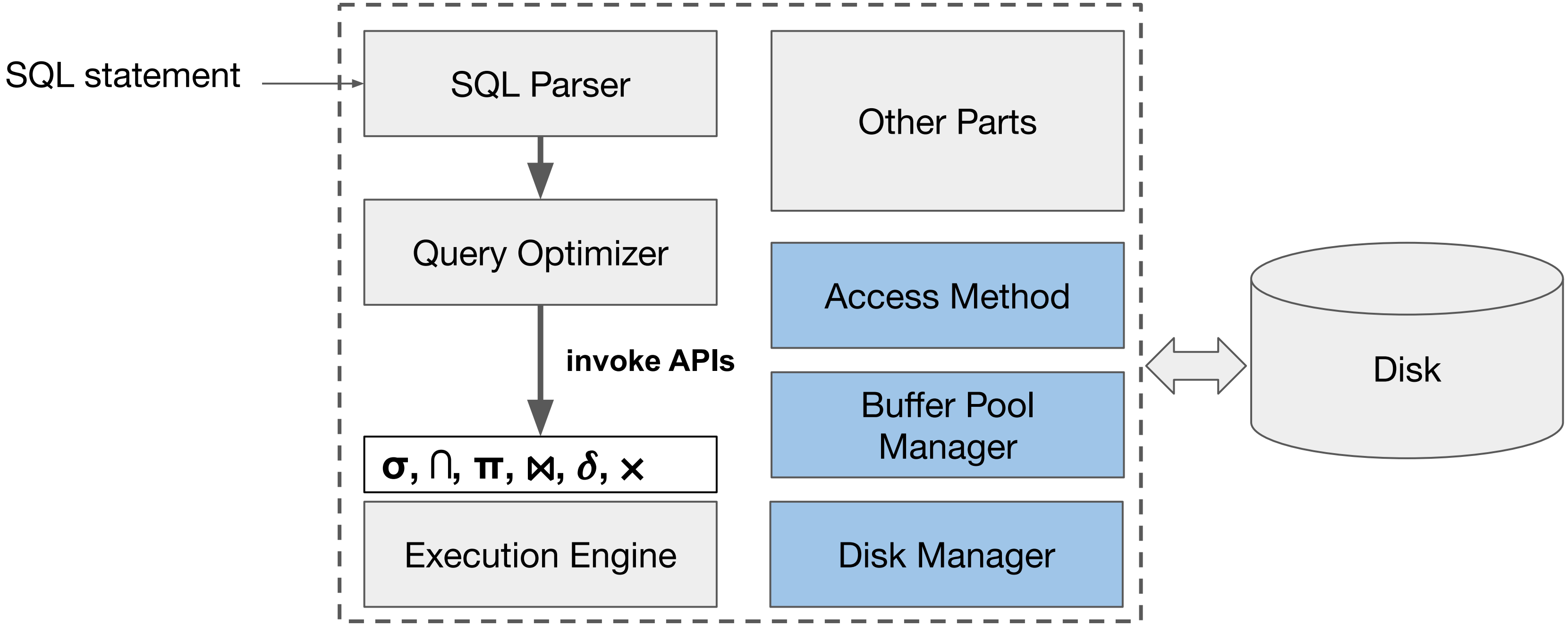
Database Systems

Lab 5

Today

- Recap
- Storage & Index

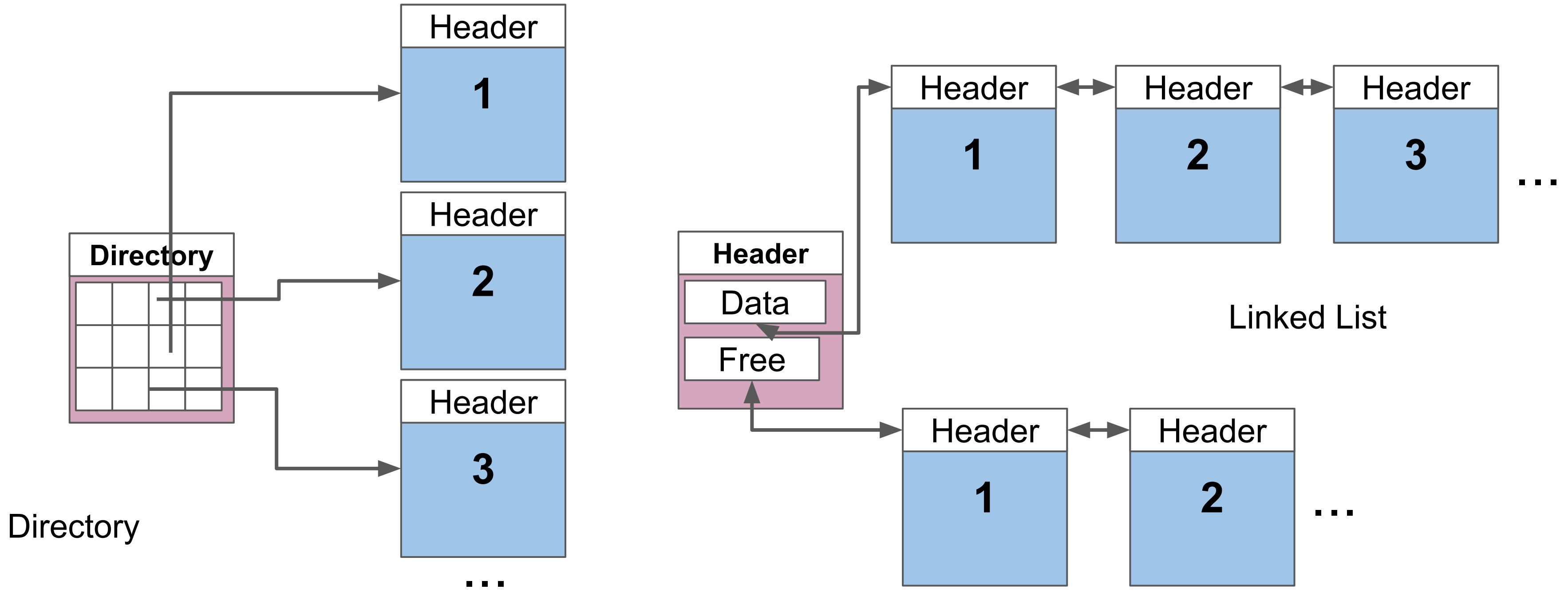
Database Internal



Disk Manager

