¡le
DATABASE DESIGN, Gio Wiederhold.
Updated CHAPTER 3, Part 2, file i3dbd-seq.tex April 4, 1996

Chapter 3 - part 2

Sequential Files

3(2)-2 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. The 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 a easy-to-use form of storage, most information which arrives can be put on a pile. They are quickly created, but costly to search.

171pt Figure space

was Fig.3-1

Figure 3(2)-3 A pile file with a representative record.

3(2)-2-1 Structure and Manipulation

Some restrictions, though, will have to be observed to allow correctness of processing of data. The pile has to be such that we can to extract information. A record most 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, such as `height` in Fig. 3(2)-3, or a code, or a position which indicates its *attribute* type as the records of Fig. 3(2)-1. 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`.

Figure 3(2)-4 Data element

The set of two entries in Fig. 3(2)-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.

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

Figure 3(2)-5 Self-describing data record

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.

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 of 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 specifcation. Successive application of all other search specifications narrows down the possibilities until the desired record, or set of records, is obtained.

ESTIMATION OF SELECTIVITY Using Fig. 3(2)-5, the specification `Tower` applied to a file containing all objects 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 the San Francisco telephone book, combined with the assumption that half the towers are named after people), so that the second search attribute would yield 25 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 determine which tower is desired, or the nonexistence of a tower answering the given request.

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. This frequently is an adequate goal. In the example, the height of the tower still is available as a goal-data element.

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

Figure 3(2)-6 Complex attribute

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 Fig. 3(2)-6.

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 Fig. 3-5 is representative for the file, then a' for that file would be 4. We typically disregard subsidiary attributes as shown in Fig. 3(2)-6 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(2)-2-2 Use of Piles

Pile files are found where data are collected prior to processing, where data are 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 may exist which defy a priori compartmentalization. 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(2)-3 PERFORMANCE OF PILES

We now estimate the 7 performance parameter values for a pile organization as described in Sec. 3(2)-2. We have already some intuition which parameters will be small and which will be large.

3(2)-3-1 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 are 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 Figs. 3(2)-5 and 3(2)-6 above A is 5 and V is 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

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

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¿.

3(2)-3-2 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. The expected average then is the sum of all the times to reach and read any of the blocks, divided by the number of choices, or

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

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,

$$T_F = \frac{1}{2}\, b\, \frac{B}{t'} \qquad 3(2)\text{-}3$$

Figure 3(2)-7 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 $n\,R$:

$$T_F = \frac{1}{2}\, n\, \frac{R}{t'} \qquad 3(2)\text{-}4$$

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.

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 value 1/2 above to 2/3 or 3/4 for two or three requests. For more requests, this factor will approach 1, so that we simply state for the expected cost of executing a batch of L fetch requests

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

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 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.

456pt Figure space

was Fig.3(2)-2

Figure 3(2)-7 Search through a pile.

3(2)-3-3 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

$$T_N = T_F \qquad 3(2)\text{-}6$$

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.

3(2)-3-4 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 will be then

$$T_I = s + r + btt + T_{RW} \qquad 3(2)\text{-}7$$

When disregarding block boundaries and batching of appends, as argued in Sec. 3-0-2, and using Eq. 2-29 we can simplify:

$$T_I = s + 3r + btt \qquad 3(2)\text{-}8$$

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

3(2)-3-5 Update Record in a Pile

Updating of 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

$$T_U = T_F + T_{RW} + T_I \qquad 3(2)\text{-}9$$

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*.

3(2)-5-6 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:

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

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 extremely high cost is avoided by sorting the records according to the attribute of the search argument prior to processing. 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.

SORTING It is well known (i.e., Knuth[73S]) that external sorting can be performed in $O(n \log_2 n)$ steps. When sorting files the $\log_2 n$ term indicates the number of passes over the file taken by the sort; it is generally an integer. Each pass steps sequentially through the files. An initial pass over the file to be sorted arranges the records within each block of the file into sorted order and distributes the blocks for merging. Subsequent passes merge the sorted blocks into sorted sequences of blocks of increasing length until the sequence is of the length of the file. Although the files will be distributed over multiple devices their total size, and hence the time for a pass, remains constant.

A typical sort step involves a sequential read and a sequential write of one record. One pass accesses all records. The time for each pass is estimated using the number of blocks b and the bulk transfer rate t'. The estimate of the time required to perform a sort on an entire file is now

$$T_{sort}(n) = 2b\,btt + 2b\lceil \log_2 b \rceil btt \; = 2n \left\lceil 1 + \log_2 \left(\frac{n}{Bfr} \right) \right\rceil \frac{R}{t'} \qquad 3(2)\text{-}11$$

This estimate will be adequate for most cases. 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.

READ ENTIRE PILE SERIALLY Using a sort to put the file into serial order prior to processing leads to an exhaustive read time of

$$T_X = T_{sort}(n) + T_X(sequential) \qquad \langle\text{serial}\rangle\; 3(2)\text{-}12$$

which will be considerably less than $n\,T_F$ for any nontrivial file.

