Secondary Storage

- External Sort
- B-tree Index
- Other Indices

Secondary Storage: *Basics*

- **Sequential Access vs. Direct Access (*Random Access*)**
  - Block access → organize files as user-defined blocks
- **File manger in OS supports …**
  - ***Cluster***: a number of contiguous sectors
  - Once a cluster has been found on a disk, all sectors in that cluster can be accessed without an additional seek.
  - If a file consists entirely of contiguous clusters, the seeking time is minimized.
  - As the number of "*non-contiguous*" clusters in a file increases, the file becomes more spread out on the disk, and the amount of seeking necessary increases.

Secondary Storage: *A Journey of A Byte*

What happens when write(textfile, 'P', 1) ***is executed?***

Part that takes place *in* memory:

- **Call OS to oversee this operation**

- **Call File manager (*A part of OS*)**
  - Check whether the operation is permitted
    - **the file is open, the type of access is allowed, …**
  - Locate the physical location where the byte will be stored (*Drive, Cylinder, Track & Sector*)
  - Make sure that the sector to locate the 'P' is already in a system I/O Buffer and deposit 'P' into it
  - Call I/O processor to send the sector back to disk

Secondary Storage: *A Journey of A Byte*

Part that takes place *outside* of memory:

- **I/O Processor**
  - Wait for an external data path to become available (CPU is faster…)
    - ***Direct Memory Access* Input/Output**
- **Disk Controller**
  - I/O Processor asks the disk controller whether the disk drive is available for writing
  - Disk Controller instructs the disk drive to move its read/write head to the right track (seek time) and then wait for the desired sector (latency time)
  - Disk spins to right location and 'P' is written (transfer time)

