

# Storing Data: Disks and Files

# Memory Hierarchy

---

- *Primary Storage*: main memory.

fast access, expensive.

- *Secondary storage*: hard disk.

slower access, less expensive.

- *Tertiary storage*: tapes, cd, etc.

slowest access, cheapest.

# Disks

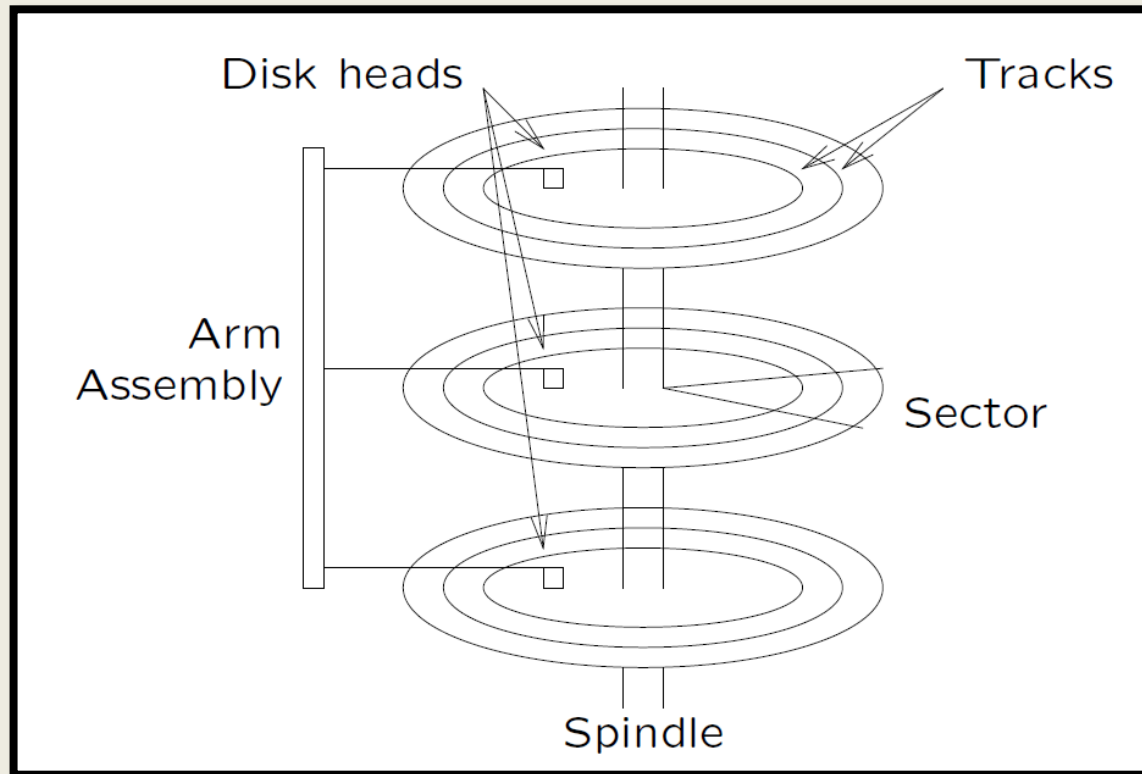
---

## Characteristics of disks:

- collection of platters
- each platter = set of tracks
- each track = sequence of sectors (blocks)
- transfer unit: 1 block (e.g. 512B, 1KB)
- access time depends on proximity of heads to required block access
- access via block address (p, t, s)

# Disks

---



- Data must be in memory for the DBMS to operate on it.
- Smallest process unit is Block (4096KB): If a single record in a block is needed, the entire block is transferred.

# Disks

---

Access time includes:

- seek time (find the right track, e.g.  $10\text{msec}$ )
- rotational delay (find the right sector, e.g.  $5\text{msec}$ )
- transfer time (read/write block, e.g.  $10\mu\text{sec}$ )

Random access is dominated by **seek time** and **rotational delay**

# Disk Space Management

---

- *Improving Disk Access:*

- Use knowledge of data access patterns.

- E.g. two records often accessed together: put them in the same block (clustering)

- E.g. records scanned sequentially: place them in consecutive sectors on same track

- Keep Track of Free Blocks

- Maintain a list of free blocks

- Use bitmap

- Using OS File System to Manage Disk Space

- extend OS facilities, but not rely on the OS file system.

- (portability and scalability)

