Storage and Indexes

Michael Yu

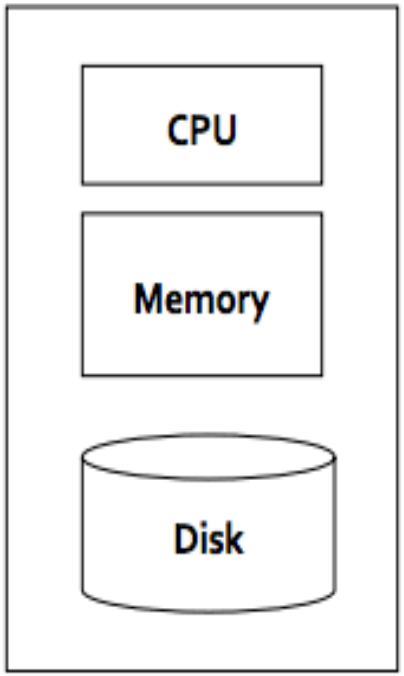
2024-2025 T1

Part One

DB Storage on disks

Primary Storage

- **Main memory**:
  - Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
  - Generally too small (or too expensive) to store the entire database
- **Volatile**
  - contents of main memory are usually lost if a power failure or system crash occurs.



Store our DB in Main Memory?

- *No… Costs too high*
  - Ram even more expensive than disk
- *No… Main memory is volatile*
  - We want data to be saved between runs

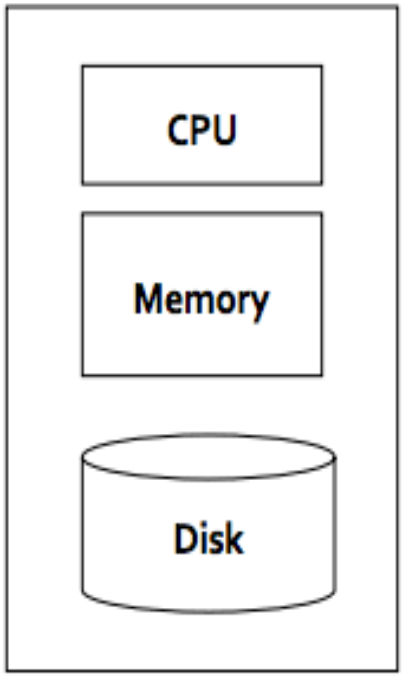
Memory Hierarchy

- Storage hierarchy
  - Main memory for currently used data
  - (secondary storage) Disk to store all data and database
  - If necessary, tapes for archiving older versions of the data

- ***Typical Memory Hierarchy***
  - ***Primary Storage***: main memory (RAM)
    - fast access, expensive
  - ***Secondary storage:*** hard disk
    - slower access, less expensive
  - ***Tertiary storage***: tapes, cd, etc.
    - slowest access, cheapest

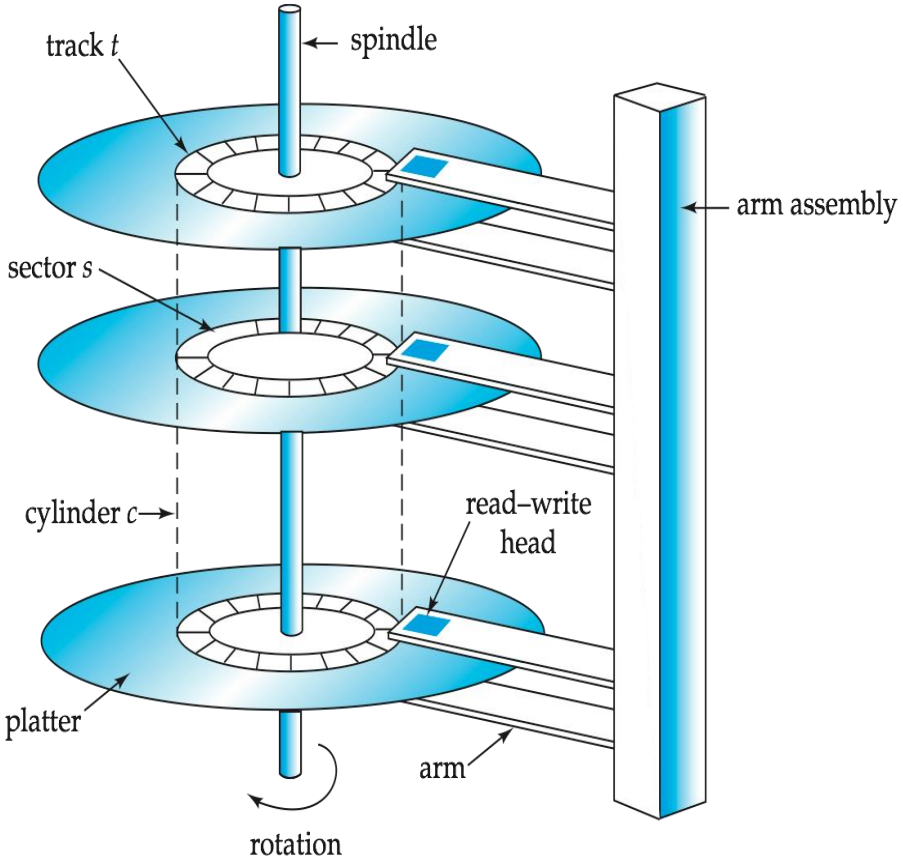
Secondary Storage

- **Disk**
  - Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- **Direct-access** – possible to read data on disk in any order.
- Survives power failures and system crashes
  - Recall: disk failure can destroy data, but is rare



Disk Mechanism

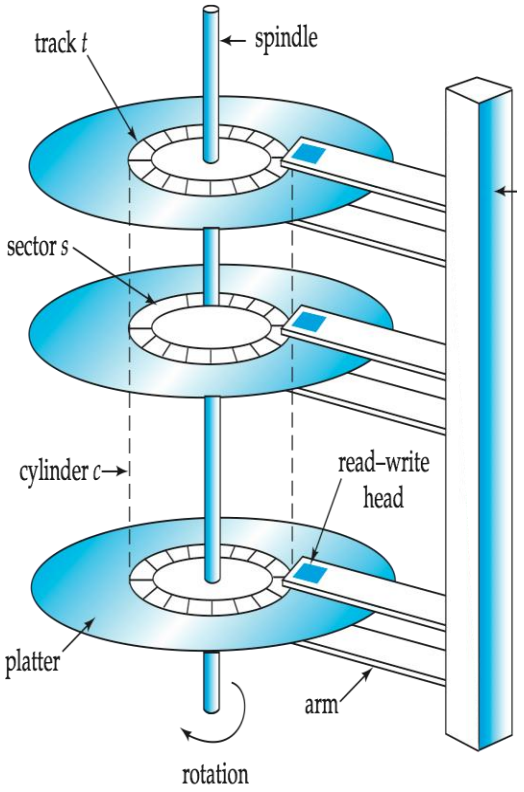
- Characteristics of disks:
  - collection of platters
  - each platter = set of tracks
  - each track = sequence of sectors (blocks)



NOTE: simplified the structure of actual disk drives

Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins.
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst-case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 **rpm** (revolutions per minute))
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate.



Summary

- Information are permanently stored on ("hard") disks.
  - A disk is a sequence of bytes, each has a ***disk address***.
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost, relative to in-memory operations.

