Struktury baz danych

Krzysztof Goczyła
Katedra Inżynierii Oprogramowania
Wydział Elektroniki, Telekomunikacji i Informatyki
Politechnika Gdańska
kris@eti.pg.gda.pl

Literatura:

H. Garcia-Molina, J.D.Ullman, J. Widom „Implementacja systemów baz danych". WNT 2003.
T. Pankowski. „Podstawy baz danych". PWN, 1992
N. Wirth. „Algorytmy+Struktury danych=Programy". WNT 2004.

File System Organisation

The basic requirements for a file system are:
- ability to keep large amount of data
- fast access for retrieval
- convenient update
- economy of storage

The stages of work with a file are:
1. *opening* - reading directory record of the file
2. *operations* - reading, writing and updating the file
3. *closing* - updating the directory record

Measures of performance:
- *storage* for *N* records S_N
- time to *fetch* an arbitrarily selected record T_F
- time to *get the next record* according to an order specified T_N
- time to *insert* a new record into the file T_I
- time to *update* (change) a record in the file T_U
- time to *delete* a record from the file T_D
- time to *read* the entire file T_E
- time for *reorganisation* of the file T_R

In the following, we will replace the *time* by the *number of disk accesses* necessary to perform an operation, so that in our reasoning we are independent of time characteristics of specific disks. If not explicitly stated otherwise, we will consider *average* values.

One of the major factors that influence the performance of a specific file organisation is *blocking factor* ***b***:

$$b = B / R$$

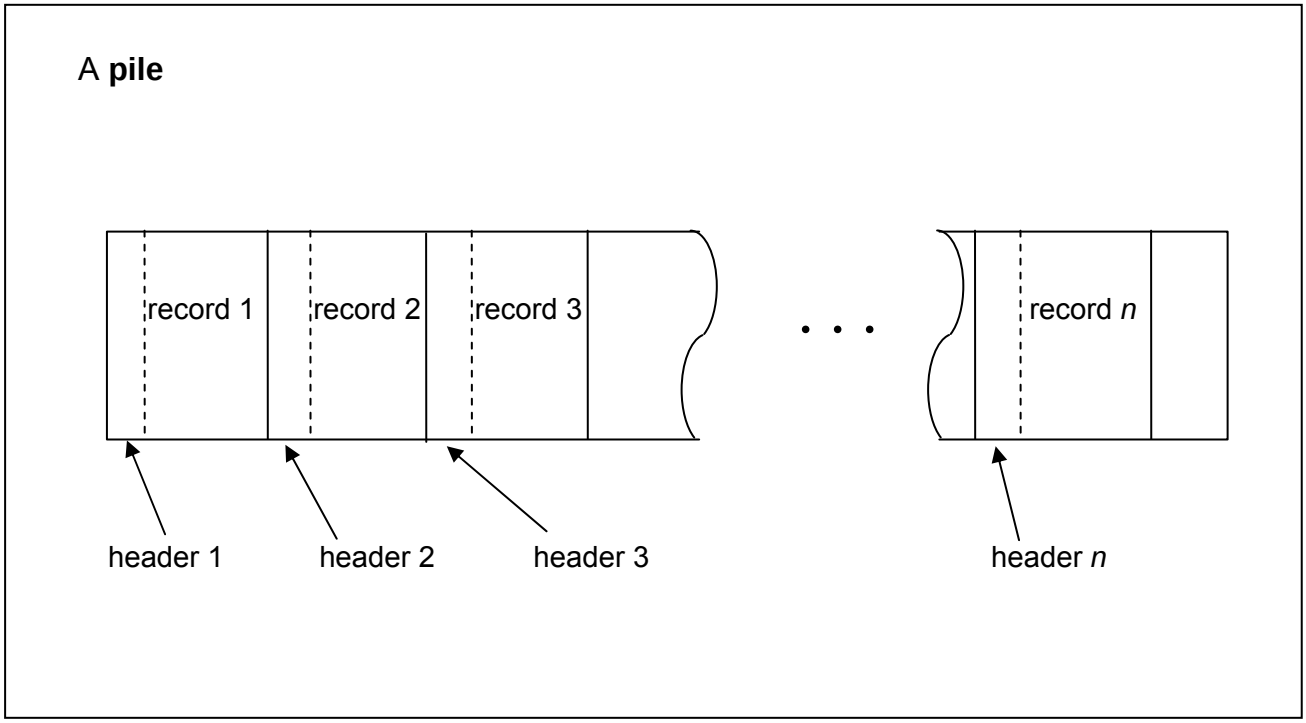
where
B - block (page) size, where block (or page) is the amount of disk space read by the system as a unit;
R - (average) record size.

