GDAŃSK UNIVERSITY OF TECHNOLOGY

ETI

Database Structures

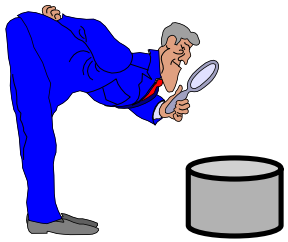
Krzysztof Goczyła

Department of Software Engineering

Faculty of Electronics, Telecommunications and Informatics

Gdańsk University of Technology

krissun@pg.edu.pl



Part I

Contents

- File systems
- Serial files
- Sequential files
- Indexed files
- Hash files
- Inverted files
- Multidimensional indexes
- Disk storage

Supplementary reading:

- ✓ H. Garcia-Molina, J.D. Ullman, J. Widom. „Database System Implementation”.
- ✓ N. Wirth. „Algorithms + Data Structures = Programs”.
- ✓ C.J. Date. „Introduction to database systems, 8th Edition”.

File systems

Basic requirements for disk file systems:

- the ability to handle large amounts of data
- quick read access
- convenient update
- good memory usage

Measures of the effectiveness of a given file organization :

S_N - the amount of memory (***disk space***) used for *N* records
T_F - access time to a specified record (associative, or random, access, ***fetch record***)
T_N - access time to the next record, according to the order given (***get next record***)
T_I - time needed to insert a new record into the file (***insert record***)
T_U - time needed to update the record in the file (***update record***)
T_D - time needed to delete the record from the file (***delete record***)
T_E - time needed to read the entire file (***exhaustive read***)
T_R - time needed to reorganize the file (***reorganization***)

NOTE:

These times are measured in the number of accesses (disk page reads/writes) to the disk memory, not in real time units!

Usually we will consider average times (expected values in terms of probability) .

Important file system parameter:

Blocking factor

$$b = B / R$$

where:

B – the size of a disk block that is a read/write unit; usually from a few to a dozen kB (such a block is also called a *disk page*);

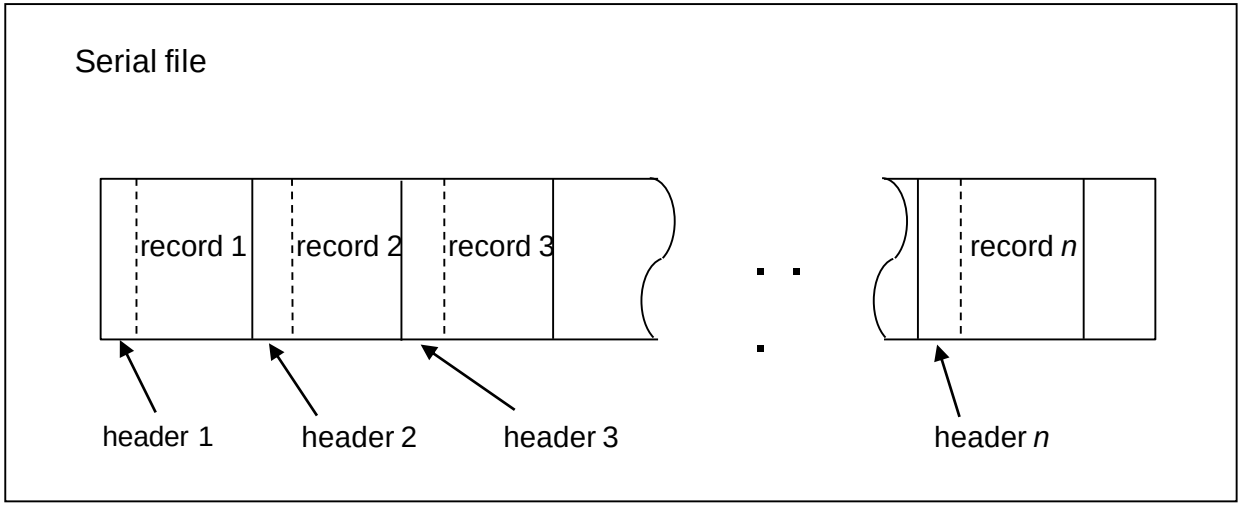
R – average record size in the file.

Thus, *b* is the average number of records on one disk page; this is also the average number of records read/written by one disk operation.

Serial files (Piles)

In a serial file:

- data is collected in the order of appearance at the system input
- records can have different lengths and may not have a fixed internal structure



A new record is appended to the end of the file.
Typically, records have headers that show the structure of the record.
Serial files can be processed only sequentially.

Applications: System log files (*journals*, *logs*), files with measurement data, etc., intended for processing and analyzing data in off-line mode.

Serial files

- Let's assume that the average length of the record is *R* bytes. Then the file takes:

  $S_N = N * R$ bytes, i.e. N / b disk pages

- The time needed to read the entire file is:

  $T_E = N / b$

- The average time needed to read one specific record is:

  $T_F = 0{,}5 * N / b$

  which can be prohibitively large. A better method is to process this type of requests in a batch consisting of, say, 100 requests.

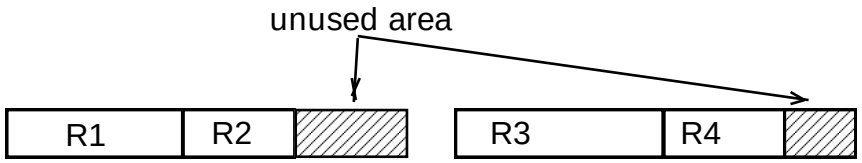
- Inserting a new record is very simple and requires:
  - either one disk access if we start a new page,
  - or two disk accesses if we read the last page of the file, insert a record and write this page to disk.

  It can be shortened to the average value $T_I = 1 / b$, if the new page is saved only when it is full (we apply *caching*).

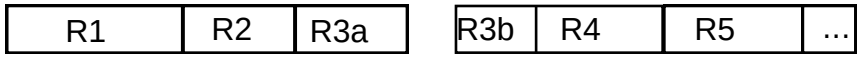
- Record update – practically impossible.

Serial files

- *Spanning* - a technique used to completely fill disk pages at the expense of dividing records between subsequent pages, which makes writing and reading records more costly.



a) Variable-length records, spanning not allowed

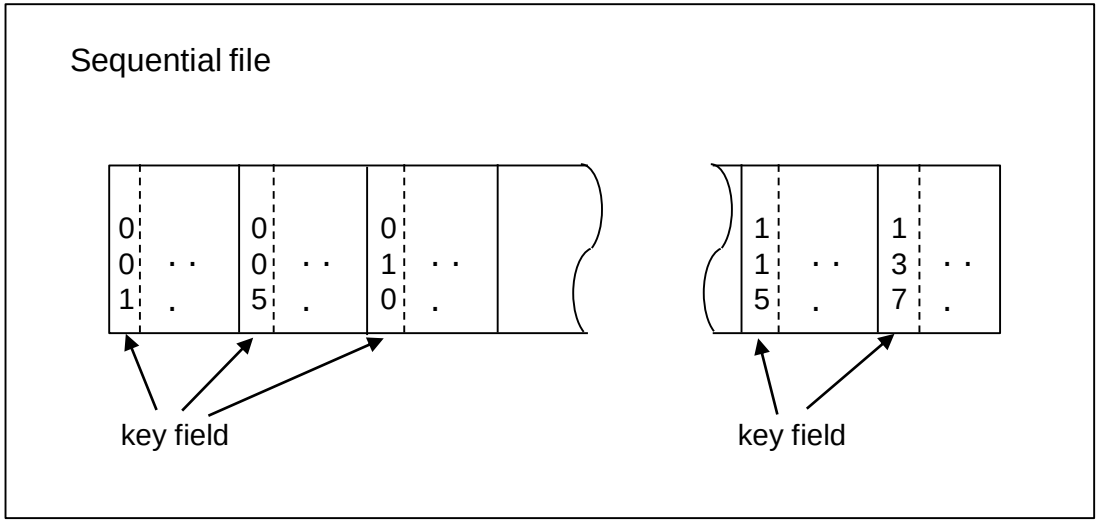


b) Variable-length records, spanning allowed

Sequential files

In a sequential file:

- data is organized according to the value of the key (simple or composite)
- the records are of fixed length and have a fixed structure.



This structure is similar to a sorted table in main memory.

Sequential files

Possible methods for fetching a record specified by the key value :

- Sequential search (record by record) of the file:

$$T_F \approx 0{,}5 * N / b$$

(makes sense if the file is small, e.g. it takes only several disk pages)

- Binary search (the same method as searching for a record in a sorted table, i.e. bisection)

$$T_F \approx \log_2 (N / b)$$

- Interpolation search (sampling): we try to guess the position of the record in the file using linear interpolation; in the next steps, we repeat the linear interpolation, or change to bisection or sequential search. This method similar to looking for an item in a large dictionary.

This method is very effective if:

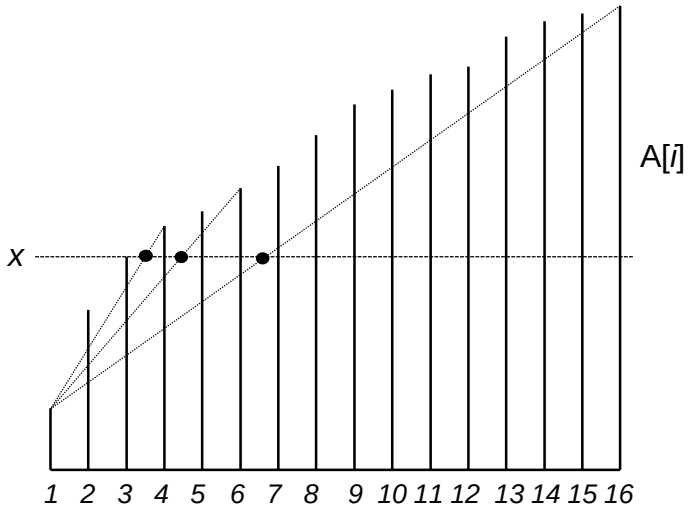
- N is very large (millions of records)
- the distribution of key values is close to linear.

In this case we can expect that

$$T_F \approx \log_2 \log_2 (N / b)$$

Sequential files
Interpolation search

Illustration of interpolation search (sampling):
searching for record with key=*x*.



1st trial: $i = 6$
2nd trial: $i = 4$
3rd trial: $i = 3$, success.

Sequential files
Interpolation search

Two extreme cases

Case 1: Distribution of keys in file F is very uniform (linear):

Keys in F: 1 3 5 7 9 11 13 15
searching for: x = 11

$\alpha_1 = (11-1)/(15-1) = 10/14$; $k_1 = \lfloor 1 + 10/14 \cdot 7 \rfloor = 6$; F[6] = 11 = x;

END (success).

Sequential files
Interpolation search

Case 2: Distribution of keys in file F is very non-uniform:

Keys in F: 1 2 3 4 5 6 7 100
searching for: x = 7

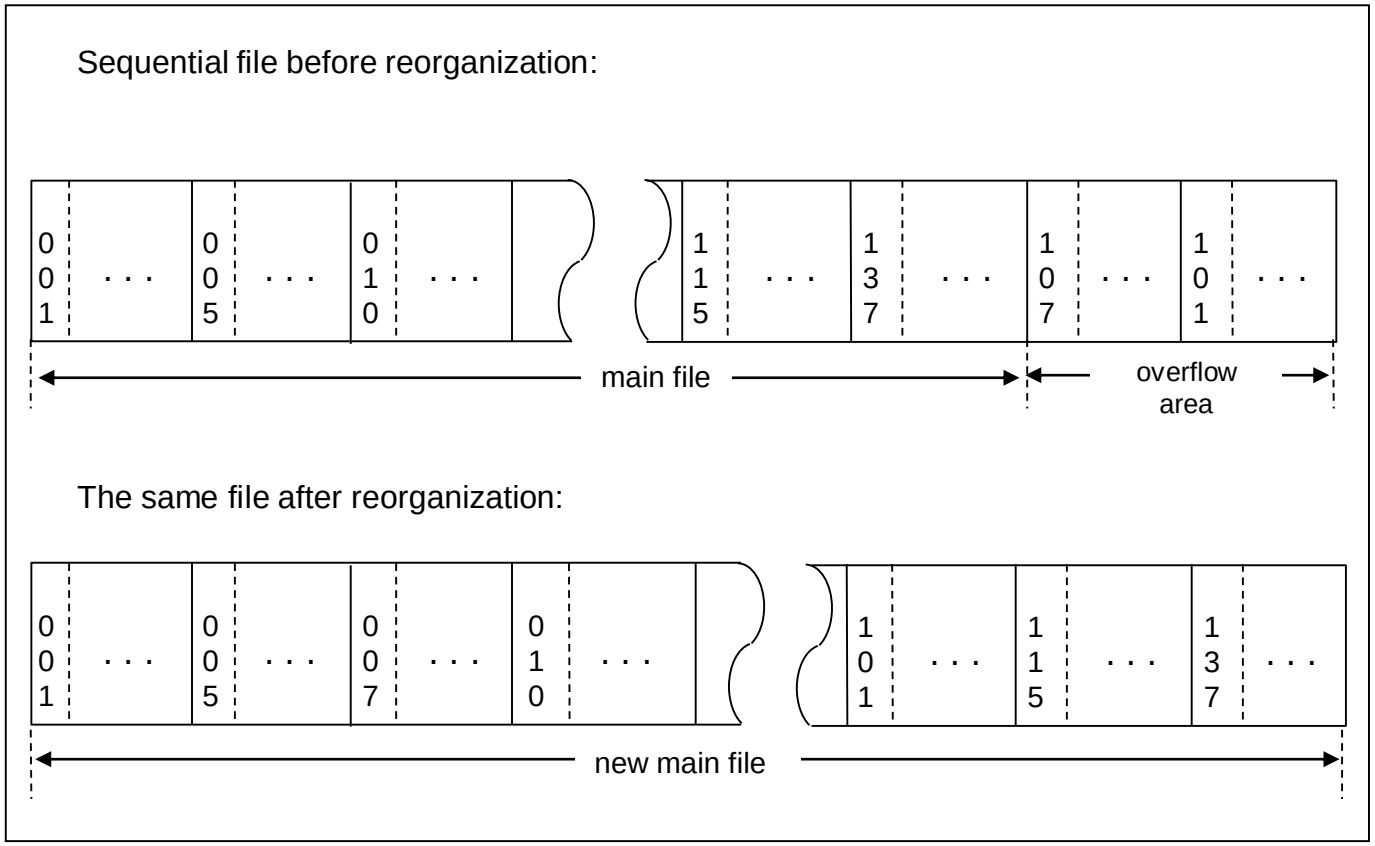
$\alpha_1 = (7-1)/(100-1) = 6/99;\ k_1 = \lfloor 1 + 6/99 \cdot 7 \rfloor = 1;\ A[1] = 1 < x;$
$\alpha_2 = (7-2)/(100-2) = 5/98;\ k_2 = \lfloor 2 + 5/98 \cdot 6 \rfloor = 2;\ A[2] = 2 < x;$
. . .
$\alpha_6 = (7-6)/(100-6) = 1/94;\ k_6 = \lfloor 6 + 1/94 \cdot 2 \rfloor = 6;\ A[6] = 6 < x;$
$\alpha_7 = (7-7)/(100-7) = 0;\ k_7 = \lfloor 7 + 0 \cdot 1 \rfloor = 7;\ A[7] = 7 = x;$

