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

Chapter 3 part 3

Indexed Files

Index-learning turns no student pale,
Yet holds the eel of science by the tail.

Alexander Pope,
The Dunciad, I, *1720.*

3(2)-0 INTRODUCTION

3(2)-1 THE INDEXED-SEQUENTIAL FILE

The indexed-sequential file design attempts to overcome the access problem inherent in the sequential file organization without losing all the benefits and tradition 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 3-6 shows a particular example of an indexed-sequential file. We find in this figure the three important components: the sequential file, the index, and the overflow area. The sketch also has a number of details which will appear in later discussions.

To be expanded to 200 pt for Figure space
was Fig.3-1

Figure 3(2)-1 Components of an Indexed Sequential File ¡¡new¿¿

¡¡shows main area, overflow area, index area¿¿

3(2)-1-1 Structure and Manipulation of Indexed-Sequential Files

This file organization allows, when reading data serially, sequential access to the main record areas of the file, shown in Fig. 3(2)-1 ¡¡new¿¿. Only some pointer fields, not present in the data records as entered, must be skipped.

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 above, 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.

THE INDEX An index consists of a collection of entries, one for each data record, containing the value of a key attribute for that record, and a 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 always 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, here perhaps groups of social security numbers with the same initial three digits.

Table 3-5 Index

We have an employee file sequenced by social security number:

Rec#	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	128-15-3412	Mary	6/31/39	Female	Engineer	...

To find employees we establish an index file as follows:

013-47-1234	1	
028-18-2314	2	
128-15-3412	3	

In practice the subsets are determined by the size of buffers, blocks, and tracks available to the file system. In Fig. 3-6 the subsetting of the index is based on the number of index entries (8) which fit into a block. Successively higher levels of the index become smaller and smaller until there remains only a small, highest level index that can be kept in memory. It will be shown later that there is rarely a need for many levels of indexing.

Indexes have been implemented in a variety of ways. We will consider here a static, multilevel index. A dynamic technique, called a *B-tree*, will be presented with the indexed files of Sec. 3-4. Another alternative to multilevel indexing is to provide for a binary search on the index file. Binary searching was discussed when sequential files were evaluated in the preceding section. Section 3(2)-1-2 presents some simpler index schemes that have been used. We continue here the discussion of this, the most common type of indexed-sequential file.

A PRIMARY INDEX The index for an indexed-sequential file is based on the same key attribute which has been 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, the use of *block anchors*, and the other is keeping most of the index on the same cylinder as the data records, *cylinder indexes*.

Block Anchors The benefit of the index is to rapidly access a file block. 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 an entry 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. 3-6 the choice of anchor point is the first record of a block.

The cost of searching within a file block for a record is minimal, 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 $V + P$.

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.

¡¡UNDERFULL¿¿

If records are often inserted at 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 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.

To be expanded to 450 pt for Figure space
was Fig.3-6

Figure 3-6 Fields of an Indexed-Sequential file.

Cylinder indexes Since much of the overhead of fetching a block is due to the seek time required to reach a particular cylinder, a reduction of seek effort can be made by placing the subset of the index according to hardware boundaries. A two-level index is typical for this method. There will be a master index which contains only key attribute values and addresses of cylinder anchors. On the initial track of each cylinder there will be a cylinder index using tracks, blocks, or records for that particular cylinder as anchors. No seek delay will be incurred between cylinder-index and data records. Example 3-7 will evaluate this type of index.

THE FANOUT RATIO OF AN INDEX 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, $V + P$,

$$y = \left\lfloor \frac{B}{V+P} \right\rfloor \qquad 3(2)\text{-}26$$

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 does this for a general block-anchored index, and Example 3-7 evaluates for the same file the case of a block-anchored index kept on the same cylinder as the referenced data.

Table 3(2)-6 Hardware-Independent Index Design

Given is a block size B of 2000 bytes, a value size V of 14 bytes, a 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 index pointer entries. 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 $20 \cdot 1000 = 20\,000$ 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 *master index* 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 master 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 Example 3-6 for 10^6 index entries. Its master level will have more entries, of course.

Table 3-7 Hardware-Oriented Index Design

Using a disk with 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. Index entries which match sequential hardware units do not need a pointer field, since entry 1 simply corresponds to block 1, etc. 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.

