations. How can we handle people that move out of the region or die?

11 Develop an algorithm to generate the minimum number of combinatorial indexes in the general case (see Fig. 4-13). The author is not aware of such an algorithm and would appreciate communications regarding a solution.

12 A supplier's manual of MUMPS provides the following design example:

We wish to store patients' identifications, each of 60 characters, for 10 000 patients numbered from 1 to 10 000. The blocks have a net capacity of 762 characters. Each combination of segment-type identification and key requires three character spaces, a pointer also requires three, and the end-of-string character one space. Due to the design of the machine, segments have to be multiples of three characters.

a How many segments fit into one block if all segments are on one level?

b What is the number of blocks used?

c What is the expected search time exclusive of the directory search?

13 We can get faster access in the above case by using segments consisting of a pointer only on level $x-1$ and keeping the patient identification on level $x-2$.

a How many segments will be in one block on level 1?

b How many segments will be in one block on level 2?

c How many blocks are being used?

d What is the expected search time?

14 Do you see any problem if the segment length would not always be 60 characters, but rather 60 characters on the average? (This problem will be refined and solved in Example 6-2.)

15 Design a file and evaluate it for questions as are posed in the epigraph from Vonnegut for Chap. 12.

16 Prove the statement in Sec. 4-2-2, that the number of empty pointer fields in binary index trees is $y+1$ (Fig. 4-8).

17 To reduce the number of entries in a multi-attribute direct-access procedure only queries which catenate at least three of the four key attributes are accepted. How many entries will be required in the pointer-list?

18 How many search arguments should you have before using direct access to the file of Fig. 4-39?

19 Programming lore advises that input should be sorted in the descending order when doing batch updates of files which keep overflows in ascending ordered chains. Why? Quantify the effect.

20 Compare a MUMPS segment with a LEAP record.