Chapter 3

Basic File-System Organization

This thinker observed that all the books no matter how diverse they might be, are made up out of the same elements: the space, the period, the comma, the twenty-two letters of the alphabet. He also alleged the fact which travellers have confirmed: In the vast Library there are no two identical books. From these two incontrovertible premises he deducted that the Library is total and that its shelves register all the possible combinations of the twenty-odd orthographical symbols (a number which, though extremely vast, is not infinite): in other words, all that is given to express, in all languages. Everything: the minutely detailed history of the future, the archangels' autobiographies, the faithful catalogue of the Library, thousands and thousands of false catalogues, the demonstration of fallacy of the true catalogue, ...

Jorge Luis Borges
from "The Library of Babel", *a story in the "Labyrinths"*

3-0 INTRODUCTION

The primary features needed in systems that store large amounts of data are fast access for retrieval, convenient update, and economy of storage. Important secondary criteria are the capability to represent real world information structures, reliability, protection of privacy, and the maintenance of integrity. All these criteria tend to conflict with each other. The choice of the file organization method determines the relative suitability of a system according to all these criteria. We will initially evaluate files according to the basic criteria. A good match of the capabilities provided by the file system to priorities assigned to the criteria, as determined by the objectives for the database, is vital to the success of the resulting system. The number of possible alternatives for the organization of files is nearly unlimited. To make the

selection process manageable, we will describe and evaluate *fundamental* file-design alternatives. Most structures used in practice either fall into one of these categories or can be reduced to combinations of these methods. These fundamental types are closely related to systems in actual use. The collection of these six does not represent a mathematically minimal set of independent file organization methods. The selected methods are also reasonable in terms of the secondary criteria, i.e., they are fairly reliable and easy to analyze and implement.

The six fundamental types of files are covered in the chapters as follows:

Sections 3-1 and 3-2: Two basic file types, without ancillary access structures:
3-1 The pile 3-2 The sequential file

Sections 3-3 1ne 3-4: Two types using indexes to aid access :
3-3 The indexed-sequential file 3-4 The multiply indexed file

Sections 3-5 and 3-6: Two further types using computer-specific methods:
3-5 The direct file 3-6 The multiring file.

Section 3-7: The principles and performance of sorting files.

Section 3-8: A summarization of the concepts introduced in this (big) chapter.

The sections follow a consistent organization: After a general introduction, the initial subsection describes the organization, the next one its use, and the third subsection evaluates its performance. This chapter begins with a section on general performance parameters.

The design of databases requires analysis for the prediction of performance. The formulas developed for the a performance evaluation use engineering-style approximations: They concentrate on first-order effects. The formulas are best viewed as concise descriptions of the behavior of the files under normal conditions. The notion used in the formulas is consistent throughout the text; the inside covers or Appendix B can be used to find the significance of the abbreviations employed.

THE METHOD OF ANALYSIS For the performance analyses a variety of approaches will be used. The general approach is to consider the required hardware functions step by step, consider the values of the computing times, c; any seeks, s; rotational latencies, r; and the transfer rates, t or t', applied to the blocks. Often functions must be executed repetitively. The sum of the values, T, provides the estimate for the cost in terms of time required for a file operation.

When alternative functions are required within an operation, typically dependent on the locality of the data, we have to resort to case analysis. For each choice the probability of using one function on the other is estimated. The expected value is found by taking the sum of the probabilities times the cost of each of the k cases, i.e., $T_e = \sum p_i T_i \,|\, i = 1 \ldots k$, as illustrated in Eq. 3-40. To simplify the analysis some assumptions are made. We assume often typical usage ratios of reading versus writing, balanced speed ratios of system components, and commonly seen user behavior. Because of variance in these factors the results will be of typical *engineering accuracy*, say $\pm 20\%$. This precision is more than adequate for predicting software performance; today, many implementors only guess the performance and results often differ by orders of magnitude from the expected value. As always, any assumptions should be noted during a design process and later verified, so that unpleasant surprises are avoided.

The consistency of the formulas permits the comparison of expected performance for the fundamental file organizations presented here. A final comparative rating of performance characteristics for the six types of files is given in the introduction to Chap. 4. ######

We now introduce general concepts of file organizations and their performance. These concepts will apply throughout, but will be specifically stressed in this chapter.

A file will have some directory information associated with it, but the bulk of the file contains data records. The analyses consider only the data portion of the files. The data portion will follow certain design rules, as defined by the file organization chosen. The previous definition of a file can now be expanded to state

> A file consists of similar records and has a consistent organization.

3-0-1 File Directories

Associated with a file is a directory. The directory contains information describing the owner, the position, and the format of the records comprising the file. Different types of file organization put different requirements on the contents of the file directory. Much of the information kept in the directory is associated with storage allocation and is managed by the operating system.

The directory information also is kept in a file, since it must be persistent as well. The owner of this file is the operating system itself, although it may permit read-access by others, for instance by the file management programs. If multiple computers are used, the directory may be partially replicated in all computers so that the right computer can be selected if a particular file is needed.

Table 3-0 Elements of a file directory.

Directory field	Example
File name	`Payroll`
Owner	`"Joe Paymaster"`
Access group	`Payroll_dep`
Create-date	`25DEC85`
Device-type	`diskdrive`
Device-name	`disk5`
Begin-point	`T12B3.000`
Organization	`ISAM`
Max.size	`60`
Size allocated	`40`
Size used	`35.3`
End-of-file pointer	`T23B2.234`
Last-used date	`1JAN86`
...	...

Content of the Directory We show directory information, as typically kept for each file, in Table 3-0. The names of the owners and authorized users for the file are coded to match the access authorization mechanism for the computer system. More on users and their access rights is presented in Chap. 12-2.

The allocation information specifies where and how much storage is allocated to a file. The precise content and organization of a file directory varies greatly among operating systems. Some of the information may be kept in a header record associated with the file rather than in the directory itself. The important elements for access to a file are the name, the begin point, its size or end point, and its organization or structure.

The operating system mainly provides a space with a name attached. Within this named space individual pages or blocks can be addressed, read, and stored. Most operating systems permit files to grow, so that the size can vary over time. Methods used by the operating system to keep track of the storage allocated to a file are presented in Chap. 6-5-1.

Accessing the Directory A practical procedure to deal with directory records is to read these records once, when a computation begins using the file, and retain the information in memory for further reference. Acquisition of directory information is performed during *opening* of a file.

When *closing* the file, the file directory will be updated if changes in the file have occurred that should be reflected in the directory. System "crashes" can damage files by failing to update directory records. Data recently stored may then not be accessible.

3-0-2 File Description and Use

For each fundamental file organization we provide a description and the type of application it is suitable for. In these descriptions definitions will be encountered that are relevant for the subsequent file organization methods.

Some general criteria for organizing data are

1 Little redundancy
2 Rapid access
3 Ease of update
4 Simple maintenance
5 High reliability

These criteria may conflict, and different file organizations will have a different balance of satisfying these criteria. Reliability is the most complex of these issues and is treated separately in Chap. 11.

For economy of storage, we want to store data with little *redundancy*. Redundancy exists when data fields are duplicated in multiple records. Redundancy of data elements also increases the effort required when values of data elements have to be changed, since we expect all copies of a data element to present a consistent description of the world. Redundancy can be useful to provide rapid access to data; it is in fact the primary tool used to increase performance. For instance, in Chap. 5 we will use indexes, which repeat terms used in the file, to rapidly access records containing these terms, just as the index to this book is meant to be used.

Table 3-1 A dense, a sparse, and a redundant file.

	Student No.	Class	Credits	Incom-pletes	Current work	Age	Grade type
1	721	Soph	43	5	12	20	PF
2	843	Soph	51	0	15	21	Reg
3	1019	Fresh	25	2	12	19	Reg
4	1021	Fresh	26	0	12	19	Reg
5	1027	Fresh	28	0	13	18	Reg
6	1028	Fresh	24	3	12	19	PF
7	1029	Fresh	25	0	15	19	Reg
8	1031	Fresh	15	8	12	20	Aud
9	1033	Empl	23	0	14	19	PF
10	1034	Fresh	20	3	10	19	Reg

(a) A dense file: Database course attendees

	Student No.	Courses taken					Exp. years
		CS101	CS102	Bus3	EE5	IE101	
1	721	F72	F73		W73		
2	843	F72	W73				
3	1019		S72	S73			1
4	1021		S72			F73	
5	1027	F73		S73			
6	1028				W73		1
7	1029	F73	W73				
8	1031	F73					
9	1033						3
10	1034					F73	

(b) A sparse file: Database course attendees

	Pre-requisite	Student No.	When taken	Years exp.	Acc. credits	...	Grade type
1	CS102	721	F73		43	...	PF
2	CS102	843	W73		51	...	Reg
3	CS102	1019	S72		25	...	Reg
4	CS102	1021	S72		26	...	Reg
5	CS102	1029	W73		25	...	Reg
6	Bus3	1019	S73		25^{3}	...	Reg
7	Bus3	1027	S73		28	...	Reg
8	EE5	721	W73		43^{1}	...	PF
9	EE5	1027	W73		28^{7}	...	Reg
10	IE103	1021	F72		26^{4}	...	Reg
11	IE103	1034	F73		20	...	Reg
12	Exp.	1019		3	25^{3}	...	Reg
13	Exp.	1028		1	24	...	PF
14	Exp.	1033		1	23	...	PF
15	none	1031			20	...	Aud

Note: the superscripts indicate records where this redundant data element already appeared.

(c) A redundant file: Database course attendees

Figure 3-1 shows a dense file, a sparse file, and a redundant file. Each of these files presents information in a form useful for a particular purpose.

3-0-3 File Performance Parameters

Quantitative measures are necessary to evaluate file-system performance. The four types of computations introduced in Chap. 1-2 balance of these computations differs among applications. If we produce information, rather than operate an archive, the `READ` frequencies should exceed the number of updates of files and records by a considerable ratio, perhaps 10:1 or more. Hence we are often willing to use data organizations which provide fast retrieval, even when this makes building and update operations more complex.

We now define six basic file operations which are needed to implement these computations, and for each operation measures of performance will be provided. Storage requirements are measured in terms of bytes, while the cost of the file operations is measured in terms of time they require for execution. In Chap. 5 storage and time requirements are reconciled by using financial measures.

Seven measures will be provided for each of the six fundamental file organization methods. These are:

R : The amount of storage required for a record
T_F : The time needed to fetch an arbitrary record from the file
T_N : The time needed to get the next record within the file
T_I : The time required to update the file by inserting a record
T_U : The time required to update the file by changing a record
T_X : The time needed for exhaustive reading of the entire file
T_Y : The time needed for reorganization of the file

To simplify cross referencing, we use corresponding subsection numbers in this introduction and in the six sections which deal with performance evaluation.

The six operations on files are executed by combining seeks, reads, and writes of blocks, so that the measures to be derived are based on the hardware parameters obtained in the preceding chapter. The use of these parameters provides independence from the physical specifics of the hardware, so that the analysis of file methods can proceed without considering details of the possible hardware implementation.

Decisions to be made in the evaluation of file performance for specific cases are characterized by four questions:

1 Is a seek required, or are we positioned appropriately; i.e., is s to be used?
2 How is the record located; i.e., is the latency 0, r, or $2r$?
3 Are we transferring only data blocks, or are we spacing through a file; i.e., do we use t or t' for the transfer rate?
4 Are we measuring the net quantity of data or the space required; i.e., do we use R or $(R + W)$ as a measure?

For each file organization we have to consider the sum of all these operations.

Parameter 1: Record Size The record stores primarily the data, but may also be used to store descriptive and access information. The files shown in Fig. 3-1 are structured, and the descriptions of the data fields appear only once, in the heading of the files. When data to be stored is more diverse, trying to define all possible fields in the heading will cause many fields to be empty and will lead to large, sparsely filled records. Section 3-1 presents a self-describing organization for records, which is advantageous for heterogeneous data, while Sect. 3-2 will deal with structured files.

Parameter 2: Fetch a Record To be able to use data from a file, a record containing the data has to be read into the memory of a processor. Fetching a record consists of two steps: locating the position of the record and then the actual reading. We use the term *fetch* when the retrieval of the record is *out of the blue*, that is, no operations to prepare for a simpler locate and read sequence have preceded this fetch. To fetch data efficiently, we have to locate the element to be read fast. A simple address computation, similar to the determination of the position of an array element in memory, seems most desirable but leads to inflexibility when the data is not tabular in nature or the entries in the file are sparse. In general, the records of a file cannot be directly located on the basis of a subscript value or record number.

The use of look-up files or indexes helps in fetching data when the position cannot be directly computed, but having such files increases redundancy. Any changes in the main files will require corresponding updates in any indexes. A look-up file, helpful in obtaining access to the previous files, is shown in Table 3-2.

Table 3-2 A look-up table.

Name	Student No.	Prerequisite entries (variable in number)
Allmeyer, John	1031	15
Bushman, Wilde	1028	13
Conte, Mary	843	2
Erickson, Sylvia	1034	11
Gee, Otto	1021	4 10
Heston, Charles	721	1 8
Hotten, Donna	1029	5
Jason, Pete	1027	7 9
Makale, Verna	1019	3 6 12
Punouth, Remington	1033	14

Parameter 3: Get the Next Record Isolated data rarely provides information. Information is mainly generated by relating one fact with another; this implies getting the next record according to some criterion. While `Fetch` can be characterized as an *associative* retrieval of a data element based on a key value, `Get-Next` can be characterized as retrieval using a *structural dependency* . A successor record can be obtained most rapidly when related data is kept together; that is, when the *locality* of these data is strong. Records having locality according to some key are considered to be *clustered*. The records in Table 3-1*a* are clustered by `Student No`.

There may be multiple relationships among data, but their representation is difficult. Only one sequential access sequence or *physical clustering* can be directly accomodated on the physical storage devices in which the data resides. To represent further relationships either redundant storage of data or pointers which link successive records can be employed. The reading or writing of records in an order according to any relationship is called *serial reading*. If one serial access path is simplified by physical ordering of the records, we can read the records following that relationship *sequentially*. Table 3-1*c* shows the alternate use of redundancy to simplify grouping of records that identify the prerequisite courses.

Parameter 4: Insert a Record Most applications require regular insertion of new data records into their files to remain up to date. Writing into a file is more costly than reading the file. In addition to locating and reading the file, the changed block also has to be rewritten. Adding records is easiest if they can be appended to the end of a file, just extending the file. When a record has to be put in a specific place within the file to permit future clustered access, other records may have to be shifted or modified to accommodate the insertion.

When data is stored redundantly, multiple `WRITE` operations will have to be carried out to perform a complete file update operation. If the number of `Credits` of student `1019` has to be changed in the file shown in Table 3-1*a*, only one record is changed but in the file of Table 3-1*c* three operations are required.

Each `WRITE` into a blocked file will require the reading of a block to merge the data from the surrounding records before rewriting the entire block.

‡‡‡‡‡ **Appending Records** Insertions to the end of the file, `APPEND` operations, are frequent. To make them more efficient they are handled differently from other insert operations in some systems. If the address of the last block of the file is known the performance of `APPEND` operations can be improved, since then the locate step is not needed. This address can be kept as a pointer in the directory.

Furthermore, a sequence of `APPEND` operations may be performed without successive `READ` and `REWRITE` operations, since a block needs to be written only when full or when a batch of appends is completed. New blocks, allocated to extend the file, do not have to be read. The relative frequency of encountering a block boundary, which determines the frequency of block writes remaining when appending, is $1/Bfr$. When a single transaction performs only one append, no benefits are derived from batching. Other considerations argue also against batching of appends.

When batching `append` operations an intervening failure of the computer system will cause multiple appends to be lost. If these `APPEND` operations are part of distinct transactions, batching of appends violates the integrity constraints of a transactions if the computer fails. The all-or-nothing consistency condition given for transactions hence requires a `REWRITE` of the block when the transaction is to be committed.

In the analyses that follow we ignore the difference of appends and general insertions and assume that a `READ` and a `REWRITE` of a block is done for every type of insert. For applications which append frequently separate evaluations which consider the benefits of not requiring to locate the record, and perhaps on batching as well, could be made. ‡‡‡‡‡

Deleting Records Deleting records is the inverse of insertion. After locating the record, the space that it occupied should be freed for reuse. Maintenance of clustering may require again that other records be shifted.

Often deletion is not performed immediately, but instead the record to be deleted is marked with an indicator that this record is now invalid. The delete is now converted to an update. A marker may be set in a special field containing {`valid, invalid`}, or the old record space may be filled with `NULL` characters, or a message such as "`* Deleted . . .*;`" is placed in the record. We will refer to such a marker as a *tombstone*. We make sure that the tombstone will fit even within the smallest record permissible in the file, so that rewriting of the old record is easy.

We do not cover the performance of deletion operations explicitly, and treat record deletion either with insertion or with update of records, as appropriate.

Parameter 5: Update a Record All changes of data do not require insertion of a new record. When data within a stored record must be changed, the new, updated record is created using the remaining data from the previous record. The old data and the changes are merged to create a new record, the new record is inserted into the position of the old record within the block, and the block is rewritten into the file.

If the record has grown in size, it may not be possible to use the old position. The old record will then have to be deleted, perhaps with a tombstone, and a new record inserted in an alternate appropriate place. Records in the look-up file shown in Table 3-2 will grow in size if a prerequisite is added. For a variable spanned organization two blocks will have to be read, updated, and rewritten whenever the record spans blocks. Again, the frequency of that event is again $1/Bfr$.

Parameter 6: Read the Entire File Some application functions require the reading of the entire file. Here again we prefer a dense, nonredundant file to avoid excessive time and errors due to the multiple occurrence of the same data item. If this cannot be achieved, the process of exhaustive reading has to maintain additional information either within the file or in separate tables. There is, for instance, no simple way to use the file shown in Table 3-1*c* to count the number of students or to compute the average credits accumulated by them.

Parameter 7: Reorganize the File Finally, it may be necessary to clean up files periodically. Reorganization removes deleted and invalidated records, reclaims the space for new data, and restores clustering. Reorganization is especially important for file organizations which create tombstones for deletion and updates.

The frequency of this operation is not only dependent on the type of file organization used but varies greatly with the application. Reorganization is initiated when a file has had many records inserted and deleted. If records are mainly appended, reorganization is not needed as often. Reorganization has many similarities with the process of "garbage collection" encountered in some computer-language systems which provide dynamic storage allocation.

Summary of the Parameters We have now defined the operations to be considered when evaluating file performance. We will also use the term *retrieve* for both fetch and get-next, and use the term *access* to denote any reference to a file. The word

search is used at times to describe the activity of locating a record prior to the actual read or write of a block.

3-0-4 File and Record Structure

Each of the six file organization methods is now described and analyzed in conjunction with one specific record organization. While the combination presented is common in practice, it should be realized that many other combinations are valid. Such alternative combinations will produce different formulas for the performance parameters. The derivations are sufficiently simple to allow readers to evaluate other alternatives on their own.

Other files used by operating systems, as directory records are not further discussed or considered in the subsequent file analyses. If the operating systems does a poor job of file management the performance of databses and application files is obviously affected; but those topics are covered in other b ooks.

3-1 THE PILE FILE

The initial method presented is a minimal method. This method, the *pile*, is rarely practical but forms a basis for evaluation of more structured methods. Data in a pile is collected in the order that it arrives. It is not analyzed, categorized, or forced to fit field definitions or field sizes. At best, the order of the records may be chronological. Records may be of variable length and need not have similar sets of data elements.

We are all familiar with piles on our desks. They are an easy-to-use form of storage; any information which arrives can be put onto a pile without any processing. They are quickly created, but costly to search.

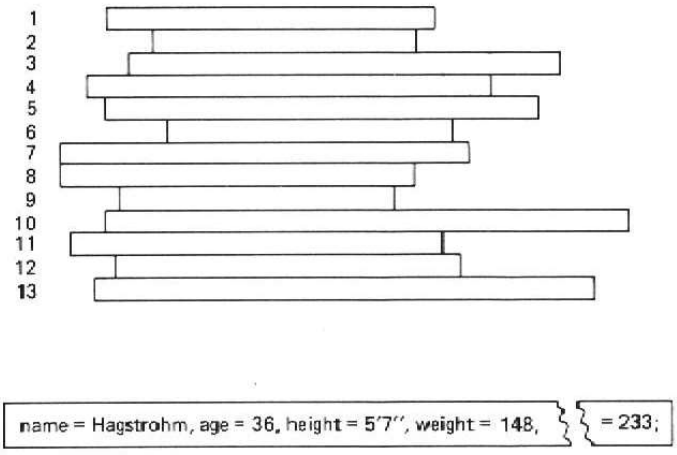


Figure 3-1 A pile file with a representative record.

3-1-1 Structure and Manipulation

Some restrictions, though, will have to be observed to allow for correctness of processing of data. The pile has to be such that we can extract information. A record must be relatable to some object or event in the world. A record should consist of related data elements, and each data element needs to have an identification of its meaning. This identification may be an explicit name for its domain, such as `height` in Example 3-1 or a position which indicates its *attribute* type as the records of Fig. 3-1. For the pile file we assume a sequence of *self-describing fields*, as shown in Example 3-2a. If multiple identical domains exist in a record, the attribute type also specifies the relationship to the object or event described by the record: `height of doorway`, `height of tower`. We will use in pile files an explicit name for the attribute description, since this matches best the unstructured form of the file.

`height = 95`	The value of the data element is `95` and the descriptive name is `height`.

Example 3-1 Self-describing data element.

was Fig F4-4 example 3-1

The set of two entries in Example 3-1 is referred to as an *attribute name-value pair*.

Just one such pair does not make a significant record. We need a number of such pairs to adequately define an object, and then, if we want to associate factual

data regarding this object, we will want to attach to the record additional attribute pairs that contain further data, as shown in Example 3-2a.

Example 3-2a Self-describing data record.

```
name=Hoover,type=Tower,locale=Serra␣Street␣(Stanford),height=95;
```

When information is to be retrieved we select records by specifying some attributes in a *search argument* and retrieve other attributes as the *goal data*. The attributes of a record to be matched to the search argument of the fetch request are the key attributes. The terms *key* and *goal* define two parts of the record for a query. Different requests may specify various combinations of attributes, so that we do not wish to preassign attributes to the role of key or goal data. The key identifies the record wanted in the file, and the goal is defined to be the remainder of the record.

Extent of Goal Data If there are no attribute pairs left in our record beyond those required for the search, the only information retrievable is the fact that this object exists in our files. Finding a match is frequently an adequate goal. With the assumptions used in the example worked out in Example 3-3, the `height` of the tower still is available as a goal-data element.

Complex attributes

An attribute value may in itself be divided into a number of attribute name-value pairs to permit a hierarchical structure within the record, as shown in Example 3-2b. Such a *complex attribute* is difficult to manage for a file system. The handling of the contents of a complex attribute is typically the users' responsibility.

Example 3-2b Complex attribute. #####

```
...,addr.=(place=Serra␣Street,town=Stanford␣U.,
                  county=Santa␣Clara,state=California),...
```

Selectivity The number of attributes required for a search is a function of the *selectivity* of the key attributes. Selectivity is the measure of how many records will be retrieved for a given value of an attribute. When we select a subset of the file according to the value of an attribute, we have partitioned the file into a potential goal set, and a set to be rejected.

Partitioning can take place repeatedly using other attributes, until only the desired goal set is left.

The selectivity can be given as an absolute count or as a ratio. After the first partitioning, we can imagine that we have a file with a smaller number of records. It is important to isolate this subset of records in such a way that it will not be necessary to search all the original data for a match according to the second attribute specified. The second search should be measured in terms of a selectivity ratio, which is applied to the partition of records produced by the previous search.

Partition sizes are computable by computing the product of the file size, n, and the selectivity ratios. If multiple ratios are used we typically assume that

the selectivity of the first search specification is independent of the second search specification. Successive application of all other search specifications narrows down the possibilities until the desired record, or set of records, is obtained. Examples of selectivities are given in Table 3-4.

Example 3-2 Estimation of selectivity.

Let us estimate the selectivity given the record of Example 3-2a. The specification `Tower` applied to a file containing all buildings in this world, or at least all towers, would restrict our search to the 10^7 towers in this world (assumption: one tower per 300 people and $3 \cdot 10^9$ people in this world). The name `Hoover` may have a selectivity ratio of $2.5 \cdot 10^{-6}$ (fraction of `Hoovers` in the world, based on 40 entries in the San Francisco telephone book: $40/8 \times 10^5$, the fraction of the English-speaking population: $1/10$, and the assumption that half the towers are named after people), so that the second search attribute would yield 4 possible records. (Assumption: As many towers are named after Hoovers as after other family names.) A third specification (the street name) should be sufficient to identify a single specific tower, or establish the nonexistence of a tower satisfying the request.

Counting the Attributes Throughout the remainder of this book, we will use the symbol a to denote the total number of attribute types in the file under consideration, and the symbol a' to denote the average number of attributes occurring in a record. If the record in Example 3-2a is representative for the file, then a' for that file would be 4. We typically disregard subsidiary attributes as shown in Example 3-2b since they cannot be searched independently. We do not need to know the value of the total attribute count, a, of an entire file in the pile organization.

3-1-2 Use of Piles

Pile files are found where data is collected prior to processing, where data is not easy to organize, and in some research on file structures. They also provide a basis for performance comparison within this text.

Data banks that have been established for broad intelligence gathering sometimes have this form, since the potential usage of a record is difficult to assess. In this type of application many attribute types which defy a priori compartmentalization may exist. Many manual data collections, such as medical records, also have the form of a pile. In pile files, data analysis can become very costly because of the time required for retrieval of a statistically adequate number of sample records.

Since much of the data collected in real-world situations is in the form of piles we can consider this file organization as the base for other evaluations. If we consider piles to be essentially free, we can estimate processing efforts needed to create more efficient file organizations for retrieval by analysis of the transformation from a pile.

3-1-3 Performance of Piles

We now estimate the seven performance parameter values for a pile organization as described in Sec. 3-1-1. We have already some intuition which parameters will be small and which will be large.

Record Size in a Pile File density in a pile is affected by two factors: negatively, by the need to store the attribute names with the data, and positively, by the fact that nonexistent data need not be considered at all. The effect is a relatively high density when the material collected is heterogeneous or sparse, and a relatively low density when the data is dense and the same attribute names occur redundantly in successive records.

We will denote the average length of the description of an attribute as A, and the average length of the value portion as V. For Examples 3-2a and b the values of A are 5 and of V are 9 bytes. Since name and data are of variable length, two separator characters (`=` and `,` or `;` in the examples above) are stored to *mark* each data element. We have an average of a' fields.

Using these definitions the expected average record length will be

$$R = a'(A + V + 2) \tag{3-1}$$

Appropriate values for a', A, and V will have to be based on an adequately sized sample. Techniques to reduce the values of A and V by encoding are discussed in Chap. 14.

Fetch Record in a Pile The time required to locate a record in a pile is long, since all the records may have to be searched to locate a data item that has a single instance in the file. If it has an equal probability of appearance anywhere in the file, we consider that at least one, and maybe all (b) blocks will have to be read. Using the *big-*$\mathcal{O}$ notation we can state that the T_F operation on a pile-file is $\mathcal{O} = n$.

More precisely, we can compute the expected average time for a record fetch by taking the sum of all the times to reach and read any of the blocks, divided by the number of choices, or

$$\text{Average blocks read} = \sum_{i=1}^{b} \frac{i}{b} = \frac{1}{2}(1 + b) \quad \approx \frac{1}{2}b \qquad \text{if} \qquad b \gg 1 \tag{3-2}$$

The time to read this number of blocks sequentially is then, using the notion of bulk transfer developed with Eqs. 2-17 and 2-19,

$$T_F = \frac{1}{2}\, b\, \frac{B}{t'} \tag{3-3}$$

Figure 3-2 illustrates the process. We now restate the sequential fetch time per record, using the fact that the file size can be expressed as either $b\,B$ or $n\,R$:

$$T_F = \frac{1}{2}\, n\, \frac{R}{t'} \tag{3-4}$$

The use of the bulk transfer rate, t', is appropriate here, since we read the file sequentially from its begin point, passing over gaps and cylinder boundaries, until we find the block containing the desired record.

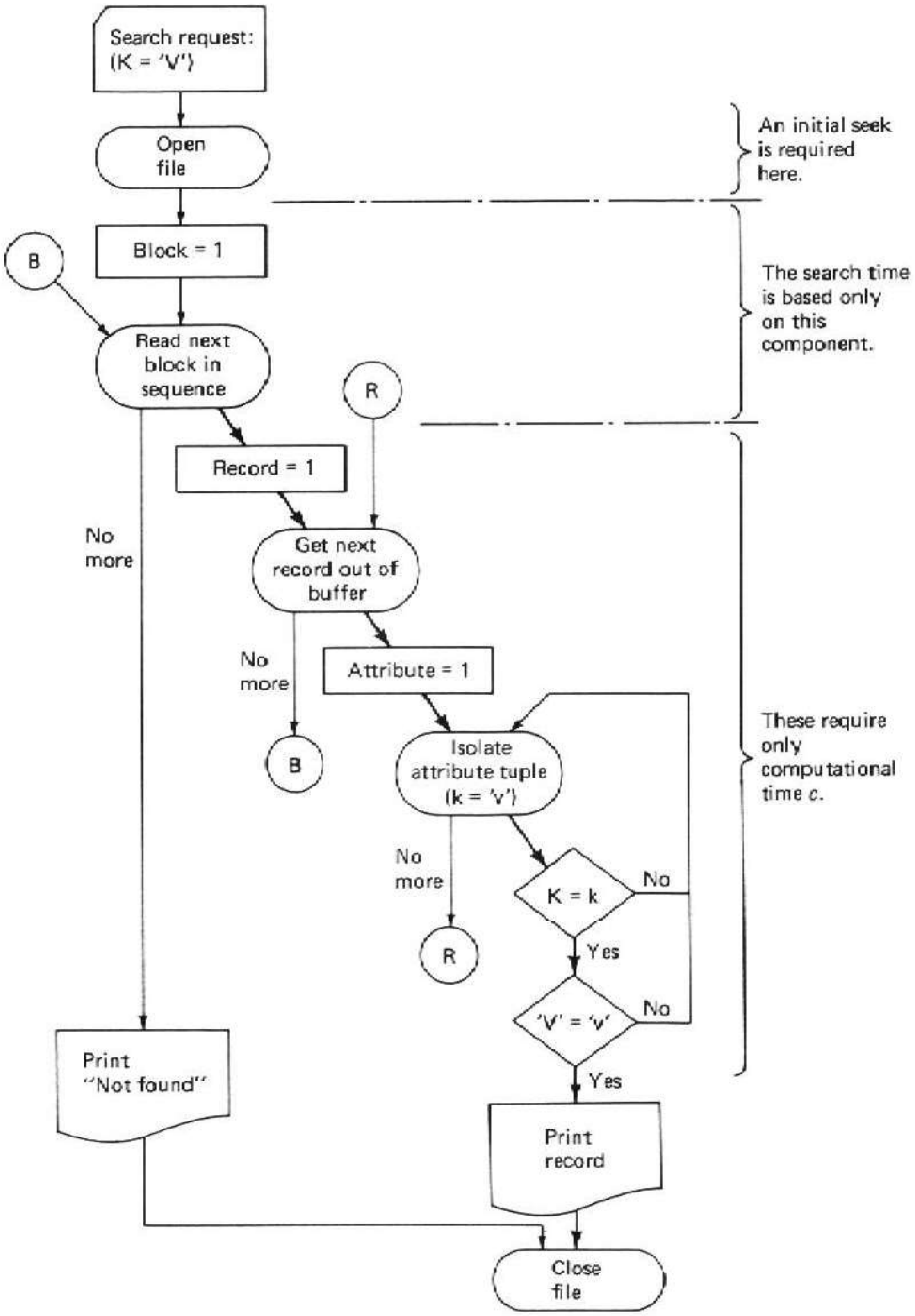


Figure 3-2 Search through a pile.

Batching of Requests It may be effective to collect search requests into batches and avoid the high processing cost of a single fetch. A batch of many requests can be processed in one pass through the entire file. The expected length of search through the file will increase from the factor 1/2 in Eq. 3-4 to 2/3 or 3/4 for two

or three requests. Eventually this factor approaches 1, so that we simply state for a batch of L fetch requests

$$T_F(L) = 2T_F \qquad \text{for} \qquad L \gg 1 \qquad \langle \text{batch} \rangle \text{ 3-5}$$

While this lowers the cost per item searched to $(2/L)\,T_F$, the time to respond to an individual request is now twice the original value of T_F. In addition, there is a delay due to the amount of time required to collect an adequate batch (L) of requests. Such batch operations are typically done on a daily cycle to make them profitable. In that case the search costs are reduced as indicated above, and the expected delay is one day for any request. If many requests are processed in one batch, an efficient processing algorithm for scanning the content of the records is required to ensure that the condition of Eq. 2-22 or $c < R/t$ still holds. Batching of requests applies also to other file organizations.

Get-Next Record of a Pile Since no ordering of records is provided in a pile, the potential successor record may be anywhere in the file. Since the position is not known, the time required to find an arbitrary successor record is also

$$T_N = T_F \qquad \text{3-6}$$

We assume that information from the previous record is required to specify the search for the successor record. If the specification of required attributes for the successor record were known initially, the search for this record could be made during the one combined fetch using the method of batching requests described above.

Insert into a Pile An insertion of a new record into a pile file will be fast because of its lack of structure. We assume that the address of the end of the file is known, a new record is simply appended, and the end-of-file pointer is updated. To obtain dense packing of records, the last block is read into memory, the new record is appended, and the block is rewritten. The time required then will be

$$T_I = s + r + btt + T_{RW} \qquad \text{3-7}$$

When disregarding costs incurred at block boundaries and possible benefits of batching of appends, as argued in Sec. 3-0-3(4), and using Eq. 2-29, we can simplify:

$$T_I = s + 3r + btt \qquad \text{3-8}$$

We confirm, as guessed from personal experience, that a pile is very easy to update.

Update Record in a Pile Updating a record consists of locating and invalidating the old record, and writing a new, probably larger, record at the end of the file, so that

$$T_U = T_F + T_{RW} + T_I \qquad \text{3-9}$$

If only a deletion is to be made, the T_I term drops out. Deletion is actually effected by rewriting the old record space with a *tombstone*.

Read Entire Pile Exhaustive processing of data in this file organization requires reading the file to the end. It is hence only twice as costly as a single fetch, at least if the order in which the records are read does not matter:

$$T_X = 2T_F = n\,\frac{R}{t'} \qquad \langle \text{sequential} \rangle \text{ 3-10}$$

If, however, we wish to read this file serially according to some attribute, the repetitive use of n individual get-next operations would cost $T_X = n\,T_N = n\,T_F$.

This cost would hence be $\mathcal{O} = n^2$. We avoid this cost by sorting the records the file, according to the attribute of the search argument prior to processing. Sorting is analyzed in Section 3-7. Using the value for T_{sort}, as given in Eqs. 3-12 below reduces the cost to $\mathcal{O} = n \log n$.

Records without a matching attribute may be deleted prior to the sort. The sorted file will provide sequentiality of the attribute values. The resulting sorted file is no longer a simple pile.

Read Entire Pile Serially Using a sort to put the file into serial order prior to processing leads to an exhaustive read time of

$$T_X = T_{sort}(n) + T_X(sequential) \qquad \langle \text{serial} \rangle \text{ 3-11}$$

which will be considerably less than $n\,T_F$ for any nontrivial file. An analysis of sorting is provided in Section 3-7. For large files the time required to sort all records is approximately

$$T_{sort}(n) \approx \log_2(n/bft)T_X(sequential) \qquad \langle \text{sort} \rangle \text{ 3-12}$$

although many commercial sort packages can improve on that that estimate. A derivation and a more precise result is provided in Sect. 3-7 as Eq. 3-107,

Reorganization of a Pile If the pile file is updated with tombstones to mark deleted records, as described above, then periodically a removal of the invalidated records is desirable. The file will shrink, and all retrieval operations will be faster.

Reorganization is accomplished by copying the file, excluding records marked with tombstones, and reblocking the remaining records. If the number of records added during a period is o and the number flagged for deletion is d, the file will have grown from n to $n + o - d$, so that the time to copy the file will be

$$T_Y = (n + o)\frac{R}{t'} + (n + o - d)\frac{R}{t'} \qquad \text{3-13}$$

Here $$o = n_{insert} + v$$

The number of records d to be removed during reorganization is

$$d = n_{delete} + v$$

where n_{insert} is the number of records inserted, $n_{isdelete}$ the number deleted from the pile, and v is the number of records that were updated by creating delete and append entries. The values of o and d are dependent on the file organization, so that this and later formulas using these parameters cannot be directly compared.

It is assumed for the pile file reorganization, as well as in other evaluations of the term T_Y, that the reading and writing activities during the reorganization do not cause additional seeks by interfering with each other. This means specifically that, if moving head disks are used, separate disk units must be used for the old and the new file. It also requires that sufficiently many buffers be available to fully utilize the disks. Overlap of disk operations is further considered in Chap. 5-4. Since reorganization is mostly scheduled to be done at times of low utilization, these conditions can frequently be met.

3-2 THE SEQUENTIAL FILE

This method provides two distinct structural changes relative to the pile organization. The first improvement is that the data records are ordered into a specific sequence, and the second improvement is that the data attributes are categorized so that the individual records contain all the data-attribute values in the same order and possibly in the same position. The data-attribute names then need to appear only once in the description of the file. Instead of storing attribute name-value pairs, an entire set of values, a column, is associated with each name. Methods to manage the attribute names will be encountered in Chap. 16 where the concept of a schema is introduced. This organization looks similar to the familiar tabular format that is generally associated with computer data, and shown in Fig. 3-1.

	Name	Age	Height	IQ
1	Antwerp	55	5'8"	95
2	Berringer	39	5'6"	75
3	Bigley	36	5'7"	70
4	Breslow	25	5'6"	49
5	Calhoun	27	5'11"	80
6	Finnerty	42	5'9"	178
7	Garson	61	5'6"	169
8	Hagstrohm	36	5'7"	83
9	Halgard	31	5'6"	95
10	Kroner	59	5'5"	145
11	McCloud	26	5'8"	47
12	Miasma	27	5'2"	75
13	Mirro	38	5'8"	52
14	Moskowitz	23	5'7"	50
15	Pop	38	5'3"	53
16	Proteus	41	5'8"	152
17	Purdy	37	5'9"	48
18	Roseberry	38	5'7"	70
19	Wheeler	23	5'8"	67
20	Young	18	5'8"	89

Figure 3-3 A sequential file.

3-2-1 Structure and Manipulation of Sequential Files

To provide a sequence for the records, we define a key for every record. One or more attributes will become the key attribute(s) for the records in the file. The set of values for the key attributes typically identifies the object described by the record, i.e., the `license number` of a `car` or the `name` of a person. In Fig. 3-3 the `Name` is the key for our `Payroll` file. We expect to be able to identify records uniquely on the basis of their keys. The records in the file are then maintained in order according to the key attributes. One key attribute will provide the primary, high-order sort key, and if this attribute does not uniquely identify the object, then secondary and further key attributes can be specified until the order is completely determined.

Serial reading of the file in the order of the key can be performed sequentially. In Table 3-1*a* and *b* the `student number` provides such a key attribute, but in Table 3-1*c* there is no single field usable as a key. Two attributes, `prerequisite` and `student number`, can be combined to form a key.

‡‡‡‡‡ **Artificial Keys** Sometimes artificial fields containing sequence or identification numbers are added to the records to obtain unique key attributes. These artificial keys will have perfect selectivity: The identification number is chosen to be unique for all the records and hence partitions the file into n individual records. Unfortunately, a separate computation may be needed to determine the identification number pertaining to the desired data. ‡‡‡‡‡

Disadvantages With the structural constraints of sequentiality and fixed records, efficiency is gained but a great deal of flexibility is lost. Updates to a sequential file are not easily accommodated. The fact that only the key attribute determines the sequence of the records introduces an asymmetry among the fields which makes sequential files unsuitable for general information retrieval. The common procedure to handle insertions to a sequential file is to collect them in a separate pile, the *transaction log file*, until the pile becomes too big, and then to perform a *batch update*. The batch update is done by reorganizing the file. At that time the transaction log file is sorted according to the same keys used for the main file, and the changes are merged into a new copy of the sequential file.

A sequential file is restricted to a limited and predetermined set of attributes. A single description applies to all records, and all records are structurally identical. If a new attribute has to be added to a record, the entire file has to be reorganized. Every record of the file will be rewritten to provide space for the new data item. To avoid this problem, one finds that sequential files are sometimes initially allocated with space to spare; a few columns of fields are left empty.

The record layout which will appear on the file is a direct representation of the `DECLARE` statement in the program, as shown in Example 3-4. Such declarations also imply fixed element and record lengths.

