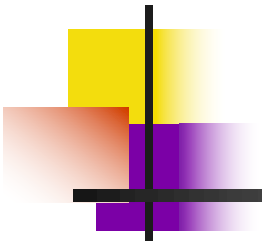
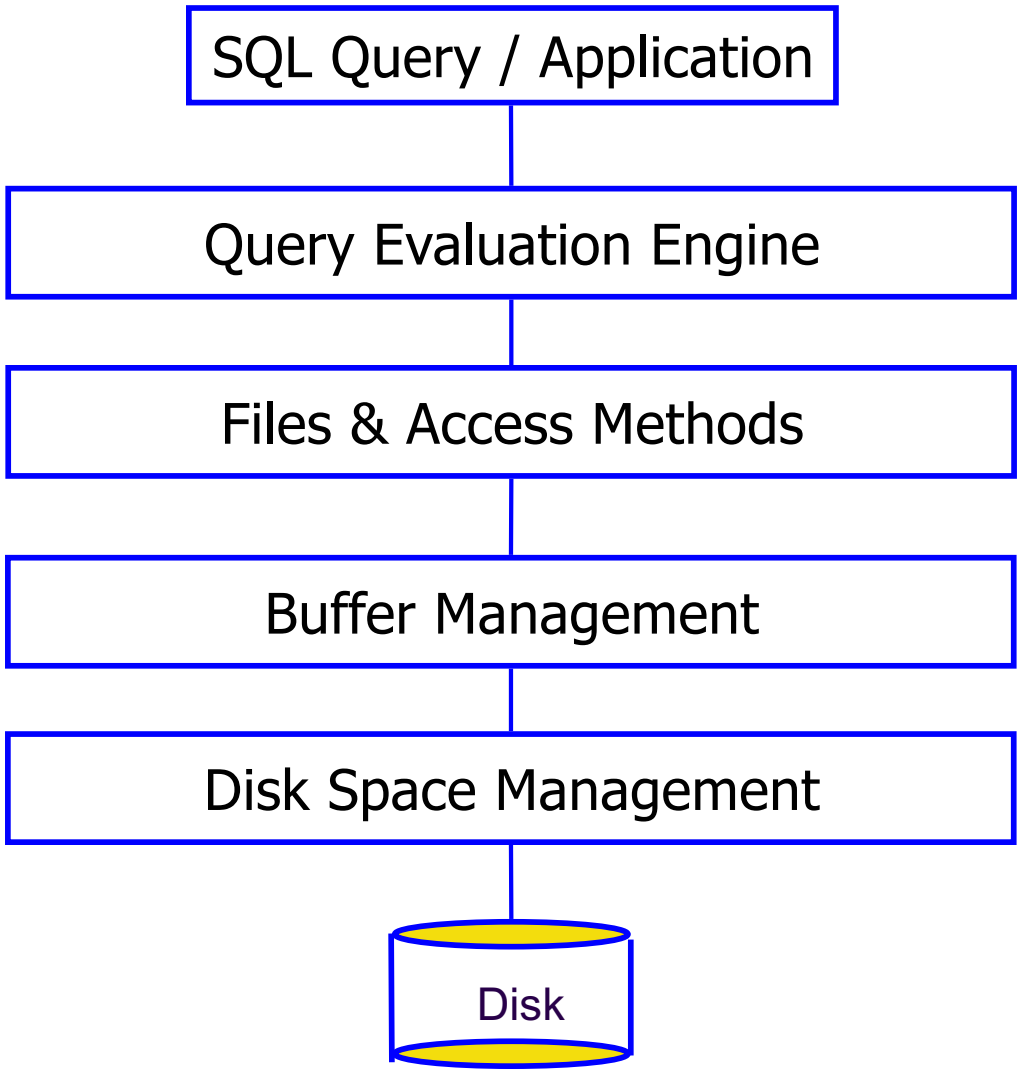


