Storing Data: Disks and Files

11.1 Memory Hierarchy

- *Primary Storage*: main memory.

  fast access, expensive.

- *Secondary storage:* hard disk.

  slower access, less expensive.

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

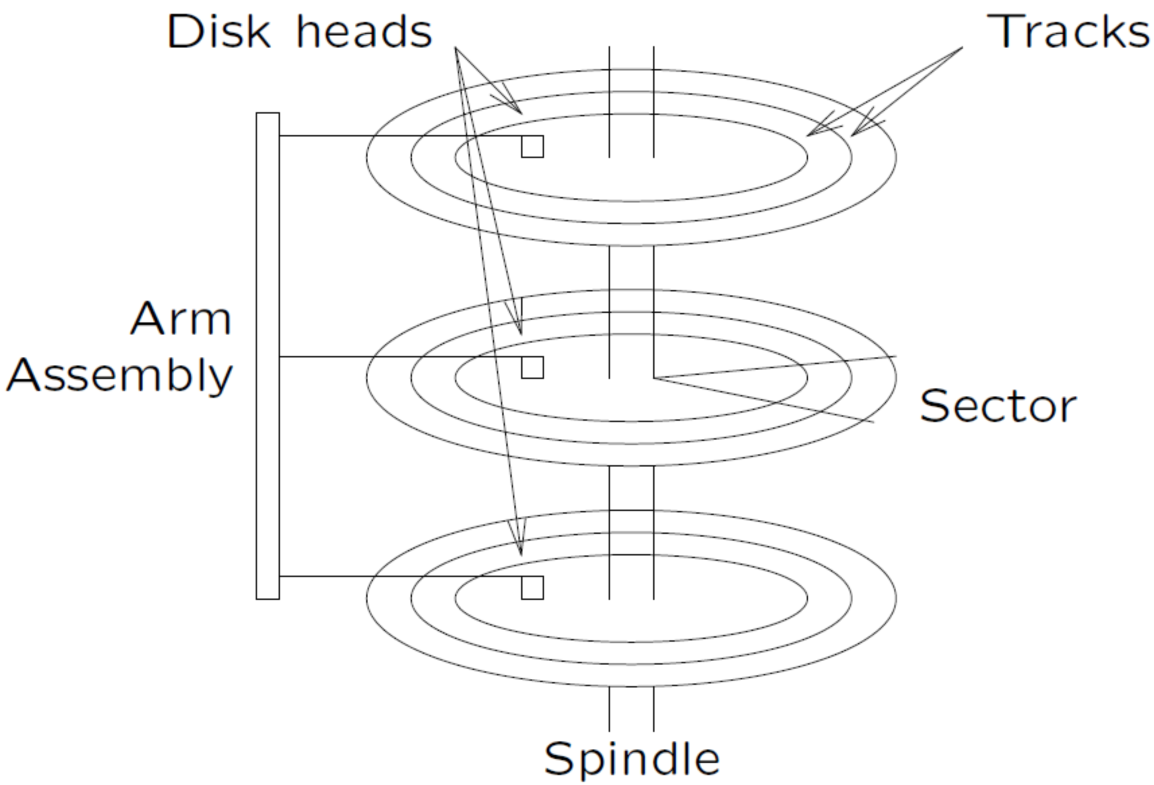
  slowest access, cheapest.

11.2 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)

11.2 Disks



- Data must be in memory for the DBMS to operate on it.
- If a single record in a block is needed, the entire block is transferred.

11.2 Disks

Access time includes:

- seek time (find the right track, e.g. 10*msec*)
- rotational delay (find the right sector, e.g. 5*msec*)
- transfer time (read/write block, e.g. 10μ*sec*)

➔Random access is dominated by seek time and rotational delay

11.3 Disk Space Management

Disk space is managed by the disk space manager.

1. *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

11.3 Disk Space Management

2. *Keeping Track of Free Blocks*

   – Maintain a list of free blocks.

   – Use bitmap.