Basic File Structures - The Pile (1)

The simplest (minimal) metod is

- Pile (serial file)

In a pile, data is collected in the order in chich it arrives. The records may be of variable length and need not have any fixed internal structure



A new record is always appended to the end of file.

Usually, each record is preceeded by a header specifying the length of the record, its contents etc. The fields within the record may also be preceeded by headers.

The piles may be processed only sequentially, so their application is very limited. They may be used as journal files (logs), files collecting measurement data and in other applications in which the collected data are to be processed off-line.

The Pile (2)

In a pile, records are of variable size. Assume that average record has *R* bytes. Then, the file occupies $S_N = N * R$ bytes, or N / b pages.

Time to perform an **exhaustive read** of a pile is equal to

$$T_E = N / b$$

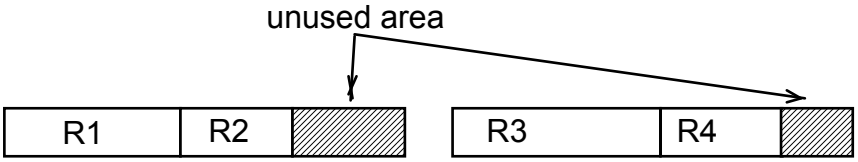
To **fetch** a specific record, we need on the average to search a half of the file, so

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

It is usually a very long time. The partial solution to this problem is to collect requests to a pile into a batch and to process the whole batch by reading the entire file.

The **insert** operation is simple, as we always append a new record to the end of the file and we can assume that the address of the last block is known. Then, T_I is equal to 1 (if we begin a new block) or 2 (if we read the last block, insert a new record into it, and then re-write the block onto the disk).

The number of necessary disk accesses will be larger if we allow for *spanning* of a record over two consecutive blocks, or if a record is larger that the block size. The algorithm for insertion remains however straightforward.



a) Variable-length records, spanning not allowed

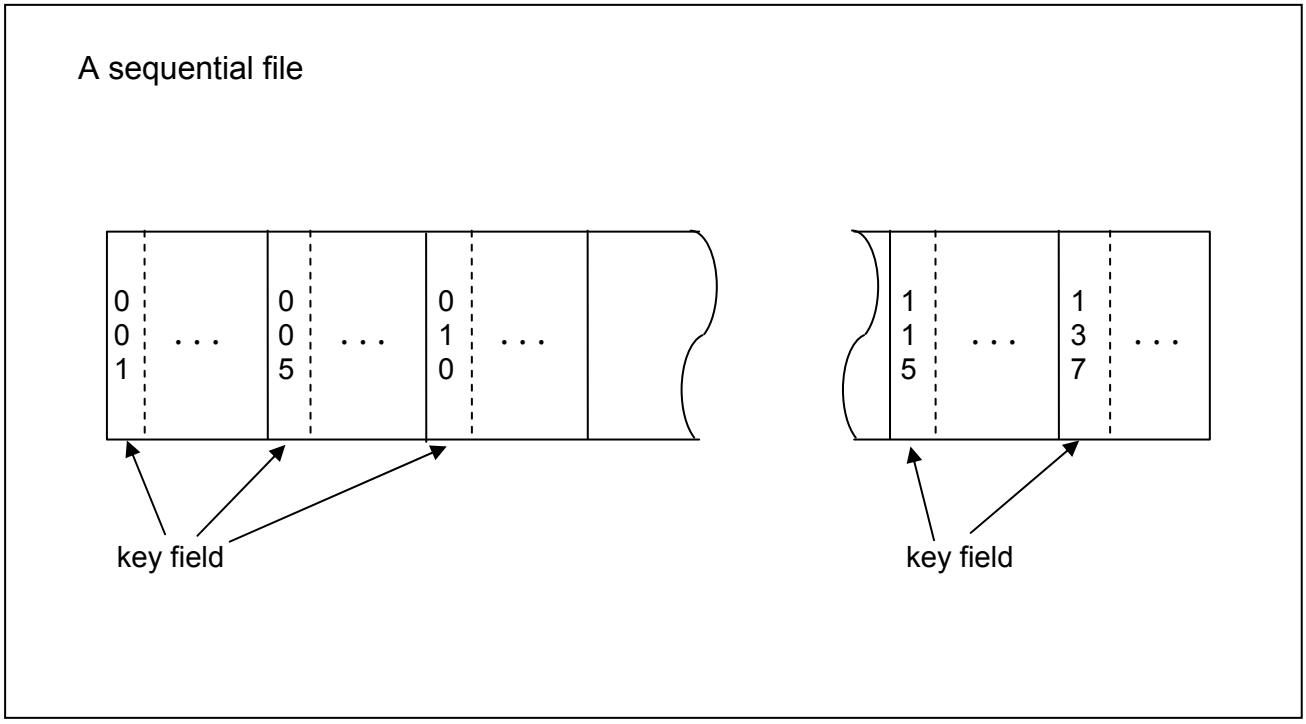
b) Variable-length records, spanning allowed

