CS 178: Database Systems

B. Narahari

Department of Computer Science
GWU

Chapter 1

Physical Implementation of Databases

CS 178: Database Systems

1.1 Sorted Files

- In a *sorted file*, records are ordered on some field.
- The field (or group of fields) is usually the *primary key* of the relation.
- Some DBMS's (e.g., Postgres, Oracle) use an internal tuple-id (integer) for each tuple ⇒files are typically sorted by tuple-id unless otherwise specified.
- File operations using a sorted file:

  Equality search on sort field:

  - Sorted files were devised to speed up search. ⇒use binary search on sort field ⇒if $X =\#$ blocks, search time is $O(\log X)$ instead of $O(X)$
  - In our model, this works out to $O(\log(n/p))$
  - Note: to retrieve blocks in the middle, a directory is required:

WHICH BLOCK	ACTUAL BLOCK NUM
⋮	⋮
13	4578
⋮	⋮

  - Advantages: Speed.
  - Important: analysis assumed sorted files are stored sequentially on disk
  - Disadvantages: Directory required. Insertion is difficult (see below).

  Insert a record:

  - Option 1: Make space to insert.
    - Find right place to insert a record to keep sorted order.
    - Push back remaining records to create space (and write those blocks back).
    - Write new record.
    - Advantages: Keeps sorted order.
    - Disadvantages: Insert is very expensive.
  - Option 2: Insert in temporary overflow area.
    - Insert record in overflow block(s).

    * Periodically reorganize file: sort the overflow blocks, then merge-sort with main file.
    * To search: use binary search in main file, sequential search in overflow blocks.
    * Advantages: Most inserts are fast. Overflow area can be controlled.
    * Disadvantages: Frequent changes (inserts/deletes) are time-consuming. Search may be slow if overflow area is large.

Delete a record:

- Option 1: delete record, then pack file.
    * Advantages: No space wasted.
    * Disadvantages: Time-consuming.
- Option 2: delete record, pack file occasionally.
    * Advantages: Simple. No space overhead. Wasted space can be controlled.
    * Disadvantages: 'Occasionally' time-consuming.
- Note: chaining deleted space is useless since records must be in sorted order.

Range search:

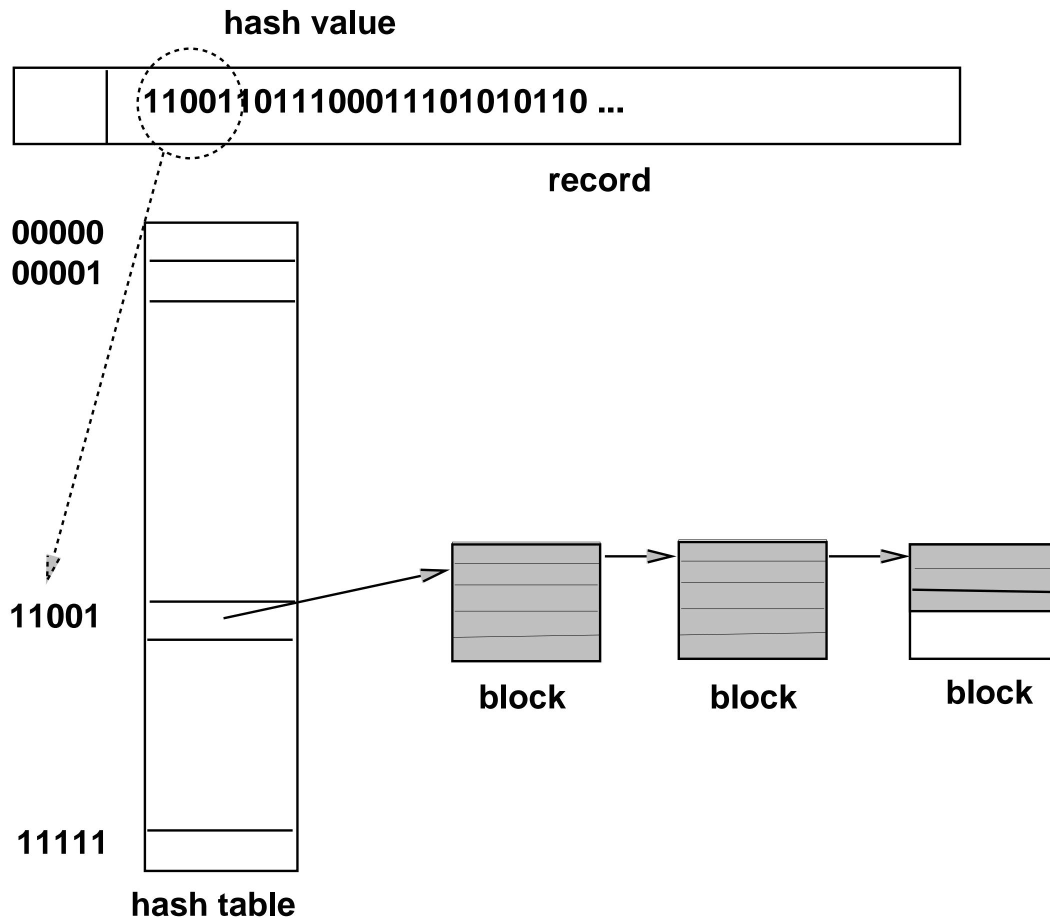
- If range search is on sort-key, find first tuple and then scan until out of range. $\Rightarrow$cannot be more efficient.
- Range search (or plain search) on any other fields requires a complete scan.

1.2 Analysis of Sorted File

- Equality search/lookup takes $O(\log n/p)$, i.e., log of number of pages in file
- since $n/p = 50,000$, this is $\log 50,000 = 16$
- insertion and deletion could result in scanning whole file

1.3 Hashed Files

- basic idea in hashing is that while domain of key is large, actual key values are much smaller
- example- if SSN is a 9 digit number, the domain is 1 billion but actual range is much smaller (250 million?)
- if Last Name is 12 character string, then domain is 12^26 but range is much smaller.
- In a *hashed file*, records are distributed among several 'buckets'.
- there is a *hash function* that takes as argument a value k and produces an integer in the range 0 to $B-1$, where B is the range of the function – also called number of *buckets*
- Key ideas in simple hashing:



– A hashing function is applied to each record
  ⇒a mapping function from the bits in the record to integers
  e.g, pick the first 5 bits in the record ⇒maps records to the integers $0, 1, 2, \ldots, 31$.
– Each integer is associated with a *bucket*: the integer is the *bucket number*.
– Initially, a block is assigned to each bucket
  ⇒as records get added to the bucket they are placed in the block.

– Later, when the first block assigned to a bucket fills up, additional blocks are used.
– The hash table itself is a (small) collection of blocks.
– Usually: try to pick hashing function so that records spread evenly across buckets.
– Some buckets could have long overflow chains ⇒resembles heap file.
– Dynamic hashing methods handle non-uniformity (to be covered later).

- File operations using a hashed file:

  Insert a record:

  – Compute hash value and insert in appropriate bucket.

  – New disk block may be needed if all disk blocks in bucket are full.

  Delete a record:

  – Remove record from disk block.

  – Return disk block to disk manager if empty.

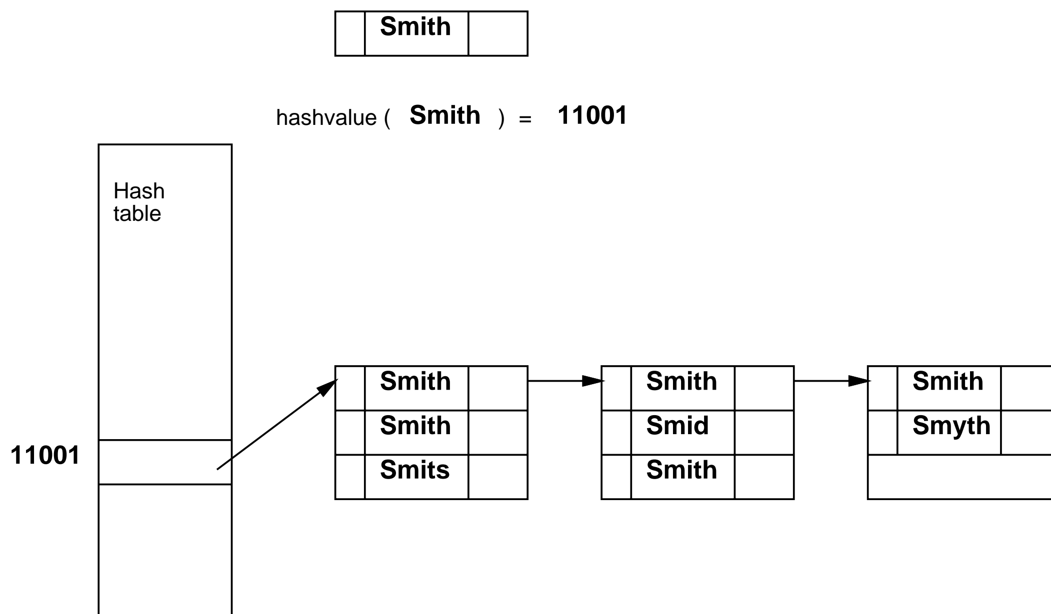
  – No need to keep track of deleted space.

  Equality search:

  – Compute hash value of equality argument.

  – Search disk blocks in corresponding bucket. ⇒if there are few blocks per bucket, few disk accesses are needed.

  – Example:



  Other searches and scans:

  – Scan all blocks in hashed file.

1.4 Time Analysis of Hash Files

- Assume, for simplicity, that records are "evenly" distributed among B buckets
- this implies n/B records per bucket
- number of disk pages/block in each bucket is therefore $\lceil n/Bp \rceil$
- Lookup/Search: compute hash function and search in appropriate bucket, takes $1/2(n/Bp)$ (on average go through half the pages in bucket) and worst case n/Bp
- Insert and Delete: takes one disk access; find last page in bucket and insert into that page.
- For the example with 1,000,000 records and 1000 buckets we have
  - lookup takes $1/2(n/2Bp) = 25$ on average and worst case of 50

1.5 File Structures: A Summary

- Key ideas in the three file structures:

  Suppose b =number blocks in file.

  Heap files:

  * Plus: Simple. Fast insertion.
  * Minus: Any search is slow ($O(b)$).
  * Best if file scans are common or insertions are frequent.

  Sorted files:

  * Plus: Search on sort-key is fast ($O(\log b)$). Range search is fast (on sort-key)
  * Minus: Insertion is slow. Any other search is slow.
  * Best if range search is needed often.