3. *Using OS File System to Manage Disk Space*

   – extend OS facilities, but

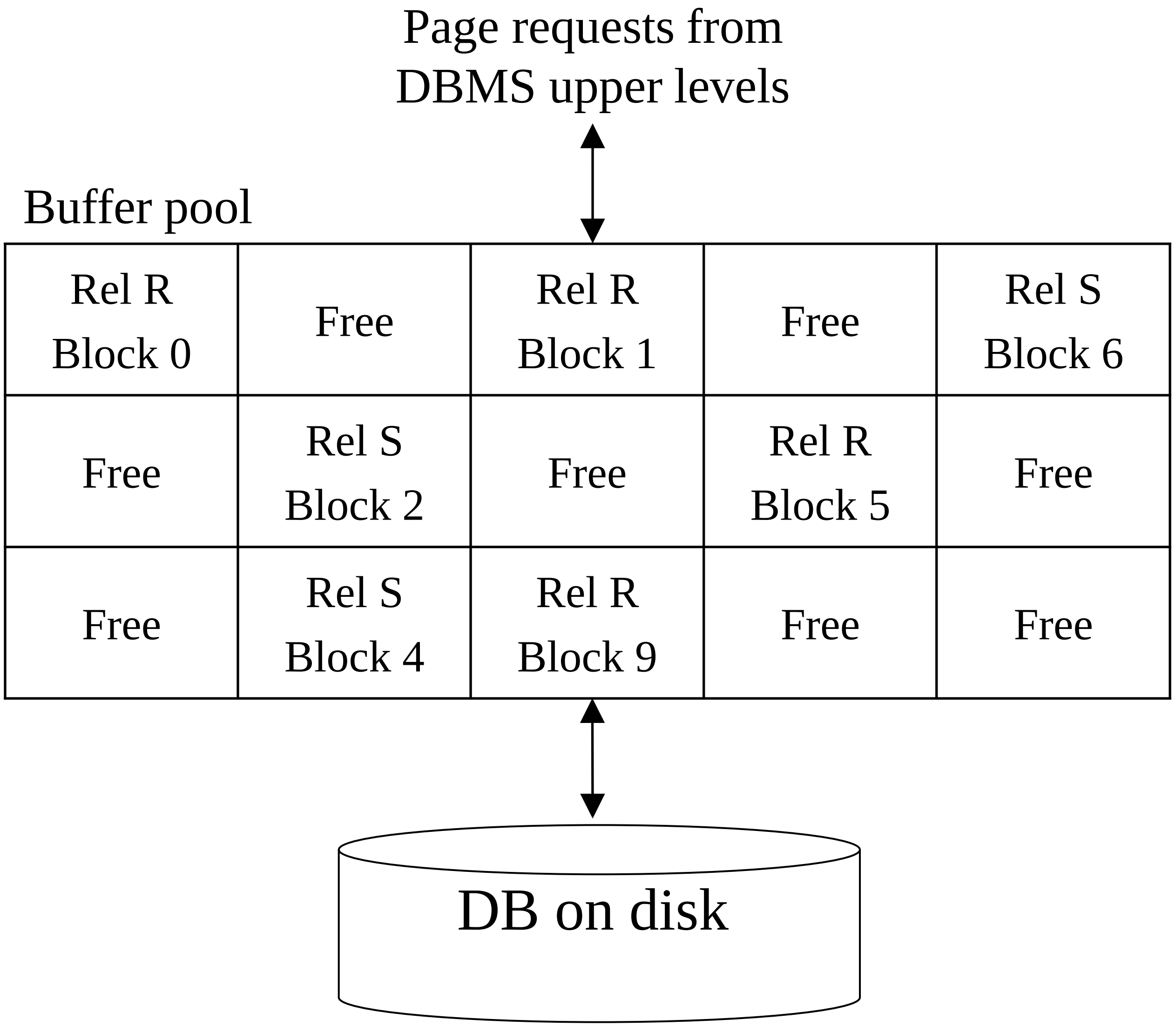
   – not rely on the OS file system.

   (portability and scalability)

11.4 Buffer Management

- *Buffer Manager*
- 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.

11.4.1 Buffer Pool



11.4.1 Buffer Pool

- The *request_block* operation replaces *read block* in all file access algorithms.
- If block *is* already in buffer pool:
  - no need to read it again
  - use the copy there (unless write-locked)
- If block is *not* already in buffer pool:
  - 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.

11.4.1 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

11.4.1 Buffer Pool

The *request_block* Operation

Method:

1. Check buffer pool to see if it already contains requested block.

   If not, the block is brought in as follows:

   (a) Choose a frame for replacement, using *replacement policy*

   (b) If frame chosen is dirty, write block to disk

   (c) Read requested page into now-vacant buffer frame (and set *dirty* = *False* and *pinCount* = 0)

2. *Pin* the frame containing requested block.

   (This simply means updating the pin count.)

3. Return address of frame containing requested block.

11.4.1 Buffer Pool

The *release_block* Operation

Method:

1. Decrement pin count for specified page.

No real effect until replacement required.