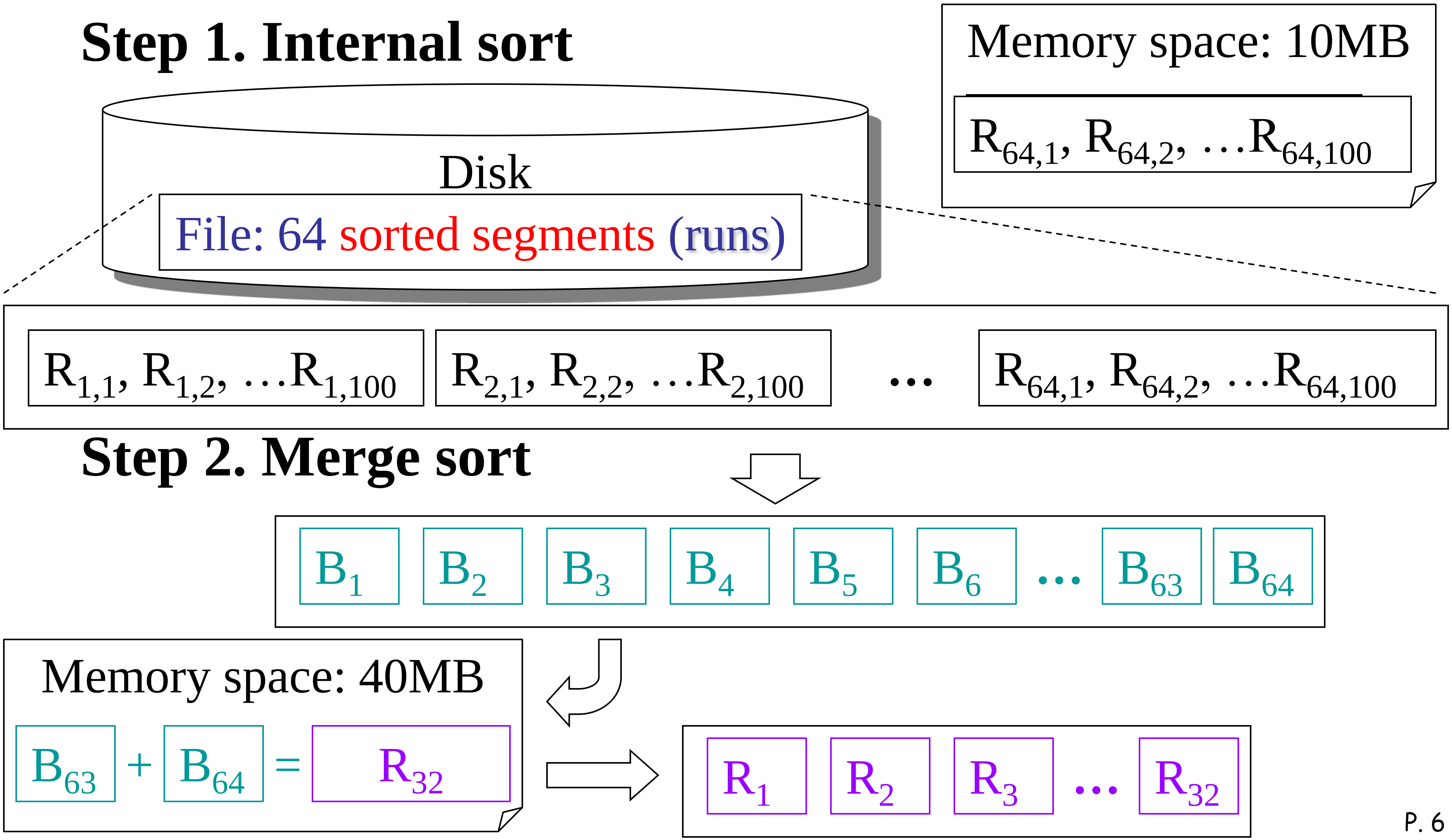
Secondary Storage: *External Sort*

- **Internal Sort: main memory**
- **External Sort: secondary storage + main memory**
- **Example**
  - How to sort 6,400 student records? (if each record takes 100KB, we need 640MB in total)
  - Can we use very less space to do the same job?
    - **The records are divided into 64 blocks**
    - **Each block keeps only 100 student records (10MB)**

    Step 1. ***Internal sort on each block***

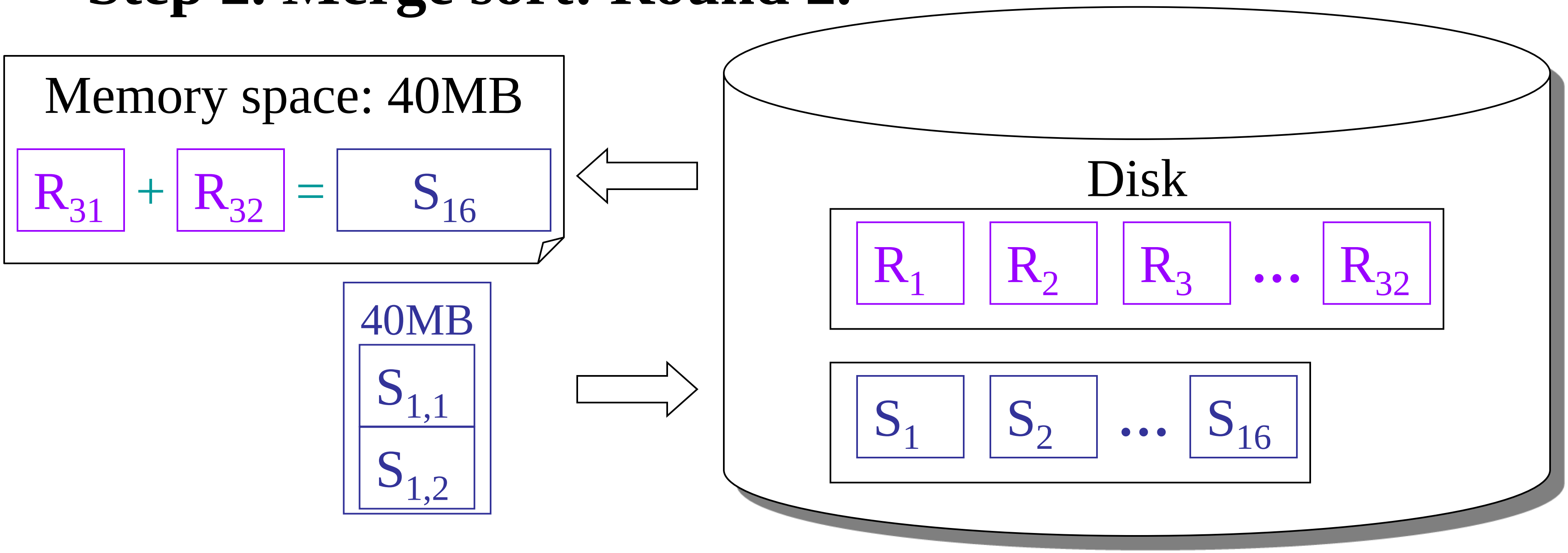
    Step 2. (external) ***Merge sort***

Secondary Storage: *External Sort*

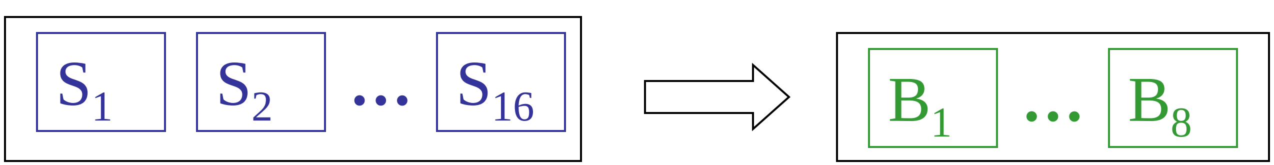


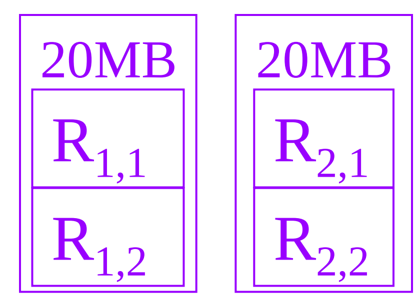
Secondary Storage: *External Sort*

Step 2. Merge sort: Round 2.