There also will be the second-level index to the cylinder indexes; its size in this organization depends on the number of cylinders required for this file. The data file occupies $\lceil n/Bfr \rceil = 10^6/10 = 10^5$ blocks or $\lceil 10^5/132 \rceil = 758$ cylinders. The index to the cylinders does not require a pointer field either, since it simply has one entry per sequential cylinder. In practice, when using cylinder-resident first-level indexes, the second-level index will reside within each diskpack. Given diskpacks of 200 cylinders, each index to the cylinder for one pack will use $200 \cdot 14 = 2800$ bytes. If block sizes cannot be increased for the second-level index, each pack will use $\lceil 2800/2000 \rceil = 2$ blocks to refer to all its cylinders, and an access to the second level requires two block reads. With two buffers the loss of one revolution can yet be avoided.

The master index for the entire file will have one entry per diskpack, or four entries for this particular file.

We find in these examples that the hardware-oriented indexing structure has a higher fanout ($y_1 = 132$, $y_2 = 200$) and hence a smaller master index ($y_3 = 4$) than a general, symmetric index structure. There is also no seek between the cylinder index and the data. The second-level index is larger and requires additional block reads, costing *btt* for retrieval, and perhaps additional buffer space as well. The actual values will depend on blocking size and file size. The hardware-independent index structure is more flexible to adapt to specific file requirements.

To be expanded to 207 pt for Figure space
was Fig.3-7

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

THE SHAPE OF INDEXES It can be seen that the trees for these index organizations are very broad, rather than high. The term used to measure the breadth of a tree was the *fanout ratio* y. Broad trees have few levels and provide rapid fetch access. The number of indexing levels or *height of an index tree* x is estimated for indexed-sequential file by evaluating the exponential growth

$$x = \lceil \log_y \lceil n/Bfr \rceil \rceil \qquad \text{easily computable as} \qquad \lceil \ln \lceil n/Bfr \rceil / \ln y \rceil \qquad 3(2)\text{-}27$$

¡¡UNDERFULL¿¿

but 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 key attribute V is unusually great; in Sec. 3(2)-2-1 we will describe techniques to keep the effective V small. In Fig. 3-7 we illustrate symbolically a small and a large fanout ratio. The trees are not presented in the 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.

Master Index in Memory During processing another reduction of access cost is common: the master index (level x) remains available in memory once it has been read. The lower levels (levels $x-1$ ... 1) of the index will be used from disk. With this scheme we require a total of x accesses to disk blocks to fetch data records in the main file.

THE OVERFLOW In order to insert records into the file, some free space has to be allocated. We can place insertions to the file in a separate file, or can reserve space in every block for insertions, or can reserve space on every cylinder for inserted records. A separate insertion file requires a separate access with seek and latency overhead at any point where an insertion had been made. We looked at this technique for the simple sequential file and 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. In general this approach requires dynamic space allocation, a file organization with that degree of complexity is described in Sec. 3(2)-3.

Keeping spare space in every cylinder provides a reasonable 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 by `Get Next` operations. In both cases the search process begins from the predecessor record and then follows pointers.

To be expanded to 153 pt for Figure space
was Fig.3-8

Figure 3(2)-8 Insertions of records into a block of an indexed-sequential file.

Pointers to overflow records are placed with the predecessor records in the primary data blocks (see Fig. 3-6). The key of the inserted record is not kept there; only the pointer is put into the sequential file, so that a search for any intervening record is directed to the overflow area. This procedure avoids modification of the index to reflect the insertion but adds one block reading time to every fetch for an overflow record. A request for a nonexistent record will also require going to the overflow file if the search argument follows the key of a primary record with an overflow indication.

Chaining of Overflow Records In order to locate multiple overflows, pointers are placed in the records in the overflow areas. All overflow records starting from one source are linked into a *chain*, possibly through many blocks of the overflow area. A new record is linked into the chain according to its key value, so that sequential order is maintained.

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. This method maintains the key sequence in the blocks of the primary file. New records are inserted after their proper predecessor; successor records are pushed toward the end of the block. Records from the end of the primary block are pushed out into the overflow area.

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.

Processing Overflow Chains Serial processing of the overflow chains requires in general fetching a new block for each record. Half the chain will have to be searched to find a record, and the entire chain must be searched to determine that a requested record does not exist. Maintaining chains in sorted order can reduce the search for non-existing records to about half the chain length. Keeping records in the chains by order of frequency of use can help find existing records faster, on the average. Since recent updates are typically more likely to be needed it can be good to insert new updates into the front of the chain.

Note that in practice overflow chains have quite unequal lengths; for instance, `append` operations create long chains connected to the last block only, unless special mechanisms are used for appends. In Sec. 6-1-5, Example 6-4, we develop estimates of the chain-length Lc for random insertions, leading to Poisson distributions. Statistical approaches to estimating chain lengths become essential if push-through is not used, since then there will be many more smaller chains to 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, and an assessment of insert versus inquiry frequencies is presented in Chap. 5.