The *write_block* Operation

Method:

1. Updates contents of page in pool

2. Set dirty bit on

Note: Doesn’t actually write to disk.

The *force_block* operation “commits” by writing to disk.

11.4.2 Buffer Replacement Policies

Several schemes are commonly in use:

• 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 is guaranteed better than the other.

For DBMS, we may predict accesses better.

Example1:

Data pages: P1, P2, P3, P4

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2; Q6: read P4;

Buffer:

P1 Q4	P2 Q5	P3

Regarding Q6,

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

Example 2:

Data pages: P1, P2, …, P11

10 buffer pages as in Example 1

Q1: read P1, P2,…, P11;

Q2, read P1, P2,…, P11;

Q3: Read P1, P2,…,P11

LRU/FIFO: I/O P1, P2, …, P11 for each query.

MRU performs the best.

11.5 Record Formats

Records are stored within fixed-length blocks.

- *Fixed-length*: each field has a fixed length as well as the number of fields.
  - Easy for intra-block space management.
  - Possible waste of space.
- *Variable-length*: some field is of variable length.
  - complicates intra-block space management
  - does not waste (as much) space.

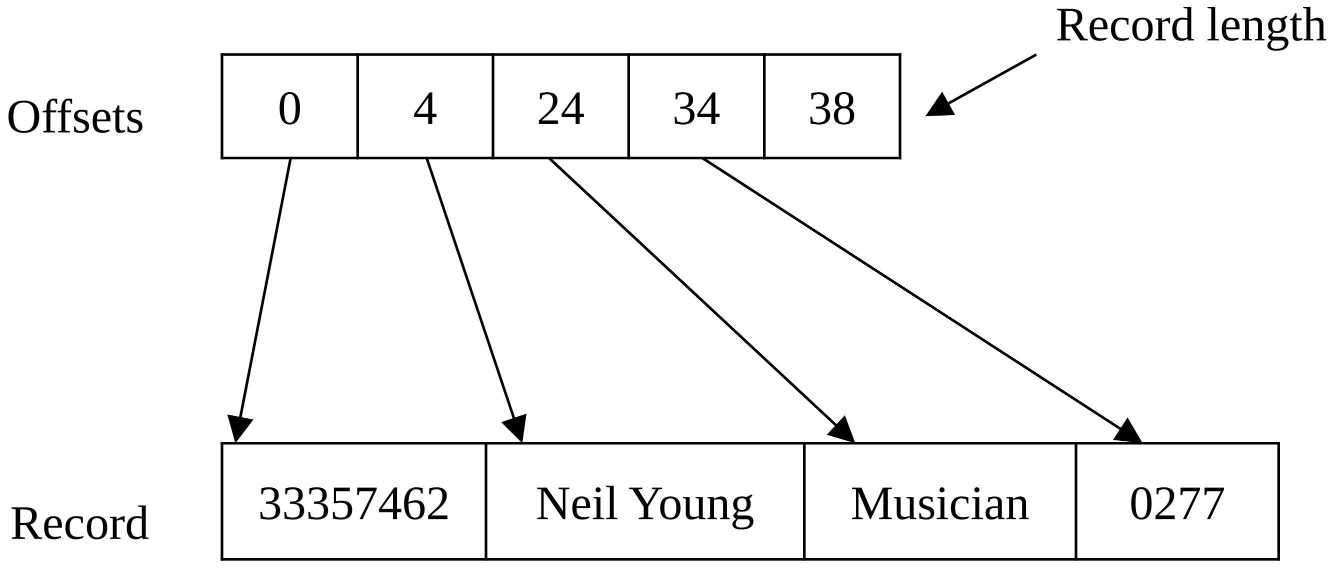
Record format info:

- best stored in data dictionary
- with dictionary memory-resident

11.5.1 Fixed-Length

Encoding scheme for fixed-length records:

- length + offsets stored in header



11.5.2 Variable-Length

Encoding schemes for variable-length records:

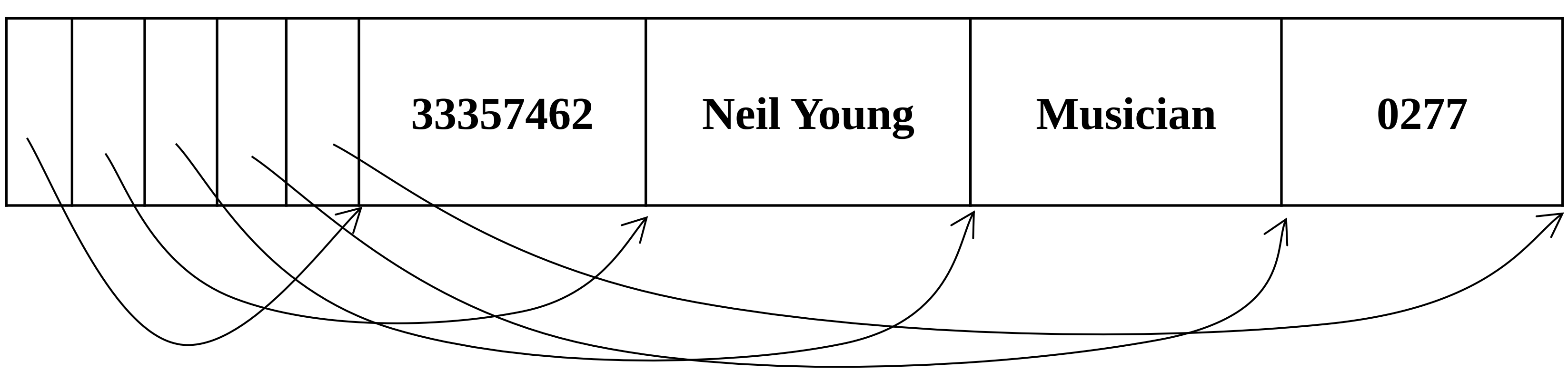
- Prefix each field by length

$\boxed{\mathbf{4}}$ xxxx $\boxed{\mathbf{10}}$ Neil Young $\boxed{\mathbf{8}}$ Musician $\boxed{\mathbf{4}}$ xxxx

- Terminate fields by delimiter

33357462/Neil Young/Musician/0277/

- Array of offsets



11.6 Block (Page) Formats

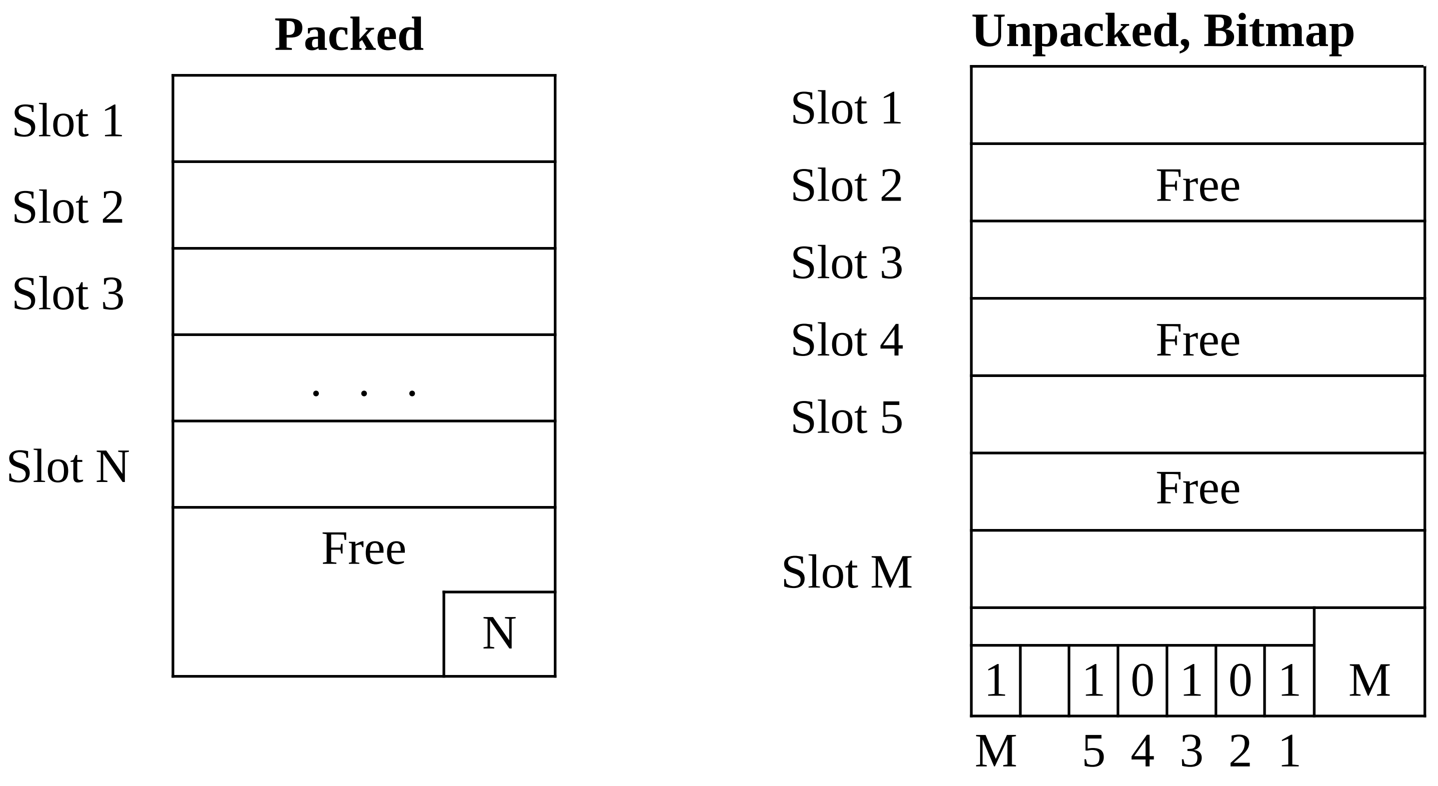
A block is a collection of *slots*.

