CSE 541: Database Systems I

Data Access Methods

Slides Credit: Tianzheng Wang, Ryan Stutsman

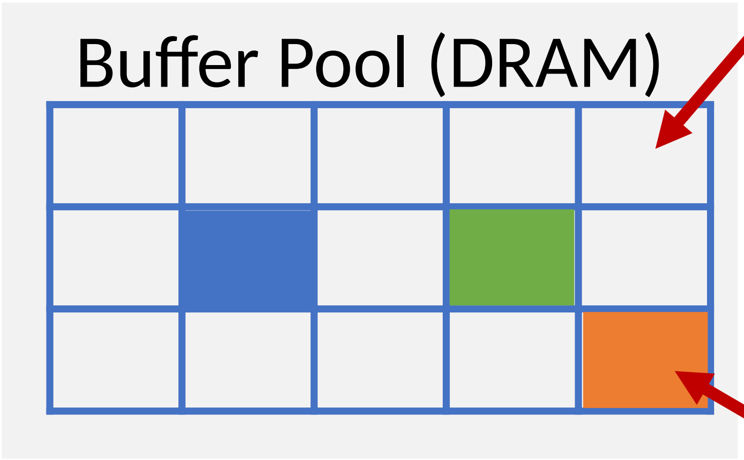
Accessing Data

- CPU has no idea about storage device!

- DB records must be brought to main memory before access
  ➔ Bring pages that contain the needed records to memory
  ➔ Evict pages from memory when they are no longer needed

Facilitated through the Buffer Pool

Buffer Pool

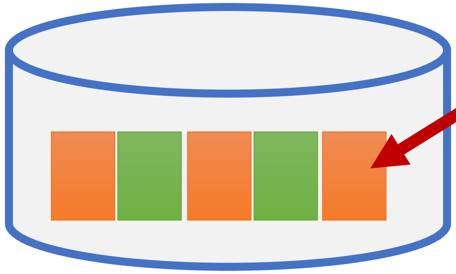


Frame

- Memory space to hold a page
- Same size as a storage page
- A pin_count to indicate how many requestors are using the page
- A dirty bit to denote it is modified

An occupied frame with a page

↕ via some *replacement policy*



Persistent storage

Page: Basic storage unit

- Typically, 4KB, 8KB, or 16KB
- Tunable database parameter

DBMS Buffer Pool vs. OS Buffer Cache

- Much similarity between OS and DBMS
  - OS: files on disk, buffer cache in memory
  - DBMS: database (files) on disk, buffer pool in memory
- So why not use OS buffer cache directly?
  - Because DBMS knows about the workload
  - Predict the access patterns for prefetching, customized replacement policies
  - Need the abilities to write-back (force) data pages to storage
- Other limitations of OS cache
  - Portability: something works for Linux may not for Windows
  - OS files usually cannot span disks