  Hashed files:

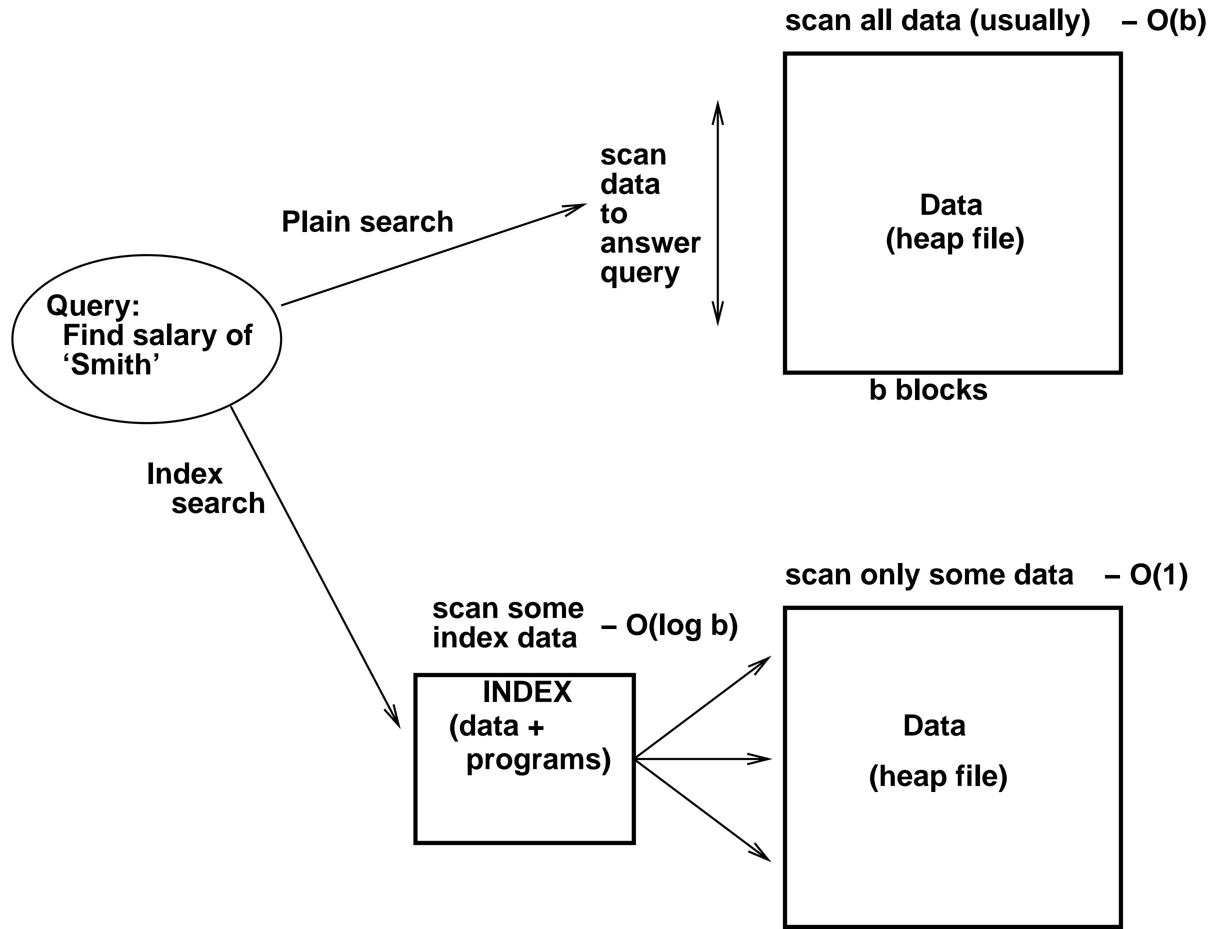
  * Plus: Insertion or search on hash-key is fast. ($O(1)$ if properly distributed).
  * Minus: Complex. Any other search is slow. Space overhead. Unbalanced buckets degrade performance. Range search is slow.
  * Best if equality search is needed on hash-key.

- In practice:

  * Sort files are rarely used, since indices such as B^+-trees provide for fast search, insert and range search.
  * Hash files are rarely used for direct storage. However they are often used as temporary files during *join* computations.
  * Unless otherwise specified, heap files are most often used.
  * Most systems allow the creation of indices to speed up file access.

1.6 What is an Index?

– Consider a relation such as EMP (NAME, SSN, SALARY).

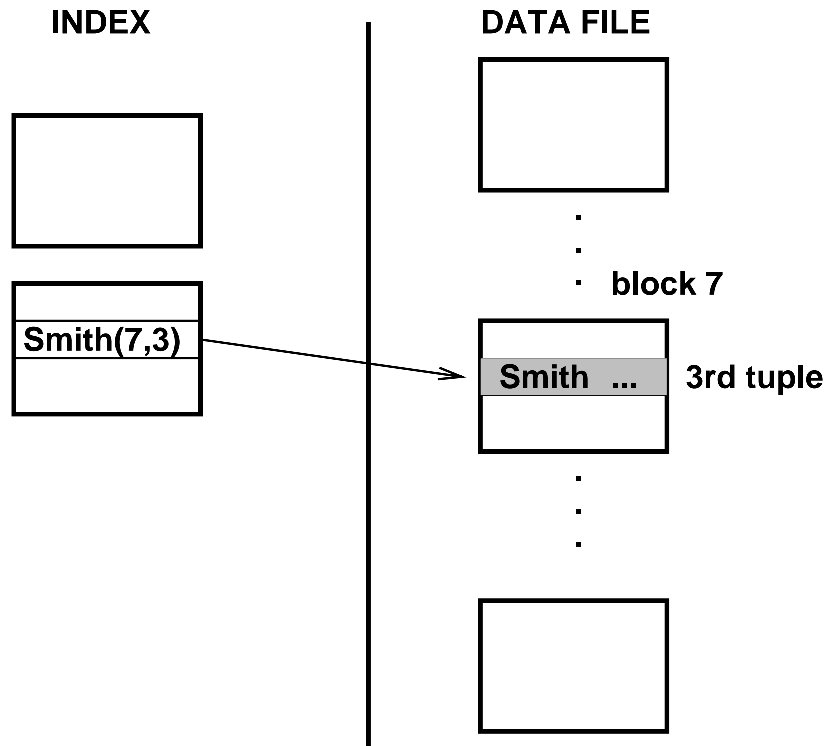


– An index is:

1. Some data extracted (copied) from the data file placed in a useful data structure (leaving the data file intact).
2. Programs that manipulate the index data.

– Key idea: narrow down the search to a few pieces of the original data file.

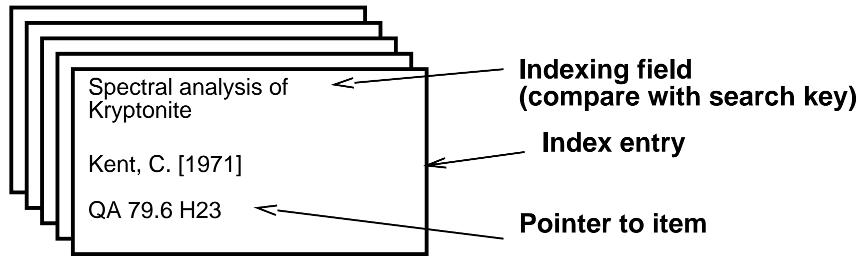
– Indices usually contain 'pointers' to data in the data file.



– A *search-key* is a value that defines a search:

  * In the query "Find the salary of Smith", the string `'Smith'` is the search-key.
  * In the query "Find all employees with a salary greater than 10000", the integer `10000` is the search-key.
  * A search-key is the value used in searching.
  * Unfortunately, the word "key" is being overloaded here.
  * Sometimes, a search-key is also a *key* for the associated relation.

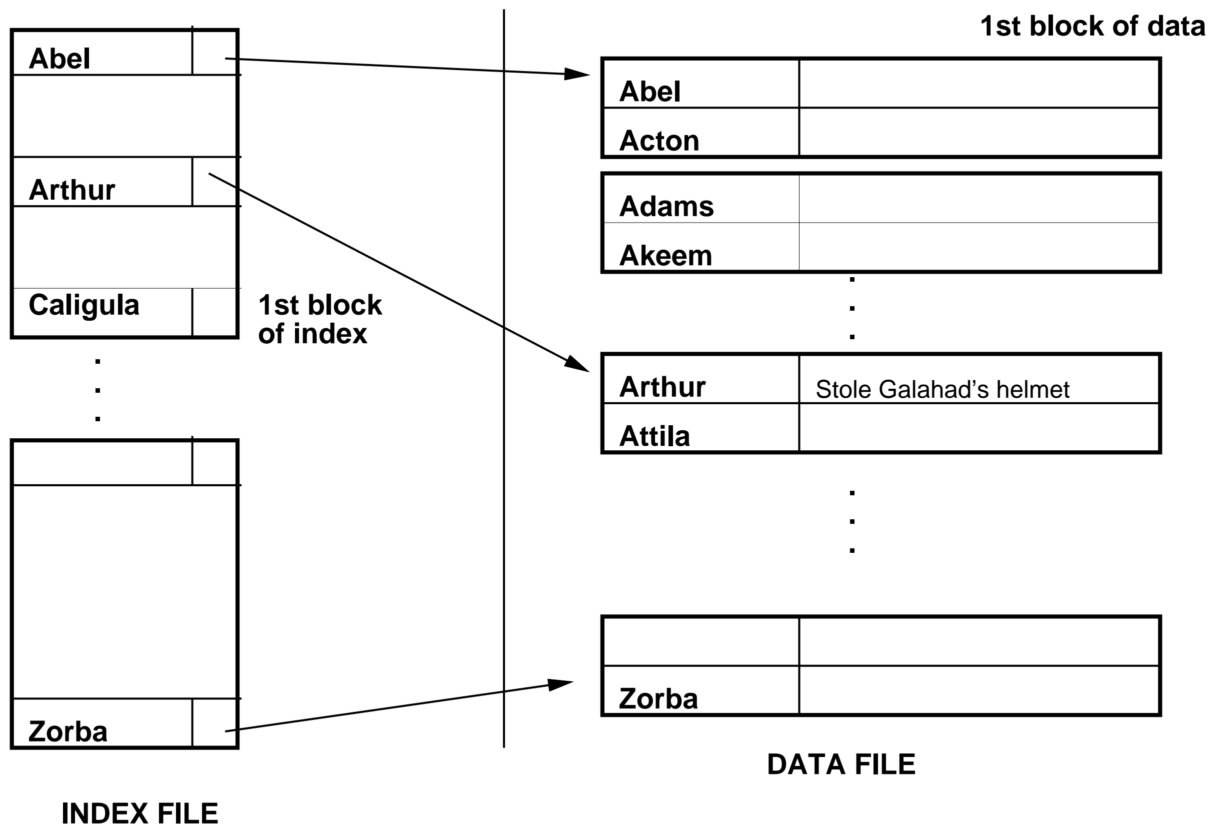
– The main idea behind an index can be found in a library card catalog:



Since each index entry (card) is *smaller* than the data (book) itself, searching the index (card catalog) is faster than search the data file (library) itself.

1.7 A Simple Index

- Suppose we have a data file and a search key.
  - Create a file of *index records*.
  - Each index record will have a possible search-key value and a pointer to the appropriate data record.
  - A 'pointer' to a data record is really the *block number* of the block in which the data resides, and the *tuple number* within the block.
- Example: consider the following relation: llll **create table** BLACKMAIL (
  NAME char(20),
  DIRTY_SECRET char(480));
  Suppose the block size is 1024 bytes:
  - 500 bytes per data record ⇒2 data records per block.
  - Suppose a pointer requires 5 bytes (4 bytes for block number and 1 byte for tuple number) ⇒25 bytes per index record ⇒40 index records per block.



Consider the query: "Find Arthur's dirty secret."

Without index:

* Suppose "Arthur" is the 10th record in the data file ⇒5 blocks of data file must be read.

Using an index:

* Since "Arthur" is 10th entry ⇒occurs in block 1 of index ⇒1 block of index file, 1 block of data file ⇒2 blocks to obtain final record

This may not seem to be much of an improvement. Consider instead the query: find information about "Ulysses".