Example 3-4 Record declaration.

Program declaration	Sample content
`DECLARE`	
`  1 payroll_record,`	`ERHagstrohm . . .10Mar50 1J...`
`    2 name,`	`ERHagstrohm . . .`
`      3 initials CHAR(2),`	`ER`
`      3 last_name CHAR(28),`	`Hagstrohm . . .`
`    2 date_born CHAR(7),`	`10Mar50`
`    2 date_hired CHAR(7)`	` 1Jan78`
`    2 salary FIXED BINARY,`	`21 000`
`    2 exemptions FIXED BINARY,`	`2`
`    2 sex CHAR(1),`	`M`
`    2 military_rank FIXED BINARY,`	`0`
`        etc.`	`etc.`
`    2 total_wages FIXED BINARY;`	`23 754`
`       . . .`	
`       . . .`	
`    WRITE FILE (payroll_file) FROM (payroll_record);`	

The fixed record layout is easy to construct by processing programs. Since similar information is found in identical positions of successive records, data-analysis programs are easy to write. The record written to the file is often simply a copy of the data in processor storage. Sometimes data is transformed by the processing languages so that files are written using a single data type, perhaps ASCII characters, but kept in binary form in memory to simplify computation. Strong support is given to such record-oriented data through `PICTURE` specifications in COBOL, `FORMAT` statements in FORTRAN and PL/1, and `RECORD` declarations in PASCAL.

3-2-2 Use of Sequential Files

Sequential files are the most frequently used type of file in commercial batch-oriented data processing. The concept of a *master file*, to which *detail records* are added periodically, as sketched in Fig. 3-4, has been basic to data processing since its inception. This concept transfers easily from manual processing to computers, and from one computer to another one. Where data is processed cyclically, as in monthly billing or payroll applications, the effectiveness of this approach is hard to achieve by other methods. Data kept in sequential files is, however, difficult to combine with other data to provide ad hoc information, and access to the file has to be scheduled if the requested information has to be up to date.

In order to combine data from multiple sequential files, sorts are performed to make the records of the files *cosequential*. Then all required data can be found by spacing forward only over the files involved. A sequential file can be in sequence only according to one key so that frequently the file has to be sorted again according to another criterion or key to match other sets of files.

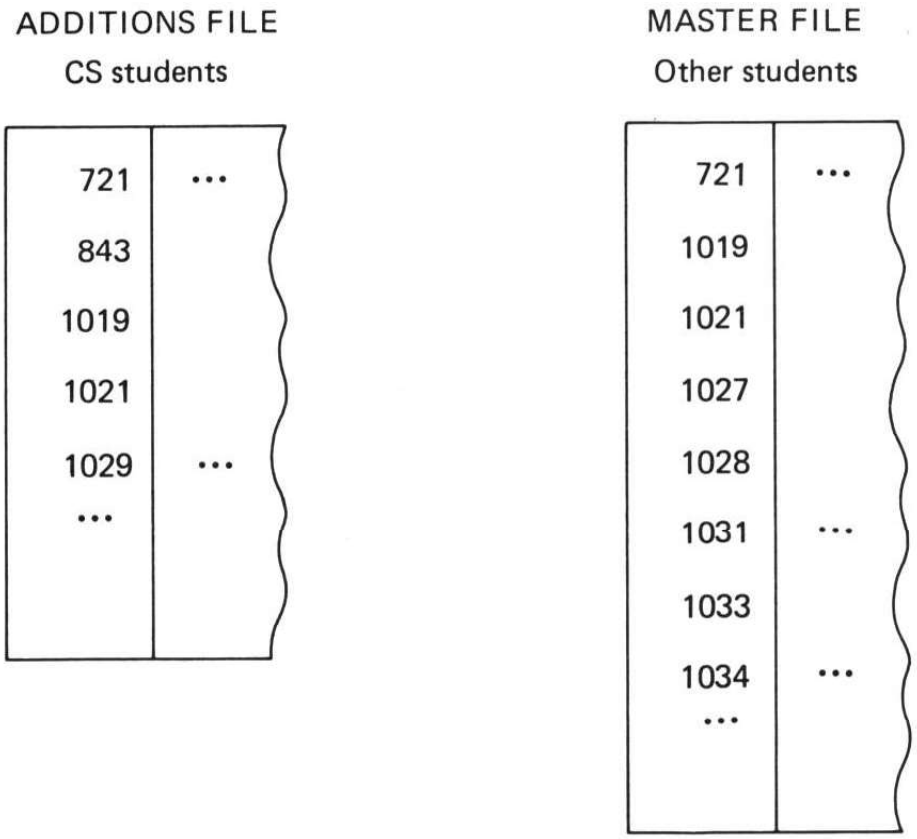


Figure 3-4 Cosequential files.

3-2-3 Performance of Sequential Files

The performance of sequential files ranges from excellent to nearly impossible depending on the operations desired.

Record Size in a Sequential File The file storage requirement for sequential files, using a fixed record format, depends on the number of all the possible attributes a. The description of the attributes appears only once per file, and thus the space required for the attribute names can be neglected. The names may appear outside the file itself, perhaps only in program documentation. However, space will be used for values even when attributes have an undefined value or are unimportant in combination with other attributes. The last two entries shown in Example 3-4 illustrate such an attribute dependency where for the category `sex = F` the next attribute value, `military_rank`, will often be `NULL`. The fixed record size is the product of the number of fields and their average size.

$$R = aV$$

If many values are undefined, the file density will be low. If the value a' is close to the value for a, the file density will be high. Some methods to reduce storage costs for sparse data ($a' \ll a$) will be discussed in Chap. 13-3. If insertions are expected, space for the transaction log to hold up to o new records of length R must also be allocated in an associated area.

Fetch Record in a Sequential File The common approach to fetch a record from a sequential file consists of a serial search through the file. The time required to fetch an arbitrary record can be significantly reduced if we have a direct-access device and use a direct-access technique. In a sequential file direct access can be applied only to the attribute according to which the file has been put in sequence. We describe two direct methods for access to a sequential file organization, binary search and probing, after Eq. 3-17.

Sequential Search When the search argument is not the key attribute used to sequence the file, the search is always sequential. The process is similar to the search through a pile file. Since the total size of the file will be different from the size of a pile file due to the difference in record organization, the relative performance will depend on the attribute density a'/a as well as on the relative length of attribute descriptors A and data values V. Half the file will be searched on the average to fetch a record, so that

$$T_F = \frac{1}{2} n \frac{R}{t'} \qquad \langle \text{main file} \rangle \; 3\text{-}15$$

For small files this time may be better than the direct methods presented below.

If the file has received o' new records into a transaction log or overflow file, this file should be searched also. The entire overflow file has to be processed to assure that any record found in the main part or earlier part of the overflow file has not been updated or deleted. This file will be in chronological order and processed sequentially. With the assumption that the overflow is on the average half-full ($o' = \frac{1}{2} o$), we obtain

$$T_{Fo} = o' \frac{R}{t'} = \frac{1}{2} o \frac{R}{t'} \qquad \langle \text{overflow} \rangle \; 3\text{-}16$$

as the term needed to process changes made to the main file.

We cannot expect that the simple systems which use sequential files will search through both the main file and the log file in parallel. The total fetch time, if both parts of the file are searched sequentially, is the sum

$$T_F = \frac{1}{2} (n + o) \frac{R}{t'} \qquad 3\text{-}17$$

where o is the capacity of the transaction log file.

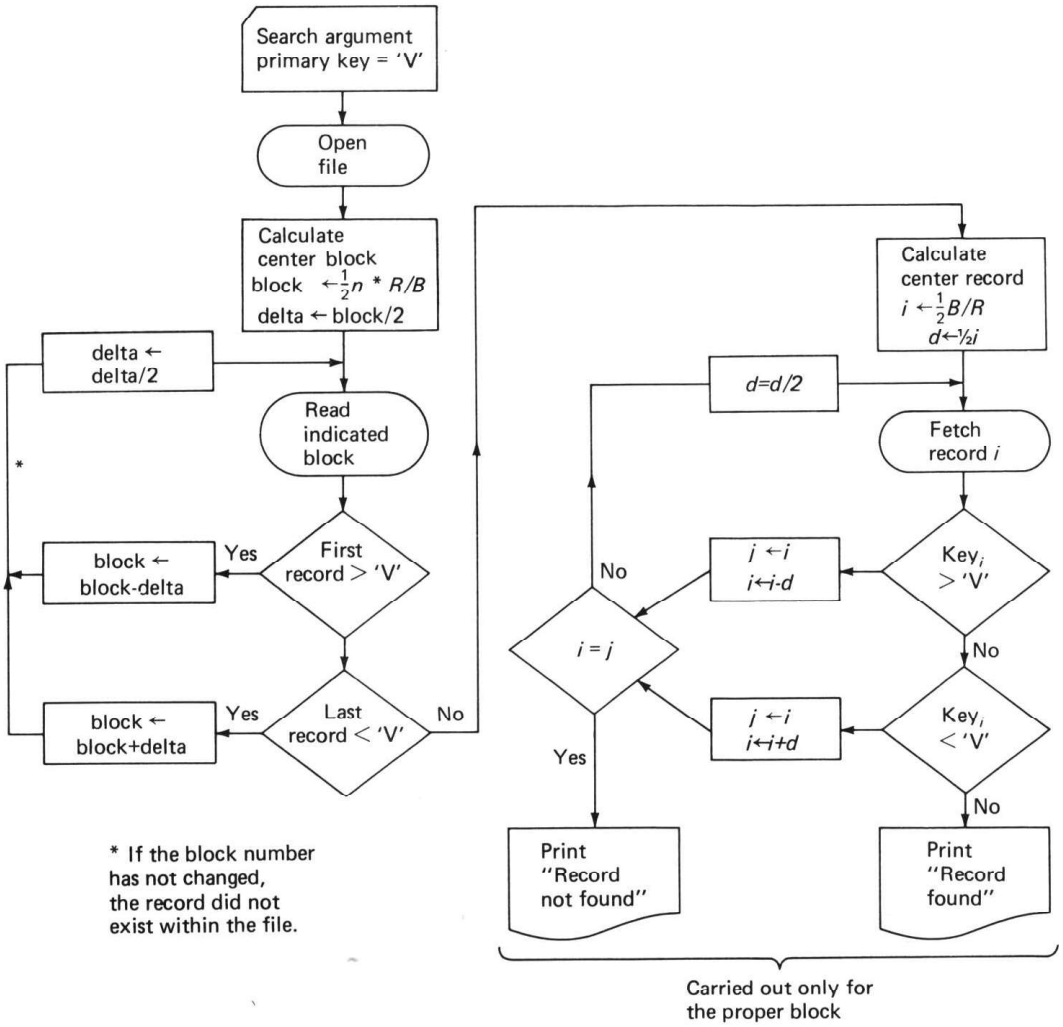


Figure 3-5 Nested binary search in a blocked sequential file.

Binary Search A well-known search technique for memory can be adapted to provide an alternate access method for sequential files. The binary search begins, as shown in Fig. 3-5, with a direct access to the middle of the file, and partitions the file iteratively according to a comparision of the key value found and the search argument. Whenever a block is fetched, the first and last records in this block will be inspected to determine if the goal record is within this block. The number of fetches does not depend on the number of records, n, but rather on the number of blocks, $b = n/Bfr$.

We find, using the expected number of block access for the binary search $log_2(b)$ (Knuth[73S]), that

$$T_F = log_2\left(\frac{n}{Bfr}\right)(s + r + btt + c) + T_{Fo} \qquad \langle \text{binary} \rangle \text{ 3-18}$$

The term for processing time, c, is included here, since until the record range in a block has been checked, it is not known which block is to be read next. The efficiencies obtained when reading the file sequentially using alternate buffers have been lost in this case. The value of c may well be negligible compared with the other times involved, but the bulk transfer rate, t', is always inappropriate. The overflow term remains unchanged.

Probing A third access method for sequential files, *probing*, is more difficult to quantify. It consists of an initial direct fetch, or probe, to an estimated position in the file, followed by a sequential search. If only forward sequential searches can be executed efficiently, the initial probe will be made to an estimated lowest matching key position, so that having to read backward is rare. Only one seek is made, and the number of blocks read sequentially is based on the uncertainty of the probe.

Likely values for an initial probe have been based on the leading digits of a social security number, if its value is used as a key, or on a percentile expectation for leading characters of names, if names are the key. Names beginning with the letters "E", for instance, may be found after $0.2446n$ records in a file sequenced by name (see Table 14-5). The corresponding block would be numbered $\lfloor 0.2446n/Bfr \rfloor$. There is some uncertainty in the distribution leading to a probe. Since on most devices a forward search is best, one can decide, given a 3% or 0.03 uncertainty, that the probe may actually best begin at the block numbered $\lfloor 0.2146n/Bfr \rfloor$ and search forward. Alternate techniques to access records rapidly are used by the indexed and direct file organization methods described in subsequent chapters.

Summary of Access Methods We can summarize the three choices of access methods as follows

1 Sequential search: $\mathcal{O}(n)$
2 Binary search: $\mathcal{O}(\log n)$
3 Probing: $\mathcal{O}(1)$

While the *big-*$\mathcal{O}$ notation clearly distinguishes the difference in growth patterns for these three alternatives, it ignores the complexity of the programs and the important factors of block access and key distribution. For many modestly-sized files the simple sequential search may remain preferable. Only when the files are quite large are the more complex access methods warranted. The simpler methods are also more susceptible to hardware improvements, as described in the section on database machines, Chap. 2-5.

Get-Next Record of a Sequential File In a sequential file, a successor record is immediately accessible and may well be in the same block. If there is a frequent need for successor records, the file system should be programmed so that it does not discard the remaining records in the block but keeps the buffer with the contents of the current block available. The probability of finding a successor record in the same block is determined by the number of records per block Bfr : in $1/Bfr$ of the cases the next block is required. If the processing speed satisfies the condition of Eq. 2-22, the expected time to get the next record is only

$$T_N = \frac{btt}{Bfr} \approx \frac{R}{t'} \qquad \text{3-19}$$

The buffering associated with the bulk transfer rate also evens out any variations in response time, so that performance is not directly affected by the periodic need to read a new block.

Insert into a Sequential File Insertion of a record into the main file requires insertion into the proper place according to the key. The sequence would not be maintained if new records were to be added to the end. For very small files, records beyond the point of insertion can be moved up to make space for putting the new record in place. This effort involves locating the insertion point by a fetch, and subsequently reading and rewriting the remainder of the file. Each phase involves again half of the blocks of the file on the average, so

$$T_I = T_F + \frac{1}{2}\,\frac{n}{Bfr}\,(btt + T_{RW}) = n\,\frac{R}{t'} + n\,\frac{r}{Bfr} \qquad \langle \text{in place} \rangle \text{ 3-20}$$

The assumptions leading to Eqs. 3-15 and 2-29 are employed. This method is feasible for data files only if insertion occurs rarely, for instance, for a list of `departments` of a company. Several insertions can be batched and handled at the same cost.

The usual method for inserting data into sequential files is to collect new records into the transaction log file and, at a later time, execute a batch update. We will use o to indicate the number of records collected for deferred insertion. The actual cost of insertions in the file is hence the immediate cost to append the records to the transaction log file and the deferred cost of the reorganization run. Append costs were given for the pile; we use the conservative method (Eq. 3-8) and write each record immediately into the log. The cost of the T_Y is allocated below to the o records that are collected into the transaction log file between reorganization periods. The reorganization time T_Y and the overflow count o are defined below.

$$T_I = s + 3r + btt \quad + \quad \frac{T_Y}{o} \qquad \langle \text{via log} \rangle \text{ 3-21}$$

The response sensed by the user when inserting a record includes only the initial append terms.

Our definition of sequential files does not allow insertion of larger records than the original record stored. Problems caused by variable-length records and spanning in insertion and update are hence avoided. The transaction log file can also serve other functions, as described in the chapter on reliability, Chap. 11-5-4.

Update Record in a Sequential File A new record is created from retrieved data and new attribute values. If the key value does not change, the record could be rewritten into the main file. If the key value changes, the update is similar to the process of inserting a record but also involves the deletion of a record at another position in the main sequential file.

Since the main file is not otherwise touched in this method, it is best to use the transaction log file also for record update. The updated record and a flag record indicating deletion are appended to the transaction log file and used for subsequent fetches and in the reorganization process. The flag record will include the key and

a tombstone. Note that the tombstone is not placed into the main file either. Both the flag and update records should be added to the transaction log at the same time; the cost of adding two records at a time to a block is not much more than adding a single one, and consistency is enhanced. The cost of an update, which reads the main file and appends to the log file is

$$T_U \approx T_F(\text{main file}) + T_I(\text{log file}) \qquad 3\text{-}22$$

Deletion of a record generates only one entry in the transaction log. No tombstone appears in the main file.

After d record deletions and v record updates, $d + 2v$ records will have been added into the count, o, the size of the transaction log file. Multiple updates to the same record will create multiple entries in the log file. Since complex updating is rarely done using this file organization, we will skip further evaluation of updating performance.

Read Entire Sequential File Exhaustive processing of the file requires reading of the main and the transaction log files in the same order. We assume that data will be processed serially, according to the key used to establish the physical sequence of the main file. The o records placed into the transaction log file must first be sorted to establish this sequence. Then both files can be read sequentially. The total cost is

$$T_X = T_{sort}(o) + (n + o)\frac{R}{t'} \qquad 3\text{-}23$$

given that all conditions for the use of t' hold. We find from the comments in the preceding paragraphs that, denoting the number of insertions as n_{insert} and the size of the prior main file as n_{old},

$$o = n_{insert} + 2v + d \quad ; \qquad n_{new} = n_{old} + n_{insert} - d$$

The value of the transaction count, o, includes here insertions, two entries for records being changed, and the number of records to be deleted. The transaction sort has to be stable, as defined in Sec. 3-7, so that multiple changes to the same record will be handled in the original chronological order.

If the transaction log file is relatively large ($o \not\ll n$), it may be best to reorganize the file as well. The records can be analyzed during the merge of reorganization, so that $T_X = T_Y$, as analyzed below.

Reorganization of a Sequential File Reorganization consists of taking the old file and the transaction log file and merging them into a new file. In order to carry out the merge effectively, the transaction log file will first be sorted according to the same key field used for the old file. During the merge the sorted data from the transaction log file and the records from the old sequential file are copied into the new file, omitting any records which are marked for deletion in the transaction log file. The time required for the reorganization run consists of the sort time for the transaction log file plus the merge time. Merging requires the sum of the times to read both files and write a new sequential file

$$T_Y = T_{sort}(o) + n_{old}\frac{R}{t'} + o\frac{R}{t'} + n_{new}\frac{R}{t'} \qquad 3\text{-}24$$

or if the number of records being deleted, $d + v$, can be neglected:

$$T_Y = T_{sort}(o) + 2(n + o)\frac{R}{t'} \qquad 3\text{-}25$$

The time required to sort a file of size o as $T_{sort}(o)$ was estimated in Eq. 3-12, and derived more precisely in Sec. 3-7.

3-3 THE INDEXED-SEQUENTIAL FILE

In this Section and its successor we describe and analyze files which use indexes: the indexed sequential file and multiply indexed files. The concept of an index is a familiar one from books: The dense, ordered index entries at the end of a book help to rapidly locate desired information in the text. One or several references may be obtained for an index entry. In this chapter we present the principles of indexes as used for computer files. In Chap. 4 we augment this material with features used in implementation.

File Indexes An index consists of a collection of entries, containing the value of a key attribute for that record, and reference pointer which allows immediate access to that record. For large records, the index entry will be considerably smaller than the data record itself. The entire index will be correspondingly smaller than the file itself, so that a smaller space will have to be searched. The index is kept in sorted order according to its key attribute so that it can be searched rapidly.

Indexes become effective when files are quite large, so that the index requires many fewer blocks. The search process within a large index itself is aided by again indexing subsets of the index, in Example 3-5 we illustrate that concept by grouping social security numbers with the same initial digit.

The indexed-sequential file design overcomes the access problem inherent in the sequential file organization without losing all the benefits and tradition

Example 3-5 Illustration of principles for indexing.

We have an employee file sequenced by social security number:

TID	Social Sec#	Name	Birthdate	Sex	Occupation	...
1	013-47-1234	John	1/1/43	Male	Welder	...
2	028-18-2341	Pete	11/5/45	Male	Creep	...
3	061-15-3412	Mary	6/31/39	Female	Engineer	...
	...-..-....	....	.....	.....	.....	...

To find employees we establish an index file as follows:

Index block id	key value	TID
a	013-47-1234	1
	028-18-2314	2
	061-15-3412	3
	...-..-....	.

To find, in turn, the index blocks rapidly we establish a second index level as follows:

Index block id	key value	index block id
m	013-47-1234	a
	102-81-2314	b
	201-51-3412	c
	...-..-....	.

associated with sequential files. Two features are added to the organization of the sequential file to arrive at this third file organization type. One additional feature is an index to the file to provide better random access; the other is an overflow area to provide a means to handle additions to the file. Figure 3-6a indicates the three important components of an indexed-sequential file: the sequential file, the index, and the overflow area. Figure 3-6b shows an example, with a number of details that will appear in later discussions.

Successively higher levels of the index become smaller and smaller until there remains only a small, highest level index. A pointer to the top level of an index structure is kept in the file directory. When the file is `OPEN` the pointer to access the index is available in memory.

In Example 3-5 the top-level index will have 10 entries — one for each `SSN`-digit, and for 500 employees the first level index blocks will have an average of 50 entries. In practice, index blocks are filled to capacity, so that the index divisions are determined by the size of blocks available to the file system, as shown in Fig. 3-6.

With every increase in level the search gets longer, but at every level a block's worth of index entries can be scanned. There is rarely a need for many levels of indexing. The recursive nature of a computer index distinguishes it from an index in a book.

The Shape of Indexes

As seen in Example 3-4 the entries for an index are quite small, containing only one value and a TID*. Hence, many index entries will fit into one block. The number of entries obtained for each retrieval of an index block is called the *fanout*. A fanout of 100 is not unusual. With two levels we can access them up to 10 000 records and with three levels up to 1 000 000. The index trees are very broad, rather than high. The term used to measure the breadth of a tree is the *fanout ratio y*. Broad trees have few levels and provide rapid fetch access.

In Fig. 3-7 we illustrate symbolically a small and a large fanout ratio. The trees are not presented in traditional computer science upside-down fashion, so that the process to fetch a leaf on the tree starts in the figure from the bottom, at the root. The fanout ratio is a very important parameter in the analysis of indexed file organizations and will be encountered frequently. Similar data structures used in memory, as *m-way trees*, are often quite narrow, most frequently binary ($m = 2$).

Static versus Dynamic Trees

The two indexed file organizations presented in this chapter differ greatly in index management. The index of an indexed-sequential file is created at the time of reorganization, and does not change due to updates. New records are placed into an overflow file and linked to their predecessors.

* The abbreviation TID derives from the term *tuple identifier*, used when modeling databases where *tuples* are used to represent records. We use it here to denote any kind of record pointer.

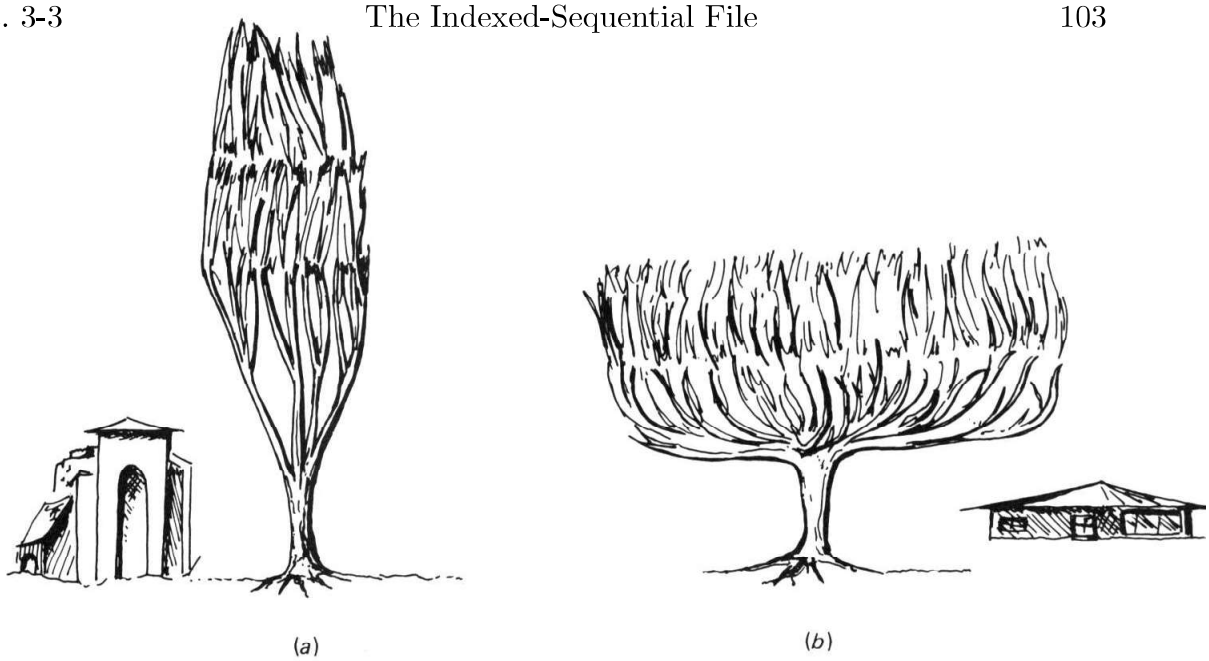


Figure 3-7 Italian and Monterey cyprus. (*a*) Low fanout. (*b*) High fanout.

The multiple-indexed file places updates into the main-file. These records must be locatable via the indexes, so that here the indexes are dynamic. We use a *B-tree index*, which trades space for the ability to remain up to date.

Inverted Files A file for which indexes have been created is sometimes referred to as an *inverted file*. This term has its origin in bibliographic indexing, as shown in Fig. 3-14 and will be used rarely and carefully in this book. Sometimes a copy of a sequential file, when sorted according to another key attribute, has been called an inverted file. The term *fully inverted* generally implies a file where all attributes have indexes, as described in Sec. 3-4.

The terms *inverted index*, *inverted list*, *inverted file*, and *partially inverted file* are used inconsistently in the literature, and frequently imply indexing as described in this chapter. We simply avoid the adjective *inverted* in this book.

3-3-2 Structure and Manipulation of Indexed-Sequential Files

The indexed-sequential file organization allows, when reading data serially, sequential access to the main record areas of the file, shown conceptially in Fig. 3-6a and by example in Fig. 3-6. Only some pointer fields, used to handle insertions, must be skipped for serial reading.

Records which have been inserted are found in a separate file area, called the *overflow area*. An overflow area is similar to the transaction log file used previously, but integrated when we have an indexed-sequential organization. The records in this area are located by following a pointer from their predecessor record. Serial reading of the combined file proceeds sequentially until a pointer to the overflow file is found, then continues in the overflow file until a `NULL` pointer is encountered; then reading of the main file is resumed.

To fetch a specific record, the index is used.

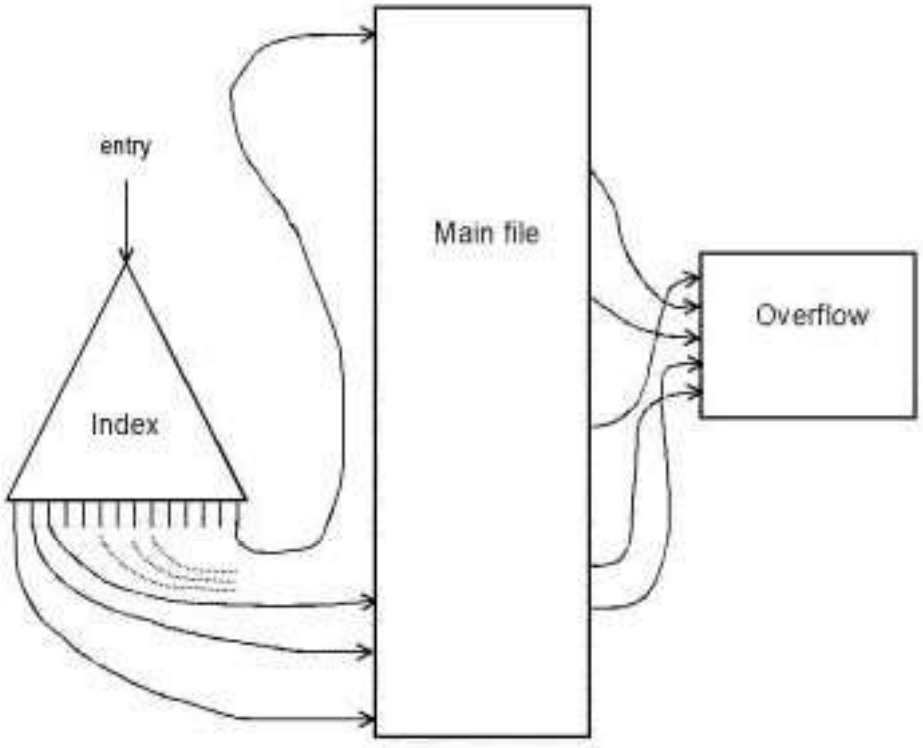


Figure 3-6a Components of an indexed sequential file.

An Index for an Indexed-Sequential File Indexes have been implemented in a variety of ways. We will consider here a static, multilevel index, using *block anchors*. An alternative, dynamic, index method, called a *B-tree*, will be presented with the indexed files of Sec. 3-3. We discuss here the most prevalent version of the indexed-sequential file. Improvements to this scheme are covered in Chap. 8-1.

Selecting the Key The index for an indexed-sequential file is based on the same key attribute used to determine the sequence of the file itself. For such a *primary index* a number of refinements can be applied. One of these is indexing only the first record in every block — using *block anchors* — and the other is keeping most of the index on the same cylinder as the data records, *cylinder indexes*. The effect of having cylinder indexes will be considered in Chap. 8-1-1.

Block Anchors The benefit of the index is to rapidly access a block of the file. Individual records in a block can be found by a search within the block, so that it is not necessary to keep in the index a TID for every record, but only a reference to one record per block. The referenced record is called an *anchor point*, and only the anchor's key value and the block pointer are kept in the index. Natural anchor points are based on blocks, tracks, or cylinders. In Fig. 3-6b the choice of anchor point is the first record of a block.

The cost of searching within a file block for a record is small, since the entire block is brought into memory whenever required and can be kept available in a buffer. A block will contain a number of records equal to Bfr. The number of entries in a block-anchored index is, hence, n/Bfr, and the size of an index entry is $V + P$.

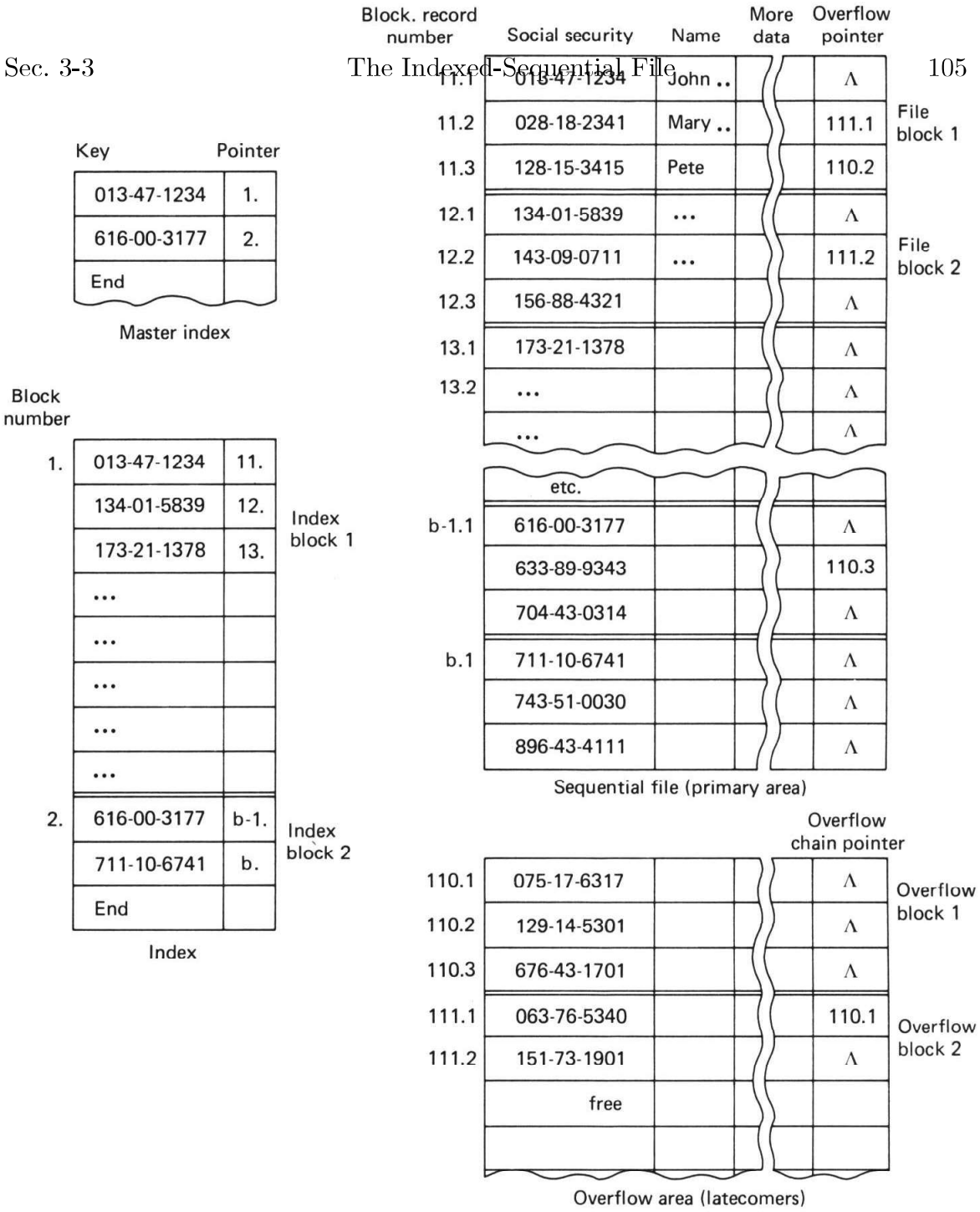


Figure 3-6 An indexed-sequential file.

When only block anchors are kept in the index, it is not possible to determine by checking the index alone if there exists a record corresponding to a specific argument. The appropriate data block has to be read also. To determine if the value of a search argument is beyond the last entry in the file, the last data block will have to be fetched, since the anchor point refers to the first record in this block.

If records are often appended to the end of the file, it can be more convenient to keep the key value of the last record in each block in the index entries instead of the key for the first record. The appropriate block for a given search argument is then found in the index through a less-than-or-equal match. ‡‡‡‡‡

Quantifying the Shape of an Index The shape of an index tree is characterized by two related parameters: *fanout* and *height*. We will discuss these two concepts in turn,

The Fanout Ratio of an Index As stated in the introduction, an important parameter of an index is the referencing capability of a block of index entries, the *fanout*. The fanout, y, is the quotient of the block size, B, and the space required for each entry, $V + P$,

$$y = \left\lfloor \frac{B}{V + P} \right\rfloor \qquad \text{3-26}$$

To evaluate the number of levels of indexing that might be required, we will take an example of a fairly large file (one million records) and a block-anchored index. In order to estimate the access time to fetch a record, we need to know how many levels of index are needed for a file of this size. Example 3-6 shows this procedure for a general block-anchored index. In Example 3-7 the same file is evaluated for the case of a block-anchored index kept on the same cylinder as the referenced data.

Estimating the Height of an Index The height of an index tree is the number of indexing levels, x, required to access all records of the file. To estimate the height we consider the exponential growth of each level, so that

$$x = \lceil \log_y \lceil n/Bfr \rceil \rceil \qquad \text{easily computable as} \qquad \lceil \ln \lceil n/Bfr \rceil / \ln y \rceil \qquad \text{3-27}$$

The estimate should be verified for a specific design, since its value is so critical to performance. Values of x found in practice range typically from 1 to 3. Larger values occur only when the size of the file is very large while the key attribute V is unusually great; in Chap. 4-3 we describe techniques to keep the effective V small.

Example 3-6 Hardware-independent index design.

Given is a block size B of 2000 bytes, a value size V of 14 bytes, a TID pointer size P of 6 bytes, and data records having a total length R of 200 bytes. With this blocking factor Bfr of 10, the 10^6 records require 10^5 data blocks and hence as many TIDs. The size of the index entry is here $14 + 6 = 20$ bytes, and the block size B is still 2000.

Now Eq. 3-26 gives a $y = 100$, so that the 10^5 lowest-index-level entries occupy 10^3 blocks which can be pointed at by 10^3 second-level index entries. This second-level index will occupy a total of $20 \times 1000 = 20\,000$ bytes. The latter number is still excessive for storage in memory, so that a third index level to the second index level will be created. Only $20\,000/2000 = 10$ entries occupying 200 bytes are required at the top level. The term *root* refers to this topmost level. The index levels are numbered from 1, for the level closest to the data, to x (here 3) for the root level.

A record-anchored index for the same file would have been Bfr or 10 times as large but will use the same number of levels, as can be seen by recomputing this example for 10^6 index entries. Its root level (x) will have more entries, of course.

Reducing the Access Costs of Indexes Two common improvements are made to indexes to increase their performance:

1 Keeping the root of the index tree in memory
2 Locating portions of the index so that seeks are minimized.

We will consider these two adaptations and then evaluate their joint effect.

Root Blocks in Memory Since the root level of an index only occupies at most one block it is common to place it in memory, trading some memory for access speed. Keeping the root block (level x) available, once it has been read, avoids repeated disk accesses for that level of an index. The lower levels (levels $x-1, \ldots, 1$) of the index will still be obtained from disk. The root block can be placed into memory when the file is opened. With this scheme we reduce the total number of accesses to disk blocks required to fetch data records from the data file from $x+1$ to x.

Cylinder Indexes Since much of the time cost of fetching a block is due to the seek time required to reach a particular cylinder, access speed can be gained by avoiding seeks during access. The primary index, having a parallel structure to the data file, can be partitioned according to hardware boundaries to yield such benefits.

Specifically, the top of the primary index (level x or levels x and $x-1$) can be used to partition the file into portions that each fit one cylinder. Two such levels are used only if the file is larger than a hardware device, say a disk unit. Then the root level assigns the hardware device and the subsidiary level partitions the storage assigned to one device into cylinder portions. On level 1 is the remainder of the index, its blocks allocated to the cylinders containing the data they reference. Allocating index space within cylinders used for data slightly reduces the capacity for data and increases the bulk data transfer rate, but the access speed gained can be substantial.

To optimize this approach, the size of an index level is best determined by the number of blocks it should reference. Only one low index level (1) needs to be allocated on a cylinder for its records; the number of entries is determined by the data capacity of the remainder of the cylinder. More than one block may be needed for the entries to one cylinder, but since the blocks will be placed adjacent on a track, only additional buffer space and block transfer time (*btt*) is needed.

The next level up, level 2, matches the number of cylinders required. Even if a file uses all of a disk unit, the number of entries will still be only a few hundred. A level $x = 3$ only comes into play when files are greater than one disk unit.

The contents of the entries are also adapted to the hardware. The optional third level uses as pointers references to the operating system directory of disk units. The level two index contains only key attribute values and addresses of cylinder anchors. If the level two entries match sequential cylinders the entries do not even need a pointer field, since entry 1 simply corresponds to block 1, etc. On the initial track of each cylinder there will be the cylinder index. The entries on the cylinder index, level 1, do not need a cylinder address segment; they reference only local tracks, blocks, or records.

We have now a three- or two-level index, with possibly more fanout at lower levels than seen in a block-oriented index design. No seek is incurred between

cylinder-index and data records. For files residing on one disk a total of two seeks suffice for data access, one for the root block and one for the cylinder containing index and data. The example shown in Example 3-7 evaluates this type of index for a large file and can be compared with Example 3-6.

Example 3-7 Hardware-oriented index design.

Again given is a block size, B, of 2000 bytes and data records having a total length, R, of 200 bytes. With this blocking factor, Bfr of 10, the 10^6 records require 10^5 data blocks. The value size, V, is 14 bytes The TID pointer size, P, of 6 bytes can be abbreviated to 4 bytes with a cylinder.

Using disks with 200 cylinders of 19 tracks, capable of holding 14 000 bytes each and $B = 2000$, we find on each cylinder $19 \cdot 14\,000/2000 = 133$ blocks.

