¡le
DATABASE DESIGN, Gio Wiederhold.
Updated CHAPTER 3, file i3dbd-intro.tex April 4, 1996

# Chapter 3 part 1

# Introduction to Files

*All our knowledge may be said to rest upon observed facts.*

Charles Sanders Peirce
*from* "Abduction and Induction" *in Justus Buchler: "Philosophical Writings of Peirce"*

## 3-0 INTRODUCTION

The basic features desired of 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. The design of databases requires analysis for the prediction of performance, and this requires that the file organization be easy to describe in abstract terms. All these criteria tend to conflict with each other. The choice of the method of file organization 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 in the organization of files is nearly unlimited. In order to make the selection process manageable, we describe and evaluate, in Chaps. 3 to 5, six 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. 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 sections of Chapter 3 follow a consistent organization: After a general introduction, one section describes the organization and use of the first method and the next section evaluates its performance. Then a second method and its use is described followed by a section evaluating its performance. A final summary section in each chapter reviews the concepts encountered.

After each description a performance evaluation is given. The formulas developed use engineering-style approximations, they concentrate on first-order effects. They 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.

In the process of analysis a wide 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., $T_e = \sum p_i T_i \,|\, i = 1 \ldots k$.

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 performance is frequently only guessed at and results often differs 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.

An important by-product of studying this material is the development of a general analysis capability, not just for files, but for the performance of engineered devices in general. The student should try to understand the formulas and may use them for reference purposes but should not attempt to remember them or apply them blindly.

The consistency of the formulas permits the comparision 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 ignore the programming complexity of the alternatives.

The six fundamental types of files covered in this chapter are the pile, the sequential file, the indexed-sequential file, the indexed file, the direct file, and the multiring file. The choice of these six has been influenced by the fact that all these fundamental models are closely related to systems in actual use. In other words, the collection of these six is not intended to represent a mathematically minimal set of independent file organization methods. The collection presented does cover all common techniques.

## 3-1 GENERAL CONCEPTS OF FILE ORGANIZATION

In this section we introduce general concepts of file organizations and their performance. These concepts will be applicable throughout, but they will be specifically stressed in Chaps. 3 and 4.

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 that

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

### 3-1-1 File Directories

Associated with a file is 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 that is kept in the directory is associated with storage allocation and managed by the operating system.

The directory information is kept also 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 tye 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.

**CONTENT OF THE DIRECTORY** The names of the owners and authorized users for the file are coded to match the access authorization mechanism for the computer system. More material on users and their access rights is presented in Chap. ¡12¿. The allocation information specifies where and how much storage is allocated to a file. We show directory information, as typically kept for each file, in Table 3-1.

| Directory field | Example |
|---|---|
| File name | `payroll` |
| Owner | `Joe Paymaster` |
| 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` |
| ... | ... |

**Table 3-1** Elements of a file directory

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. ¡11-6¿.

**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 to retain the information in memory for further reference. Acquisition of directory information is performed during *opening* of a file. When *closing* of 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. Directory records are not further discussed as part of the subsequent file analyses.

### 3-1-2 File Description and Use

For each fundamental file organization we provide a description and the type of application it is suitable for. In the 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 maintanance
5 High reliability

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

In order to gain economy of storage, we wish to store the data with a minimum of *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 Sections 3.3 and 3.4 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.

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.

| | 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: data base course attendees**

| | Student | Courses taken | | | | | Exp. |
|---|---|---|---|---|---|---|---|
| | No. | CS101 | CS102 | Bus3 | EE5 | IE101 | years |
| 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: data base 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: data base course attendees**

**Figure 3-2** A dense, a sparse, and a redundant file

### 3-1-3 File Performance Parameters

Quantitative measures are necessary to evaluate file-system performance. The four types of computations introduced in Chap. 1-2-1 were build, update, retrieve, and data reduction. The 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 exceution. In Chap. 5 storage and time requirements are reconciled by using financial measures.

The seven measures will be provided for each of the six fundamental file organization methods. The seven measures used 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 to get the next record within the file
$T_I$ : the time to update the file by inserting a record
$T_U$ : the time 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 dealk 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 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.

The critical decisions in the evaluation of file performance in specific cases are reduced to the four questions below:

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.

**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 are quite 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 3-2 presents a less structured organization, which can be advantageous for heterogeneous data, while Section 3-5 will deal with structured files.

**2. FETCH A RECORD** In order 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 Fig. 3-2.

| 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 | | |

**Figure 3-2** A look-up file

**3. GET THE NEXT RECORD** Isolated data rarely provide 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 are kept together, that is, when the *locality* of these data is strong. Records having locality according to some key are called *clustered*. The records in Fig. 3-1a are clustered by `Student No`.

Since there may be multiple data relationships, but there can only be one sequential access sequence or *physical clustering* through the physical storage. devices in which the data reside, a highly redundant storage design can help; or a liberal use of pointers to provide linkage to successor records can provide assistance. The reading, or writing, of records in a particular order is referred to as *serial* *reading*. If serial reading is simplified by a physical ordering of the records, we can read the file containing these records *sequentially*. Figure 3-1*c* shows the use of redundancy to simplify grouping of records that identify the prerequisite courses.

**4. INSERT A RECORD** Most applications require regular insertion of new data records into their files in order 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 Fig. 3-1*a*, only one record is changed; but in the file of Fig. 3-1*c* three operations are required.

Each write into a blocked file will require the reading of a block in order 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 inserts in some systems. Append operations can be improved if the address of the last block of the file is known, since then the locate step is not needed. This address can be kept as a pointer in the directory. 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 when appending, is $1/Bfr$. For this special case of insertion a separate evalution, based on these factors, could be made. Other considerations, however, argue against batching of appends.

With batching of appends an intervening failure of the computer system will cause multiple appendsto be lost. If these append operations are part of distinct transactions, batching of appends can violate the integrity constraints of a transaction. A *rewrite* of the block being updated for every insertion of a record, or at least for every transaction, is hence best . When single transaction are small, and perform only one append, no benefits are derived from batching.

We will ignore the difference of appends and general insertions in the analyses that follow and assume that a read and a rewrite of a block is done for every type of insert.

**DELETING RECORDS** Deleting records is the inverse of insertion. After locating the record the space that it occupied should be freed and made available for reuse. Maintenance of clustering may require again that other records be shifted.

Often deletion is not performed immediatly, but rather the record to be deleted is updated with an indicator that this record is now invalid. There may be a special field having a marker {`valid, invalid`}, or the old record space may be filled with `NULL` characters, or a message 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 record insertion or record update, as appropriate.

**5. UPDATE A RECORD** All changes of data do not require insertion of a new record. When data within a stored record have to be changed the new, updated record is created using the remaining data from the previous instance of the 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 Fig. 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. The frequency of that event is again $1/Bfr$.

**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 Fig. 3-1*c* to count the number of students or to compute the average credits accumulated by them.

**7. REORGANIZE THE FILE** Finally, it may be necessary to clean up files periodically. Reorganization removes deleted and invalidated records, reclaimes 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 with 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** 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.

Each of the six file-organization methods in this chapter is 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.