# Storage and Indexing

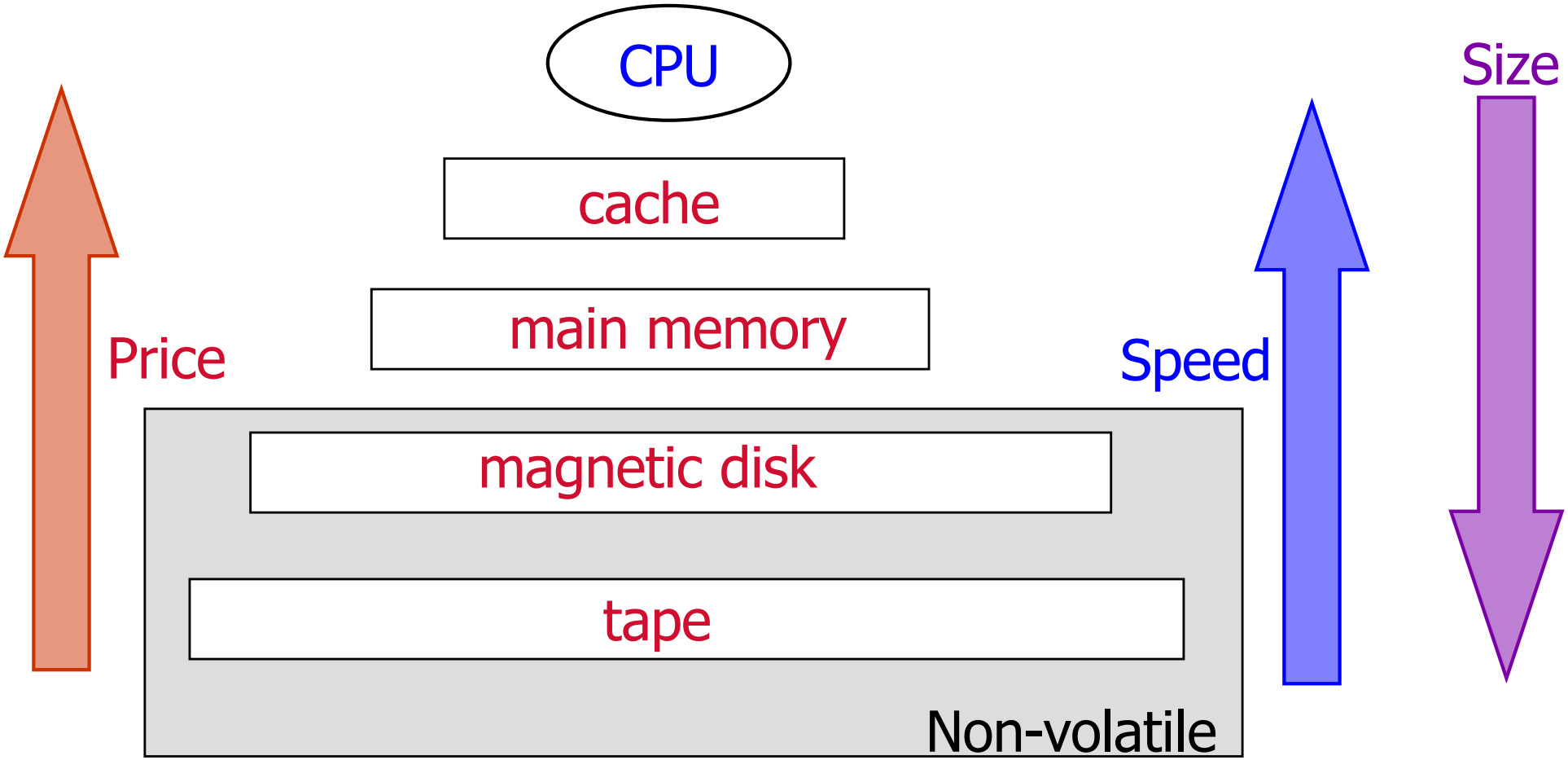
Chapter 8

# DBMS Organization



# The Memory Hierarchy

**Performance of Microprocessors and Memory improving faster than disks and tapes**

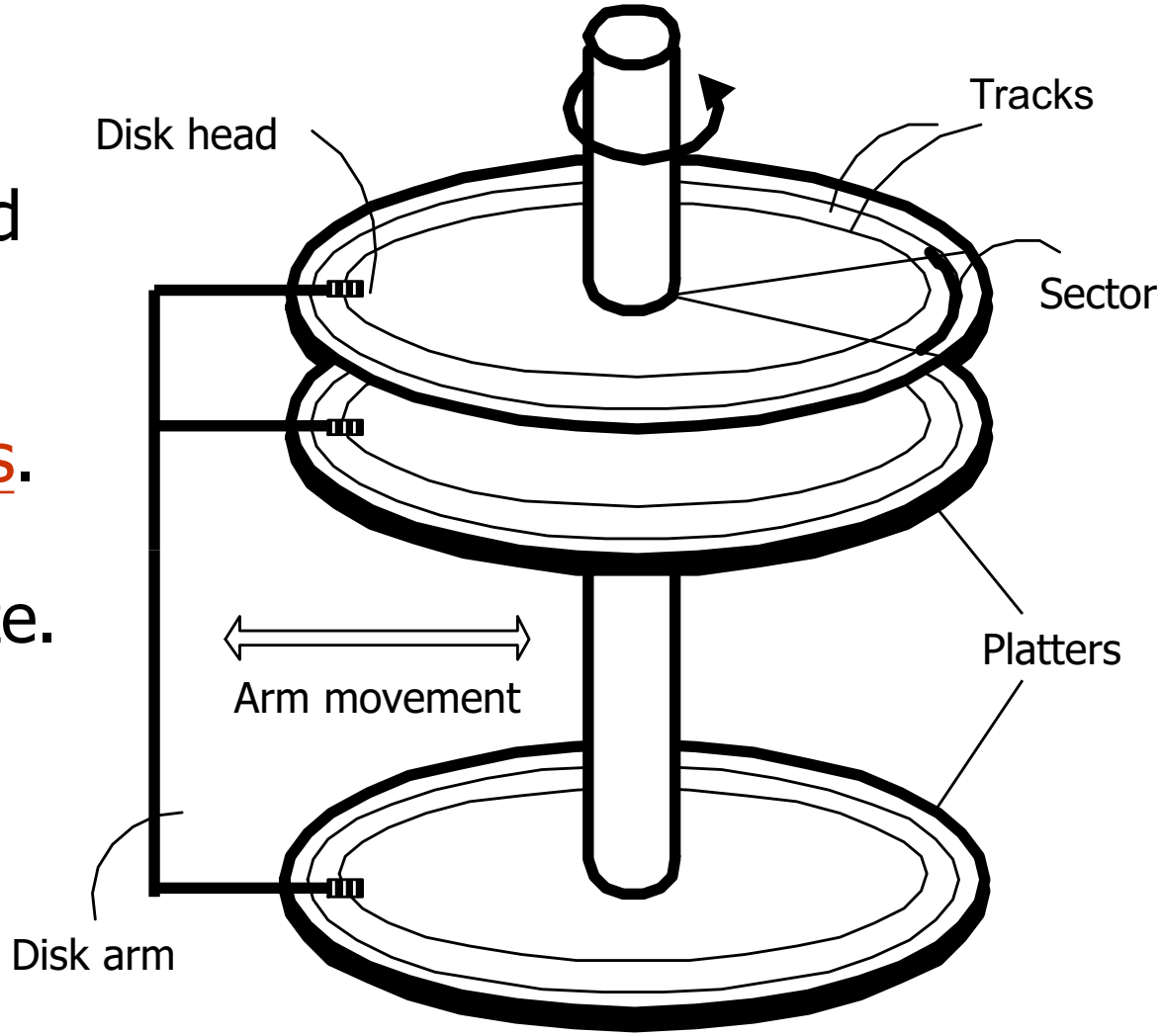


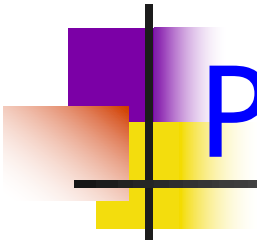
# Hard Drives

Set of tracks with same diameter called a cylinder

Data stored in blocks. Size of block is a multiple of sector size.

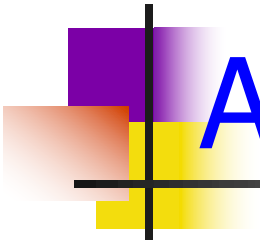
Only one disk head reads or writes at a time.





# Performance Implications

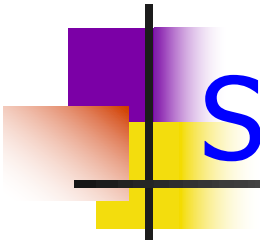
- Data must be in memory for DMBS to use.
- Unit of transfer is a block. Whole block must be transferred. Reading or writing a disk block is called an I/O.
- Disk geometry affects access time (hard drives)
  - Seek time: time to move disk head to appropriate track
  - Rotational delay: time waiting for block to move under disk head
  - Transfer time: time to read or write block once head is positioned



# Arranging Blocks on Disk

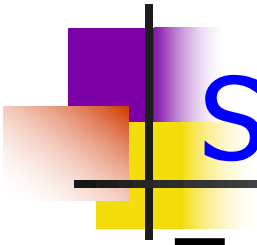
Access time = seek time + rotational delay + transfer time

- GOAL: Minimize seek time and rotational delay
- 'Next' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Arranging blocks so they are read and written *sequentially* is important to reducing time spent doing disk I/O



# Solid State Drives vs. Hard Drives

- Another kind of drive: solid state
  - 2016 Cost/TB: About $250 for SSD vs. $50 for hard drive (5x)
  - Lower latencies and higher throughput
  - Access times independent of block placement
  - Failures: a recent FAST 2016 paper from Google.
    - Both disks age. Planned lifetime: 4 years.
    - Fewer annual replacement rate for SSDs,
    - BUT, a higher rate of uncorrectable read errors with SSD