Update operation can usually be performed by invalidating the record to be changed (by flagging it as "deleted") and inserting the new version at the end of the file, unless the update does not change the size of the record.

Basic File Structures - The Sequential File (1)

In a sequential file, records are ordered according to a key attribute. The records are of fixed size.

This organization is similar to the tabular organization in main memory



Fetching (**retrieving**) a specified record may be accomplished by:

- Sequential search through the file, particularly if the search argument is not the key attribute; in this case, we must search, on the average, through the half of the records; $T_F \cong 0{,}5^* N / b$;

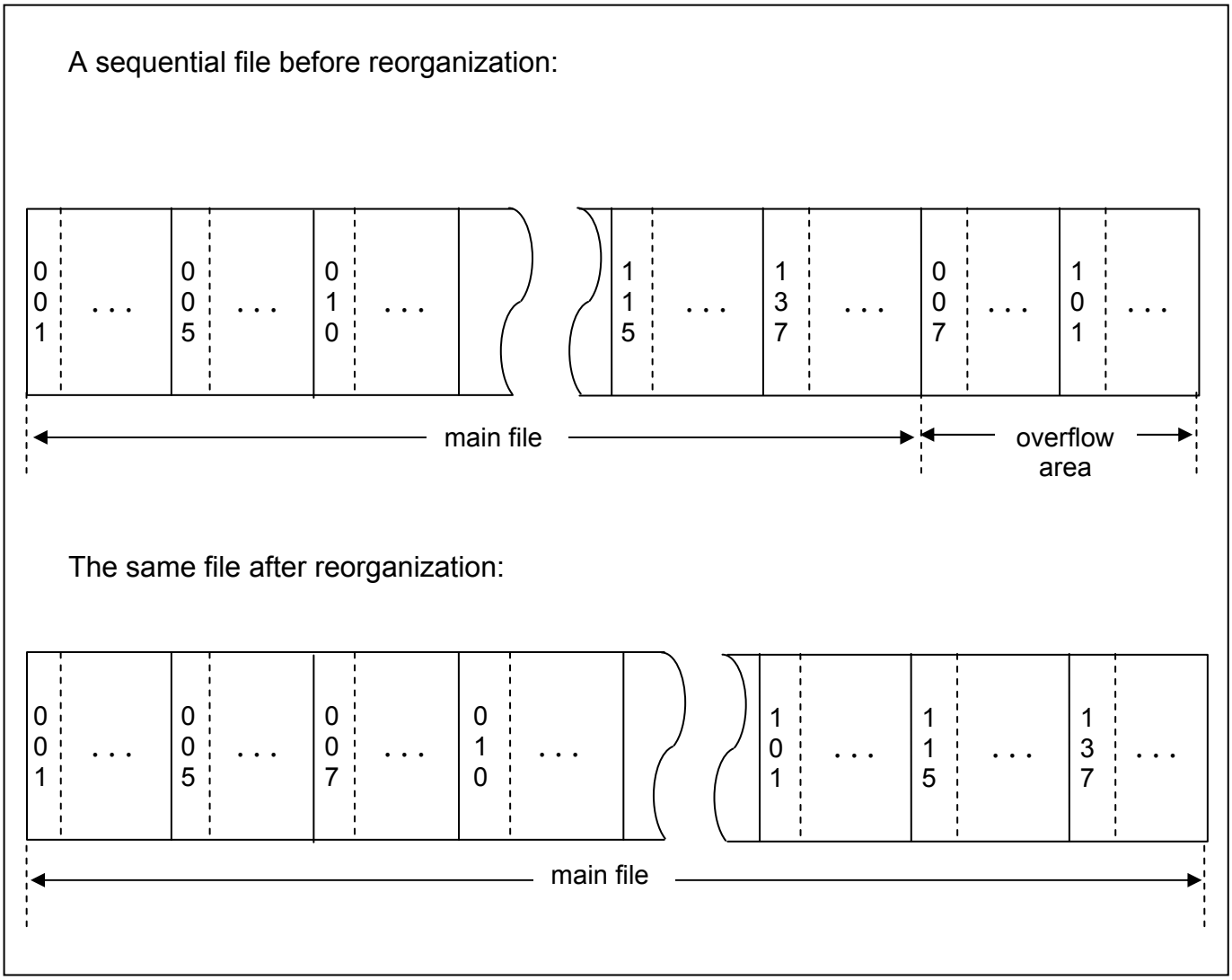
- Binary search, if the search argument is the key attribute; on the average, the number of disk accesses $T_F \cong \log_2 (N / b)$;