Databases on Disks (0)

- Databases are also permanently stored on ("hard") disks.
- **From a relation to disk sectors**
  1. A relation is stored as a "**file**" on **disk**.
     1. A file is a sequence of blocks, where a block is a fixed-length storage unit.
- **From a relation to fields** (attribute value or column value)
  - A relation is a sequence of records
  - A record is a sequence of data fields
    - There are fixed-length data fields (say using char(n))
    - There are variable-length data fields (say using varchar(n))
  - If all data fields are fixed-length, we use fixed-length records.
  - If at least one data field is variable-length, we use variable-length records.

Databases on Disks (1)

- To simplify the setting, a database is partitioned into fixed-length storage units called blocks.
  1. Blocks are units of storage allocation
  2. Blocks are the units of data transfer to and from disk
     - 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)
  3. Each block contains several records
     - We shall assume that no data item spans two or more blocks.

Closer Look at Blocks (Pages)

Format:

- A block is a collection of *slots*.
- Each slot contains a record.
- A record is identified by rid = *<page id, slot number>*.

Record Format

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

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

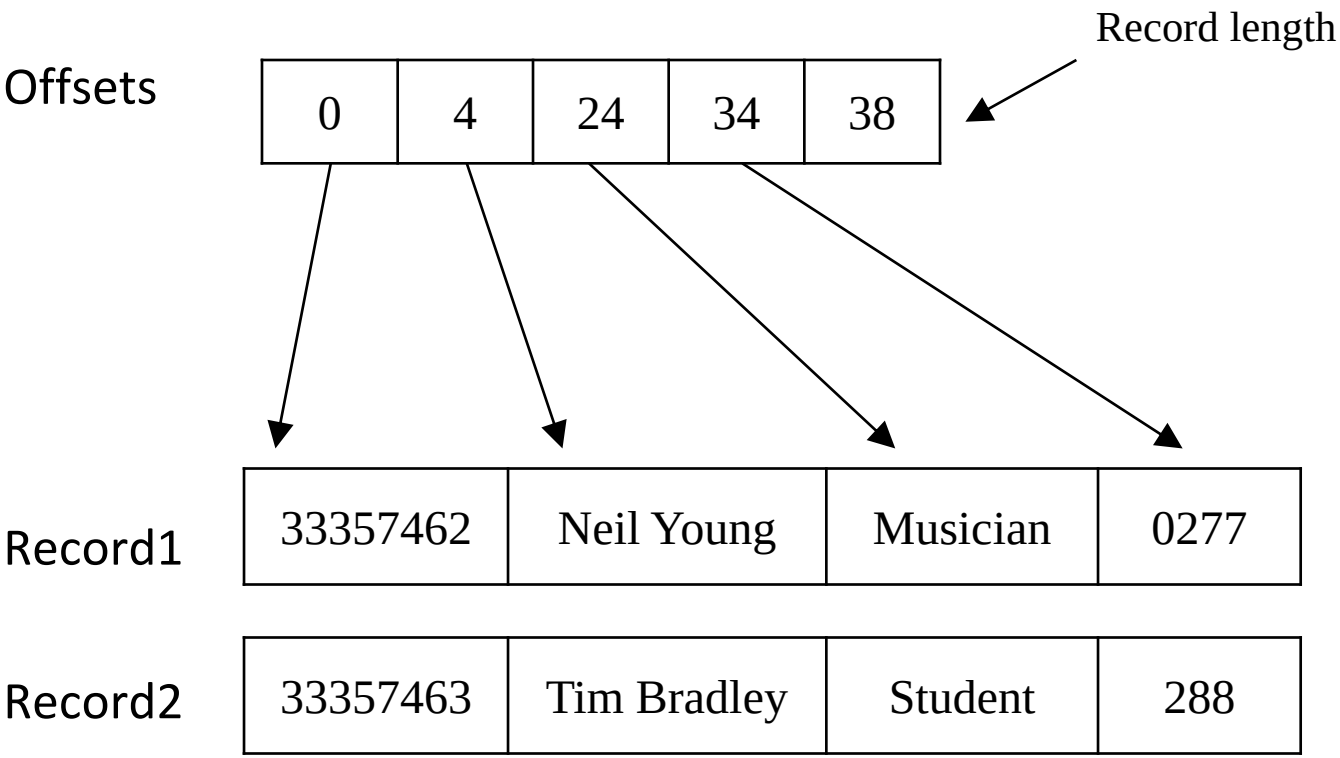
33357462	Neil Young	Musician	0277

4 bytes 10 bytes 8 bytes 4 bytes

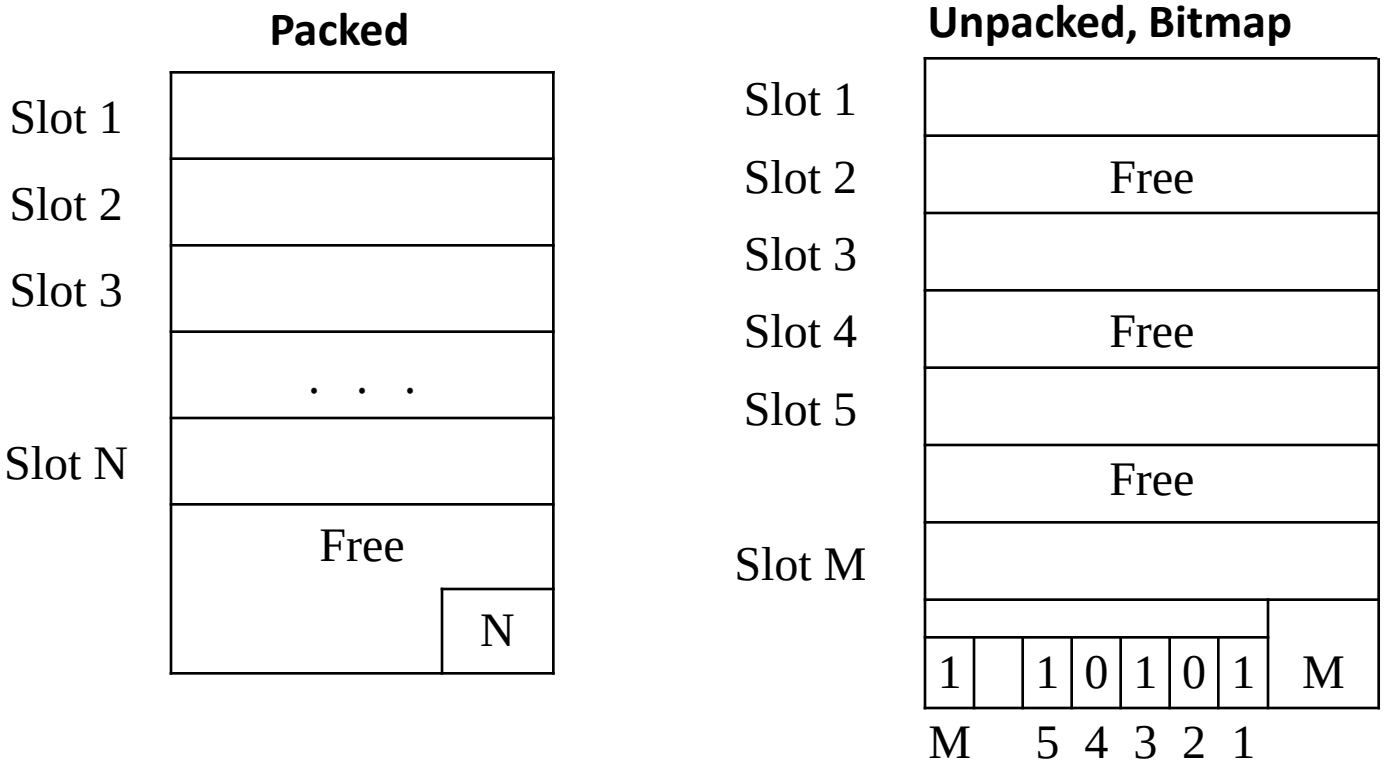
- complicates intra-block space management
- does not waste (as much) space.