3(2)-3-7 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

$$T_Y = (n + o)\frac{R}{t'} + (n + o - d)\frac{R}{t'} \qquad 3(2)\text{-}13$$

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

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_{delete} the number deleted from the pile, and v the number of records that were updated by deletion and appending. 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 here and in the 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 sufficient buffers are available to fully utilize the disks. Overlap of disk operations is further considered in Sec. ¡9-4¿. Since reorganization is mostly scheduled to be done at times of low utilization, this condition can frequently be met.

3(2)-4 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. ¡14¿ 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(2)-1.

3(2)-4-1 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. 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 this order now can be performed sequentially. In Figs. 3(2)-1*a* and *b* the `student number` provides such an attribute, but in Fig. 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.

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.

With these structural constraints, and the attendant increased efficiency in tabular-oriented processing, 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.

333pt Figure space

was Fig.3-3

Figure 3(2)-8 A 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 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 are 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.

```
    Program declaration                               Sample      content
  DECLARE 1 payroll_record,                    |
            2 name,                             |
             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);
```

Figure 3(2)-9 Record Declaration

The record layout which will appear on the file in Fig. 3(2)-9 is a direct representation of the DECLARE statement in the program. The declarations also imply fixed element and record lengths.

3(2)-4-2 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(2)-10, 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 are 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 only forward 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 in order to match other sets of files.

To be expanded to 198 pt for Figure space
was Fig.3-4

Figure 3(2)-10 Cosequential files.

3(2)-5 PERFORMANCE OF SEQUENTIAL FILES

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

3(2)-5-1 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 Fig. 3(2)-9 illustrate such an attribute dependency where for the category `sex = F` the next attribute value, `military_rank`, will probably be `NULL`. The fixed record size is the product of the number of fields and their average size.

$$R = aV \qquad 3(2)\text{-}14$$

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. 14. 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.

3(2)-5-2 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(2)-17.

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

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

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 or in the 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

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

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

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

where o is the capacity of the transaction log 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 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 the 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

$$T_F = log_2\left(\frac{n}{Bfr}\right)(s + r + btt + c) + T_{Fo} \qquad \langle\text{binary}\rangle\ 3(2)\text{-}18$$

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.

To be expanded to 365 pt for Figure space
was Fig.3-5

Figure 3(2)-11 Nested binary search in a blocked sequential file.

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 ¡11-5¿). The corresponding block would be numbered $\lfloor 0.2446n/Bfr \rfloor$. There is some uncertainty in the distribution leading to a probe. Since a forward search is best on most devices one can decide, given a 3% uncertainty the probe may actually best begin at the block numbered $\lfloor 0.2146n/Bfr \rfloor$ and then search forwards. Alternate techniques to access records rapidly are used by the indexed and direct file organization methods described in subsequent chapters.

3(2)-5-3 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

$$T_N = \frac{btt}{Bfr} \approx \frac{R}{t'} \qquad 3(2)\text{-}19$$

3(2)-5-4 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

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

The assumptions leading to Eqs. 3(2)-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 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 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(2)-8) and write each record immediately to 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.

$$T_I = s + 3r + btt \quad + \quad \frac{T_Y}{o} \qquad \langle\text{via log}\rangle\ 3(2)\text{-}21$$

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 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, Sec. ¡11-4¿.

3(2)-5-5 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. Then update reads the main file and appends to the log file

$$T_U \approx T_F(\text{main file}) + T_I(\text{log file}) \qquad 3(2)\text{-}22$$

Deletion of a record generates only one entry in the transaction log.

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.

3(2)-5-6 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 for the key used to establish the physical sequence of the main file. The o records placed into transaction log file must first be sorted to establish this sequence. Then both files can be read sequentially. The total cost is

$$T_X = T_{sort}(o) + (n + o)\frac{R}{t'} \qquad 3(2)\text{-}23$$

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 (Knuth[73b]) 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.

3(2)-5-7 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

$$T_Y = T_{sort}(o) + n_{old}\frac{R}{t'} + o\frac{R}{t'} + n_{new}\frac{R}{t'} \qquad 3(2)\text{-}24$$

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

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

The time required to sort a file of size o is given as $T_{sort}(o)$ and can be estimated using Eq. 3(2)-11, provided in Sec. 3(2)-5-6.

3(2)-6 SUMMARY

In this chapter we have introduced a large number of performance oriented concepts, and applied them to two fairly simple file organizations.

Useful concepts encountered are partitioning and selectivity. This concept will be applied later when selecting indexes and becomes 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 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 be used to recover from disasters. Transaction logs can also provide an audit trail when problems with files or their use or misuse have to be investigated.

The two file intial organizations seen, 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 some simple uses.

> There is a tradeoff 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 can be so great that application of constraints is neccessary to make a design useful. It is important to chose the least onerous constraints.

The alternative record structures 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 and the remainder of the information uses named fields.

The method used to derive the performance formulas should now be clear. The formulas are simply derived from the description of the operation being performed.