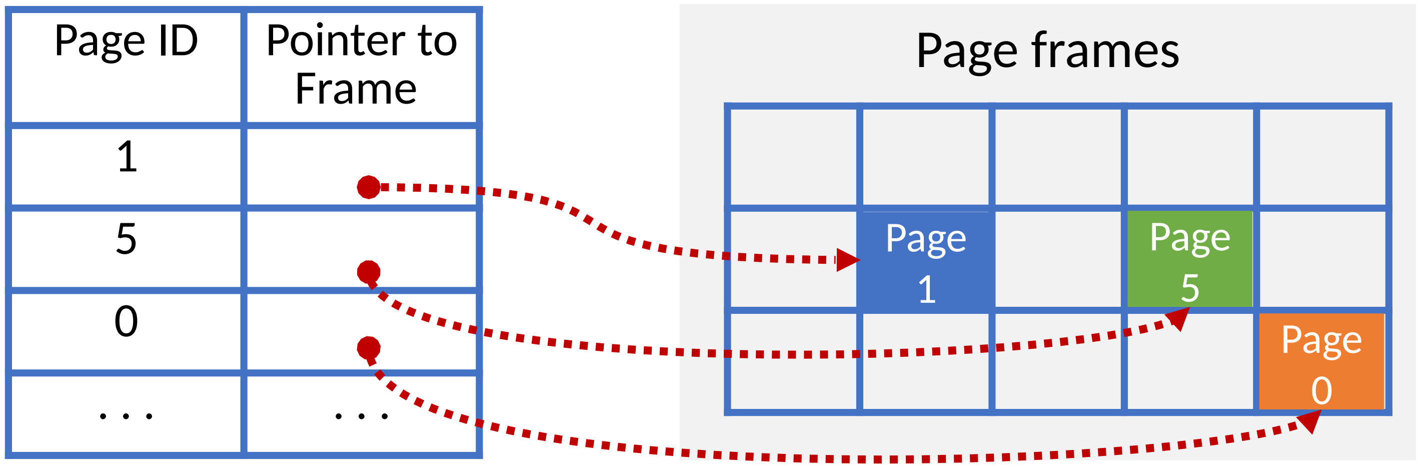
Handling Page Requests

1. Already in buffer pool?
   - Yes – increment pin_count and return it, done.
   - No – continue to the next step
2. Choose a page frame to accommodate the page
   - Vacant frame available?
     - Yes – use it
     - No – evict a page using some page replacement policy
3. Load the page from storage to the chosen page frame
   - Increment pin_count by 1 and return

- If requests can be predicted (e.g., sequential scans), page can be prefetched to improve performance
- After using the page, unpin it by decrementing pin_count

Buffer Pool Structure

- Higher levels (e.g., transactions) use page IDs and record IDs to access data
- Need mapping between page ID and buffer pool page frame
- Typically a concurrent hash table, e.g., C++ map with locking



Hash table (conceptually)

Buffer Pool Structure

The buffer pool is a global component (logically)

- All accesses depend on it, need to support concurrent access

Two solutions that can be combined:

Solution 1: concurrent hash table to support multiple threads

- Fine-grained locks
- Lock-free
- A global mutex (simple but slow!)

Solution 2: use multiple buffer pools

- E.g., partition by table, page set, database, user. . .
- Can help reduce contention on each hash table

Buffer Frame Replacement Policies

Buffer pool usually <<< data size

- Some pages may have to be evicted to make room for new pages
- Big impact on performance, depending on access pattern (workload)
- Many different policies, each suitable in different situations
  - Random
  - First-in-first-out (FIFO)
  - **Least Recently Used (LRU)**
  - **Clock Replacement**
  - Most Recently Used (MRU)

Least Recently Used (LRU)

Evict the page that has not been used for the longest time

- A possible implementation
  - Maintain a queue of pointers to page frames whose pin_count is zero
    - i.e., no one is using it, otherwise cannot evict
  - When a page's pin_count becomes 0, enqueue at tail.
    - i.e., it becomes a candidate for eviction
  - Upon page access, If page frame exists in the list, remove it
    - i.e., no longer a candidate for eviction
  - Queue head == lease recently used
    - Always evict the page in the frame pointed to by the queue head