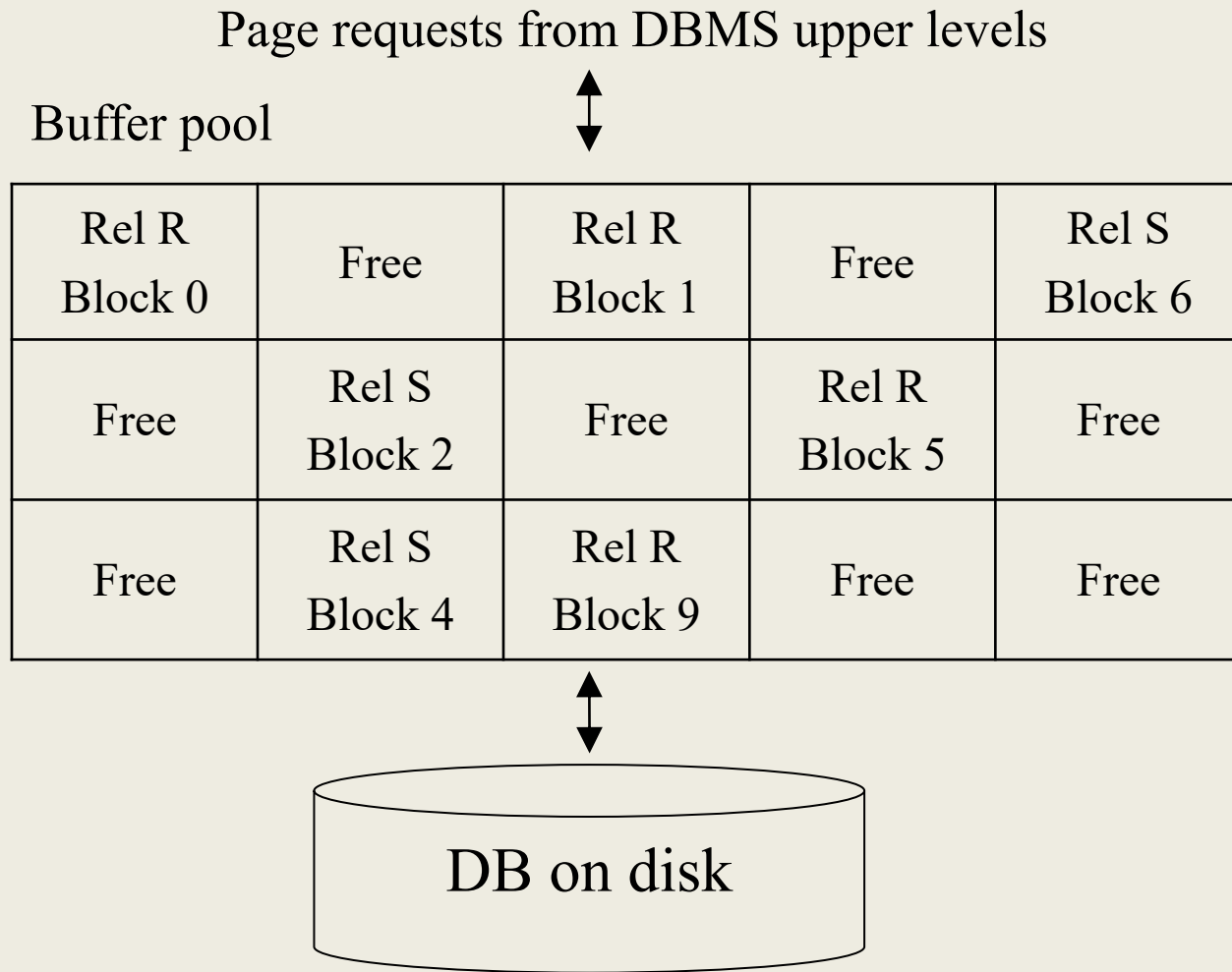
# Buffer Management

---

- Manages traffic between disk and memory by maintaining a *buffer pool* in main memory.
- Buffer pool
  - collection of *page slots* (frames) which can be filled with copies of disk block data.
  - One page = 4096 Bytes = One block

# Buffer Pool

---





# Buffer Pool

---

- The *request\_block* operation
  - If block *is* already in buffer pool:
    - no need to read it again
    - use the copy there (unless write-locked)
  - If block *is not* in buffer pool yet:
    - need to read from hard disk into a free frame
    - if no free frames, need to remove block using *a buffer replacement policy*.
- The *release\_block* function indicates that block is no longer in use
  - good candidate for removal / replacing