# Solid State Drives vs. Hard Drive

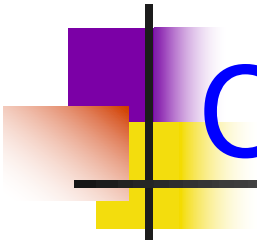
- Traditional disks
  - Sequential I/O faster than random I/O
- Solid-state drives becoming popular
  - Same speed: sequential or random I/O (Layout not relevant)
  - 5x higher cost, but lower latencies
  - Similar lifetime
- Most databases today still use traditional disks, but could change over time.
- Hybrid drives that use solid state drives as a cache also becoming common

# Why Not Store Everything in Main Memory?

- *Too expensive:* RAM costs around $40 for 8GB in 2016.
  - 20x cost per GB versus SSD and 100x per GB versus Hard Drive.
- *Main memory is volatile:* Want data to persist between runs
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data
  - Disk for the main database (secondary storage)
    - Non-volatile storage.
  - Tapes for archiving older versions of data (tertiary storage)
    - Sequential access devices

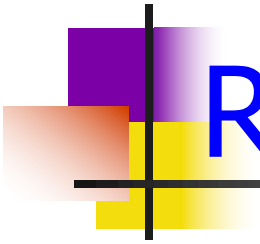
# Pages – Physical Abstraction

- Unit of information for disks is a page
  - Page size is typically 4KB to 16KB
- Data must be brought into memory from disk to be read/written
- Typically: Memory size << Disk size
- Why:
  - Cost: memory (100x), SSD(5x), HD (1x)
- Performance:
  - Main memory reference: 100ns
  - Read a 4K random block from SSD: 150,000 ns
  - Read 1MB sequentially.
    - from hard disk: 20ms
    - From SSD: 1 ms (20x slower)
    - From main memory: 0.25ms (about 4x slower than SSD)
- Arrange in a hierarchy to get the best speed at the lowest cost.



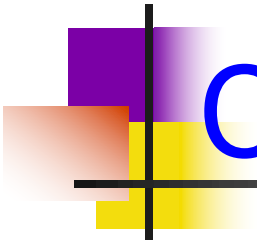
# Check your understanding

- Why do databases need disks? Why not just use main memory?
- Which has the lowest per GB cost?
  - Main memory
  - Solid state drives
  - Magnetic drives
- Of the above, for best speed, non-volatile storage, highest capacity, and lowest cost, which of the above would you use?



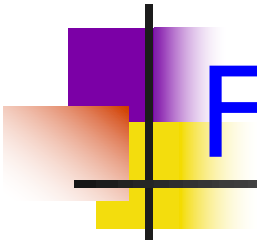
# Records

- Pages are physical units of data.
- How are tables stored in pages?
  - A table is stored as a *file of records*
  - Records are logical units
  - Records stored in pages
  - The term "file of records" here simply means a named set of pages, not necessarily an OS file.
- Typically, one page has multiple records
- One table (file) will typically require multiple pages



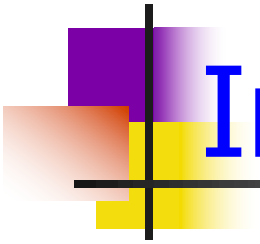
# Operations on File - Example

- Employees (Name, Age, Salary)
- Operations
  - Scan: Fetch all employees from disk
  - Equality Search: age = 21
  - Range Selection: age >= 18 AND age < 65
  - Insert a record
  - Delete a record
- *How do we organize the file to accomplish these tasks efficiently?*



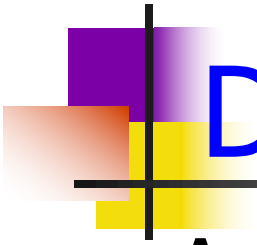
# File Organization

- Each record in a file has a unique Record ID, or RID, which is sufficient to locate the record on the disk.
  - E.g., an RID may be (page#, offset, length)
- Some methods of arranging file of records on disk
  - Heap (random order) files: No particular order defined for records
  - Sorted Files: Records sorted based on one or more attributes
  - Indexes: Organize records using trees or hashing



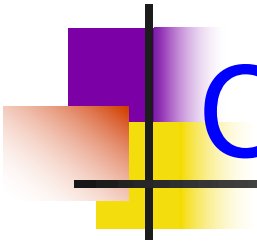
# Indexes

- A data structure that organizes data records on disk to optimize certain operations
- Speed up selections on the search key field of the index (denoted *k*)
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index file contains a collection of data entries k* for each search key value k
  - k* should allow us to get to the record contents



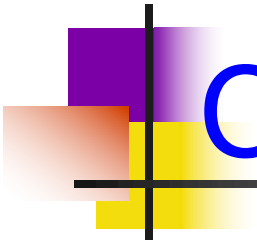
# Data Entries k*

- A data entry k* must give us a way to get to the data for k:
- Alternative 1: Data entry k* is an actual record (with search key k)
  - Index == File !
- Alternative 2: Data entry k* is (k, rid) pair, where rid refers to a record with search key k. Actual data records stored in a different file
- Alternative 3: Data entry k* is (k, rid-list) pair, where rid-list refers to list of records with search key k



# Choosing among alternatives

- Only one index can use alternative 1
  - Otherwise, you would get redundancy
  - Other indexes can be Alternative 2 or 3
- Alternative 3 is more compact than alternative 2, but leads to variable-length index entries



# Check Your Understanding

Can search key k be a composite value, e.g., pair of values or attributes?