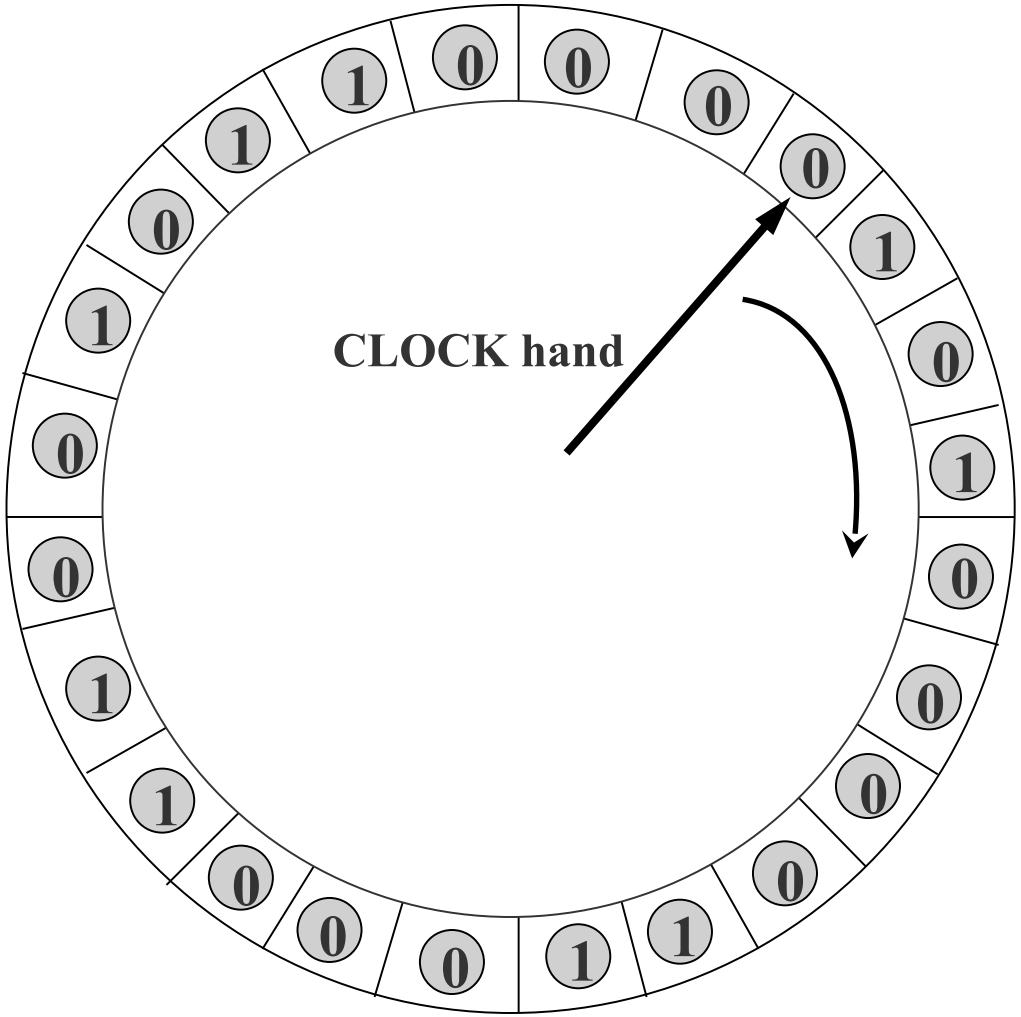
- **Why this might be slow??**

Clock Replacement

- Approximate LRU
- “Second Chance”

- On miss, move hand.
- When hand sweep through:
  - If bit is set, clear it.
  - If bit is clear, evict it.

- On hits, set bit.
- Why this is somehow faster than LRU?

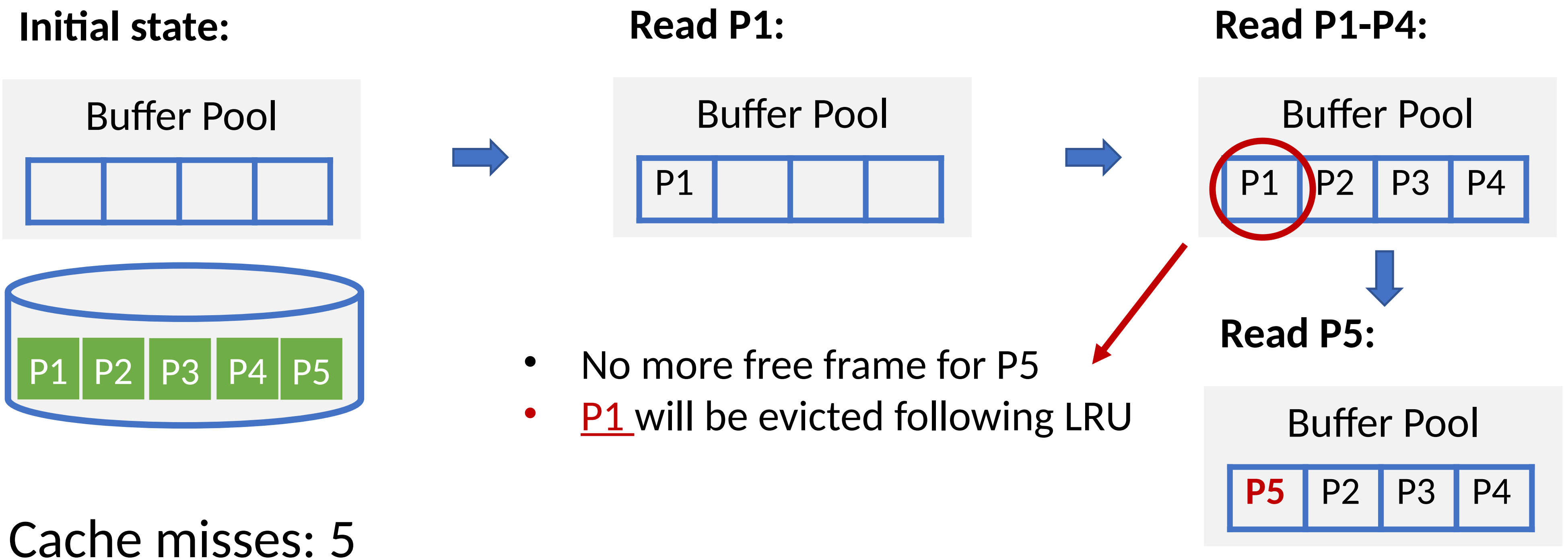


When can LRU/Clock be very bad??