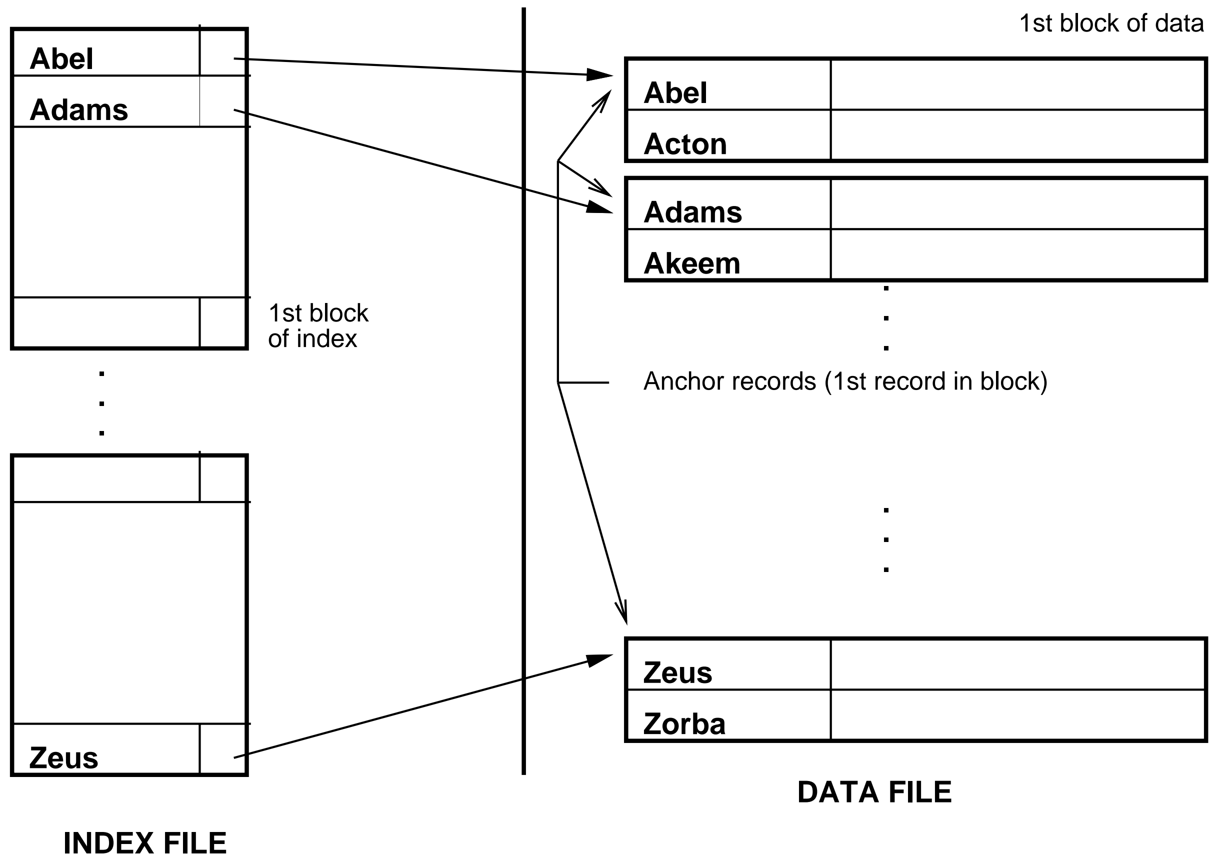
* Suppose "Ulysses" occurs as 30,000-th record ⇒15,000 blocks accessed in data file scan (no index). ⇒30,000/40 = 750 blocks accessed in index file ⇒751 block accesses using index.

– The impact of sorting:

* Suppose lll b = number of data file blocks
  b_i = number of index file blocks

* The index file is typically sorted ⇒binary search can be used.

- * If data file is sorted, binary search can be used in direct method. ⇒ratio of performance is $O(\log b)/O(\log b_i)$.
  - * If data file is not sorted (most common case), then ratio is $O(b)/O(\log b_i)$.
- – If the data file is sorted, the index can be made smaller by using *anchor* records (first record in a block):



Since only the correct block needs to be located, knowing that a data item lies between two successive anchor record key-values is enough to determine the block.
In the example:

* 2 data records per block ⇒15,000 anchor records. ⇒15,000 index entries ⇒15,000/40 = 375 index file blocks

⇒example of a *non-dense* index.

1.8 Types of Indices

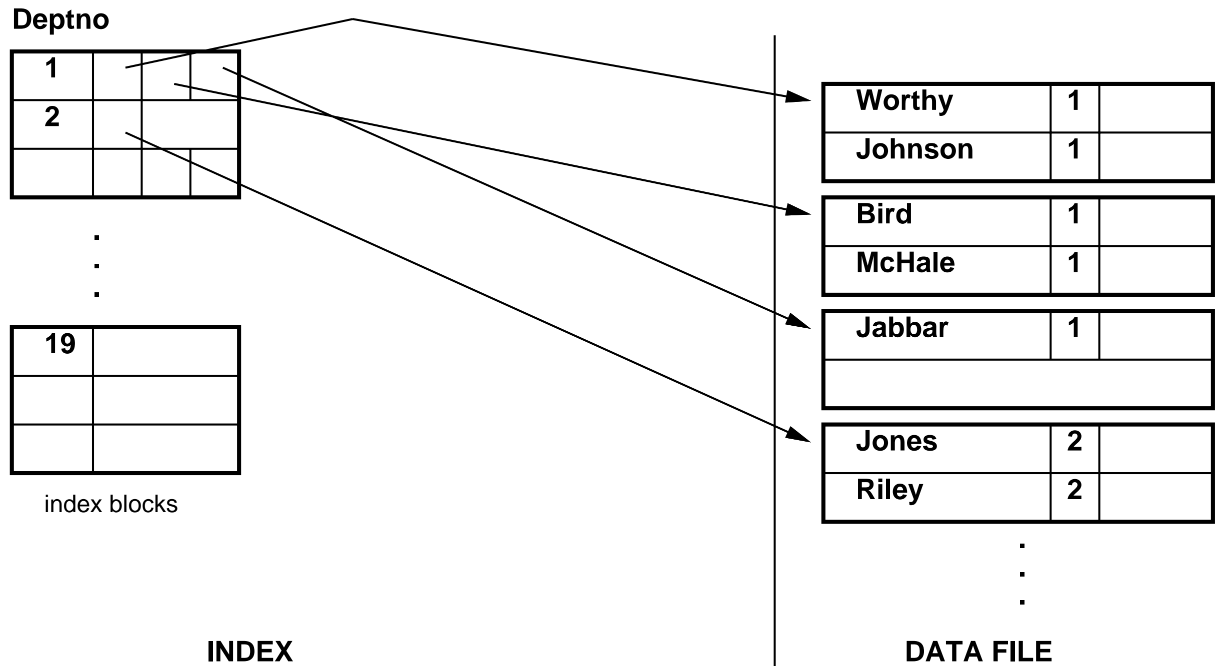
- Note: "Indexes" and "Indices" are both valid.
- An index that has one entry for each data record is called a *dense index*. Otherwise, a *non-dense index* or *sparse index*.
- The set of attributes corresponding to the search-key is called the *indexing field*.

  For example, one can create an index for the combination (NAME, SSN). Then, the bits corresponding to (NAME,SSN) in each record constitute the *indexing field*.
- A *primary index* is an index such that the indexing field includes the primary key of the data file.

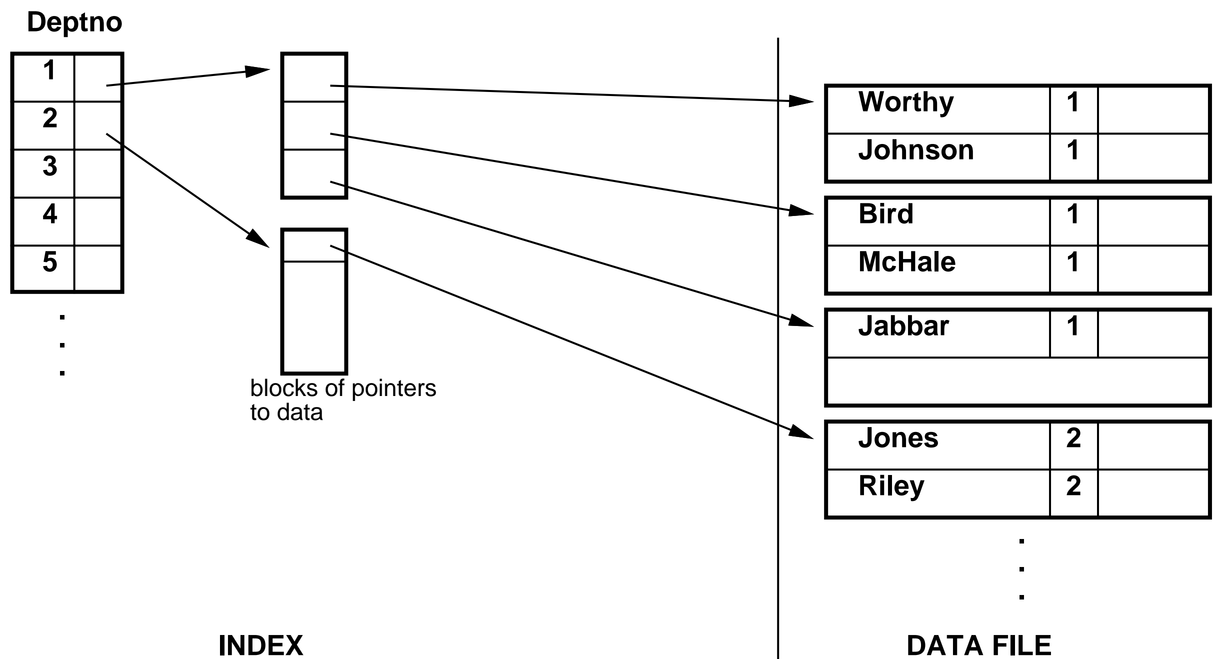
  Note: in some books, a *primary index* additionally requires the data file to be sorted.
- A *secondary index* is any index that is not a primary index.

  For example, consider lSTUDENT (NAME, SSN, STUDENT_ID, ADDR). Then, an index on SSN is a primary index. An index on STUDENT_ID is a secondary index.
- when field used to organize file is same (or close) to search key in index, it is called a *clustering index*; else we have a *non-clustering index*. Also, clustering index sometimes defined as secondary index on a non-key set of attributes.