The index to the data on one cylinder, level 1, will require one entry per block, at most $133V = 1862$ bytes, or one block on each cylinder, leaving 132 blocks for data. The portions of the level 2 index to the cylinders are kept on their particular disks. Their entries do not require a pointer field, there is simply one entry per sequential cylinder. On each device we use $200 \cdot 14 = 2800$ bytes or two blocks for the level 2 index. The reduction in disk capacity is negligble. The data file oupies $\lceil 10^5/132 \rceil = 758$ cylinders. This file requires $\lceil 758/200 \rceil = 4$ disk units. We need a third-level index to the cylinder indexes; this root level of the index for the entire file will have one entry per disk, or four entries for this particular file. Only $4 \cdot (14 + 6)$ bytes are needed here.

Comparing this example with the original design of Example 3-6 we find that the hardware-oriented indexing structure has a higher fanout ($y_1 = 132$, $y_2 = 200$) than the general, symmetric index structure. A by-product is that the root level is smaller ($y_3 = 4$). For very large files restricting an index to three levels may create larger root blocks. The second-level index is larger and requires additional block reads, costing *btt* for retrieval and additional buffer space.

The actual values for a file depend on the interaction of file, record, and key sizes with hardware boundaries. The hardware-independent index structure is more general and easier to adapt to specific file requirements. ♯♯♯♯♯

The Overflow In order to insert records into the file, some free space has to be available. Let us consider three choices for placing the records to be inserted

1 Use a separate file, as in the sequential file organization
2 Reserve space on every cylinder used by the file
3 Reserve space in every block for insertions

Let us look at the extreme cases first. A separate insertion file requires a separate access with seek and latency overhead at any point where an insertion had been made. We want to do better now. Allocating space in every block is feasible only if blocks are large and insertions are well distributed; otherwise, it is too easy to run out of space in some blocks. To make such an approach successful requires dynamic space allocation, as used by multiply indexed files described in Sec. 3-3.

Keeping spare space in every cylinder provides a practical compromise. This is the method chosen in the typical indexed-sequential file organization. Locating an overflow record will require rotational latency but not a seek. To insert a record the cylinder address is obtained from the index by matching the attribute key value of

the new record to the entry for the nearest predecessor. The new record is placed in the next sequential free position in the cylinder overflow area.

Linkage to Overflow Records The records in the overflow areas should be found both by `Fetch` and `Get_Next` operations. In both cases the search process begins from the predecessor record and then follows pointers.

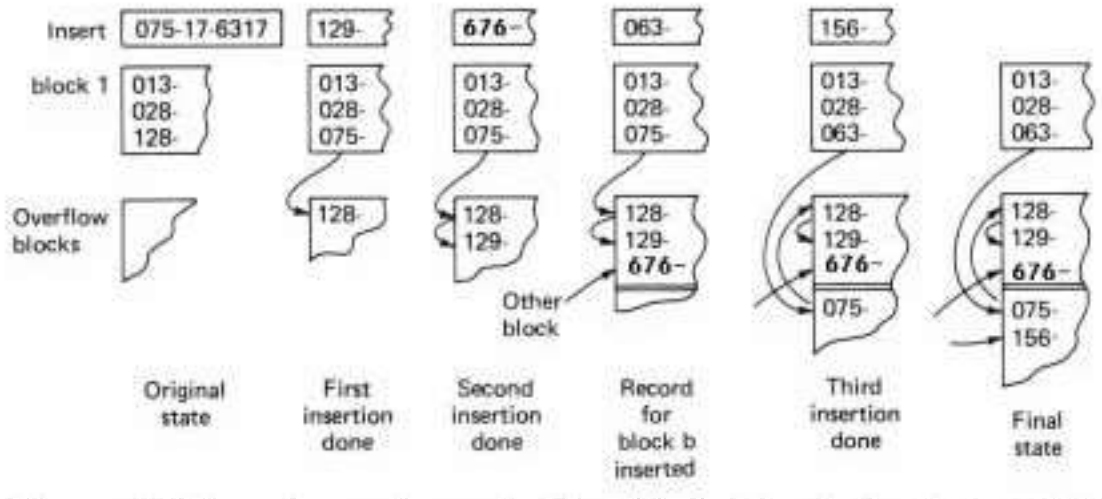
Overflow pointers are placed following the predecessor records in the primary data blocks (see Fig. 3-6). The key of the inserted record is not kept there. A search for any intervening record is directed to the overflow area. Modification of the index to reflect the insertion is avoided, but one more block access is needed on every fetch of an overflow record. A request for a nonexistent record will also require going to the overflow file if an overflow pointer is set with the predecessor record.

Chaining of Overflow Records To locate multiple overflows a *linked list* is created. Linkage pointers are placed in the records in the overflow areas as well, so that all overflow records starting from one source are linked into a *chain*. A new record is linked into the chain according to its key value, so that sequential order is maintained. The chain can go through many blocks of the overflow area.

When the fetch has to proceed via many overflow records in a large number of blocks, following the chain to a specific record may actually be less efficient than simply searching the overflow area exhaustively. On the other hand, serial processing is greatly simplified when we can follow the chain. In order not to lose the records from the sequential file buffer when processing, a separate overflow buffer should be available.

Push-through Instead of having one overflow pointer per record in the data file it is common to use only one pointer per block. With this method the key sequence in the blocks of the primary file is maintained:

1 New records are inserted after their proper predecessor
2 Successor records are pushed toward the end of the block.
3 Records from the end of the primary block are pushed out into the overflow area.



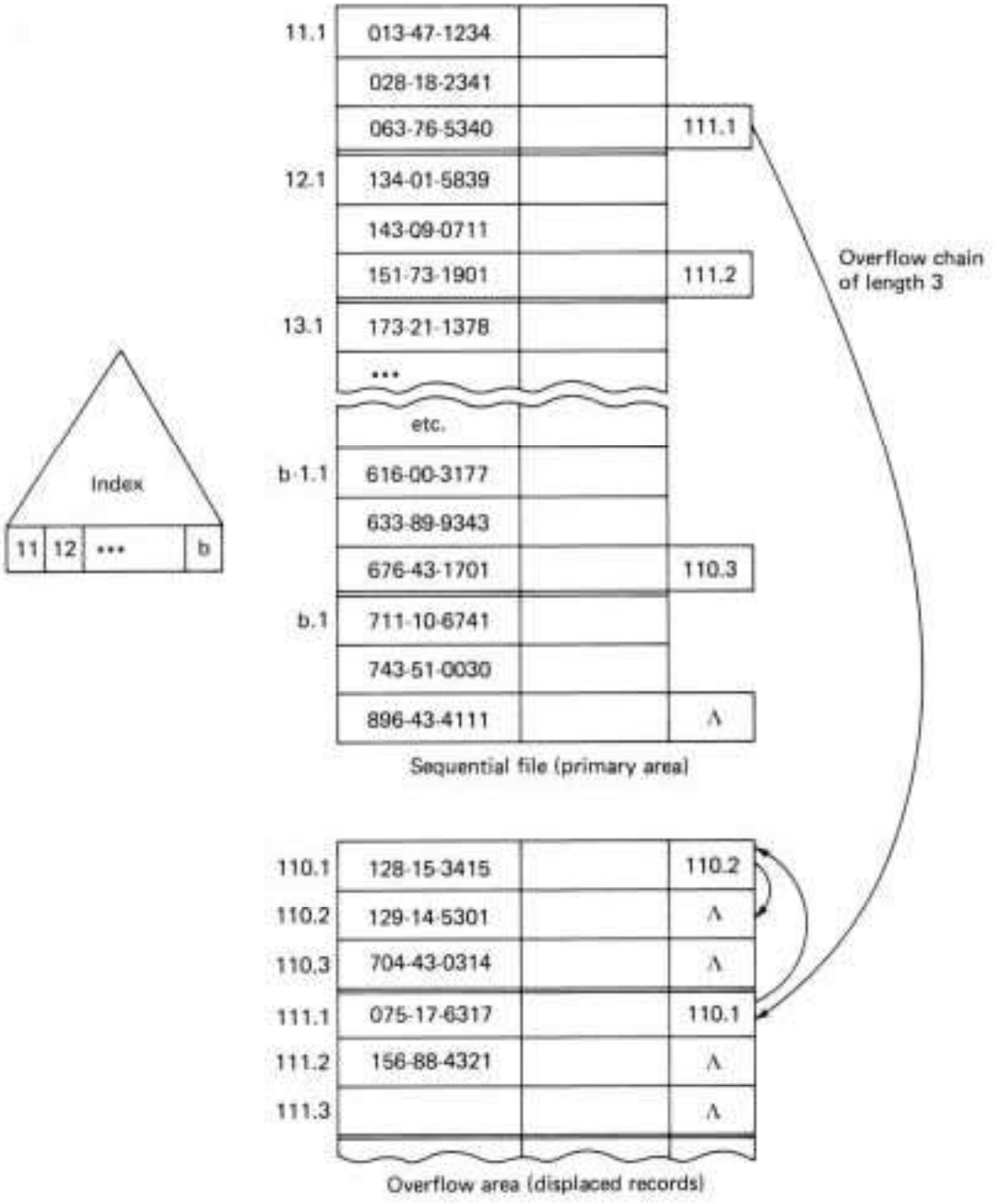


Figure 3-9 Indexed-sequential file overflow with push-through of records.

Figure 3-8 illustrates the *push-through* process using the same insertion sequence which led to Fig. 3-6. The final state of the file is shown in Fig. 3-9. The index is identical, since block positions do not change. Pointers are depicted in the form `block.record` number. Now only one overflow pointer is required per primary block and only one jump is made to the overflow file per block.

The chains will be longer than if each record had its own successor chain in the overflow area, by a factor of *Bfr*. This means that a fetch of a record placed in the overflow area will take longer. The average fetch and serial access will be better

since there will be less switching between the sequential file area and the overflow file area, especially when the two areas are not on the same cylinder.

As the number of overflow records increases and with fairly many records per block, under conditions of uniform distribution the probability of having an overflow chain for every block rapidly approaches certainty.

‡‡‡‡‡ **Processing Overflow Chains** Overflow chains are costly to process. The serial processing required for overflow chains requires fetching a new block for each record. We will consider three issues here.

1 *Searching for nonexisting records.* Half the chain will have to be searched to find a record, but the entire chain must be searched to determine that a requested record does not exist. Maintaining chains sorted by key reduces the search for nonexisting records to about half the chain length as well.
2 *Fetching frequently needed records.* An alternative, keeping records in the chains by order of frequency of use can help find existing records faster. Since recent updates are more likely to be needed it can be beneficial to insert new updates into the front of the chain. This strategy reduces update cost as well.
3 *Considering the distribution of chain lengths.* In practice, overflow chains have quite unequal lengths; for instance, `append` operations create long chains connected to the last block, unless special mechanisms are used for appends. For an inventory file, activity may be high in a particular item, while other items are unused. In Chap. 6-1-5, Example 6-4, we develop estimates of the chain length, Lc, for random insertions, leading to Poisson distributions.

Statistical analysis becomes important if push-through is not used, since then many small chains must be considered. ‡‡‡‡‡

Size of Overflow Areas Cylinder overflow areas have to be carefully dimensioned. If insertions cluster in certain areas, the corresponding cylinders will need large overflow areas. If the system provides a space allocation so that all cylinder overflow areas are of equal size, as most do, then much space can be wasted in cylinders that are not receiving many insertions. An escape hatch can be the provision of a secondary overflow area that is used when any cylinder overflow area itself overflows. Now the objective of avoiding seek time is lost for the records placed in the secondary overflow area; specifically, serial access will be quite slow.

Some methods to estimate the size of overflow areas required, given a specific distribution of record insertions, can be found in Chap. 6-1, and an assessment of insert versus inquiry frequencies is sketched in Chap. 5-1.

Reorganization At or before the point when the overflow areas themselves overflow, a file reorganization is required. Reorganization can also be needed when, because of the creation of long chains, the fetch or serial processing times become excessive. Such a reorganization consists of the steps shown in Table 3-3. During this process, the reorganization programs will create a completely new index based on new anchor point values for the blocks of the data file. The previous index is simply discarded.

The need for regular reorganization is a major drawback of indexed-sequential files, since it involves monitoring of the files and periodic special processing.

Table 3-3 Reorganization steps for an indexed-sequential file.

1	Read the file in the manner that would be used when doing serial processing, using both the sequential and the overflow areas.
2	Leave out all records that are marked deleted.
3	Write all new and old remaining records sequentially into the sequential areas of the new file.
4	Create and insert into a memory buffer an index entry with every block placed into the sequential area.
5	Write the index blocks out when they get full.
6	Create and insert a higher level index entry for every index block written.
7	When all data blocks have been written the remaining index buffers are written out.
8	Free the areas used by the old index.

The frequency of reorganization is dependent on the insertion activity within the file. In practice, one finds time intervals ranging from a day to a year between reorganization runs. Since a reorganization run can take a long time, the reorganization is generally performed before the file is actually full to avoid unpleasant surprises at busy times. It may simply be scheduled at periodic intervals or be done at a convenient instance after the entries in the overflow area exceed a preset limit. In Chap. 14-5 an algorithm for determining reorganization intervals will be presented.

3-3-2 Use of Indexed-Sequential Files

Indexed-sequential files of the basic type discussed above are in common use in modern commercial processing. They are used especially where there is a need to keep files up to date within time frames that are less than the processing intervals which are possible with cyclical reorganization of sequential files. Since individual records can be inserted and retrieved through the index, so that a limited number of block accesses are required, this type of file is suitable for *on-line* or terminal-oriented access. On-line use is not feasible with the pile and sequential file types unless the files are quite small.

At the same time sequential access is relatively simple and efficient. Without overflows, after a reorganization, sequential access is practically as fast as for the sequential file. An indexed-sequential file can, for instance, be used to produce an inventory listing on a daily basis and be reorganized on a weekly basis in concert with a process which issues notices to reorder goods for which the stock is low.

Indexed-sequential files are also in common use to handle inquiries, with the restriction that the query must specify the key attribute. Typical of these are billing inquiries based on account numbers. Sometimes copies of the same data are found sequenced according to different keys in separate indexed-sequential files to overcome this restriction. Updating cost and space requirements are multiplied in that case.

The effects of the specific design implemented for an indexed-sequential file are frequently not understood by the users, so that many applications which use

indexed-sequential files take longer to process data than seems warranted. In situations where files receive updates in clusters, the generated chains can be costly to follow. Often, clustered updates are actually additions to the end of a file. By treating these separately, or by preallocating space and index values in the indexed-sequential file during the prior reorganization, the liabilities of appended records can be reduced. Within one indexed-sequential file method the options are often limited, but a number of alternative indexed-sequential file implementation are available from computer manufacturers and independent software producers.

The restriction that only one attribute key determines the major order of the file, so that all other attribute values are not suitable as search arguments, is common to all sequential files. Indexing with multiple keys is presented in Sec. 3-4. There a different scheme of index management is used and the sequentiality of the file is abandoned.

3-3-3 Performance of Indexed-Sequential Files

Performance evaluation of indexed-sequential files is more complex than evaluation of the two preceding file organization methods because of the many options possible in the detailed design. We will base our evaluation on a simple form similar to the most common commercial designs.

> The index is on the same key attribute used to sequence the data file itself. The first-level index is anchored to blocks of data, and a second-level index has one entry per first-level index block. Push-through is used when records are inserted into a block. Records in an overflow area of size o are linked in key order to provide good serial access.
>
> Records to be deleted are not actually removed, but only marked invalid with a tombstone. An additional field in each record of the main file is used for the tombstone. A field for a pointer for chaining is maintained within the overflow area. For both functions the same field position, of size P, is allocated in the records. We permit records of variable length in our analysis, although only few systems support such records.
>
> After a reorganization the main data areas and index areas are full and the overflow areas are empty. No change of the index takes place between reorganizations, simplifying the insertion of records. All areas are blocked to an equal size B. The total file may occupy multiple cylinders.

Record Size of an Indexed-Sequential File In the sequential part of the file, a record requires space for a data values of size V and for a possible tombstone of size P.

$$R = aV + P \qquad \langle \text{net} \rangle \text{ 3-28}$$

In the main file are n_m records and in the overflow file are o' records. Ignoring deleted records, $n = n_m + o'$. Initially or after reorganization, $n = n_m$. Space is allocated for up to o overflow records.

In addition, there exists an index with entries to locate the file blocks. The first-level index contains one entry per data block, so that, for level 1,

$$i_1 = \frac{n_m}{Bfr} \qquad 3\text{-}29$$

entries are required. The sizes of higher index levels are determined by the fanout, y, which in turn is determined by the block size, B, and the size of each index entry, $V + P$, as shown in Eq. 3-26. On each successive level will be one entry per lower-level index block, so that

$$i_{level} = \left\lceil \frac{i_{level-1}}{y} \right\rceil \qquad 3\text{-}30$$

until one block can contain all the pointers to the lower-level index. The number of blocks, bi, required for any index level is

$$bi_{level} = \left\lceil \frac{i_{level}}{y} \right\rceil = i_{level+1} \qquad 3\text{-}30a$$

was F5-6 The total size of the index, SI, is then obtained by summing these until the single block at the root level, $bi_x = 1$, is reached. Using Eq. 3-30b we can simplify

$$SI = (bi_1 + bi_2 + \ldots + 1)B = (i_2 + i_3 + \ldots + 1)B \qquad 3\text{-}31b$$

The total space per record, including space allocated to overflows, is then

$$R_{total} = \frac{n_m R + oR + SI}{n} \qquad \langle \text{gross} \rangle \; 3\text{-}32$$

The space used for the file remains constant during insertions, until a reorganization frees any deleted records and moves the overflow records into the main file. The new main file then has n records; the overflow file is empty but has space for o insertions, and the index size is probably pretty much the same.

Fetch Record in an Indexed-Sequential File To locate a specific record, the index is used. The primary fetch procedure consists of an access to each level of the index, and a `READ` of the data block (Fig. 3-10).

$$T_{Fmain} = x(s + r + btt) + s + r + btt = (x + 1)(s + r + btt) \qquad \langle \text{primary} \rangle \; 3\text{-}33$$

Hardware-oriented indexes can reduce this cost.

If insertions have occurred, the procedure is to also search for records that have been pushed into the overflow file area. A first-order estimate is that when o' overflows have occurred, the fetch time, T_F, increases proportionally, so that for a file that has n records,

$$T_F = (x + 1 + o'/n)\,(s + r + btt) \qquad \langle \text{simple} \rangle \; 3\text{-}34$$

This result is valid while the number of inserted records is modest, say, $o' < 0.2n$. In the next section we investigate the effect of overflow in more detail and derive a more precise value for T_F. However, since many assumptions must be made about uniformity of overflows the derivation shown is mainly to illustrate a method, rather than to provide better estiamtes for typical data retrievals.

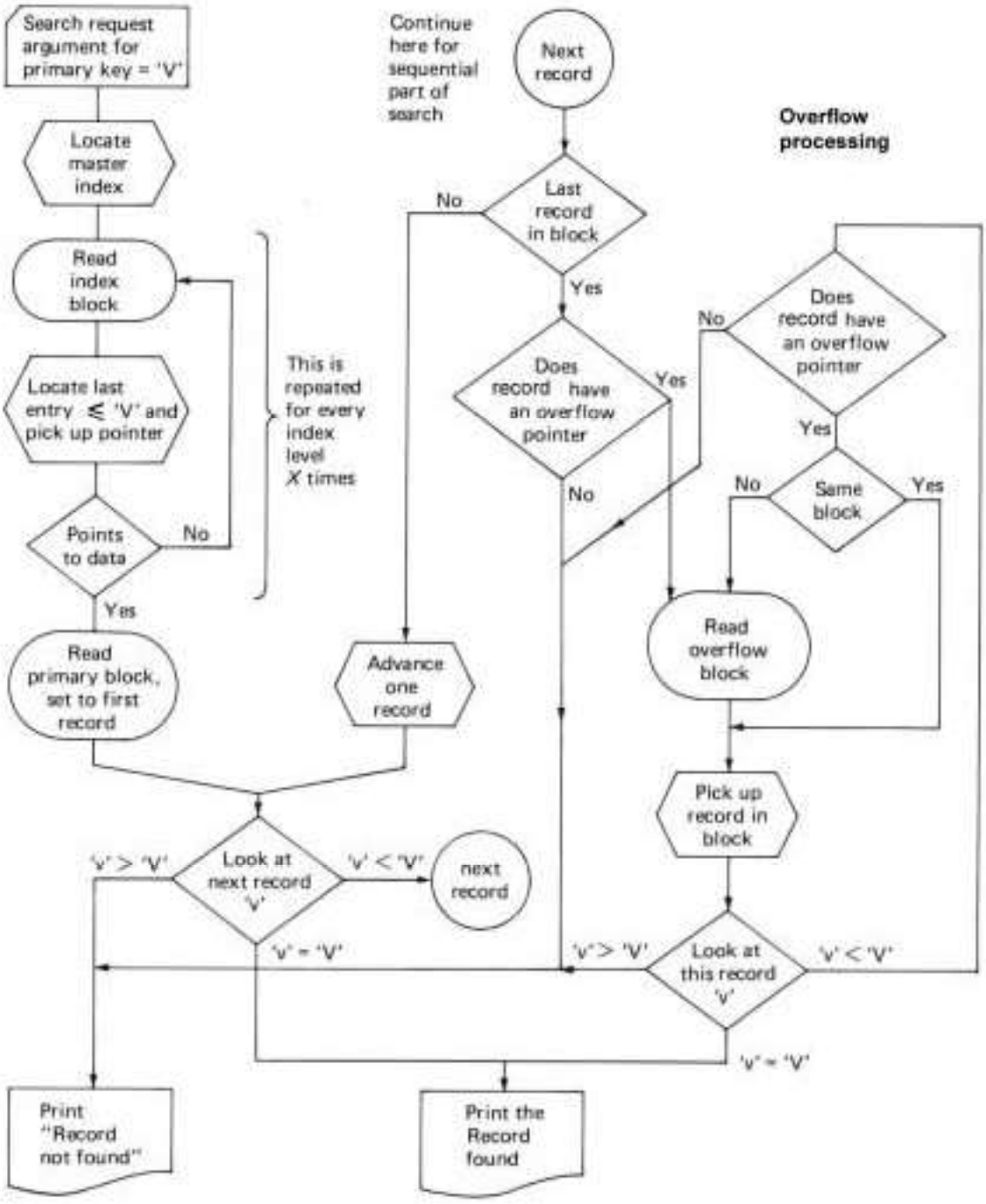


Figure 3-10 Fetch a record in an indexed-sequential file with push-through overflow.

‡‡‡‡‡ **Estimating Overflow Costs** The probability, *Pov*, that the desired record is in the overflow area depends on the number of insertions, o', that the file has received since it was reorganized. There are now n records and $n_m = n - o'$ are in the main file. The probability that a record which has been pushed into the overflow area is being fetched is

$$Pov = o'/n_m \qquad 3\text{-}35$$

The overflow records are attached in a chain to the block retrieved by the initial part of the fetch operation. The length of each chain can be estimated by considering the number of overflow records per main file block

$$Lc = o'\, Bfr/n_m \qquad 3\text{-}36$$

The number of blocks to be accessed in a search through the chain is $(Lc + 1)/2$ (see Eq. 3-2).

If the area for overflows is on another cylinder, a seek, $s^\dagger$, is required whenever the first overflow record of a chain is accessed. Any of the $Lc - 1$ further records from the chain are probably on different blocks, although on the same cylinder, and obtained at a cost of $r + btt$. The expected cost of getting to the overflow area and fetching a record is

$$T_{Foverflow} = Pov\,(s^\dagger + \frac{Lc+1}{2}(r + btt)) \qquad 3\text{-}37a$$

Both *Pov* and *Lc* are dependent on the amount of overflow, o'. If overflow areas are placed on the same cylinder the $s^\dagger$ term drops out. We can express the cost in terms of the overflows using Eqs. 3-34 and 3-35 and assume that the number of insertions was modest, so that $n_m \approx n$. Now

$$T_{Foverflow} = \frac{o'}{n}(s^\dagger + \frac{1}{2}(r + btt)) + \frac{1}{2}\frac{{o'}^2}{n}\, Bfr(r + btt) \qquad \langle\text{typical}\rangle\ 3\text{-}37b$$

The second term, although squared in o/n, may remain significant since values of *Bfr* may be such that $(o/n)^2 \times Bfr \approx o/n$.

An adequate estimate for record fetch in common indexed-sequential file configurations and usage is obtained by combining Eqs. 3-33 and 3-37b to give

$$\begin{aligned} T_F &= T_{Fmain} + T_{Foverflow} \qquad 3\text{-}38 \\ &= x\,s + x\,r + x\,btt + s + r + btt + \frac{o'}{n}s^\dagger + \frac{o'}{2n}(r + btt) + \frac{1}{2}\frac{{o'}^2}{n}\, Bfr(r + btt) \\ &= \left(x + 1 + \frac{o'}{n}^\dagger\right) s + \left(x + 1 + \frac{1}{2}\left(1 + \frac{o'}{n}Bfr\right)\frac{o'}{n}\right)(r + btt) \end{aligned}$$

Now, if $o'/n \times Bfr \approx 1$ we can combine the terms due to Eq. 3-37b. We are left with

$$T_F = (x + 1 + o'/n)\,(s + r + btt) \qquad \langle\text{shown above as}\rangle\ 3\text{-}39$$

Below are sample calculations illustrating the precision under the various assumptions. ‡‡‡‡‡

Estimation of fetch time. To estimate the fetch time with overflows we use the same file size presented in the Examples 3-6, so that x is 3. If reorganizations are made when the overflow area is 80% full, the average value of o' will be $0.4o$. If we consider a case where an overflow area equal to 20% of the prime file has been allocated, then $o' = 0.20 \times 0.4\,n = 0.08\,n$. Using the relations above, and a Bfr of 10,

$$Pov = 0.08/1.08 = 0.0741 \qquad \text{and} \qquad Lc = 10 \times 0.0741 = 0.741$$

With these assumptions we arrive at an average fetch time using Eq. 3-38 versus the simplification of Eq. 3-39.

$$T_F = 4.072(s + r + btt) \qquad \text{versus} \qquad T_F\langle simple\rangle = 4.080(s + r + btt)$$

If we use $n_m = n - o'$ instead of n we would obtain

$$T_F = 4.0645(s + r + btt)$$

Get-Next Record of an Indexed-Sequential File In order to locate a successor record, we start from the last data record and ignore the index. We have to determine whether serial reading can be done sequentially or whether we have to go to another area. We can expect to have to go to the overflow area in proportion to the ratio of records to be found there, o'/n_m, but in the main file we only have to read a new block $1/Bfr$ times. An estimate is then

$$T_N = \frac{1}{Bfr}(s + r + btt) + \frac{o'}{n_m}(r + btt) \qquad \langle\text{estimate}\rangle\ 3\text{-}40a$$

A more precise case analysis has to consider all the possibilities which exist and consider the locations of the predecessor and of the successor record. The six cases we can distinguish are illustrated using the record numbers of Fig. 3-9. The process is flowcharted in Fig. 3-11. The result, after simplification, becomes Eq. 3-41.

‡‡‡‡‡ **Path Choices for Next Record** We consider again block-anchored overflow records, and begin with the most likely case:

a The current record is in a main data file block and the successor record is in the same block, and hence already available in a memory buffer (e.g., the current record is 11.1, 11.2, 12.1, ...).

b The current record is the last one in the block, there were no insertions, and the successor record is in a following block on the same cylinder.

c The current record is the last one in the block, and there have been no insertions, but the record is in a new block on another cylinder. Given that there are β blocks per cylinder, and the file begins at a new cylinder, this would happen between records $\beta.3$ and $(\beta + 1).1$, $2\beta.3$ and $(2\beta + 1).1$, once for each cylinder occupied by the file.

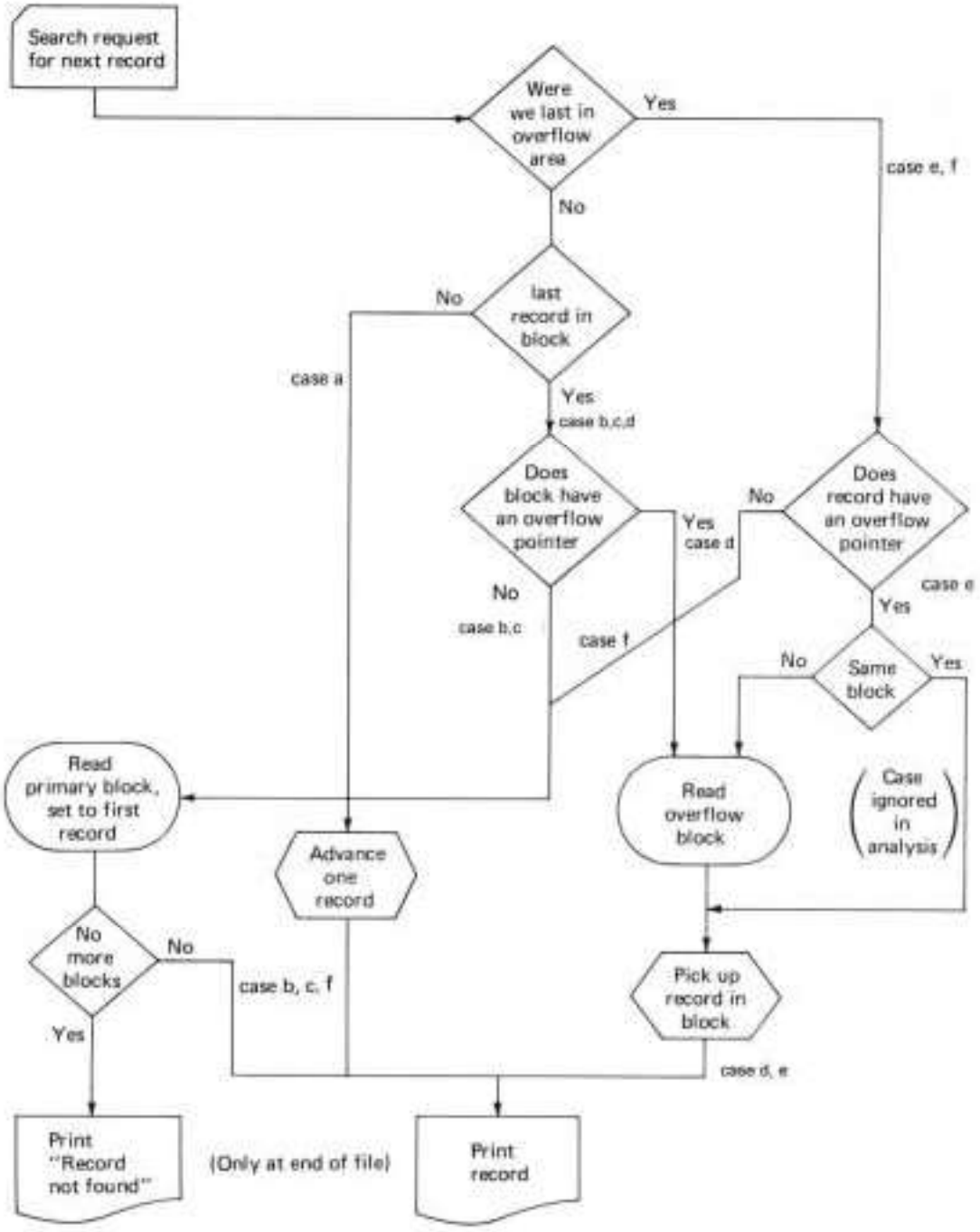


Figure 3-11 Get the successor record in an indexed-sequential file. Note that the flow in this figure overlaps Fig. 3-10 to a great extent.

d The current record is the last one in the block, but there has been an insertion, and the successor record will be in an overflow block (the current record is 11.3 or 12.3).

e The current record is an inserted record, and the successor is found following the chain to another overflow block on the same cylinder (current record is 110.1 or 111.1).

f The current record is an inserted record, and to get to the successor record a new data block has to be read (current record is 110.2, 110.3, or 111.2).

We will evaluate each of these cases using probabilities of the events which lead to their occurrence. The following notation is chosen to define the more likely condition of each event:

Pd: The current record is in a primary data block $= 1 - Pov$.
Pb: The successor record is in the same block $= 1 - 1/Bfr$.
Pm: There has been no insertion into the block $= 1 - Bfr\ Pov$.
Pc: The next block is in the same cylinder $= 1 - 1/\beta$.
Pl: The current overflow record is not the last of the chain. $= 1 - 1/Lc$

These values can now be used for the cases shown in Fig. 3-12.

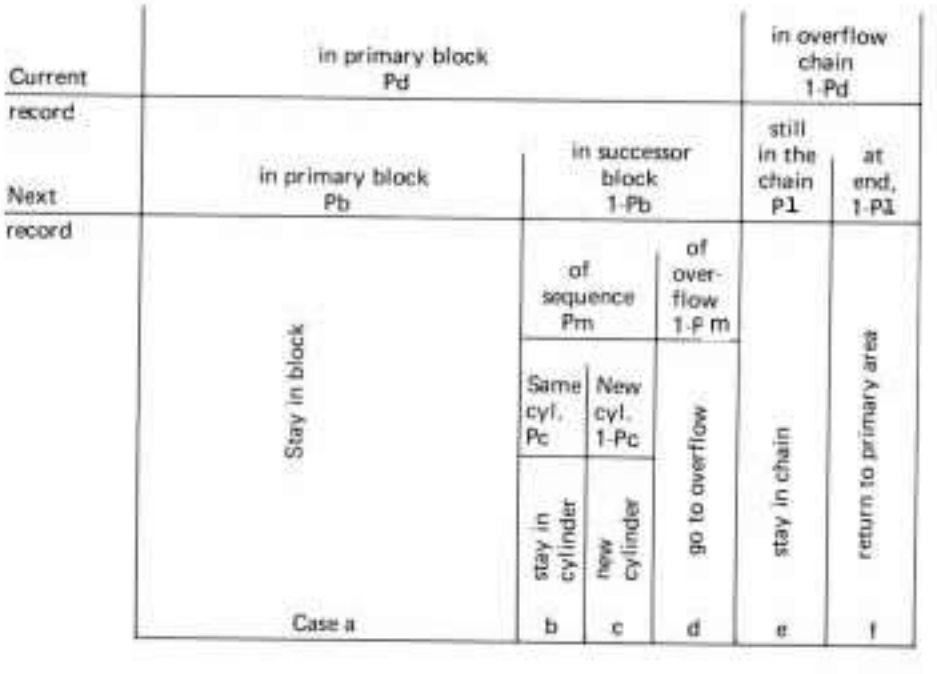


Figure 3-12 Conditions in the search for a successor record.

Figure 3-12 indicates the conditions based on Fig. 3-11. By applying the appropriate combinations of probabilities to their costs, we obtain for the cases considered

$$
\begin{aligned}
T_N = \quad & (Pd)(Pb)(c)+ && /* \, case\ a \, */ \\
& (Pd)(1-Pb)(Pm)(Pc)(r+btt)+ && /* \, case\ b \, */ \\
& (Pd)(1-Pb)(Pm)(1-Pc)(s+r+btt)+ && /* \, case\ c \, */ \\
& (Pd)(1-Pb)(1-Pm)(s^{\dagger}+r+btt)+ && /* \, case\ d \, */ \\
& (1-Pd)(Pl)(r+btt)+ && /* \, case\ e \, */ \\
& (1-Pd)(1-Pl)(s^{\dagger}+r+btt) && /* \, case\ f \, */
\end{aligned}
\qquad \text{3-40b}
$$

The seek terms in cases d and f, marked with a †, disappear when the overflow areas are kept on the same cylinders as the corresponding data blocks. The probabilities Pm and Pl can be rewritten in terms of n and o' using Eqs. 3-35 and 3-36.

Simplification of Get-Next Estimate Equation 3-40 was obtained by a detailed case analysis, but as is typical, the result can be made practical by consideration of the relative importance of the terms. If we neglect the memory search time, c, for records within the block, if cylinder seeks can be ignored (i.e., β is large, so that the value of $Pc \approx 1$), and if the overflow areas are on the same cylinders as the data blocks ($s^{\dagger} = 0$, so that cases b and d, as well as e and f combine), then

$$T_N \approx \left(\frac{1 - Pov}{Bfr} + Pov \right) (r + btt) = \frac{n + o' \, Bfr}{(n + o')Bfr} (r + btt) \qquad 3\text{-}41$$

We note that the chain length Lc does not affect this approximation for T_N, but overflows still increase the estimate. ######

Insert into an Indexed-Sequential File Adding a record to the file will always cause an addition to the overflow chain, either because of push-through or because the new record follows serially a record already in the overflow chain. Each insertion requires the reading and rewriting of a predecessor data or overflow block, since a pointer will have to be inserted or changed and also a `READ` and `REWRITE` of the overflow block for the pushed or inserted record. The probability of using the same overflow block for both is small if records are randomly inserted into a large data file. We avoid a detailed case analysis now by using the previous result for T_F and making some further assumptions. The fetch time for the predecessor is equal to T_F, the overflow block is on the same cylinder and requires $r + btt$ to be reached, and each `REWRITE` will take one revolution $T_{RW} = 2r$ (given the conditions of Eq. 2-30), so that

$$T_I = T_F + T_{RW} + r + btt + T_{RW} = T_F + 5r + btt \qquad 3\text{-}42$$

The effect of the length of the overflow is incorporated into T_F.

Note that the index is not affected when a record is inserted. The cost of insertion is only due to appending the record to the chain and linking it to the predecessor record. Alternative issues of chain management, discussed earlier, may be considered.

Update Record in an Indexed-Sequential File An updated record of equal size and identical key can be placed into the place of the previous version of the record, so that the process can be evaluated as a fetch followed by a `REWRITE` with a cost of T_{RW}:

$$T_U = T_F + T_{RW} = T_F + 2r \qquad \langle \text{in place} \rangle \; 3\text{-}43$$

Deletion of a record, done by setting a tombstone into the record, is also done using this process. Equation 3-43 is always appropriate for systems which disallow both the updating of key fields and variable-length records.

In the general case, the previous version of the record has to be deleted and the new record inserted appropriately. The old record is rewritten with the tombstone; the key and pointer fields are kept intact so that the structure of the file is not violated. Then

$$T_U = T_F + T_{RW} + T_I = 2T_F + 7r + btt \qquad \langle \text{in general} \rangle \; 3\text{-}44$$

If the cases which permit in-place updates are recognized by the file system, then T_U is to be computed based on the mix of in-place and general updates.

Read Entire Indexed-Sequential File An exhaustive search of the file has to be made when the search argument is not the indexed attribute. The file may be read serially by following the overflow chains for every block, or if seriality is not required, the entire data area on a cylinder can be read sequentially, followed by sequential reading of the entire overflow area. In either case, the index can be ignored unless it contains space-allocation information.

Most systems provide only the ability to read serially, so that

$$T_X = T_F + (n + o' - 1)T_N \approx (n + o')T_N = \frac{n + o'\,Bfr}{Bfr}(r + btt) \qquad \langle \text{serial} \rangle \text{ 3-45}$$

The assumptions leading to Eq. 3-41 are valid here.

In the alternative case the evaluation would consider the effective transfer rate, neglecting the delay when skipping from data blocks to overflow blocks. Unused overflow blocks will not be read, so that o' can be used to estimate the size of the overflow areas read. Now

$$T_X \approx (n + o')\frac{R}{t'} \qquad \langle \text{sequential} \rangle \text{ 3-46}$$

Reorganization of an Indexed-Sequential File To reorganize the old file, the entire file is read serially and rewritten without the use of overflow areas. As a by-product a new index is constructed. The prior index can be ignored, since the file is read serially. Additional buffers in memory are needed to collect the new data and index information. For the index it is desirable to have at least one buffer for every level. The new mainfile should be at least double-buffered. All outputs can be written sequentially. Then

$$T_Y = \frac{n + o'\,Bfr}{Bfr}(r + btt) + (n + o' - d)\frac{R}{t'} + \frac{SI}{t'} \qquad \text{3-47}$$

We assume that o' new records are in the overflow areas; however, the value of o' will be larger at reorganization time than in the cases considered in Eqs. 3-33 to 3-46. The value of o' will still be less than o, the number of records for which overflow space has been allocated.

Following the discussion of reorganization in Chap. 3-3-2(7), we assume that $o' = 0.8\,o$. Such a value is justified if the reorganization policy were as follows:

> *Reorganization of a file is to be done the first night the overflow area exceeds 75% utilization, given that the average daily increase of the overflow area is 10%.*

A simpler assumption that $o' = o$ will provide a conservative approximation for the number of overflow records to be processed.

3-4 THE INDEXED FILE

Indexed-sequential files only provide one index, but searching for information may have to be done on other attributes than a primary key attribute. In a generalized indexed file we permit multiple indexes. There may be indexes on any attribute, and perhaps on all attributes. A number of changes to the file organization follow from that extension.