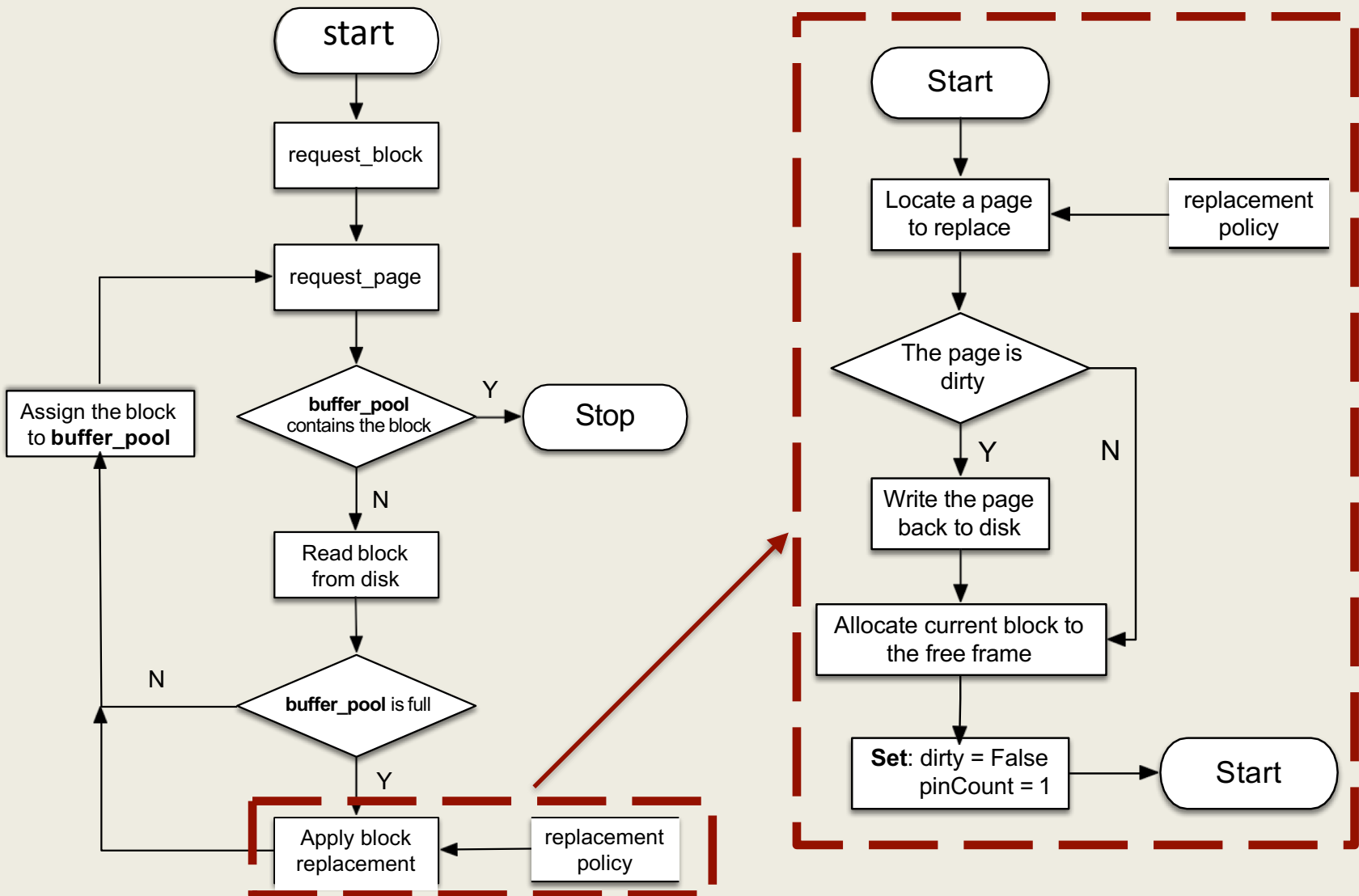
# Buffer Pool

---

For each frame, we need to know:

- whether it is currently in use
- whether it has been modified since loading (*dirty bit*)
- how many transactions are currently using it (*pin count*)
- (maybe) time-stamp for most recent access

# The *request\_block* Operation



# Buffer Pool

---

## **The *release\_block* Operation**

1. Decrement pin count for specified page.

No real effect until replacement required.

## **The *write\_block* Operation**

1. Updates contents of page in pool
2. Set dirty bit on

Note: Doesn't actually write to disk, until been replaced, or forced to commit

## **The *force\_block* operation**

1. "commits" by writing to disk.

# Buffer Replacement Policies

---

- Least Recently Used (LRU)
  - release the frame that has not been used for the longest period.
  - intuitively appealing idea but can perform badly
- First in First Out (FIFO)
  - need to maintain a queue of frames
  - enter tail of queue when read in
- Most Recently Used (MRU):
  - release the frame used most recently
- Random

No one is guaranteed to be better than the others. Quite dependent on applications.

## Example1:

**Data pages:** P1, P2, P3, P4

**Queries:**

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

**Buffer:**

<b>P1</b> Q4	<b>P2</b> Q5	<b>P3</b> Q3
--------------	--------------	--------------

Q6: read P4:

- **LRU:** Replace P3
- **MRU:** Replace P2
- **FIFO:** Replace P1
- **Random:** randomly choose one buffer to replace

## Example 2:

**Data pages:** P1, P2, ..., P11

**Queries:**

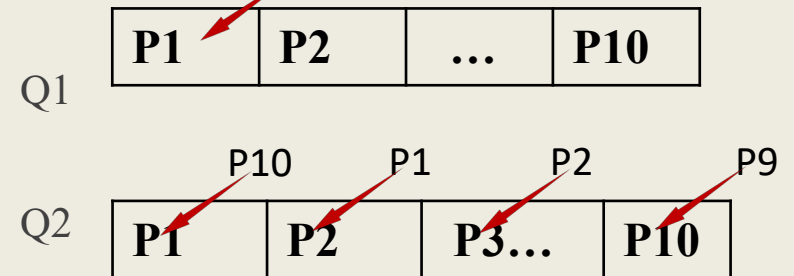
Q1: read P1, P2,..., P11;

Q2, read P1, P2,..., P11;

Q3: Read P1, P2,...,P11

**Buffer:** 10 pages like Example 1

LRU/FIFO: p11



**Boom: We need to get in/out every page**

MRU: performs the best in this case.

**Practice yourself!!**

# Record Formats

---

Records are stored within fixed-length blocks.

- ***Fixed-length***: each field has a fixed length as well as the number of fields.

33357462	Neil Young	Musician	0277
4 bytes	40 bytes	20 bytes	4 bytes

- Easy for intra-block space management.
- Possible waste of space.

- ***Variable-length***: some field is of variable length.