Sequential Flooding in LRU

Example: repeated scans (a common operation in DBMS)

- Query: scan pages 1, 2, 3, 4, and 5

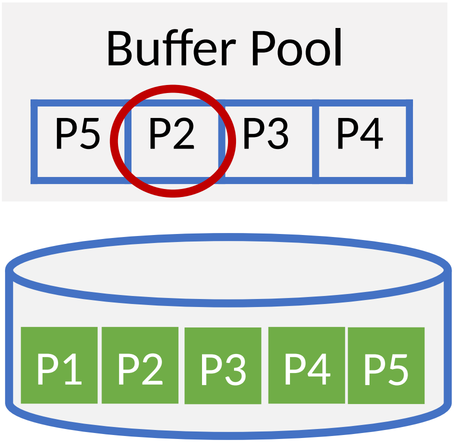


- Cache misses: 5
- Cache hits: 0

Sequential Flooding in LRU

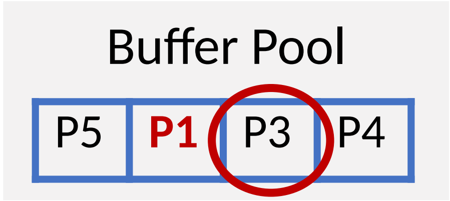
- Now do the same query again: scan pages 1, 2, 3, 4, and 5

Initial state:



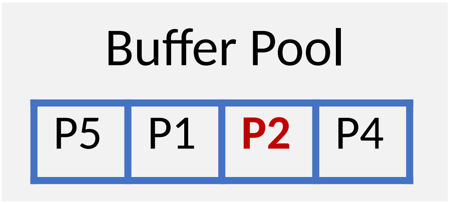
Read P1:

- No free frame, need to evict
- P2 will be evicted



Read P2:

- Need to evict P3
- **P2 was just evicted**



- Cache misses: 5 (previous query) + 5 (new)
- Cache hits: 0

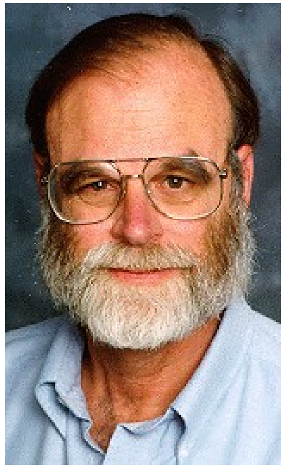
Keep evicting the next needed page
(Everything equally old)
Clock replacement suffers from this, too

What should be in-memory??

- **Placing data in memory can save I/O.**
- **Is it always a good idea to do that?**

- Neither memory nor disk is free!
- Costs ($$$) associated with
  - **Storing** data in memory
  - **Accessing** data in storage (e.g., disk)

- The Five-Minute Rule
  - Pages referenced every 5 minutes should be kept in memory



Jim Gray

The Five-Minute Rule (1987)

- Assumption: page size is 1KB
- Hardware: Disk and DRAM
- Disk speed: 15 accesses/second, at $15k cost
  - Plus extra CPU/controller cost: **$2000 for 1 access per second**
  - $2000 / 2 = $1000 if we access once every two seconds
    - Or if we do “0.5 accesses” per second
  - $2000 / 10 = $100 if we access once every 10 seconds

  ➔ Cost of disk access is $2000 / accessInterval
- Memory (DRAM): $5000 for 1MB → **$5 for 1KB**
  - Keeping a 1KB page in DRAM costs $5
- Break even point: when $5 == $2000 / accessInterval
  - accessInterval = 400 in this case ➔ roughly 5 minutes

The Five-Minute Rule (1987)

- RI: expected interval in seconds between page references
- A$: cost (price) of one disk access per second ($/access/s)
- M$: cost (price) of memory per byte ($ per byte)
- B: page size to be referenced (unit of I/O)