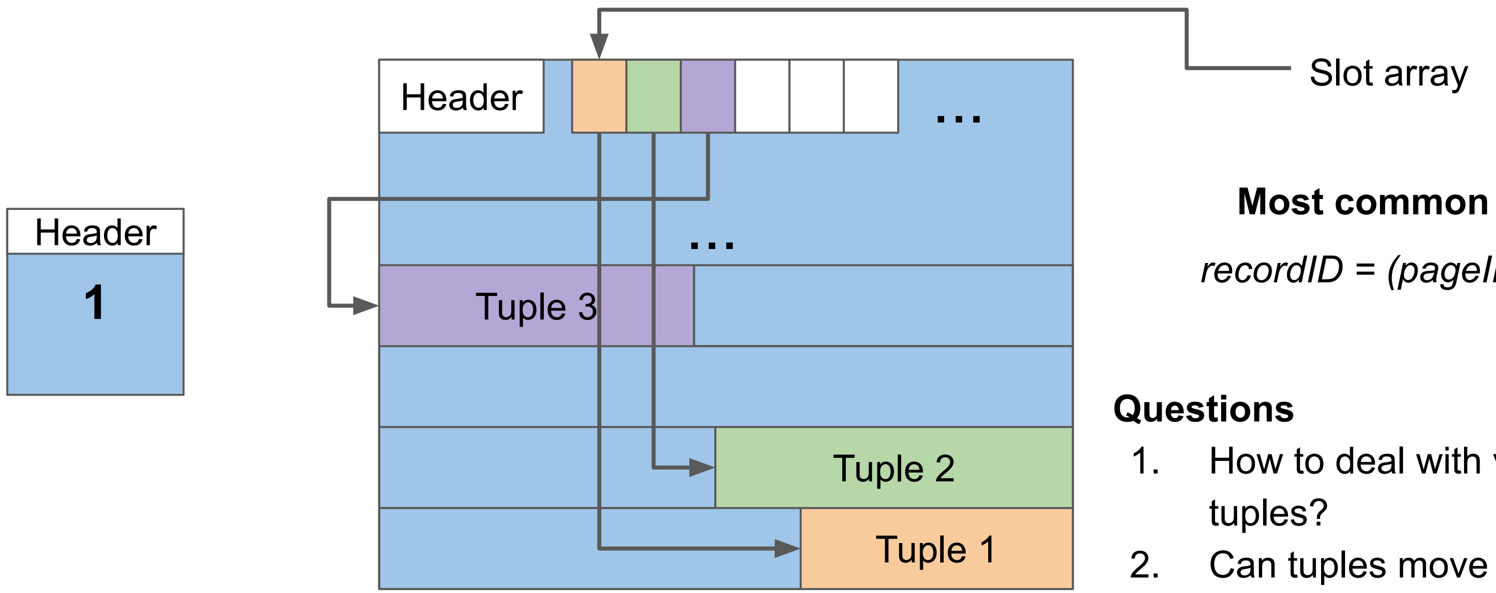
- How to organize pages
  - **Heap file:** *unordered* collection of pages

Not to be confused with Heap data structure



Disk Manager

- How to organize data in a page?