33357462	Neil Young	Musician	0277
4 bytes	10 bytes	8 bytes	4 bytes

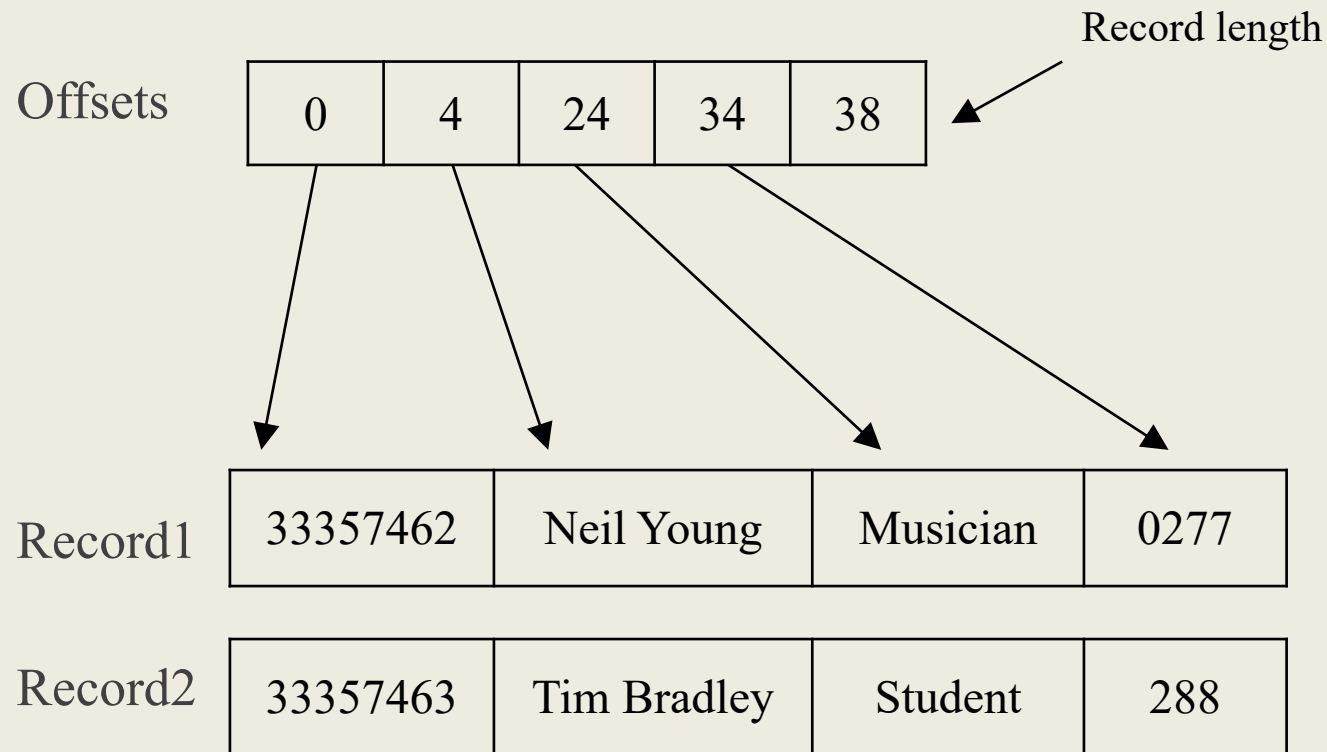
- complicates intra-block space management
- does not waste (as much) space.

# Fixed-Length

---

Encoding scheme for fixed-length records:

- length + offsets stored in header





# Variable-Length

---

Encoding schemes for variable-length records:

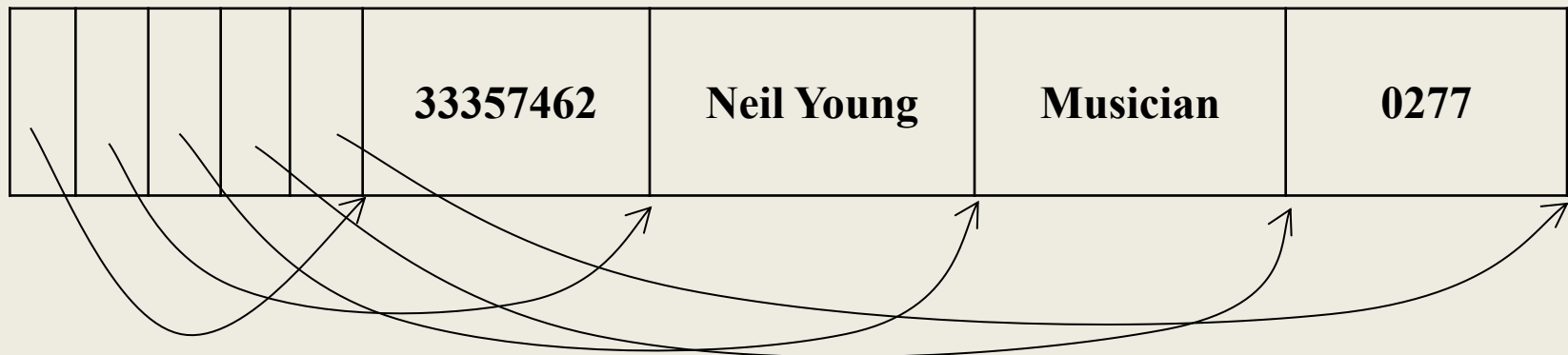
- Prefix each field by length

**4** xxxx **10** Neil Young **8** Musician **4** xxxx

- Terminate fields by delimiter

33357462/Neil Young/Musician/0277/

- Array of offsets



# Block (Page) Formats

---

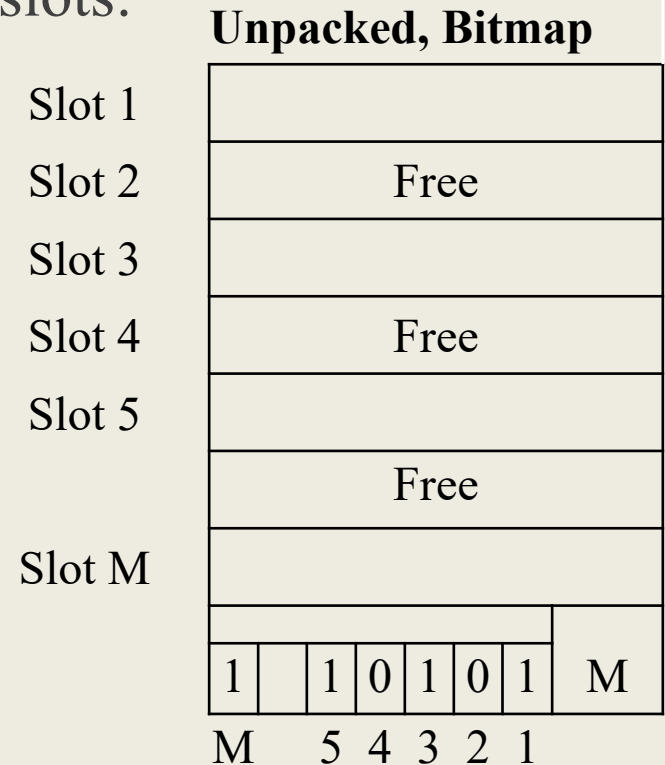
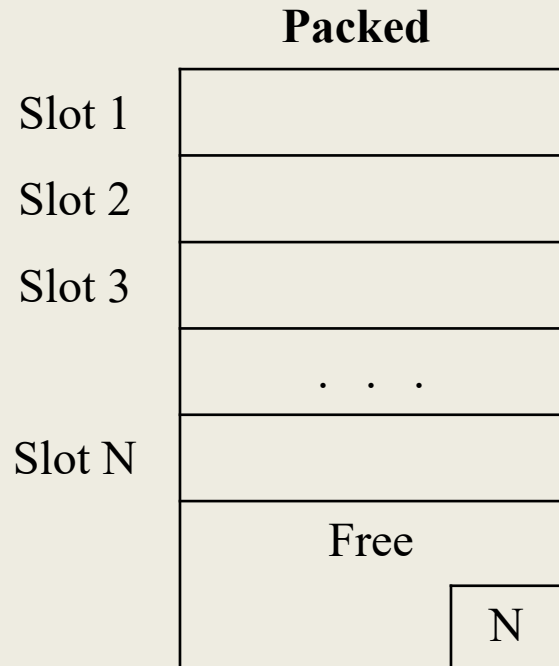
A block is a collection of *slots*.

Each slot contains a record.

A record is identified by  $rid = \langle \text{page id, slot number} \rangle$ .

# Fixed Length Records

For fixed-length records, use record slots:



Insertion: occupy first free slot; packed more efficient.

Deletion: (a) need to compact, (b) mark with 0; unpacked more efficient.

# Variable-Length Records

---

For variable-length records, use slot *directory*.

Possibilities for handling free-space within block:

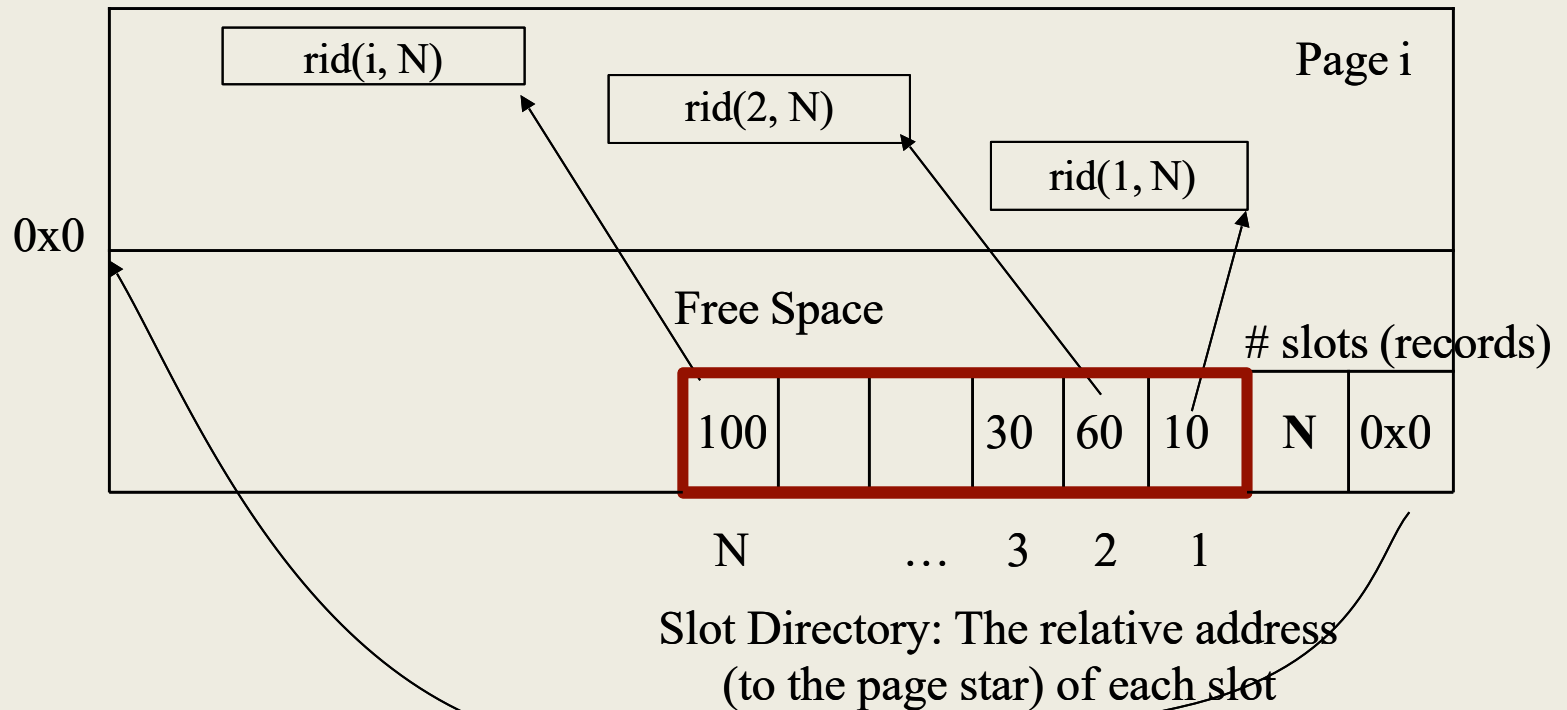
- compacted (one region of free space)
- fragmented (distributed free space)