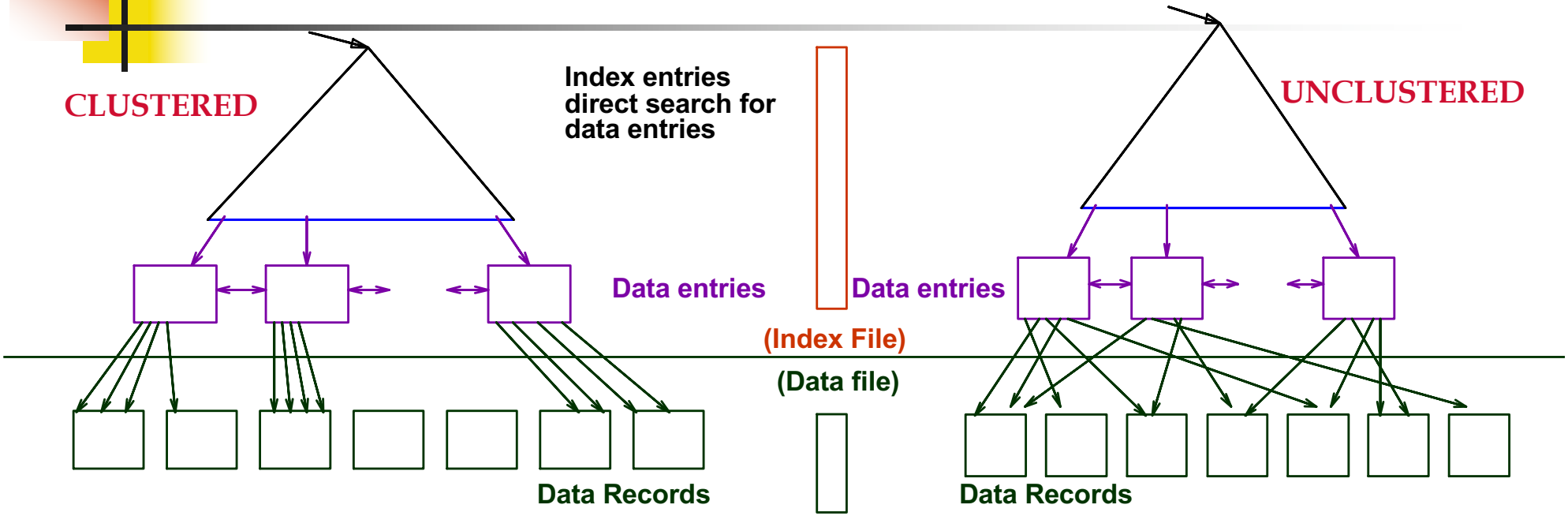
Can a file (relation) have more than one index?

Can a file (relation) have more than one index using data entry alternative 1?

# Indexing Terminology

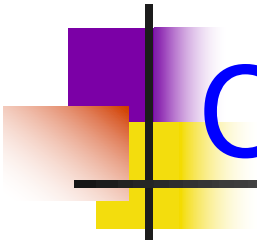
- *Primary* vs. *secondary index*: If search key contains primary key, then called primary index
  - Otherwise, called secondary index
- *Clustered* vs. *unclustered index*: If order of data records is the same as, or ‘close to’, order of data entries, then called clustered index.
  - Alternative 1 implies clustered index. A file that uses Alternative 1 is also called a *clustered file*.
  - But not every clustered index uses Alternative 1.
- A file can be clustered on at most one search key.

# Clustered vs. Unclustered Index




- Suppose: data records in heap file, index with Alt. 2
- To build clustered index, first sort the Heap file

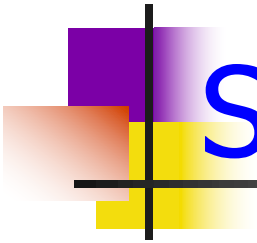
Suppose you have an index on Age, and you want to do a range selection (e.g., Age >= 18) Which index would you prefer? Why?