END (success) but the numer of trials is *N*-1

sequential search! unacceptable!

Sequential files

Inserting a new record into a sequential file without breaking its structure is practically impossible in online mode (it would require O (*N*) time), therefore an additional overflow area is assigned, which is in fact a serial file. Periodically, the entire file is reorganized.

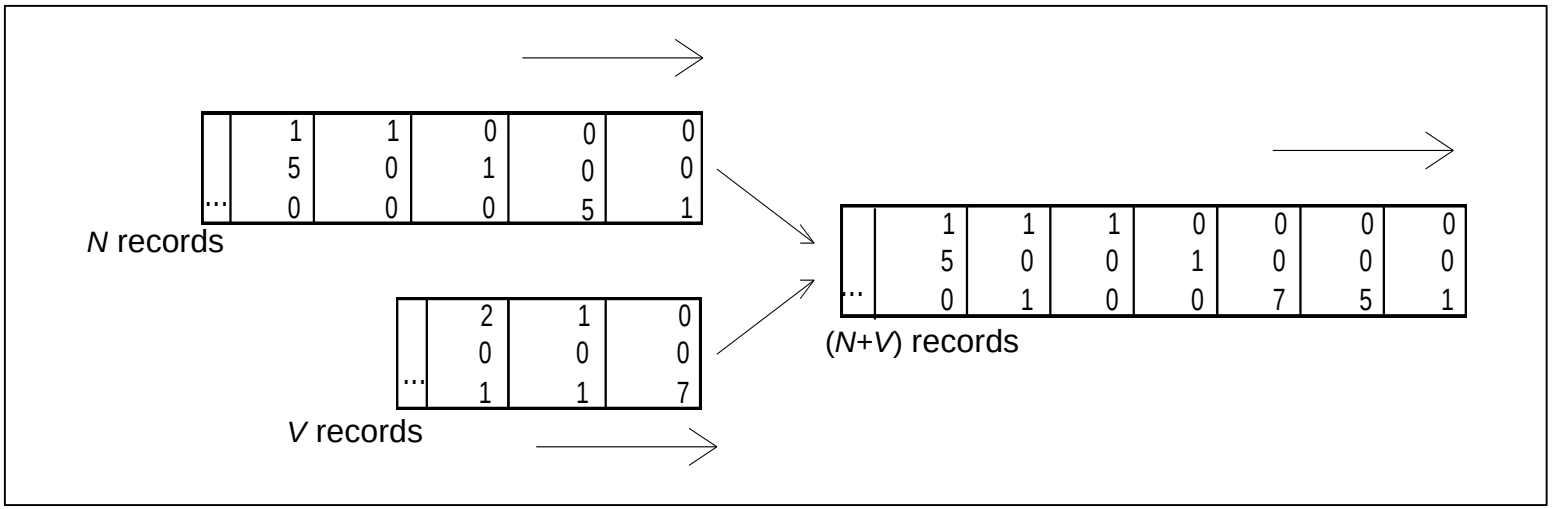


Sequential files

Methods for reorganizing the sequential file:

- Sorting the whole file (*N* + *V* records), which requires O((*N* + *V*) log(*N* + *V*)) disk operations (see Section "Sorting sequential files")

- Sorting the overflow area (cost: O(*V* log *V*) disk operations) and merging it with the main area (cost: 2(*N* + *V*) / *b* disk operations). If *V* is much (e.g. 10 times) smaller than *N* (which is usually the case), then this method is faster.



Merging the main area and sorted overflow area into a sorted sequential file

Sequential files

Getting the next record is very easy in the sorted sequential file (no overflows, $V = 0$): you should just fetch the next record from the file, which costs:

- either 0 if the page with the next record is already in the main memory; probability: (b-1) / b;
- or 1 if you need to read the next page from the disk; probability: 1 / b.

Thus: $T_N = (b\text{-}1)/b * 0 + 1/b * 1 = 1/b$

However, if $V > 0$, then:

- either we must search through the overflow area every time,
- or ignore the overflow area until the file is reorganized.

Note that the same alternative occurs for associative access (**fetch record**)

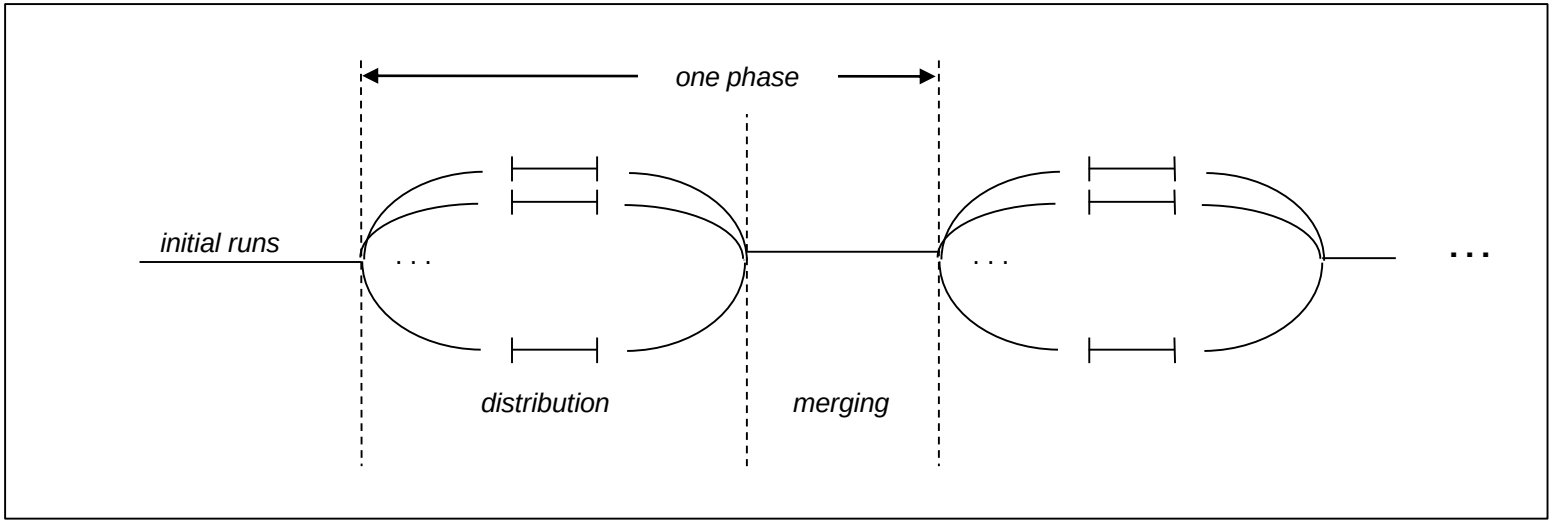
Sorting sequential files

Problem:

There is a sequential file with N records with keys x_1, x_2, ..., x_N. Operating memory (RAM) can simultaneously hold only $m << N$ records. Order the file in such a way that $x_1 \leq x_2 \leq ... \leq x_N$.

***General strategy**:* („*tape merge pattern*")

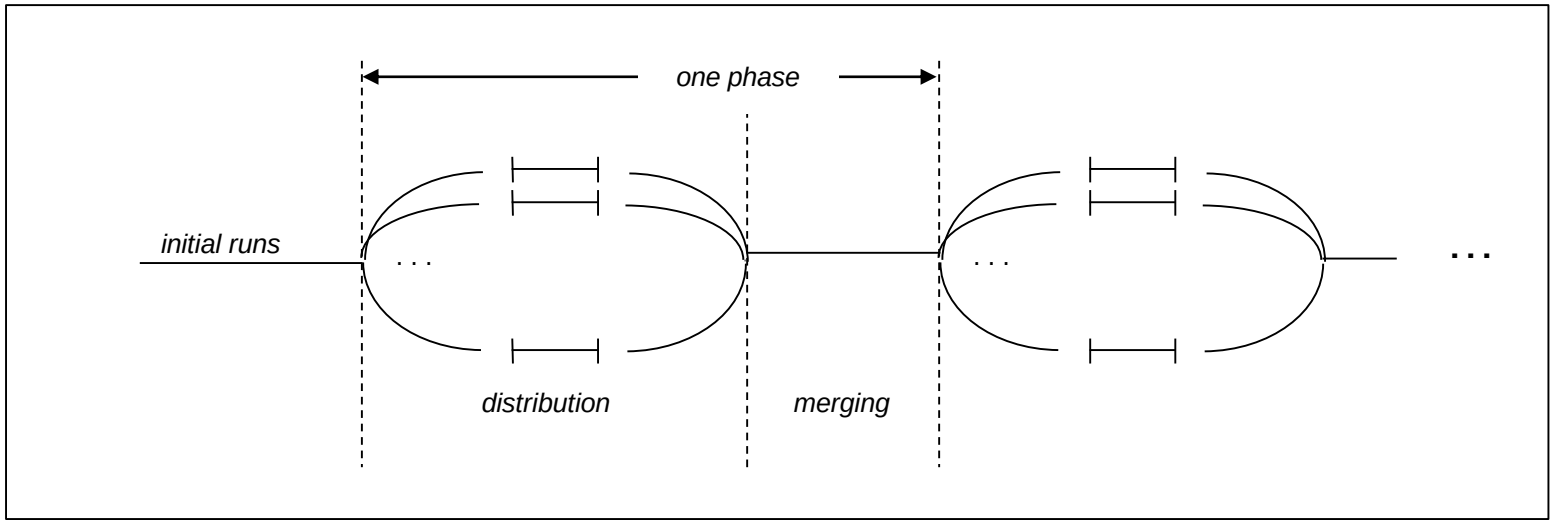
Sort the file with pieces called *runs* (a run - sequence of elements of the file already sorted in a given order): initial runs are merged, creating a fewer number of longer runs. Act that way iteratively until you get one run as long as the entire file.



General idea of sorting by merging (merge sort)

Note:
Disk files used in merge sort are usually called "tapes" due to sequentiality in processing of them.

Sorting sequential files



Sorting takes place in phases. Each phase consists of distribution and merging. If we use t tapes for distribution, then each merging will reduce the number of runs t-fold. So, if the file initially has r runs, then we do not need more than $\lceil \log_t r \rceil$ phases to sort the entire file.

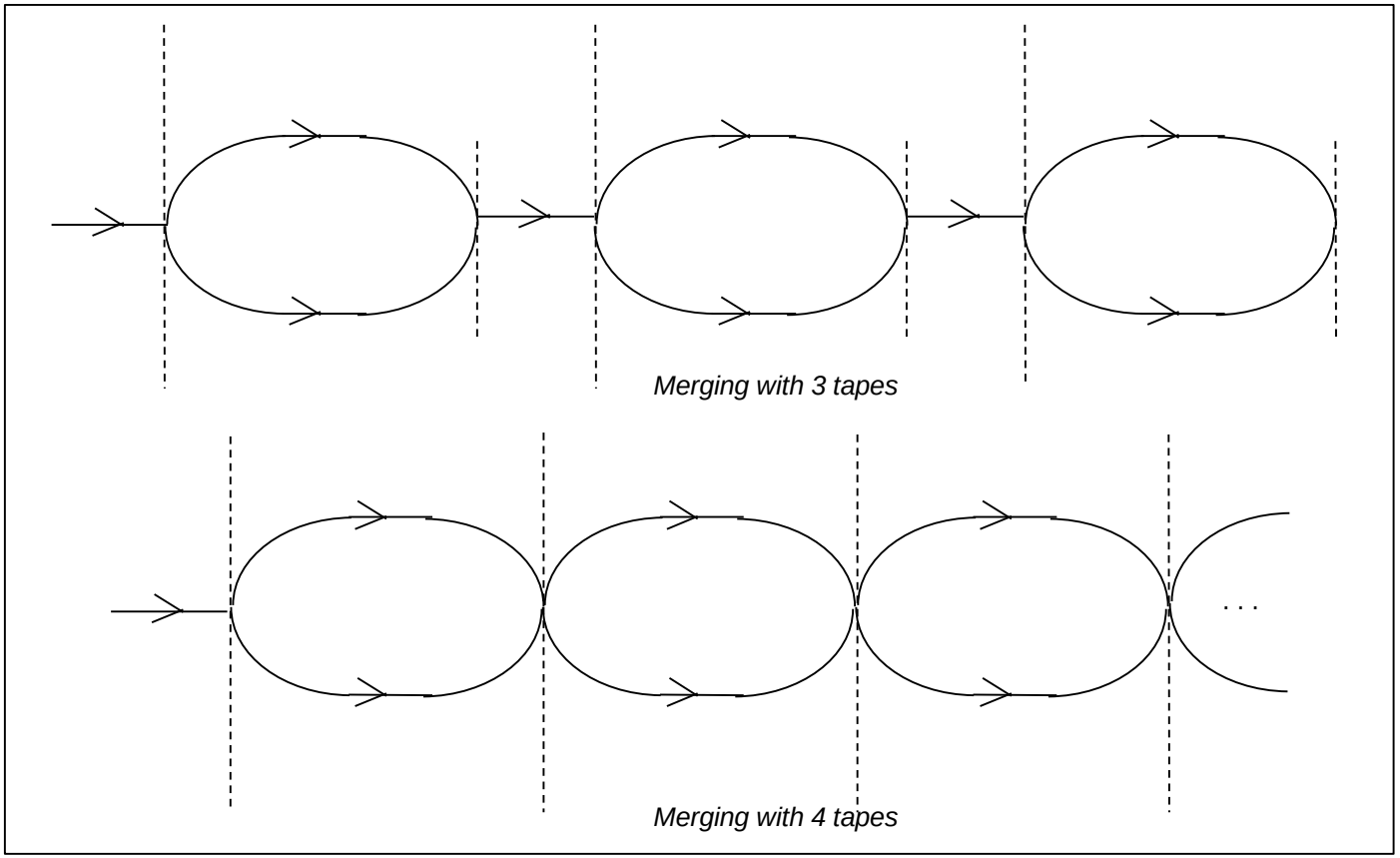
Notes:

1. In the above schema, only t+1 tapes are needed, regardless of the number of phases, since in the next phase the tapes from the previous phase can be re-used.

2. In general, sorting by merging can also be used to sort arrays in RAM. However, the space complexity of this method is O(N), which usually (for large N) limits its use to external (disk) memory.

Sorting sequential files

Typical simple schemes of sorting by merging:



Scheme 2+1:
4 disk operations for each record in each phase:
2 reads and 2 writes

Scheme 2+2:
2 operations for each record in each phase:
1 read, 1 write

In Scheme 2+2, distribution and merging are performed simultaneously.

Advantage: 2 times less disk operations
Disadvantage: 1 additional tape is needed.

Natural merge

Example: Natural merge, scheme 2+1.

File: (*N* =8)

44 55 | 12 42 94 | 18 | 06 67 (4 runs of different length)

The runs are distributed *alternately* on 2 tapes:

1st phase:

t1: 44 55 | 18
t2: 12 42 94 | 06 67

t3: 12 42 44 55 94 | 06 18 67 (2 runs)

2nd phase:

t1: 12 42 44 55 94
t2: 06 18 67

t3: 06 12 18 42 44 55 67 94 (1 run, file sorted)

Note: No sorting by merging method requires initial counting the number of runs in the file to be sorted. Everything is made by the algorithm ”on the fly”.

Natural merge

Analysis:

In the worst case (file ordered inversely), the number of phases is $\lceil \log_2 N \rceil$, because the initial number of runs is N. The, the number of disk reads/writes (scheme 2+1) is:

$$4N \lceil \log_2 N \rceil / b$$

In the average case (in a probabilistic sense) the initial number of runs is less than N. If this number is r, then the maximal number of phases is $\lceil \log_2 r \rceil$, and the number of disk reads/writes will not exceed

$$4N \lceil \log_2 r \rceil / b$$

(Note: For scheme 2+2 this number will be twice as small: $2N \lceil \log_2 r \rceil / b$).

It can be shown that the expected number of series in a random file with N elements is $N/2$ (prove this!).
So, on average natural merge "saves" at least one phase compared to the worst case.

Natural merge

Example: Natural merge, scheme 2+2.

File: (*N* =8)

44 55 | 12 42 94 | 18 | 06 67 (4 runs of different length)

The series are merged from two tapes and immediately distributed alternately into 2 tapes:

1st phase:

t1:	44 55 | 18
t2:	12 42 94 | 06 67
t3:	12 42 44 55 94
t4:	06 18 67

2nd phase:

t1:	06 12 18 42 44 55 67 94	(1 run, file sorted)
t2:	*empty*	
t3:	*empty*	
t4:	*empty*	

Natural merge

In natural merge the effect of coalescing adjacent runs may occur:

Example: Natural merge, scheme 2+1.

File: (*N*=8; the file element 18 from the previous example replaced by 60)

	44 55 | 12 42 94 | 60 | 06 67	(still 4 runs of different length)
1st phase:		
t1:	44 55 ' 60	(two adjacent runs coalesced)
t2:	12 42 94 | 06 67	
t3:	12 42 44 55 60 94 | 06 67	(2 runs)
2nd phase:		
t1:	12 42 44 55 60 94	
t2:	06 67	
t3:	06 12 42 44 55 60 67 94	(1 run, file sorted)

Note that in this example, coalescing some runs did not reduce the number of phases.
However, this efect may be very profitable, as you will see in the next example.

Natural merge

In general case, the effect of coalescing adjacent runs can significantly reduce the number of phases.

Example: Natural merge, scheme 2+1.

Initial file (*N*=8):

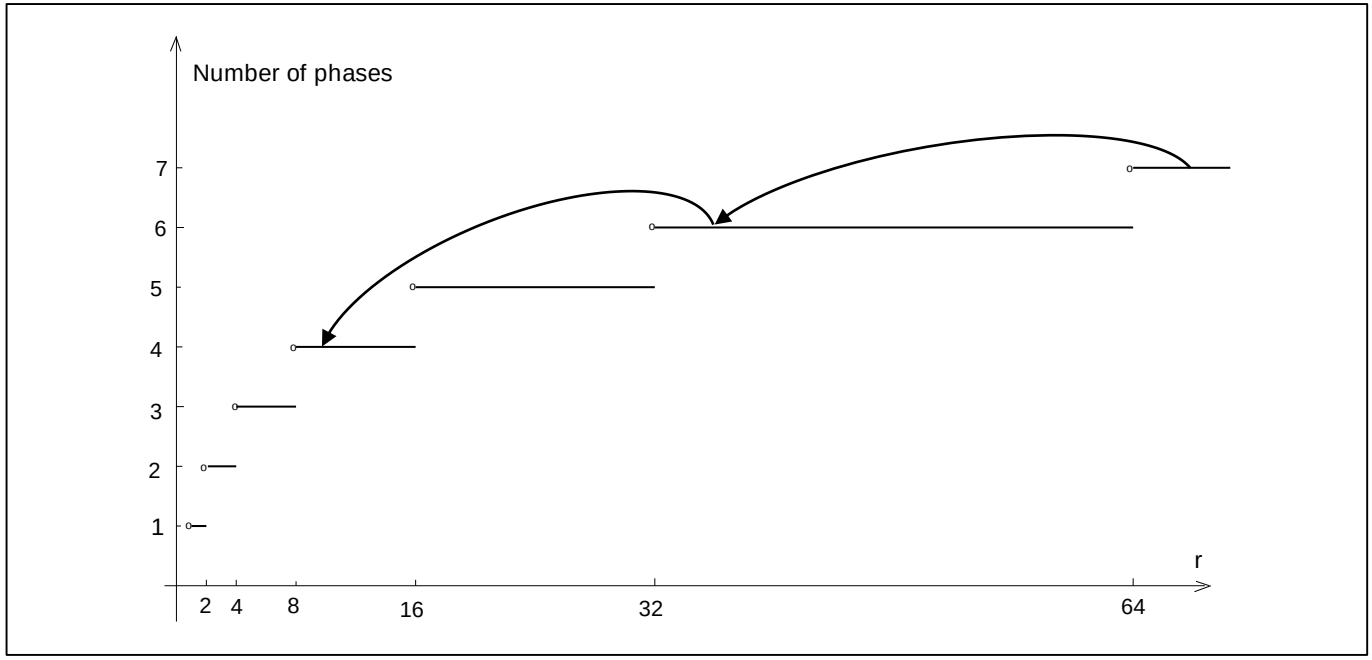
10 | 9 12 | 11 15 | 14 18 | 16 (5 runs)

1st phase:

t1: 10 ' 11 15 ' 16

t2: 9 12 ' 14 18

t3: 9 10 11 12 14 15 16 18 (1 run, file sorted; only 1 phase was needed)



Natural merge

The effect of coalescing runs may cause the number of series on particular tapes to differ significantly.