To be expanded to 414 pt for Figure space
was Fig.3-9

Figure 3(2)-9 Indexed-sequential file overflow with push-through of records.

REORGANIZATION At or before the point in time 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

1 reading the file in the manner that would be used when doing serial processing, using both the sequential and the overflow areas
2 leaving out all records that are marked deleted
3 writing all new and old remaining records sequentially into the sequential areas of the new file
4 creating and inserting into a memory buffer an index entry with every block placed into the sequential area
5 writing out index blocks when they get full
6 creating and inserting 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 and the old areas will be freed.

During this processing, the reorganization programs will create new indexes based on new anchor point values.

The frequency of such 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 on a periodic basis or be done in the next convenient period after the overflow area exceeds a certain limit of entries. In Sec. 6-5 an algorithm for determining reorganization time will be presented.

3(2)-1-2 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. This is not feasible with the pile and sequential file types.

At the same time sequential access is relatively simple and efficient. Without overflows, after a reorganization, sequential access is practically identical to that of 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 take very long 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 methods are available from computer manufacturers and from 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 of multiple keys is presented in Sec. 3-4, there a different scheme of index management is used and the sequential file is abandoned. Sometimes multiple indexed-sequential files are combined to provide access by more than one key.

3(2)-2 PERFORMANCE OF INDEXED-SEQUENTIAL FILES

The performance evaluation of indexed-sequential files is more complex than the evaluation of the two preceding file organization methods because of the many options possible in the detailed design. We will base the 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. The first-level index, the data it references, and the corresponding overflow area are all kept on the same cylinder. Push-through is used when records are inserted into a block. Records in an overflow area are linked in key order to provide good serial access. Records to be deleted are not actually removed, but only marked as 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 the 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.

3(2)-2-1 Record Size of an Indexed-Sequential File

In the sequential part of the file, a record requires space for a data values and for a possible tombstone. The number of records in the main file is n_m and in the overflow file is o'. Ignoring deleted records $n = n_m + o'$. Initially or after reorganization $n = n_m$. Space is allocated for up to o overflow records. For any actual record

$$R = aV + P \qquad \langle\text{net}\rangle\ 3(2)\text{-}28$$

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

$$i_1 = \frac{n_m}{Bfr} \qquad 3(2)\text{-}29$$

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

$$i_{level} = \left\lceil \frac{i_{level-1}}{y} \right\rceil \qquad 3(2)\text{-}30$$

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

$$bi_{level} = \left\lceil \frac{i_{level}}{y} \right\rceil = i_{level+1}$$

The total size of the index SI is then obtained by summing these up the single block at the root level, $bi_x = 1$,

$$SI = (bi_1 + bi_2 + \ldots + 1)B = (i_2 + i_3 + \ldots + 1)B \qquad 3(2)\text{-}31$$

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

$$R_{total} = \frac{n_m R + oR + SI}{n} \qquad \langle\text{gross}\rangle\ 3(2)\text{-}32$$

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.

To be expanded to 432 pt for Figure space

was Fig.3-10

Figure 3(2)-10 Fetch a record in an indexed-sequential file with push-through overflow.

3(2)-2-2 Fetch Record in an Indexed-Sequential File

To locate a specific record, the index will be used. The master index will not require a disk access, since it will have been placed into memory when the file was opened. We will also assume that the remaining index levels reside on the same cylinder as the data. The primary fetch process will consist of a look-up in the master table, a seek to the cylinder, a read for each remaining level of the index, and a read of the data block, requiring

$$T_{F\,main} = c + s + (x-1)(r + btt) + r + btt \qquad \langle\text{primary}\rangle\ 3(2)\text{-}33$$

However, if insertions have occurred, the procedure may fail to find a record because it has been pushed into the overflow file area, and the search will have to continue to the overflow area. 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. For a fetch where any record is equally probable the overflow probability per stored record is

$$Pov = o'/n \qquad 3(2)\text{-}34$$

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

$$Lc = \frac{o'}{n_m}\,Bfr \qquad 3(2)\text{-}35$$

We do not wish the chains to grow too long, since 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 but on the same cylinder. Per record of the file the cost of searching the first overflow records is

$$T_{F\,overflow} = Pov\,(s^{\dagger} + 1/over2Lc + 1(r + btt)) \qquad 3(2)\text{-}36$$

the number of ¡¡¿¿ the additional records accessed to locate the desired

Without considering locality this is equal to the number of additional blocks to be accessed at a cost of $r + btt$ each.