In practice, probably use a combination:

- normally fragmented (cheap to maintain)
- compact when needed (e.g. record won't fit)

# Variable-Length Records

- Compacted free space:



# Variable-Length Records

---

## Overflows

Some file structures (e.g. hashing) allocate records to specific blocks.

- What happens if specified block is already full?
- Need a place to store “excess” records.

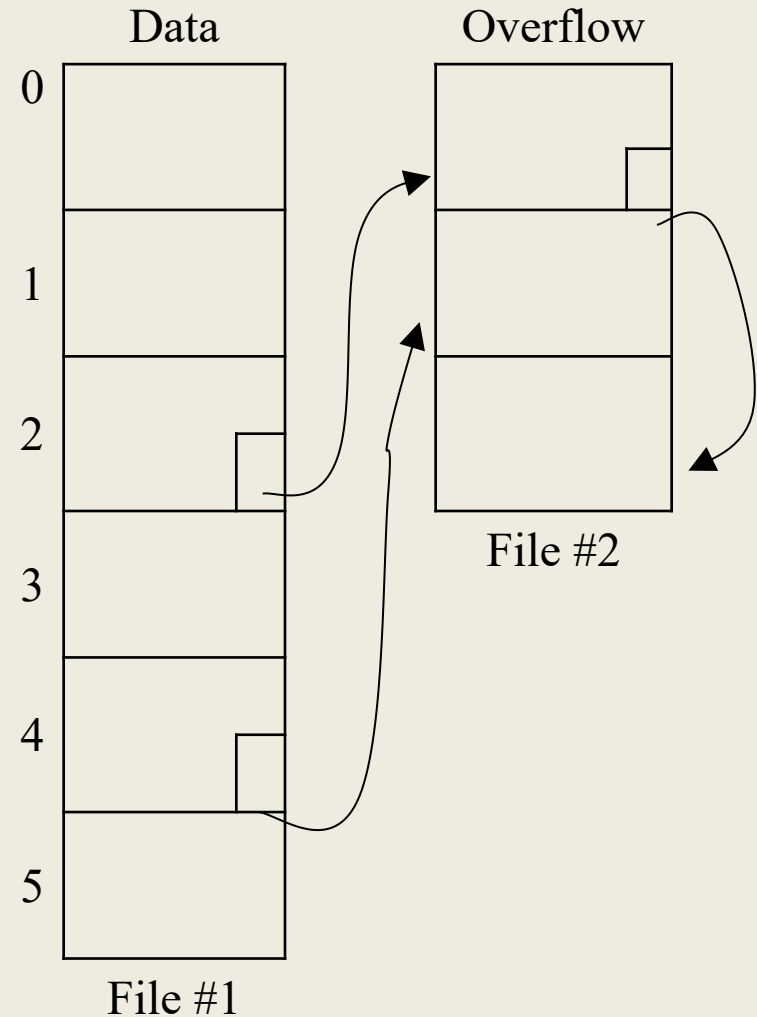
Introduce notion of *overflow blocks*:

- located outside main file (don’t destroy block sequence of main file)
- connected to original block
- may have “chain” of overflow blocks

New blocks are always appended to file.