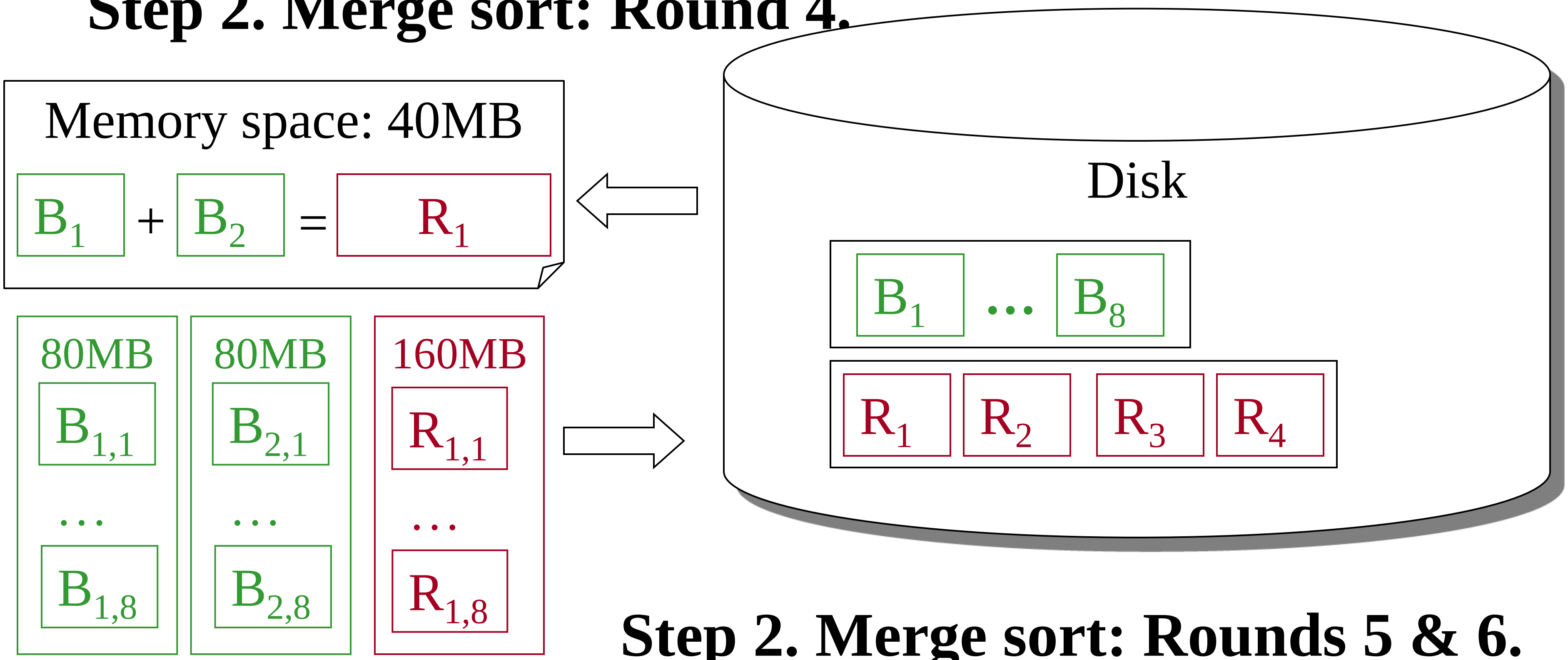
Step 2. Merge sort: Round 3.



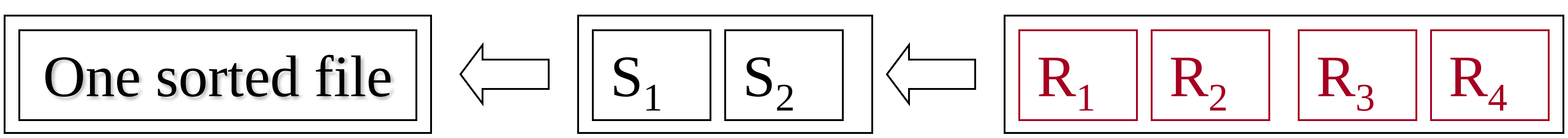


Secondary Storage: *External Sort*

Step 2. Merge sort: Round 4.