Example:

File (*N*=8):

10 | 9 12 | 8 15 | 14 18 | 13 (5 runs)

1st phase:

t1: 10 | 8 15 | 13 (3 runs)
t2: 9 12 ' 14 18 (only 1 run)

t3: 9 10 12 14 18 | 8 15 | 13 (3 runs)

2nd phase:

t1: 9 10 12 14 18 | 13
t2: 8 15

t3: 8 9 10 12 14 18 | 13 (2 runs)

3rd phase:

t1: 8 9 10 12 14 18
t2: 13

t3: 8 9 10 12 13 14 15 18 (file sorted)

Multiway natural merge

Using more tapes may further reduce the number of phases needed to sort the file.

Example: Scheme 3+1

File: (N=8)

44 55 | 12 42 94 | 18 | 06 67 (4 runs)

1st phase

t1:	44 55 | 06 67	(2 runs)
t2:	12 42 94	(1 run)
t3:	18	(1 run)
t4:	12 18 42 44 55 94 | 06 67	(2 runs)

2nd phase:

t1:	12 18 42 44 55 94	(1 run)
t2:	06 67	(1 run)
t3:	*empty*	
t4:	06 12 18 42 44 55 67 94	(file sorted)

If we use t+1 tapes (t for distribution and 1 for merge), then the maximal number of phases is $\lceil \log_t r \rceil$:

t + 1	No. of phases
3	$1{,}00 \lceil \log_2 r \rceil$
4	$0{,}63 \lceil \log_2 r \rceil$
5	$0{,}50 \lceil \log_2 r \rceil$
6	$0{,}43 \lceil \log_2 r \rceil$
...	...

Polyphase (Fibonacci) merge

In polyphase merge, in subsequent phases each tape is cyclically used for distribution and for merging.

In the examples we assume:
3 tapes,
initially r = 13 (NOTE: only the number of runs on tapes is shown)

Problem: What is the optimal initial distribution of the runs between the tapes?

Guess 1: Initial distribution – most uniform: 7, 6, 0

	t1	t2	t3
Phase No	7	6	0
1.	1	0	6
2.	0	1	5
3.	1	0	4
4.	0	1	3
5.	1	0	2
6.	0	1	1
7.	1	0	0

7 phases were needed to sort the file.

Notes on polyphase merge:

1. In each phase, the element that participates in the merging is once read from the disk and once written to the disk.
2. That not all elements of the file participate in the merging.
3. After initial distribution, there is no risk of coalescing adjacent runs.

Sorting sequential files

Polyphase merge

Guess 2: Initial distribution – most nonuniform : 12, 1, 0

	t1	t2	t3
Phase No.	12	1	0
1.	11	0	1
2.	10	1	0
...	...	...	...
9.	3	0	1
10.	2	1	0
11.	1	0	1
12.	0	1	0

As many as 12 phases were needed!

Polyphase merge

Guess 3: Initial distribution of runs – according to the Fibonacci sequence: 8, 5
(after the distribution, the third tape is always empty)

	t1	t2	t3
Phase No.	8	5	0
1.	3	0	5
2.	0	3	2
3.	2	1	0
4.	1	0	1
5.	0	1	0

Only 5 phases were needed. It has been proved that the distribution by the Fibonacci sequence is optimal.

Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$f_1 = 1, f_2 = 1,$
$f_n = f_{n-1} + f_{n-2}$

Polyphase merge

Analysis:

The number of phases needed to sort the file with r runs for "Fibonacci distribution" is (approximately)

$$1{,}45 \log_2 r$$

(Can you prove this?)

But note that in each phase not all records participate in merging. It has been proved that every record is read and written (approx.)

$$1{,}04 \log_2 r$$

times („effective" number of phases). Therefore, the total number of disk operations (reads/writes), including the initial distribution, is (approximately):

$$2N\,(1{,}04 \log_2 r + 1) / b$$

(Compare with natural merge for scheme 2+1!)

NOTE: The above estimates are the better, the higher the value of r is (are asymptotic when r grows to ∞)

Polyphase merge

Question: What if the initial number of runs in the file is not a Fibonacci number?

In this case, we add empty (dummy) runs to one of the tapes in such a way that the total number of runs on both tapes is a Fibonacci number.

Note: Dummy runs do not have to be written to a tape; just store in program variables the number of real runs and the number of dummy runs.

Example: Initial number of runs r = 19. We distribute them *alternately* on two tapes, according to the Fibonacci sequence :

t1: 1, 1+1=2, 2+3=5, 5+8(2)=13(2)
t2: 1, 1+2=3, 3+5=8

Denotation 13(2) means that a tape contains 13 runs: 11 real ones and 2 dummy ones.

	t1	t2	t3
Phase No.	13(2)	8	0
1.	5	0	8
2.	0	5	3
3.	3	2	0
4.	1	0	2
5.	0	1	1
6.	1	0	0

In that way, the number of phases for r = 14, 15, ..., 21 will be the same: 6.

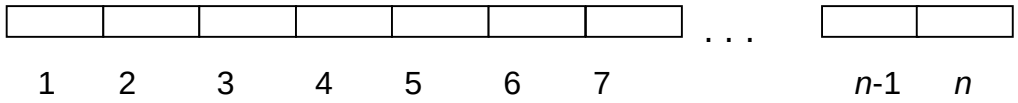
Note 1: During the initial distribution, the adjacent runs on the same tape may coalesce. The distribution algorithm must take this into account.
Note 2: Dummy runs - if present - will always appear on the tape with more runs and will disappear after the first phase.

Merging with large buffers

If large memory is available for the sorting process (and not just a few buffers for disk pages), we can use it to create the initial series as long as possible, then to merge them as in natural merge.

Denotations:

N - number of records in the file
b - blocking factor (number of records in one buffer for one disk page in the main memory)
n - number of buffers in the main memory available for the sorting process



General procedure:

Stage 1. (Creating runs)

1. Read the first *nb* records from the file into the buffers and sort them using an efficient in-memory sorting method (QuickSort, HeapSort, …), creating a run of length *nb*.
2. Write the run on the disk.
3. Repeat steps 1. i 2. until you reach the end of file.