This operation is done only if any record is fetched from the overflow (*Pov*)

$$T_{Fchain} = Pov\,\frac{Lc-1}{2}\,(r+btt) = \frac{1}{2}(Bfr\,Pov^2 - Pov)\,(r+btt) \qquad 3(2)\text{-}37$$

This factor, although including the squared term Pov^2, may remain significant, since the derivation assumed high values of Bfr.

An adequate estimate for record fetch in common indexed sequential file configurations and usage is obtained by combining Eqs. 3(2)-33, 3(2)-36, and 3(2)-37 to give

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

If overflow areas are placed on the same cylinder, the $s^{\dagger}$ term due to Eq. 3-36 drops out, and if $n/o' \gg Bfr$ the term due to Eq. 3-37 drops out. We are left with

$$T_F = c + s + \left(x + \frac{o'}{2(n+o')}\right)(r+btt) \qquad \langle\text{simple}\rangle\ 3(2)\text{-}39$$

ESTIMATION OF FETCH TIME We use the same file size presented in Examples 3-6 and 3-7, 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 $o' = 0.20 \cdot 0.4\,n = 0.08\,n$. Using the relations above, and a Bfr of 10,

$$Pov = \frac{0.08}{1.08} = 0.0741 \qquad \text{and} \qquad Lc = 10 \cdot 0.0741 = 0.741$$

With these assumptions we arrive at an average fetch time of

$$T_F = c + 1.0741s + 3.0645(r+btt)$$

Nonuniform insertion distributions can increase T_F considerably.

3(2)-2-3 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. A number of possibilities exist depending on the locations of the predecessor and of the successor record. The six cases distinguished are illustrated using the record numbers of Fig. 3-9. The process is flowcharted in Fig. 3-11.

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.

¡¡UNDERFULL¿¿

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 β.3 and $(\beta+1)$.1, 2β.3 and $(2\beta+1)$.1, once for each cylinder occupied by the file.

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).

To be expanded to 423 pt for Figure space
Get the successor record in an indexed-sequential file.
Note that the flow in this figure overlaps Fig. 3-10 to a great extent.

Figure 3(2)-11 was Fig.3-11

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 $= 1 - 1/Lc$

To be expanded to 216 pt for Figure space
was Fig.3-12

Figure 3(2)-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

$$
\begin{aligned}
T_N = \;& (Pd)(Pb)(c)+ && /* \mathit{case}\ a */\\
& (Pd)(1-Pb)(Pm)(Pc)(r+btt)+ && /* \mathit{case}\ b */\\
& (Pd)(1-Pb)(Pm)(1-Pc)(s+r+btt)+ && /* \mathit{case}\ c */\\
& (Pd)(1-Pb)(1-Pm)(s^{\dagger}+r+btt)+ && /* \mathit{case}\ d */\\
& (1-Pd)(Pl)(r+btt)+ && /* \mathit{case}\ e */\\
& (1-Pd)(1-Pl)(s^{\dagger}+r+btt) && /* \mathit{case}\ f */
\end{aligned}
\qquad 3(2)\text{-}40
$$

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 Eq. 3-34 and 3-35.

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 ($s^{\dagger} = 0$, cases b and d, as well as e and f combine), then

$$T_N \approx \left(\frac{1 - Pov}{Bfr} + Pov\right)(r + btt) = \frac{n + o'\,Bfr}{(n + o')Bfr}(r + btt) \qquad 3(2)\text{-}41$$

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

¡¡¡¡¡¡¡¡¡¡¡¡¡¡

3(2)-2-4 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

$$T_I = T_F + T_{RW} + r + btt + T_{RW} = T_F + 5r + btt \qquad 3(2)\text{-}42$$

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

3(2)-2-5 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}

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

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

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

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.

3(2)-2-6 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

$$T_X = T_F + (n + o' - 1)T_N \approx (n + o')T_N = \frac{n + o'\,Bfr}{Bfr}(r + btt) \qquad \langle\text{serial}\rangle\ 3(2)\text{-}45$$

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

$$T_X \approx (n + o')\frac{R}{t'} \qquad \langle\text{sequential}\rangle\ 3(2)\text{-}46$$

3(2)-2-7 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. Two additional buffers in memory are needed to collect the new data and index information. These outputs can be written sequentially. Then

$$T_Y = \frac{n + o'\,Bfr}{Bfr}(r + btt) + (n + o' - d)\frac{R}{t'} + \frac{SI}{t'} \qquad 3(2)\text{-}47$$

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. 3-34 to 3-45. 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 Sec. 3-3-1, we may assume $o' = 0.8\,o$. Such a value would be 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 where 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(2)-4 THE INDEXED FILE

When there is no requirement for sequentiality to provide efficient serial access, an *indexed file* can be used. In an indexed file the records are accessed only through one or more indexes. Now there is no restriction on the placement of a data record, as long as a pointer exists in some index that allows the record to be fetched when the goal data from the record are wanted. One index will be associated with one key attribute, and indexes can exist for all attributes for which a search argument can be expected.