- Probing (interpolation search): initial fetch is made at an estimated position in the file; the next fetches may be sequential, binary searches or further probing (this procedure is similar to that performed by a human while searching for an item in a large telephone directory).
  Probing is very efficient if:
  - *N* is very large
  - distribution of key values is close to uniform

The Sequential File (2)

Inserting a new record to a sequential file is normally impossible on line, because it would require considerable time to make space for it at proper place in the file.

New records may be added to the end of file (to the overflow area) and, if the retrieval in the main file fails, searched through. The whole file must be periodically **reorganized** to restore the proper sequence of records.



The reorganization results in the whole file being sorted according to the key field.

The Sequential File (3)

Get next record operation on a sequential file with no overflows may require no disk accesses at all, as the accessed record may reside in the page buffer:

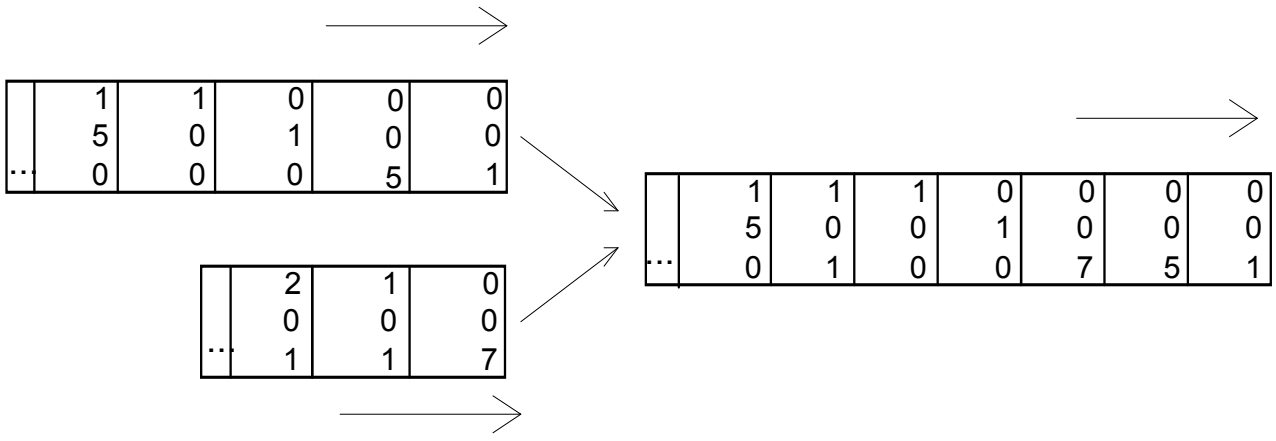
$T_N = (b\text{-}1)/b * 0 + 1/b * 1 = 1/b$

In the presence of overflows, the performance of a sequential file deteriorates, as we have to seqentially scan the overflow area if we want to access a record. A solution is to neglect the presence of overflows at all under on-line operations, and to reorganize the file periodically to include the records from the overflow area into the main area of the file.

Let us assume that in the overflow area there are *V* records. The **reorganisation** of a sequential file may be performed in the two following ways:

1. **Sorting the whole file** (i.e. *N*+*V* records), which requires O((*N*+*V*) log(*N*+*V*)) operations (as we will see later from the discussion of file sorting methods);

2. **Sorting** only **the overflow area**, which costs O(*V* log *V*) disks accesses, and then **merging** the main file and sorted overflow area into a new sequential file, which costs 2(*N*+*V*)/*b* disk accesses. If *V* is small in comparison with *N*, it is a cheaper way of reorganisation.