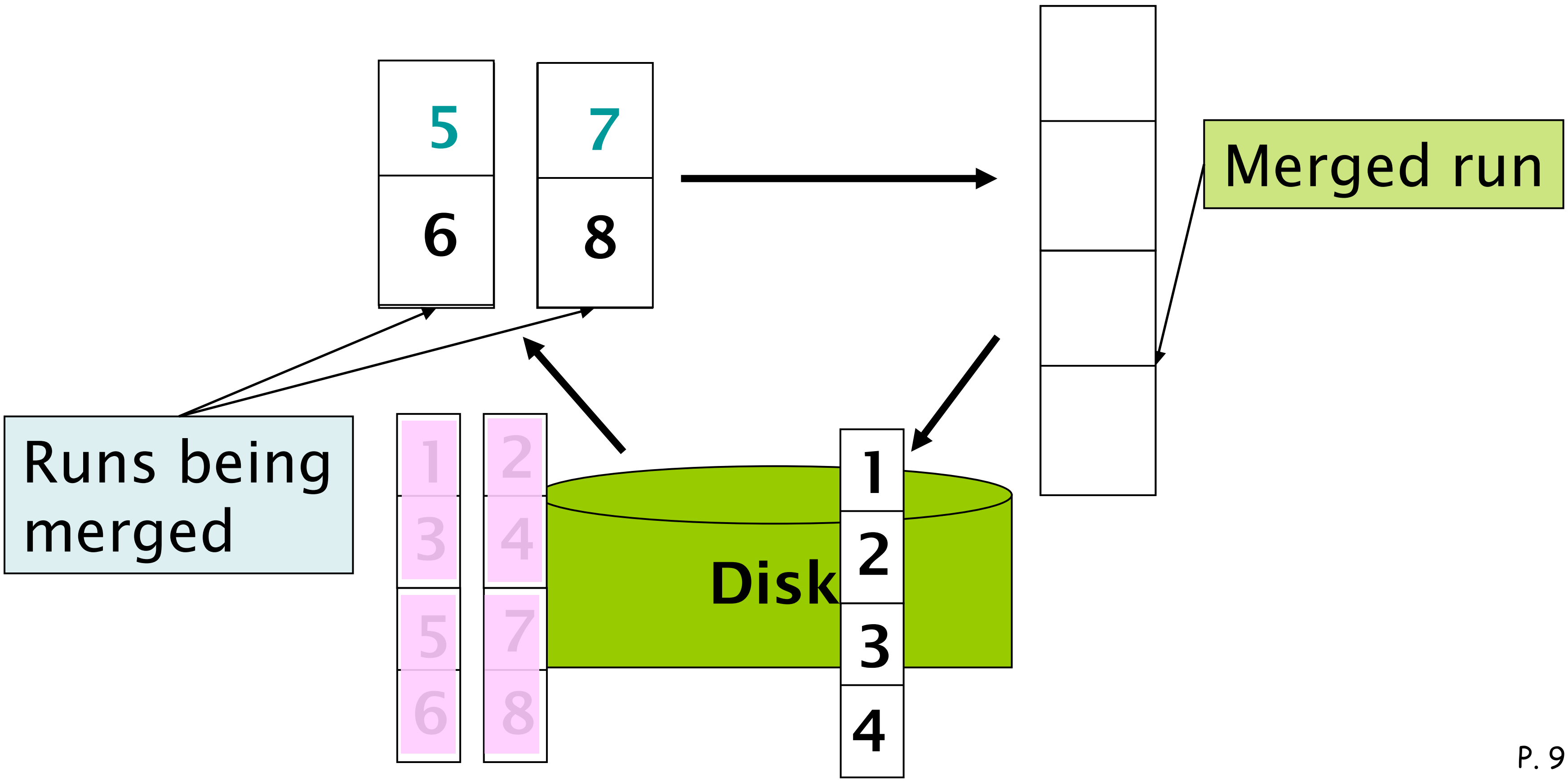


Step 2. Merge sort: Rounds 5 & 6.