With the loss of sequentiality relative to a primary key attribute, a gain in flexibility is obtained which makes this organization preferable to the indexed-sequential file organization in many applications. The actual physical placement of records in indexed files can be determined by secondary considerations, as ease of management or reliability. The availability of indexes on more than one attribute greatly increases the potential for use of the data in information retrieval and advanced processing systems. Variable-length records are common in these applications.

The flexibility of indexed files has created a great variety of actual designs. We will evaluate again a specific approach which is becoming increasingly common, based on the use of *B-trees* for the indexes. 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.

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.

To be expanded to 342 pt for Figure space
was Fig.3-13

Figure 3(2)-13 Record linkage in an indexed file.

DIRECTORIES Since a multiply indexed file consists of multiple distinct portions there is a need to be able to determine what indexes exist, how they are organized, and where they are located. An extension of the notion of file directory, introduced in Sec. 3-0-1 can satisfy that role. Programs which retrieve data or update the file must consult that directory in order to carry out their function. In database management systems we will find that role taken on by *schemas*.

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 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.

One older use of inversion is the creation of a single index to all the words in a body of writing. The 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. If the pointers are augmented by a sample of the text, the result assumes the form of a *concordance*. The language used for the text is, of course, not restricted to English. 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. The term *indexing* in a bibliographic context is the selection or creation of significant attribute values for subsequent retrieval. These may be placed with the text sections so that there is now a second attribute type.

The terms inverted index, inverted list, inverted file, and partially inverted file are also used inconsistently in the literature, and frequently imply indexing as described in the remainder of this book.

To be expanded to 361 pt for Figure space
was Fig.3-14

Figure 3(2)-14 Sample of a concordance of the Chanson de Roland.

3(2)-3-1 Structure and Manipulation of Indexed Files

There may be as many indexes as there are attribute types in the file. An index for an attribute of an indexed file consists of a set of entries, one for every record in the file. The technique of using block-anchor points to reduce index size, which was used in the indexed-sequential file, is not available here, 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 pointer to the record. 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 require multiple levels, as 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 pointers 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.

A directory of attributes is needed to locate the index corresponding to an attribute name. We will discuss such facilities in Chap. 8, since these directories serve more than just the needs of the file systems. 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. Since many indexes may be required, we do not expect to find master indexes in memory.

EXHAUSTIVE AND SELECTIVE INDEXES Indexes may be *exhaustive*, that is, have pointers to every record in the file; or *selective*, that is, have only pointers 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 partitioning effectiveness.

An example of a selective index could occur in a personnel file which includes health data. Only indexes to current or chronic diseases are maintained, although a more complete record is kept for statistical or individual purposes in the data file itself. 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 selective index for a health attribute could point to all individuals with a cholesterol_level greater than 250, if frequent reference to this fact is made, for instance, in meal preparation.

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.

MAINTENANCE OF INDEXES The major problem when using indexed file is caused by the need to update all the indexes that refer to a record whenever a record has been added or deleted or is moved. Single indexes have to be changed when a single data field value of an indexed attribute has been 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.

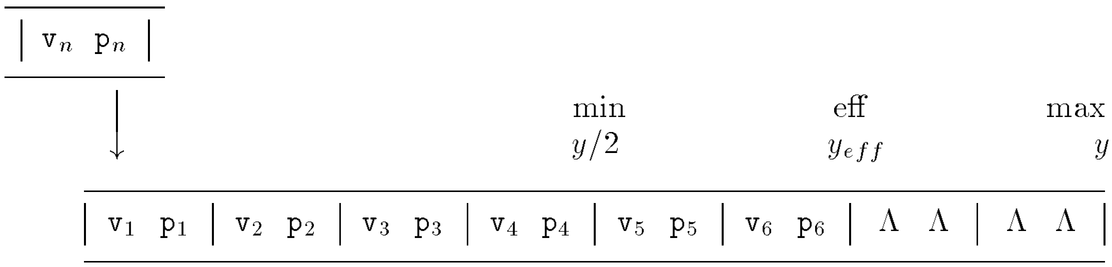
AN INDEX STRUCTURE FOR DYNAMIC UPDATING – B-TREES In indexed files every change of an indexed attribute will require the insertion, deletion, or both, of an index entry into the appropriate index block. To make such changes feasible, we reduce the density of the index. Extra space is left initially empty 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 full is another block obtained. Half of 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 in order 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.