Most common layout

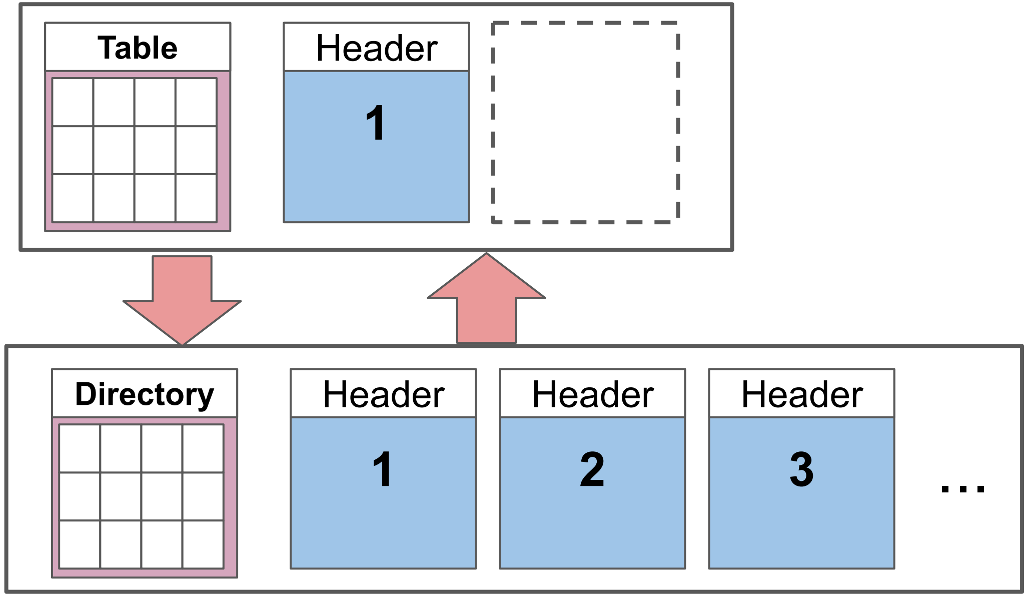
recordID = (pageID, slotID)

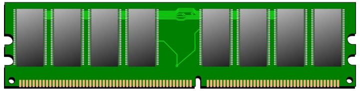
Questions

1. How to deal with variable-length tuples?
2. Can tuples move around?

Buffer Pool

- Problem: how to manage the limited amount of memory
  - Illusion of working with data in memory
  - How to move data back and forth from disk





Buffer Pool

- Dirty Frame
- Pinned Frame

FrameId	PageId	Dirty?	Pin Count
1	1	N	0
2	2	Y	1
3	3	N	0
4	6	N	2
5	4	N	0
6	5	N	0

Pin a page to the pool

- If page is in the pool, increment `pincount`
- Else:
    - Find one frame with `pincount = 0`
        - If dirty, write to disk
        - Load page to this frame, `dirty=N`
        - Increment `pincount`
- Return the frame

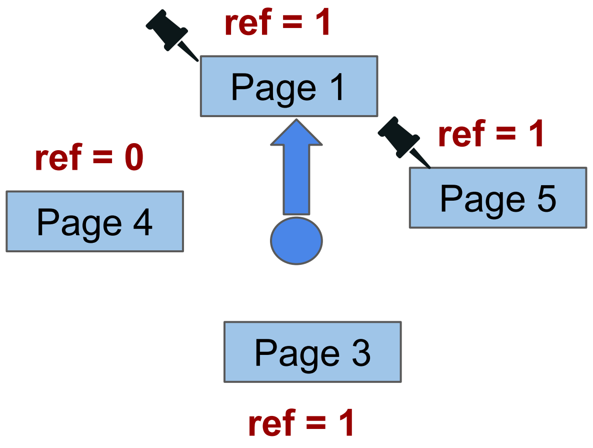
pinPage(6) = ?
pinPage(7) = ?

Unpin a page

- If page is not in the pool, do nothing
- Else, decrease the corresponding `pincount`

Buffer Pool

- Clock policy
  - Simple approximation of LRU, with a clock hand
  - Use reference bit instead of timestamp



Pinning a page

If found in the pool:

    Set `ref=1`; increment `pincount`.
    Return frame

Else, repeat the following

    If current frame X is pinned, advance hand
    Else
        If `ref=1`, set to 0 and advance hand
        Else
            Load the new page to X
            Set `ref=pincount=1`
            Advance hand.
            Return frame X