  For example, consider lEMP (NAME, DEPTNO, SSN, ADDR) An index can be created on DEPTNO by *clustering* together data records having the same DEPTNO:



Sometimes, if there are too many values, separate blocks of pointers (block numbers) are used:

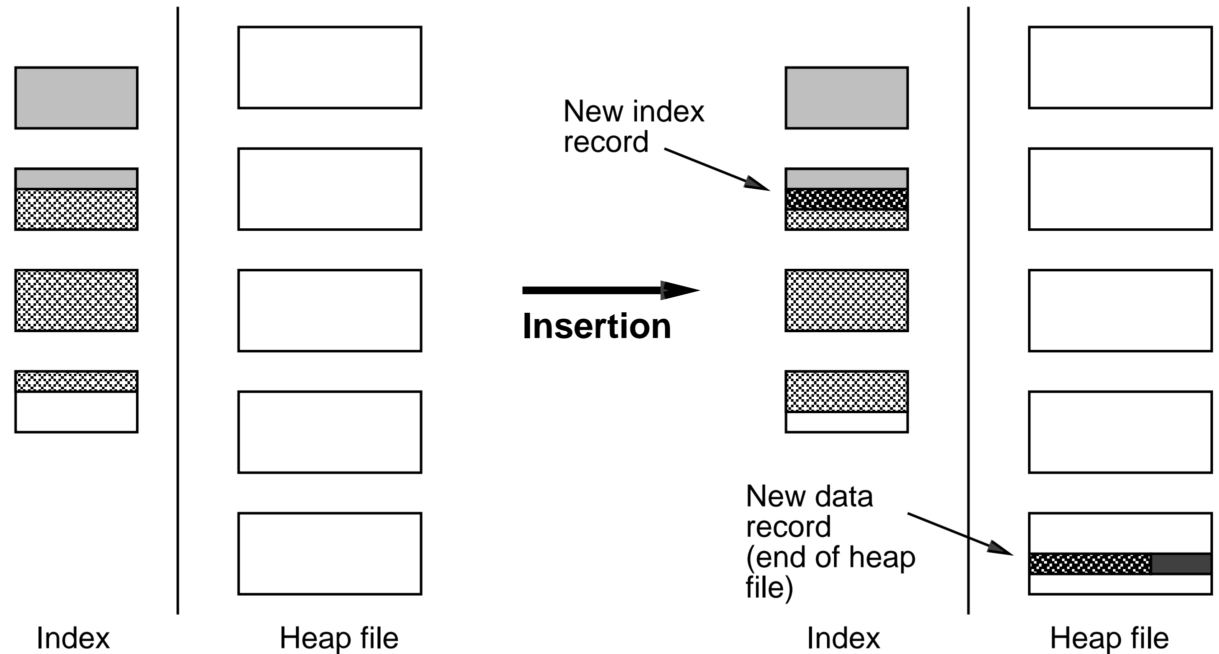


– Insertion and deletion:

Insertion:

∗ Index file is sorted ⇒new index record needs to be inserted in proper place.

* If data file is a heap file, insertion of data record is easy.



* For a sorted data file, data record needs to be in correct place. ⇒either squeeze record in or use overflow blocks. If a non-dense index is used ⇒anchor records may change in re-packing ⇒large part of index may need to be re-built.
* Overflow blocks can be used for index file as well ⇒periodic re-organization needed.
* In both index and data files, space can be initially reserved for future inserts.
* Insertion into a clustering index is easy: data file is a heap file. New block pointers are simply added to collection of pointers.

Deletion:

* Deletion from data file follows earlier techniques.
* If anchor record is deleted, new anchor record must be designated and written into index entry.
* Several options for deleting index record:
    * Delete and pack ⇒time-consuming, but no space wastage
    * Delete and leave space ⇒could waste space and later slow down access.

– Primary vs. Secondary indices:

* Recall: a primary index is an index on a primary key.
* A secondary index is an index on any other set of attributes.

* If the data file is sorted by primary key, the primary index can be made *non-dense* ⇒fewer index entries ⇒binary search on index is faster.
* However, sometimes improvement using a secondary index can be greater, if the data file is sorted on primary key.
  · for data file of b blocks and index file of b_i blocks:
  · ratio of performance for primary index is $O(\log b)/O(\log b_i)$.
  · ratio of performance for secondary index is $O(b)/O(\log b_i)$.

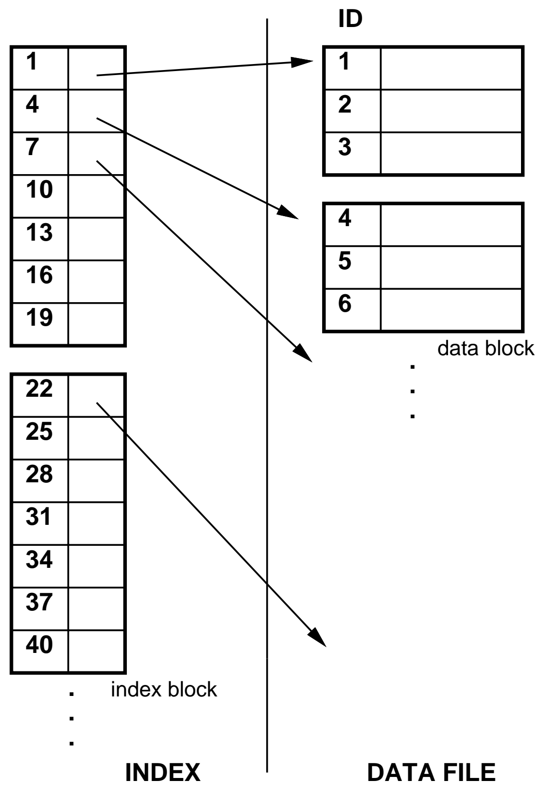
  where $b = \#$ data blocks and $b_i = \#$ index blocks.
* Example:

1.9 Analysis of Index Files

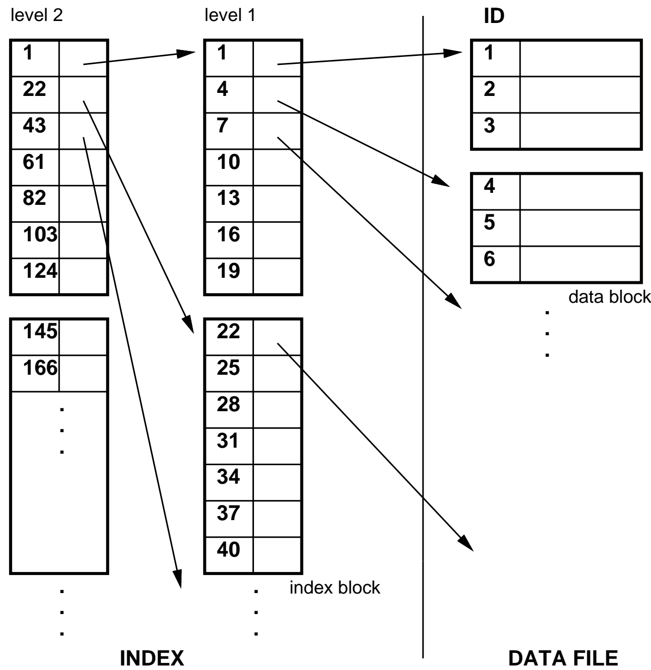
- Using our running example
- assume key field uses 20 bytes and pointer to block takes 4 bytes
- we need 50,000 records in index file (one for each data block)
- each index record has 24 bytes
- therefore, $4096/24 = 170$ index records fit on block;
- this does not leave room for unused bits etc.
- so assume we fit 150 index records per disk block
- size of index file is $50,000/150 = 334$ blocks
- binary search on index file takes $\log 334 = 9$
- to retrieve data item we need another access to data block, so 10 accesses
- performance improvement over sorted data file is $(\log 50,000)/(\log 334) = 16/9$

1.10 Multilevel Indices

· Consider an index on STUDENT (ID, NAME, ADDR):



Typically: binary search is used on index file.
Key observation: index file is itself a sorted "data" file. ⇒can create a primary index on the index:



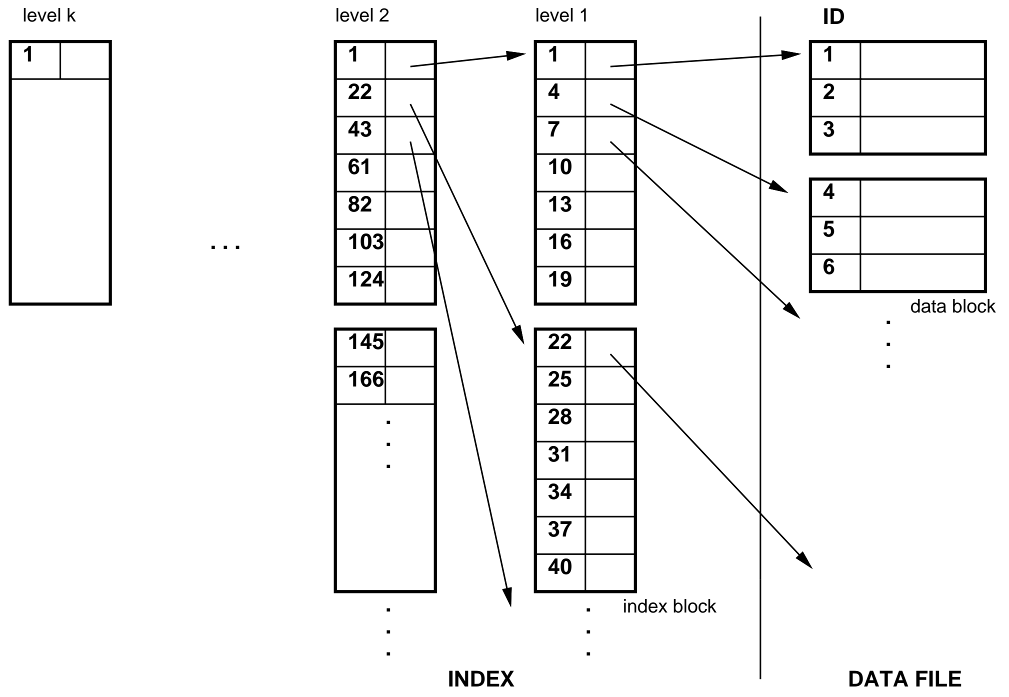
In this case:

- The second index file (level-2) is an index on the first one (level-1).
- Level-2 contains the anchor values of level-1.
- This is called a *multilevel index*.
- Suppose level-1 has b_i blocks and an index block contains f index records: $\Rightarrow$One level-2 entry for each of the b_i blocks $\Rightarrow b_i$ records in level-2 $\Rightarrow \frac{b_i}{f}$ blocks in level-2 $\Rightarrow$level-2 search cost is $\lceil \log_2 \frac{b_i}{f} \rceil$ $\Rightarrow$total search cost is: $\lceil \log_2 \frac{b_i}{f} \rceil + 2$

  (1 block in level-1, 1 block from data file). $\Rightarrow$search time is reduced.
- What about a level-3? $\Rightarrow \frac{b_i}{f}$ blocks at level-2 $\Rightarrow \frac{b_i}{f}$ records at level-3 $\Rightarrow \frac{b_i}{f} \div f$ blocks at level-3

  $\vdots$

  $\Rightarrow \frac{b_i}{f^{k-1}}$ blocks at level-k.
- When do we stop? $\Rightarrow$when level-k has only *one* block $\Rightarrow \frac{b_i}{f^{k-1}} = 1$ $\Rightarrow k = 1 + \log_f b_i$.


level k
level 2
level 1
ID
1
1
22
43
61
82
103
124
145
166
1
4
7
10
13
16
19
22
25
28
31
34
37
40
index block
1
2
3
4
5
6
data block
INDEX
DATA FILE

1.11 Multi-Level Index: Example

- using our running example
- we have one level index file of 50,000 records and 334 index blocks
- if we create a second level index:
- 334 entries, and we need $334/150 = 3$ records.
- search will take $\log 3 = 2$ to search level 2 index
- find appropriate block at level 1- one more access
- find data block pointer in the record at level 1 index, and fetch data block, one more access
- total is 4 disk accesses

1.12 Multi-Level Indexing: Why

- What is the search cost with a *full* multilevel index? $\Rightarrow$1 block per level plus data block $\Rightarrow (k+1) = 2 + \log_f b_i$.
- Is it worth the effort?
- Cost with one-level index: $1 + \log_2 b_i$.
- Cost with multi-level index: $2 + \log_f b_i$.
- If $f \gg 2 \Rightarrow 2 + \log_f b_i \ll 1 + \log_2 b_i$.
- Example: $b_i = 4000$, $f = 100 \Rightarrow 1 + \log_2 b_i = 12$ and $2 + \log_{100} b_i = 3$ $\Rightarrow$four times faster.
- How much extra space? $\Rightarrow O(b_i)$ additional blocks.
- What about insertions and deletions?
  Insertions and deletions are problematic.