The original B-tree was defined with space for y pointers and $y-1$ values, and kept at least $y/2+1$ pointers in a block. The value corresponding to the first pointer ($\mathtt{v}_1$) in a block is the value ($\mathtt{v}_n$) in the entry associated with this block in the next-higher-level index block. In the index-file approach presented here, we keep this value redundantly. This simplifies the formatting of entries, permits index blocks to be accessed independently of their ancestor blocks, and has little effect on the analyses when y is reasonably large.

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

$$y = y_{max} = 8,\ y_{initial} = y/2 = 4,\ y_{eff} = 6$$



To be expanded to 3 pt for Figure space
was Fig.3-15

Figure 3(2)-15 Block of a B-tree index after two insertions.

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 will ignore the effect due to this one 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 $y_{eff}/y \rightarrow \ln 2 = 0.69$. Then

$$y/2 \leq y_{eff} \leq y \qquad \text{or} \qquad y_{eff} \rightarrow \ln 2 \cdot y = 0.69\,y \qquad 3(2)\text{-}48$$

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.

When an entry is deleted, an index block may be left with less 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 height of the tree is reduced by one. 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.

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.

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.

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 valuess, we find, similarly to Eq. 3-27,

$$x = \left\lceil \log_{y_{eff}} n' \right\rceil \qquad 3(2)\text{-}49$$

The search process is flowcharted in Fig. 3-16. 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.

To be expanded to 378 pt for Figure space
was Fig.3-16

Figure 3(2)-16 Fetch using one index of a multi-indexed file.

DEFERRED UPDATING OF INDEXES The algorithms to maintain dynamic indexed files do generate many more block accesses and rewrites than those used for indexed-sequential files. It can be impractical to attempt to change all indexes immediately after a record has been changed. High-priority fetch requests using the file may need to be processed before all indexes are brought up to date.

The continuous maintenance of indexes can also be quite time-consuming. A batch of updates may have to rewrite the same index block again and again. Hence we find systems where indexes are created from a file for a particular analysis but are not kept up to date when the file changes. Sometimes indexes are updated only periodically, which causes recent information not to be accessible. This approach is akin to a periodic file reorganization.

DEFERRED UPDATING The updating of the indexes may be done at times when the computer is relatively idle. If such updating can be performed incrementally, a periodic wholesale reorganization can be avoided so that the file is kept available, although retrievals may not include recent changes. Such index updating will be done by a process which is kept active at a low priority until it is completed. It is necessary to keep the old record in place until assurance exists that all referring indexes have been updated.

Updating 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 Chaps. 11 and 13. Deferred updating of derived data is presented in Sec. 8-3-4.

When index update is deferred, a retrieval may not find recently inserted data but may also retrieve data that has been previously deleted or changed. A new record will not be retrieved by a request based on attribute values until the relevant index has been updated. An index that is out of date will also include references to recently obsoleted data. This problem can be avoided by marking the old record with a tombstone. If the record has been moved, the marker can also provide an indication of the new location.

Many applications can operate with deferred updating. When queries involve management decisions for planning, say, analysis of a `sales` pattern, use of data which is at most 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 index update may have to be forced to assure the stock counts are up to date. 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.

3(2)-3-2 Use of Indexed Files

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

When an item of information is obtained, e.g., a free 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.

Multi-indexing of 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 are highly variable and dynamic. The flexibility of the record format and record placement available with indexed files does not exist in other file systems. Much of Chap. 4 is devoted to specific alternatives and examples of indexed and related files.

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

3(2)-4 PERFORMANCE OF INDEXED FILES

The performance of indexed files is easier to evaluate than the performance of the preceding indexed-sequential file designs. B-trees have a very predictable behavior since all single indexing 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. We have noted that attributes that have low partitioning effectiveness are poor candidates for indexing. Not having indexes for these attributes, or the use of selective indexes, will reduce the size of the index space and accelerate updating, but will not affect the other performance factors, as long as no searches try to use 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 are then 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 $dens = y_{eff}/y = 0.69$. 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 $1/0.69 = 1.45$ of the net requirement. Note also that the height of the index tree may be greater than the height of a denser index.