- All indexes are treated equally:
  1 All indexes are record-anchored.
  2 The concept of a primary attribute is not retained.
  3 No sequentiality according to a primary index is maintained.
- No overflow chains can be maintained:
  1 Any insertions are placed into the main data file.
  2 The main file format should make insertion convenient.
  3 All indexes must be updated to reflect insertions.

We expand on the trade-offs implied above in more detail throughout this section.

By givng up the requirement for sequentiality to provide efficient serial access, much flexibility is gained. In the generalized indexed file the records are accessed only through their indexes. There is now no restriction on the placement of a data record, as long as a TID exists in some index that allows the record to be fetched when the goal data from the record is wanted. Each index is associated with some attribute.

The gain in flexibility obtained makes this file organization preferable to the indexed-sequential file organization in many applications. The actual physical placement and format of records in generalized indexed files can be determined by secondary considerations, as ease of management or reliability. Having indexes on more than one attribute greatly increases the availability of the data in information retrieval and advanced processing systems. Variable-length records are common in these applications.

The flexibility of generalized indexed files has created a great variety of actual designs. The variety of designs has unfortunately also created a diversity of terminology, often quite inconsistent, so that anyone intending to evaluate a specific approach to indexed files will have to translate terms used in describing such systems into standard concepts. We will evaluate again a specific approach which is becoming increasingly common, based on the use of *B-trees* for the indexes.

Figure 3-13 shows schematically three indexes into a `Personnel` file, for the attributes `Names`, `Professions`, and `Chronic_diseases`. Each of the variable-length spanned records of the file can be located by giving a name or profession value. The third record, in block 2 at position 35, has a field with the attribute name `Chronic_diseases` and can also be located via that index.

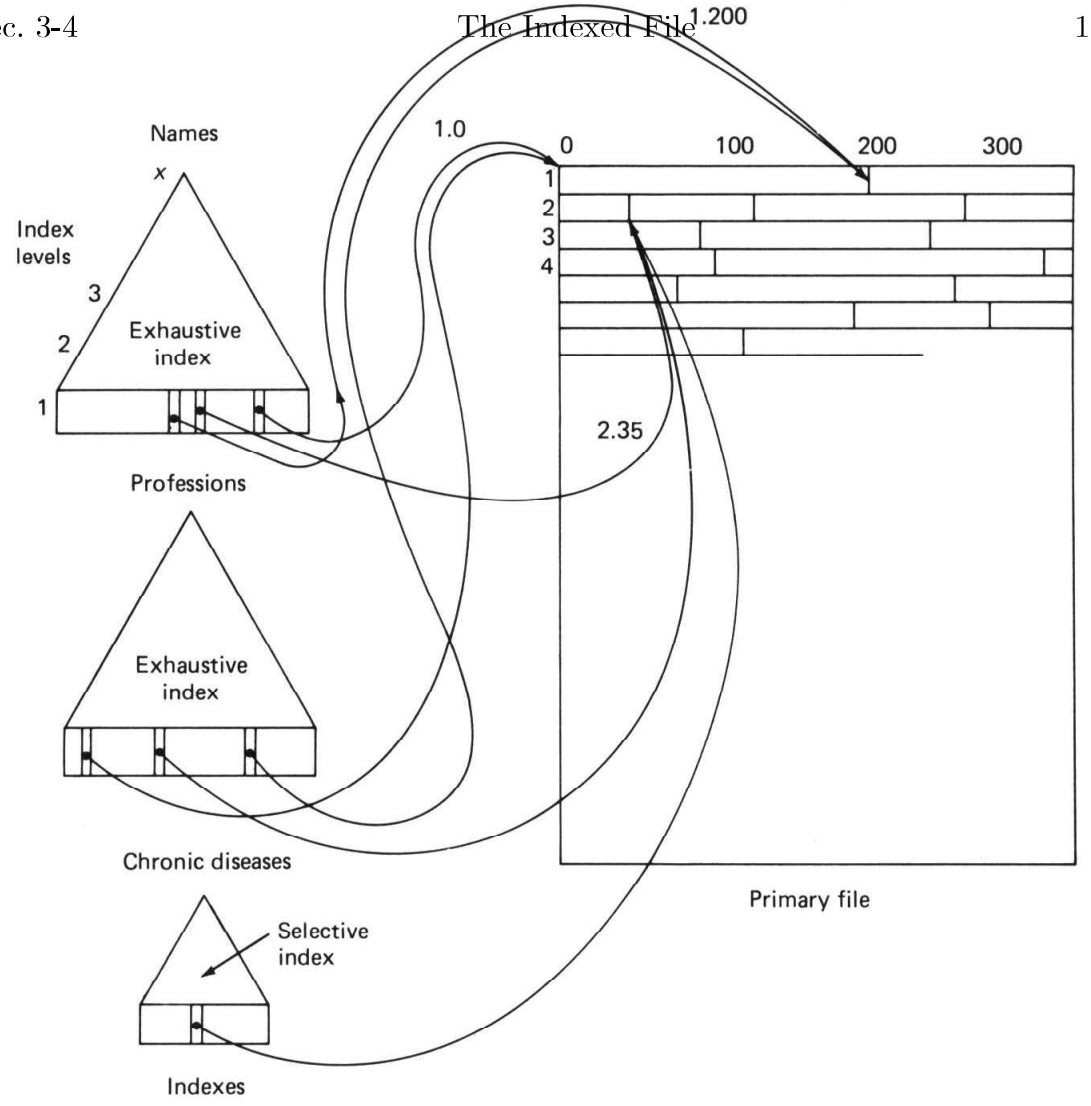


Figure 3-13 Record linkage in an indexed file.

Directories Since an indexed file with multiple indexes consists of several distinct component files we must be able to determine what indexes exist, how they are organized, and where they are located. An extension of the notion of the file directory, introduced in Sect. 3-0-1, can satisfy that role. Programs which retrieve data or update the file must consult that directory to carry out their function. In database management systems *schemas* take on that role, as indicated in Chap. 8-2. For our evaluations we assume that such a directory is read into a memory buffer when the file is opened, and can be rapidly searched.

3-4-1 Structure and Manipulation of Indexed Files

There may be as many indexes as there are attribute columns in the file; even more if we build combined indexes for fast partial-match retrieval as described in Sec. 4-2-5. An index for an attribute of an indexed file consists of a set of entries, one for every record in the file. We cannot use block-anchors here to reduce the index size, as we did in the indexed-sequential file, since index and data are not cosequential.

The entries are ordered as determined by the attribute values. Each entry consists of the attribute value and a TID. In indexed files successor records are reached using the next index entry rather than by sequentiality or via pointers from the predecessor record. Each index may again require multiple levels, just as we have seen in an indexed-sequential file.

The data record format may be similar to any of the previous organizations. Records containing attribute name-value pairs, as seen in the pile file, are the choice permitting greatest flexibility; otherwise, structured records may be employed. Since the TIDs in the index specify the block address and record position for every record, there is essentially no restriction on record size or on the placement of records within a specific block. Records can be inserted wherever the file system finds sufficient free space.

Maintenance of Indexes The major problem when using indexed files is that all the indexes to a record must be updated whenever a record has been added or deleted or is moved. A single index has to be changed when a single data field value of an indexed attribute is updated.

In indexed-sequential files dynamic updating of the index was avoided through the use of pointer chains to insertions. With indexing for multiple attributes such overflow chaining is not feasible; each record would need many overflow pointers. The pointers would have to link a new record to many predecessor records. The alternative is to update the actual indexes as the file changes, but a dense index structure as used with indexed-sequential files would require for each insertion or deletion of an index entry the rewriting of the entire index.

Exhaustive and Partial Indexes Indexes may be *exhaustive*, that is, have TIDs to every record in the file; or *partial*, that is, have only TIDs to records where the attribute value is significant. Significance can mean simply that a value exists (is not undefined or zero) or that the value has a good selectivity.

A partial index could occur in a personnel file which includes health data. Only indexes to current employee health problems are maintained, although complete historical data is kept in the records for review purposes. This was shown symbolically in Fig. 3-13 as a `Chronic_diseases` index; here `NULL` or `Healthy` data values do not lead to entries. Another partial index could identify all individuals that smoke, to help in risk assessments.

If there is not even one exhaustive index, there is no easy way to list each record once. Now a space allocation table is required to allow the file to be properly maintained. Such a table gives an accounting of all the space allocated to the file. If the file is read serially according to this table, a record ordering similar to the pile file may be perceived. #####

A Concordance One older use of computerized indexing by linguists is a *concordance*. Here a single index is created to show all the words in a body of writing. Such an index has as entries all the unique words (types) in the file, and is essentially a vocabulary. With each entry there will be pointers to all the instances (tokens) in the text where the words appear. In a printed concordance the pointers are augmented by a sample of the text. An excerpt from a concordance is shown in Fig. 3-14.

Following our concepts a text file has only one type of attribute, *words*, and the index has multiple entries per record, one for each word. Partial inversions may exclude high-frequency words or other words not of interest, such as initials of authors.

Sample of text

Key-attribute value Pointer

```
. quant vit pasmer  Rollant, / dunc out tel doel unkes mais n'out si grant. / Tendit sa mai 2223
 la sele en remeint quaste. / Mult ad grant doel  Carlemagnes li reis, / quant  Naimun veit 3451
c. / Co dist li reis: "Seignurs, vengez voz doels, / si esclargiez voz talenz e voz coers,  3627
chevaler." / Respont li quens: " Deus le me doinst venger!" / Sun cheval brochet des esperu 1548
ad mort  France ad mis en exill. / Si grant dol ai que ne voldreie vivre, / de ma maisnee,  2936
d sanc. / Franceis murrunt,   Carles en ert dolent. /  Tere  Majur vos metrus an present.    951
ent, / e cil d' Espaigne s'en cleiment tuit dolent. / Dient  Franceis: "Ben fiert nostre gu 1651
alchet ireement, / e li  Franceis curucus e dolent; / n'i ad celoi n'i plurt e se dement,   1825
 ma gent." / E cil respunt "Tant sy jo plus dolent. / Ne pois a vos tenir lung parlement:   2835
 sil duluset; / jamais en tere n'orrez plus dolent hume! / Or veit  Rollant gue mort est su 2023
 devers les porz d' Espaigne: / veeir poez, dolente est la rereguarde; / ki ceste fait, jan 1104
e vient curant cuntre lui; / si li ad dit: "Dolente, si mare fui! / A itel hunte, sire, son 2823
pereres cevalchet par irur / e li  Franceis dolenz e curucus: / n'i ad celoi ki durement ne 1813
aienur, / plurent e crient, demeinent grant dolor, / pleignent lur deus.  Tervagan e Mahum  2695
perere,' co dist  Gefrei D' Anjou, / "ceste dolor ne demenez tant fort! / Par tut le camp f 2946
ance ad en baillie, / que me remembre de la dolur e l'ire, / co est de  Basan e de sun frer  489
amimunde, / pluret e criet, mult forment se douset; / ensembl'od li plus de xx. mil humes,  2577
out mais en avant. / Par tuz les prez or se dorment li  Franc. / N'i ad cheval ki puisset e 2521
ad apris ki bien conuist ahan. /  Karles se dort cum huse traveillet.  Seint  Gabriel li a  2525
poeent plus faire. / Ki mult est las, il se dort cuntre tere. / Icele noit n'unt unkes esca 2494
it le jur, la noit est aserie. /  Carles se dort, li empereres riches. /  Sunjat qu'il eret  718
ent liquels d'els la veintrat. /  Carles se dort, mie ne s'esveillat. AOI. / Tresvait la no  736
le cel en volent les escicles. /  Carles se dort, qu'il ne s'esveillet mie. / Apres iceste,  724
s  Deu co ad mustret al barun. /  Carles se dort tresqu'al demain, al cler jur. / Li reis   2569
et les os, / tute l'eschine li desevret del dos, / od sun espiet l'anse li getet fors,       1201
gemmet ad or, / e al cheval parfundement el dos; / ambure ocit, ki quel blasme ne quil lot. 1588
eruns a or, / fiert  Oliver derere en mi le dos. / Le blanc osberc li ad descust el cors,   1945
ros; / sur les eschines qu'il unt en mi les dos / cil sunt seiet ensement cume porc. AOI.   3222
re joe en ad tute sanglente; / l'osberc del dos josque par sum le ventre. /  Deus le guarit 3922
ele les dous alves d'argent / e al ceval le dos parfundement; / ambure ocist seinz nul reco 1649
t li ber. / De cels d' Espaigne unt lur les dos turnez, / tenent l'enchalz, tuit en sunt cu 2445
a fuls: / de cent millers n'en poent quarir dous. /  Rollant dist: "Nostre ume sunt mult p  1440
s e l'osberc jazerenc, / de l'oree sele les dous alves d'argent / e al ceval le dos parfund 1648
tet en ad, ne poet muer n'en plurt. / Desuz dous arbres parvenuz est . . . li reis. / Les c 2874
 Dedesuz  Ais est la pree sult large: / des dous baruns justee est la bataille. / Cil sunt  3874
agne, ki est canuz e vielz! / Men escientre dous cenz anz ad e mielz. / Par tantes teres ad  539
t vielz, si ad sun tens uset; / men escient dous cenz anz ad passet. / Par tantes teres at   524
```

Figure 3-14 Sample of a concordance. (From Joseph J. Duggan, "A Concordance of the Chanson de Roland"; Ohio State University Press, Columbus, 1969.) ▨

Index Selection Many file and database systems do not have the capability to manipulate TIDs. In that case it becomes very important to select the most selective index for a query. Even when we can combine indexes as shown above, it may not be worthwhile to merge indexes with a relatively low selectivity. The attribute `bedrooms` in Ex. 3-4 is of dubious value. We discussed earlier that attributes with absolute poor selectivity should not even be indexed.

Estimation of selectivity of an attribute is often difficult. Most systems make the assumption of a uniform distribution. Unfortunately, uniform distributions are rare in practice unless the partitioning of the attribute values has been planned that way. Most common in practice are *Zipfian distributions*, as presented in Chap. 14-3-2. Other distributions, as *normal* and *Erlang*, are covered in Chap. 6-1. All these are distinguished by having frequent occurrences of some values, and fewer occurrences of other values.

Table 3-4 shows some attributes and their likely distribution of values. Then we show for each attribute a sample value expression for a query, and the estimated percentage of the file retrieved using the assumption of a uniform distribution, followed by the actual value, obtained from an experiment using real data.

Table 3-4 Selectivity of some attributes.

Attribute	Range or cardinality	Type of distribution	Query	Estimate of Result	Actual Result
`Style`	$n = 20$	Zipf	`=Colonial`	5%	13%
`Price`	\$12K – 262K	normal	`>212K`	20%	3%
`Bedrooms`	$n = 8$	Erlang(2)	`=3`	12.5%	42%
`Location`	$n = 20$	uniform	`=Wuth.Heights`	5%	4.5%

The estimate can be improved if for each attribute additional information is kept. For small value sets the actual counts may be kept. Counts would provide exact values for the `Style`, `Bedrooms`, and `Location` attributes. Keeping, instead of the range, ten decile boundaries of the value distribution permits a more precise estimation of continuous values, as `Price` above. A better scheme is to maintain *distribution steps*, where the partitions have equal sizes.

Distribution information tends to be associated with indexes, and is best updated as the indexes are. When only a few parameters are kept to describe the distribution, we can expect to find the information in the root block and rapidly retrieved. The values in the root block provide already an initial estimation of the distribution steps, since in the sizes of the subtrees for each entry will be similar. For instance, given that the root block of the `Price` index has 100 entries we would find that the value $= 212K$ will appear in entry 97 or 98, giving a much more accurate estimate of the selectivity than that provided by the assumption of uniform distribution.

All these techniques still ignore the correlation of values. In `Wuth.Heights` the dominant `Style` may be Greek Revival and the `Price`s always above \$150 000. The storage and update demands for such second-order information make its retention impractical.

An Index Structure for Dynamic Updating — B-trees In indexed files every change of an indexed attribute will require the insertion, deletion, or both, of an index entry into the appropriate index block. To make such changes feasible, we reduce the density of the index. Extra space is left initially empty in each index block, and now one insertion will affect only that index block. The blocks can accommodate a number of insertions, and only when the block is filled is another block obtained. Half the entries from the full block are distributed to the new block. There has been a trade-off made here: space has been given up to have reasonable maintenance of the index.

A method based on this solution is the *B-tree*, and we will describe a specific version of this algorithm useful for index trees. A B-tree has index blocks which are

kept at least half full; the effective fanout y_{eff} is hence between y and $y/2$ as shown in Fig. 3-15.

‡‡‡‡‡ **B-trees and Index B-trees** The original B-tree was defined with space in each block for y pointers and $y-1$ values. In a block of an index B-tree, however, up to y values are kept, and the value ($\mathtt{v}_1$) appears redundantly, as shown in Fig. 5-14. The value in the first entry ($\mathtt{v}_1$) is the same as the value ($\mathtt{v}_n$) in the entry referring to this block in the next-higher-level index block. This redundancy is avoided in pure B-trees. The index B-tree approach makes the entries consistent in format, permits index blocks to be accessed independently of their ancestor blocks, and has little effect on the analyses when y is reasonably large. ‡‡‡‡‡

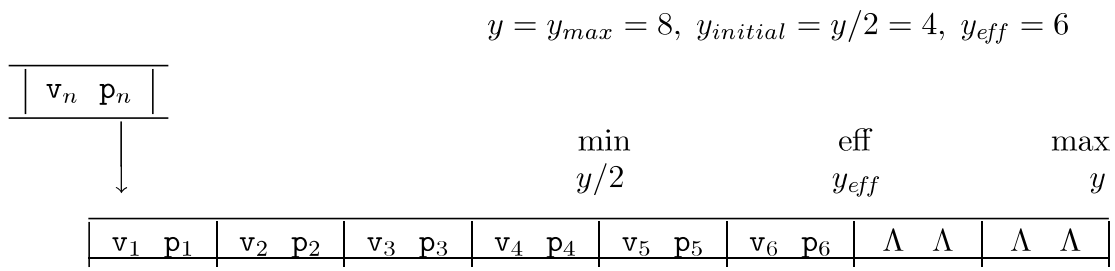


Figure 3-15 Block of a B-tree index after two insertions.

Index B-tree Algorithms The insertion and deletion algorithms for B-trees must maintain the condition that index blocks are at least half full.

Insertion New entries are inserted into the first-level index blocks until the limit y is reached. The next entry to be inserted will require the allocation of a new, empty index block, which is then initialized with half the entries taken from the block which was full: The block has been *split*. The entry which forced the split can now be inserted into the appropriate level one index block. At the same time a new entry has to be created for the new index block at the next higher level, containing a pair$\{\mathtt{v}_{n+1}, \mathtt{p}_{n+1}\}$. The value $\mathtt{v}_{n+1}$ is the former $\mathtt{v}_{y/2+1}$ taken from the split block, which is now $\mathtt{v}_1$ in the new block.

The next-level block may in turn be already full and also require such a split. If the top or root block is full, a new root block is created and initially filled with two entries, one for the previous root block and one for its new partner. The tree has now grown by one level. We note that the root block only may have fewer than $y/2$ entries; y_{eff} may be anywhere from 2 to y but we ignore the effect due to the root block in our analyses.

This insertion algorithm maintains the value of y_{eff} in the desired range; the deletion algorithm below does the same. For a file receiving only insertions the average y_{eff} for the entire index will become $0.75\,y$, but we will henceforth use a result of Yao[78], which specifies that under conditions of random insertions and deletions the B-tree eventually achieves a density of $y_{eff}/y \to \ln 2 = 0.69$. Then

$$y/2 \le y_{eff} \le y \qquad \text{or} \qquad y_{eff} \to \ln 2 \times y = 0.69\,y \tag{3-48}$$

where y is again defined as in Eq. 3-26 as $\lfloor B/(V+P) \rfloor$. In order to simplify the analysis, we also assume an initial loading density of 0.69.

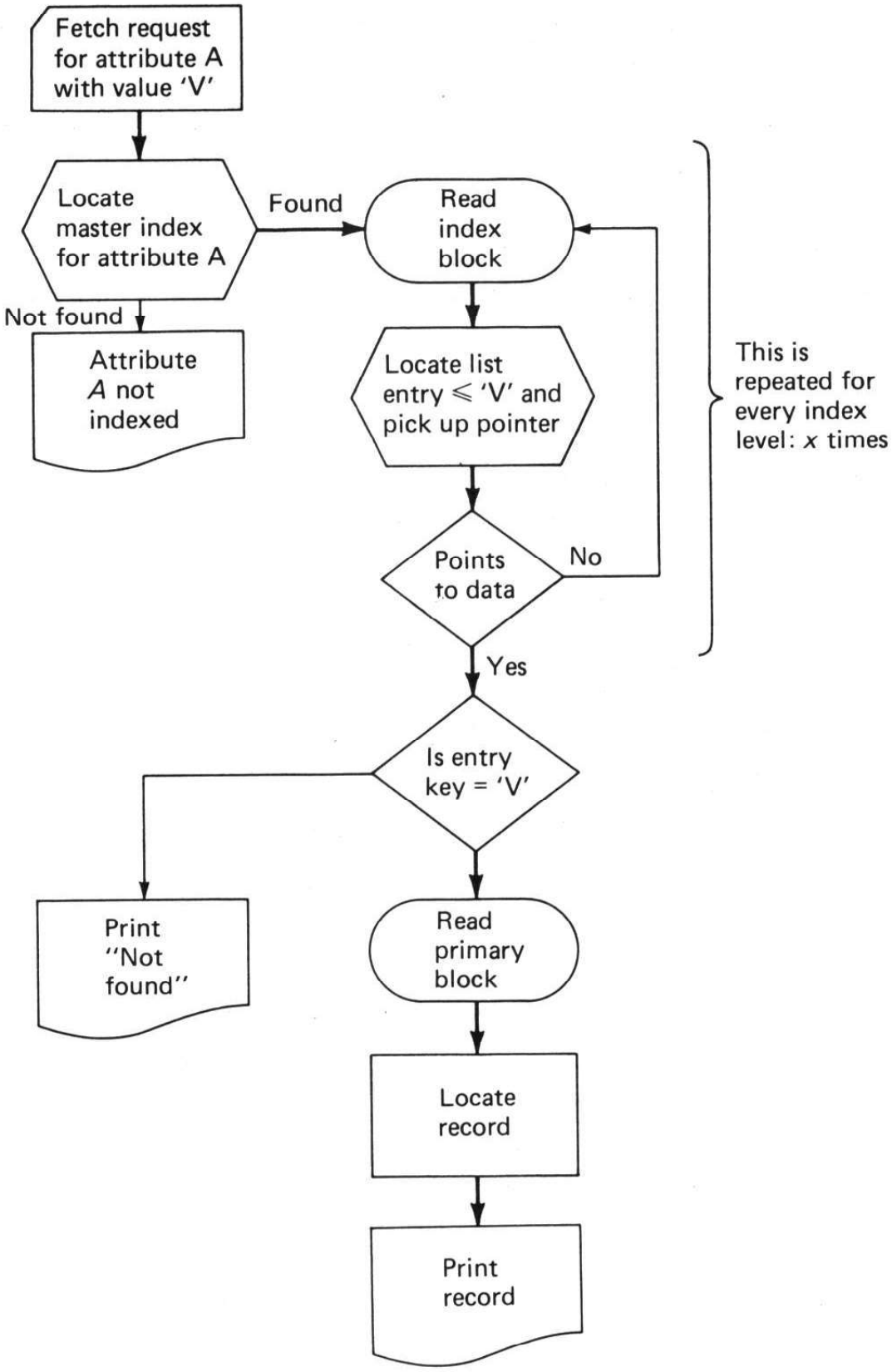


Figure 3-16 Fetch using one index of a multi-indexed file.

Deletion in a B-tree When an entry is deleted, an index block may be left with fewer than $y/2$ entries. Its partner should now be inspected. If the total number of entries in both is less than y, they should be combined. An entry in the higher-level index block is now also deleted, and this deletion may propagate and even lead to the deletion of an entry in the root block. If the root block has only one entry left, it can be deleted so that the height of the tree is reduced by one.

In practice, deletion in B-trees is often modified. For instance, if the total of two blocks is exactly y, then it is best not to combine the blocks to avoid excessive costs if deletions and insertions alternate. Furthermore, since the cost of inspecting the partner index blocks for its occupancy is so high, testing for block merging is often deferred.

The Height of B-trees The number of levels of indexing required is a function of the number of index entries, y_{eff}, that appear in one index block. In order to evaluate the height of the B-trees we will assume here that stability has been achieved and that all index levels partition their data evenly. Given that one index will refer to n' records, where n' is determined by the expected number of records having indexable attribute values, we find, similarly to Eq. 3-27,

$$x = \left\lceil \log_{y_{eff}} n' \right\rceil \qquad 3\text{-}49$$

We note that because of the reduced density and fanout the height of the index B-trees is greater than the height of a dense index. The height, x, will be one level greater in many practical cases. A greater increase occurs only in large files with small fanouts, say, $n' > 10^6$ and $y < 12$. With large fanouts there will be wide ranges of n' where the index height is the same for a dense index and a corresponding B-tree.

The height of a B-tree controls the time needed to retrieve records. The search process is flowcharted in Fig. 3-16.

‡‡‡‡‡ **Deletion Pragmatics in B-trees** Deletion in B-trees carries with it the high cost of inspecting the partner blocks. In order to find out if the index blocks, where a deletion has occured, can be combined with a partner, the occupancy of its partner has to be such that sum of the entries is y or less.

Many systems use alternate deletion strategies, either to simplify the programs or to reduce effort. Perhaps index blocks will not be checked until one has fewer than $0.69/2y$ entries or will not even be combined at all. In the last case index blocks will be deleted only if they become empty. Often deletion is deferred, as discussed next, in a more general sense. ‡‡‡‡‡

Deferred Updating of Indexes A solution is to defer the updating of the indexes. The updating of the indexes is then done at a lower priority, when the computer is relatively idle. The data file is kept available. Deferred index updating will be done by separate processes which are active at a modest priority until completed. It is necessary to keep the changed or deleted records in the main file, marked with a tombstone, until all referring indexes have been updated.

A retrieval to such a file may now not include recent changes, and may include records which have actually been deleted. In some applications that may not matter. Many applications can operate with deferred updating. When queries involve

management decisions for planning, say, an analysis of a `sales` pattern, use of data which is perhaps a few hours obsolete will not matter. These types of analysis transactions tend to access many records so that a relaxation of their access requirements can be very beneficial to the overall system.

A counterexample is a `sales` transaction which actually commits to deliver specific items from stock. Then an immediate update of the index used for access may have to be forced to assure the stock counts are up to date.

Alternative methods to assure that critical retrievals are up to date when deferred index updating is used are:

1 Assign to the reading process a yet lower priority than the index update process, so that index updating will be completed before any critical retrieval is performed.
2 Mark the indexes which have not yet been updated, and force an index update for attributes used in the retrieval.
3 Search in parallel through the list of records in the process queue for insertion or update to locate any conflicts.

The third method is computationally the costliest, but will not require forced updating of the indexes. In all cases deleted records are recognized by their tombstones.

High-priority fetch requests using the file may be processed before all indexes are brought up to date, with a warning that the response reflects all changes made before a certain time, say $m = 10$ minutes ago. A relatively higher priority may be assigned routinely to a process which updates important attribute indexes than the priority given to other index-update processes. If the priority given is higher than the priority of certain retrievals, these retrievals will always find up-to-date information.

Deferred updating of indexes can also reduce the total effort spent on updating when updates are frequent, or tend to come in batches. The continuous maintenance of indexes is quite time-consuming and a batch of updates may rewrite the same index block again and again.

Thus, we also find systems where indexes are created for a particular analysis but are not kept up to date as the file changes. Without updated indexes recent information will not be accessible. A new index is created when needed, or perhaps periodically. This approach is akin to a periodic file reorganization.

Updating of indexes while the file is otherwise active requires careful sequencing of update operations and the maintenance of status information on incomplete updates. Approaches to monitor concurrent file operations and maintain reliability will be discussed in Chap. 11-1.

3-4-2 Use of Indexed Files

Indexed files are used mainly in areas where timeliness of information is critical. Examples are found in airline reservation systems, job banks, military data systems, and other inventory type applications. Here data is rarely processed serially, other than for occasional, maybe only yearly, stocktaking.

When an item of information is obtained, e.g., an available seat on a certain flight, the data should be correct at that point in time, and if the item is updated,

i.e., a seat is sold on that flight, that fact should be immediately known throughout the system.

Having multiple indexes to such information makes it possible that one can find the same data by flight number, by passenger name, by interline transfer record, and so forth, without file reorganization or data redundancy. There is now, of course, redundancy between the contents of the index and the data.

Other instances where indexed files are desirable occur when data is highly variable and dynamic. The flexibility of the record format and record placement available with generalized indexed files does not exist in other file systems. Chapter 8 is devoted to specific alternatives and examples of indexed and related files.

The use of indexed files is increasing, specifically on modern systems, where new software is being developed. There is also more literature in this area than about any other file organization.

3-4-3 Performance of Indexed Files

Indexed files are easier to evaluate than indexed-sequential files. B-trees have a very predictable behavior since all search operations require an effort based on the height of the index tree. The critical design decision is the selection of the attributes that are to be indexed.

The space required for exhaustive indexes for all attributes will easily exceed the size of the original file. In practice, there are always some attributes for which indexing is not justified. Attributes that have low selectivity, as defined in Section. 3-1-1, are poor candidates for indexing. Not having indexes for these attributes, or the use of partial indexes, will reduce the size of the index space and accelerate updating. Any searches must avoid the use of omitted indexes.

> The evaluation which follows considers a completely indexed file, so that all existing attributes of every record are indexed. There will hence be a indexes with up to n, say n', entries. Each index is a B-tree, and the main data file has the form of a pile file. The records may be of variable length and have a variable number of attribute name-value pairs. Insertions are placed into the pile file itself and cause updating of all indexes.
>
> We also assume that steady state has been achieved, so that the density of the index is assumed to be stable at $dens = y_{eff}/y = 0.69$ (Eq. 3-48). This means that deletion is done so that partners have never jointly fewer than y entries. The actual space requirement for the index entries is now $1/0.69 = 1.44$ of the net requirement. Note also that the height of the index tree may be greater than the height of a denser index.

Record Size in an Indexed File The space required for the data portion of such a file is identical to the space required for a pile file, as derived for Eq. 3-1. In addition, there will be a indexes to provide an index for every attribute. Since the data attributes are sparse, there are only a' attribute entries per record; the average index contains

$$n' = n \frac{a'}{a} \qquad 3\text{-}50$$

index entries referring to data records.

Each index entry is of size $V_{index} + P$ and the size of one index is estimated as

$$SI(\text{one})_1 = n' \frac{(V_{index} + P)}{dens} = 1.44 \frac{n\,a'}{a}(V_{index} + P) \qquad \langle \text{one index} \rangle \text{ 3-51}$$

Higher-level index blocks require little additional space; for instance, the second level adds only $SI_2 \approx SI_1/y_{eff}$ bytes. To account for these higher index levels, we can add 5% to SI, changing the factor 1.44 to 1.5. The total index space becomes

$$SI_{total} = a \sum_{i=1}^{x} SI_i \;\approx\; 1.5\, n\, a'(V_{index} + P) \qquad \text{3-52}$$

The space requirement per record is equal to the sum of data and index space. No overflow area exists.

Since all attribute values which exist in a given record are indexed, the space allocated to a record is

$$\begin{aligned} R_{total} &= R_{main} + a'\, R_{index} \\ &= a'(A + V + 2) + 1.5\, a'(V_{index} + P) \end{aligned} \qquad \text{3-53}$$

for index and data. If V_{index} is taken to be equal to V, then

$$R = a'(A + 2.5V + 1.5p + 2) \qquad \langle V_{index} = V \rangle \text{ 3-54}$$

based on Eqs. 3-48 and 3-53. An evaluation is shown in Example 3-8.

Example 3-8 Calculation of levels for an indexed file.

Consider a `Personnel file` of $n = 20\,000$ `employees`, containing an inventory of their `skills`. Each `employee` has an average of 2.5 `skills` so that for this particular index a'/a is actually greater than 1. The skills are given a code of 6 characters. To find an employee record, an 8-digit TID is used which occupies 4 character positions, giving $V + P = 10$ bytes per index entry.

The blocks in the file system are $B = 1000$ characters long, so that $y = 100$ index elements can appear per block.

$$\begin{aligned} n' &= n\, a'/a = 20\,000 \times 2.5 = 50\,000 \\ b_1 &= n'/(0.69y) = 50\,000/69 = 725 \\ b_2 &= b_1/(0.69y) = 725/69 = 11 \\ b_3 &= b_x = b_2/(0.69y) = 11/69 = 1 \end{aligned}$$

Estimating x directly, we obtain the same result:

$$x = \lceil \log_{69}(20\,000 \times 2.5) \rceil = 3$$

An estimate for the space is based on Eq. 3-51, as modified for all levels:
$SI(\texttt{skill}) = 1.5n\, a'/a(V + P) = 750\,000$ bytes, and used to derive Eq. 3-53.
The space requirement for this index is actually
$SI_{skill} = \sum b_i B = 737\,000$ bytes, because of the high fanout.

---

‡‡‡‡‡ **Attribute Sizes in Indexes** The value field in an index, V_{index}, is often not the same size as in the data, so that it should be estimated separately. The value field in an index may, for instance, be required to be of fixed length to allow fast searches through the index, so that V_{index} will be larger than the average value field, V, in a data record. On the other hand, if there are frequently multiple records related to a certain attribute value. For example, if the attribute category is `profession`, there will be several entries in this index for an attribute value like `"welder"`); and one value entry may serve many TIDs.

Techniques discussed in Chap. 4-3-3 reduce key value sizes of indexes further, so that often $V_{index} < V$. ‡‡‡‡‡

Fetch Record in an Indexed File The expected fetch time for an indexed file is similar to the time used for an indexed-sequential file. However, no overflow areas exist, and hence the term that accounts for the chasing after overflow records is eliminated. The indexes will be of the larger record-anchored variety and also will contain space for insertions, so that their height, x, will be greater. We add the accesses for index and data and find

$$T_F = x(s + r + btt) + s + r + btt = (x + 1)(s + r + btt) \qquad 3\text{-}55$$

where x is given by Eq. 5-26.

If each index can be kept compactly on a single cylinder, then some seeks can be avoided, so that

$$T_F = 2s + (x + 1)(r + btt) \qquad \langle \text{compact index} \rangle \; 3\text{-}56$$

Get-Next Record of an Indexed File The search for a successor record is based on the assumption that the last index block is kept available in a buffer, so that only a new data record has to be fetched.

$$T_N = s + r + btt \qquad 3\text{-}57$$

This holds as long as the effective fanout ratio $y_{eff} \gg 1$, since the probability that the current index block contains also the TID of the next record is $Pd = (y_{eff} - 1)/y_{eff}$.

If indeed a successor index block is needed, the second-level index block has to be accessed to locate the successor. If the second-level block is not kept in a memory buffer as well, two block accesses are needed with a frequency of $1 - Pd$. The frequency of having to access a third- or higher-level index block is generally negligible.

Insert into an Indexed File To add a record to an indexed file, the record is placed in any free area, and then all a' indexes referring to existing attributes for this record have to be updated. The insertion process will have to find for each index the affected index block by searching from the root block of each index, and rewrite an updated index block at level 1; we will call this time T_{index}. With a probability of $Ps = 1/(y/2) = 2/y$ there will be a block split. A block split requires

the fetch of a new partner block, some computation to distribute the entries, and the rewriting of the new block and the ancestor block with a new entry, all this in addition to the rewrite of the old block; the incremental time is T_{split}.

Summing the times for data block fetch and rewrite, for following a' indexes down x levels, for first-level index rewrites, and for the possible split, we find

$$\begin{aligned} T_I &= T_{data} + a'(T_{index} + Ps\,T_{split}) \qquad \text{3-58} \\ &= s + r + btt + T_{RW} + a'\left(x(s + r + btt) + T_{RW} + \tfrac{2}{y}(c + s + r + btt + 2T_{RW})\right) \\ &= \tfrac{2}{y}\,c + \left(1 + a'\left(x + \tfrac{2}{y}\right)\right)(s + r + btt) + \left(1 + a'\left(1 + 2\tfrac{2}{y}\right)\right) T_{RW} \end{aligned}$$

Keeping indexes within a cylinder can reduce the time by requiring only one seek per index. Although Eq. 3-58 looks forbidding, it actually shows that cost of insertion is mainly dependent on the number of indexes, or $\mathcal{O}(a'(1 + \log_y n + 1/y))$. If all indexes are small so that $x = 2$, further ignoring the cost of splitting ($y \gg 1$) and using $T_{RW} = 2r$ gives an estimate which is $\mathcal{O}(a')$:

$$T_I = (1 + a')s + (3 + 4a')r + (1 + 2a')btt \qquad \langle \text{simple} \rangle \text{ 3-59}$$

In the absolutely worst case the insertion could cause all index blocks to overflow so that they all have to be split. The delay of even several splits could be quite disconcerting to a user entering data on a terminal. With deferred updating only one index access and the actual record rewriting is noted by the user. A technique, *presplitting*, reduces the likelihood of multiple splits in one index.

‡‡‡‡‡ **Presplitting in B-trees** A problem with B-trees is that they are difficult to share. When a block at the lowest level (1) is split, the resulting insertions at the next higher level may propagate all the way to level x. Another user, wanting to access another portion of the same file, will use the same high-level blocks. If those change while they are being used, the program may not behave correctly.

To prevent errors a B-tree may be locked when in use for update. Then the changes due to the update cannot interfere with other access operations. Now the entire file becomes unavailable to others. In applications where many users may be active, for instance, in a reservation system, such a constraint is unacceptable.

An alternate and improved insertion strategy is *presplitting*. When the search for an index entry is made from the root to the lowest level, each index block read is checked; any index block which is already full ($y_{eff} = y$) is immediately split. This assures that the block at the next level up will never be found full, so that no split has to propagate from the lowest to the higher levels.

This technique becomes more difficult to implement when attribute values, and hence index entries, are of variable length. The definition of *not full* above has to leave enough space for a new entry of any expected size. The pragmatics of deletion operations remain the same.

When multiple indexes are used, then presplitting has to be performed by searching from the rootblock, but this is the best way anyhow. ‡‡‡‡‡

Update Record in an Indexed File An update of a record in an indexed file consists of a search for the record, the update of the data record, followed by a change of those indexes for which the new record contains changed values. An update changes one data record and a_{update} indexes. The new field values may be far removed from the old ones, so that the new index entries are in blocks other than the old ones. Now each index update requires the search for and rewriting of two index blocks, the old one and the new one, doubling the term $T_{index} + Ps\,T_{split}$ found in Eq. 5-35. The new index block may require a split because of the insertion, and the block used previously may have to be combined because of the deletion of an entry. We assumed that deletion is as costly as insertion and use below T_{split} for either operation.

The pile organization of the data file leads us to expect that the old data record will be invalidated and a new copy will be inserted ($T_{newcopy}$). The TID value for all indexes will then have to be updated so that the remaining $a' - a_{update}$ indexes also have to be fixed. Fixing the TID requires finding the blocks and rewriting them; no splits will occur here; this term is T_{fixTID}.

We collect all the costs.

$$\begin{aligned} T_U &= T_F + T_{RW} + T_{newcopy} \\ &\quad + 2\,a_{update}(T_{index} + Ps\,T_{split}) + (a' - a_{update})(T_{fixTID}) \\ &= (x+1)(s + r + btt) + T_{RW} + s + r + btt + T_{RW} \\ &\quad + 2\,a_{update}\left(x(s + r + btt) + T_{RW} + \tfrac{2}{y}(c + s + r + btt + T_{RW})\right) \\ &\quad + (a' - a_{update})\,(x(s + r + btt) + T_{RW}) \\ &= (x+2)(s + r + btt) + 2\,T_{RW} \\ &\quad + a_{update}\left(x(s + r + btt) + T_{RW} + \tfrac{4}{y}(c + s + r + btt + T_{RW})\right) \\ &\quad + a'\,(x(s + r + btt) + T_{RW}) \end{aligned} \qquad 3\text{-}60$$

Making the same simplifications used to obtain Eq. 3-59, we find that a simple update for a modest file will take approximately

$$T_U = (4 + 2\,a_{update} + 2\,a')(s + 2r + btt) \qquad \langle \text{simple} \rangle \; 3\text{-}61$$

If the updated record does not have to be moved, the $a' - a_{update}$ indexes do not have to be changed, and now the same simplification gives

$$T_U = (4 + 4\,a_{update})(s + 2r + btt) \qquad \langle \text{in place} \rangle \; 3\text{-}62$$

We observe that updating is, in most cases, costlier than insertion and that techniques for deferring part of the update process may be quite beneficial to gain response time for the user.