> $T_{any} = c^* + s^* + r^* + B/t^*$ where we use the $*$ to indicate the repetive nature of the operations being performed.

For one file organization more than access method is feasible, and the best one should be chosen.

BACKGROUND AND REFERENCES

The sequential file designs presented here can easily be traced to early developments in data processing. A pile is, of course, the most obvious data collection. Most of the pieces of paper that cross our desks are self-identifying 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 homogenous 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.

The 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]. Severance[72] considers sequential files and buckets, and the difference between fetch and get-next operations. Shneidermann[76] and Palvia[85] analyze batch fetching in sequential files. Gildersleeve[71] is devoted to sequential files and Nevaleinen[77] has rules for their blocking.

The problem of merging co-sequential files has often been analyzed, a recent example is Levy[82]. In Shneiderman[78] a form of binary search is developed. 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.

More references to specific file methods will be found with Chap. 4. Further information is best obtained from software suppliers (see Appendix B), we do not cite their manuals.

EXERCISES

1 Suggest ways to reduce the redundancy in the dense file of Figure 3(2)-1.

2 What is the difference between writing a block and inserting a record? What is the difference between reading a block and fetching a record?

3 Program the algorithm for a continuous-search described as the topic of *Batching of Requests* in Sec. 3(2)-3-3. 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 Sec. 6-2 for a description of a simulation process.

4 Give the expected insertion time per record inserted into a pile file where $R = 0.7B$ and spanned records are used (Sec. 3(2)-3-3).

5 Determine the time for a batch insertion into a pile (see Sec. 3(2)-3-3). What conditions are required for your result to be valid?

6 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?

7 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.

9 When reorganizing a sequential file, it is possible that a new record on the transaction file is indicated to be deleted by a later transaction. What does this require when sorting the transactions and merging the files? Draw the flowchart for the merge process.

Solutions

1. Problem #4 in Chapter 3

Notice that $1/Bfr$ is an approximation to the probability that an insertion does not fit in the last block. When $Bfr \gg 1$ this consideration is not very relevant since $1/Bfr$ is very small. However, in this case $Bfr = 1/0.7$ or $1/Bfr = 0.7$. So the probability of needing to write an extra block is approx. 70%. Therefore,

$$T_i = (\text{read last block}) + (\text{rewrite last block}) + (\text{write an extra block})$$
$$= (s + r + btt) + 2r + (1/Bfr)btt$$
$$T_i = s + 3r + 1.7btt$$

ANOTHER approach:

Notice that every 10 records you fill 7 blocks exactly. The first insertion will fit at the beginning of the block (Don't need to read that block before writing). 3 out of the other 9 insertions fit in the last partially filled block (Read last block and rewrite it with new record). In the rest 6 cases, record need to be spanned to next block (This case requires read last block, rewrite it, and write the next one). Hence we get:

$$\begin{aligned}
T_i = &(s + r) && \text{\% get to end of file}\\
&+(1/10)btt && \text{\% no block needs to be read}\\
&+(3/10)(2r + btt) && \text{\% insertion fits in last block}\\
&+(6/10)(2r + 2btt) && \text{\% insertion needs to be spanned}
\end{aligned}$$

Simplifying it we get: $T_i = s + 2.8r + 1.6btt$

2. Problem #5 in Chapter 3

Assumptions: (1) No spanned records ($Bfr \gg 1$), and
(2) Batch file is in a different device.

Let N be the number of records to be inserted ($N \gg Bfr$), p be the probability of having to read last block of main file to insert some records there. Then, an approximation to this probability is $p = (1 - 1/Bfr)$. So we get:

$$\begin{aligned}
T_b = &(s + r) && \text{\% to get to the end of file}\\
&+p \times (btt + 2r) && \text{\% to read and write the last block}\\
&+btt' \times \lceil (N - p \times (Bfr/2))/Bfr \rceil && \text{\% to write the succesive needed blocks}
\end{aligned}$$

Note: $btt' = (B + G)/t$

3. Problem #6 in Chapter 3

Probing is a searching method used to estimate the position of a desired record followed by a sequential search. Obviously the search key has to be the *sequence key* for the file.

In this problem, the search argument is the "NAME" attribute. The data to analyze for estimating the entries in the probing table could come from the file itself (if the file is in a static state – no updates). Another possibility is to obtain statistics from a phone book or Table ¡11-4¿. Because some letters have a high frequency of occuring in the first position, using two or three letters in our probing table could be a wise decision. For example, last names starting with "S" are very common, and if we use only 1 probing character, we have to do a lot of sequential search in the data file; so, we'd better partition these situations by using 2 or 3 characters in out probing table.

To decide the appropriate size of the table is very important. We are trading off space in memory to keep the probing table against how close could we get to the desired record. If we start using many characters in the probing table, we are getting closer to an index approach and we don't generally expect to fit the whole index in memory. On the other hand, if the table gets too small we don't get to the neighborhood of the desired record and therefore requires a long sequential search. One important guideline on how big the probing table should be is to avoid seek by hitting the right cylinder.