3(2)-3(2)-1 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

$$n' = n\,\frac{a'}{a} \qquad \text{3(2)-50}$$

index entries referring to data records, each entry is of size $V_{index} + P$ and the size of one index is estimated as

$$SI_1 = n'\,\frac{(V_{index} + P)}{dens} = 1.45\,\frac{n\,a'}{a}(V_{index} + P) \qquad \langle\text{one index}\rangle\ \text{3(2)-51}$$

Higher-level index blocks are ignored here; for instance, the second level would add only $SI_2 \approx SI_1/y_{eff}$ bytes. The total space needed for all indexes becomes then

$$SI_{total} = a\sum_{i=1}^{x} SI_i \approx 1.45\,n\,a'(V_{index} + P) \qquad \text{3(2)-52}$$

The space requirement per record is equal to the sum of data and index space. No overflow area exists. We will again consider only the lowest level index entries.

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

$$R_{total} = R_{main} + a'\, R_{index} \\ = a'(A + V + 2) + 1.45\, a'(V_{index} + P) \qquad 3(2)\text{-}53$$

for index and data.

The value field in an index V_{index} is often not of 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, the attribute category is `profession`, so that we have many entries in this index for an attribute value like `welder`), then one value entry may serve many pointers. Techniques discussed in Sec. 3(2)-2-2 can reduce key value sizes of indexes further, so that often $V_{index} < V$. If V_{index} can be assumed to be equal to V, then

$$R = a'(A + 2.45V + 1.45P + 2) \qquad \text{if} \qquad V_{index} = V \qquad 3(2)\text{-}54$$

based on Eqs. 3-48 and 3-53.

Additional index blocks will be used for index levels above 1. In most designs, that additional number amounts to only a small percentage of the storage space, as will be shown in Example 3-8.

Table 3(2)-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 pointer 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.

$$n' = n\, a'/a = 20\,000 \cdot 2.5 = 50\,000$$
$$b_1 = n'/(0.69y) = 50\,000/69 = 725$$
$$b_2 = b_1/(0.69y) = 725/69 = 11$$
$$b_3 = b_x = b_2/(0.69y) = 11/69 = 1$$

Estimating x directly, we obtain the same result:

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

An estimate for the space was given by Eq. 3-51, $SI_{skill} = 1.45n\, a'/a(V + P) = 725\,000$ 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 higher-level indexes.

3(2)-3(2)-2 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. Since there will be, in general, several index subfiles, we can no longer assume that these indexes and the data will both occupy the same cylinder. 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

$$T_F = x(s + r + btt) + s + r + btt = (x + 1)(s + r + btt) \qquad 3(2)\text{-}55$$

where x is given by Eq. 3-49.

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

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

3(2)-3(2)-3 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.

$$T_N = s + r + btt \qquad 3(2)\text{-}57$$

This holds as long as the effective fanout ratio $y_{eff} \gg 1$, since the probability that the current index block contains also the pointer to 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.

3(2)-3(2)-4 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

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

Keeping indexes within a cylinder can reduce the time by requiring only one seek per index. If all indexes are small so that $x = 2$, further ignoring the cost of splitting $(y \gg 1)$ and using $T_{RW} = 2r$ gives a less forbidding estimate

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

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. Presplitting reduces the likelihood of multiple splits in one index. With deferred updating only one index access and the actual record rewriting is noted by the user.

3(2)-3(2)-5 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. 3-58. 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 pointer 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 pointer requires finding the blocks and rewriting them; no splits will occur here; this term is $T_{fixpointer}$. We collect all the costs.

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

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

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

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

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

SMALL VALUE CHANGES When an update changes an attribute value by a small amount, further savings will occur if both old and new index values fall into the same index blocks. In that case one search and rewrite may be saved. The ratio between out-of-index block and within-index-block changes is difficult to predict and depends on the behavior of the attribute change. 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 Pi to be the probability of out-of-index-block updates, we find for a single update, given all the conditions leading to Eq. 3-62,

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

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

$$b_1 = 1.45\, \frac{n}{y}\, \frac{a'}{a} \qquad 3(2)\text{-}64$$

primary index blocks for one attribute; then the probability, given random data changes, of requiring another index block is $Pi = (b_1 - 1)/b_1$. For a specific attribute the specific values of V and a'/a should be used, so that y and b_1 are appropriate. For large files and frequently occurring attributes Pi will be close to 1, so that Eqs. 3-60 to 3-62 remain appropriate.

However, Eq. 3-62 is of interest if Pi can be shown to be low because of the behavior of data value changes for updated attributes. A number of important cases exist where the database monitors changes in the operation of some physical process. Changes in `pressure`, `temperature`, or `flow` occur mainly gradually, and the data may be acquired at high rates, so that file performance could be critical.