Fixed-Length Records (0)

Encoding scheme for fixed-length records: length + offsets stored in header



Fixed-Length Records (1)



- Insertion: occupy first free slot; packed more efficient.
- Deletion: (a) need to compact, (b) mark with 0; unpacked more efficient.

Fixed-Length Records (2)

- Simple approach:
  - Store record i starting from byte $n \times (i - 1)$, where n is the size of each record.

- Consider three ways in deleting record i:
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - do not move records, but link all free records on a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records (0)

Encoding schemes where attributes are stored **in order**.

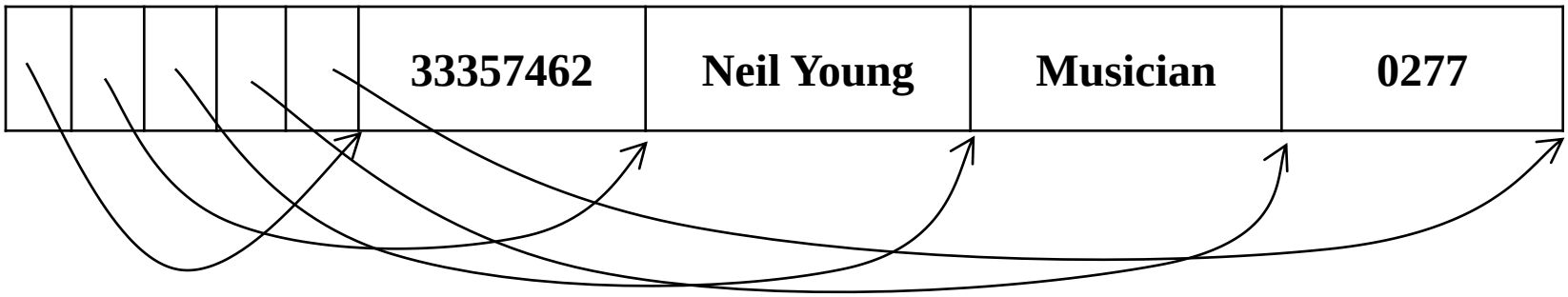
- Option1: Prefix each field by length

4	xxxx	10	Neil Young	8	Musician	4	xxxx

- Option 2: Terminate fields by delimiter

33357462/Neil Young/Musician/0277/

- Option 3: Array of offsets

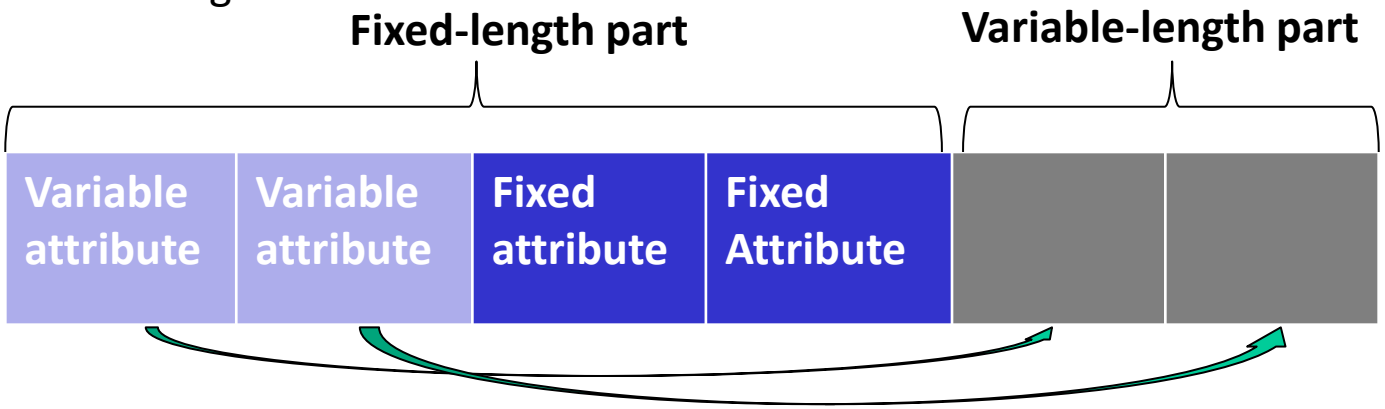


Variable-Length Records (1)

One more encoding scheme: attributes not stored in order.

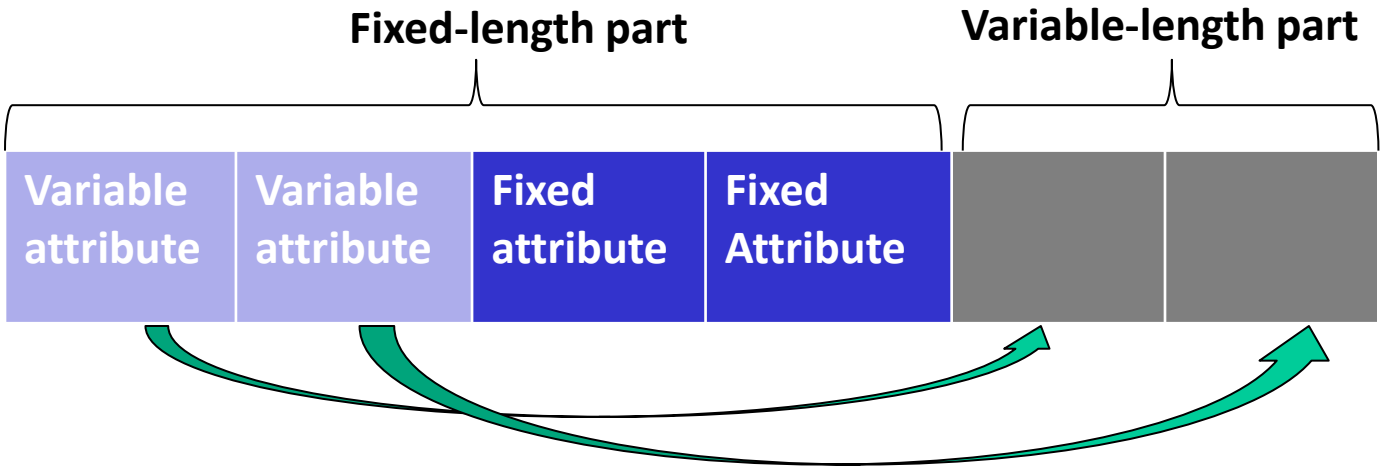
- Variable length attributes are represented by fixed size (offset, length) in the fixed-length part, and keep attribute values in the variable-length part.
- Fixed length attributes store attribute values in the fixed-length part.
- Fixed-length part followed by variable-length part.
  - (b) The fixed-length part is to tell where we can find the data if it is a variable-length data field.
  - (c) The variable-length part is to store the data.