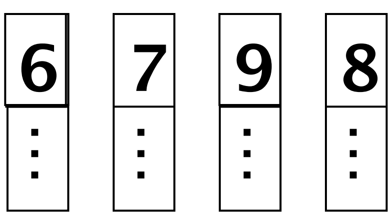
External Sort: *2-way Merge*

- Input buffer
- Output buffer



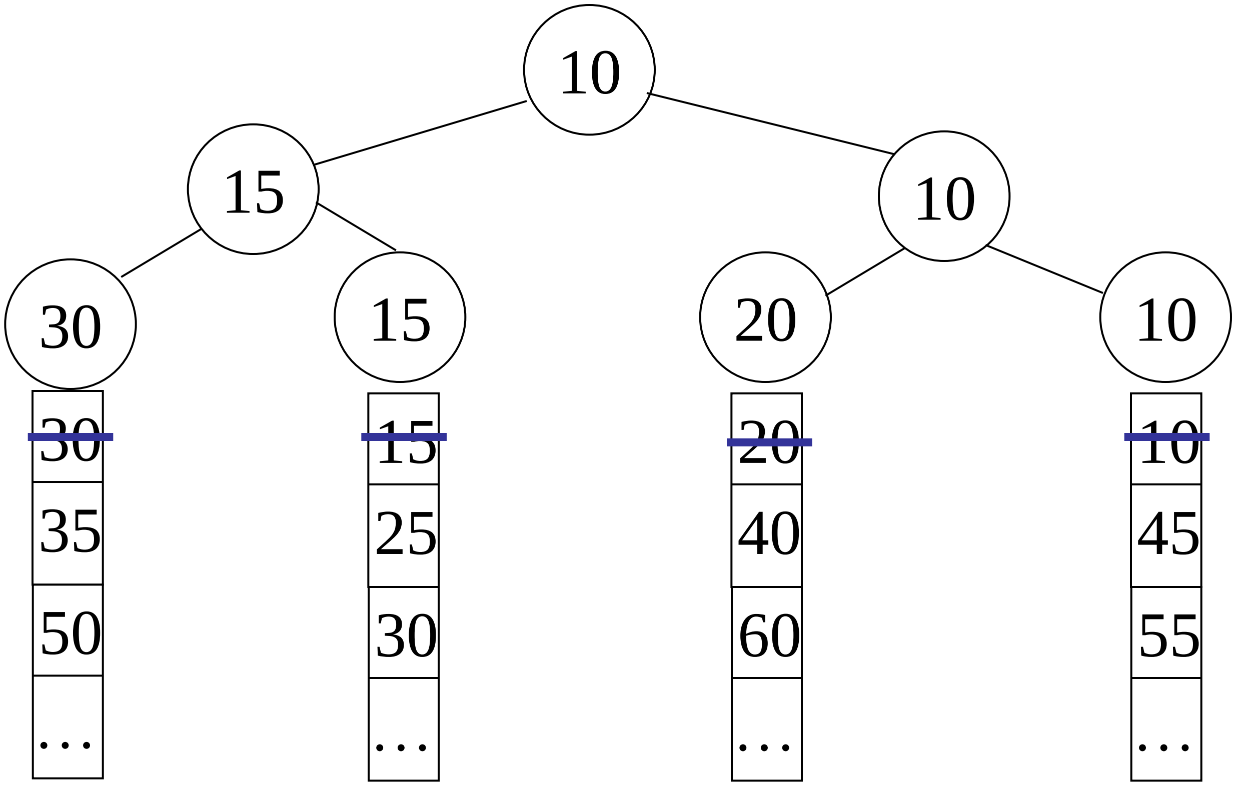
External Sort: *K-way Merge*

- **2-way merge**
  - 64 runs → 32, 16, 8, 4, 2, 1 → $\log_2 64 = 6$ passes
  - 16 runs → 8, 4, 2, 1 → $\log_2 16 = 4$ passes
- **A k-way merge on m runs needs $\log_k m$ passes**
- **Higher-order merge can reduce I/O time**
- **4-way merge**
  - 64 runs → 16, 4, 1 → $\log_4 64 = 3$ passes
  - 16 runs → 4, 1 → $\log_4 16 = 2$ passes