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

The uniform behavior assumed for insertion and update patterns represents the best behavior in terms of index growth over the long term. Updating tends to be periodically heavy in one area, and then concentrate again on other attributes. These irregularities can specifically affect system behavior. The analysis of these conditions requires statistics of update behavior which we do not have available here, and we will not evaluate their effect. 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, using the concepts discussed above.

3(2)-3(2)-6 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

$$T_X = T_F + (n-1)T_N \qquad 3(2)\text{-}65$$

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 equal to the time needed to read a sequential file. 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,

$$T_X = \frac{n}{Bfr}\left(s + r + \frac{B}{t'}\right) \qquad \langle\text{by blocks}\rangle\ 3(2)\text{-}66$$

In this approach 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.

3(2)-3(2)-7 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 may be required for the data file in order to recover unusable fragments of space 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. The largest, lowest level of an index will have b_1 blocks as given in Eq. 3-64. The number of higher-level blocks is much less, as was shown in Example 3-8.

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.

$$T_{Y\,i} = T_X + T_{sort}(n') + \frac{SI}{t'} \qquad \langle\text{one index}\rangle\ 3(2)\text{-}67$$

Estimates for these terms were given as Eqs. 3-66, 3-11, 3-50, and 3-51. 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 pointers 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 the indexes. The time requirement then is

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

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. Some assumptions made in Eq. 3-11 about sequentiality will then not hold.

¡¡UNDERFULL¿¿

BACKGROUND AND REFERENCES

In offices large sequential files in drawers are partioned using folders. Labels on the drawers provide a 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 IBM3330 disk design was greatly influenced by their behavior.

The construction of index trees has been frequently discussed in the computer science literature. It is important to realize when using this literature that the index trees found in storage file have very high fanout ratios and do not contain the data within themselves. Trees containing data are discussed in Sec. 3(2)-4. 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.

The B-tree algorithm was presented and evaluated by Bayer[72O,S]. Comer[79] provides an excellent summary and Knuth[73S] includes some file-oriented considerations. Yao[78] analyzed the density of B-trees under random insertions and deletions. Yamamoto[79] and Rosenberg[81] consider strategies for optimal behavior. Aho[79U] evalutes their use in relational databases. Gosh[69] combines B-trees with probing. The technique of pre-splitting is described in Guibas[78].

EXERCISES

10 Choose three out of the attributes shown for indexing a personnel file:
`Name, age, sex, job, number of dependents, salary of dependents, salary`
and explain why you chose them.

solution

4. Problem #10 in Chapter 3

The most important factors to choose an attribute to index a file are the frequency of its use and its partitioning effect.

The *NAME* attribute should be your primary choice as a key to access the personnel file described in the problem. *NAME* is a natural way to access the file and its partitioning is very effective (name is "unique"). Other attributes that are used as a key in a query and may have a relatively good partitioning effectiveness are *JOB* and *SALARY*. The *JOB* attribute provides a way to find employees with a particular skill. To allocate them in the departments, etc. Indexing the *SALARY* attribute might help in budgeting, salary updating, and managerial decisions.

11 Flowchart and compare the performance of a binary search through an index with the multilevel approach for indexed files.

12 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.

21 Calculate the total space required for a file as shown in Fig. 3-39, using record-anchored indexes.

To be expanded to 270 pt for Figure space
was Fig.5-39

Figure 3(2)-39 Illustration of indexed sequential file layout.

Consider a design where the primary index level is placed on the same cylinder as the data, which will reduce the number of seeks needed to access the data. The calculation will initially evaluate the capacity of one cylinder and then determine the number of cylinders required. hereninepoint Parameters to be used:

$R = 200$ (including the linkage pointer of 6 bytes length),
$B = 2000$,
$V = 14, P = 6$,

blocks per track = 7, surfaces per cylinder $k = 19$, and overflow allocation = 20%.

6. 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.

5. The solution is straightforward from Sec.3-3 formulas. You only need to recall that a seek is avoided when overflow occurs by placing the overflow areas in the same cylinder.

$$T_f = T_f(main) + T_f(overflow) + T_f(chain)$$
$$T_f(main) = c + s + x \times (r + btt)$$

Since $o = 0.25n$ and reorganizations are made when overflow is 80% full, $o' = 0.25n \times (0.8/2) = 0.1n$, leading to $Pov = (0.1n)/(n + 0.1n) = 0.09$. Then,

$$T_f(overflow) = 0.09(r + btt) \qquad \text{\% overflow seek time is 0!}$$

Using $Bfr = 20$ and Pov, we compute

$$T_f(chain) = Pov \times ((Lc - 1)/2) \times (r + btt)$$
$$= 0.036 \times (r + btt)$$

So we get: $T_f = c + s + (x + 0.126) \times (r + btt)$