‡‡‡‡‡ **Small Value Changes** When an update changes an attribute value by a small amount, then it is likely that both old and new index values fall into the same index block. In that case one search and rewrite may be saved. The ratio between out-of-index block and within-index block changes depends on the application. In some cases the behavior of the attribute value changes is predictable. For instance, if the attribute type is a person's `weight`, we can expect the within-index block case to be predominant. If the update changes a value from undefined to a defined value, only a new index entry has to be created. If we define $P\!\!\!/_i$ to be the probability of out-of-index block updates, we find for a single update, given all the conditions leading to Eq. 3-63,

$$T_U = 2(3 + P\!\!\!/_i)(s + 2r + btt) \qquad \langle \text{single change, in place} \rangle \quad 3\text{-}63$$

In the general case we have to assume the index changes will be randomly distributed among the blocks of an index. Given the expected loading density of 69%, there will be

$$b_1 = 1.44 \frac{n}{y} \frac{a'}{a} \qquad 3\text{-}64$$

first level index blocks for one attribute; then the probability, given random data changes, of requiring another index block is high: $P\!\!\!/_i = (b_1 - 1)/b_1$. For large files $P\!\!\!/_i$ will be close to 1, so that Eqs. 3-60 to 3-62 remain appropriate.

However, Eq. 3-62 is of interest whenever it can be shown that $P\!\!\!/_i$ will be low because of the behavior of data value changes for updated attributes. Important cases exist in *real-time systems*, where the database receives values from devices that monitor changes in the operation of some ongoing physical process. Changes in `pressure`, `temperature`, or `flow` occur mainly gradually, making $P\!\!\!/_i$ small. In such systems the data is often acquired at high rates, so that the improved file performance could be critical. ‡‡‡‡‡

‡‡‡‡‡ **Non-Uniform Update Distribution** The uniform behavior assumed for insertion and update patterns represents the best behavior that can be expected. Unfortunately the growth of an index is rarely evenly distributed over the range of attribute values. Updating tends to be periodically heavy in one area, and then concentrate again on other attributes. Consider for instance a stock-market file, over some hours there is much activity in a stock which is in the news, and later there will be little activity there. These irregularities affect system behavior. An analysis of these conditions requires statistics about application update behavior which we do not have available here. We will not evaluate their effect here, but note that if there exists a good understanding of the frequency of attribute changes and of the value-change ranges, a more precise estimate of update cost can be made, by refining the concepts presented above. ‡‡‡‡‡

Read Entire Indexed File The basic fully indexed organization is poorly suited for exhaustive searches. When necessary, such searches may be accomplished by using the space-allocation information, or by serial reading of the file using some exhaustive index. An exhaustive index is created when the referred data element is

required to exist in every record. A brute force approach using such an index will cost

$$T_X = T_F + (n-1)\,T_N \qquad \langle \text{serially} \rangle \text{ 3-65}$$

If one can follow the space-allocation pointers, seeks need to be done only once per block, so that here the read time for a consecutively allocated file would be similar to the time needed to read a sequential file. The records appear to the user in a random order.

$$T_X = nR/t \qquad \langle \text{random seq.} \rangle \text{ 3-66a}$$

If the blocks are distributed randomly or found from the space allocation in a random fashion, then, neglecting the time needed to read the space-allocation directory,

$$T_X = \frac{n}{Bfr}(s + r + \frac{B}{t'}) \qquad \langle \text{random} \rangle \text{ 3-66b}$$

still improving on Eq. 3-65. In both of these approaches the records appear in a logically unpredictable order. A reduction in actual search efficiency may be due to the processing of empty spaces created by previous deletions.

Reorganization of an Indexed File Indexed files are not as dependent on periodic reorganization as are the previous file organizations. Some implementations of indexed files in fact never need to reorganize the files. Reorganization of the data file may recover unusable fragments of space left over from deletions. A specific index may require reorganization to recover from updating failures or from having poorly distributed index entries due to clustered insertions or deletions. Index reorganization can be done incrementally, one index at a time.

In order to reconstruct one index, the data file is read. A reorganization of an index separately would propagate any existing errors in the index. It is best to use the space-allocation directory, not only for speed but also to assure that every record is read. Since the records will not appear from the data file in the proper logical sequence for the index attribute, it is best to collect them in a dense temporary file having n' small $(V + P)$ records, then sort this file, and then generate the index blocks sequentially. Index generation can be combined with the final sort-merge pass, as will be pesented in Sect. 3-7.

$$T_Y(\text{one}) = T_X + T_{sort}(n') \qquad \langle \text{one index} \rangle \text{ 3-67}$$

Estimates for these terms were given as Eqs. 3-66, 3-12 or 3-107, and 3-50. The sort will be rapid if the space required for sorting $n'(V + P) = 0.69\,SI_1$ is small relative to memory capacity.

If the data file is reorganized, all the index TIDs become invalid. An effective way to reorganize data and the indexes is to read the old data file and write the file anew and reconstruct all the indexes. The time requirement then is

$$T_Y = 2T_X + a\,T_Y(\text{one}) \qquad \langle \text{data and indexes} \rangle \text{ 3-68}$$

We cannot expect to have sufficient memory to process all indexes, so that here a total of a sort files will be generated during the processing of the data file. We do assume that sufficient buffer space exists for all a indexes. Since reading and writing of the main file will frequently be interrupted, Eq. 3-66b is appropriate for T_X in this case.

3-5 DIRECT FILES

Direct file access uses a computation, ofden referred to as *h*ashing. to determine the file address for a specific record. The direct file exploits the capability, provided by disk units and similar devices, to access directly any block of a known address. To achieve such direct addressing, the key of the record is used to locate the record in the file. Hashed access is diagrammed in Fig. 3-17.

There are two components to a direct file:

1 The file space, organized into m slots. Each slot has the capacity to hold one of the n records of the file.
2 The computation τ which provides a slot address for a record given its key.

We present the basic organization of direct files in Sec. 3-5-1. The issues of computation τ are dealt with to some depth with in four subsections: 3-5-1.1: How to compute a suitable address for the record, 3-5-1.2: methods that accommodate growth in terms of m of a direct file, and 3-5-1.3: a survey of alternate algorithms for the key-to-address computation. Subsection 3-5-1.4 provides a method for evaluating their effectiveness and a brief summary. Practical computations cannot prevent conflicts, i.e., produce the same address for two distinct keys. Subsection 3-5-1.5 covers the methods that deal with the cases when the computation leads to a conflict, or generates a *collision*.

Sec. 3-5-2 shows applications which can use a direct file organization profitably. Section 3-5-3 evaluates the performance parameters; for this section the material introduced in Subsections 3-5-1.1 and -1.4 is adequate.

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

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

3-5-1 An Overview of Direct files

Direct files are based on direct access to a file, using a relative address as described in Chap. 3-3-3. In immediate implementations of direct access, identification numbers that provide a relative address into the file are assigned to the data records.

Thus, employee `Joe` is designated as `257`, and that tells us directly that his payroll record is to be found as record 257. We will list a number of problems

with the use of direct access to locate data records, and then present the common methods used currently.

Problems with Use of Direct File Addresses

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

into the file.

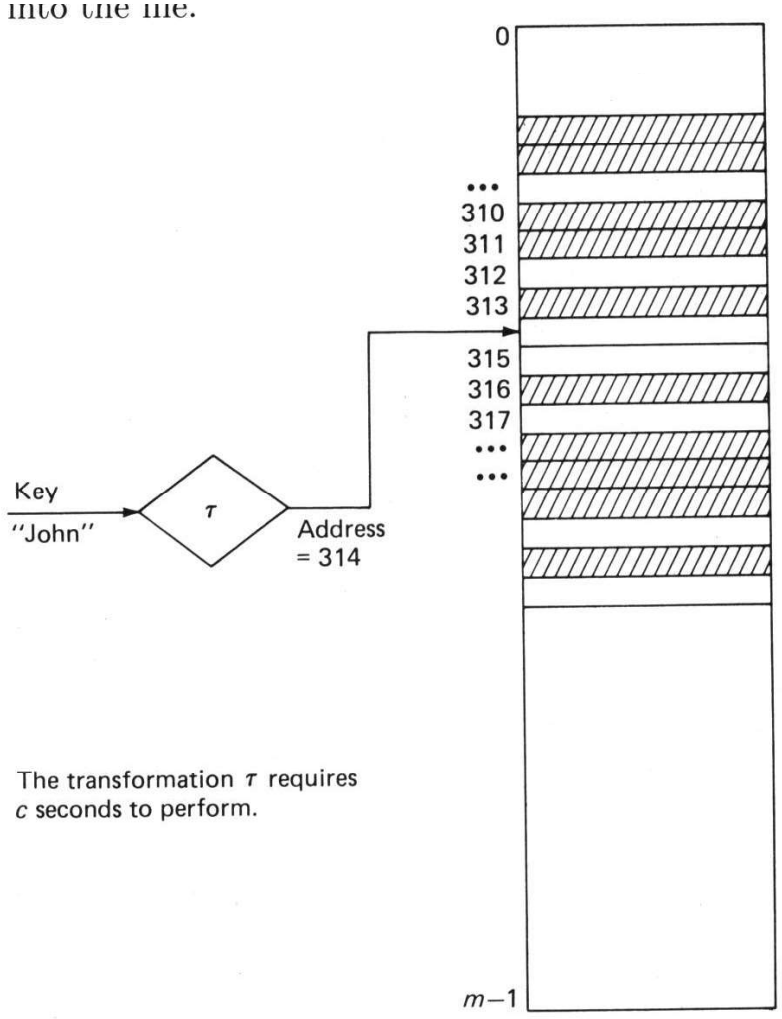


Figure 3-17 Record accessing in a direct file.

Definitions for Direct files We define now the concept of a *key-to-address transformation* or KAT, also denoted by the function τ. The objective of the computation τ is to give each arriving record its own slot, based on the key provided. This is achieved if we can distribute the n records uniformly over the m slots. The argument for τ is the key field from the record. When a record is to be stored, the key is given with the record and τ computes the slot address for the insertion function of the hashed access method. The common algorithms for τ perform this task by a *randomizing hashing function* on the argument.

When a record is to be retrieved a search key must be provided. It will be transformed by the same function τ into a slot address, and the retrieval function of the method will fetch the record from the slot.

The file space is partitioned into m *slots*: each slot can hold one record. The result of τ is a *relative slot address*, ranging from 0 to $m-1$. We need at least one slot per record expected, so that $m \geq n$. In practice the *density* of a direct file $n/m \approx 0.6$ to 0.95. The higher densities are only feasible for relatively small records, as shown later in Fig. 3-23.

A subsequent procedure, as outlined in Chap. 2-3-3, translates the relative slot address to an actual storage address. Then the block containing this slot can be brought in from storage. An actual access to the file obtains a block or a *bucket* of slots at a time. The size of a bucket, y, is typically determined by the block size so that $y = B/R$. We can expect to find $y_{effective} = \frac{n}{m}y$ records in a bucket.

Key-to-Address Transformations The problems identified with the immediate use of keys such as file addresses are solved by interposing a computational procedure. Such a *key-to-address transformation* (KAT) translates the key attribute values into relative addresses within the file space.

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

Figure 3-17 shows the access to a direct file for a new record with key "`John`". The KAT algorithm τ applied to the string "`John`" has generated the relative record address "`314`". The same KAT is applied to a search key when a record is to be fetched.

Most files will have their own key type and slot range. Even with the same key the procedure may differ. The KAT τ to compute the relative addresses from social-security-number key of a person in the 1000 slot `Employees` file differs from the KAT for the same key in the 150 slot `Sales_personnel` file.

Requirements for KATs An ideal key-to-address translation should satisfy two requirements, we mark the second one for further discussion:

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

$2^{?}$ The slot addresses generated should be unique.

These two objectives conflict. Although the source keys are presumably unique, it is impossible to compute smaller numbers which are unique for arbitrary source keys. Hence, we settle for a lesser criterion

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

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

There is a wide variety of choices for key-to-address transformations. In this section we will use the *remainder-of-division* algorithm throughout.

> We obtain relative addresses using a randomizing KAT
> by computing the remainder-of-division of the given key.

In Subsec. 3-5-1.3 this method is defined in more detail, and other methods, often more suitable, are surveyed and evaluated as well. All the required principles for hashing can be developed using a single type of KAT. Example 3-9 shows a small file where records are placed into slots using as KAT the remainder-of-division by 500.

Example 3-9 Key-to-address transformation, with a collision.

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

Record for	SSN	$\rightarrow \tau \rightarrow$	Slot address
`Al`	`322-45-6178`		`178`
`Joe`	`123-45-6284`		`284`
`Mary`	`036-23-0373`		`373`
`Pete`	`901-23-4784`		`284`

Even though it was unlikely, given 4 out of 500 employees, by chance two identical slot addresses were generated: `Joe` and `Pete` both received slot number 284; the records for `Joe` and `Pete` will collide if both are placed directly into a file.

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

Consider using the KAT from Example 3-9. The algorithm computed for `Pete`, with `social security number = 901-23-4784`, generated a slot number of `284`, but this did not provide a free space for `Pete`'s record, since `Joe` was assigned to that slot earlier.

Insertion Procedure As demonstrated in Example 3-9, before storing a new record into a slot of a direct file the content of the slot itself has to be tested. The slot can be

1 empty
2 it can contain an older record for the same key
3 it can contain a colliding record for another key
4 if deletions don't rearrange records it can contain a tombstone

The insertion procedure has to consider these three or four cases. To determine on insertion whether a collision is occurring we check if the slot at the computed address is empty. Insertion into an empty slot is straightforward. If the slot is full, there is a collision. Now we compare the key value of the record found in the slot with the key value of the new record. If key fields match, then earlier a record with the same key was inserted into the file. Perhaps the record found should be replaced by the new one; the implied operation is an `update` of all attributes but the key.

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

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

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

Deletion of records from a direct file using open addressing must be handled with care, since just setting slots "empty" can confuse the retrieval process. This problem is further addressed in Subsec. 3-5-1.5.

Termination Both insert and retrieval procedures are sure to terminate as long as the number of slots m is greater than the number of records n. If m is sufficiently greater than n the probability of finding an empty slot is good, and it will only be rarely neccessary to go to slots in a successor block. These concepts will be expanded on in the performance evaluation of Sec. 6-2.

We summarize both the insert and retrieval algorithm as a decision rule in Table 3-5 below.

The Decision Rule The corresponding outcomes are now summarized in Table 3-5; we assume that all source keys are unique. The precise handling of collisions is elaborated in Subsec. 3-5-1.5.

Table 3-5 Decision rules for collisions.

Condition	Insert	Fetch
Slot_key = Λ	Ok to insert	No record found
Keys match	Replace record if permitted	Record found
Keys don't match	Locate free successor slot	Check successor slot

Nonunique Record Keys Typically, records of direct files are defined to be *unique*. If in this file application multiple records have the same key, the records to be inserted can still be stored according to the collision resolution scheme used.

If nonunique records, i.e., records with duplicate keys, are to be retrieved, a search up to an empty is needed to find all matches to the search argument. For deletion, either all records with that key must be deleted, or other fields of the record must be inspected. We will not consider duplicate source keys further in this chapter.

Initialization When a direct file is created it must be initialized to "empty". This means that for all slots the field to be used for the key is set to some tombstone. In our examples we use Λ to indicate an empty slot.

3-5-1.1: Randomizing Key-to-Address Transformations The KATs we consider using use a randomizing transformation when translating the key value into relative file addresses.

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

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

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

Using Alphabetic Identifiers Since many of the key-to-address algorithms depend on a numeric key value, it may be desirable to convert alphabetic keys into integers. If a value from `0` to `25` (`lettervalue`) is assigned to each of the 26 letters used, a dense numeric representation can be obtained using a polynomial conversion, as shown in Table 3-6. Numbers and blanks occurring in keys require an appropriate expansion of the `lettervalue` table which controls the results of this routine. For large keys numeric overflow has to be considered.

Table 3-6 Character string key to numeric key conversion.

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

Computing the Relative Address Given a numeric key, we apply τ to yield a relative address. For the remainder-of-division algorithm we must choose a divisor, m. Since the desired density, $\frac{n}{m}$, is an approximate goal, we have some flexibility in choosing m.

A value of $\frac{n}{m}$ of 1 would nearly guarantee a collision for the final, n^{th} record to be inserted, since only one slot will then be unoccupied. A very low density of, say, $\frac{n}{m} = 0.10$, would yield only a 10% probability of a collision for the last record, but the file space utilization would be very poor. We will work below with values of $\frac{n}{m} \approx 0.6$.

The actual value of the divisor, m, is also a concern. Values including factors of 2, 10, 26, etc., may lead to results which are not fully random, since some bias from the key argument may be retained. Good candidates are prime numbers, or at least numbers of the form $2^k - 1$. The latter are often prime as well, and waste only one slot if the storage allocation is in powers of 2, giving 2^k slots. We show in Table 3-7 the reasoning to be followed and the resulting program fragment.

Table 3-7 Key-to-address transformation using remainder-of-division.

We need to select a divisor for a file of up to $n = 500$ employees. For a desired density $\frac{n}{m} = 0.64$ we find $m_{c1} = n/0.64 = 780$. This number turns out to have factors 10 and 26, so we'd better change it. Given a *Bfr* of 7 we would be using $b = \lceil 780/7 \rceil = 112$ blocks. Since 112 blocks can hold $7 \times 112 = 784$ slots, we have another candidate $m_{c2} = 784$, still with an undesirable factor 2. The next lower divisor $m = 783$ turns out to be prime. Our function τ now becomes:

```
/* Slot number for the key; uses remainder-of-division by a prime */
    rel_address = MOD(numeric_value, 783);
```

The file system uses relative addressing, as detailed in Chap. 2-3-3, to translate sequential slot numbers, computed as `rel_address`, from their range of 0 to $m - 1$ into the physical addresses used by the storage devices. If these devices are disk units we can seek directly to the right track and access the block containing the data.

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

3-5-1.2 Accommodating Growth of Direct files The familiar notion that a file grows by appending records does not apply to the direct file. The space to accomodate n records is initially allocated and performance diminishes as n approaches the number of slots m allocated. An increased density, $dens = n/m$, leads to higher rates of collisions, as quantified by Eqs. 3-69, 3-73, etc. To keep performance acceptable some reorganization of the file is needed. A complete reorganization means determining a new value for m, allocating the new area, finding a new KAT, and rehashing all records into the new area.

Such a reorganization, performed by an exhaustive read and a reload of the entire file, is often unacceptable:

1. It causes a long service interruption.
2. Programmer intervention is required to select the new allocation and key-to-address transformation.

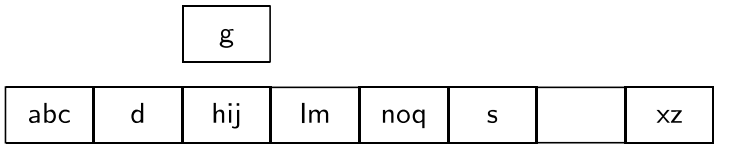
Several methods have been proposed to avoid the loss of performance or the need for a complete reorganization.

Extensible hashing techniques add index tables into the file access paths where bucket overflow occurs. The tables will direct retrievals to overflow blocks when warranted, avoiding chains and open addressing. If those tables are kept in memory, all accesses can be kept within a

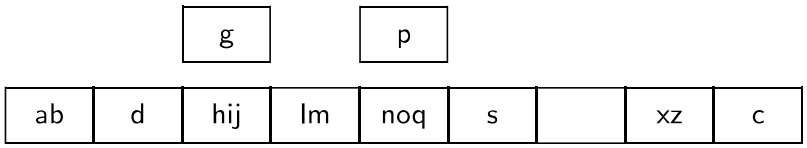
single block access.

Table 3-8 Actions during linear hashing.

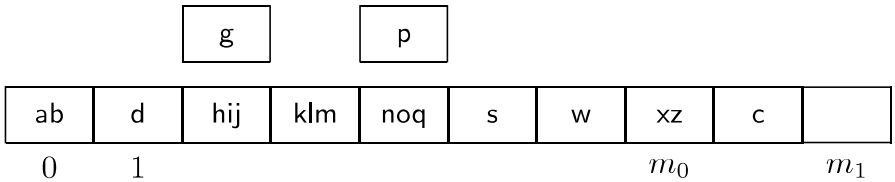
Initial allocation was $m = 8$ blocks with $y = 3$, the desired density is $\frac{n}{m} = 0.66$.
We insert records with addresses a, d, i, m, j, n, c, x, h, l, z, o, g, b, q, s.



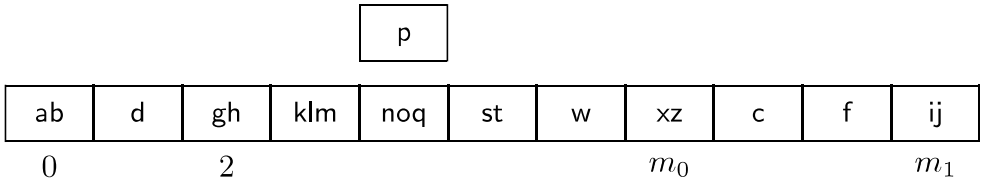
After 16 insertions the desired density is reached. One bucket (2) overflowed already.



New insert (of p) causes the first split, on bucket $m_1 - m_0 = 0$, to be performed.
Then m_1 is incremented.



After two more inserts (k, w) the next block (1) is split, the new block gets no entries.



After the next two inserts (f, t) another split is made; it removes an overflow block.

Linear Hashing More interesting, and consistent with the general concept of hashed access, is a technique termed *linear hashing*, which is described in the this section and illustrated in Table 3-8. The principle of linear hashing is incremental reorganization. In linear hashing (Litwin[80]) the file size grows linearly, block by block, keeping the density approximately constant. A constant density leads then to a constant fraction of overflows.

The file space grows by appending blocks linearly to the end of the hash space. For collisions which still occur overflow blocks are used. We will use m_0 to indicate the initial basic allocation, and let m_0 be a power of 2. The current allocation, m_1, begins with m_0 slots. The procedure operates unchanged until the file reaches a size of $2m_0$ slots. At that point the key-to-address transformation is adjusted as well. No wholesale reorganization of space is required at any time.

The actual space allocation in a file is again by buckets which each have y slots. The file actually grows from $m_1 \Rightarrow m_1 + y$ slots every $\frac{n}{m}y$ insertions. Table 3-8 illustrates three splits.

Whenever a new block is assigned it is populated by splitting an existing block $m_1 - m_0$. Note that the block to be split is assigned differently from a split seen in B-trees: The block to be split is **not** the block where the collision occurred, but in linear hashing the block to be split is the *lowest numbered* block in the range 0 to m_0 not yet split. This scheme avoids auxiliary tables to indicate which blocks have been split; only the current limit m_1 has to be known.
The method for addressing and the method for splitting cooperate. The KAT produces a `rel_address` for $m = 2m_0$ slots, so that an extra high-order bit is available. The generated address is reduced to the current range $0, \ldots, m_1$ by an additional statement as shown in Table 3-9. This statement corrects the addresses of all blocks not yet split: $(m_1 - m_0)$, ..., m_0.

Table 3-9 Address adjustment for linear hashing.

```
/* Diminish address if file is still growing */
IF rel_address > m1 THEN rel_address = rel_address - m0
```

When the file grows a new block is appended and the current `m1` is incremented. Some of the content of the block at $m_1 - m_0$ is now distributed to the new block. The `rel_address`es of the records in that block and in any overflow blocks attached to that block are recomputed using the KAT and the adjustment with the incremented m_1. All those records will either have the original address (the high-order bit was zero and their `rel_address = m1-m0`) or will have the new address `m1`, referring to the new block. The records which now belong in the new block are moved and both the split block and the new block are rewritten.

When the file has grown to twice its size ($m_1 = m$) the KAT is revised to produce by another bit: m and m_0 are doubled, and the process can continue with $m_1 = m_0$. No reorganization is required if the new KAT generates the same low-order bits. Methods to actually allocate these blocks are presented in Chap. 6-5.

3-5-1.3 A Survey of Key-to-Address Transformation Methods

Of the many KAT techniques which have been proposed and evaluated for hashing, only some are used for files. We consider here also some transformations which do not randomize or hash. Two categories of these computations may be distinguished, *deterministic procedures* which translate identification fields into unique addresses, and *randomizing techniques* which translate the keys into addresses which are as unique as possible but do not guarantee uniqueness.

A *deterministic procedure* takes the set of all key values and computes a unique corresponding relative address. Algorithms for such transformations become difficult to construct if the number of file entries is larger than a few dozen. Adding a new entry requires a new algorithm, since the algorithm is dependent on the distribution of the source keys; hence only static files can be handled. Methods for finding such transformation algorithms exist; however they are very costly as n increases, and impractical for insertion into a file access method. The replacement of a computational algorithm with a table makes the problem of transformation more tractable: We have invented again the indexed file! We will not discuss deterministic hashed access further.

The alternative, randomizing, procedures can cause collisions. A small class of transformations is *sequence-maintaining*; the other algorithms are referred to as *hashing techniques*. The first class tries to preserve the order of the records while computing addresses, but this leads to complexity and inflexibility. The goal of the common hashing transformations is to maximize the uniqueness of the resulting addresses. A family tree of key-to-address transformations is displayed in Fig. 3-18.

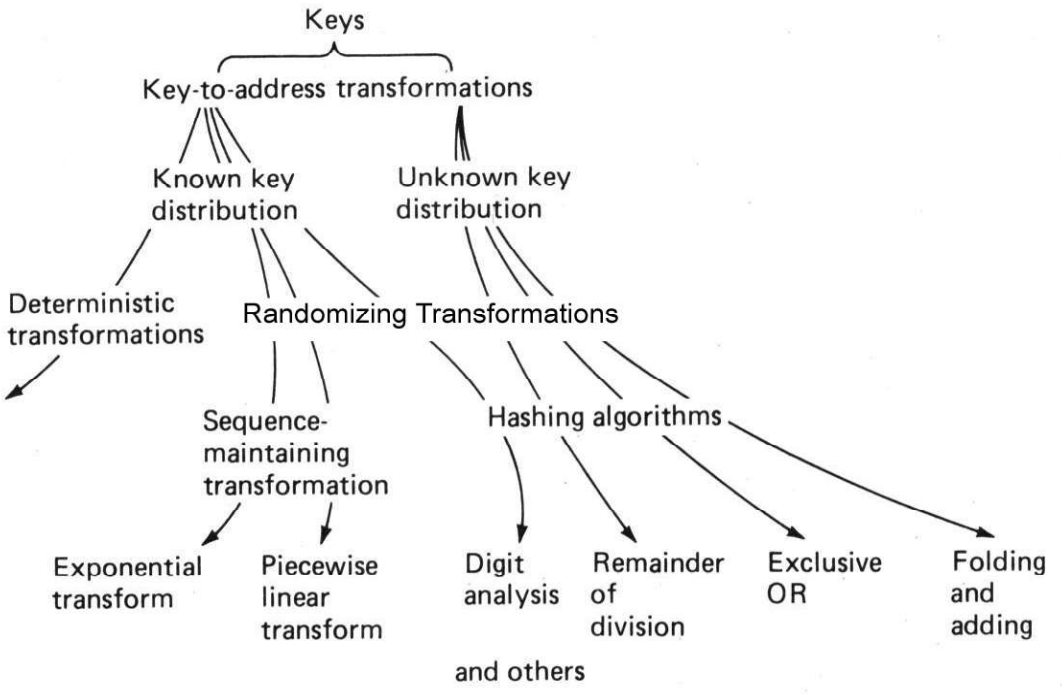


Figure 3-18 Key-to-address transformation types.

Distribution-Dependent Methods These methods depend on some knowledge of the distribution of keys for the expected records. If we have, for instance, assigned random numbers to identify 1000 samples of some experiment being recorded, we can expect that any three digit positions of the sample identification will be randomly distributed, and suitable as direct-access keys.

The liabilities of distribution-dependent transformations are major, since a change in key distribution can cause these methods to generate many more collisions than previously. A benefit of some distribution-dependent key-to-address transformations is that they allow maintenance of sequentiality. Two distribution-dependent methods are *digit analysis* and *sequence-maintaining transformations*.

‡‡‡‡‡ **Digit Analysis** attempts to capitalize on the existing distributions of key digits. A tabulation is made for each of the individual digit positions of the keys using a sample of the records to be stored. The tabulation lists the frequency of distribution of zeros, ones, twos, and so on. The digit positions that show a reasonably uniform, even distribution are candidates for use in the slot address. A sufficient number of such digit positions has to be found to make up the full address; otherwise, combinations of other digit positions can be tested. In Example 3-9 the three low-order digits of a social security number were considered to be uniformly distributed. Similar use can be made of character-string keys. Here the set of 26 possible letters may be divided into 10 groups to yield digits, or into groups of different sizes to yield factors other than 10 which can be used to produce an access record number. ‡‡‡‡‡

‡‡‡‡‡ **Sequence-maintaining Methods** for key-to-address transformation generate addresses that increase with increasing keys. Serial access is made possible, whereas generally direct files do not have capability for serial access.

A sequence-maintaining transformation can be obtained by taking an inverse of the distribution of keys found.

This inverse function is applied to the keys. The addresses generated will maintain sequentiality with respect to the source key.

In a *piecewise-linear-transformation* the observed distribution is approximated by simple line segments.

This approximation then is used to distribute the addresses in a complementary manner. An *exponential transformation* is presented in Fig. 12-6 as an illustration of the evaluation of the uniformity of obtained addresses. These approaches are valid only if the source key distribution is stable. It may be difficult to find a simple inverse function. ‡‡‡‡‡

Hashing Methods The three hashing methods presented here (*remainder-of-division*, *exclusive*-OR, *folding-and-adding*) randomize the source key to obtain a uniform address distribution. Operations such as arithmetic multiplication and addition, which tend to produce normally distributed random values (see Chap. 12-1), are undesirable when hashing.

Remainder-of-Division of the key by a divisor equal to the number of record slots allocated m was used throughout this chapter. The remainder does not preserve sequentiality. The remainder-of-division is in some sense similar to taking the low-order digits, but when the divisor is *not* a multiple of the base (10 in Example 3-9) of the number system used to represent the key, information from the high-order portions of the key will be included. This additional information scrambles the result further and increases the uniformity of the generated address.

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

The need to use a division operation often causes programming problems. The key field to be transformed may be larger than the largest dividend the divide operation can accept, and some computers do not have divide instructions which provide a remainder. The remainder then has to be computed using the expression, as shown in Table 3-10.

Table 3-10 Using division to produce the remainder.

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

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

Exclusive-OR (`X-OR`) is an operation which provides a very satisfactory randomization and can be used when key lengths are great or division is otherwise awkward. The bits of two arguments are matched agains each other. It yields a 1-bit where the argument bits differ and a 0-bit otherwise. The `x-or` operation is available on most computers or can be obtained by a procedure as defined in Table 3-11.

Table 3-11 Key-to-address transformation using `X-OR`.

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

As Table 3-11 shows, the key is segmented into parts (here 2) which match the required address size. The `x-or` operation produces equally random patterns for random binary inputs. If the segments contain characters, their boundaries should

be avoided. For instance, `x_or('MA','RA')` will be equal to `x_or('RA','MA')`, so that "`MARA`" will collide with "`RAMA`". Care has to be taken also where the binary representation of decimal digits or characters is such that certain bit positions always will be zero or one. Both problems can be controlled by making the segment sizes such that they have no common divisor relative to character or word sizes. The `x-or` operation is generally the fastest computational alternatives for hashing.

‡‡‡‡‡ **Folding and Adding** of the key digit string to give a hash address has been used where the `x-or` has not been available. Alternate segments are bit-reversed to destroy patterns which arithmetic addition might preserve. A carry from a numeric overflow may be added into the low-order position. This method is available in the hardware of some large Honeywell computers. ‡‡‡‡‡

3-5-1.5 Probability of Successful Randomization

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

Since the randomization method is chosen without any prior knowledge of the keys, the selection of any reasonable method out of the m^n choices gives a probability of that method causing no collisions ($o_0 = 0$) of only $p_0 = (m!/(m-n)!)/m^n = 0.2$.

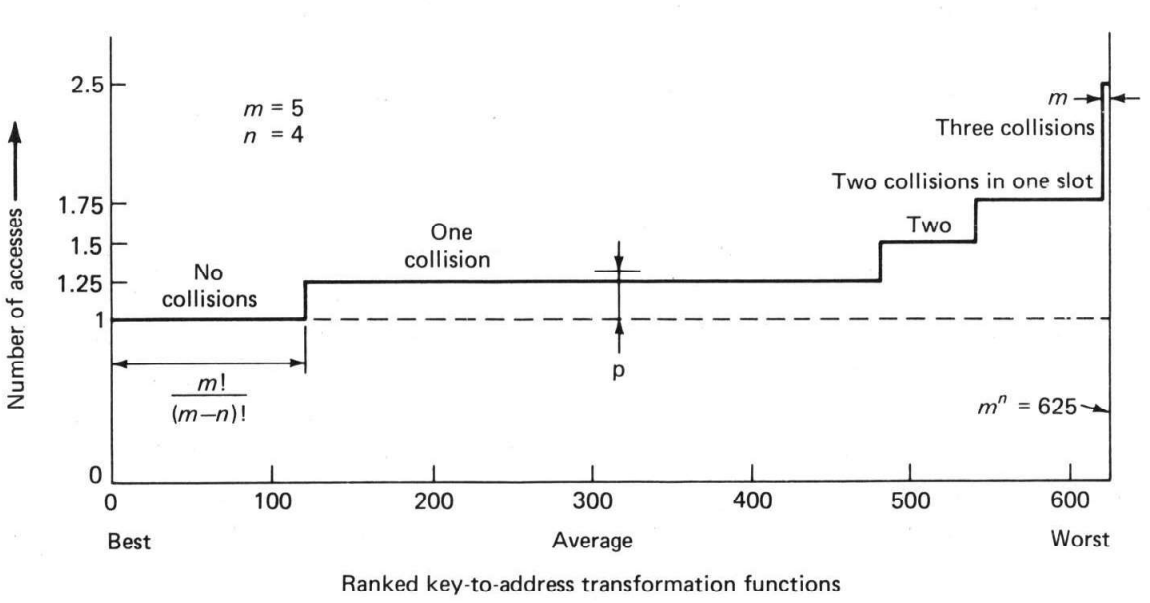


Figure 3-19 Fetch-length distribution.

On the other hand, the probability of selecting one of the five worst methods for this case, which would lead to $c = n-1 = 3$ collisions and $o_{max} = 1+2+3$ extra accesses, is only $p_{max} = m/m^n = 0.008$. The remaining three cases of one collision, two collisions in two slots, and two collisions in the same slot can be analyzed similarly and are shown in Fig. 3-19. Given the assumption that we have just an average randomization, we find that we may expect an average of $p = \sum_c p_c o_c = 0.30$ additional accesses per record loaded.

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

The distribution will have a similar shape for other values of m and n. The number of steps rapidly increases with n and m so that the distribution function becomes smooth. Estimates for the number of collisions in the general case are derived in Chap. 12-1, but the intent of this discussion is to show the relative unimportance of the specific choice of randomization method, as long as it is chosen to be outside the obviously worst areas. It might be mentioned here that a change of keys, that is, a new set of data to be randomized, will completely reorder the position of the m^n functions in the figure, but the shape of the exhaustive distribution remains the same.

Summary of Key-to-Address Transformations It is obvious from the above that many choices exist for transforming keys of records to record addresses. We find also that the average performance of reasonable transformations is such that perfectly good results are obtained with less than perfect methods. On the other hand, cute methods, such as deterministic and sequence-maintaining schemes, carry a high risk of failure when conditions change.

3-5-1.5 Collision Management

When we obtain the slot at the address computed above we may find it occupied. The probability of finding every slot in a bucket occupied is much lower. We will first consider slots by themselves, or buckets having only one slot.

When a key-to-address transformation produces identical addresses for distinct records, we have a collision. There is no way that a randomizing procedure can avoid collisions completely, and deterministic, nonrandomizing KAT procedures (briefly discussed in Sec. 3-5-1.3 above) are not practical for large or changing files. We hence have to learn to deal with collisions. Figure 3-20 shows a key-to-address transformation and a collision; a new entry, `Shostakovich`, collides with an earlier entry, `John`.

The more slots a file has relative to the number of records to be stored, the lower the probability of collisions will be. While we will evaluate such probabilities in detail subsequently, we note that at typical B-tree density ($n/m = 0.69$), we incur a rate of collisions that is quite bearable; even with one slot per bucket, we expect collisions less than 35% of the times we access the file.

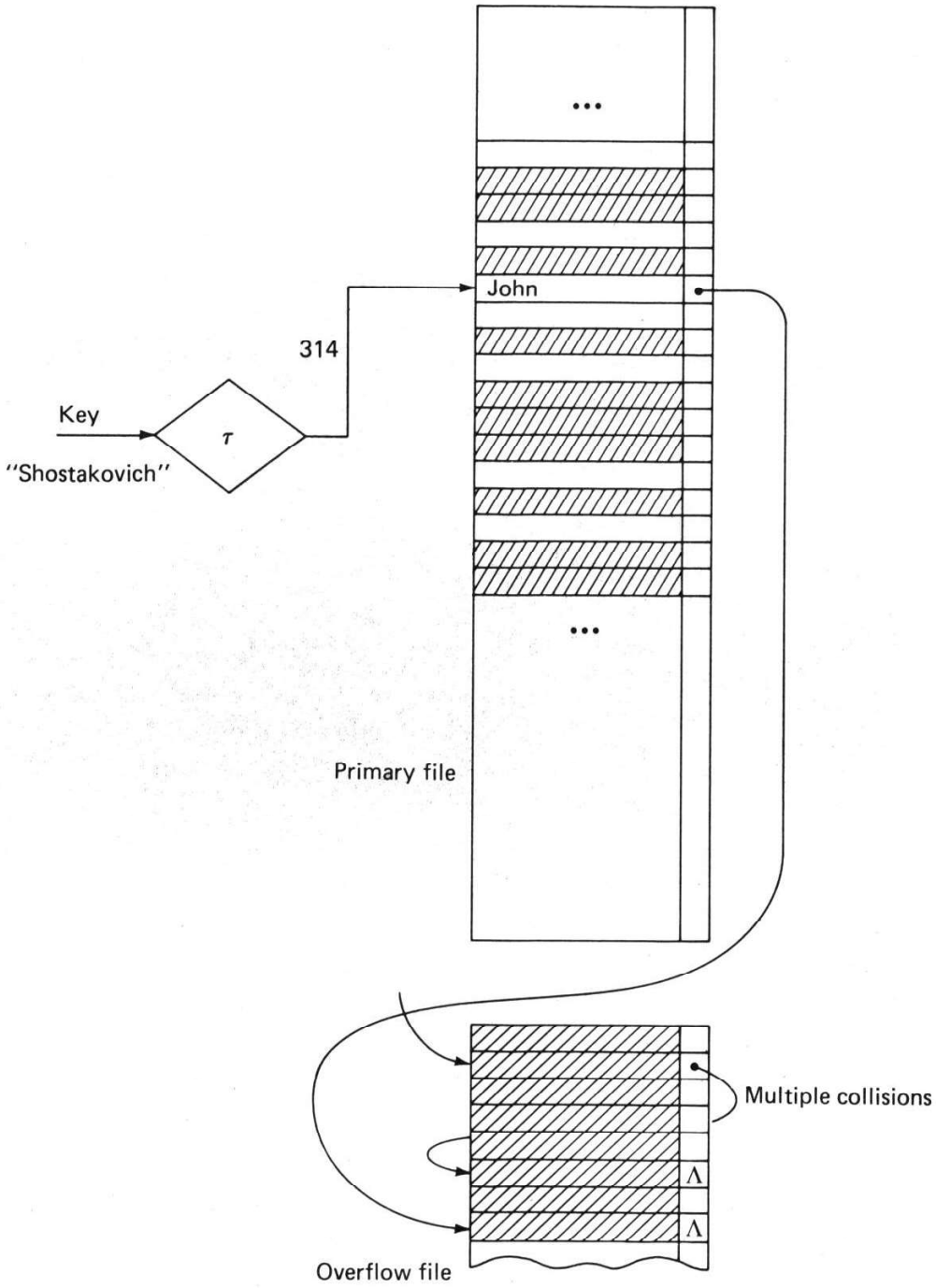


Figure 3-20 Direct file with collision.