# Cost of a search query

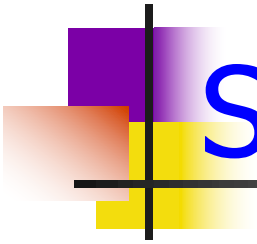
- Cost of Page I/O >> memory read
- # of pages read more important than # of records read

- Searching a heap (unordered)
  - All pages must be read.



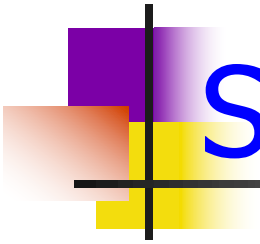
# Search on unclustered index

- Let's say you want to find records of U. of Michigan students who are 18 years old.
- Let's say you have an unclustered index on age and 25% of students are 18 years old
- Each page can hold 10 records
- 100,000 records
- How many pages are you likely to read?
  - Close to 100%, i.e., 10,000 pages?
  - Close to 25%, i.e., 2,500 pages?



# Search on clustered index

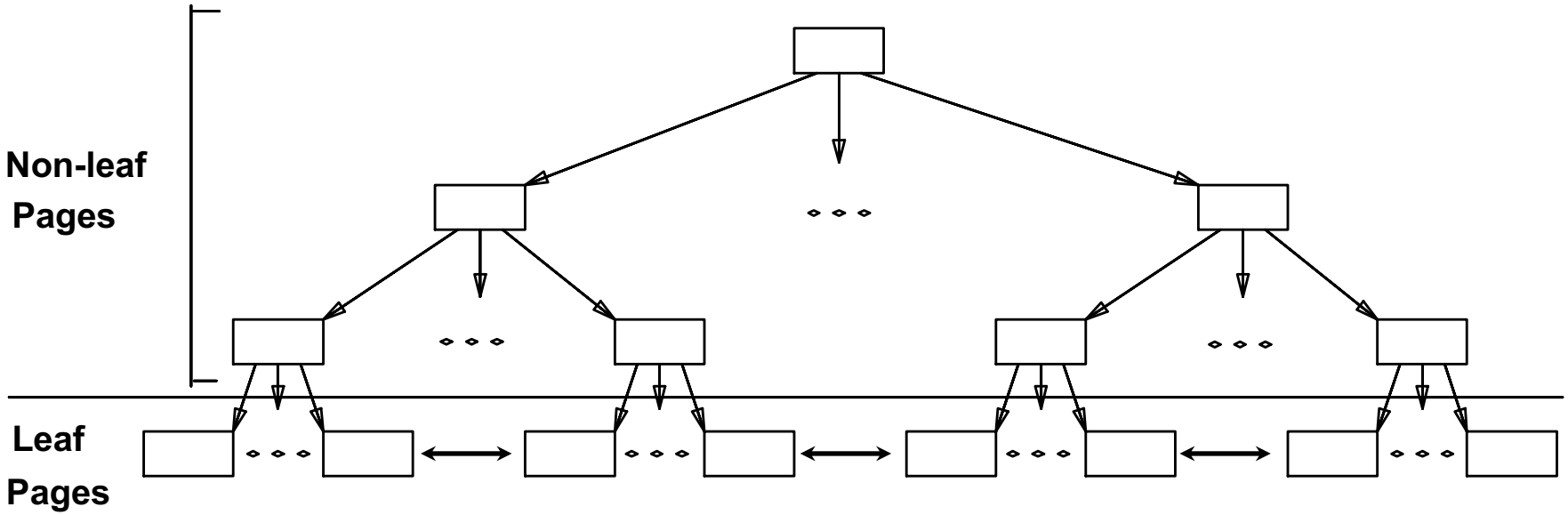
- Same problem and paramenters
- Let's say you have a clustered index on age What percent of pages are you likely to read, taking advantage of the index?
  - Almost all, i.e., 10,000 pages?
  - About 25%, i.e., 2,500 pages?



