## II. PRIMARY FILE ORGANIZATIONS

**Goal:** To choose a file organization with space efficiency and high performance.

(How we store or organize info. ) (How we process or access info. )

| File Org. | File Access |
|---|---|
| sequential | sequential |
| indexed sequential | sequential & direct |
| Direct | Direct (Random) |



## File Access Types

**Sequential Access :** accessing multiple records, often the entire file, acc. to a predefined order.

**Direct (Random) Access :** locating a single record.

How can we have an effective organization ?
- Org. should be matched with the type of intended access.
- Use & structure of info should match



## Sequential File Organization

**Record:** | $field_1$ | $field_2$ | ..... | $field_n$ |

Ex: An employee record :

| Emp-name | number | title | dept | mgr | salary | .... |
|---|---|---|---|---|---|---|

**Primary Key :** is a field, or a combination of several fields which uniquely distinguishes a record from all others.

**Secondary Keys or Attributes :** All remaining fields.



A file consists of recs. of the same format.
- Fixed-length recs: all recs within a file have the same length. *Example?*
- Variable-length recs: all recs. within a file do not need to be the same length. *Example?*

**Sequential File Organization:**

The (i+1)'th element of a file is stored contiguous to and immediately after the (i)'th element.

| 1 | 2 | ... | i | i+1 | ... | n-1 | n |
|---|---|---|---|---|---|---|---|



**Sequential Access:** a simple process to move from one record in the file to the next by incrementing the address of the current record by the record size.
- Compare this to accessing a sequential array.

**Direct Access :** If we know the subscript, we can process a single record directly.

**Sequential Search:** processes the records of a file in their order of occurrence until:
- it either locates the desired record
- or processes all the records

We assume that the subscript isn't the primary id of a record.



## Direct Access on a Sequential File

- not efficient

**Probe** is an access to a distinct location

- In a file with **R** recs,
  - **R/2** probes are needed to locate the record of interest
  - **R** probes are needed if record is not part of the file.
- Performance:
  - For Large **R**, unacceptable performance.
  - For small **R**, simple & fast.

## Use of Sorting to Improve Retrieval Performance

- Sorting the records gives us a linear ordering based upon the key values of records:

  $key_1 < key_2 < ..... < key_i < ... key_n$

- For a sorted file:
  - Average cost of searching for the desired record is **R/2**
    - for either successful or unsuccessful searches.



## Assumptions & Notation

- Files are stored in a small number of contiguous extents.
- Single-user (single-process) environment.
- Fixed-length records
- One (or sometimes two) disk drives.
- 10 MBs of main memory

$\mathbf{T_F}$ = time to fetch (find & read) one record.
$\mathbf{T_N}$ = time to fetch next record in order.
$\mathbf{T_I}$ = time to insert a new record.
$\mathbf{T_U}$ = time to update by modifying values.
$\mathbf{T_D}$ = time to delete a record.
$\mathbf{T_X}$ = time for exhaustive reading of a file.
$\mathbf{T_Y}$ = time for reorganization of a file.



## The File Header

Every file has a **header** that contains some descriptive info about the file:

- number and length of recs.
- address of last block.
- whether file is an ASCII or binary file.
- number of bytes in file.
- whether file has an index.
- addresses of all extents.

- When the file is opened, this header is brought into memory and becomes available.
- When the file is closed, it is updated in disk if necessary.
- Header read/write has zero cost.



## The Pile File

A pile file is a succession of records simply placed one after another with no additional structure.

- We assume fixed-size records.
- Even if a file has an index, it may still have to be regarded as a pile file when searching on the basis of a different field. (i.e., ?)



## Fetching One Record

Given the value of some field in a record, e.g.,last name, we must search sequentially through whole file and compare values until:

- we either find a record,
- or we reach end of file.

[We can't read less than one block at a time.]