  Typical options for **insert**:
- Reorganize entire index with each insert: $\Rightarrow$expensive operation but keeps search efficient.
- Leave some space in each (index and data) block for future inserts $\Rightarrow$potential wastage of space $\Rightarrow$slower search.
- Place new data and index records in overflow area $\Rightarrow$slower search.

  Delete options are similar to those seen earlier.
- Note:
- Additional levels can be built on any index file, since level-1 is sorted (even for clustered and secondary indices).
- A multilevel index is sometimes called a ISAM (Indexed Sequential Access Method) by IBM-types.
- Multilevel indices were used in practice *before* the arrival of B-trees.
- Some ISAM products are still available.

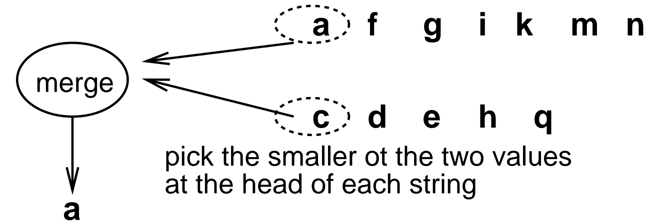
1.13 External Sorting

- Popular sorting techniques:
- Insertion sort.
- Heapsort.
- Mergesort.
- Quicksort.

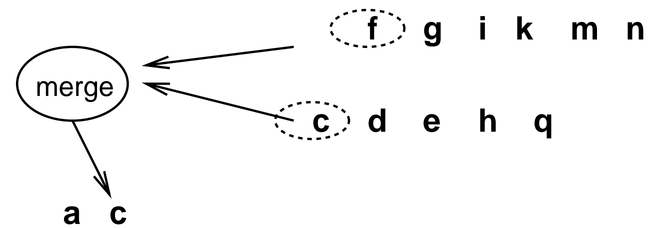
  These are all *internal* sorting methods ⇒data is in memory.
- In an *external* sorting problem:
- Data is too large to fit into main memory.
- Data must be sorted in pieces.
- Data is usually a heapfile of records.
- Desired end result: a sorted file (sorted on some key).
- Key ideas in *external sorting*:
- Individual blocks can be easily sorted ⇒read them in and use an internal sorting method.
- Create *runs*:
- A *run* is a group of sorted blocks in sort order (a sorted piece of a file).
- Runs can be created by merging data from several blocks in memory.
- Merge shorter runs into longer runs.
- Finally, merge runs into final sorted result.

1.14 Binary Merge-Sort

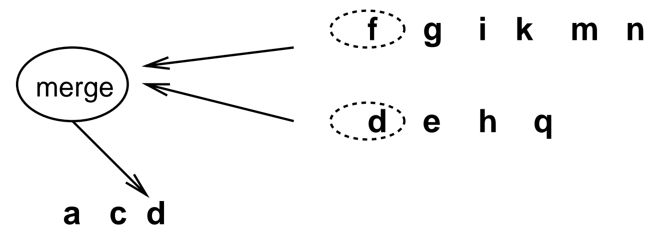
· The basic idea in merging can be conceptually explained as follows:
e.g., Merge the two strings 'afgikmn' and 'cdehq' in alphabetical order
Step 1



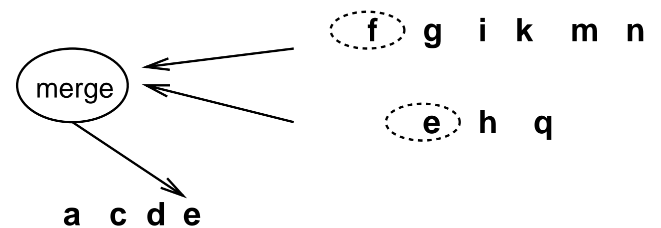
Step 2



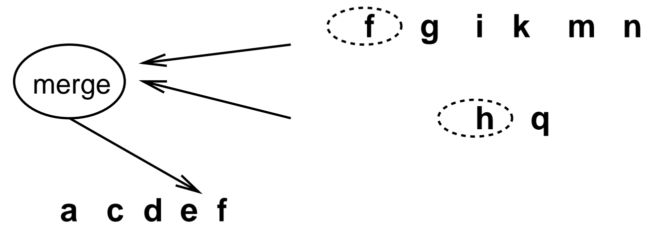
Step 3



Step 4

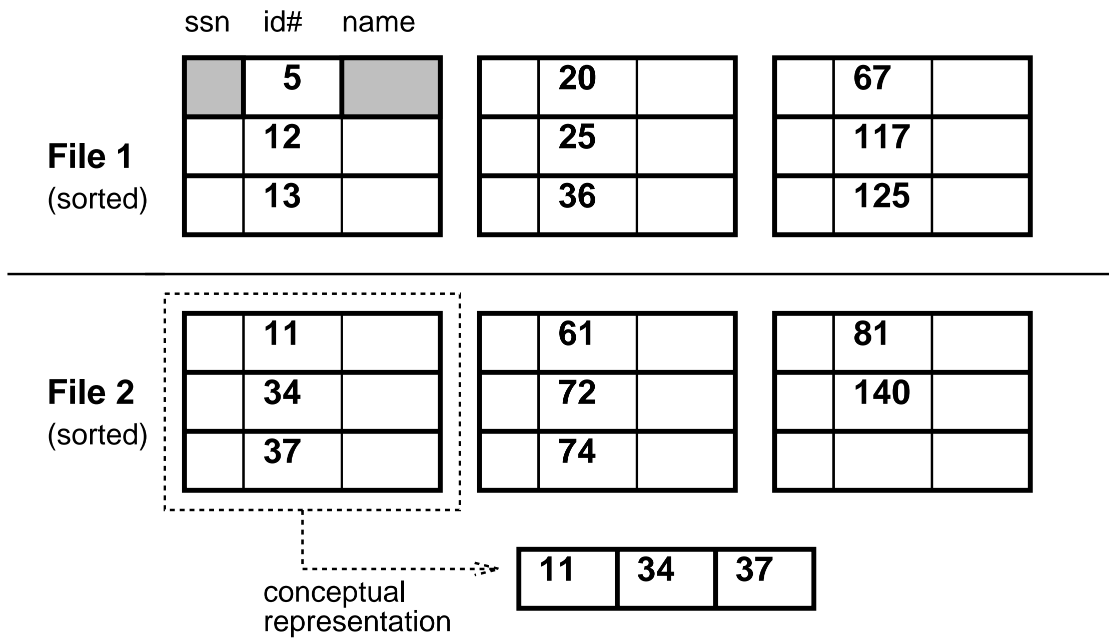


Step 5

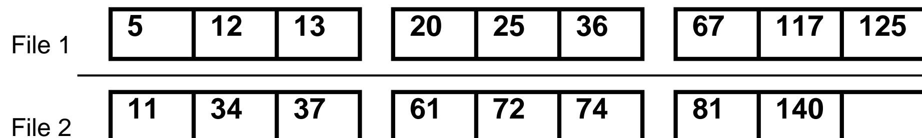


... and so on.

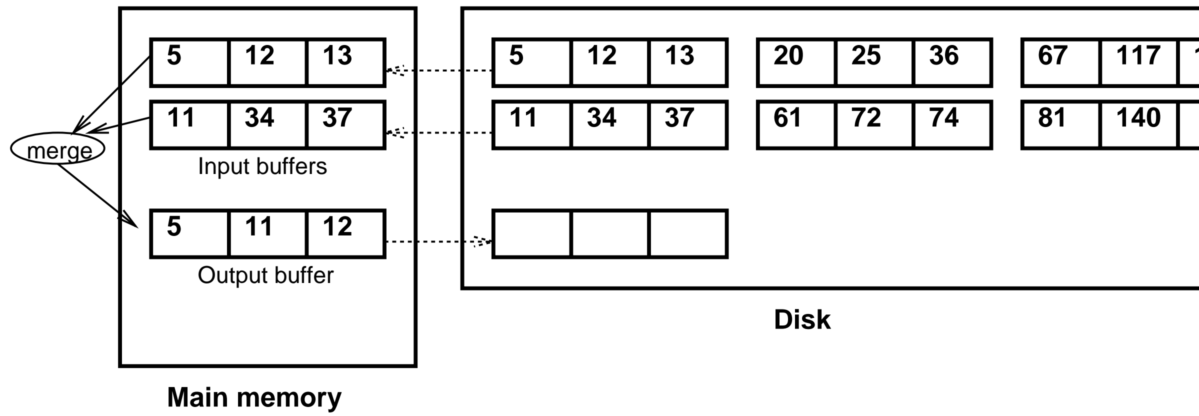
- Now suppose we have two sorted files (runs) that need to be merged: Suppose record structure is (SSN, ID#, NAME) and we are sorting by ID#:



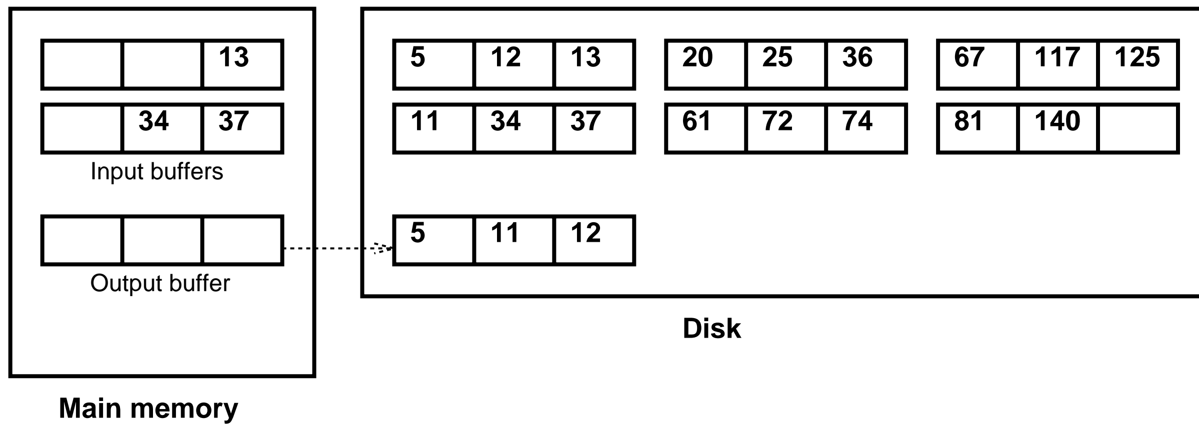
Conceptually,



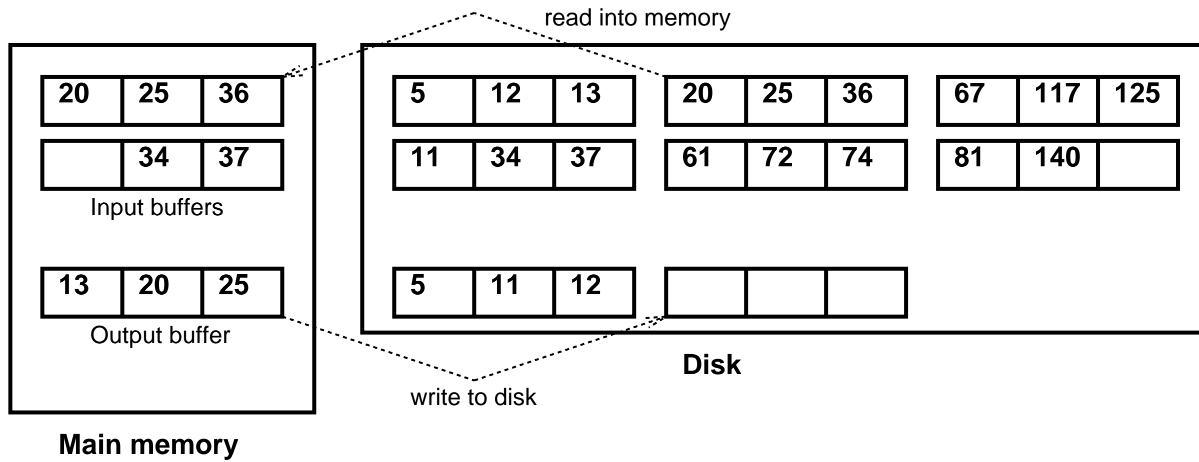
To merge these runs, we bring in two blocks at a time into memory:



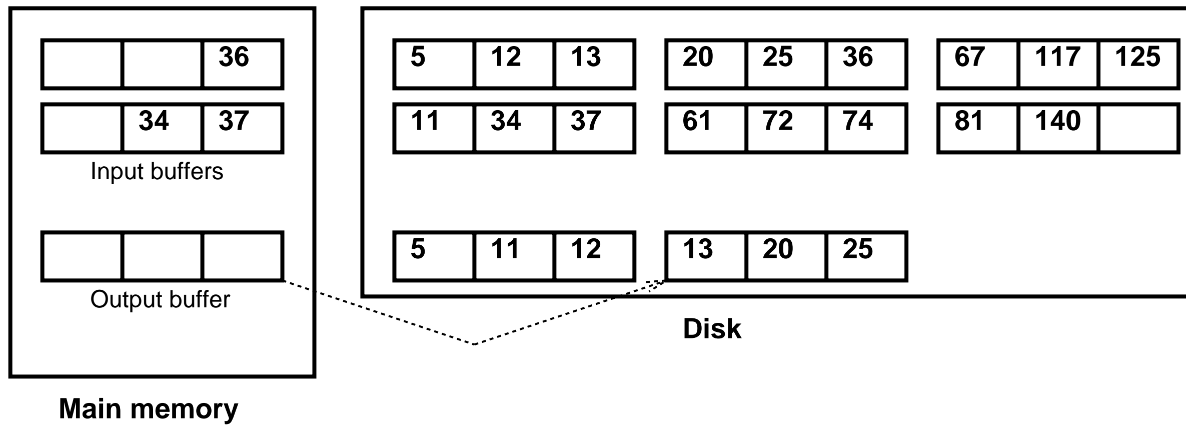
Write output block:



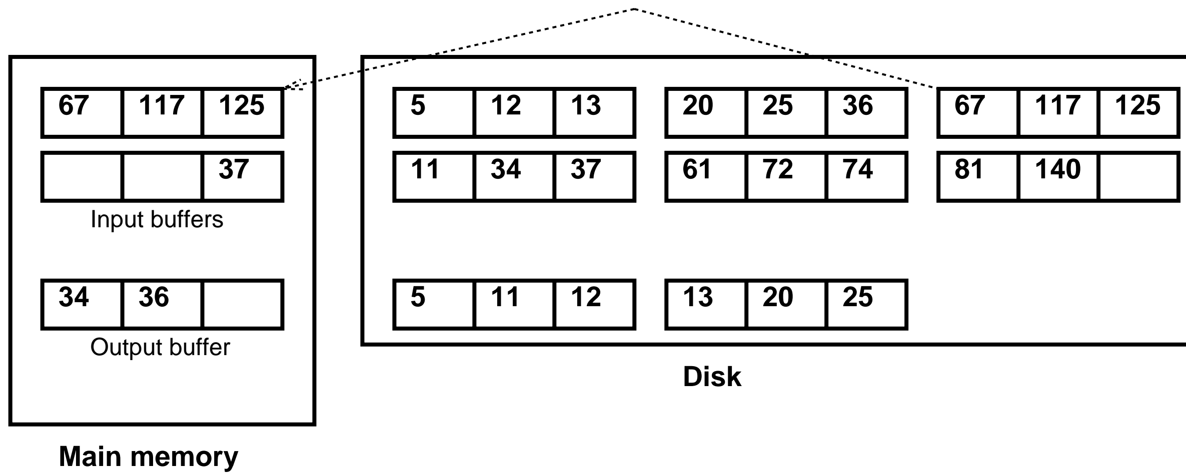
After '13' is in the output buffer, we must read next block from File 1:



Write next output block:

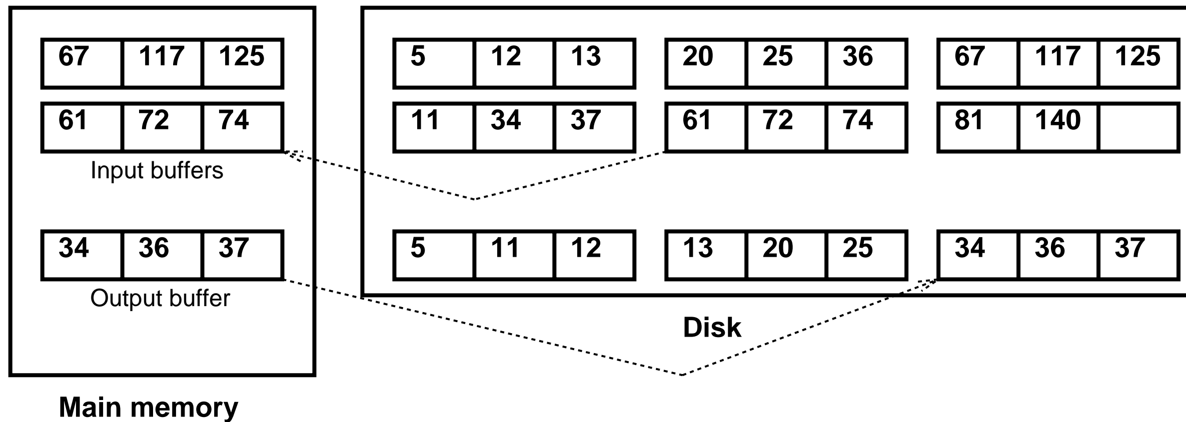


After ‘36’ is copied to output buffer, read next File 1 block:



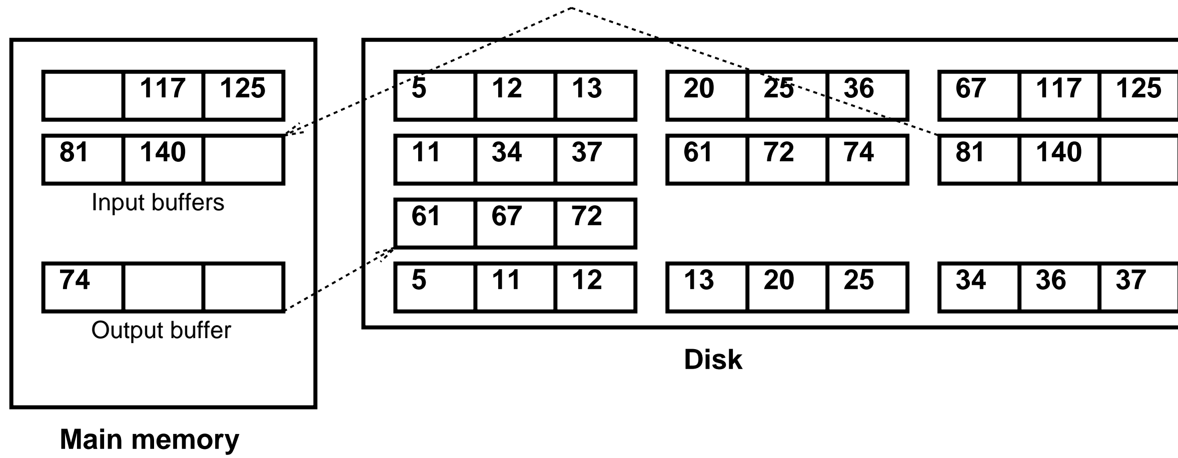
Next:

- Copy over ‘37’.
- Write current output block.
- Read next File 2 block.

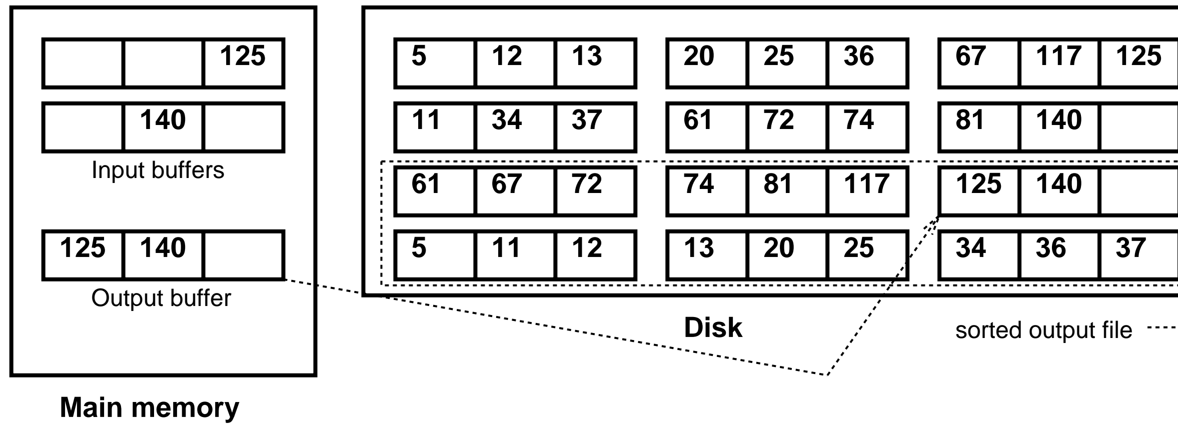


Next:

- Create next output block: '61-67-72'.
- Write output block.
- Copy '74' over to output buffer ⇒must read next File 2 block.



Continue in this fashion until finally:



- What does this have to do with sorting a single file?
  Key idea:
- Break up file into smaller (sorted) runs.
- Merge runs until a single file emerges.

· Example:
Unsorted file:

61	36	5	34	11	72	12	125	81	67	140	37	78	117	

First sort individual blocks:

5	36	61	11	34	72	12	81	125	37	67	140	78	117	

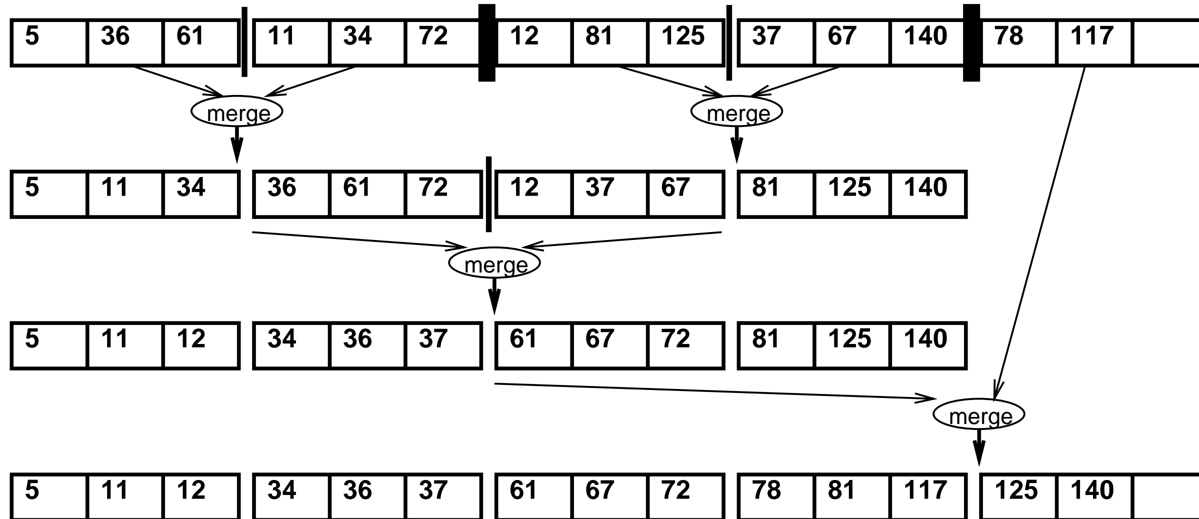
Next, repeatedly apply binary merges:

· Divide file into groups of two blocks ⇒3 groups.