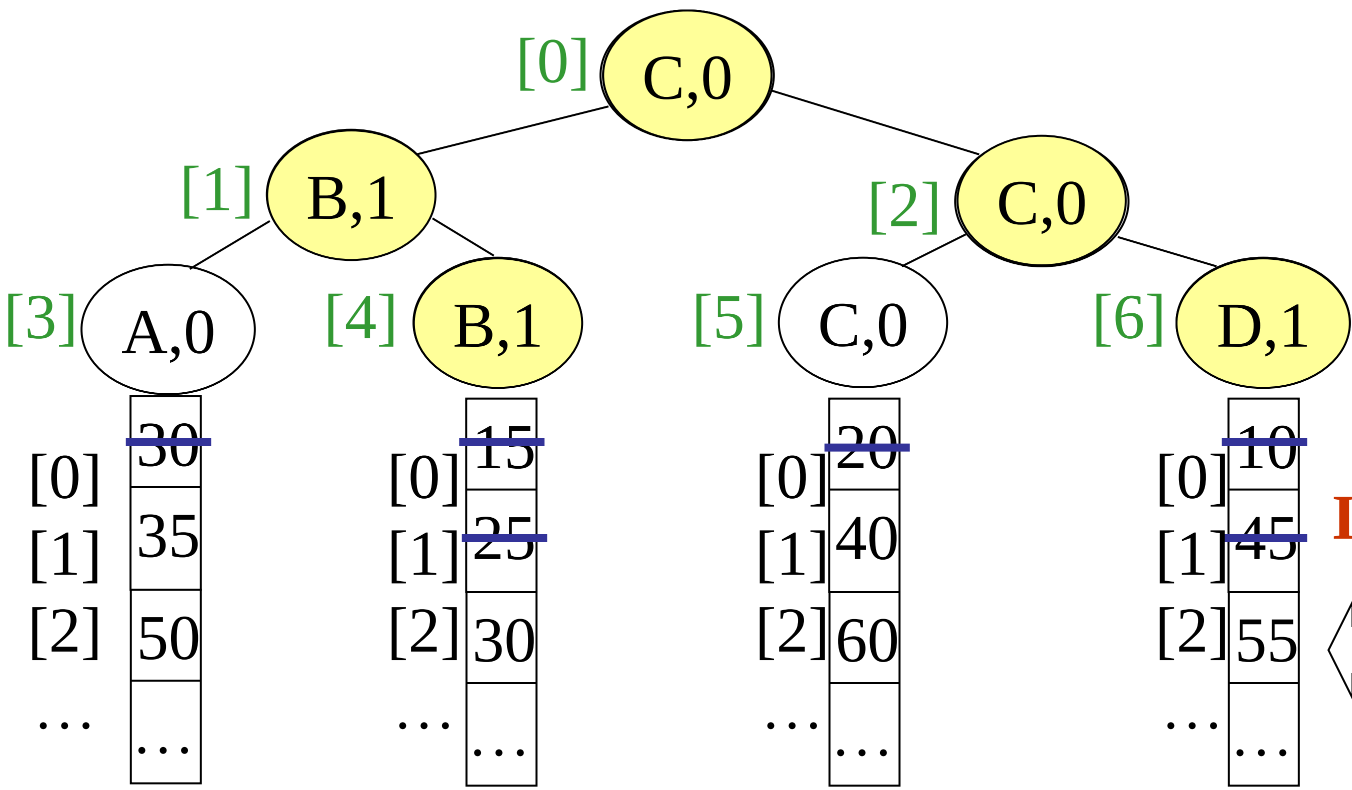
K-way Merge: *Selection Tree*

- Given *k* sorted runs to be merged, a data structure named ***selection tree*** can reduce the number of comparisons for *finding the next smallest element*.



K-way Merge: *Selection Tree*

- **Selection tree on *k* sorted runs**



[0] C,0
[1] B,1
[2] C,0
[3] A,0
[4] B,1
[5] C,0
[6] D,1

Index: locations

Data: values

Secondary Storage: *File Structures*

- **Record**
  - Field
    - **name**
    - **value**

Last Name: ‘Jordan’
First Name: ‘Michael’
Score: 30
Assist: 5
Rebound: 6
Champion: 6
MVP: 5

Last Name: ‘Chamberlain’
First Name: ‘Wilt ’
Score: 30
Assist: 4
Rebound: 22
Champion: 2
MVP: 4

Last Name: ‘Johnson’
First Name: ‘Earvin’
Score: 20
Assist: 11
Rebound: 7
Champion: 5
MVP: 3

Last Name: ‘Abdul-Jabbar’
First Name: ‘Kareem ’
Score: 25
Assist: 4
Rebound: 11
Champion: 6
MVP: 6

File Structures: *Fields*

- **Many ways of adding structure to files to maintain the identity of *fields*:**
  1. Force the field into a predictable length
  2. Begin each field with a length indicator
  3. Separate the fields with delimiters
  4. Use a "*fieldname = value*" expression to identify each field and its content.
     - *Self-describing*

Jordan	← unused space →
Michael	← unused space →
…	

Q&A: Which is the best choice if some fields are missed?

File Structures: *Records*

- **Methods to organize the records of a file:**
  1. Requiring that the records be a predictable number of *bytes* in length.
  2. Requiring that the records be a predictable number of *fields* in length.
  3. Beginning each record with a length indicator consisting of a count of the number of bytes that the record contains.

```
Jordan,Michael,30,5,6,6,5      ← unused space →
Johnson,Earvin,20,11,7,5,3          ← unused space →
…
```

File Structures: *Records*

- **Methods to organize the records of a file:**
  4. Placing a delimiter at the end of each record to separate it from the next record.

     Jordan|Michael|30|5|6|6|5|#Johnson|Earvin|20|11|7|5|3|#…

  5. Using a second file (index) to keep track of *the beginning byte address for each record.*

4 | Jordan|Michael|30|5|6|6|5|Johnson|Earvin|20|11|7|5|3|…

Header

01 27 54 …

Offset: beginning address of a record

Variable-length vs. *Fixed-length*

- **Offset of fixed-length record *RRN = i***

  (Header record length) + (i – 1) * (record length)

  RRN=2: offset = 1+(2-1)*100 = 101 *bytes*

RRN: relative record number

[1] Jordan,Michael,30,5,6,6,5 ← unused space →
[2] Johnson,Earvin,20,11,7,5,3 ← unused space →
… …

4 | Jordan|Michael|30|5|6|6|5|Johnson|Earvin|20|11|7|5|3|…

01 27 54 …

Fixed-length Records: *Deletion*

- **Deletion of record *RRN=2***
  1. move records *3*, *4* to *2*, *3*
  2. move record *4* to *2*
  3. **do not move any record, but *link* all free records as a *free list***

2

[1] Jordan,Michael,30,5,6,6,5 ← unused space → 0

[2] ~~Johnson,Earvin,20,11,7,5,3 ← unused space →~~ 0

[3] Abdul-Jabbar,Kareem,25,4,11,6,6 ← unused space → 0

[4] Chamberlain,Wilt,30,4,22,2,4 ← unused space → 0

Fixed-length Records: *Free List*

- **List head**
  - Stored in the file header
  - Keep the *RRN* of one *deleted record*
- **Use one field of the deleted record to keep the *RRN* of *the next deleted record***
- **Regard these *RRNs* (*offset*) as pointers in the file**

$\boxed{1}$

[1]	~~Jordan,Michael,30,5,6,6,5~~	~~← unused space →~~	2
[2]	~~Johnson,Earvin,20,11,7,5,3~~	~~← unused space →~~	0
[3]	Abdul-Jabbar,Kareem,25,4,11,6,6	← unused space →	0
[4]	Chamberlain,Wilt,30,4,22,2,4	← unused space →	0

Key Sort

- **When sorting a file in memory, the only thing that really needs sorting are the keys of records.**
- **Key sort algorithms work like *internal sort*, but with 2 important differences:**
  - Rather than read an **entire record** into a memory array, we simply read each record into a temporary buffer, *extract the key and then discard the record*.
  - If we want to write the records in sorted order, we have to *read them one more time*.