Stage 2. (Merging)

4. Merge the first *n*-1 runs using the *n*-th buffer to create the output run, and write the output run to disk.
5. Repeat step 4 for subsequent runs from the file until you reach the end of the file.
6. Repeat steps 4 and 5 until only one run remains.

Problem: What data structure will you use to effectively merge multiple series (step 4)?

Merging with large buffers

Analysis:

After **Stage 1** we have $\lceil N/(nb) \rceil$ runs on the disk.

The cost of **Stage 1** is $2N/b$ disk operations (each record is read once and once written to disk, b records at once).

After one cycle of **Stage 2** the length of runs grows approximately n-fold (one buffer of n is used for merging and writing to disk, however $n >> 1$, so we can assume $n \cong n\text{-}1$).

After the first cycle of **Stage 2**, the length of runs is approx. n^2b, after the second cycle (if it is needed at all) it is n^3b, etc., so the number of the cycles needed in this stage is:

$$\lceil \log_n (N/(nb)) \rceil = \lceil \log_n (N/b) \rceil - 1$$

Each cycle in **Stage 2** requires $2N/b$ disk operations. The cost of **Stage 1** is also $2N/b$, so the total cost of sorting with large buffers can be approximated by ($\lg n$ means $\log_2 n$):

$$2\frac{N}{b}\log_n\left(\frac{N}{b}\right) = 2\frac{N}{b\lg n}\lg\frac{N}{b}$$

Merging with large buffers

Example 1:

Let $N = 10^7$, $b = 10$, n=1001.
After Stage 1 we have 1000 runs.
After the first cycle of Stage 2 we get 1 run (the file has been sorted).
Cost: $2 \cdot 2 \cdot 10^7/10 = 4\,000\,000$ disk operations.

(Indeed, if $N \leq n^2 b$, then only one cycle is needed in Stage 2.)

Note: The expected cost of natural merge (Scheme 2 + 2) will be

$$2 \cdot 10^7/10 \lceil \lg (10^7/2) \rceil \cong 48\,000\,000$$

Example 2:

Let $N = 10^9$, $b = 100$, n=1001.
After Stage 1 we have 10 000 runs.
After the first cycle of Stage 2 we get 10 runs.
After the second cycle of Stage 2 we get 1 run (not all buffers are needed)
Cost: $3 \cdot 2 \cdot 10^9/100 = 60\,000\,000$ disk operations.

Note: The expected cost of natural merge (Scheme 2 + 2) will be

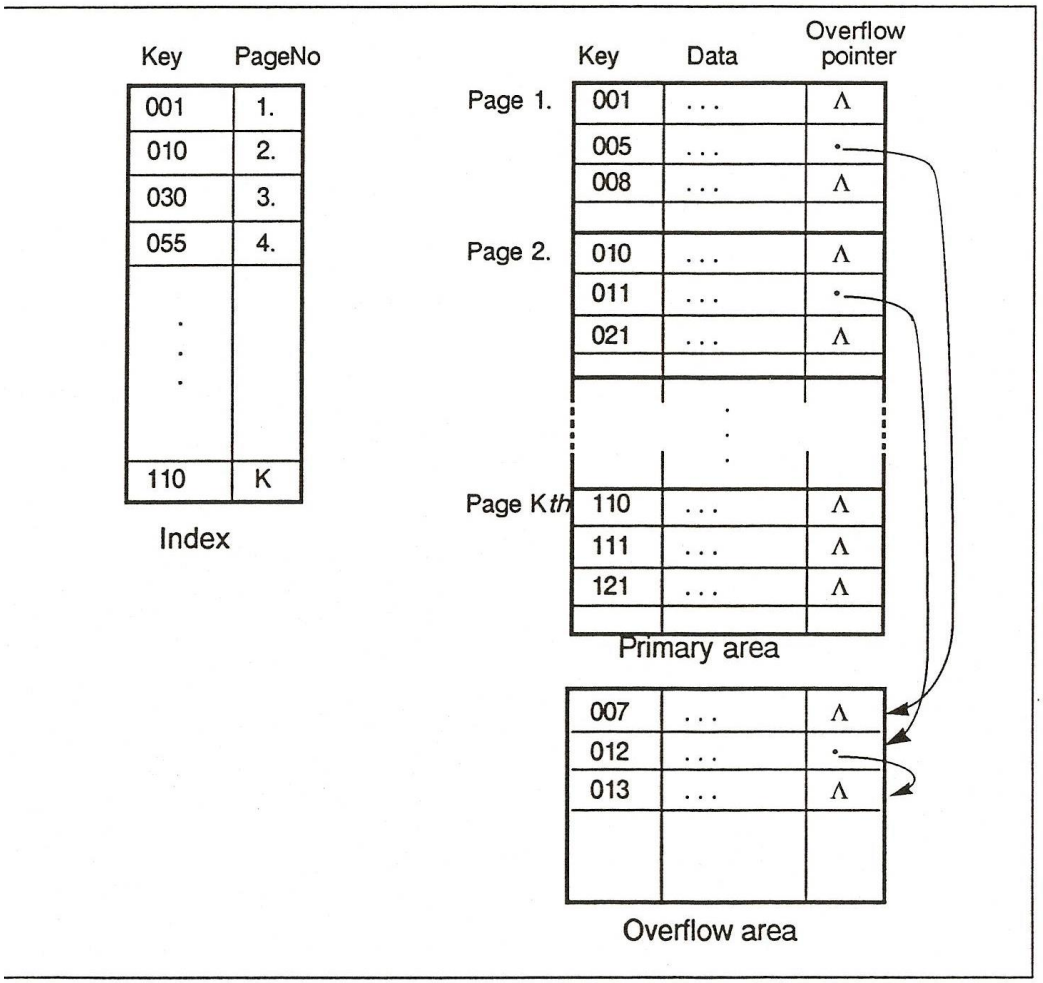
$$2 \cdot 10^9/100 \lceil \lg (10^9/2) \rceil \cong 600\,000\,000$$

Note: In database systems, main memory is a valuable resource due to the fact that many transactions are performed simultaneously and the main memory must be shared among them. As a result, the number of buffers for one sort is limited. However, the **sorting of large sequential files must be performed using a large number of buffers**, otherwise the sorting time becomes unacceptable.

Indexed-sequential files

The indexed-sequential organization (variants: ISAM, CSAM, VSAM, ...) includes the following improvements over the sequential organization:

- index providing faster associative access,
- support for inserting new records.