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

Search in the Bucket the cost of overflows from collisions can be greatly reduced by grouping of the slots into buckets. A typical bucket is equal to one block, as shown in Fig. 3-21. A search through all the slots in the bucket is used to locate space for the new record. Only computational time is required to access the records within a bucket. Only when the bucket itself is filled will a disk access be needed. The cost of the disk access depends on which of the three alternative overflow methods, shown next, is in use. If the bucket contains y slots there will be space for *ovpb* collisions within the block, where

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

namely, the fraction of free space times the number of slots. We can use the blocking factor *Bfr* for y if we are careful to remember that we are talking here about slots, and not records. The cost of most collisions is now drastically reduced. An analysis of the effect of the use of buckets is unfortunately quite complex. Figure 3-23 gives results for the use of buckets with a linear search in the file. We find for the earlier example ($n/m = 0.69$) and $Bfr = 8$ a reduction in collision cost from 35% to 5%, as marked with an x in Fig. 3-23.

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

We now consider methods to resolve collisions.

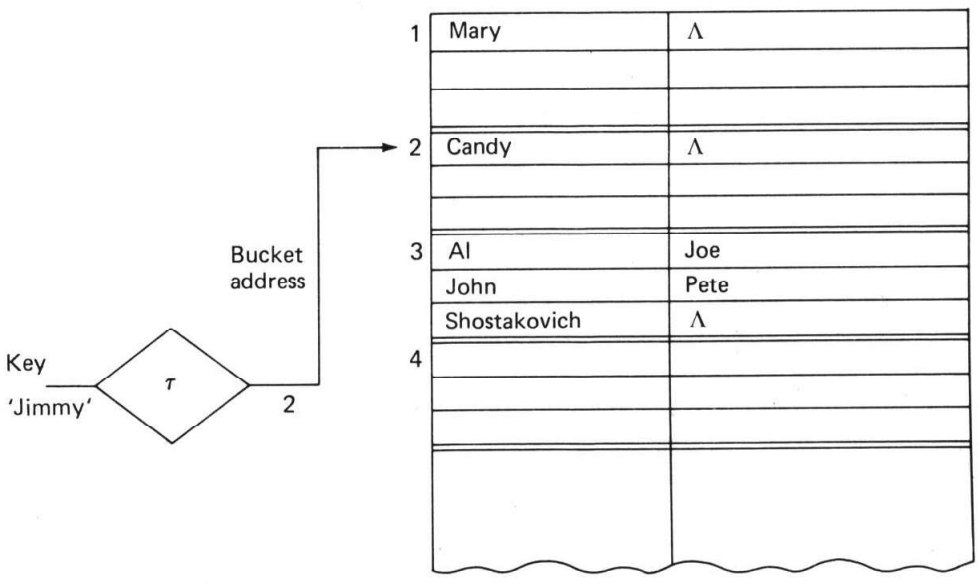


Figure 3-21 Buckets.

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

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

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

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

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

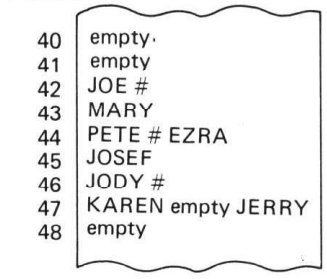
‡‡‡‡‡ **Linkage to Colliding Records** Once the block for an overflow is located, the record is yet to be found. Overflow records in successor blocks can be found either by simply matching all successive records in the block to the search argument, or by maintaining a chain to link the overflow records together.

Without a chain the search for a record or for a free slot for an insertion terminates when an empty slot is reached. This approach constrains the deletion of records, as discussed below. In a densely loaded file it may take many accesses to find an empty slot, because many intervening slots may be filled by records belonging to the set associated with a different primary bucket address. To identify

an arbitrary stored record as a member of the set of colliding records for a specific prime bucket, the key of the stored record accessed has to be transformed to the corresponding address, which is then matched to the initial bucket address. If the addresses are equal, this record belongs to this *collision set.*

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

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



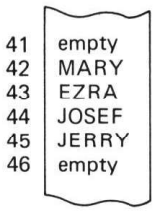
Sequence of additions (+) and deletions (–) to a cluster of a direct file:

Entry →	Address	Action
+ JOE	42	No collision, insert
+ PETE	44	No collision, insert
+ KAREN	47	No collision, insert
+ MARY	42	Collision, overflow to 43
+ JOSEF	43	Collision with overflow, second collision, gets 45
– KAREN	47	Simple delete, empty slot
– PETE	44	Cannot mark slot empty to avoid break in search process from 43 or 42 to 45 etc. Mark deleted (#)
+ EZRA	42	Can reuse slot 44
+ JODY	44	Gets slot 46
– JOE	42	Mark deleted to avoid losing overflows from this slot (MARY)
+ JERRY	45	Overflows to 47
– JODY	44	Mark 46 deleted for JERRY's sake

A search in this cluster, as it appears now, will require an average of

$$\frac{2(\text{MARY}) + 3(\text{EZRA}) + 3(\text{JOSEF}) + 3(\text{JERRY})}{4(\text{number of entries})} = 2.75 \text{ comparisions.}$$

An optimal distribution, as sketched below, for these entries would require:



$$\frac{1+2+2+1}{4} = 1.5 \text{ comparisions on the average}$$

Rearrangement of members of chains when deleting can improve the search time experienced in the cluster above.

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

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

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

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

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

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

The performance can be recovered during reorganization. Breaking clusters can be performed incrementally, on a block by block basis. During such a reorganization it is possible to collect sufficient information about the collisions to either or both

1 reset deleted slots which are not within any collision set to `Empty` to break clusters no longer needed
2 rearrange the records in any clusters to optimize access

A complete reorganization would rehash all the keys in the block and place the records in order by computed address. ♯♯♯♯♯

3-5-2 Use of Direct files

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

Hashing is not appropriate for attributes which have a poor selectivity. The clusters in the source key are reserved when hashing, and may coalesce with other clusters. Managing deletion in clusters is costly as well.

Direct files also play an important role as a component of more complex file organizations. It provides rapid access by identifier, as `SSN` or `inventory_code`, which are already designed to be unique, while other techniques are used for secondary attributes. We will find many uses of hashed access in Chap. 7.

3-5-3 Performance of Direct files

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

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

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

Record Size in a Direct file The space required for an individual record is the same as for the indexed-sequential file: $R_{actual} = aV + P$, but the space for an entire file, S_F, as described previously, is

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

or, per record,

$$R_{effective} = \frac{m + o}{n}(a\,V + P) = \frac{m + o}{n} R_{actual} \tag{3-71}$$

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

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

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

An analysis which estimates the collision cost, p, for this design is given in Chap. 12-1-5, leading to Eq. 12-15. The result of the analysis shows that the expected value of p is

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

for the case of hashing to single-slot buckets and using separate overflow areas for collisions.

Effect of Open Addressing In open addressing with a linear search clusters of collisions can occur. The detrimental effect of clusters increases rapidly with the file density, n/m, as seen in Fig. 3-23 for the bucket size $y = 1$.

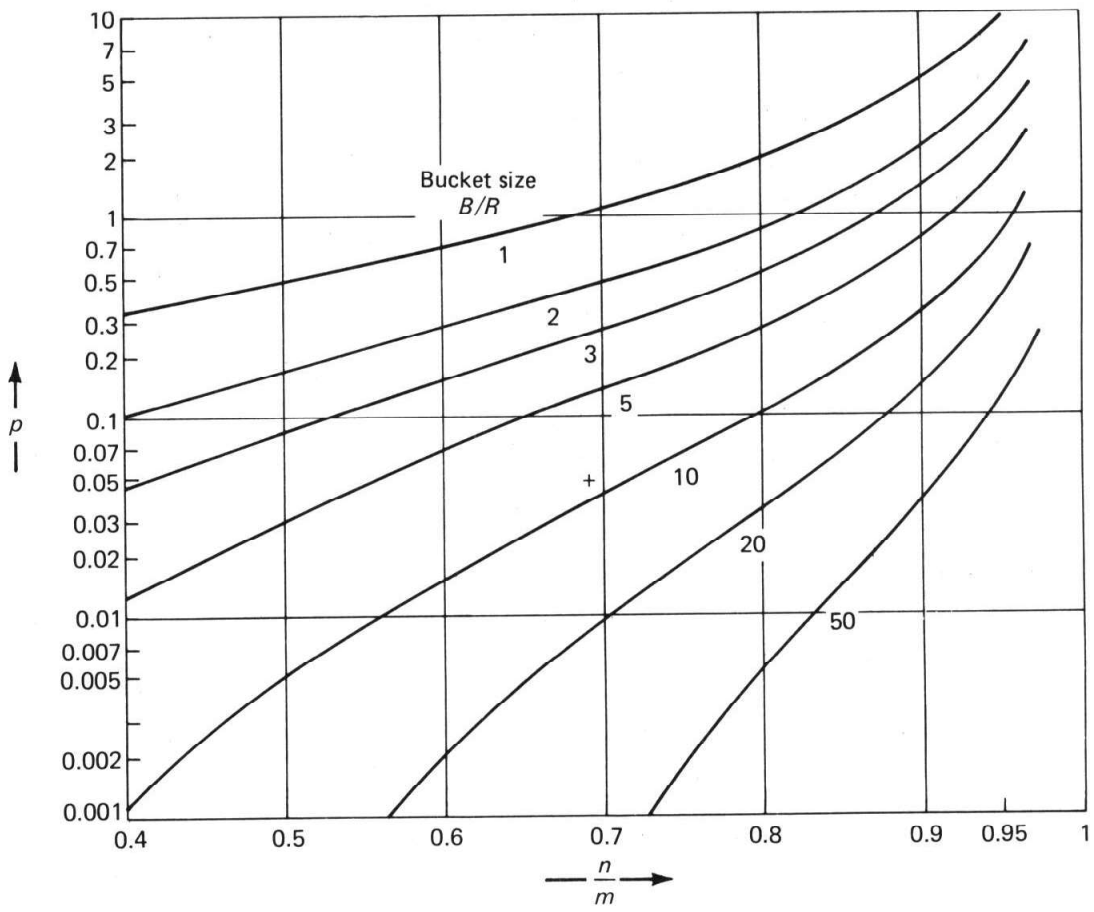


Figure 3-23 Overflow cost for multirecord buckets, open addressing, and linear search.

Knuth[73S] has derived, using the arguments of an "average" randomly selected randomization discussed in Subsec. 3-5-1.4, that the number of additional accesses becomes

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

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

It is important to note three aspects of open addressing multi-record bucket hashing:

1 the overflow probability and cost can be kept very small with good combinations of bucket capacity y and file density n/m
2 no overflow area is required when open addressing is used, so that for the same total amount of storage the value of m may be appropriately larger. Increasing m to $m + o$ will not compensate fully for the increased number of collisions, as demonstrated in the worked-out example in Example 3-10.
3 the deterioration of performance with increasing density is very rapid as shown in Fig. 3-23.

Example 3-10 Collision cost assessment.

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

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

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

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

$$p = \frac{1}{2}\frac{1}{1.5-1} = 1.0 \qquad \text{or 100\% extra accesses}$$

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

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

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

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

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

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

Locating the Successor Block In open addressing with linear searching, if the decision to read the next block can be made immediately, no revolution needs to be wasted. In general, however, one revolution is required to read the next block if the read decision is based on the contents of the predecessor block, so that the access time for an overflow is $T_{RW} + btt$ or $2r + btt$. Alternate-block accessing, as shown in Fig. 3-24, can reduce this delay to $2\,btt$, leaving a time $c < btt$ for the comparisons within a bucket. The decision of where to place overflows is specific to the procedure for hashed-file access.

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

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

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

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

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

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

In summary, the fetch time for the various cases becomes

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

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

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

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

Fetch for a Record Not in the File

In a fetch for a nonexistent record, we will be forced to follow the chain search until we find a Λ pointer. This time is greater than the time required for fetching an existing record, since we expect to find records on the average halfway into the chain. This condition occurs also when insertions are considered (T_{Iov}), since all the records of an overflow chain have to be tested to locate an empty slot for insertion.

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

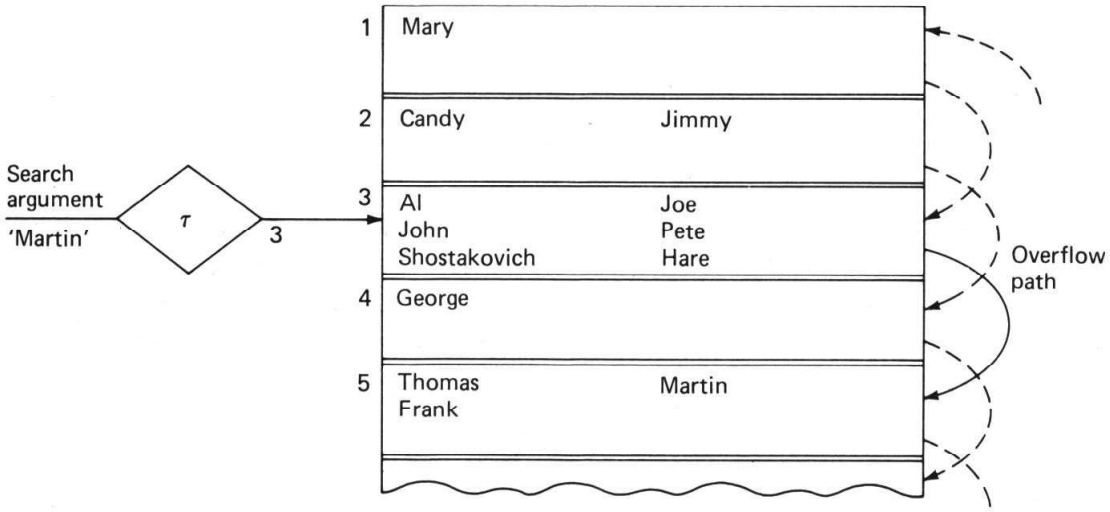


Figure 3-24 Alternate bucket overflow accessing.

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

$$T_N = T_F \qquad \langle \text{known next key} \rangle \text{ 3-78}$$

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

This represents the major liability for hashing, so that for special cases KATs have been built which overcome that problem.

Using a Sequence-maintaining KAT

For direct files using a sequence-maintaining key-to-address transformation, a successor record can be found. The next record is located by sequential search, skipping over any unused slots. The probability of finding the record in the next slot can be estimated using the density of the main file n/m. A low density reduces performance here. Separate overflow areas must be used to maintain serial linkage of overflows, increasing the cost of fetch and next access to overflowed records. To evaluate the cost a case-analysis approach, as used for indexed-sequential files (Eq. 3-40), is necessary.

Insert into a Direct file To insert a record into the file, an empty slot has to be found which can be reached according to the key. The key-to-address transformation

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

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

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

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

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

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

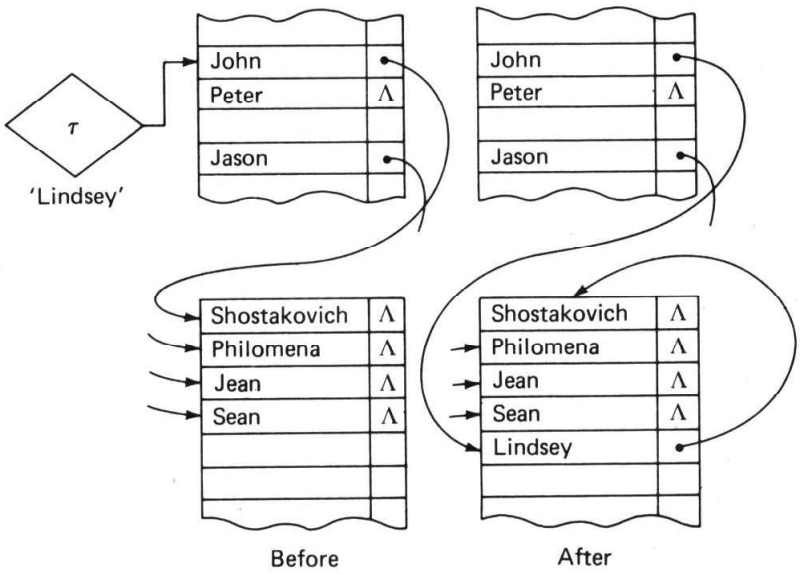


Figure 3-25 Overflow record linked as second chain member.

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

pointer value obtained from the primary record. The two rewrites require

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

But placing the insertion immediately does not assure that there are no duplicated records with the same key.

If duplicate records with the same key are not permitted, *all* records in the bucket and in any overflow have to be checked prior to insertion of the new record. If records are chained the entire chain must be followed. The considerations made when evaluating the fetch of a nonexisting record apply here as well, $p(\texttt{not\_in\_file})$, and the search time for checking involves the terms used in Eq. 3-77. When the end of the chain is reached (pointer is Λ), a block with the selected prior record and the new inserted record is rewritten. If the block where the search terminated is full, an additional block fetch and rewrite sequence is required. The probability of this event is only $1/y$. The primary block is not rewritten unless it was the terminal block of the chain.

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

The cost of following the chain is similar for overflow areas and for open addressing, the parameters used are given with Eq. 3-77.

Equation 3-82 applies also if we wish to optimize the record placement into blocks of the overflow area. Here the simple procedure of Fig. 3-25 is not adequate and the chain will have to be followed until a suitable block is encountered.

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

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

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

If buckets with many slots per bucket are used, the probability of overflow out of block (p as given in Fig. 3-77) reduces rapidly and any chains will be short. Now we can consider simply that insertion requires just one fetch and rewrite sequence. In open addressing the inserted record is likely to fit into the block found. This is expressed by the factor $\frac{m}{m-n}$ in the final term of Eq. 3-84; use of a separate overflow area would remove that factor, as seen in Eq. 3-82.

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

The use of large buckets and chaining does affect the value of c.

These evaluations assumed that the number of records n was as expected. When a direct file is not yet filled there will be fewer, n', records, leading to better performance. This factor is considered when a direct file is being reorganized.

Update Record in a Direct file The process of updating the record consists of finding the record and rewriting it into the original slot, so that

$$T_U = T_F + T_{RW} \tag{3-85}$$

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

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

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

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

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

Reorganization of a Direct file Reorganizations are mainly required if the total number of records to be kept has grown and no provisions were made for extension of file space; linear hashing was presented in Sec. 3-5-1.2 as a method to accommodate growth incrementally. Growth causes the density, n/m, to exceed the design goals. Before n approaches m reorganization is needed. Reorganization allocates a larger space to the primary area for the file; the randomizing procedure has to be rewritten or adjusted. The file is exhaustively read from the old space and reloaded into the new space. The performance of the terms T_X and T_{Load} defined in Eq. 3-87 should be estimated using the old value of m for T_X and the new value of m for T_{Load}.

A simple reloading procedure is to perform n insertions, each taking T_I. Two other alternatives are presented here as well.

Reorganization is also beneficial when many deletions have occurred. Deletions in clusters cause the search paths to be tangled and long. Large clusters in open addressing, the effect of the tombstones left to assure search paths can reduce performance greatly.

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

$$T_Y = T_X + T_{Load} \tag{3-87}$$

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

‡‡‡‡‡ **LOADING A DIRECT FILE** We present three methods for loading a direct file:

1 Successive insertion
2 A two-pass procedure designed to avoid clusters
3 A procedure where the input is sorted to speed up loading.

Successive Insertion The most obvious way to load or reload a direct file is to rewrite the records one by one into the new space; then

$$T_{Load} = n\,T_I(n', m) \tag{3-88}$$

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

Two-pass Insertion A two-pass procedure can be used to reduce the effect of clusters. In the first pass records are stored only if they have primary slots; any colliding records are placed into the remaining empty slots of the file during a second pass. The technique is mainly used for files that are frequently read and rarely updated. The second pass may try to optimize record placement.

Sorting and Block Writing A major reduction in loading cost can be accomplished by sorting the file to be loaded into the direct file sequence, i.e., by computed address. To do this, the key-to-address transformation is applied to all the record keys of the records to be loaded into the file, and the address obtained is attached to the records. The records are then sorted based on the attached address. Equations 3-2 and 3-107 provide the speed of a sort operation. The sorted file is then copied (without the addresses) into the space for the direct file; slots which do not have a matching address are skipped, and colliding records are placed into successor slots. This writing time is equal to the sequential exhaustive read time, so that

$$T_{Load} = c + T_{sort} + T_X \tag{3-89}$$

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

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

$$T_Y = c + 2\,T_X + T_{sort} \tag{3-90}$$

If the last sort pass can be combined with filling the direct file, then one term T_X drops out. ‡‡‡‡‡

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

3-6 THE MULTIRING FILE

The three previous file organization methods dealt with the problem of finding individual records fast. This last of the six fundamental methods, the *multiring* file, is oriented toward efficient processing of subsets of records. Such a subset is defined as some group of records which contain a common attribute value, for instance, all `employees` who speak `French`. The multiring approach is used with many database systems; we consider only the file structure of this approach.

Subsets of records are explicitly chained together through the use of pointers. The chain defines an order for the members of the subset. One record can be a member of several such subsets. Each subset has a header record which is the origin for the chain. A header record contains information which pertains to all its subordinate member records. The header records for sets of subsets can also be linked into a chain.

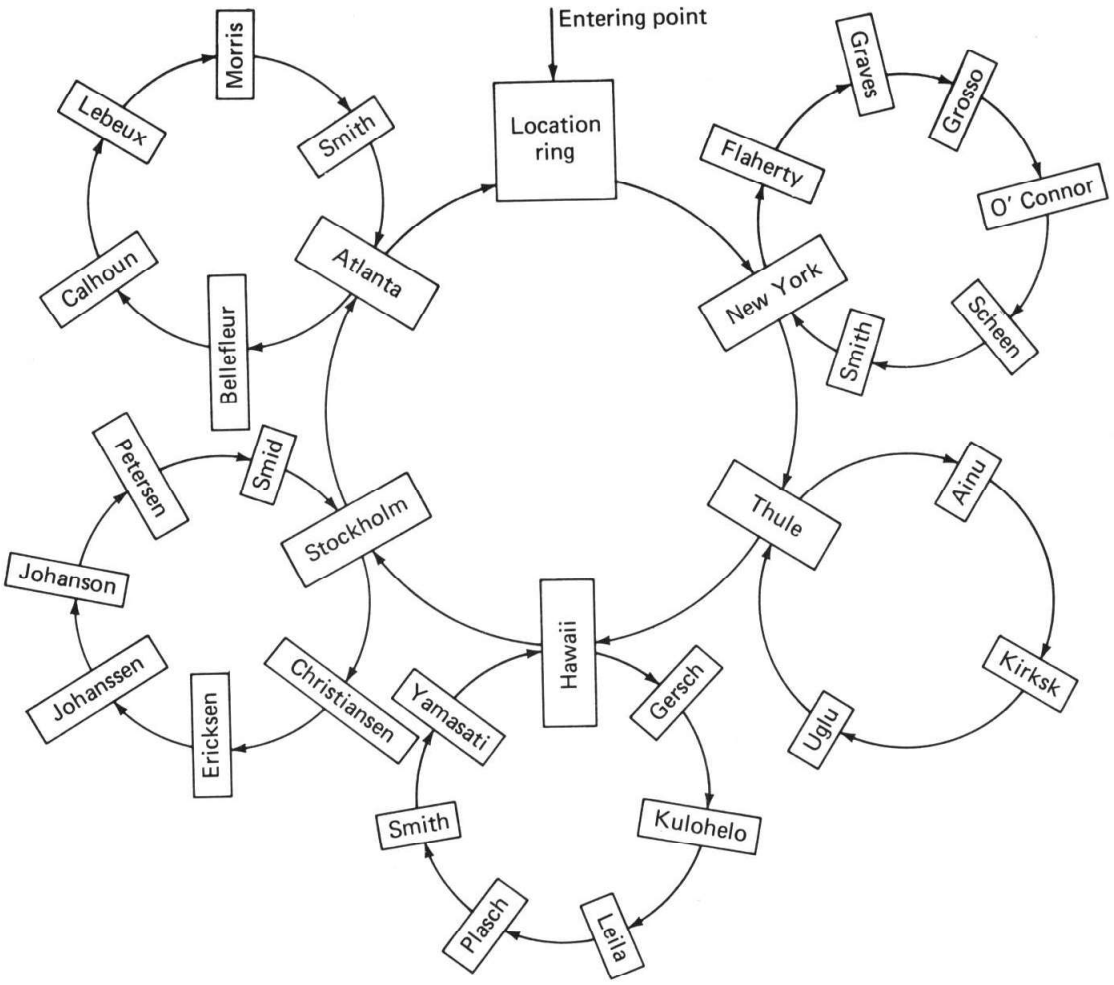


Figure 3-26 Record linkage in a simple multiring file.

The particular type of chain which will be used to illustrate this approach is the *ring*, a chain where the last member's pointer field is used to point to the header record of the chain. Similar file structures are called *threaded lists* or *multilists* in the literature, and can be implemented either with rings or with simple chains. Rings can be nested to many levels of depth. In that case member records of a level i ring become the header records of subsidiary rings at level $i-1$. The bottom level rings, with the final data, are considerd to be on level 1.

Figure 3-26 shows a simple hierarchically linked ring structure. The individual records in this example are formatted as shown in Fig. 3-27.

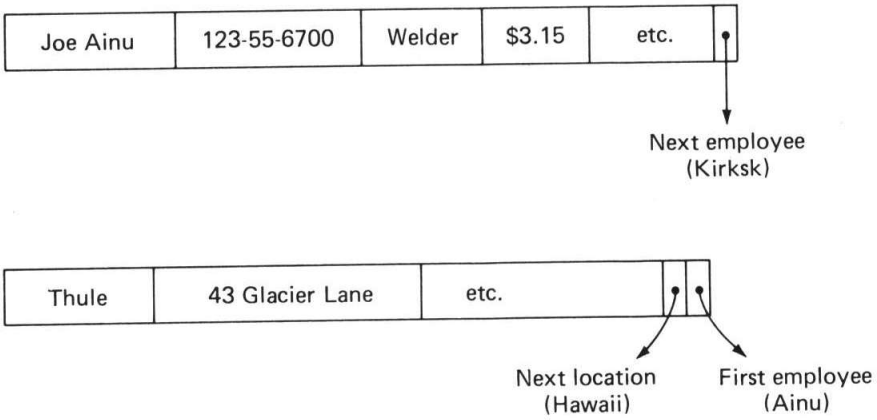


Figure 3-27 Records in a ring structure.

An example of the type of query for which this organization is well suited is

"`List all employees in Thule`"

A search in a multiring file follows a chain for an attribute type until a match is found for the search attribute value. Then a new chain is entered to find the subordinate attribute records. This process is repeated if neccessary until the desired record or set of records is found. For the example the location ring is read until the record for `Thule` is found; then the three `employees` living there could be listed.

Depiction of Multiring Files To simplify the presentation of these rings, we use boxes and a special arrow to indicate rings and their relationship. The concept of the arrow to depict $1 \rightarrow m$ relationship of the one master record to many subsidiary member records is due to Bachman[66]. The shape of the arrow we use ($\multimap$) is chosen to denote that many records will exist at the terminal point, although only one box is shown in the diagrams. A simple arrow is used to indicate an *entry point*, a ring whose header record is immediately accessible. All rings will be listed in the file directory, and their entry points will provide an initial access for processing queries. Figure 3-28*a* depicted the structure of the sample shown as Fig. 3-26 with an entry point using hashed access.

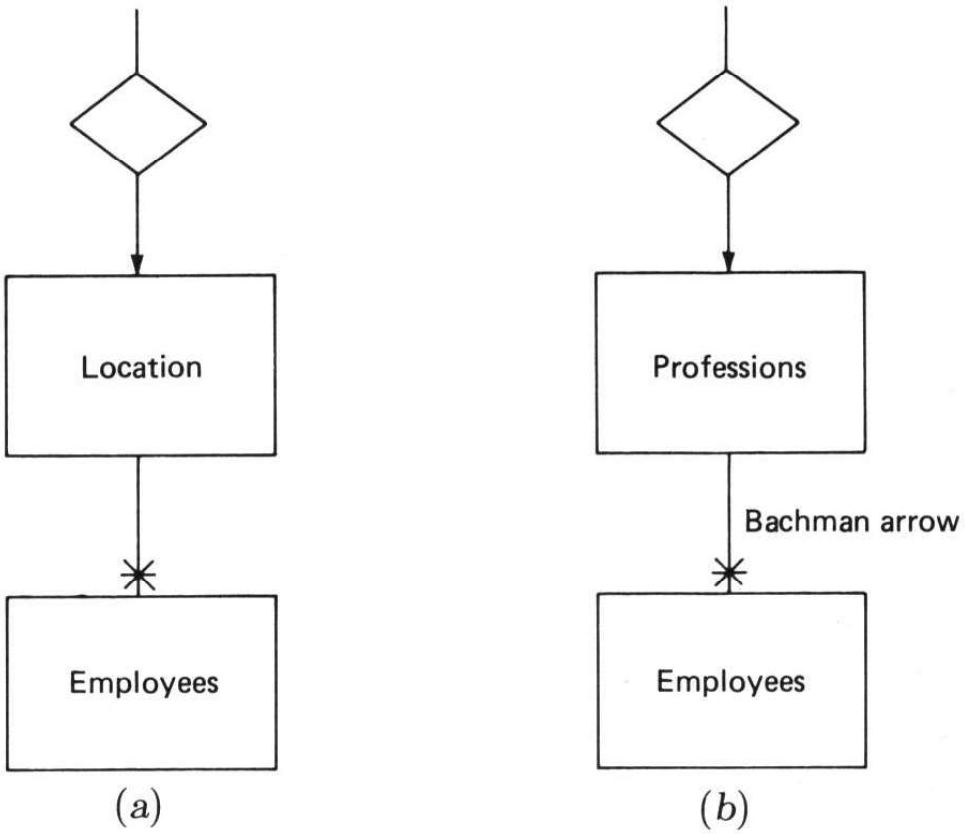


Figure 3-28*a* and *b* Two employee files.

Interlinked Rings Another query of possible interest, which should be answerable from this file, might be

"`List all welders in our company`"

Using the structure shown in Fig. 3-26, this would require an exhaustive search, traversing the `department` ring and each `employee` subsidiary ring in turn. Figure 3-28*b* shows a ring structure which allows answering this second query.

We now have two redundant `employee` files. They should be combined by linking both by `profession` and `location`, as shown in Fig. 3-29. The actual ring structure with these two relationships becomes quite complex. Real world applications include many more rings.

If we expand the example of Fig. 3-29 by segregating the `employees` in the various `locations` into specific `departments`, allow access also in order of `seniority`, add a `warehouse` at each `location`, and keep `stock` information available, then the structure diagram will be as shown in Fig. 3-30. If the actual connecting pointers had to be drawn, the picture would look like a bowl of spaghetti.

Relationships between rings are not necessarily hierarchical. Linkages may be implemented that relate members of the same ring (Fig. 3-31), that provide multiple pathways between records, or that relate lower rings back to higher-order rings. Two examples of multiple pathways occur in Fig. 3-32, which may be helpful to readers who understand football.

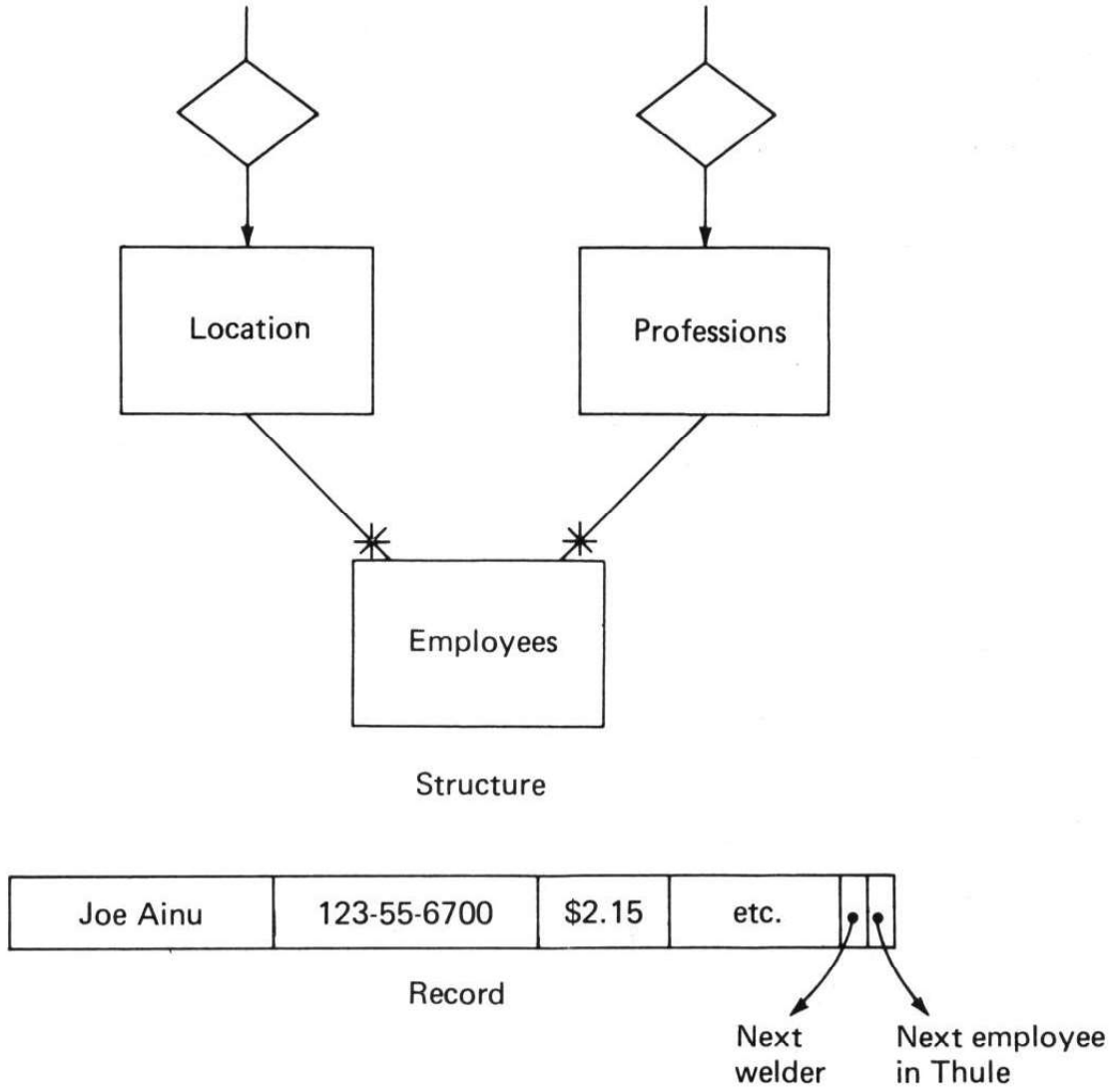


Figure 3-29 Interlinked rings.

Not all constructs are allowed or implementable in practice. The ease with which the membership arrows can be drawn hides the complexity of underlying structure very effectively. Loops and other nonhierarchical relationships between records may require a variable number of pointer fields in the records and are hence undesirable. In Fig. 3-32 an implementation may limit the `spouses` relationship to one entry. The `visit_history` relationship may be best implemented using a search argument from `children` to an index for `clinics`. Network database management systems support such alternatives.

As the structures become more complex, the query processor may have to choose among alternate access paths to retrieve a record or a subset, and the choices may have quite different costs. A query applied to the structure shown in Fig. 3-30 as

"`Find an employee in Thule who can weld`"

can be processed beginning at the `location` or at the `profession` ring. The appropriate records will be found at the intersection of any `department` at that `location` and the `welder` ring.

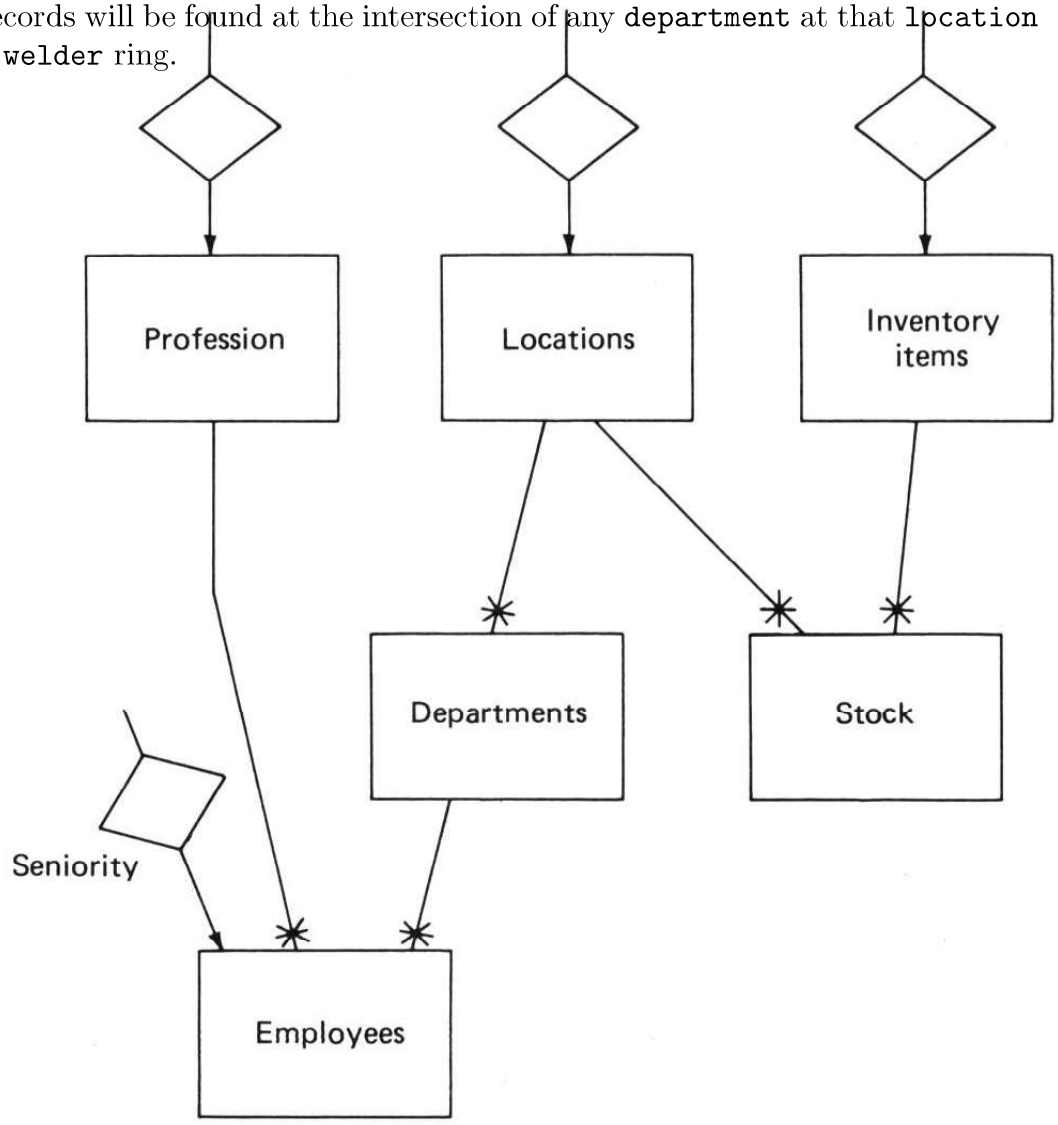


Figure 3-30 Complex hierarchical structure.

The effectiveness of a process in locating a record depends strongly on a good match of the attribute pairs forming the query argument with the structure of the file. If the file is not organized appropriately, the process cannot proceed efficiently, and user intervention may be needed. For instance, if there is no `profession` ring, the path for the query above would have to begin at the entry point for `location`. Since the `location` record does not give a clue about the `profession`, an exhaustive search would be necessary through all `department` rings at this `location`. In an interactive environment, the system might have asked at this point,

"`Which department do you suggest?`".

The process of finding the best path for a query to such a database has been termed *navigation* by the principal developer of this file-design concept (Bachman[73]).

Structure of Multiring Files In a multiring file all records will have similar structures, but their contents and size will be a function of the rings to which they belong. A multiring file may have a considerable number of distinct record categories. We note here a violation of the earlier definition of a file: The records now are not identical in format, and ring membership as well as file membership has to be known before processing.