Average number of blocks read = $(1/b)*\Sigma^{b}_{i=1}\, i \cong b/2$

$T_F=(b/2)*ebt$



## The Hospital File Example:

Consider a hospital with n=100,000 patient records of

- R=400 bytes each (≈ 40 MB file)
- B=2400 bytes (block size)

Then b= (100000*400)/2400 = 16667 (number of blocks)

$T_F = (b/2)*ebt$

= 8333 * 0.84 = 7000ms =7sec. (w/o an index)

If we need to look up 10,000 names,
total time required $7x10^4$=19 hours

[If so, then it is better to organize file in another way!]

## Fetching the Next Record

$T_N = T_F$

Why?[Because records in a pile file is not stored in order.]

What is the difference between a list of consecutive values for some field and an index?

- An index has the addresses of the records in question.

### Exhaustive Reading of the File

$T_x$(beginning to end) = $b*ebt = 2*T_F$



## Finding Averages and Sums

All recs must be read but not in any particular order. [Best to use a pile file]

**i :** count of the number of records read so far

**A:** average of the value for records up to this point.

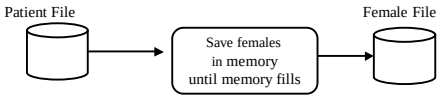
New average = $A*(i/(i+1)) + (\text{new data}/(i+1))$

[We use double buffering.]



## Creating a File of Female Patient Records

Find all records satisfying some criterion by reading through all records (Best to use a pile file!)




Total time = 21 secs.

(14 secs to read whole file + 7 secs to write female patient recs.)

[Switching from R to W mode requires a seek but it is insignificant]



## Average Length of Stay in a Hospital by Medical Condition

Calculating some average w.r.t. some grouping (same as average calculation.) [Best to use a pile file!]

- We have 200 diff. medical conds.: 1..200
- Two 200-element arrays of integers are kept in memory
  - running_average for each medical condition
  - running_count of occurences of each condition

Ex: Medical cond =43, length of stay=4, Prev avg=2.50, Prev count=29

Total Time = 2.50*(29/30)+(4/30)=2.55

With double buffering, Total time = $T_x$

[read thru file one to obtain table of results in memory]



## Average Length of stay in a Hospital by Medical Condition

Calculating some average wrt. some grouping. (same as average ex.)

[Best to use a pile file]

200 different medical cond. : 1 ... 200

Two 200 element arrays of ints in menu:

- running average for each medical cond.
- Running count of occurences of each cond.

With double buff, Total time= $T_X$

(read thru file once to obtain table of results in mem)

Ex: medical cond = 43 Length of stay =4

prev av. = 2.50 prev count = 29

2.50*(29/30)+(4/30) =2.55



## Exhaustive Reading in Order of a Field

We go through a list and look up one record for each entry on the list. Each record need to be fetched separately:

$T_x$(independent fetches) = $n * T_F$

= $n * (b/2)*ebt$

= $n * [(n/Bfr)/2]*ebt$

### Examples

Example 1: Cambridge Property Tax Records

n=30000 records, R=100 bytes

For 30000 independent fetches:

$T_x$ = 30000 * [(30000/24)/2] * 0.84

= 15750000 ms = 15750 sec = 4.28 hours

## Examples

Example 2: The Hospital File

n=100000 records of R=400 bytes

$T_X$ = 100000* (7000 ms) = $7*10^9$ = 194 hours ≅ 8days

Example 3: California State Income Tax

n= 6 million personal income tax records.

R=400 bytes each

$T_X$ = 6000000* [(6000000/6)/2]*0.84 = $2.52*10^{12}$ ms

= $2.52*10^6$sec = 700000 hours = 80 years

It would be more efficient first to sort the file and then read it in order. [1.5 hours sorting time would be required]



## Inserting a New Record

To insert a new record, we place it at the end of the file. Why?

Assumptions:

- Duplicate records are allowed. No check is necessary.
- Address of last record is kept in the file header in memory.

We read in the last block, and append new record:

$$T_I = s + r + btt + 2r$$
$$= \underbrace{s + r + btt}_{\text{read last rec}} + \underbrace{2r - btt}_{\text{rotate back}} + \underbrace{btt}_{\text{write last rec}}$$

What if the last block is full ?

- Last block is the address of the next empty block if the last one is completely full.
- Any optimizations? Keep the last block in memory.