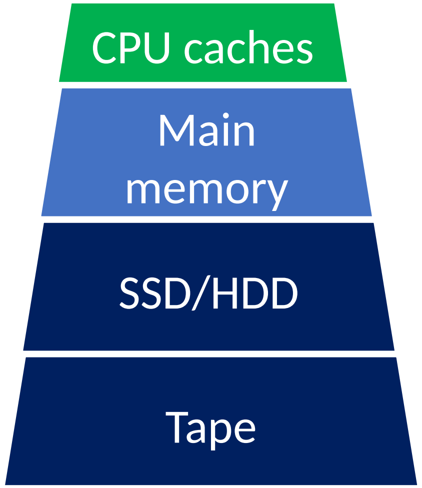
$$M\$ \times B = \frac{A\$}{RI} \Rightarrow RI = \frac{A\$}{M\$ \times B}$$

Cost to keep data in DRAM

Cost to access data in disk

- Pay A$ every RI seconds

The Five-Minute Rule (post-1987)



- Everything is cheaper and better
- New additions to the storage hierarchy
  - Flash/SSDs
  - Persistent memory*
- Is five-minute rule still should be “five-minute”?

See also:

- *The five-minute rule ten years later, and other computer storage rules of thumb*, SIGMOD 1997
- *The Five-minute Rule: 20 Years Later and How Flash Memory Changes the Rules*, CACM 2007
- *The five-minute rule thirty years later*, CACM, 2019

Indexing

Indexing

Conceptual view:

StudentID	Name	GPA
100	John	3.0
101	Jack	3.0
102	May	3.5

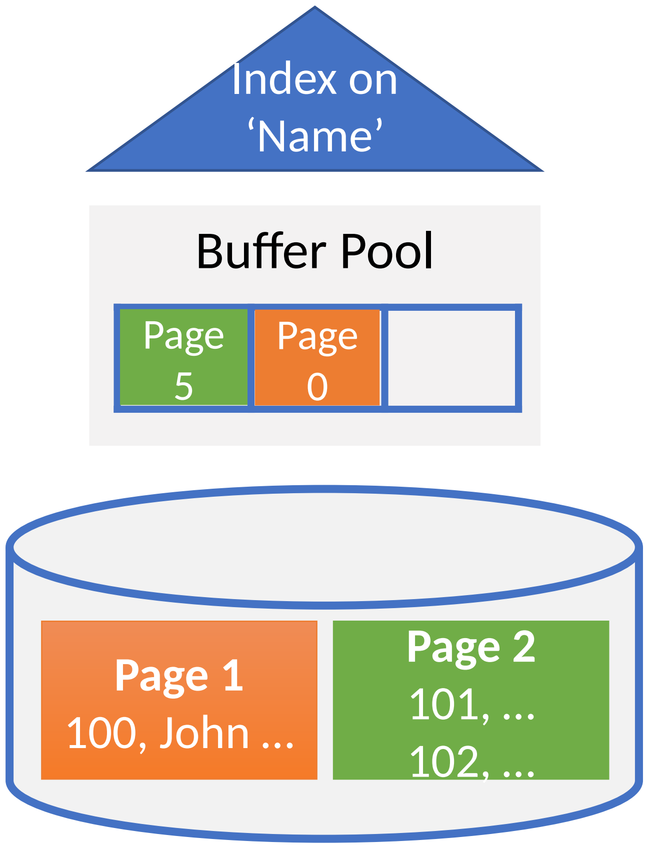
Q: How to find a particular row?

- E.g., Get John's record

Solutions:

- Full table scan – can be slow and wasteful, unless reading many records
- Indexing – map keys to data entries

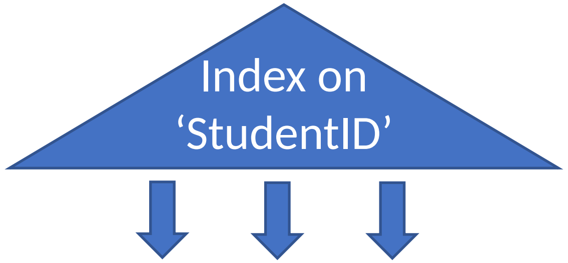
Physical implementation:



Index

Data structure that optimize data retrieval operations

- Map search keys to values (aka data entries)
- Allow fast retrieval of all records satisfying the search conditions on the search key fields
  - Search key can be values of a single or multiple fields
  - Input: key
  - Output: record or location of a record
- Typical types
  - Hash table
  - Trees