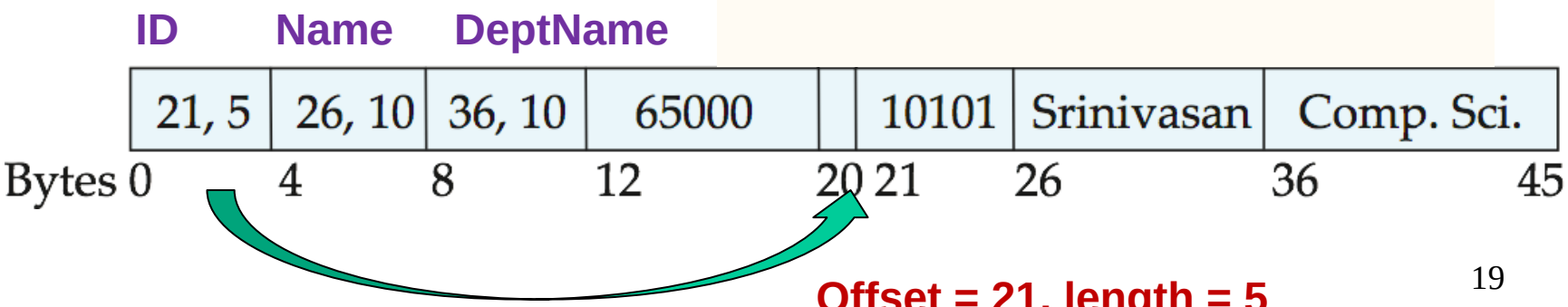
- Example: Suppose there is a relation with 4 attributes: 2 fixed-length and 2 variable-length.



Variable-Length Records (2)

- Example: a tuple of (**ID, Name, DeptName**, Salary) where the **first three** are variable length.



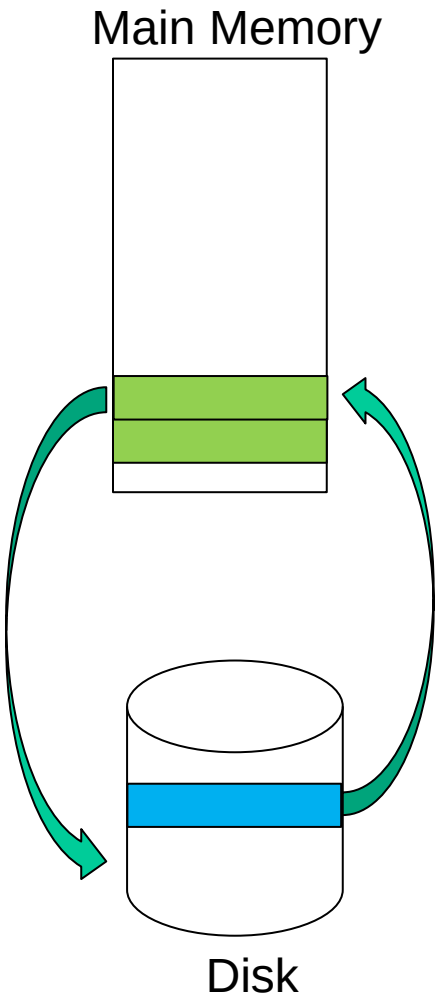


Part Two

Transfer of Information to and from Disks

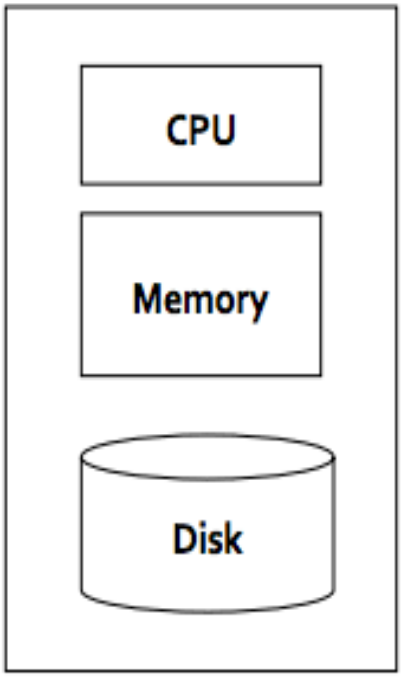
Storage Access

- Transactions input information from the disk into main memory and then output the information back onto the disk.
- The input and output operations are done in block units.
  - If a single record in a block is needed, the entire block is transferred.
- Database system seeks to minimize the number of block transfers between the disk and memory!
  - We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.



Context (CPU cost vs I/O cost)

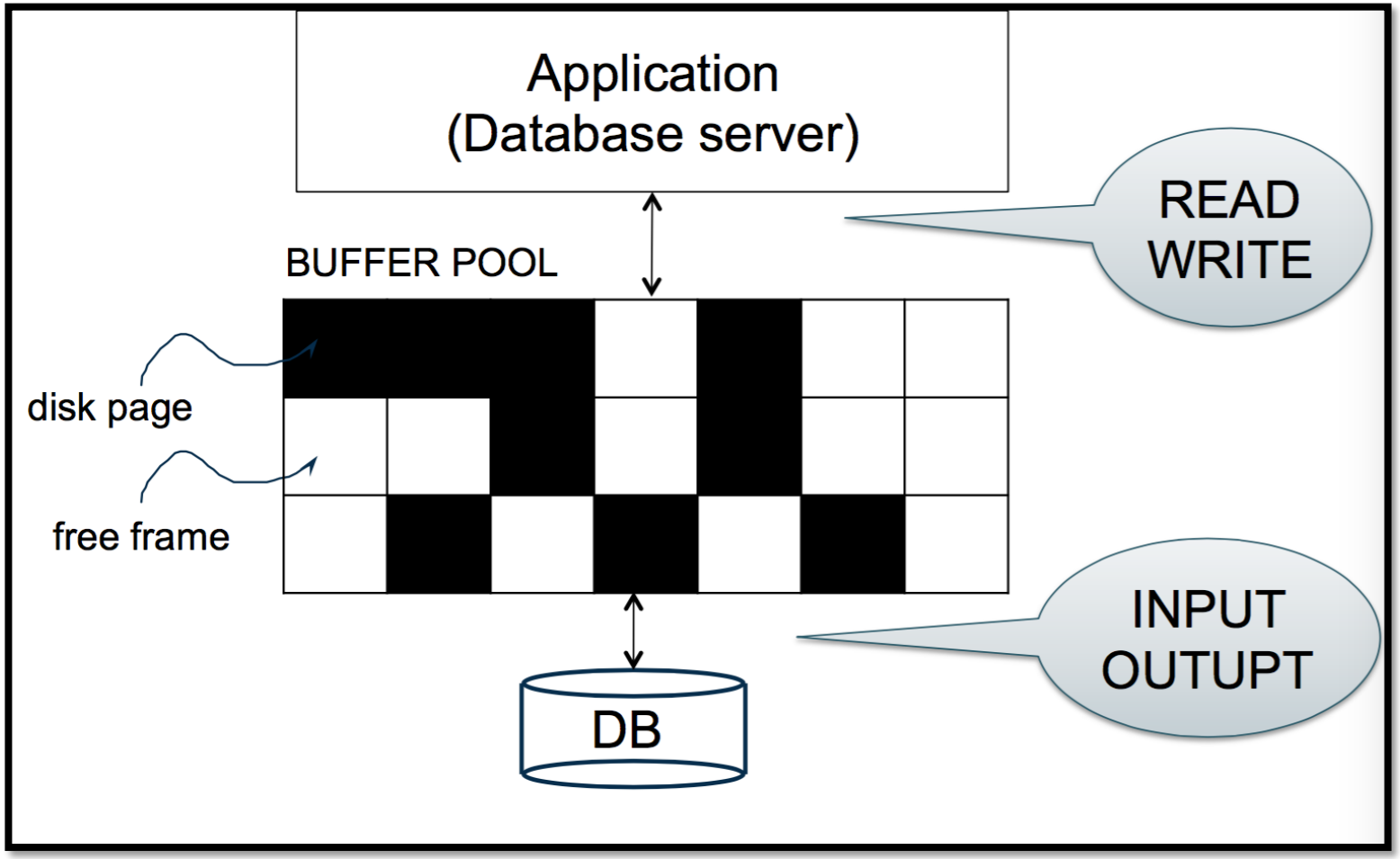
- **Data must be moved from disk to main memory for access, and written back for storage**
  - **Much slower access than main memory**
  - Two main costs
    - CPU cost is to process data in main memory
    - I/O cost is to read/write data from/into disk
  - The dominating cost is I/O cost. For query processing in DBMS, CPU cost can be ignored.
  - The key issue is to reduce I/O cost.
    - It is to reduce the number of I/O accesses.