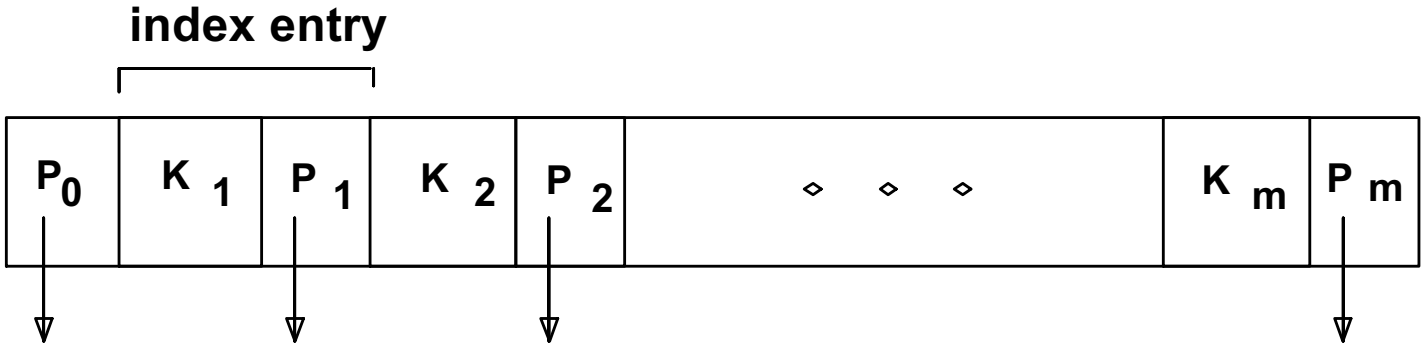
# Search on clustered index

- In practice, pages are often only around 2/3rd full because deleting records will leave holes in each page over time
- The file will thus occupy around 15,000 pages (10,000 * 3/2).
- So, on the last problem, number of pages fetched will be approximately:
  - 3,750 pages (25% of 15,000 pages).