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

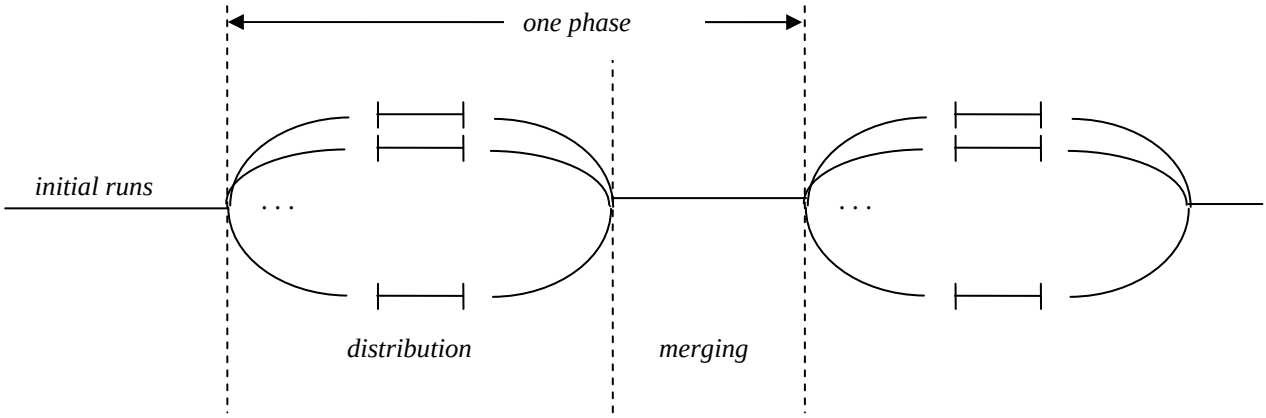
Sorting sequential files (1)

Problem:

We have a large table $x_1, x_2, ..., x_n$, stored externally (in a file). The internal memory can hold only $m \ll n$ elements. Sort the table (i.e. the file) so that $x_1 \leq x_2 \leq ... \leq x_n$.

General strategy: (*Tape merge* pattern)

To sort the elements in a piecemal fashion: initial *runs* (a *run* - a series of elements that are ordered in ascending/descending order) are *merged* together into a smaller number of longer runs, until finally we obtain one run as long as the sorted file.



General idea of sorting by merging

If we merge onto t tapes, each merging decreases the number of runs by a factor of $1/t$. If there are r initial runs, $\lceil \log_t r \rceil$ phases are required. Each phase consists of: distributing the runs onto t tapes and then merging them together.

Remark:
Theoretically, the merging technique can be used for sorting arrays in main memory. However, the space complexity is $O(N)$, which usually prohibits its use for this purpose.

Sorting sequential files (2)

Simplest method: **straight merging** on **three** tapes:

Steps:

1. Split the file into two halves (*distribution*).
2. Merge the halves creating two-element runs (*merging*)
3. Split the file into two halves.
4. Merge the halves creating four-element runs, etc.

. . .

Since the length of initial runs = 1, we have N initial runs, and the number of phases equals $\lceil \log_2 N \rceil$.

Example:

Initial file (N =8): 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 (length of runs = 1)

1st phase:

44 | 55 | 12 | 42
94 | 18 | 06 | 67

44 94 | 18 55 | 06 12 | 42 67 (length of runs = 2)

2nd phase:

44 94 | 18 55
06 12 | 42 67

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

3rd phase:

06 12 44 94
18 42 55 67

06 12 18 42 44 55 67 94 (sorted file: 1 run of length 8)

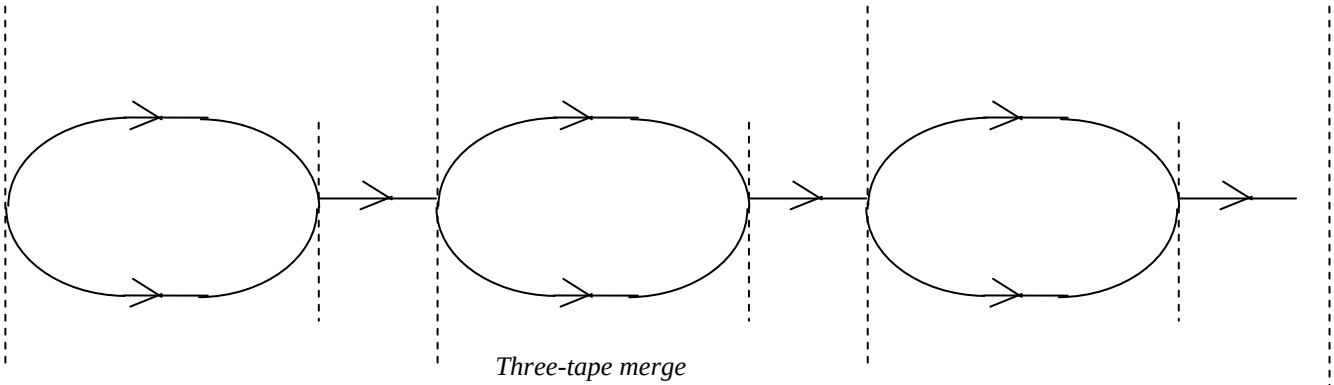
Sorting sequential files (3)