StudentID	Name	GPA
100	John	3.0
101	Jack	3.0
102	May	3.5

Index Operations

Insert: establish key-value mapping

Read/Get: retrieve value for a given key

Update/Put: update the value of a given key

Range Scan: return all values that for a range of keys

- Range can be [start key, end key] or [start key, number of records to scan]
- Forward scan: scan from smaller to larger keys
- Reverse scan: scan from larger to smaller keys
- **Typically not supported by hash tables**

Index Types

Unique Index: One key maps to exactly one value, no duplicates allowed

Primary index: index on a set of fields that includes the primary key

- Must be a unique index: no duplicates
- No null values allowed

Secondary index: index other fields

Example:

- Unique index: index on StudentID
- Non-unique index: index on GPA

StudentID	Name	GPA
100	John	3.0
101	Jack	3.0
102	May	3.5

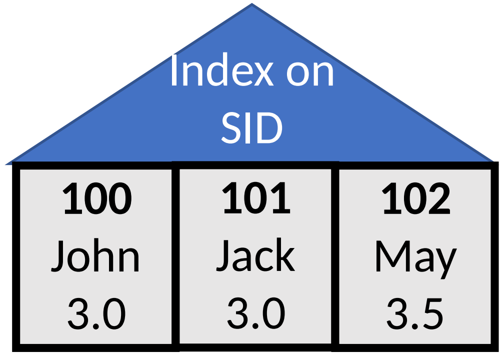
Data Entries – What to store as value?

Alternative 1: Actual record

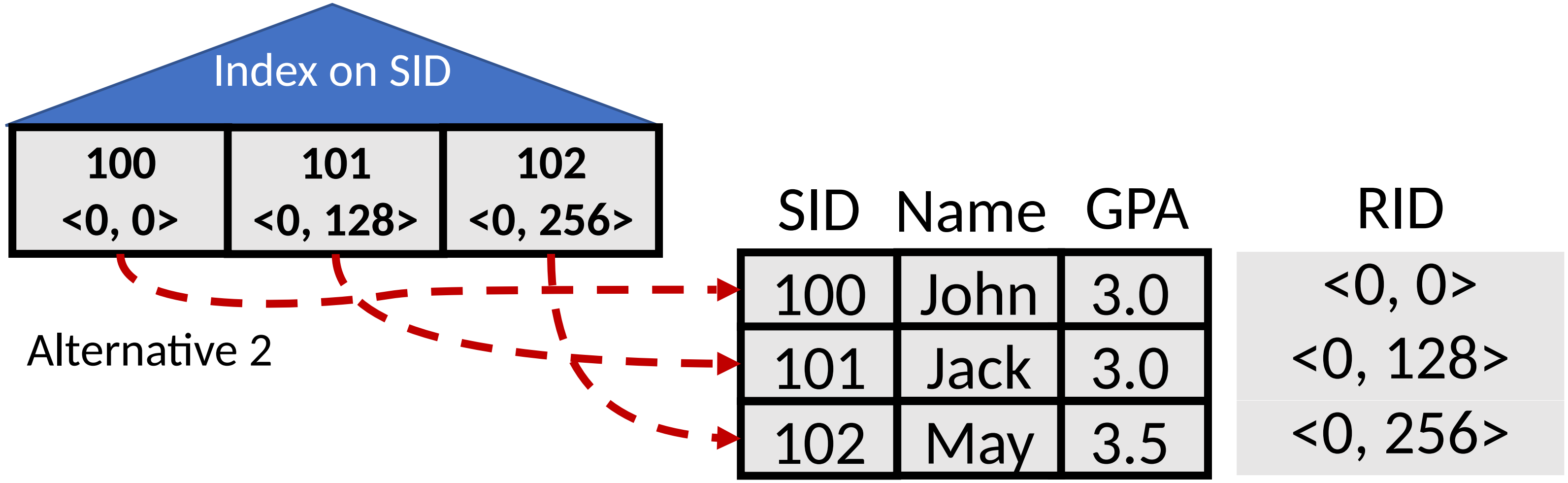
- A special file organization, index == file
- At most one such index per table
  - May need to duplicate data otherwise

Alternative 2: <key, RID>

- Map keys to RIDs
- Independent of the table's organization – may lose locality



Alternative 1



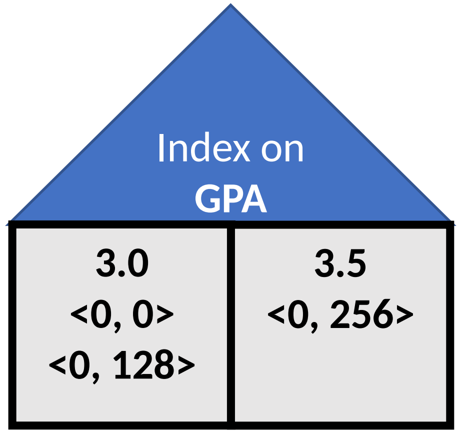
Alternative 2

Data Entries – What to store as value?