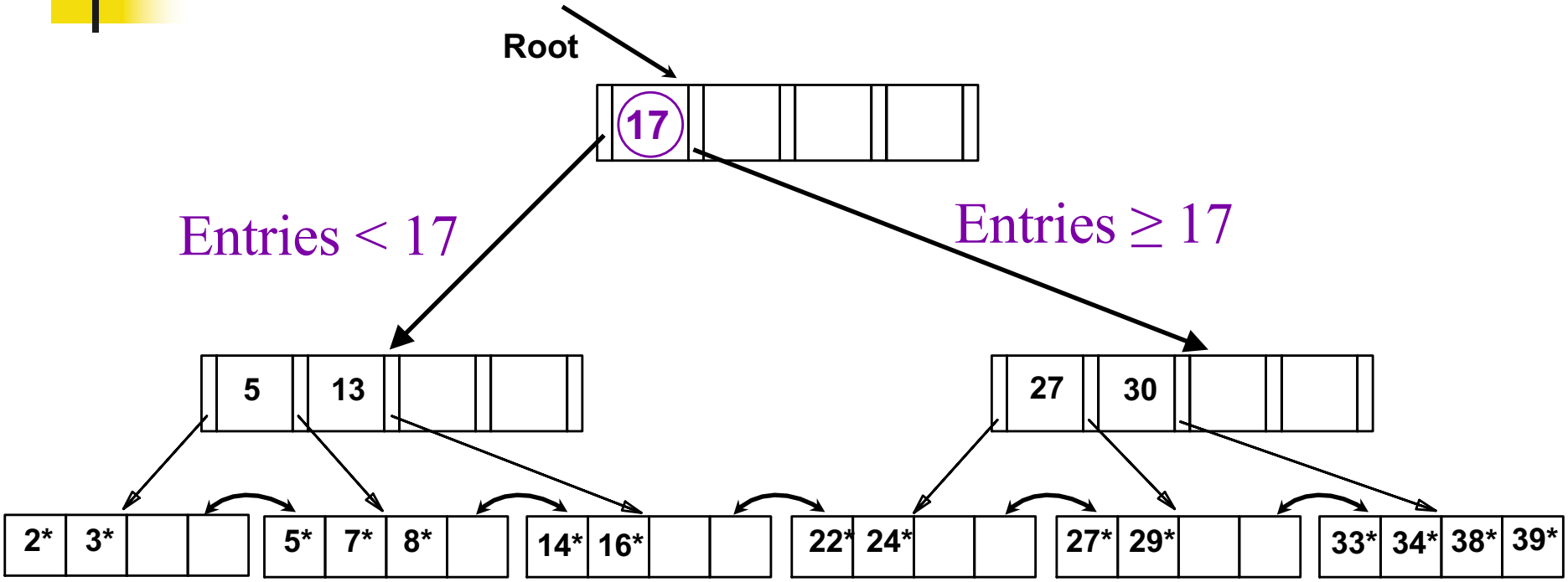
# B+ Tree Indexes




- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



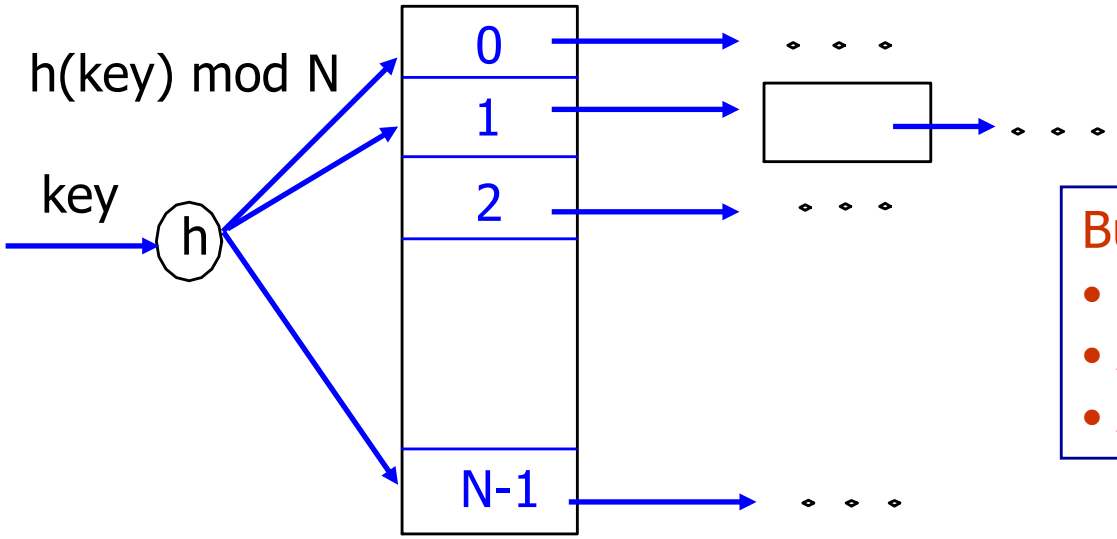
# Example B+ Tree




- Find 29*? 28*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

# Hash-Based Indexes

- Good for equality selections.
  - Index is a collection of *buckets.* Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r.*

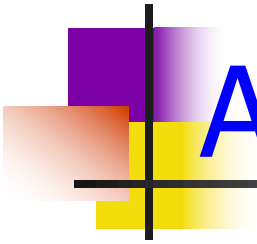



**Primary bucket pages** **Overflow pages**

Buckets contain:
- If Alternative (1) is used → data records
- Alternative 2 →<key, rid>
- Alternative 3 → <key, rid-list> pairs

# Comparing File Organizations / Indexes - Example

- Employees (Name, Age, Salary)
- 10 records/page and 10,000,000 records.
- Assume that people are uniformly distributed between ages 1 to 100.
- Operations.
  - Fetch the names of employees:
    - Scan: all employees
    - Equality Search: age = 21
    - Range Selection: age >= 18 AND age < 65
  - Insert a record
  - Delete a record



# Analysis of I/O Cost

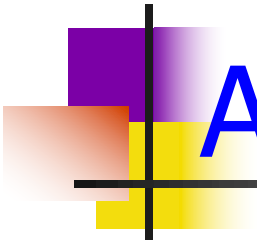
- For simplicity, ignore CPU cost
- Important Factors for I/O Cost:
  - How many pages (approx) read/written?
  - Are I/Os sequential or non-sequential?
- With your neighbor, do this for:
  - HEAP and Clustered index on Age
  - All 5 ops: SCAN, EQUALITY, RANGE, INS, DEL

# Heap File

- **Scan:** Read all pages in file
  - sequential
- **Equality Search (age = 21):**
  - Read all pages in files sequentially
  - *worst case*
- **Range Selection on age:**
  - Read all pages in file sequentially
- **Insert:** Read and write last page or write into a new page (2 page I/Os)
- **Delete:** Searching cost (expensive) + rewrite the page with the record

# Clustered index (on Age)

- **Scan:** Real all pages on disk. FileScan. Same as unclustered.
- **Equality Search (age = 18):**
  - Traverse height of tree, read records page
  - non-sequential
- **Range Selection:**
  - Traverse height of tree (non-sequential)
  - Scan leaf records (sequential)
- **Insert:** Traverse height of tree (non-sequential) + write
- **Delete:** Traverse height of tree (non-sequential) + write



# Analysis of I/O Cost

- With your neighbor, do this for:
  - Heap-with-Unclustered-Tree-Index (on Age)
  - Heap-with-Unclustered-Hash-Index (on Age)
  - All 5 ops: SCAN, EQUALITY, RANGE, INS, DEL

# Heap File w/ Unclustered Tree Index (on Age)

- **Scan:** Same as heap file
- **Equality Search:**
  - Traverse height of tree (H pages)
  - Use k* to find M matching records (non-sequential)
  - Approx. cost: H + M pages (non-sequential)
- **Range Selection:**
  - Traverse height of tree (non-sequential). Get k* values.
  - Scan leaf records for each RID (**non-sequential**)
  - Worst-case: almost one page/record! Very expensive
- **Insert:** Update index + update heap file
- **Delete:** Update index + update heap file

# Heap File w/ Unclustered Hash Index (on Age)

- **Scan:** Same as heap file
- **Equality Search:** Very few blocks
- **Range Selection:** Hash index no use!
  - Scan heap file
- **Insert:** Update index + update heap file
- **Delete:** Update index + update heap file