In the three-tape merge, during each phase each element is read into main memory and written back to disk twice, so the total number of reads and writes equals to

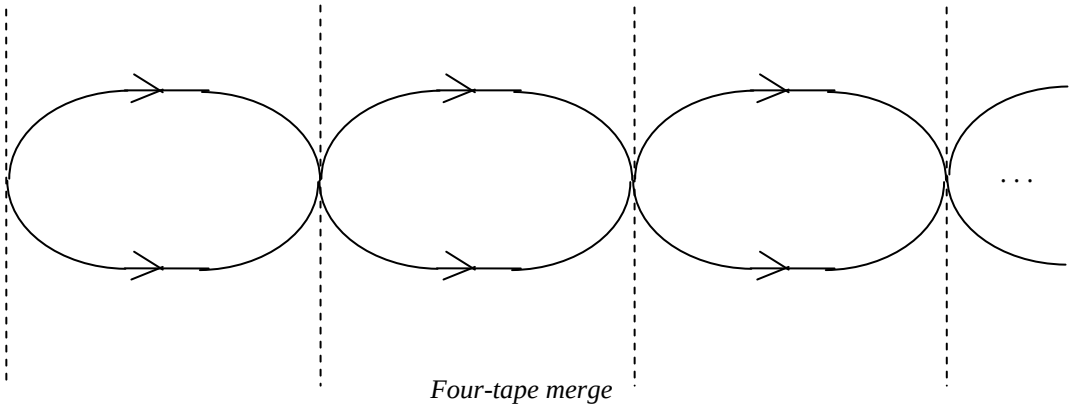
$$4N \lceil \log_2 N \rceil / b$$

(Due to the blocking factor we can read or write *b* elements at once).

Distribution and merging sub-phases can be combined into one distribution-merging phase, if we allow for an additional tape (***four-tape merge***). In this case, the number of necessary disk requests can be reduced by the factor of 2.



Three-tape merge



Four-tape merge

The price we pay is additional tape (file).

Sorting sequential files (4)

The straight merging does not try to take advantage of the initial ordering of the file. Consider, for instance, the behaviour of straight merging if the file is initially sorted.

The next method, **natural merge**, merges runs that are as long as possible.

Example: natural merge on three tapes.

Initial file - the same as in previous example (*N* =8):

44 55 | 12 42 94 | 18 | 06 67 (there are 4 runs of different lengths)

The runs are distributed *alternately* onto two tapes:

1st phase:

44 55 | 18
12 42 94 | 06 67

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

2nd phase:

12 42 44 55 94
06 18 67

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

In the worst case (file in reverse order) , the number of phases in the natural merge is the same as for straight merge, because the number of initial runs = *N*. However, in the average case the number of initial runs will be less than *N*. If we have initially *r* runs, then the maximal number of phases equals $\lceil \log_2 r \rceil$, and the number of read-write requests equals $4N \cdot \lceil \log_2 r \rceil / b$ (three tapes assumed).

It can be shown that the expected number of initial runs is equal to *N*/2, hence on the average the natural merge reduces the number of phases at least by one.

Sorting sequential files (4a)

The same example as before, but on four tapes (2+2 pattern):

Initial file - the same as in previous example (N =8):

44 55 | 12 42 94 | 18 | 06 67 (there are 4 runs of different lengths)

The runs are merged from two tapes and simultaneously distributed onto two tapes:

1st phase:

Tape 1	44 55 | 18
Tape 2	12 42 94 |06 67
Tape 3	12 42 44 55 94
Tape 4	06 18 67

2nd phase:

Tape 1	06 12 18 42 44 55 67 94	(1 run, file sorted)
Tape 2	*empty*	
Tape 3	*empty*	
Tape 4	*empty*	

In the 2+2 pattern the number of read-write requests is reduced by factor of 2:

$$2N \cdot \lceil \log_2 r \rceil / b$$

because each element in each phase is read once and written once.