Each slot contains a record.

A record is identified by rid =< page id, slot number >.

11.6.1 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.

11.6.2 Variable-Length Records

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

Possibilities for handling free-space within block:

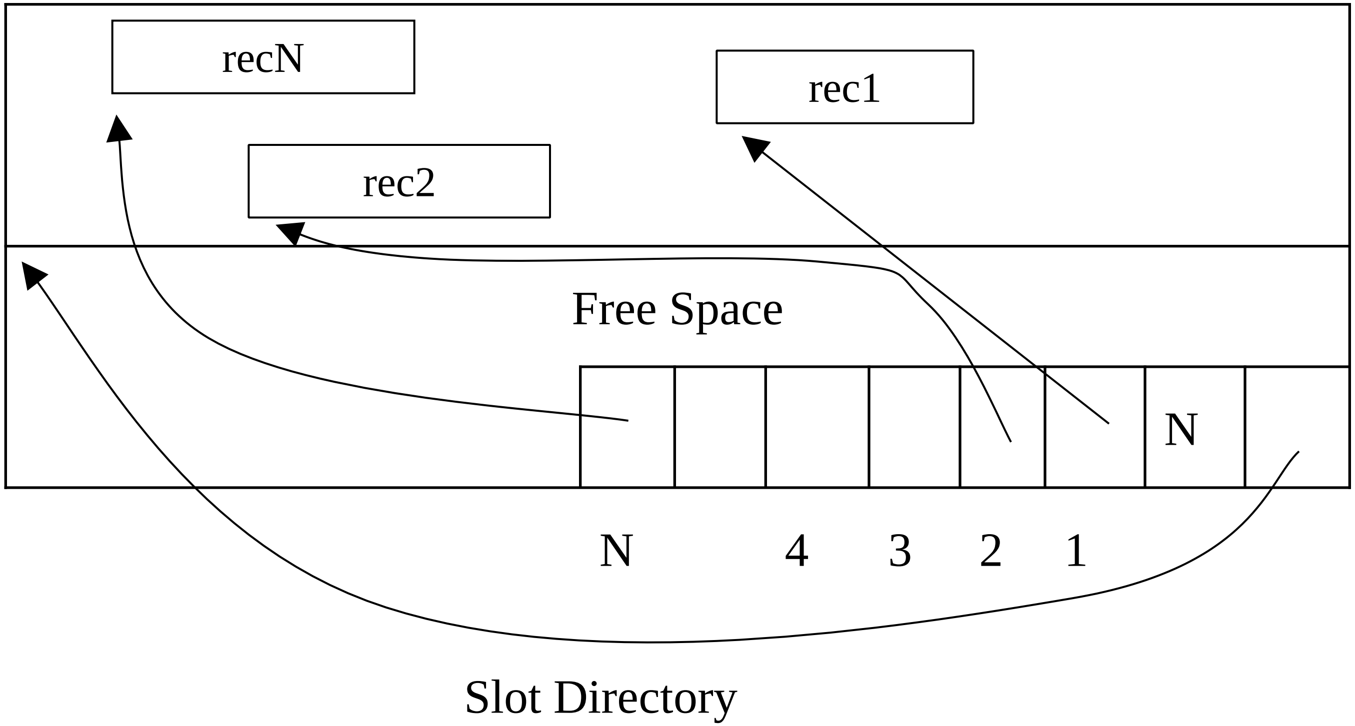
- compacted (one region of free space)
- fragmented (distributed free space)