## Deleting a Record

1- Fetch the record

2- Modify the contents by placing a tombstone in the record to denote its deletion

3- Write back the modified record ♦ See Fig 3.2 for an example.

$$T_D = T_F + \underbrace{2r - btt}_{\text{rotate back}} + \underbrace{btt}_{\text{write back}} = T_F + 2r$$

$2r << T_F$ ($T_F$=7sec, 2r=166 ms)

So $T_D \cong T_F$



## Modifying a Record

- Assuming fixed-length records (Why?)

  $T_U$(Fixed length)= $T_F$ + 2r (same as $T_D$)

- Variable-length records.
  - Modified record may be longer and may not fit in the same place as the old version.
  - Treat update as a combination of delete & insert.

$$T_U(\text{Variable length}) = T_D + T_I$$
$$= \underbrace{T_F}_{7000} + \underbrace{2r}_{16.6} + \underbrace{T_I}_{42}$$
$$@T_F$$

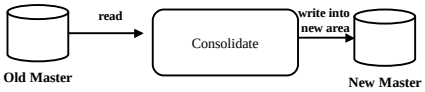


## Reorganizing

After many insertions & deletions, file organization deteriorates:

- Time for finding a block grows proportionally to the # of blocks.
- Time for sequential reading grows since more seeks will be involved
  - File has been spread over the disk.

♦ See Fig 3.3 for an example.






Too much seeks ?

- Read in as many of the nondeleted records as possible (use double buffering)
- When memory fills, write reorganized blocks into disk

Time to read in old file = b * ebt

Time to write out new file = (n/Bfr)*ebt

$$T_Y = (b + (n/Bfr)) * ebt$$

(Extra time required for seeks, rotations etc. is negligible).

## Reorganizing with Two Disk Drives

1- Read from old file on disk1

2- Simultaneously write new records into disk2

3- Input speed & output speed are not the same

- We can't use plain double buffer w/blocks, why?
- Buffers for slower process use track-sized buffers.
- When a track fills up for output, it is written to disk:
  - rotational latency for writing out is minimized
  - time for reading in dominates
  - time for writing out is overlapped by reading in.

$$T_Y = b * ebt$$

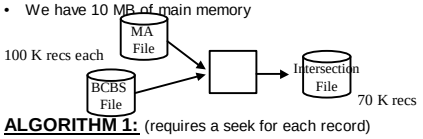


## Extended Example:An Intersection File

An **intersection file** contains all the records common to given files.

Two unordered hospital files with:

- n=100,000 records of R=400 bytes each.
- One for MA residents, one for BCBS insurance
- 70,000 of these records are in both files
- We have 10 MB of main memory




**ALGORITHM 1:** (requires a seek for each record)

1- Read in one record from MA file

2- Compare that record with all records in BCBS file

3- If in both files, write into new file



**ALGORITHM 2:** (Unnecessary seeks are eliminated)

1- Read as much of MA file into memory first

2- Read in 1/4 of (10 ms) MA file

3- Read through whole BCBS file

4- Compare each record in memory with all BCBS records (70% of the time it will be there, 30% ?)

5- Write out 7 MB to the intersection area

6- Back to step 1.

**Analysis:**

- Time for reading in 40 MB of data in 4 segments (i.e., 4 seeks & 4 rotational latencies plus time for reading 40 MB)

  $\underbrace{4*(s+r)}_{97msec} + \underbrace{\ldots}_{14\ msec}$

- Time for writing out intersection file is 70% of 14 sec. = 10 sec.



## Time Spent for Comparisons

For each record in MA file, we search through BCBS file:

- For 70 K records, there is a duplicate

  $T_F$ = (b/2)*ebt = 7 secs.

  $T_X$(independent fetches) = n * $T_{F(average\ case)}$ = 70000*7

- For 30K records,no duplicate(i.e.,whole file needs to be read)

  $T_X$(independent fetches) = n * $T_{F(worst\ case)}$ = 30000*14

Total Time = 70000*7 + 30000*14 = 910,000sec=15.166min

= 253 hours ≅10.5 days

**Result:** It is not a good idea to make intersection files from pile files w/o sorting first.