Sorting sequential files (5)

The number of phases in the natural merge can be even more reduced by the effect of coalescing (joining) adjacent runs during the distribution.

Example:

Let us replace 18 by 60 in the file from the previous example:

44 55 | 12 42 94 | 60 | 06 67 (there are still 4 runs)

1st phase:

44 55 ' 60 (two adjacent runs coalesced)
12 42 94 | 06 67

12 42 44 55 60 94 | 06 67 (2 runs)

2nd phase:

12 42 44 55 60 94
06 67

06 12 42 44 55 60 67 94 (1 run, file sorted)

The effect of coalescing runs can considerably reduce the number of phases necessary to sort the file, in comparison with the straight merge.

Example:

Initial file:

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

1st phase:

10 ' 11 15 ' 16
9 12 ' 14 18 '

9 10 11 12 14 15 16 18 (1 run, file sorted)

One phase was enough to sort the file. Straight merge would require 3 phases.

Sorting sequential files (6)

- **Implementation remarks**

In the straight merge, after the distribution onto the tapes the difference between the number of runs on each tapes is at most 1. This means that only the last run on one of the tapes must possibly be merged with an empty run on the other tape. In the natural merge, due to the effect of coalescing adjacent runs, the number of runs on each tape may differ much more.

Example:

Initial file:

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

1st phase:

10 | 8 15 | 13 (this tape has 3 runs)
9 12 '14 18 (this tape has only 1 run)