# Variable-Length Records

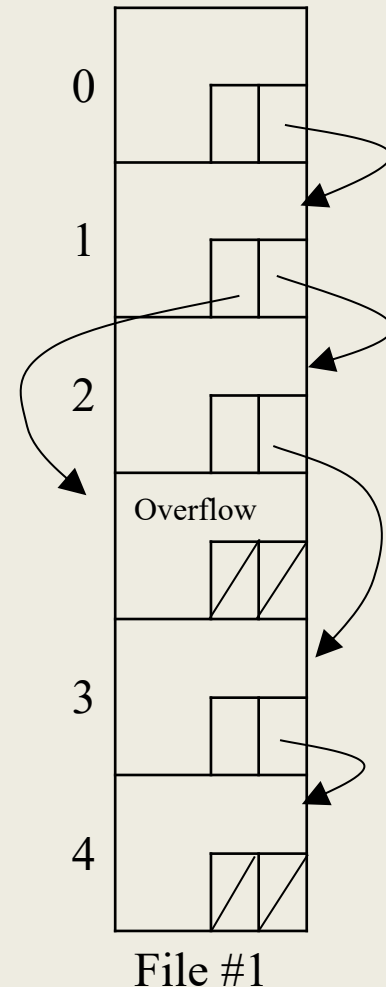
- Overflow blocks in a separate file:
- Note: “pointers” are implemented as file offsets.



# Variable-Length Records

- Overflow blocks in a single file:
- Not suitable if accessing blocks via offset (e.g. hashing).

**Data + overflows**





# Files

---

A *file* consists of several data blocks.

*Heap Files*: unordered pages (blocks).

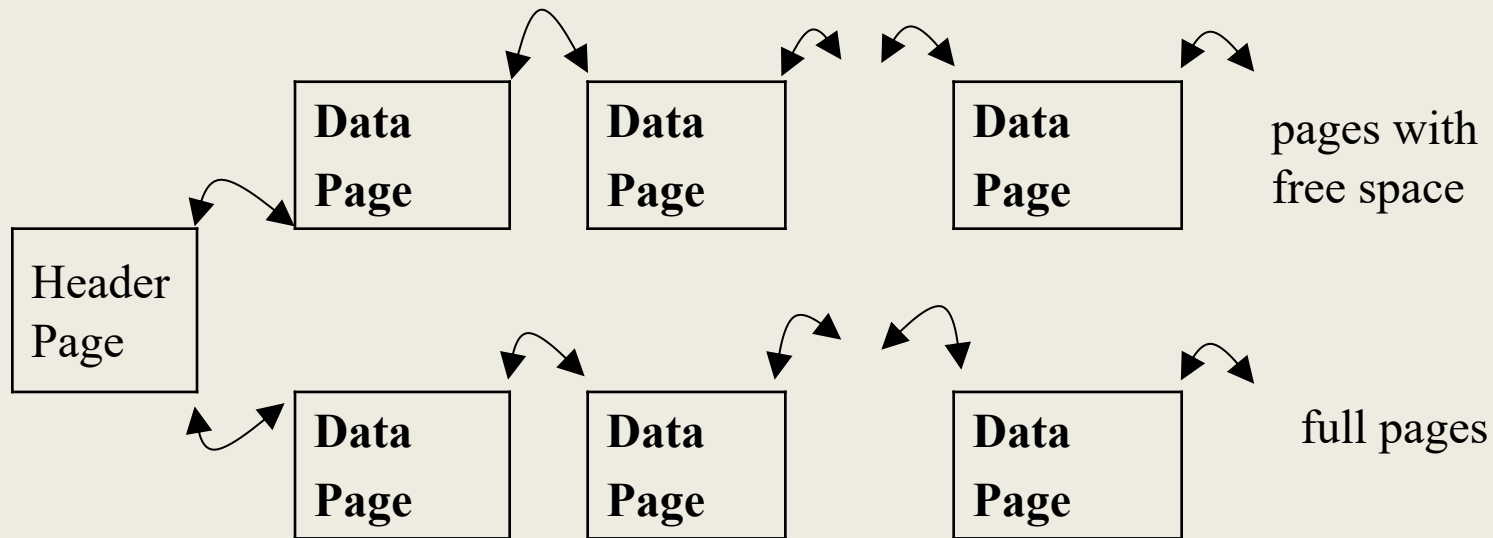
Two alternatives to maintain the block information:

- Linked list of pages.
- Directory of pages.

# Linked List of Pages

---

Maintain a heap file as a doubly linked list of pages.