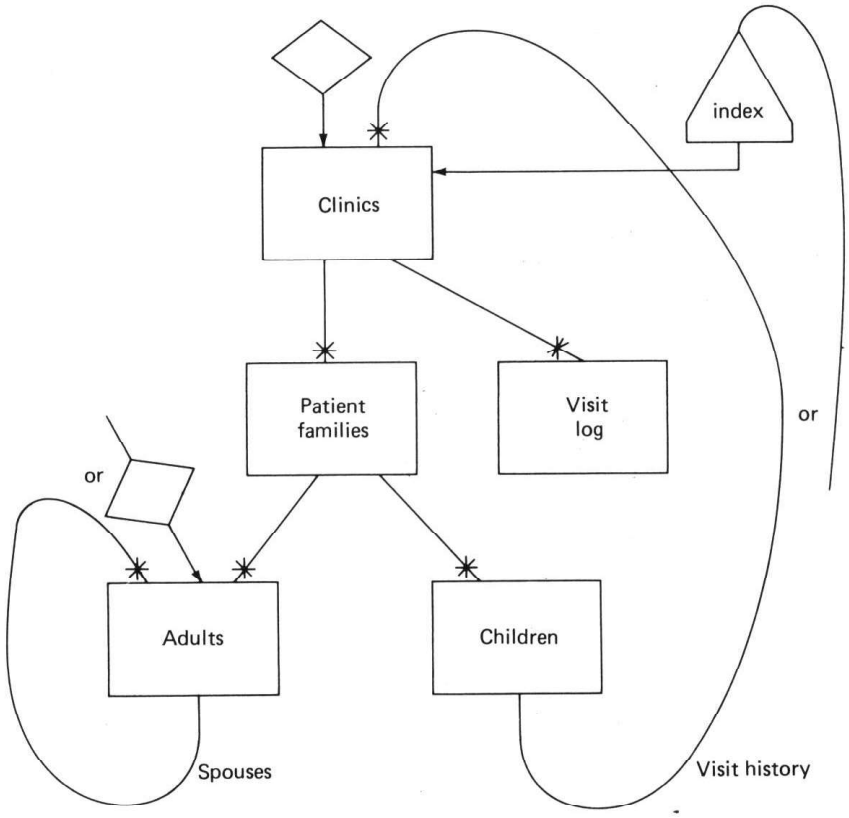


Figure 3-31 Loops in ring structures.

Record Formats The precise format of a record depends on the combination of ring types of which the record is a member. The attribute-value pairs could be self-identifying, as seen in the pile file; but typically they are not, and each record will have a record-type identifier. In Fig. 3-33 the field, `t`, identifies this record as being an `employee` record. Each record of type `t` will have similar data fields and seven pointer fields. This identifier will allow reference to an appropriate record format description, stored with the general description of the file.

To link the records into their rings, many pointers will appear in a typical record. A record can belong to as many rings as it has pointers. A given pointer position in the record format is assigned to a specific category of rings. In the example above, `profession` may be such a category and `welder` one specific ring of that category. If an instance of a record is not a member of a certain category ring, there will be an unused pointer field containing a `NULL` entry (Λ), as shown in Fig. 3-33.

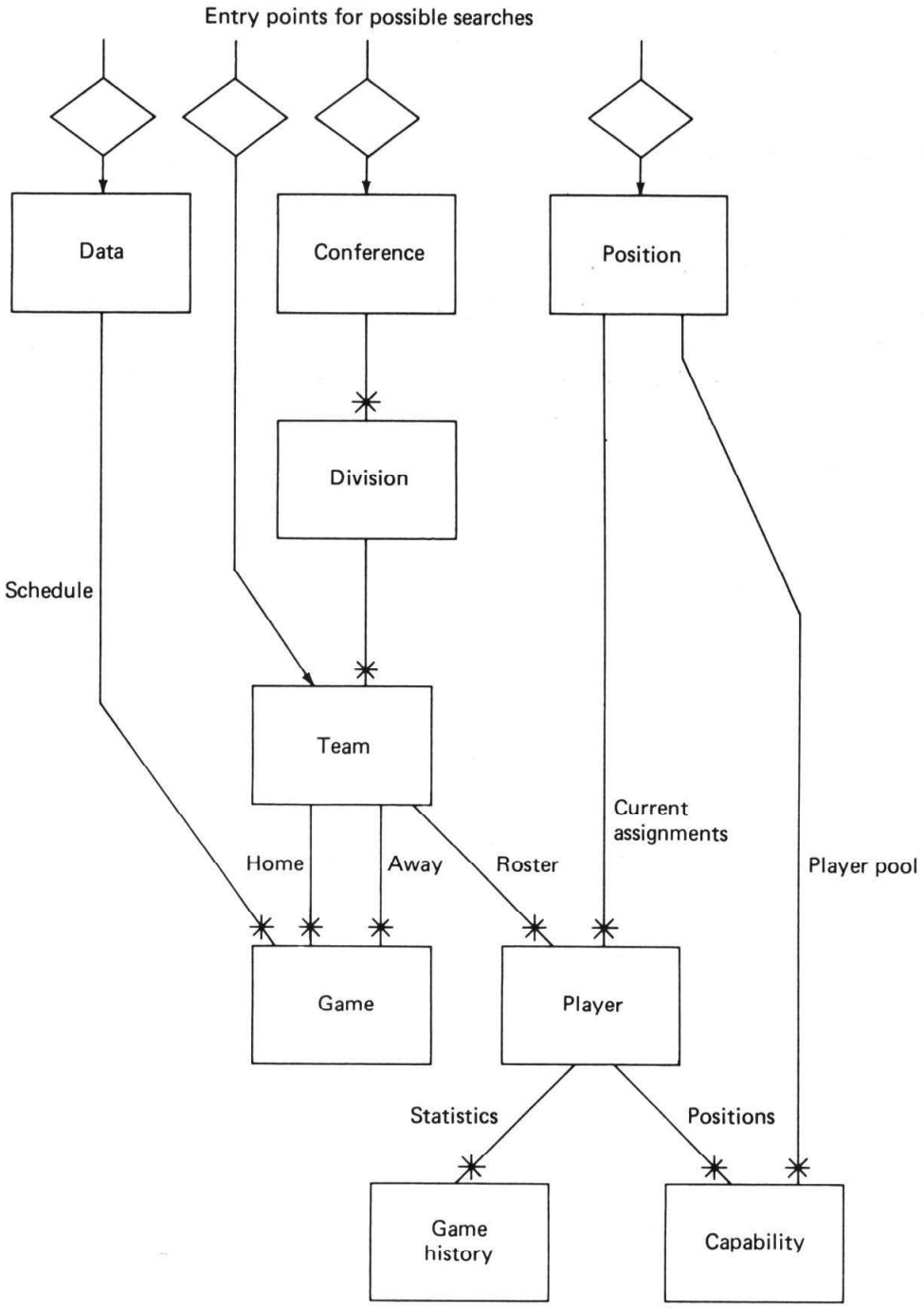


Figure 3-32 NFL football database.

There can also be `NULL` data fields, but since there are many types of records for specific purposes the overall file will be relatively dense. If all data fields are filled, and no redundant data is kept, the number of fields used is commensurate with the parameter a' which was used to enumerate the number of valid attributes for a record. The use of header records in fact reduces the redundancy of attributes found in files without the structure provided in the multiring file.

Header Records Each ring will have a header. This header is either an entry point, a member of another ring, or both. The header for the ring of department `employees` will be the list of `departments`; the header for the ring of `welders` will be a member of the ring of `professions`; and the `employee`'s record may be the header for a ring listing family members. When a ring is entered during a search, the point of entry is noted so that when this point is reached again, the search can be terminated.

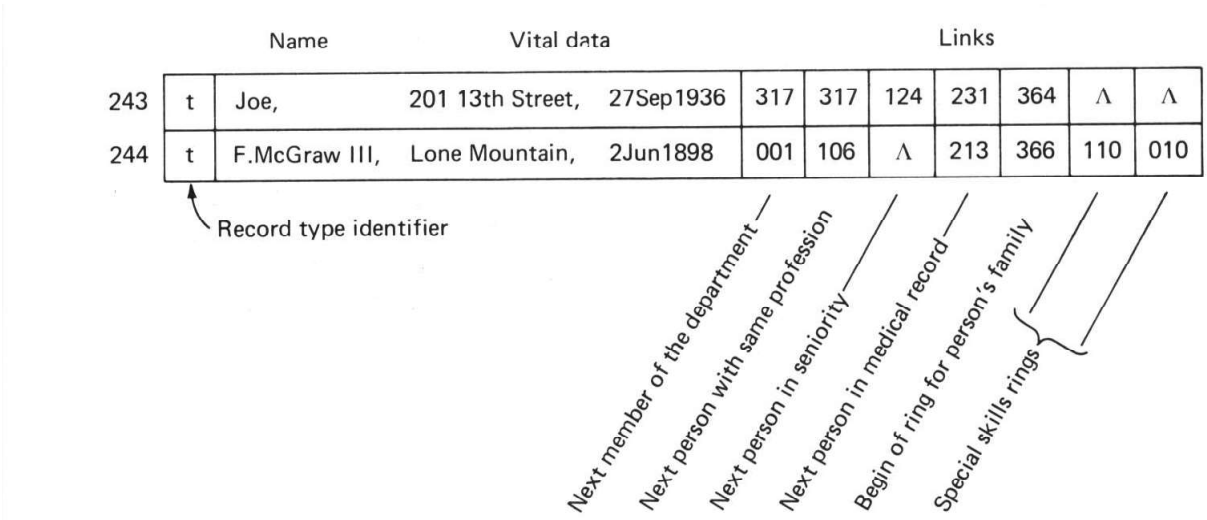


Figure 3-33 Record field assignments for a ring file.

Manipulation of Rings As shown in the examples above, the typical multiring file organization avoids redundancy of data by placing data common to all members of a ring into the header record of the ring. For example, the name and other data about the `department` of an `employee` appears only in the header record. A negative effect is that, in the basic ring design, whenever a record is to be fetched based on a combination of search arguments, the match of all search arguments with data values applicable to the record cannot always be performed using only the attributes kept in the member or the header records accessed during the search along one path. Two alternatives can be used:

1 A *parallel search* through all rings identified in the search argument can be made, terminating at the records at the intersection of these rings.

2 An *initial search* can be made according to the attribute with the best selectivity. The records collected are then checked for appropriateness by locating the header records for the other attribute types needed and rejecting records with inappropriate data values.

This latter process applied to the simple structure of Fig. 3-26 yields the steps shown in Example 3-11.

Example 3-11 Searching through a nonredundant multiring file.

```
Query:
Find an Employee with Location ="Thule" and Profession="Welder".
    Enter Location chain;
    For each member record determine if key =  Thule;
    When found follow Employee chain;
      For every Employee record the profession must be determined
        Follow the profession chain;
        When its header record is reached,
          then inspect profession header for key =  Welder
          If the field matches the search key
            then Employee member record becomes output;
      Continue with the next Employee record;
    When its header record, the Location =  Thule is reached,
      then the result is complete.
```

The importance of the final pointer to the header which transforms a chain into a ring is clear here, but the cost of obtaining data from the header can still be great. Hence it may be desirable to still keep important descriptive data redundantly in the record, or to expand the structure to allow easy access to header records. Auxiliary constructs for that purpose are presented in Chap. 4-7.

Design Decisions for Rings Selecting the attributes for which rings should be established uses similar critera used for the decisions to determine the attributes to be indexed in an indexed file. The cost of following the chains increases linearly with chain sizes. The sizes of individual chains can be reduced by increasing the number of chains and the number of levels in the structure of the file.

Increasing the number of levels (x) reduces the expected chain lengths (y), since the number of partitions for the lowest-level (1) records increases. If all chain lengths in the file are equal, the expected chain length according to one hierarchy, say `locations, departments, employees` in Fig. 3-30, is

$$y = \sqrt[x]{n} \qquad \langle \text{equal chain lengths} \rangle \; 3\text{-}91$$

since the product of all the chains (`#locations * #departments_per_location * #employees_per_department`) is $y^3 = n$. This effect is shown graphically in Fig. 3-35. Equation 3-91 is equivalent to stating that $x = \log_y n$, similar to the number of levels found for indexes (i.e., Eq. 3-48). However, in this file organization the partitioning of the file by attributes and hence the number of levels determines the chain sizes; whereas the computed fanout determines the number of levels in an index.

We consider now the case where the desired record can be recognized without having to trace back to alternate headers. The search time for a lowest-level record decreases then proportionally to the x^{th} root of the record count, n, and increases proportionally to the number of levels, x.

Example 3-12 Alternative costs of searching through a multiring file.

Query:
`Find the welder with social security number = '123-45-6789'.`
Given the multiring file of Fig. 3-30 with the following record counts:
$10\,000$ employees with $10\,000$ social security numbers,
and 50 professions (of equal size, 200 members each);
also 20 locations with 10 departments of 50 employees each.
The first search alternative is to enter the employee ring at the entry provided for ordering by seniority and search for the employee by social security number,
takes $5\,000$ block accesses.
The second search alternative - by profession and then within a profession -
takes $25 + 100 = 125$ expected block accesses.
The optimum for two levels is $2\,\frac{1}{2}\sqrt{10\,000} = 100$
A third alternative - by location, department, and then employee -
takes $10 + 5 + 25 = 40$ block accesses
but is only possible if location and department are known.
The optimum for three levels is $3\,\frac{1}{2}\sqrt[3]{10\,000} = 33$

The block fetch counts seen here are much greater than the expected number when indexes are used; the retrieval of successor records, however, is more economical in this structure than it is when using indexed files. Locating an entire subset is quite fast, as shown in Ex. 3-13

An attribute which does not partition the file into multiple levels, such as a `social security number` which has a perfect selectivity, is not very useful as a ring element in a `personnel` file, since to find someone in this manner would require $n/2$ fetches. Finding a specific `welder`, however, is done fairly efficiently by searching down the `profession` chain and then through the `welder` chain. The file designer hopes that these two chains will partition the search space effectively; the optimum is attained when both chains are equal in length, namely, $\sqrt{n}$. In practice this is of course never quite true.

Example 3-13 Summarization in a multiring file.

To summarize data for all the `welders`, we expect $25 + 200 = 225$ accesses. This number of records probably cannot be obtained more efficiently by any of the preceding methods. In order to gain a complementary benefit in access time, locality of rings has also to be considered in the file design.

Clustering of Rings Records which are frequently accessed together are obviously best stored with a high degree of locality. One single ring can, in general, be placed entirely within one cylinder so that all seeks are avoided when following this particular *clustered* ring.

When frequent reference to the header record of the ring is needed, that header record may also participate in the cluster. Now the ring at the next-higher level will be difficult to cluster, unless the total space needed by all member records and their

ancestors is sufficiently small to be kept on one or a few cylinders. With success traversal according to a clustered hierarchy will take very few seeks. In substantial files a given record type at a lower level will require many cylinders, and traversals of rings other than the clustered ring will require seeks.

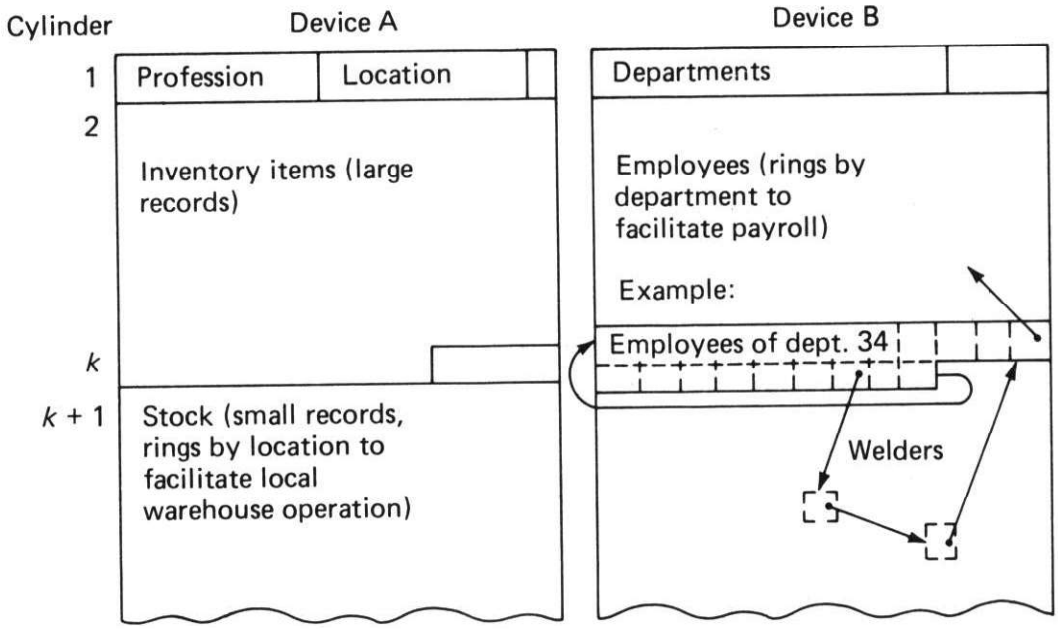


Figure 3-34 Assignment of rings to storage.

A possible assignment for the data of Fig. 3-30 is sketched in Example 3-14. Locating an `employee` via `department` will require only a few seeks within the `department` ring, to find the `employee` ring for the `department`. If there are many `employees` in some `department`, there may be some cylinder breaks in the ring. Finding an `employee` by `profession` may require seeks between nearly every `employee` record. `Stock` in a warehouse is found with few seeks for a given `location`; these rings are apt to be long, and will require some seeks at cylinder break points. To relate `stock` to the `inventory` data will take seeks between records, but these rings may be short if there are few `locations`.

In a dynamically changing database, optimal clustering is difficult to maintain and its benefits should be partially discounted. A reorganization may be needed to restore clusters. ‡‡‡‡‡

‡‡‡‡‡ **Categorizing Real Attributes** Some attributes, such as `profession`, can be naturally enumerated as a number of discrete categories, e.g., `welders, bookkeepers`. Such a categorization will have good selectivity and the file will be partitioned to match the natural structure of the data. Attributes that represent real or continuous data, e.g., `weight` or `height`, although they may have a high selectivity, do not provide effective partitioning unless they are artificially categorized. A statement to define discrete categories for a continuous variable is shown in Example 3-14.

Example 3-14 Discretization of continuous data.

```
DECLARE height
  ONE OF { <150cm, 150-154cm, 155-159cm, 160-164cm, ..., >189cm }.
```

If continuous variables are used at low levels, so that these rings area reasonably small, then the rings need not be broken. A simple ordering within the ring will aid access. Ordering of members of a ring is frequently used. ######

3-6-2 Use of Multiring Files

The concept of related master and detail records can be traced to traditional data-processing procedures. Multiring structures are the basis for some of the largest databases currently in use. Management information systems where much of the system operation involves tabulating, summarizing, and exception reporting have been implemented using these multilinked lists. Examples of such operations were shown in the introduction to this section.

Some problems in geographic and architectural space representation also have been approached with a multiring approach. Current developments in integrated multifile systems depend greatly on the capabilities provided by ring structures. A problem with the multiring structure is that a careful design based on prior knowledge of the data and usage pattern is required before a multiring file can be implemented.

Ring structures have been implemented in practice with a number of important additional features. We will mention several of these in Chap. 4; they all will make the analysis of file performance more complex.

3-6-3 Performance of Multiring Files

The performance of a multiring system depends greatly on the suitability of the attribute assignments to particular rings. In the evaluation below, we will assume that the file structure is optimally matched to the usage requirements.

> We will analyze the performance for access to lowest level records in a hierarchy of rings. Each record can be adequately identified when accessed, so that no tracing from a candidate goal record to its header records is required. The record formats are fixed within a record type. In each record, there will be a fixed number of data and linkage fields, denoted as a'_{data} and a'_{link}, respectively.
>
> We also assume an ideal partitioning of the file by the attributes, so that the rings on the various levels are of equal size. We neglect benefits to be gained by locality management of the rings, or clustering. The effects of these two factors may balance each other.

Record Size in a Multiring File Since many distinct record types will exist in a multiring file, an accurate estimate is obtained only by listing all types, with their respective frequency and size.

In a file designed for minimal redundancy we expect that in the most frequent lower-level records

$$a'_{data} + a'_{link} \leq a' \qquad \langle \text{nonredundant} \rangle \text{ 3-92}$$

since each data element is either explicitly represented or found by reference to some header record; one linkage pointer can replace a number of attributes.

If there is complete duplication of information in attribute fields and linkage fields to permit rapid selection of individual records, but still no empty fields, then

$$a'_{link} \leq a'_{data} = a' \qquad \langle \text{redundant} \rangle \text{ 3-93}$$

Linkages to the same header record are not included.

In practice we find values in between these ranges, and relatively few linkage fields. Some important data fields will be kept redundantly to avoid excessive header referencing. A reasonable first order estimate is to let

$$a'_{data} + a'_{link} = a' \qquad \langle \text{estimate} \rangle \text{ 3-94}$$

if the files have been carefully designed to satisfy a known query pattern.

The size of a data field is again averaged at V, and a linkage field contains pointer values of size P. One field of size P is prefixed to each record to allow identification of the record category. Then, for an average record,

$$R = P + a'_{data}V + a'_{link}P \qquad \text{3-95}$$

If the records are members of relatively few rings, the size difference of data values and pointers will not matter, so that with the estimate of Eq. 3-94

$$R \approx a'V \qquad \langle \text{estimate} \rangle \text{ 3-96}$$

Again, individual record categories will vary about this estimate, and where R is used to estimate y and block retrievals the specific values are needed.

Fetch Record in a Multiring File The time to fetch a record is a function of the number of chains searched and the length of the chains. We assume that the record contains enough data so that when it is found according to one particular accessing sequence, it can be properly identified. The record hence is found by searching down a hierarchy of x levels. We thus have to consider here only one ring membership per record type.

The length of a ring (y) depends on the size of the file, the number of levels, and how well the file is partitioned into the rings, as shown in Fig. 3-35. An estimate for the ideal case was given in Eq. 3-91 The lowest level (1) contains a total of n records in all its rings together, discounting other record types at higher levels or other hierarchies in the file.

If a single record is to be fetched, the number of hierarchical levels should be matched by the number of search arguments, a_F, in the search key. Example 3-12 illustrated this interdependence.

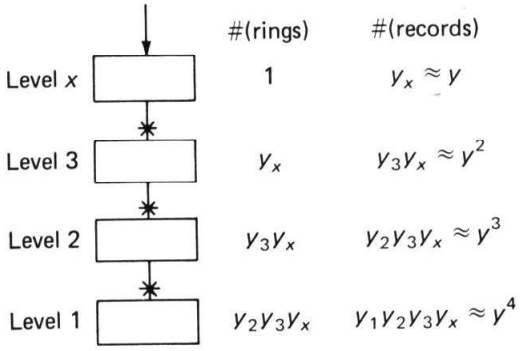


Figure 3-35 File size at four hierarchical levels.

If there are fewer search arguments in the key, an entire ring, i.e., a subset of the records of the file, will be retrieved; if there are more arguments, there is redundancy, and attributes unproductive in terms of an optimum search strategy can be ignored. For a standard fetch the condition $a_F = x$ holds, and with Eq. 3-91

$$a_F = x = \log_y n \qquad 3\text{-}97$$

In order to traverse one level, we expect to access $\lceil y/2 \rceil$ records, using the same reasoning which led to Eq. 3-2. In order to locate a record at the lowest level, the most likely goal, we will traverse x rings so that $\frac{1}{2}x\,y$ records will be accessed. Assuming random placement of the blocks which contain these records,

$$T_F = \frac{x\,y}{2}(s + r + btt) \qquad 3\text{-}98$$

Using the expected number of levels, a_F for the hierarchy used by the query, and the corresponding value for y,

$$T_F = \frac{a_F \sqrt[a_F]{n}}{2}(s + r + btt) = fna(s + r + btt) \qquad \langle \text{in query terms} \rangle \; 5\text{-}99$$

where *fna* represents the terms based on the values of a_F and n.

Table 3-12 lists values for *fna* given some typical values for n and a_F.

Table 3-12 Values for the file-access factor *fna*.

	fna		
a_F	$n =$ 10 000	$n =$ 100 000	$n =$ 1 000 000
1	5 000	50 000	500 000
2	100	318	1 000
3	33	72	150
4	20	36	72
5	20	25	32
6	18	24	30

fna = file-access factor
a_F = number of levels of rings in the search hierarchy
n = number of records on the level to be accessed

We see that the fetch performance is critically dependent on the optimum structure of the data, since the query format matches the relationships inherent in the file structure. The search time may be reduced when important rings are placed

so that they tend to cluster on a single cylinder. Large databases of this form have also been implemented on fixed-head disk hardware where no seeks are required, so that the term $s = 0$. Auxiliary rings or linkages will have to be employed if the search length for some foreseen and important search attribute combinations becomes excessive. Unplanned queries are difficult to satisfy with this structure.

Get-Next Record of a Multiring File The next record for any of the linked sequences can be found simply by following that chain

$$T_N = s + r + btt \qquad 3\text{-}100$$

Note that in this file organization we ahve serial ordering by multiple attributes. The "next" record can be selected according to one of several (a_{link}) attribute types. The only other fundamental file organization that provides multiattribute serial access is the indexed file.

If serial access via one particular ring is frequent, the records for one ring can be clustered within one cylinder, as discussed with Fig. 3-34. Then $s = 0$ when accessing members of that ring. Getting a next record tends to be an important operation in data processing using this file structure, so that multiring files will often use clustering. It is wise, however, to evaluate the effect conservatively.

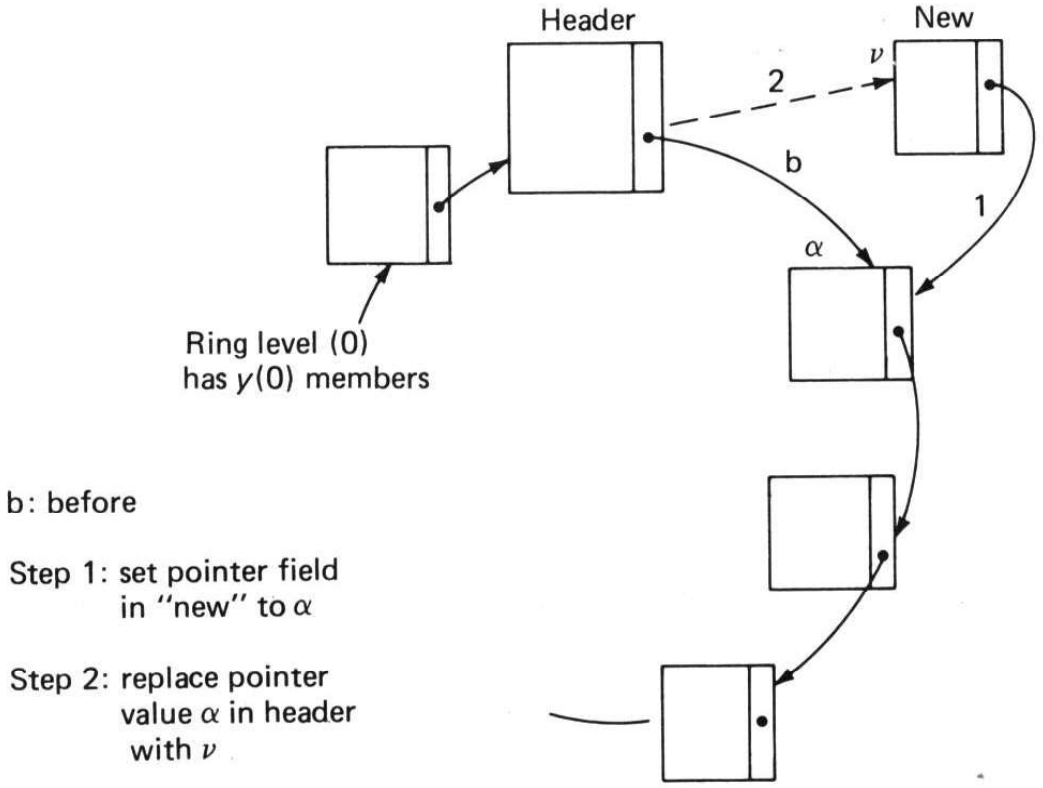


Figure 3-36 Insertion into a ring.

Insert into a Multiring File Adding a record to a multiring file is done by determining a suitable free space for the record, locating all predecessors for the new record, taking the value of the appropriate links from the predecessors, setting it into the new record, and placing the value of the position of the new record into the predecessor link areas.

The total effort, except for placing the record itself, is hence proportional to a'_{link}. Some effort can be saved at the bottom levels when the chains are not ordered. The sum of the fetches in all hierarchies into which the record is linked, the predecessor rewrites, and the placing of the final record is basically

$$T_I = a'_{link}(T_F + T_{RW}) + s + r + btt + T_{RW} \qquad 3\text{-}101$$

The cost of inserting a new record is obviously quite high, especially if the record is a member of many rings. In Chap. 4-7-1 structures that allow faster location of the predecessor records are shown.

‡‡‡‡‡ **Unordered Attributes** If the records are ordered by attribute value, the chain has to be searched for the correct insertion point. If the chain connects identical attribute values, such as all `welders`, then only the header record has to be read, modified, and rewritten. For rings where the order does not matter, new records can also be inserted in the initial position. This will result in an inverse chronological sequence. Such a sequence is also desirable for fetching single records, since recent data is found rapidly. The process is illustrated in Fig. 3-36.

This linking has to be carried out for all a'_{link} rings of which the new record is a member. For each ring which is maintained in unordered form, $y/2 - 1$ block accesses can be saved in insertion or update. We will not account for this saving in the evaluations, but the term can be subtracted if appropriate from Eqs. 3-101, 3-103, and 3-104. ‡‡‡‡‡

Update Record in a Multiring File If only data fields are to be changed, the update requires only finding the record and rewriting it. We can assume that updated records do not change type and keep the same length. Then

$$T_U = T_F + T_{RW} \qquad \langle\text{unlinked attributes}\rangle\ 3\text{-}102$$

for data field changes. If the record to be updated is initially located via a `Get-Next` operation, as is frequent in this type of file, the initial T_F can be replaced here, and in Eqs. 3-103 and 3-104, by T_N.

Updating of records can also require changes to the linkage. Only linkages whose values have changed need to be altered, since the updated record and its predecessors can be rewritten into the original position. Two cases exist here, since a new predecessor can be either in the same ring or in another ring, although at the same level, as the current predecessor.

If the changed value for one linkage is still within the type of the search attribute used to locate the record, then the point where the new record is to be inserted will be in the same ring and will require only $\frac{1}{2}y$ expected accesses of records in the ring. If, for instance, `stock items` of a given type are kept sorted within their

ring by `weight`, a change due to a new design can require an item to shift position within the ring. To achieve such a shuffle, the new and the old predecessors have to be located for every link that is to be changed. We assume conservatively that a circuit of length y around each ring is needed to find both.

To relink the updated record into a new place within the ring, the pointers of the three records are interchanged as shown in Table 3-13:

Table 3-13 Procedure for update in a ring.

The new predecessor is set to point to the updated record.
The old predecessor is set to point to the former successor record of the updated record.
The new record is rewritten after all the new predecessor records have been read, since its successor links are copied from the new predecessors.

For this final rewrite no reading is required, but the position has to be recovered, requiring $s + r$ before the updated block can be written (btt).

The total for this case includes the initial fetch, the search for the predecessors of a_U links, the rewrite of these new and old predecessor records, and the final repositioning and rewrite of the new record. All this requires

$$T_U = T_F + a_U\,(y(s + r + btt) + 2\,T_{RW}) + s + r + btt \qquad \langle\text{links in ring}\rangle \text{ 3-103}$$

If, during the search for the old position of the record, a note is made of each predecessor and also if the new position is passed, the accesses to locate one of the predecessors and sometimes the accesses required to find the new predecessor record can be eliminated. The term, y, can then be reduced to $\frac{1}{2}y$ or even to 1 for one of the links. Since a_U is often 1 or small, the difference can be significant.

If updates cause ring membership changes within the same ring type, the new insertion positions have to be located. An example of this case is an `employee` who changes `departments`. This could be done by searching from the top, as was necessary in the case of record insertion. It may be faster to go to the header records using the ring, and then search through those header records for the desired place (`'Hawaii'`). The predecessor in the old ring (`'Calhoun'`) has also to be located and rewritten with a pointer to the previous successor (`'Morris'`). This process is illustrated in Fig. 3-37. The insertion place (`'Kulahelo'`) still has to be found.

We denote the number of links requiring access outside the current ring by a_W. The first choice, searching for the header records of a_W links from the top of the hierarchy, again requires a_W fetches of time, T_F. For updates less than three levels from the top (level numbers x, $x-1$, and $x-3$), it is advantageous to locate the insertion point from the top. The general approach is to navigate via the headers.

Using the header requires, for each changed outside link, finding the current header, finding the new header, and finding the new insertion point. A fourth scan of a ring is required to find the old predecessor, needed to achieve unlinking. The expected accesses in each ring are again $y/2$. The entire process requires for all fields together $a_W\,4\,y/2$ accesses. The entire process now takes

$$T_U = T_F + a_W\,(2\,y(s + r + btt) + 2\,T_{RW}) + s + r + btt \quad \langle\text{links outside of ring}\rangle \text{ 3-104}$$

In practice there may be both a_U links that can be updated within the ring and a_W links that require the complexity of ring changes. In that case both terms apply with their factors.

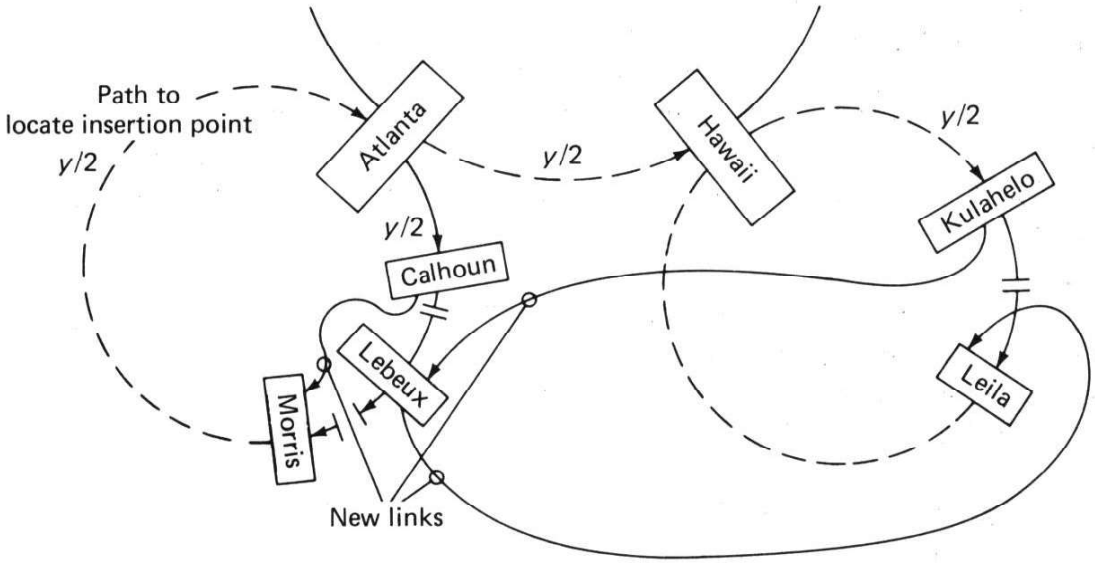


Figure 3-37 Finding a new linkage point.

Read Entire Multiring File Exhaustive searches may be carried out by serially following any of a variety of possible linkages. A proper understanding of the file design is required to assure that no records will be read more than once. The cost will be relatively high, since the process will have to follow the linkages given in the records. The alternative, searching sequentially through the space, may not be easy, since the records have a variety of formats, and the description of a record type is necessary to make sense of the fields obtained.

Reading according to the chains requires, in addition, that a header record is accessed for each subsidiary ring. Both the old and new header record is needed to move between two rings, but it should be possible to keep a stack of x old header records in primary memory. Then

$$T_X = n\,(1 + \tfrac{1}{y})(s + r + btt) \qquad 3\text{-}105$$

Reorganization of a Multiring File Reorganization is not required as part of normal operating procedures. This is made obvious by the fact that a database system based on multiring files, IDS, available since about 1966, did not have an associated reorganizationto be followed program available until 1975. Only when reformatting of record types is required will such records have to be rewritten. This may require only a partial reorganization of the file, since the changes are limited to rings of the levels using those records types. If one ring can be processed

in memory at a time the costly rewriting of pointers found in updating can be avoided. The entire ring of y records is read, rearranged, and then rewritten at a cost of $y(s + r + btt) + c + y(s + r + btt)$. To fetch the rings to be reorganized we access the headers with one fetch and subsequent get-next operations.

A reorganization of n' records of one level, according to this incremental approach, is then

$$T_Y(n') = c + T_F + \frac{n'}{y} T_N + 2\,n'(s + r + btt) \qquad \text{3-106}$$

where, if the reorganization takes place at level λ from the top (level $= x - \lambda + 1$), an estimate for $n' \approx y^{\lambda}$.

3-7 EXTERNAL SORTING

Arranging records in order is the most frequent and time-consuming activity in data processing. It has been estimated that 25% of all computer time is devoted to sorting. Although new approaches to data-processing reduce this fraction, sorting remains an important activity.

3-7-1 Motivation There are several reasons for sorting data:

1 Presentation: when voluminous results are presented to the users, they should be presented in an easy-to-use form. For instance, a summary of company sales is best shown by product name to help checking if problems with a product have affected sales.

2 Merging of files: often information on files created at different times or at different sites has to be combined. If the files are sorted differently, then looking for related records requires scanning all of one file once for each record in the other file. With files of size n the cost is $\mathcal{O}(n^2)$. If the files are sorted the same way, then scanning the files forward together reduces the effort to $\mathcal{O}(n)$. Sorting is then profitable if it costs sufficiently less than $\mathcal{O}(n^2)$.

3 Creating an index: indexes permit quick retrieval to records, as shown in Chap. 5. An index, even for a large file, can be created rapidly from a sorted file.

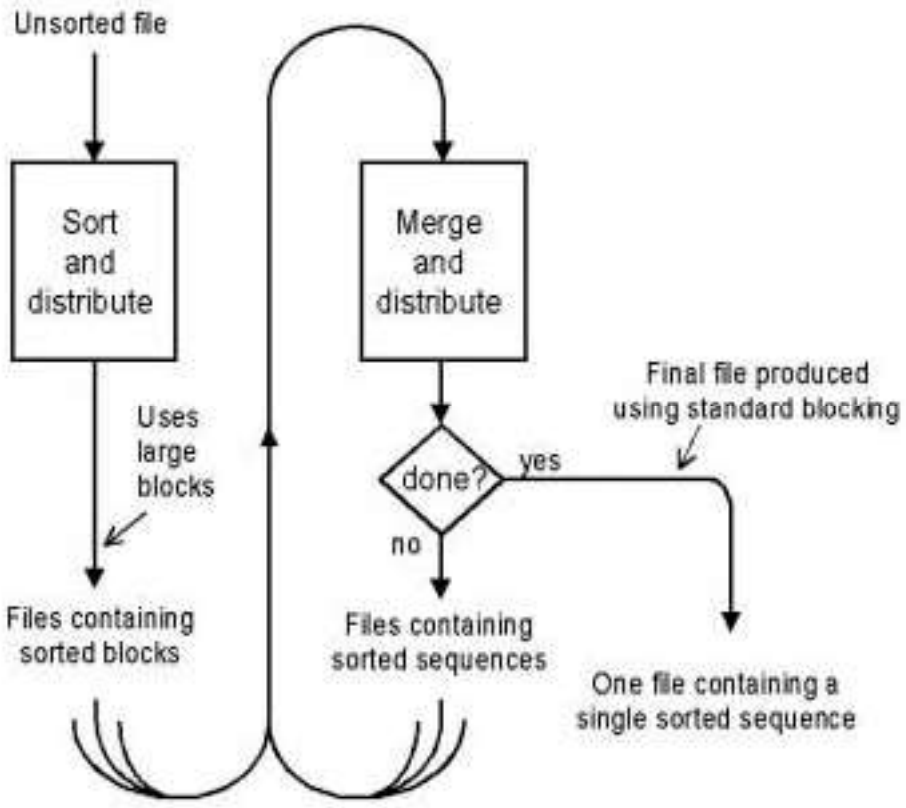


Figure 3-38 Control flow in merge-sort.

Many algorithms for sorting have been developed. When files do not fit into memory we require *external sorting*. Only the class of *merging algorithms* is effective for external sorting. In this box we present the principal algorithm used to reorder the records of large files.

3-7-2 A Sorting Method

The sorting methods of interest to us here are *e*xternal sorts, where the amount of data is greater than be sorted in memory. Most external sorts are *m*erging sorts, where partially sorted sequences are successively combined, until only one sorted sequence remains.

A *merge-sort.* sorts n records in $\mathcal{O}(n \log n)$ operations. It consists of two phases, sort and merge. The merge phase consists of multiple merge passes, as shown in Fig. 3-39.

The merge-sort operates on sets of records placed into large working buffers in memory of size B_s, which we call sort buffers, often much larger than the typical buffer size B used in prior analyses. The sort buffers will be written and read onto working files, each buffer will correspond to a *sort block* on file. As seen below, the performance of a merge-sort improves when the sort blocks are large.

The constraint on B_s is the available real memory. A two-way merge-sort algorithm requires four sort blocks, so that the sort blocks may be made as large as a quarter of available memory (as shown in Fig. 3-39), once all systems software, sorting programs, and other buffers have been accounted for.