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

2nd phase

9 10 12 14 18 | 13
8 15

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

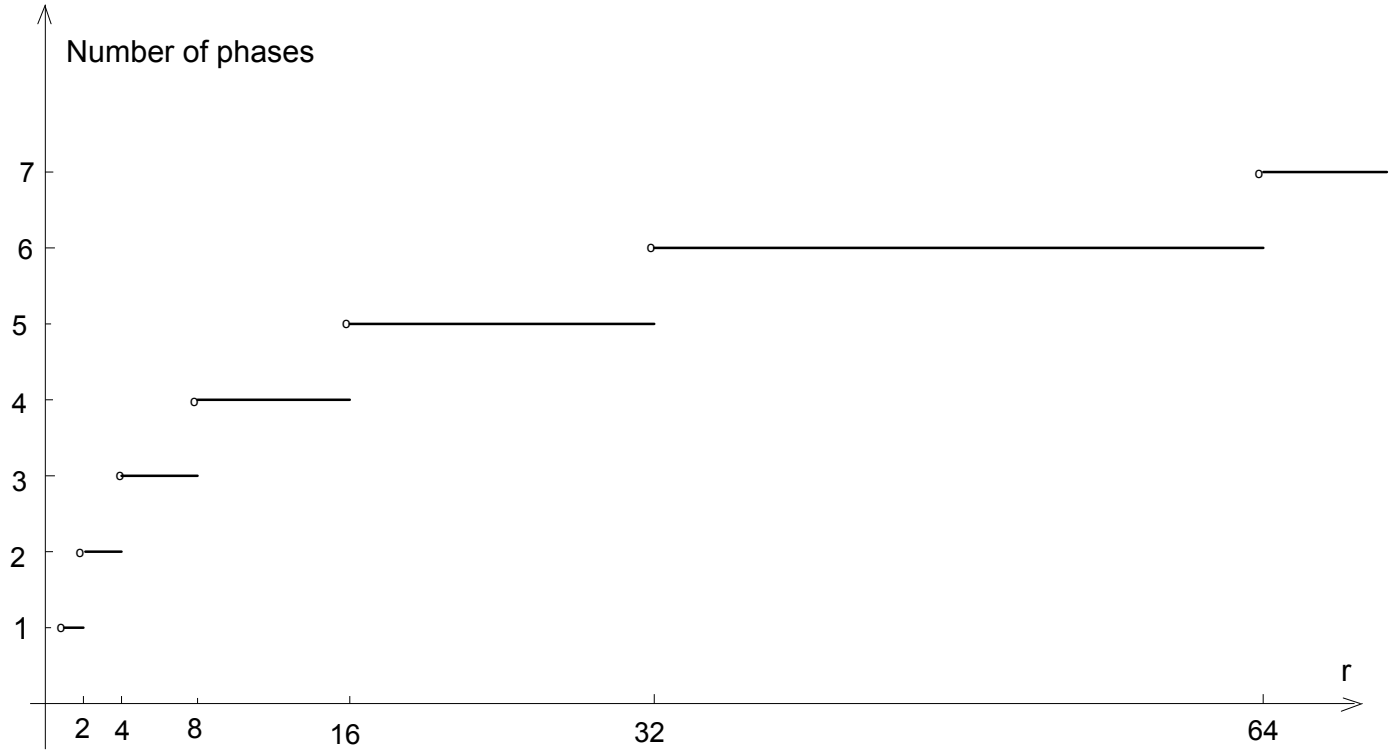
3rd phase

8 9 10 12 14 18
13

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

Sorting sequential files (7)

The maximal number of phases required to sort a file with *r* initial runs as a function of *r*:



Sorting sequential files (8)

If we use more tapes for natural merge, we can further reduce the number of phases required to sort the file.

Example:

Distribution onto 3 tapes, merging onto one tape (t = 4)
Initial file:

44 55 | 12 42 94 | 18 | 06 67 (there are 4 runs of different lengths)

1st phase:

44 55 | 06 67	(2 runs)
12 42 94	(1 run)
18	(1 run)

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

2nd phase:

12 18 42 44 55 94	(1 run)
06 67	(1 run)
-------	(no runs)

06 12 18 42 44 55 67 94 (file sorted)

It is an example of *multiway merging.* In general, if we use t+1 tapes (t for distribution and one for merging), the maximal number of phases is equal to $\lceil \log_t r \rceil$.

t + 1	No of phases
3	$\log_2 r$
4	0,63 $\log_2 r$
5	0,50 $\log_2 r$
6	0,43 $\log_2 r$

Sorting sequential files (9)

In a **polyphase merge**, each tape is alternately used for distribution and merging. Let us look at an example of polyphase merge on three tapes (file has initially r = 13 runs).

In the following examples, to make them more legible, only the numbers of runs are presented.

Example.

Initial distribution of runs: 7, 6, 0 (most uniform):

	Tape 1	**Tape 2**	**Tape 3**
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

The file has been sorted in 7 phases. Note that in each phase each element of the file is read from and written back to disk only once, so there are total of two read-write requests per element in one phase.

Let us try another initial distribution of runs: 12, 1, 0 (most non-uniform):

	Tape 1	**Tape 2**	**Tape 3**
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

In this case, we need as many as 12 phases (compare with at most 4 phases for "standard" natural merge...)

The **minimal number of phases** we obtain if the initial runs distribution conforms to the **Fibonacci sequence**: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
In our example, 13 = 5 + 8:

	Tape 1	**Tape 2**	**Tape 3**
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

5 phases were enough to sort the file of 13 runs.

It can be shown that the number of phases necessary to sort the file of *r* runs (assuming the initial "Fibonacci distribution") approximates to

$$1,45 \cdot \log_2 r$$

But note that during each phase not all records are processed.
It can be shown that each element is processed

$$1,04 \cdot \log_2 r$$

times, which gives the overall number of read-write requests approx. equal to

$$(2,08N \cdot \log_2 r + 2N) / b$$

(the component 2*N* corresponds to the read-write requests necessary for initial distribution).

Compare this figure with the corresponding one for natural merge!

Sorting sequential files (11)

How to proceed if the initial number of runs is not a Fibonacci number? We add empty (dummy) runs to one of the tapes so that the total number of runs on each tape is a Fibonacci number.

(Actually, the dummy runs do not have to be kept on the tapes; it is enough to remember the *numbers* of real and dummy runs).

Example:

Initial number of runs r = 19. We distribute them alternately onto two tapes according with increasing Fibonacci numbers:

Tape 1: 1, 1+1=2, 2+3=5, 5+6(2)=11(2)
Tape 2: 1, 1+2=3, 3+5=8

11(2) means that the Tape 1 has 11 real runs and 2 "dummy" runs.

Sorting proceeds as follows:

	Tape 1	**Tape 2**	**Tape 3**
Phase No	11(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 this way, the number of phases for r = 14, 15, ..., 21 is identical: 6.

Remark:

If we denote the numbers in Fibonacci sequence as f_1=1, f_2=1, f_3=2, f_4=3, f_5=5, etc, then for the number of initial runs equal to f_k the number of phases is equal to k -2.