Organized by a Linked List

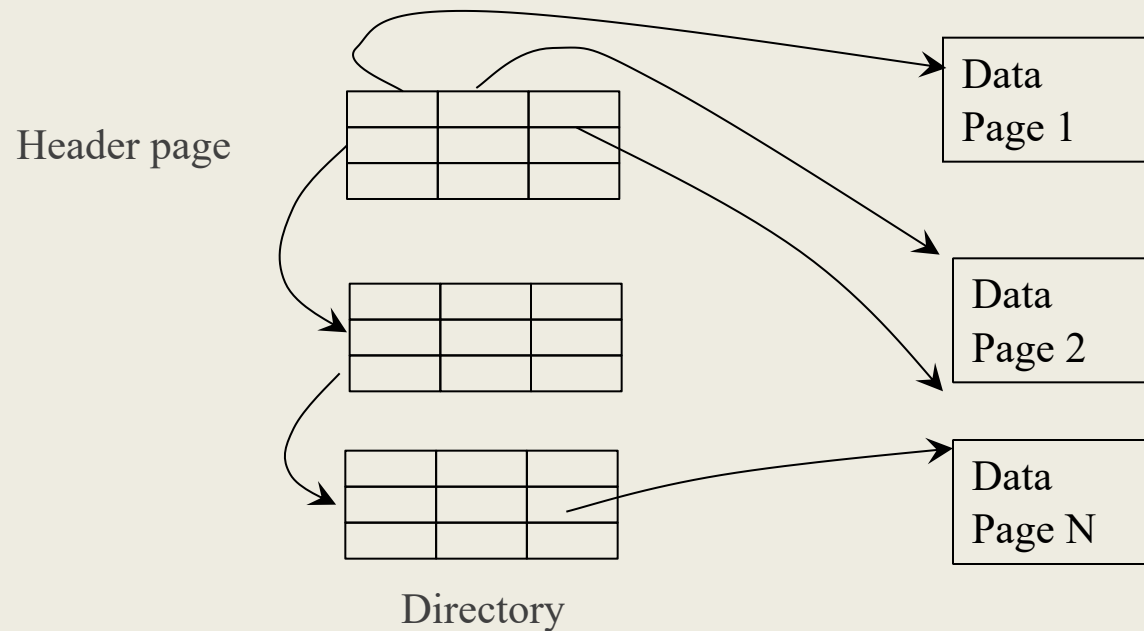
**Disadvantage:** all pages will virtually be on the free list of records if records are of variable length. To insert a record, several pages may be retrieved and examined.

# Directory of Pages

---

Maintain a directory of pages.

- Each directory entry identifies a page (or a sequence of pages) in the heap file.
- Each entry also maintains a bit to indicate if the corresponding page has any free space.



# File Organizations

---

- Three types of file organisations
  - Heap, Sorted and Hashed
- The COST of processing DB queries
  - SCAN, Search (Single, Range), Insert, Delete

# File Organizations

---

- Recall
  - The basic store unit on disk (in memory) is block (page)
  - We will use page/block interchangeably.
  - One page consists of multiple data records.
- Three types
  - Heap Files (The simplest)
    - ❑ Page after page, always insert at the end
  - Sorted Files
    - ❑ Records are sorted (within and among pages) w.r.t. the search key
  - Hashed Files.
    - ❑ The pages are assigned to multiple buckets, each containing several pages
    - ❑ Each page has been assigned with an ID (bucket ID) to tell where to find the page

# Cost Model

---

- Notations
  - **B**: the number of pages (blocks)
  - **R**: the average number of records in a page.
  - **D**: average time to read or write a disk page.
  - **C**: average time to process a record.
  - **H**: the time required to apply a hash function to a record.

# Operations to be investigated

---

- *Scan*: fetch all records in a table.
- Search with equality selection. (*SWES*)
  - “Find the students record with sid = z1234567”
- Search with Range Selection. (*SWRS*)
  - “Find all students with name alphabetically after ‘Smith’”
- *Insert*: Insert a given record into the table.
- *Delete*: Delete a record with given rid.

Below, we examine the costs of these operations with respect to the 3 different file organizations.

# Heap Files

---

- **Scan:**
    - $B(D + RC)$
  - **SWES:**
    - $0.5B(D + RC)$  on average if the selection is specified on a key.
    - Otherwise  $B(D + RC)$ .
- 
- **B**: the number of pages
  - **R**: the average number of records in a page (block).
  - **D**: average time to read or write a disk page.
  - **C**: average time to process a record.



# Heap Files

---

- **SWRS:**
  - $B(D + RC)$
- **Insert:**
  - $(D + C) + D = 2D + C$  (Always insert to the end of the table)
- **Delete:**
  - Only one record is involved.
    - ❑ The average cost is  $0.5B(D + RC) + D$  if rid is not given;
    - ❑ otherwise  $(D + C) + D$ .
  - Several records are involved. Expensive.
    - **B**: the number of pages
    - **R**: the average number of records in a page (block).
    - **D**: average time to read or write a disk page.
    - **C**: average time to process a record.

# Sorted Files

---

Sorted on a search key

a combination of one or more fields.

Query against the search key:

- Scan:  $B(D + RC)$ .
- SWES:
  - $O(D \log_2 B + C \log_2 R)$  if single record. (log comes from the binary search)
  - $O(D \log_2 B + C \log_2 R + \#matches)$ .
- SWRS:  $O(D \log_2 B + C \log_2 R + \#matches)$ .
- Insert: expensive.
  - Search cost plus  $2 * (0.5B(D + RC))$ .
- Delete: expensive.
  - Search cost plus  $2 * (0.5B(D + RC))$ .

# Hashed Files

---

- The pages in a file are grouped into buckets.
- The buckets are defined by a hash function.
- Pages are kept at about 80% occupancy.

Assume the data manipulation is based on the hash key.

- Scan:  $1.25B(D + RC)$ .
- SWES:  $H + D + 0.5RC$  if each hash bucket contains only one page.
- SWRS:  $1.25B(D + RC)$ . (No help from the hash structure)
- Insert: Search cost plus  $C + D$  if one block involved.
- Delete: Search cost plus  $C + D$  if one block involved.

# Summary

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5 BD	BD	Search + D	Search + D
Sorted	BD	D logB	$D \log B + \# \text{ matches}$	Search + BD	Search + BD
Hashed	1.25 BD	D	1.25 BD	2 D	Search + BD

A Comparison of I/O Costs

**(I/O Costs: The cost of reading/writing from/to disk )**