Key Sort: *Pseudocode*

1. read record from IN_FILE into BUFFER (*in order*)
2. KEY_ARRAY [i].KEY = extract key from BUFFER
3. KEY_ARRAY [i].RRN = i

Op

Cre

Re

REC_COUN ... count stored in *head record*

// read all th ... rds in sequence

for i = 1 to ... EC_COUNT

  do *loop 1*

sort (KEY_A...

// repeatedly ...

for i = 1 to REC_C...

  do *loop 2*

Close IN_FILE and OUT_FILE

1. j = KEY_ARRAY [i].RRN
2. seek in IN_FILE to the record with RRN=j
3. read record from IN_FILE into BUFFER
4. write BUFFER to OUT_FILE (*in order*)

Illustration I: *Key Sort*

(KEY RRN)
(**99**, 1)
(**95**, 2)
(**88**, 3)
(**99**, 4)
(**95**, 5)
…

(KEY RRN)
(99, **1**)
(99, **4**)
…
(95, **2**)
(95, **5**)
…
(88, **3**)
…

REC_COUNT (number of records)
record 1: BS001, **99**, Lee, …
record 2: BS003, **95**, Lin, …
record 3: BS004, **88**, Wang, …
record 4: BS005, **99**, Liu, …
record 5: BS008, **95**, Chen, …

record 1: BS001, 99, Lee, …
record 4: BS005, 99, Liu, …
…
record 2: BS003, 95, Lin, …

IN_FILE

OUT_FILE

Key Sort: *Limitation*

- **Writing the records in sorted order requires as many** random seeks **as there are records.**
- **Since writing is interspersed with reading, writing also requires as many seeks as there are records.**
- **Why bother to write the file of records in the order of their keys?**
  - *Simply write back* ***the sorted index!***