In practice, probably use a combination:

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

11.6.2 Variable-Length Records

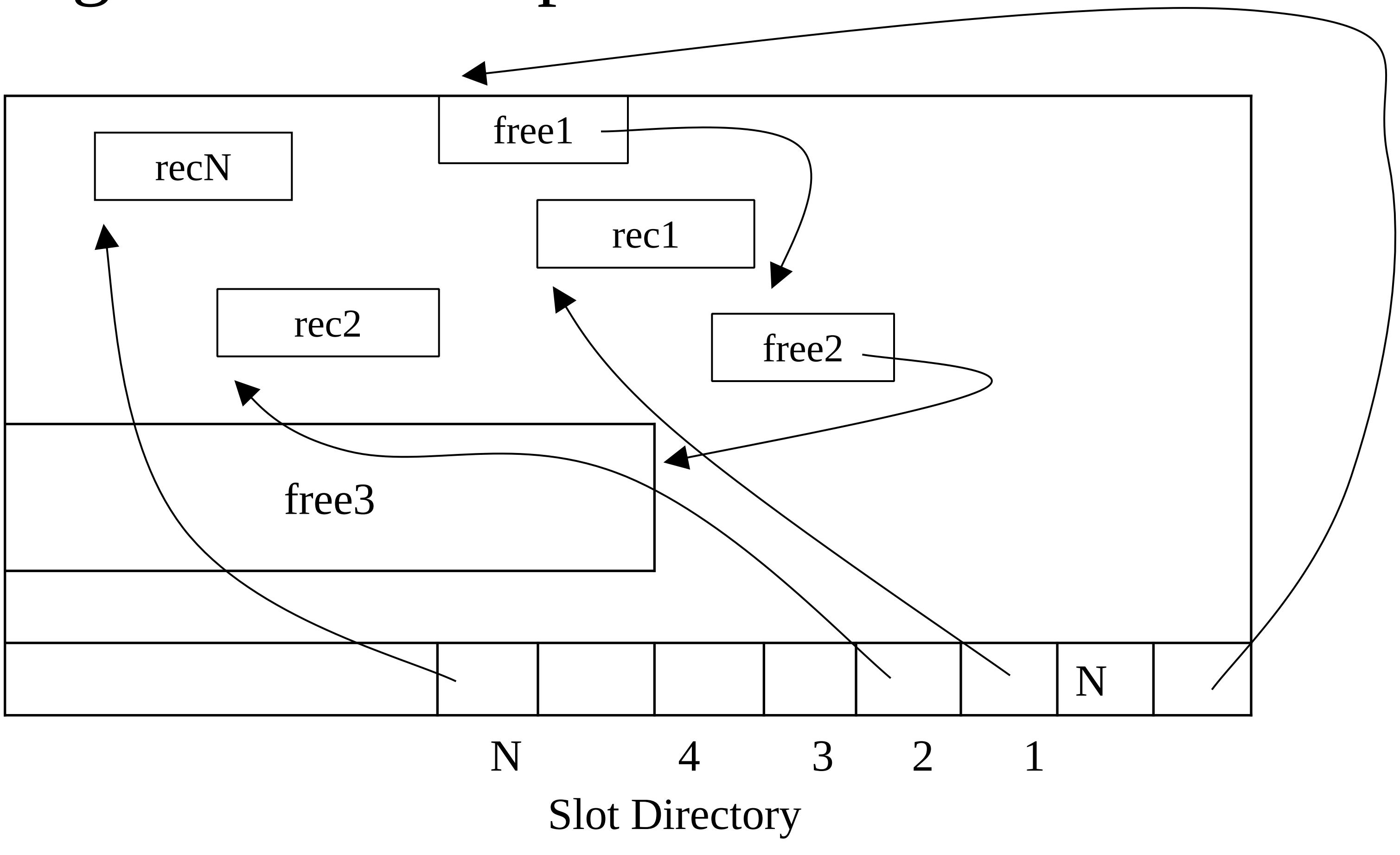
- Compacted free space:



- Note: “pointers” are implemented as offsets within block; allows block to be loaded anywhere in memory.

11.6.2 Variable-Length Records

- Fragmented free space:



11.6.2 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.