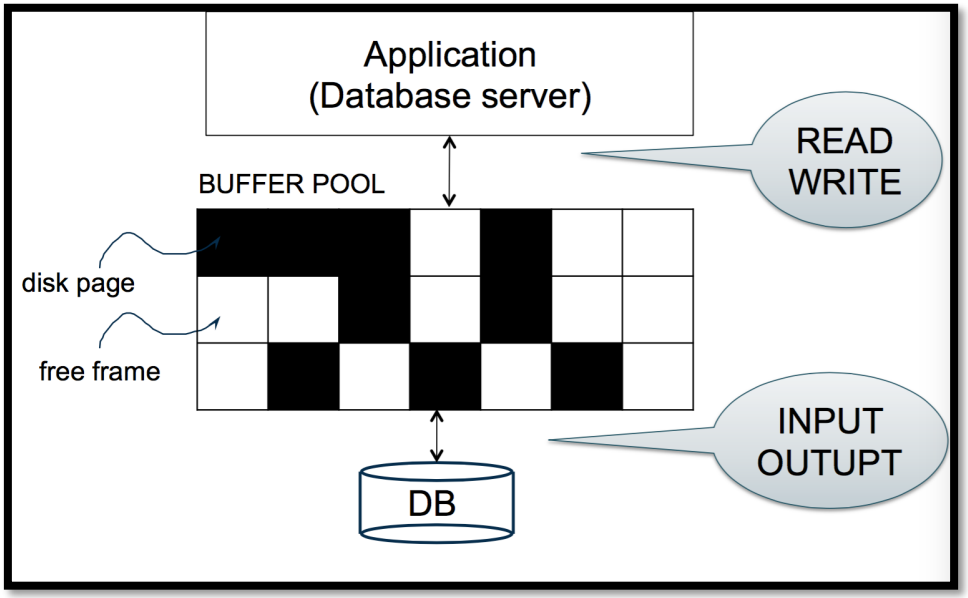


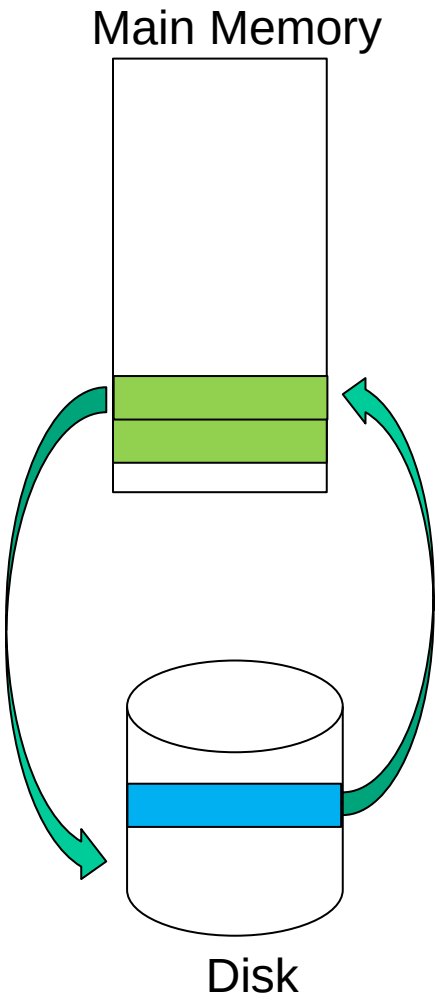
Disk Space Management

- *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
    - Using OS File System to Manage Disk Space?
      - Extend OS facilities, but not rely on the OS file system (portability and scalability concerns)

Buffer Management in a DBMS







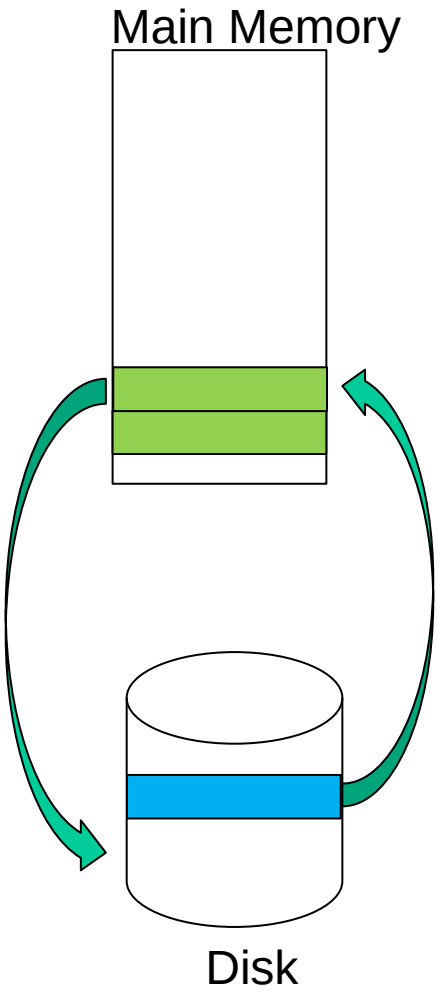
- **Terminology**
  - **Disk Buffer** – portion of main memory available to store copies of disk blocks.
  - **Buffer manager** – subsystem responsible for allocating buffer space in main memory.
  - The blocks residing on the disk are referred to as **physical blocks**; the (copy of) blocks residing temporarily in main memory are referred to as **buffer blocks**.

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.
  - One page = 4096 Bytes = One block

Buffer Manager

1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
2. If the block is not in the buffer, the buffer manager
    1. Allocates space in the buffer for the block
        1. **Replacing** (throwing out) some other block, if required, to make space for the new block.
        2. Replaced block written back to disk only if it was **modified** since the most recent time that it was written to/fetched from the disk.
    2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



When a Page is Requested ...