Illustration II: *Secondary Index*

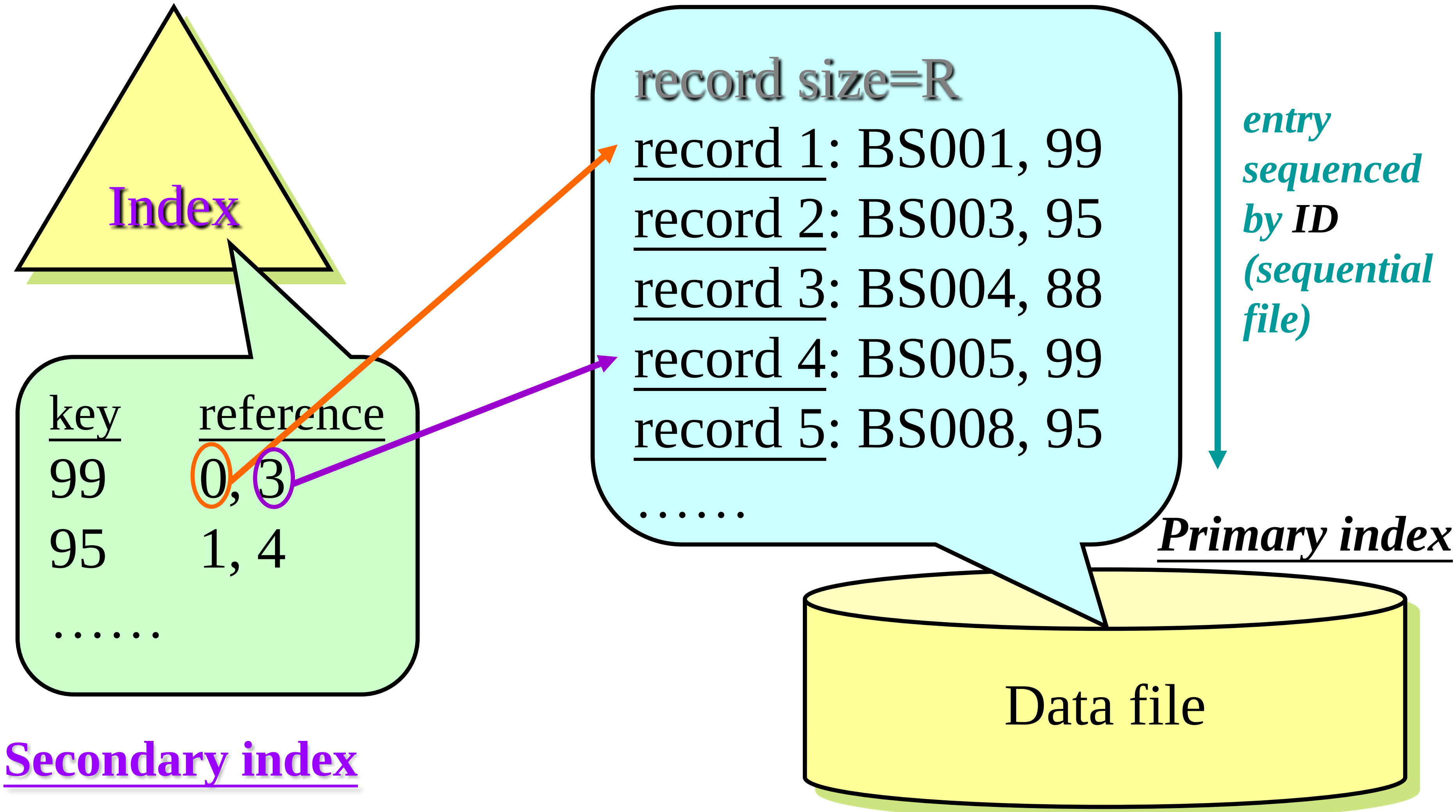


Illustration IV: *Bucket Chaining*

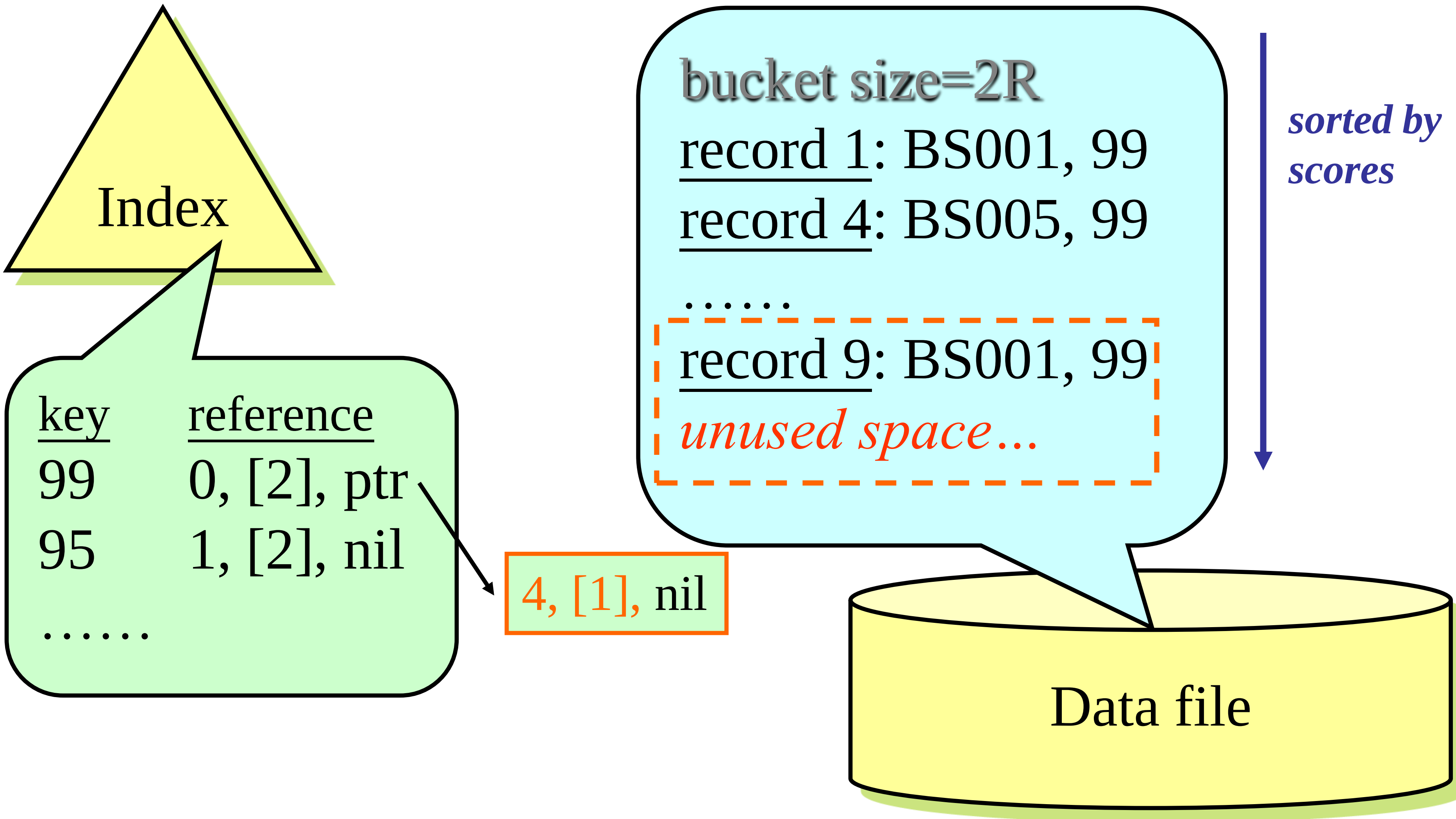


Illustration V: *Backup & Reload*