Alternative 3: <key, RID list>

- A list of records that match the search key
- Independent of the table's organization

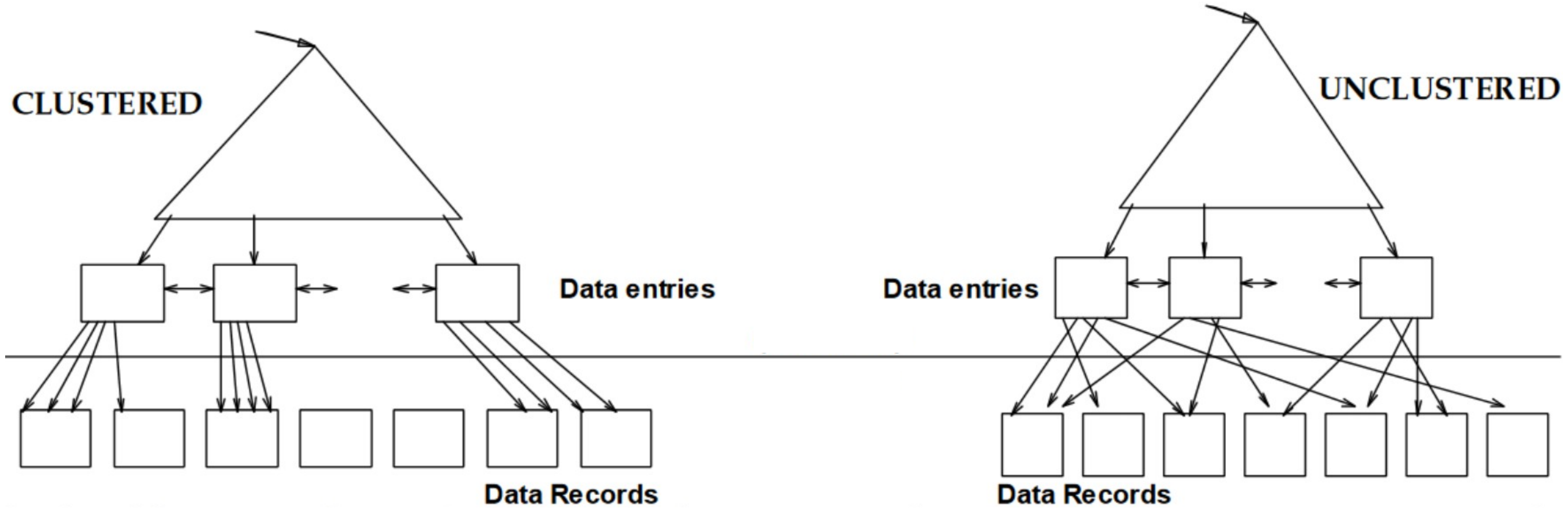
SID	Name	GPA	RID
100	John	3.0	<0, 0>
101	Jack	3.0	<0, 128>
102	May	3.5	<0, 256>



Alternatives 2 vs. 3:

- Basically the same for unique indexes
- For non-unique indexes, Alternative 3 provides better space utilization (no repeated key storage)

Clustered vs. Unclustered Index



Order in index == order in data file

- Better performance for scans (faster sequential reads)

Order in index != order in data file

- May be slower: potentially more random reads

Alternative 1 (key → actual record): always clustered

Alternative 2 and 3 (key → RIDs): could be clustered or non-clustered

Summary

- Buffer Pool
  - Operations and metadata: page pinning, dirty flag
    - May prefetch pages
  - Internal structure
    - Page frames, and hash table to map page IDs to page frames
    - Need proper concurrency handling
    - Can be partitioned
  - Replacement algorithms to minimize cache misses: LRU, Clock, MRU . . .
    - The sequential flooding problem under LRU and Clock
- Index optimizes data accesses
  - Map keys to records or record locations
  - Basic operations: insert, delete, search, scan, update
  - Unique vs. non-unique indexes, clustered vs. unclustered indexes