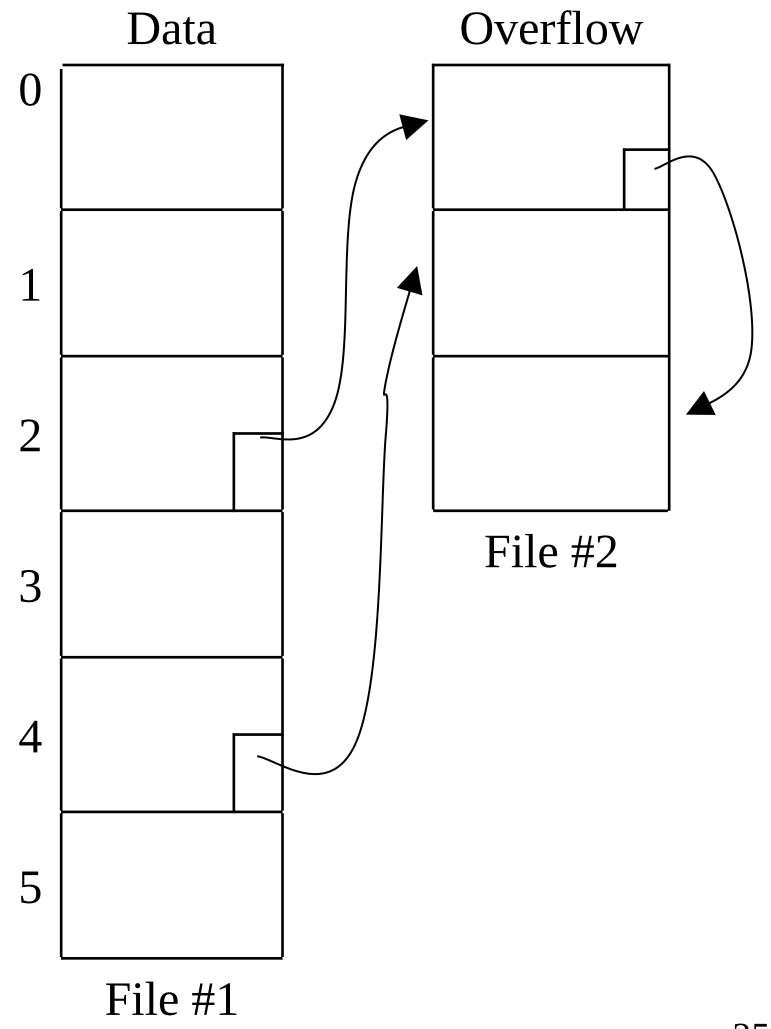
Introduce notion of *overflow blocks*:

- 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.

11.6.2 Variable-Length Records

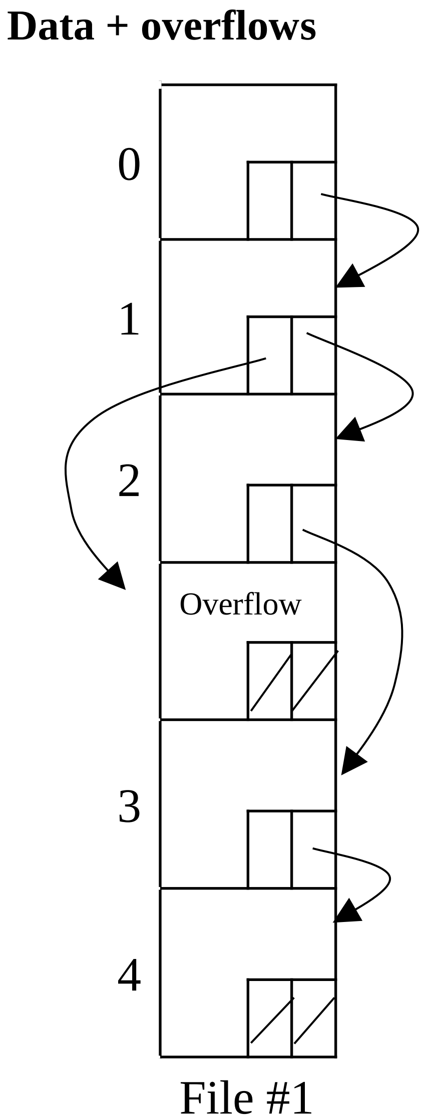
- Overflow blocks in a separate file:
- Note: “pointers” are implemented as file offsets.



11.6.2 Variable-Length Records

- Overflow blocks in a single file:

- Not suitable if accessing blocks via offset (e.g. hashing).



11.7 Files

A *file* consists of several data blocks.