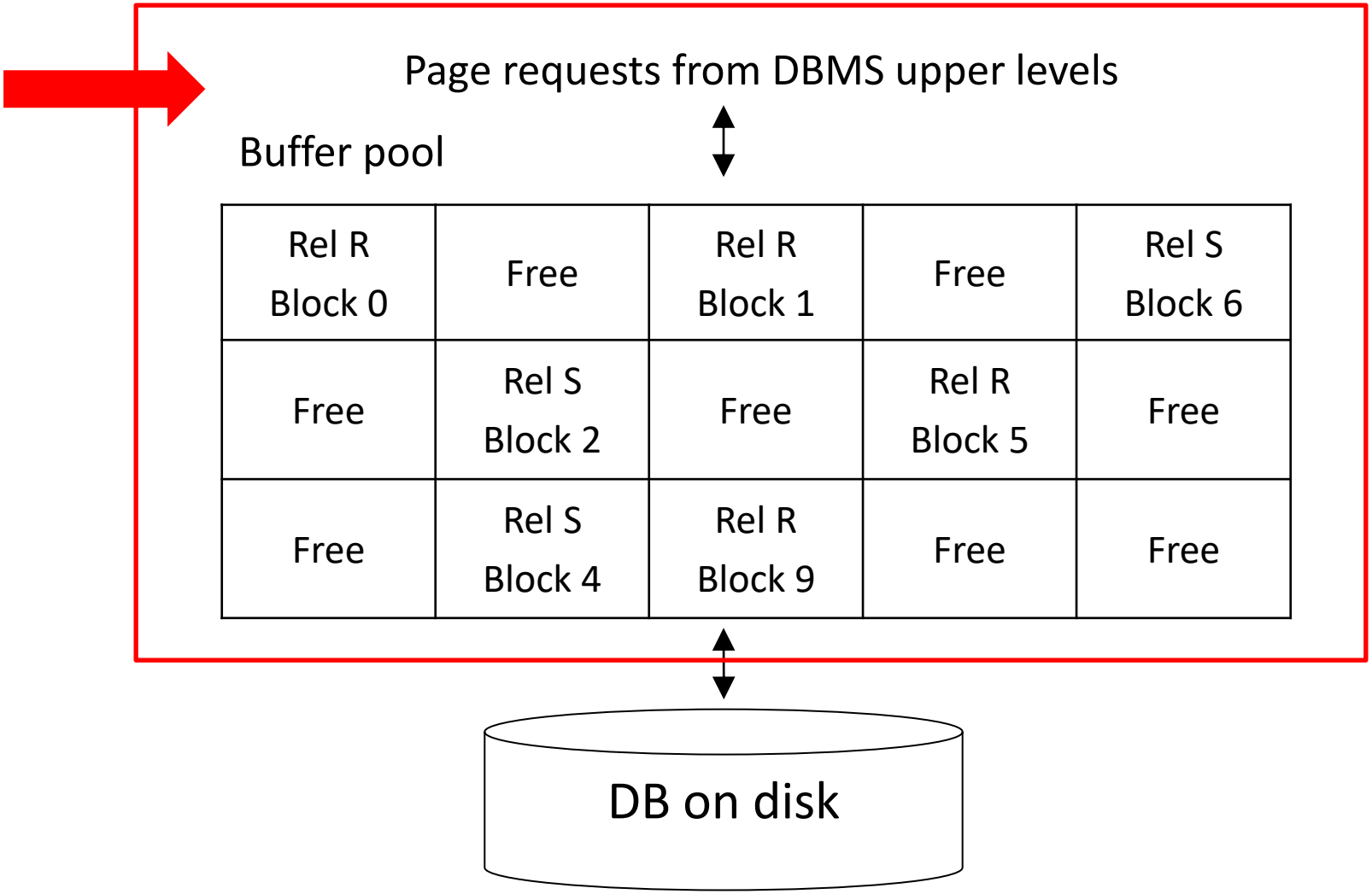
- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address

Note: If requests can be predicted pages can be pre-fetched several pages at a time (e.g., sequential scans)

More on Buffer Management

- Requestor of page must unpin it, and indicate whether a page has been modified:
  - *dirty* bit is used for this.
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.

Buffer Pool (Part 1)



Applications and Buffer (0)

Read

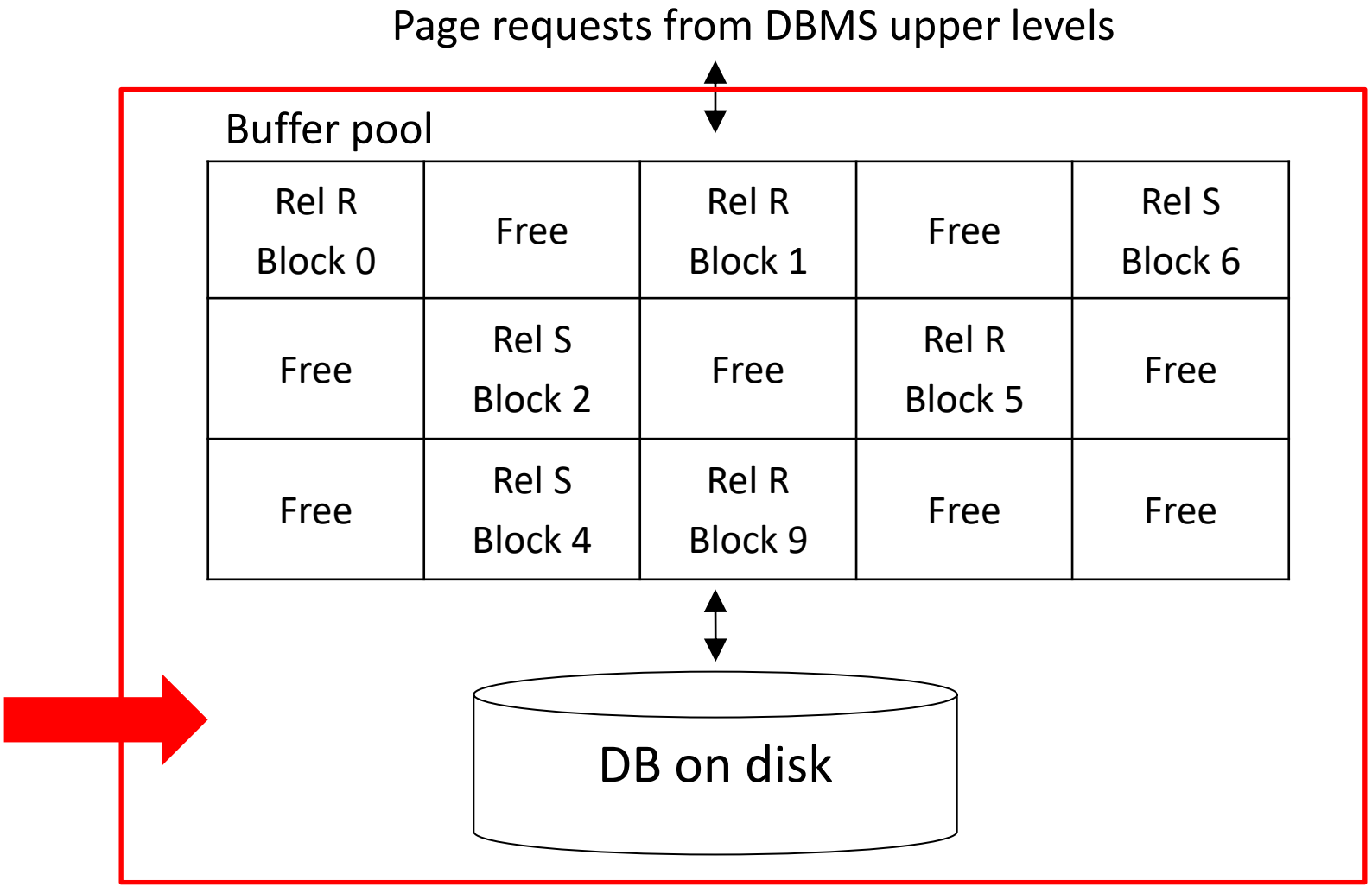
1. Compute the data block that contains the item to be read
2. Either
   A. find a buffer containing the block, or
   B. read from disk into a buffer
3. Copy the value from the buffer