· Sort each group using binary merge. ⇒3 sorted groups (runs).

· Divide runs into groups of two runs ⇒2 groups.

· Sort group using binary merge. ⇒final result.



· How many steps (phases) does it take to sort a file of n blocks?
For example, suppose $n = 400$:

· Phase 0: sort individual blocks in file.

· Phase 1: Create groups of 2 blocks and merge within groups. ⇒200 groups (2 blocks each) ⇒200 runs after merging (2 blocks each).

· Phase 2: Group 200 runs into groups of 2 runs, then merge. ⇒100 groups (2 runs each, of size 2 blocks) ⇒100 runs (4 blocks each) after merging.

· Phase 3: Group and merge: ⇒50 groups (2 runs each, of size 4 blocks) ⇒50 runs (8 blocks each) after merging.

· Phase 4: 25 runs.

- Phase 5: 13 runs (integral number of runs).
- $\vdots$
- Phase 8: 2 runs.
- Phase 9: 1 run $\Rightarrow$one sorted file.

  How many phases? $\Rightarrow$10 ($= 1 + 9$). $\Rightarrow$In general, $1 + \lceil \log_2 n \rceil$ phases.

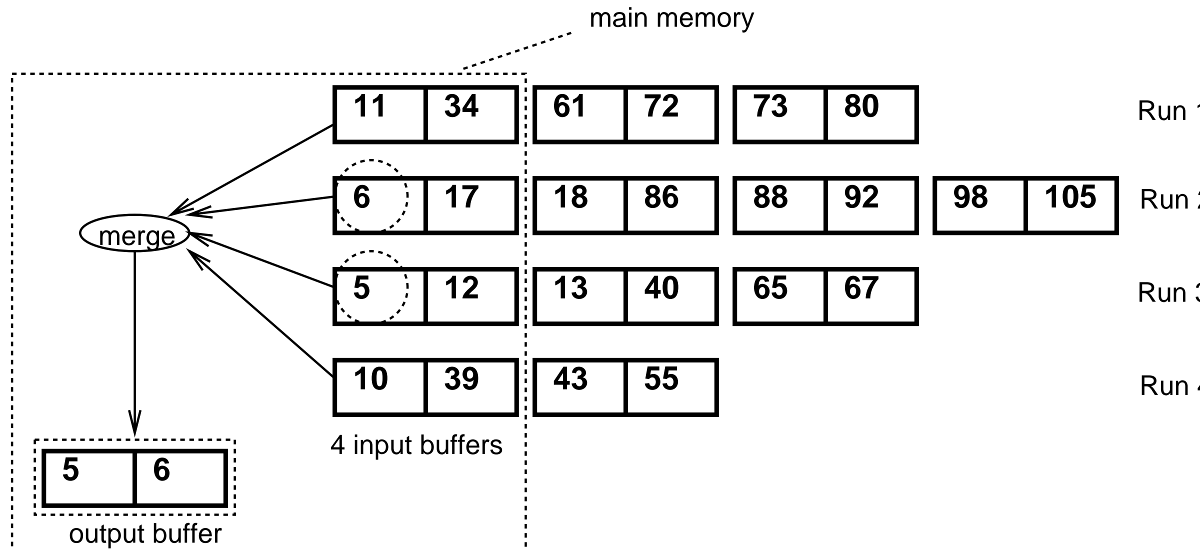
  NOTE: we can avoid Phase 0 (sorting block contents during Phase 1).

  In each phase: all blocks are read once, written once $\Rightarrow 2n\lceil \log_2 n \rceil$ block accesses.
- How much memory is needed? $\Rightarrow$2 blocks for input, 1 block for output $\Rightarrow$only 3 blocks!

  Q: Can we do better by using more memory?

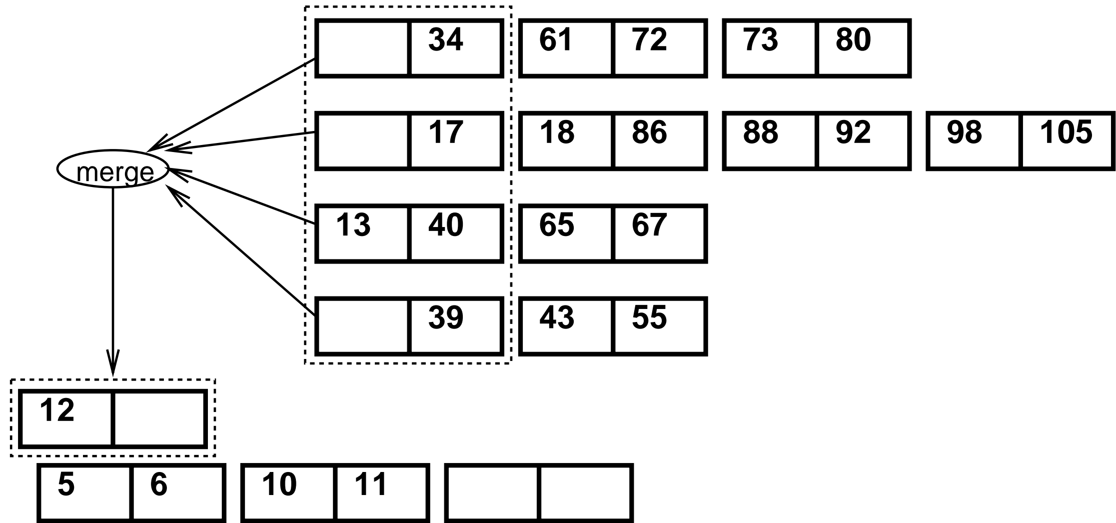
1.15 M-way Merge Sort

- A binary merge is easily generalized to an M-way merge:
- Merge M input streams.
- Use M input blocks and 1 output block.
- Example
- 2 records per block.
- M=4 runs. ⇒4 input buffers, 1 output buffer.



Next:

- Write '5-6' block.
- Copy '10' and '11' to output buffer.
- Write '10-11' block.
- Copy '12' to output buffer.
- Read next block from Run 3.



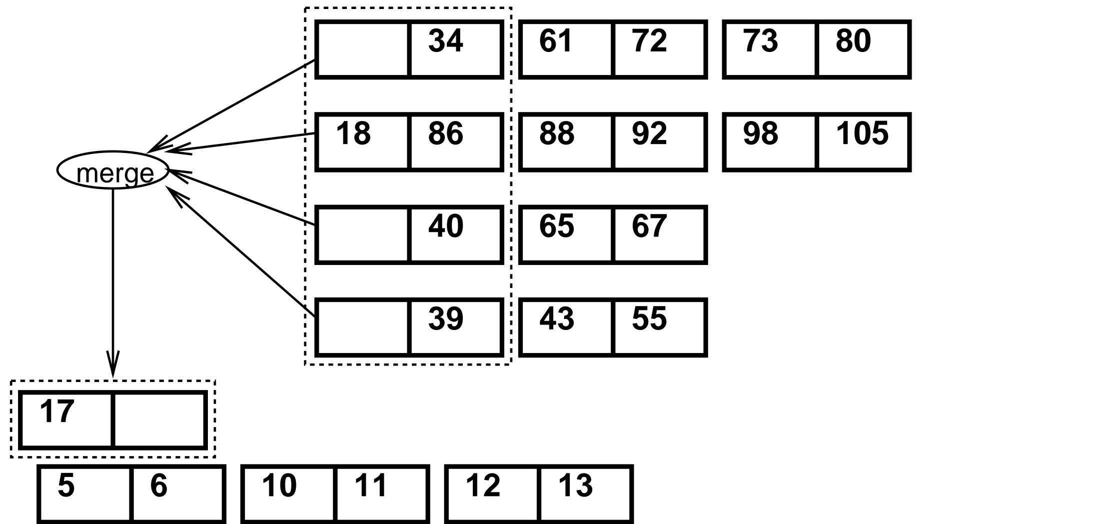
Next:

· Copy '13' over to output buffer.

· Write output buffer ('12-13' block).

· Copy '17' to output buffer.

· Read next block from Run 2.



...and so on, until all runs are exhausted.

· Using an M-way merge to sort a file:
Consider a file with n blocks.

· Divide file into groups of M blocks each.

· Merge each group into a sorted run $\Rightarrow \lceil \frac{n}{M} \rceil$ runs.

· Create groups of M runs.

- Merge each group into a single run...
- ... and so on, until a single sorted file remains.

For example, suppose $n = 400$, $M = 4$:

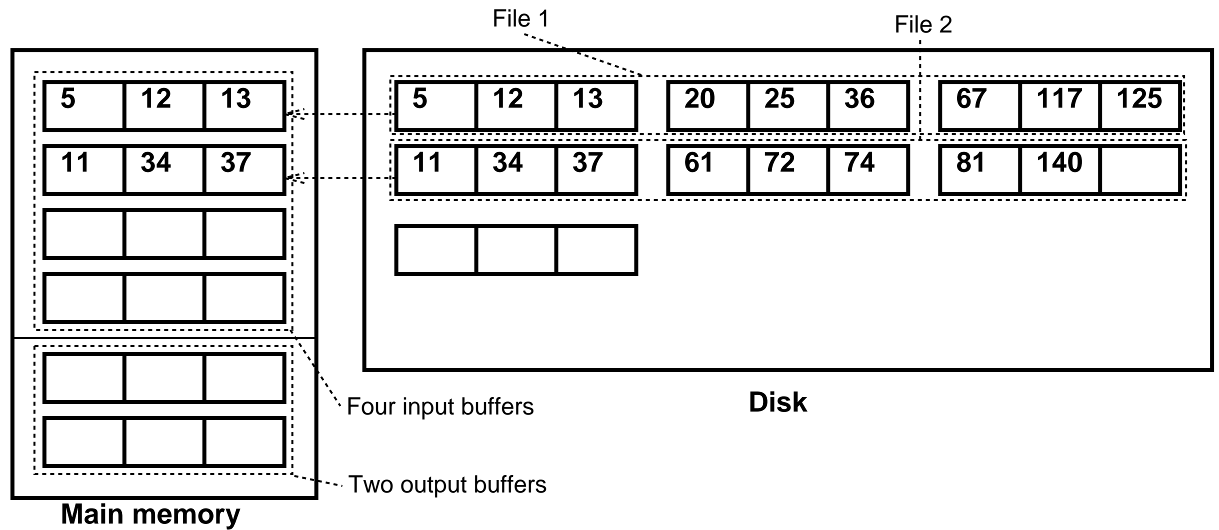
- Phase 1: Create groups of 4 blocks ⇒100 groups ⇒100 runs (of 4 blocks each) after merging.
- Phase 2: Create groups of 4 runs ⇒25 groups ⇒25 runs (of 16 blocks each) after merging.
- Phase 3: 7 runs.
- Phase 4: 2 runs.
- Phase 5: 1 run (sorted file).