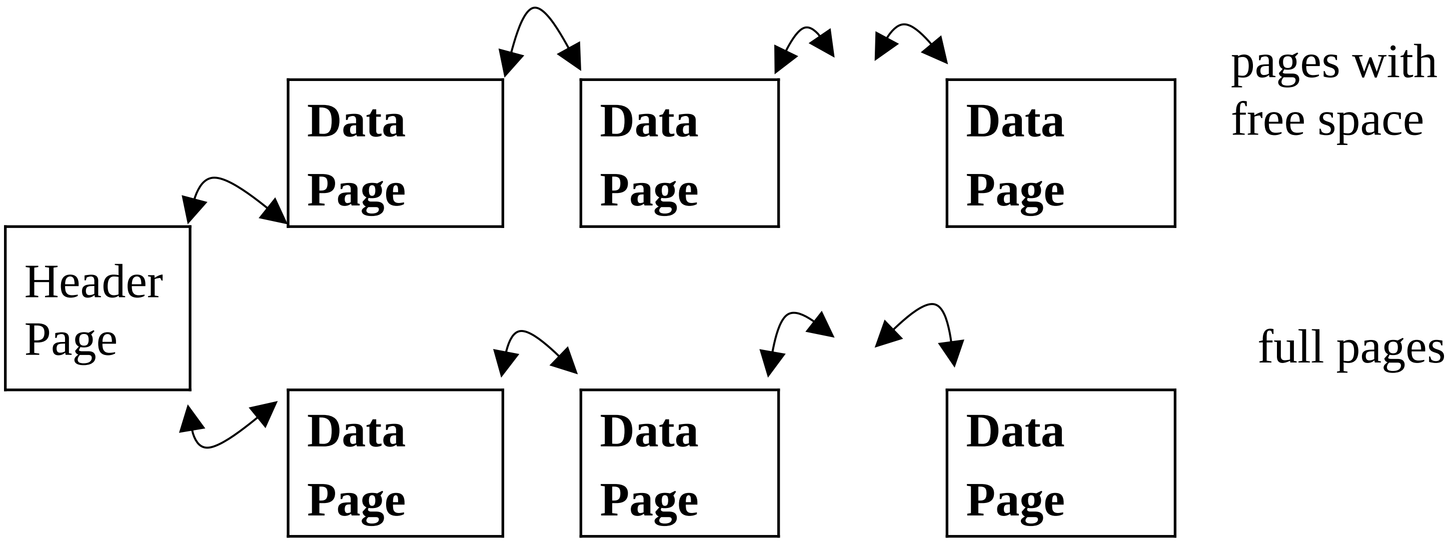
Heap Files: unordered pages (blocks).

Two alternatives to maintain the block information:

- Linked list of pages.
- Directory of pages.

11.7.1 Linked List of Pages

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



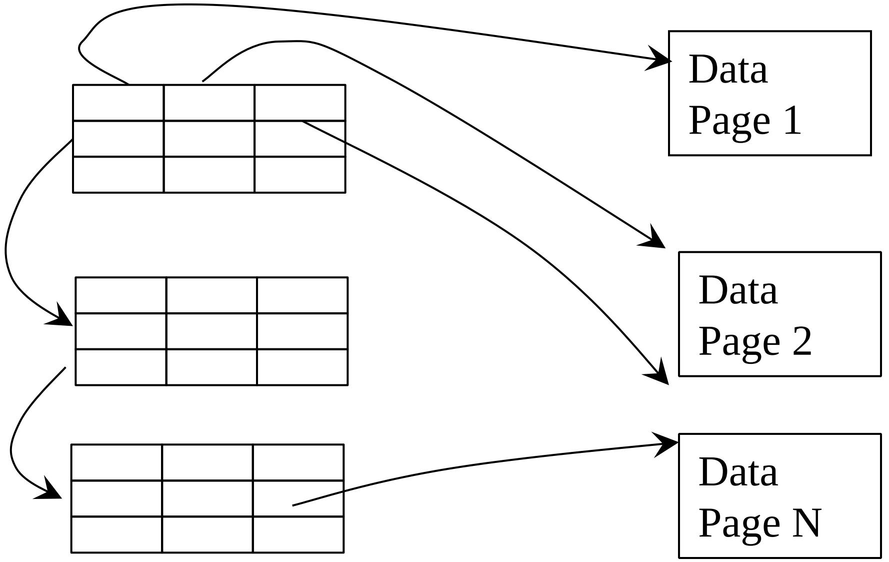
Organized by a Linked List

- **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.

11.7.2 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.



Organized with a Directory