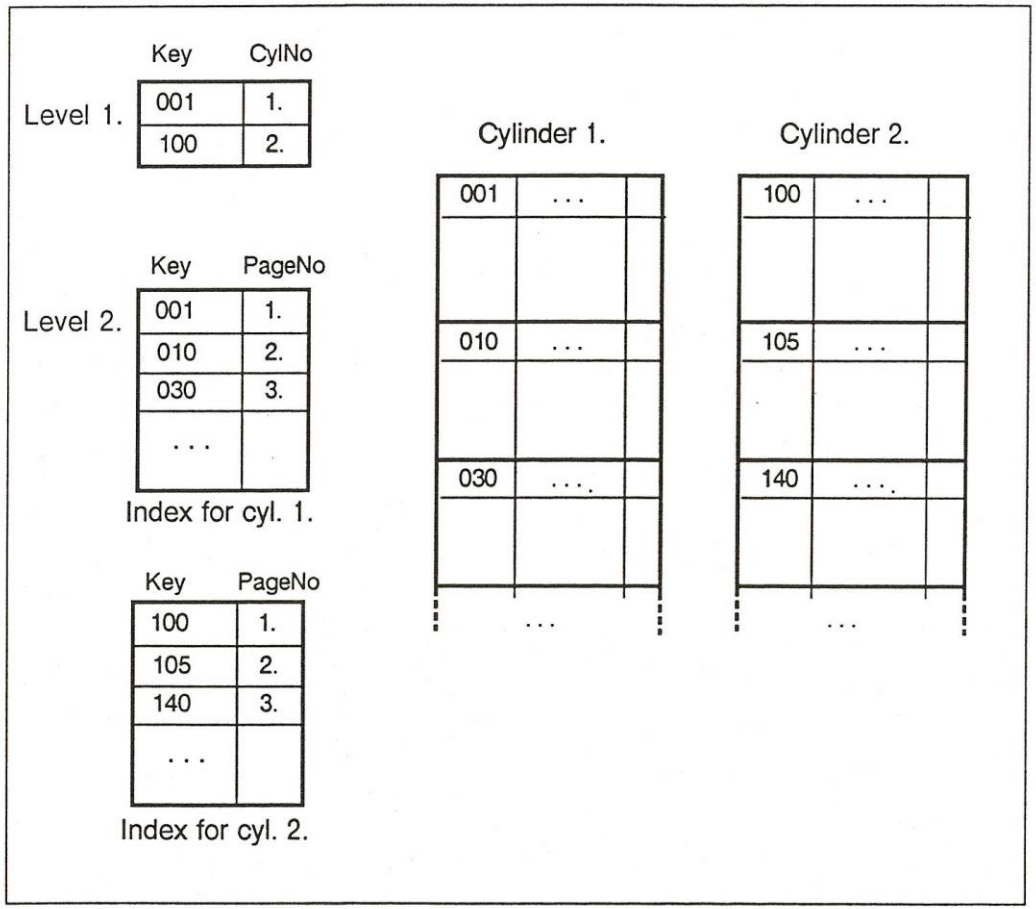


New records that fall into a page that is full are inserted into the overflow area. Records that fall into the same page form a chain (an ordered list).

Searching in the overflow area increases the time to fetch a record, so periodically it is necessary to reorganize the file by re-allocating records from the overflow area to the primary area.

Indexed-sequential files

If the file is very large, multi-level indexes are used to shorten the searching through the index.
For example, cylinders or disk tracks can be indexed at level 1,
and disk pages inside a cylinder or track at the second level.



Indexes are always ordered, which allows you to quickly search though the index. In essence, an index is a sequential file that much smaller than the data file. The entire index or its large fragments (e.g. level 1 indexes) can be kept entire in the main memory.

Indexed-sequential files

Basic parameters

Denotations:

- N – the number of records in the main area
- V – the number of records in the overflow area
- B – disk page capacity (in bytes)
- R – the size of record (in bytes)
- K – the size of key (in bytes)
- P – the size of pointer (in bytes)
- α – page utlilization factor in the main area just after reorganization, $\alpha < 1$

Parameters:

- bf – blocking factor in the main area, $bf = \lfloor B / (R+P) \rfloor$
- bi – blocking factor for index, $bi = \lfloor B / (K+P) \rfloor$
  - bi is $(R+P)/(K+P)$ times bigger than bf (as the whole record is much greater than the key)
- S_N – the size of main area, in pages; $S_N = N/(bf \times \alpha)$
- SO_V – the size overflow area, in pages; $SO_V = V/bf$ (the pages of the overflow area are filled completely)
- SI_N – the size of index (one level assumed), in pages; $SI_N = \lceil S_N/bi \rceil$

Usually, $bi >>$ and i $SI_N << S_N$, so searching through the index on the disk is very fast (or the index fits in the main memory and you do not need access to the disk at all).

NOTE: Indexes in the indexed-sequential organization are called ***sparse*** indexes: not all records are indexed, only the first ones on the pages in the main area.

Performance parameters

Access time to the record specified by the key:

No overflows:

T_F= (time to search through the index) + (page read from the main area = 1 disk access)

If the index is searched by bisection, then:

$$T_F \cong \log SI_N + 1$$

If there are overflows, then the "1" component in the above formula should be replaced by "1 + (average overflow chain length)/2". It can be shown that the average length of the overflow chain is V/N, so in the case that oveflows exist:

$$T_{F\ overfl} \cong \log SI_N + 1 + V/(2N)$$

Time to insert a new record:

No overflows:

$T_I = T_F + 1$ (reading a page and writing the updated page)

If there are overflows, it may happen that you must update the pointer in the preceding record and insert a new record with appropriate pointer, which requires at most 4 additional disk accesses:

$$T_{I\ overfl\ max} = T_{F\ overfl} + 4$$

Performance parameters

Get the next record requires either 0 disk accesses when the next record is on the same disk page as the previous record (this page is cached in main memory), or 1 disk operation, if you need to read the new page:

$$0 \leq T_N \leq 1$$

Delete record costs the same as the read record operation plus one access needed to mark the record as deleted (the physicall delete of the record is postponed until the reorganization of the file):

$$T_D = T_F + 1$$

Update record without changing the key costs the same as the delete record operation.
When the key changes, the update of the record consists of deleting the record and inserting a new record.

The number of disk accesses needed to **read the entire file** is in the best case equal to the number of pages occupied by the main area and the overflow area (each disk page is read only once):

$$T_{E\,min} = S_N + SO_V$$

In practice, the number of necessary page reads will be greater because there is little chance that the page with overflows will be read only once. In the worst case, each overflow page will be read as many times as there are records on it, thus:

$$T_{E\,max} = S_N + V$$

However, this is a very pessimistic case. We will go back to this important problem while discussing *clustered indexes*.

Reorganization

Periodically, when the number of overflows becomes too large, the indexed-sequential file must be reorganized. Reorganization consists in allocating new disk space to the main area and placing overflow records in the main area.

Reorganization (sketch):

Input: N and V in the old file, α for the new file (np. $\alpha = 0.5$), bf, bi.

begin
Allocate disk space for the new main area:
$S_{N\,new} = \lceil (N+V)/(bf \cdot \alpha) \rceil$ pages.

Allocate disk space for the new index (one level assumed):
$SI_{N\,new} = \lceil S_{N\,new}/bi \rceil$ pages.

Allocate disk space for the new overflow area assuming some reasonable V/N ratio, e.g. 0.2).

while not all records from the old file have beed processed **do:**
Read the next record from the old file.
if the current page of the new file already contains $\alpha \cdot bf$ records:
Insert the key of the new record and the addresss of the new page to the new index.
if the page of the new index already contains bi records:
Start a new page of the index.
endif.
Start a new page in the main area of the new file.
end if.
Insert the record into the current page of the new file.
end while.
Delete the old file and the old index.
end.

The cost of reorganization: $T_R = T_{E\,old} + S_{N\,new} + SI_{N\,new}$