In general: $\lceil \log_M n \rceil$ phases. ⇒$2n\lceil \log_M n \rceil$ block I/O's.
Example: 200 Mb file, 1 Mb memory, 2K blocksize. ⇒500 blocks of memory, 50000 file blocks ⇒499-way merge ⇒2 phases ⇒$2 \times 2 \times 50000$ block accesses ⇒200,000 block accesses.

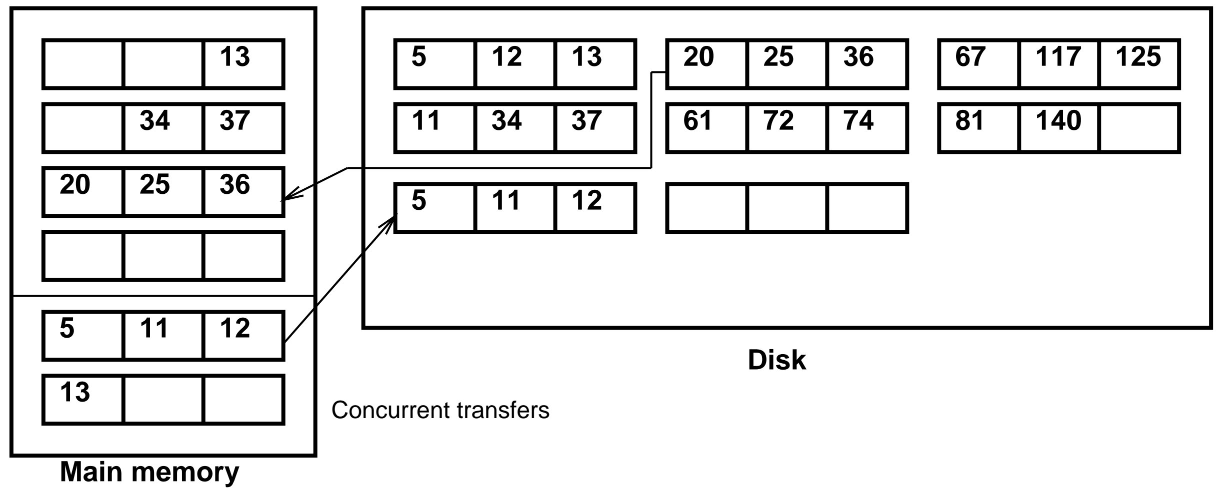
* Parallelism in I/O:
  - Suppose multiple disks are available ⇒can read and write simultaneously.
  - Use 2 sets of input buffers and 2 sets of output buffers.
  - If $M+1$ buffers are available, choose K such that

$$\begin{aligned} 2 + 2K &\leq M + 1 \\ K &\leq \frac{M-1}{2} \end{aligned}$$

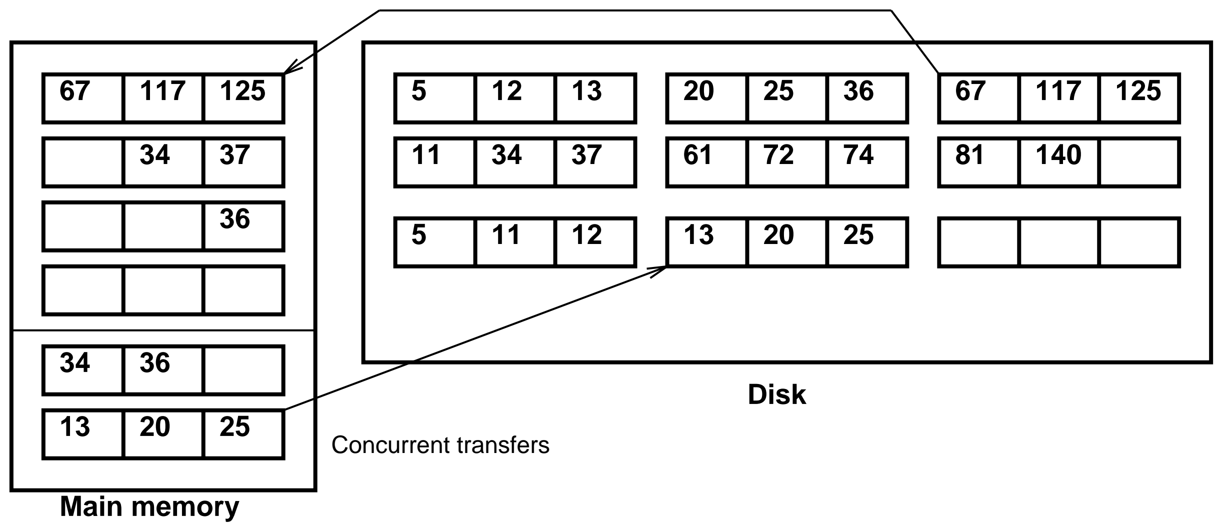
    ⇒at best a K-way merge.
  - Example: $M = 6 \Rightarrow K = 2$.
* Example:
  Consider an example with $K = 2$. A block is read from each run.



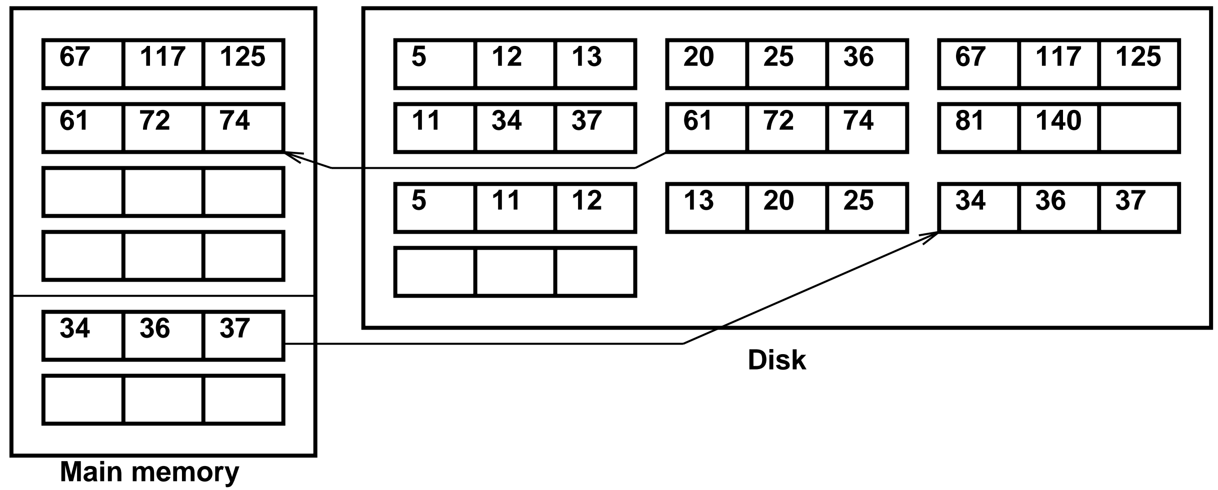
Next, while the first output block is being written, the next input block is read:



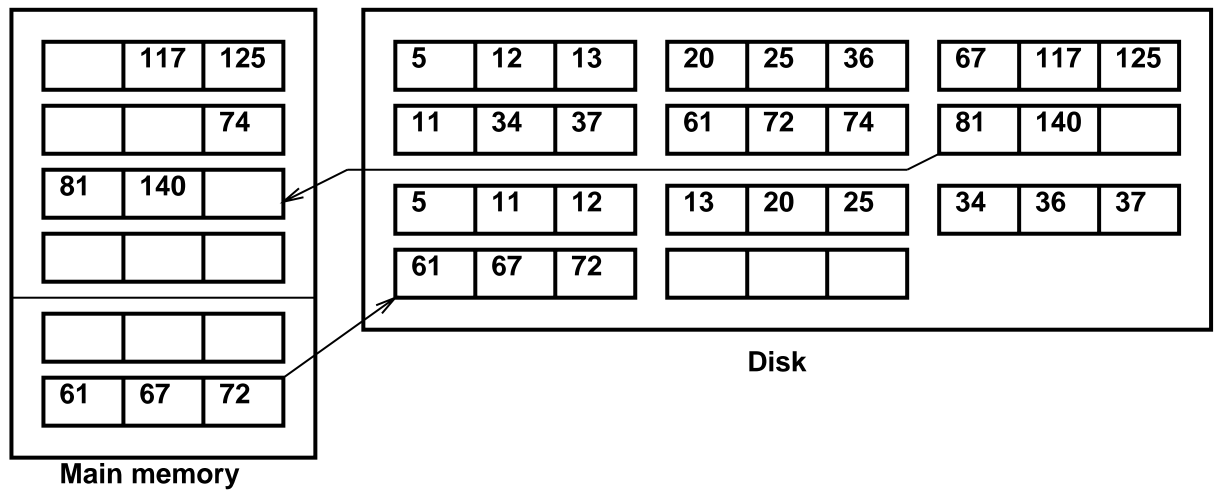
Write the ‘13-20-25’ block while reading in the ‘67-117-125’ block.



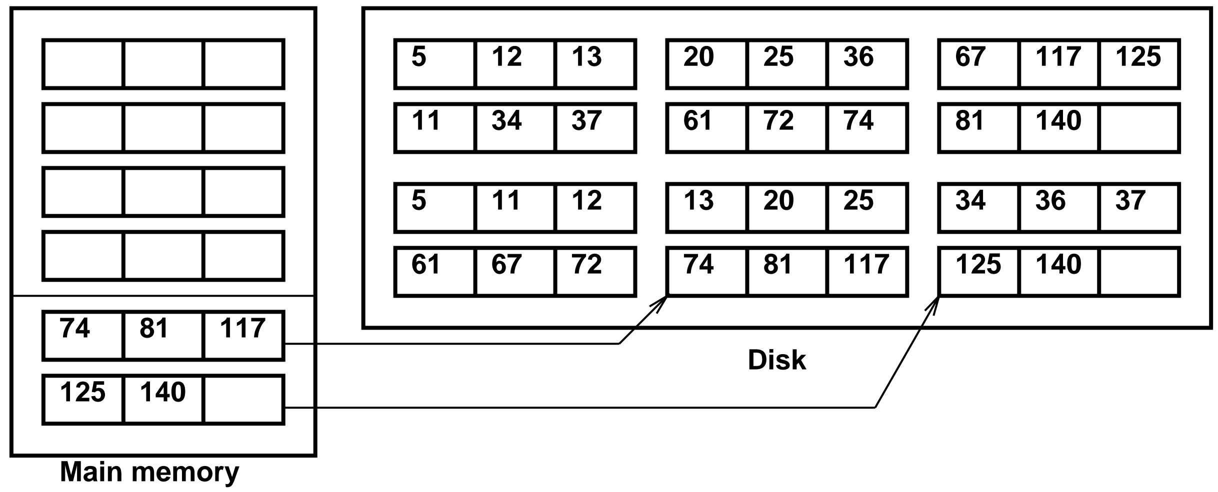
Write the ‘34-36-37’ block while reading in the ‘61-72-74’ block:



Write the '61-67-72' block while reading in the '81-140' block:



Finally,


5
12
13
20
25
36
67
117
125
11
34
37
61
72
74
81
140
5
11
12
13
20
25
34
36
37
61
67
72
74
81
117
125
140
74
81
117
125
140
Disk
Main memory