Buffer Pool

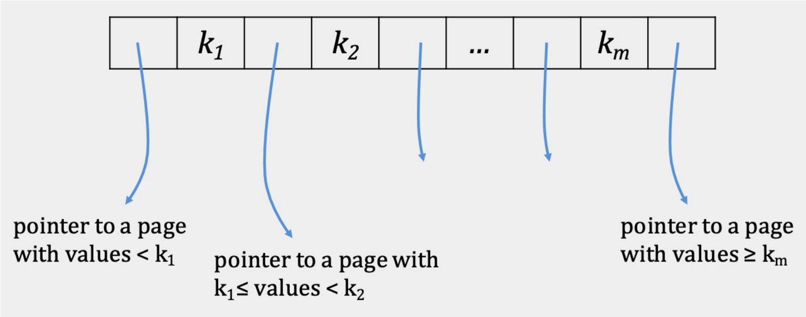
- API:
  - Load/pin a page
  - Release/unpin a page
- When talk about page replacement policy:
  - Access pattern: <Page 1, Page 2, Page 3,...>
  - No pinning by default:
    - Means that pages are loaded, then released immediately.

Access Method

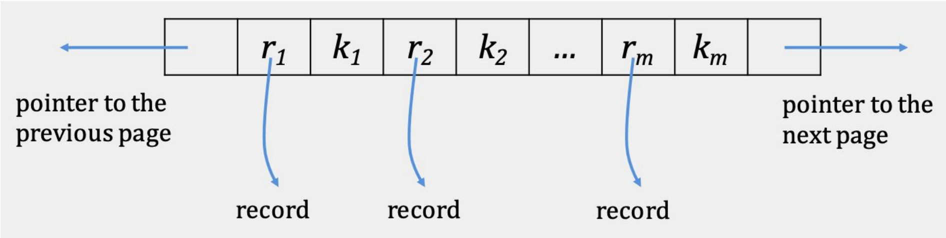
- Access methods :
  - Data structures and algorithms to access data
- Heap file
- Hash:
  - Cuckoo, chained, extendible

B+ Tree

- B+ tree:
  - Perfectly balanced
  - Clustered:
    - Heap file is sorted on the index attribute
  - Unclustered:
    - Heap file not sorted by the index attribute



Non-leaf node



Leaf node

B+ Tree

- Insert: O(logN)

- find correct leaf node **L**
- insert data entry in **L**
  - If **L** has enough space, DONE!
  - Else, we must split **L** (into **L** and a new node **L'**)
    - redistribute entries evenly, **copy up** the middle key
    - insert index entry pointing to **L'** into parent of **L**
- This can propagate recursively to other nodes!
  - to split a non-leaf node, redistribute entries evenly, but **push up** the middle key

B+ Tree

- Delete: O(logN)

- find leaf node **L** where entry belongs
- remove the entry
  - If **L** is at least half-full, DONE!
  - If **L** has only *d-1* entries,
    - Try to **re-distribute**, borrowing from sibling
    - If re-distribution fails, **merge L** and sibling
- If a merge occurred, we must delete an entry from the parent of **L**

Merge could propagate to root, decreasing height

Exercise 1

You have 4 pages in the buffer pool. Given this access pattern (buffer is empty at start)

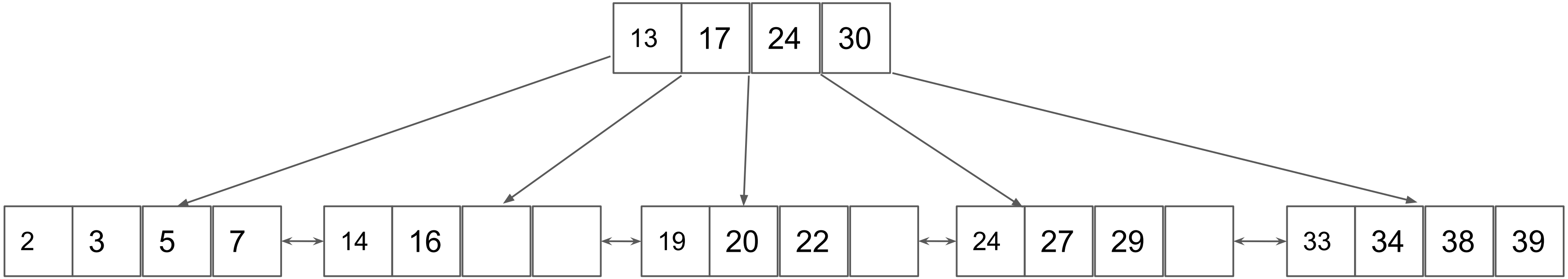
A B C D A F A D G D G E D F

1. What is the hit rate if you use LRU policy? Show the final state of the buffer pool.

2. Same question, but for MRU policy.

3. When would MRU be better than LRU?

Exercise 2



- Given the B+ tree above, **d=2**
  - Draw the tree after inserting **13, 15, 18, 25, 4** then deleting **4, 25, 18, 15, 13**
  - What did you observe?

Exercise 3

Consider an extendible hashing scheme:

- Hash function is the binary representation of the key
- Starting from an empty table, insert **15,3,7,14**
- Draw the final table hash table, including the slot array