The Sort Phase We begin by reading in data records from file blocks of the source file, filling a sort-block buffer. A buffer will contain $Bfr_s = B_s/R$ records. Once a buffer has been filled with records, it is sorted in memory, using any reasonable internal sort algorithm which is *stable*. (A stable algorithm keeps records with equal keys in the same order that there were submitted [Knuth^{73S}.)

If we have a fast internal sort algorithm, the time to sort the record can be overlapped with the time needed to fill a second sort buffer from the source file. This requirement is not easily met, but since the sort phase of the merge-sort is not the dominant cost we will assume this condition to be true.

When the first sort block has been sorted, and the second sort block has been filled, we start three processes in parallel:

1 The first sort buffer is written onto merge file i, initially $i = 1$.
2 The second sort buffer is internally sorted. An extra buffer is available if needed.
3 A third sort buffer is filled from the source file.

The processing of the source records is repeated for $i = 2, \ldots, i = m$, where m is the number of merge files to be created. Then i is reset to 1, and the processing continues until the source file is empty and each of the m merge files contain about n/m records. More precisely the merge files will contain up to $\lceil n/m/Bfr_s \rceil$ sort blocks; files labeled $i > 1$ may contain only $\lfloor n/m/Bfr_s \rfloor$ sort blocks.

The number of merge files m may vary from 2 to about 10. Having many merge files permits multiple files to be read or written in parallel. The amount of

parallelism available depends on the hardware architecture, as detailed in Chap. 5-4-4. Having more merge files increases the number of sort buffers needed in memory, reducing their size and reducing the effectiveness of the initial sort phase. Since even a two-way merge ($m = 2$) can keep 4 files operating in parallel, we will orient the discussion on $m = 2$, and also assume we have 4 sort-blocks in memory. This means that the merge file address parameter i, used above, alternates between $i = 1$ and $i = 2$.

The Merge Phase We are now ready to merge the merge files. The initial sort blocks from of each of the m merge files are brought into memory input buffers, and their records are compared, starting from the lowest ones.

A Merge Pass The record with the smallest key is placed in a merge output buffer $j = 1$, and omitted from the merge input. This process of selecting the smallest record continues, interrupted only by two conditions:

C1 When a merge output buffer is filled, it is written to a merge output file j.

C2 If a merge input buffer has been processed completely, that merge input block can be ignored during selection. When only one merge input buffer contains records, those records are directly copied to the merge output buffer, and the buffer is written out.

We have now written to the merge output file $j = 1$ a sequence of sorted records that is m sort blocks long. In the case of $m = 2$ we have merged two input blocks, each from one of the from two source files to create a sorted sequence of two output blocks on one merge output file. We then switch to a new merge output file by incrementing j. For a two-way merge j alternates between $j = 1$ and $j = 2$. A new set of m merge input blocks can be read and processed.

The merge processing continues for all input sort blocks. At the end of the merge input we may not have input blocks for all the buffers, but this case is treated as if the input buffers had no records left. This merge pass is now complete. The input files can be discarded or reused to hold the output of the successor pass.

Successive Merge Passes The files which were the output of the completed merge pass are reset to become the new input files. They now contain sorted sequences of length m sort blocks, and the successor pass has to take this into account. For the condition that an input has been completed, C2 in the description of the initial merge pass, the action is now generalized to become:

C2′ If a merge input buffer has been processed, that buffer is refilled using the next block from the same merge input file until the entire sorted sequence has been read. If the sorted sequence is processed completely, the merge input buffer can be ignored during selection. When only one merge input sorted sequence remains, the remaining records are directly copied to the merge output files and a new set of sorted sequences can be read.

The merging of sorted sequences continues to the end of the merge input files. Each merge pass processes the output sequences from the prior merge pass. With each merge pass the length of the sorted sequences increases by a factor m. For a two-way merge the sequences double, redouble, etc.

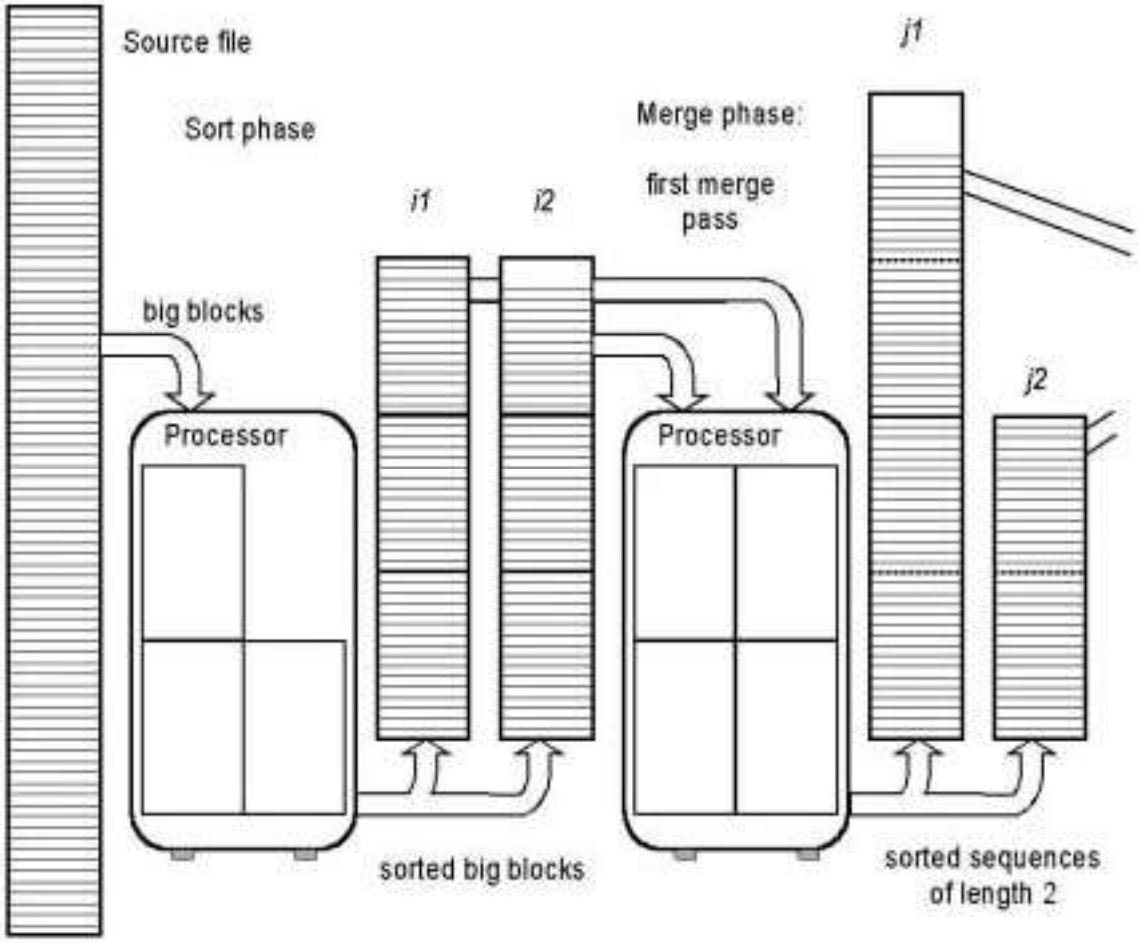


Figure 3-39*a* Data flow in merge-sort.

Merge Termination Eventually the sorted sequence is as large as the source file was. This means that we are done. Only one merge output file is written, and that file contains the desired output. Since we can predict when the final file is being written we can direct the output to a final destination. An output file can be blocked again to standard form or the results can be used immediately, say, to create an index.

3-7-3 Performance of External Sorting

The first phase of a merge-sort requires reading and writing of the entire file content, at a cost of $2T_X$. Since the files are processed sequentially $2T_X = 2b\,btt$ for b blocks of the source file. Parallel operations can reduce that cost, although then the cost of internal sorting must satisfy the condition of Eq. 2-22.

Each merge pass also requires reading and writing of the entire file contents, at a cost $2T_X$. The selection and copy operations during the merge are simpler than internal sorting, and are easily overlapped with buffered input and output.

The number of passes required in the merge phase depends on the size of the initial sort blocks and the power m of the merge. For n records of size R we create $b_s = n/Bfr_s$ sort blocks. To merge b_s blocks requires $\lceil \log_m b_s \rceil$ passes.

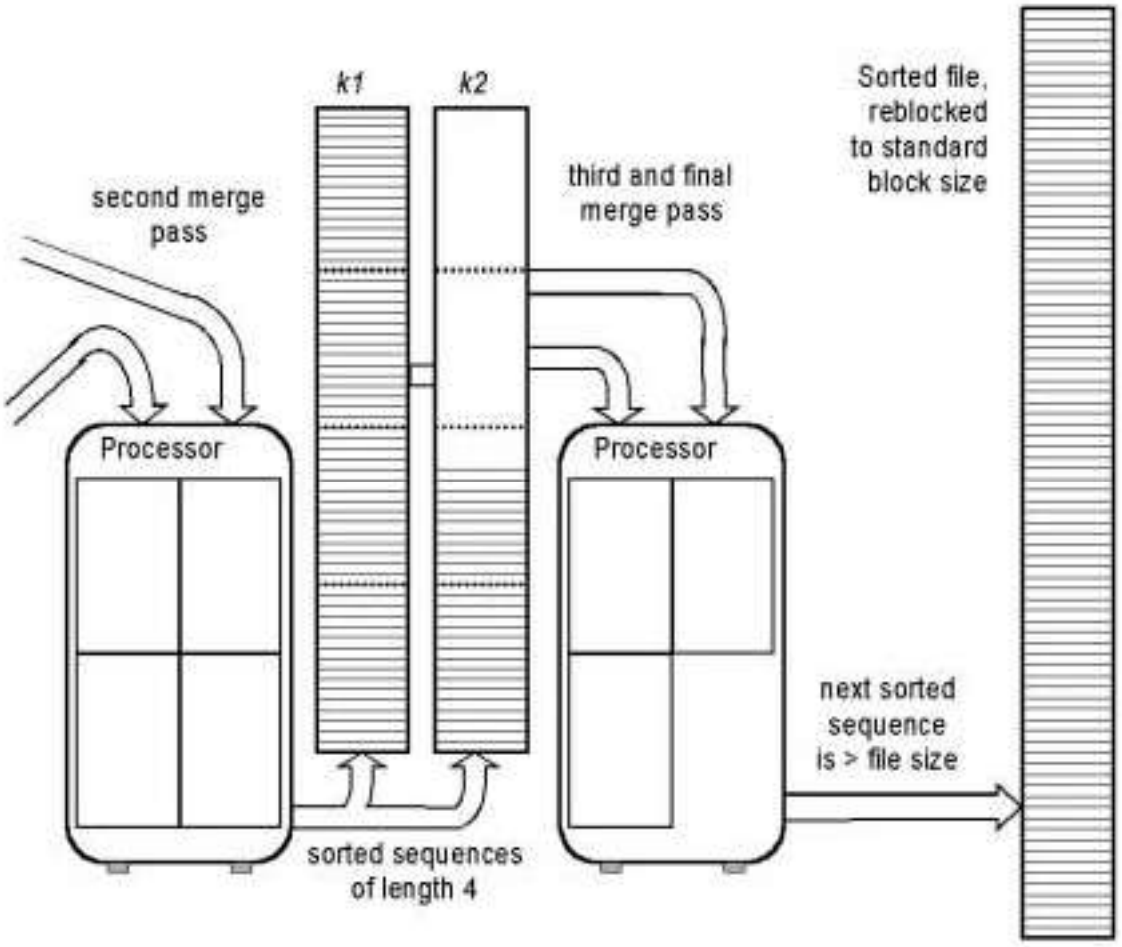


Figure 3-39*b* Data flow in merge-sort.

An estimate of the total cost for a merge sort with $m = 2$ is hence

$$T_{sort}(n) = 2b\,btt + 2b\lceil \log_2 b_s \rceil btt = 2n \left\lceil 1 + \log_2 \left(\frac{n}{Bfr_s} \right) \right\rceil \frac{R}{t'} \qquad 3\text{-}107$$

This estimate will be adequate for most cases, and matches the conceptual Equation 3-12. Since many systems have very specialized and efficient programs for sorting, it can be useful to determine the value of T_{sort} in a particular system from documentation or by experimentation.

More details can be found in Knuth[73S], which presents the algorithms and analyzes the performance of a number of external sorts. All have a performance which is $\mathcal{O} = n \log n$, and do not differ significantly from the simple two-way merge-sort described in Section 3-7-2.

3-8 REVIEW

In this chapter we have introduced a large number concepts uses in file structures and applied them to analyze their performance. We focus on data files, although computer systems have many other files, as directory records. Fig. 3-40 shows some of that complexity.

Review of Terms In this chapter we frequently referred 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 the *goal*. For 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. Search arguments can address other attributes; for instance, "`jobtitle ="welder"`". Now multiple records may be retrieved, each with different keys.

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 often does 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.

3-8-1 Basic Files

Useful basic concepts in encountered in Sections 3-1 and 3-2 are partitioning and selectivity. These concepts will be applied later when selecting indexes and will become critical for database query optimization.

> Attributes of a file have selectivity, ranging from perfect for unique attributes, to small for, say, binary attributes. To find an entry in a file the file is partitioned using selection on the attributes.

Useful techniques are self-describing fields having attribute names as well as values, the use of tombstones to indicate deletion, and use of probing to search a large file space. The use of a transaction file will also be seen to have broader applicability. If the transaction file is retained we have a log of all the changes applied to the file. Such a log can also be used to recover from disasters. Transaction logs provide an audit trail, useful when problems with files or their use or misuse have to be investigated.

The two initial file organizations shown here, while both simple, are structured quite differently. The pile file organization is flexible, and costly for information retrieval. The sequential file is rigid, and fast for common simple uses.

> There is a trade-off between flexibility and efficiency. If constraints are applied early in the process of design, flexibility will be reduced, but efficiency will be gained. The difference in efficiency is typically so great that constraints on flexibility must be applied to make a design practical. It is important to choose the least onerous constraint.

The distinction can be summarized as follows:

$$\begin{array}{rl} T_U(pile) = \mathcal{O}(1) & T_U(seq) = \mathcal{O}(n) \\ T_F(pile) = \mathcal{O}(n) & T_F(seq) = \mathcal{O}(n), \mathcal{O}(\log n), \text{ or } \mathcal{O}(1) \\ T_N(pile) = \mathcal{O}(n^2) \text{ or } \mathcal{O}(n) & T_N(seq) = \mathcal{O}(1) \end{array} \qquad 3\text{-}108$$

However, to obtain the better performance in sequential fetching requires elaborate programs. To better exploit the potential of sequential files we must add access structures, and that is the topic of the next chapter.

Alternative record structures demonstrated here are applicable to other file organizations, as well. At times, mixed record structures are seen, where stable information is placed into a fixed part which uses unnamed fields, while the remainder of the information uses named fields.

The method used to derive performance formulas should now be clear. The formulas are simply derived from the description of the operation being performed. In essence we obtain factors f which represent the frequency of the primitive computer and disk operations: compute, seek, latency, and block transfer.

$$T_{any} = f_c^* \, c + f_s^* \, s + f_r^* \, r + f_{btt}^* \, B/t$$

The f^* indicates the repetitive nature of the operations being performed.

As the three approaches to fetching data from a sequential file show, it is feasible to have more than one access method for one single file organization. One file organization has to serve all applications, but for a given access application program or transaction the best access method should be chosen.

3-8-2 Indexed Files

Indexes are the dominant means to improve access to files. They add a small amount of redundancy and provide rapid access to individual records for fetch and update. Since the performance of indexed files is dominated by the number of index levels accessed, we can summarize the result and state that

$$T_F = \mathcal{O}_{indexes}(x) = \mathcal{O}(\log_y n)$$

Within this growth factor are hidden many parameters, especially the fanout y, which can cause great differences in observed performance.

Fanout determines the performance of an index.
A large fanout reduces the number of index levels.
Fanout is increased by having large blocks and small entries.

Abbreviation of entries to increase fanout will be presented in Chap. 4-2-1.

We showed the alternative of static and dynamic index updating. Static indexes were adequate for indexed-sequential files. Dynamic indexes require much effort when inserting records, but reduce reorganization requirements. In situations where there is little data-processing staff they are preferable. Programs for static indexes are simpler, but create chains. We assumed uniform distributions when following index chains. Nonuniform insertion distributions can increase T_F considerably. The problem gets worse if recently inserted records are referenced more frequently. Push-through reduces that effect.

Having multiple indexes which produce TIDs permits partial match retrieval without having to retrieve other than the goal records from the data file.

Other decisions which are important when designing indexes are

- Selection of attributes for indexing
- Use of record versus block anchors
- Immediate versus tombstone record deletion techniques

Other design issues are covered in a description of index implementation, Chap. 4-3.

3-8-3 Review of Direct Files

The outstanding feature of direct files is that the records can be accessed in constant time, $\mathcal{O}(1)$, so that growth of the file has no effect on performance, as long as the density can be maintained. If the density varies, then, for open adressing, $\mathcal{O}(\frac{n}{m-n}) \approx \mathcal{O}(C^{2n/m})$ in common ranges of m/n.

Direct access trades flexibility to gain both constant and high performance. Since record placement is determined by a hashing computation, only one attribute of the record can be used for retrieval. Another inherent restriction due to the computation is that data records are expected to be of fixed length.

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

Some solutions to the problems listed here are shown in Chap. 4-6; they all carry additional access costs.

3-8-4 Review of Ring Files

The performance of ring structures is greatly affected their initial design, and eventually by the distribution of insertions. The depth of a search, x, is constant once

the design is established, so that the performance then grows as the fanouts, y, grow on each level, so that performance is $\mathcal{O}(n)$.

In this structure speed and complexity was traded to gain generality. In practice they lose some of the generality, because it is rare that all conceptually possible access paths will be implemented in an application. The user has then to know what paths will lead to the data.

Knowing which paths are most important causes the designer to select those that become rings. No changes can be accommodated later when the users and their programs are set, even though that actual usage differs from initial assumptions. It is even difficult to determine that the design was wrong. Measurements only show the implemented paths, and provide no direct information on other candidate paths. The user of ring structures then has to be careful and use foresight during the design phase.

The features of the*direct file* organization and the *ring file* organization, being quite different, actually so different that they complement each other, so that we will see them used together in Chap. 4-7. For instance, entry points into ring structures are typically found by hashing. Some of the differences are

	Direct files	Ring files
Access by	One attribute	Many attributes
Oriented towards	Fetch	Get-next
Record placement	Rigid	Flexible
File growth	Complex	Easy

BACKGROUND AND REFERENCES

The basic file designs presented here can easily be traced to early developments in data processing. References where these files are compared with other techniques are found with Chap. 4. Specific information on file access methods is best obtained from software suppliers, we do not cite their manuals.

A pile is, of course, the most obvious data collection. Most of the pieces of paper that cross our desks are self-describing documents, which can be processed without reference to an external structure description.

Ledgers and card files are the basis for sequential files; they are often divided into pages or folders with headings. There exist corresponding algorithms and structures applicable to memory. But when access is into a homogeneous space the access techniques differ, so that the evaluation of the algorithms and their options leads to misleading results when files are being dealt with.

The benefit of sequential processing is assessed by Charles[73]. Lomet[75] describes the use of tombstones and Leung[86] quantifies their effect in dynamic files. Rodriguez-Rosell[76] shows that sequential processing remains effective even on multi-programmed computers and Slonim[82] evaluates sequential versus parallel access.

File organizations have been analyzed at several levels of abstraction. A pile file is described by Glantz[70] and partitioning is exploited by Yue[78]. A language with a persistent pile file has been implemented by Gray[85]. Gildersleeve[71] and Grosshans[86] address programming for sequential files. Severance[72] evaluates sequential files and buckets, and the difference between fetch and get-next operations. Shneiderman[76] and Palvia[85] analyze batch fetching in sequential files. Willard[86] presents algorithms for direct update of sequential files and Nevaleinen[77] has rules for their blocking.

The problem of merging cosequential files has often been analyzed, a recent example is Levy[82]. In Shneiderman[78] a form of binary search is developed, Li[87] extends it to batched operations. Ghosh[69] proposes to use probing instead of lower-level indexes. Concurrent access of direct primary and overflow areas is proposed by Groner[74].

File systems which implement various methods are described by Judd[73]. Martin[75] describes many file operations in detail and provides design guidance.

Indexing In offices, large sequential files in drawers are partitioned using labeled folders. Labels on the drawers provide a second level of partitioning of these files to accelerate access. Multiple indexes, based on title, authors, and subject, can be found in any library.

Chapin[68] includes the programming issues of indexed-sequential files. Further information is available in the literature of software suppliers (see Appendix B). Seaman[66] and Lum[73] treat indexed-sequential files analytically. Allen[68] describes an application using large indexed-sequential files. Indexed-sequential files were considered in Brown[72] to comprise such a significant fraction of disk usage that the IBM 3330 disk system design was greatly influenced by their behavior.

The construction of index trees in memory has been frequently discussed in the computer science literature. It is important to realize when using these references that the index trees found in storage files have very high fanout ratios and do not contain the data within themselves. Trees containing data are discussed in Chap. 9-2. Landauer[63] considers dense, dynamically changing indexes. Bose[69], Welch in King[75], and Larson[81] evaluate access to indexed files using formal and mathematical models. Maio[84] evaluates block anchors versus record anchors.

The B-tree algorithm was presented and evaluated by Bayer[72O,S]. It is further analyzed by Horowitz[78]; Comer[79] provides an excellent summary, and Knuth[73S] includes some file-oriented considerations. Ghosh[69] combines B-trees with probing. A review by March[83] considers partial indexes.

Yao[78] analyzed the density of B-trees under random insertions and deletions. Held[78] and Aho[79U] evalute B-trees in relational databases. Indexes are used for bibliographic retrieval in Dimsdale[73]. Guttman[84] extends B-trees for spatial data.

An early system using multiple indexes, TDMS, is described by Bleier[68]; Bobrow in Rustin[72] develops this approach. Multiattribute access via indexes is evaluated by Lum[70], Mullin[71], and Schkolnick in Kerr[75]. O'Connell[71] describes use of multiple indexes.

Selection, and its optimization, among multiple indexes is considered by many authors, among them Lum[71L], Stonebraker[74], Shneiderman[74,77], Schkolnick[75] Hammer[76], Yao[77], Anderson[77], Comer[78], Hammer[79], Yamamoto[79], Rosenberg[81], Saxton[83], as well as Whang[85]. The work is limited by assumptions on the distribution of the index value distributions for the queries. Piatetsky[84] suggests modeling the distribution by a stepwise function.

Indexes are used for partial matching by Lum[70]. Strategies of combinatorial index organization were developed by Ray-Chaudhuri[68], Bose[69], Shneiderman[77], and Chang[81]. Procedures for estimating selectivity in a file are surveyed by Christodoulakis[83], Flajolet[85] and Astrahan[87] do it in $\mathcal{O}(n)$, Chu[82] uses statistical models. Federowicz[87] finds Zipfian distributions.

Ghosh[76] includes construction of optimal record sequences for known queries.

Direct Files and Hashing Finding data by transforming keys was first used for data in memory, for instance to refer to symbol tables of assemblers and compilers. Peterson[57] and Heising publicized randomized hashed access for early disk files. Hashed access with chaining and open addressing was analyzed by Schay[62, 63], Buchholz[63], Hanan[63], Heising[63], and later by Vitter[83] and Chen[84] as *coalesced hashing*. Larson[83] analyzes uniform hashing and Yao[85] proves its optimality. Amble[74] discusses optimal chain arrangements for direct files. Open addressing with linear probing has been further analyzed by Morris[68], Kral[71], Knuth[73S], and Bradley[85]. VanDerPool[73] analyzes for the steady state the optimal file design for various storage cost to processor cost ratios, both for a file using a separate overflow area and for the case of open addressing.

Many papers concentrate on the choice of hashing algorithms and the expected fetch chain length. Summaries of hashing techniques are given by Maurer[75] and Knott[75]. Gurski[73] considers digit selection by bit, a manual (IBM J20-0235) describes digit selection, Kronwal[65] presents the piecewise-linear-distribution algorithm, and Sorensen in King[75] presents an evaluation of distribution-dependent hashing methods, and Garg[86] describes some applications and algorithms. Kriegel[87] evaluates a variant. Waters[75] defends and analyzes linear transformations.

Bays[73] presents rules for reallocating entries to new, larger hashed access file areas, Fagin[78] uses tables to locate extensions and avoid multiple seeks for collisions. Litwin[80] developed linear hashing and Scholl[81], Burkhard[83], Larson[85], Veklerov[85], and Ruchte[87] present variants. Robinson[86] even maintains sequentiality.

Batch updating of a direct file is suggested by Nijssen in Codd[71]. Concurrent access of hashed primary and overflow areas is proposed by Groner[74] and Ellis[85] analyzes concurrency with linear hashing.

Ring Files Prywes[59] first proposed multiple chains for files, and Bachman[64] describes the multiring approach; and Dodd[66] implemented file access based on this concept. These ideas influenced the CODASYL standards, which encouraged numerous applications of this structure. Lang[68] presents another ring implementation and many illustrations are found in Kaiman[73]. Whang[82] provides a design algorithm for ring-structure selection.

Validation of ring pointers is considered by Thomas[77]. Further aspects of ring structures can be found in descriptions of database systems.

Sorting A detailed analysis of external sort algorithms is provided by Knuth[73S]. All the important algorithms have an $\mathcal{O}(n \log n)$ behavior, and differ by less than 28% between the best and the worst case shown. Many further references are given as well. Advances in hardware have changed the relative importance of the parameters used; for instance, tapes are rarely used today for sorting, so that rewinding and backward reading of tapes are no longer issues.

EXERCISES

1 Obtain the operating system manual for a personal computer (PC) and locate the table which describes the file directory of the system and compare it with information cited in Table 3-0. Note the differences. For each directory element missing in the PC state what the effects of the lack of information are.

2 Suggest ways to reduce the redundancy in the redundant file of Table 3-1*c*.

3 What is the difference between writing a block and inserting a record? What is the difference between reading a block and fetching a record?

4 For the list of applications which follows, choose either a pile file organization, a sequential file organization, or reject both, if neither can perform even minimally. State your reason succintly.

a A payroll which is processed once a week
b A file to collect accident data in a factory
c An airline reservation system
d A savings account file in a bank
e A file listing your record, cassette, and CD collection
f The files of the social security administration. (They are processed periodically.)
g The catalog of an automobile factory
h The catalog of a department store

5 Program the algorithm for a continuous search described as the topic of *Batching of Requests* in Sec. 3-1-3. Test it using a fetch-request arrival rate that is derived from a random-number generator which interrupts a pass through the file 10 times on the average. The file can be simulated by an array in memory. Report the result. Refer to Chap 6-2 for a description of a simulation process.

6 Give the expected insertion time per record inserted into a pile file where $R = 0.7B$ and spanned records are used (Sec. 3-1-3).

7 Determine the time for a batch insertion into a pile (see Sec. 3-1-3). What conditions are required for your result to be valid?

8 To speed retrieval of name records from a sequential file, you want to use probing. Where would you look for data to build the required table? How big a table would you use?

9 Write in a high-level language the procedure to actually insert a record into an unspanned and into a spanned blocked sequential file. Discuss the buffer requirements for both cases.

10 Discuss the differences in updating records for a pile file and a sequential file.

11 When reorganizing a sequential file as described in Sec. 3-2, it is possible that a newly inserted record on the transaction file is to be deleted by a later transaction entered in the log file. What does this require when sorting the transactions and merging the files? Draw the flowchart for the merge process.

12^p Consider again your initial file application from Exercise 1 of Chap. 1. List the attributes you will need, and assign them to files. Identify which attributes will require variable length fields.

13^p Discuss if use of pile files or sequential files would be reasonable for some of the files you expect in your application. List any objections the users would have if either method were chosen.

14^p Would periodic reorganization be acceptable for your application? What would be a natural reorganization period? Who would be responsible for assuring that reorganizations were done on a timely basis?

15 Compute the average fetch time for an indexed-sequential file where the overflow areas are placed on the same cylinder. Assume overflow area is equal to 25% of the prime file and reorganizations when overflow area is 80% full. Assume B = 2000, R = 100.

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

17 Equation 3-53 contains a simplification that upper level indexes add 5% to the index storage space. What does this say about the effective fanout y_{eff}?

18 Compare Eq. 3-40a and Eq. 3-41 for T_N. Why are they different? Evaluate both equations for

a $n = 100\,000$ and $o' = 1000$ and for
b $n = 100\,000$ and $o' = 50\,000$.

19 What type of indexed-sequential overflow processing would you chose (assuming you could) for the applications listed in Exercise 3-4.

20 Choose three out of the attributes shown for indexing a personnel file: `name, age, sex, job, number_dependents, salary_of_dependents, salary` ; and explain why you chose them.

21 Estimate the number of goal records obtained for a partial-match query on `sex` and `job` for 500 employees. There are 20 job classifications. Give a specific example of values for these attributes where you expect many more goal records than the average expected, and one where you expect fewer.

22 Prepare a flowchart for and compare the performance of a binary search through an index with the multilevel approach for indexed files.

23 Find a file system where you study or work or in the literature and determine the seven measures of performance (R, T_F, T_I, etc.) for its organization.

24 What is the difference in an index reorganization performance if $2x$ buffers can be allocated rather than just x?

25^p For the project started in Chap. 1, select which files should be piles, sequential, indexed-sequential, or multiply indexed. Evaluate the 7 performance parameters for each of those files, using values of the size parameters n, a, a', A_i, V_i appropriate for your application, and values of s, r, t for a reasonable storage device for that application.

26^p For the project started in Chap. 1, assume you are using fully indexed files. Estimate the selectivity for all attributes, consider their usefulness, and decide which ones should be indexed.

27 You are designing an indexed file system for use on a personal computer. A requirement is that, for 90% of the cases, the insertion of a record should be completed in 1 s. To assure meeting this constraint you will limit the number

of indexed attributes permitted on a file. What will this number be? State all assumptions.

28 Design a file and evaluate it for questions as are posed in the epigraph from Vonnegut, on the title page of Chap. 12.

29 In Example 3-9 division by 500 caused a collision. A nearby recommended value would be $2^9 - 1 = 511$ would not do so. Is that because 500 was a poor choice?

30 Write a program to perform a key-to-address transformation from a 20 character string using the 94 printable ASCII characters as shown in Figure 14-1 and generating addresses for 800 entries with a density of n/m of about 0.6 .

31 Insert records labeled r, e, y, ç, u, ê, ä, ö, ü, è, é, v into the file of Table 3-8 and diagram the growth of the file.

32 Discuss the collision probabilities when linear hashing is used.

33 Discuss the benefits and liabilities of combining a sequence-maintaining KAT and linear hashing.

34 You have a large direct file using separate overflow chains. You are aware that 10% of your records are used 50% of the time, and the other 90% of your records are used the remaining 50% of the time. You assume that within both groups the access distribution is uniform (equally probable).

Describe the processes required to keep the average access to records as fast as possible by ordering overflow records. Calculate the gain or loss in total efficiency versus the unordered organization for a file of 10 000 records with a primary allocation of 12 000 slots.

35 Compute the cost of retrieving all 17 `welders` from files that have as primary attribute the `Employees`' `SSN`. Parameters are $B = 2000, R = 125, A = 7, P = 3, s = 50, r = 20, btt = 10$. Make any further assumptions you need explicitly. Do this for an indexed file, a direct file, and a ring file.

36 What is the reorganization performance (T_Y) using the two-pass insertion procedure for a direct file described in Sec. 3-5-3?

37 Sketch the actual ring structure for Fig. 3-29.

38 Evaluate the average access time for a direct file using buckets of three records, linear search for free record space, and 20% excess capacity.

39 In Fig. 3-40 elements of a file system are shown: A Device and Unit Table indicates that we have two devices, one of type Unicon, a mass-storage unit, and a moving-head disk of type 2319. On the first track of each device is a pointer to a File Name Table (FNT) which lists the files stored on the device. The files on the disk are Personnel, Equipment, etc. Each file has a directory, which lists a.o. the subfiles. The Equipment file is comprised of a Data subfile, a Schema subfile, which lists the attributes stored in the Data file and indicates if they are indexed, and a number of Index subfiles. As shown in Fig. 9-26 two types of index structures are used: dense indexes with overflows (`I`) and TID-lists accessed by hashing (τ). Note that Horsepower is not indexed at all. The actual Data file is stored on the mass-storage unit (INFSYS02), at position thr4.

Identify and list the various component tables of Fig. 9-26 and indicate for each whether they belong to the operating system or to the file structure.

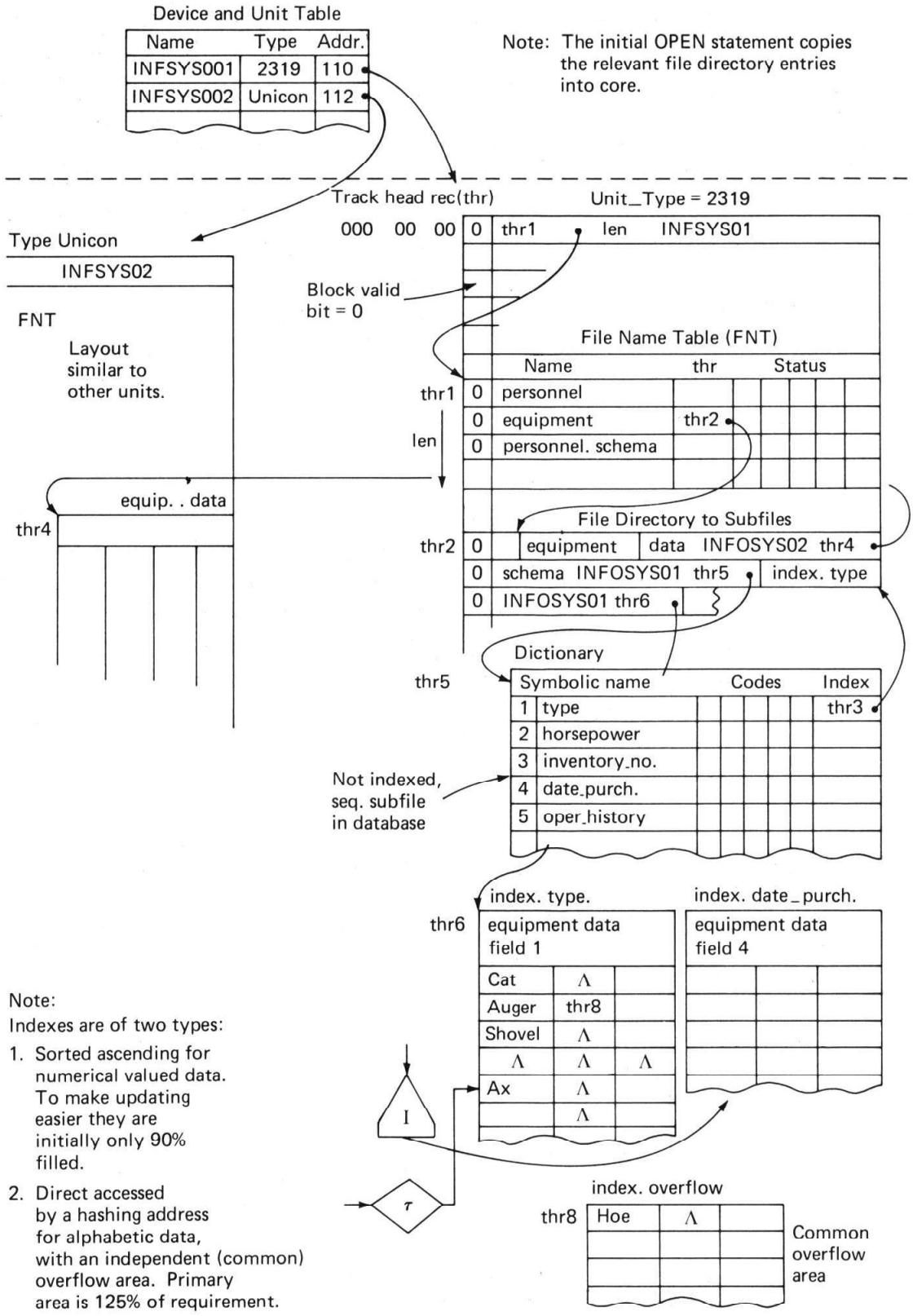


Figure 3-38 Structure of a file system to support database operations. The initial OPEN statement copies the relevant file directory entries into memory.

40 Initially the operating system obtains the Device and Unit Table in Fig. 3-40. To begin processing, a user transaction issues initial `OPEN` statements which copy the relevant file directory entries into memory. Make these two steps the beginning of a processing flowchart. Then continue the flowchart, showing the expected file access steps for a transaction program to answer the following questions:

a `Find the inventory number of our "Shovel".`
b `Find the inventory number of our "Hoe".`
c `Find the inventory number of equipment purchased "12Jun83".`
d `Find the inventory numbers of equipment having > 200 horsepower.`

41 * Using the sketch of a database system application given in Fig. 3-40 make a genaeral flowchart for a program to answer queries. Queries may be directed to single files for personnel or equipment, or to both. Since the sketch is not complete you have freedom to specify implementation details.

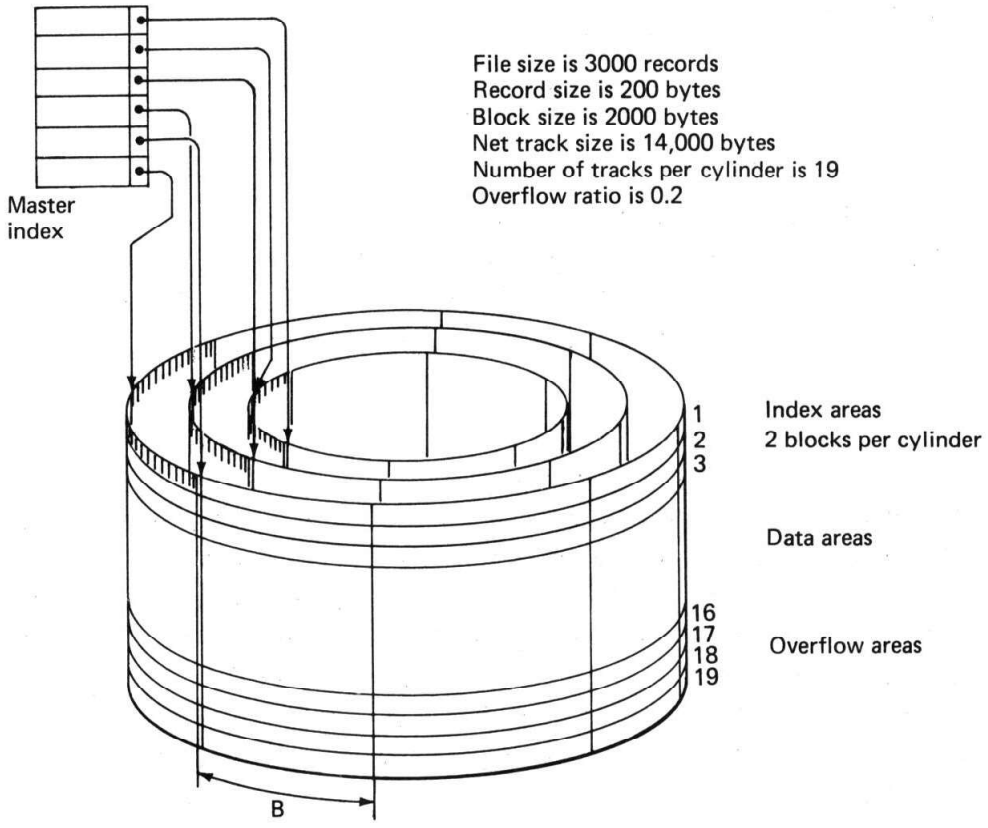


Figure 3-39 Illustration of indexed sequential file layout.

42 Calculate the total space required for a file as shown in Fig. 10-11, using record-anchored indexes. The calculation will initially evaluate the capacity of one cylinder and then determine the number of cylinders required.

Parameters to be used:

* More exercises for this design are found in Chaps. 4 and 11.

$R = 200$ (including the linkage pointer of 6 bytes length),
$B = 2000$, $V = 14, P = 6$,
Blocks per track = 7, surfaces per cylinder $k = 19$, Overflow allocation = 20%.

43^p Are there files in your application which are suitable for hashing? List all files you have defined and give the arguments for and against, and your decision.

44^p Are there natural hierarchies among the files in your application? Sketch a diagram using Bachman arrows among your files, use plain arrows for other relationships. Make a table which lists all relationships, hierarchical or not, and give a name to each, like `Author's publications`. List for each the destination and the source files, as `Author` and `Books`. For each file name the attributes which are matched along the relationship, as `Author.name` and `Publication.first_author`. Also state the type of relationship (`hier` or not) and the expected cardinality (0–99 books per author).

This page intentionally left blank.