Applications and Buffer (1)

Write

1. Compute the disk block containing the item to be written
2. Either
   A. find a buffer containing the block, or
   B. read from disk into a buffer
3. Copy the new value into the buffer
4. At some point (maybe later), write the buffer back to disk

Buffer Pool (Part 2)



Buffer Pool Operations (0)

- The ***request_block*** operation
  - [CASE ONE] If block ***is*** already in buffer pool:
    - no need to read it again
    - use the copy there (unless write-locked)
  - [CASE TWO] 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 Operations (1)

- For each frame, the manager keeps track of:
  1. whether currently in use
  2. whether modified since loading (***dirty bit***)
  3. how many transactions are currently using it (***pin count***)
  4. (some but not all) time-stamp of most recent access

Buffer Pool Operations (2)

- The ***release_block*** Operation
  - Decrement pin count for specified page
    No real effect until replacement required

- The ***write_block*** Operation
  - Updates contents of page in pool
  - Set dirty bit on
  - (Doesn't actually write to disk, until been replaced, or forced to commit)

Buffer Pool Operations (3)

- Eventually, a buffer block is written out back onto the disk
  1. buffer manager needs the memory space for other purposes
  2. database system wishes to reflect the change to B on the disk

- The ***force_block*** operation
  - "commits" by writing to disk.

Buffer-Replacement Policies

- There are several ways Buffer can be replaced
  - Referred to as Buffer-replacement policies
  - Generally, uses past usage patterns of block as an indicator of future references

- Least Recently Used (LRU) – used by most sys.
  - release the frame that has not been used for the longest period
  - intuitively appealing idea but can perform badly
  - Aka: the one I used just now may be needed very soon

Buffer-Replacement Policies

- An example: department (4 records, 2 blocks) and instructor (8 records, 4 blocks)
  - Our buffer has 4 blocks.
  - select * from department, instructor
    where department.dept_name = instructor.dept_name

department

D1 D2	D3 D4

instructor

D1 D4	D2 D3	D1 D2	D1 D4

Buffer-Replacement Policies

SQL:

*select * from department, instructor*
where department.dept_name = instructor.dept_name

Implementation:

for each tuple *t* of ***department*** do:
    for each tuple *s* of ***instructor*** do:
        if the t and s have the same department name:
            output all the attributes

Department

D1 D2	D3 D4
b1	**b2**

Instructor

D1 D4	D2 D3	D1 D2	D1 D4
b1	**b2**	**b3**	**b4**

Buffer

b1			
b1	**b1**	**b2**	**b3**

Now we need to access b4. But we need to replace it. Which do we replace with LRU? b1!

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

None is guaranteed to be better than the others. Quite dependent on the application.

Buffer-Replacement

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q4	**P2** Q5	**P3** Q3

Q6: read P4:

- **LRU**: Replace P3
- MRU: Replace P2
- FIFO: Replace P1
- Random: randomly choose

one buffer to replace

Example

Data pages: P1, P2, …, P11

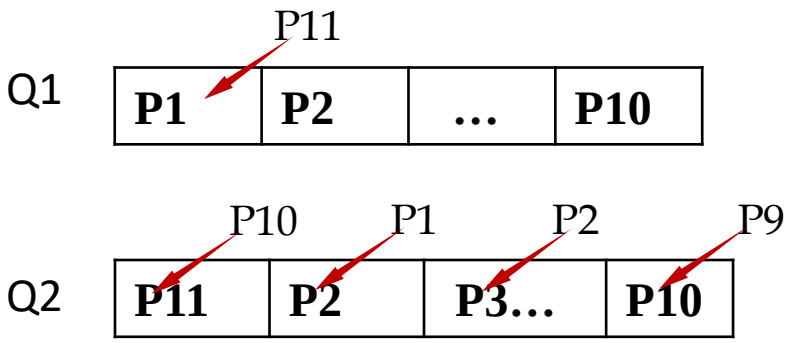
Queries:

Q1: Read P1, P2,…, P11

Q2, Read P1, P2,…, P11

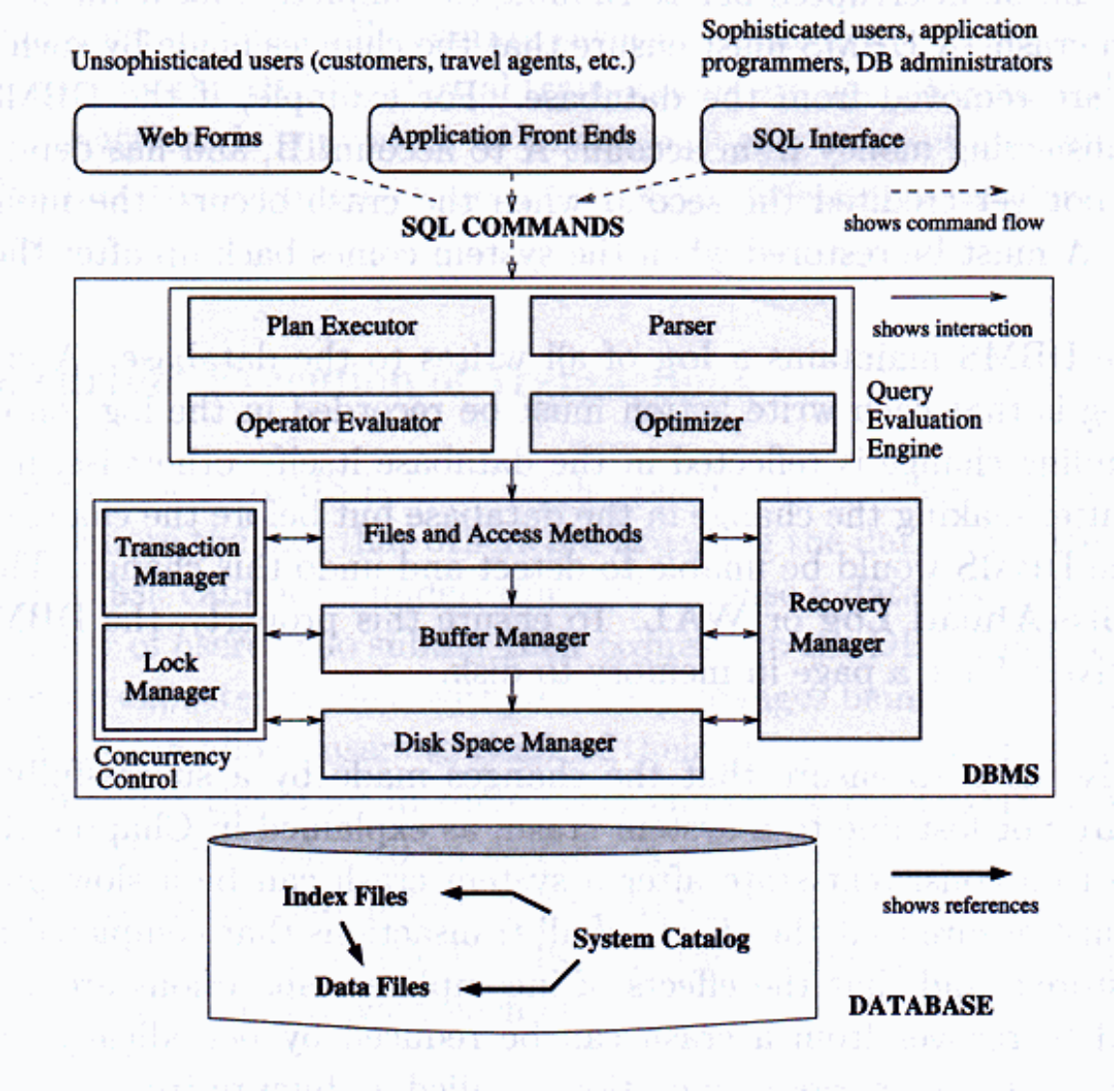
Q3: Read P1, P2,…,P11

Buffer: 10 pages LRU (FIFO produces the same results in this case):



Behaviour : We need to get in/out every page

MRU: performs the best in this case.



Architecture of a DBMS

Concept of Indexing

Books have been arranged via categories, subjects

Same categories

stored in the same area

Book is data

Catalogue is the index

What is Index?

Phone number has been arranged in Alphabets, groups (work, friends, …)

Phone number is data

Alphabets and the groups page are the index

What is Index?

- Auxiliary data
- Properly organised (data structure)
- To facilitate data search

Indexes

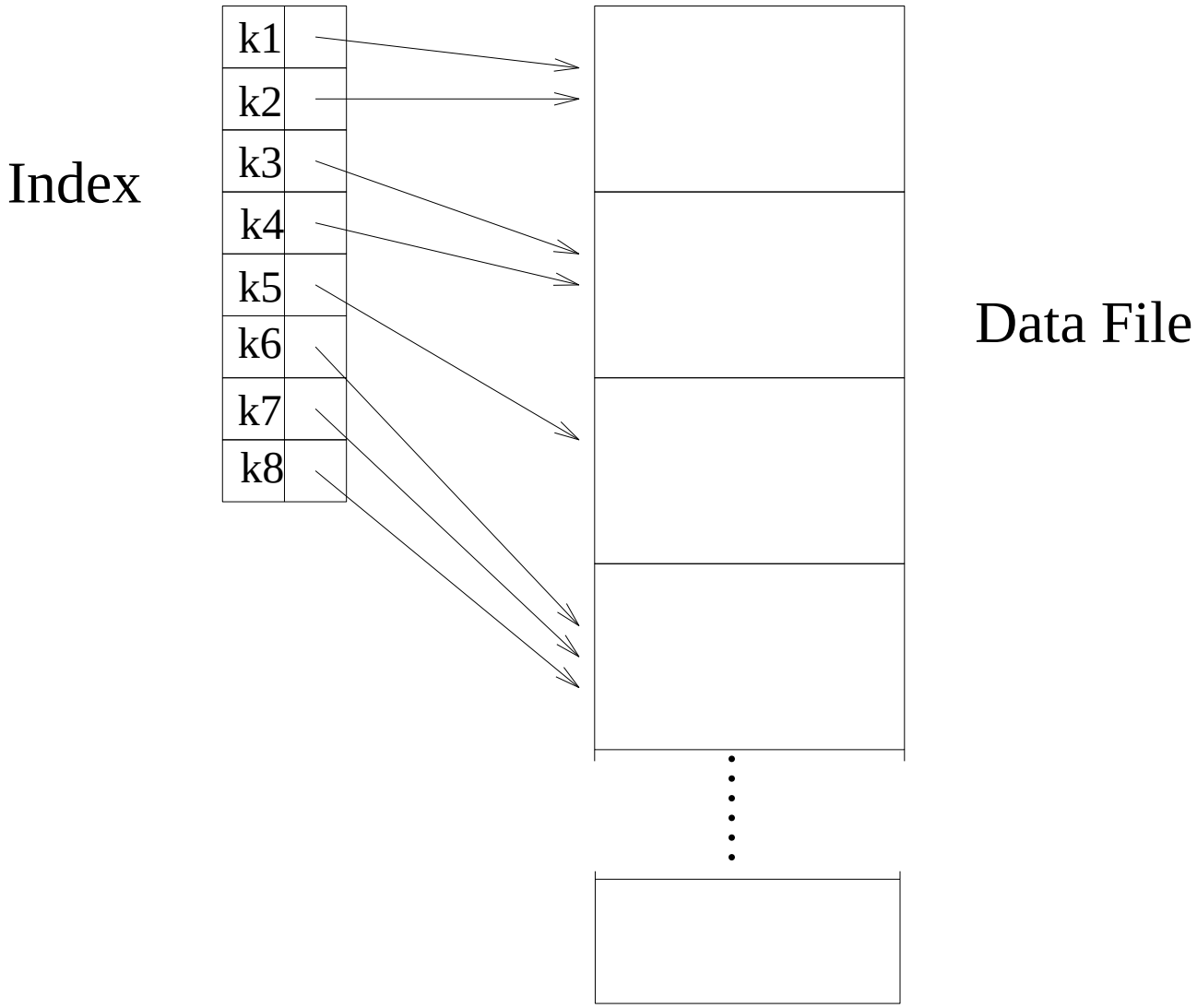
Word indexes in a book:

INDEXES

aardvark 25,36	lion 18
bat 12	llama 17,21,22
cat 1,5,12	sloth 18
dog 3	tiger 18
elephant . . 17	wombat. . . 27
emu 28	zebra 19

- A table of key values, where each entry gives places where key is used.
- Aim: efficient access to records via key.

Indexing Structure



Indexing Structure

- Index is collection of data entries k∗.
  - Each data entry k∗ contains enough information to retrieve (one or more) records with search key value k.
- Indexing:
  - Index must support efficient retrieval of all data entries **k*** with a given key value **k**
  - Structure of data entry in more detail (i.e., how could the index store this information)

Search Key

- An *index* on a file speeds up selections on the *search key fields* for the index
  - Search keys are created to search for a single entry or a set of entries in an index
  - Search keys may only be constructed for the key columns in the index, and may contain one or more column values.
  - Any subset of the fields of a relation can be the search key for an index on the relation
    - E.g., just one, or multi-attribute search key

What could this Index store for k*

- Possible index structures
  1. <k, data record with search key value k> (not often used)
  2. <k, **one rid of one** data record with search key value k>
  3. <k, **list of rids** of data records with search key value k>
- Suppose, field 'name' is the **search key**
  1. <"Lin Wang", ("Lin Wang", 25, "12 First Street", 26094359)>
  2. <"Lin Wang", rec10101>
     - 10101 is the rid of a record that contains "Lin Wang"
  3. <"Lin Wang", rec10101, rec10111, rec11010>
     - 10101, 10111, 11010 are records which all contain "Lin Wang"

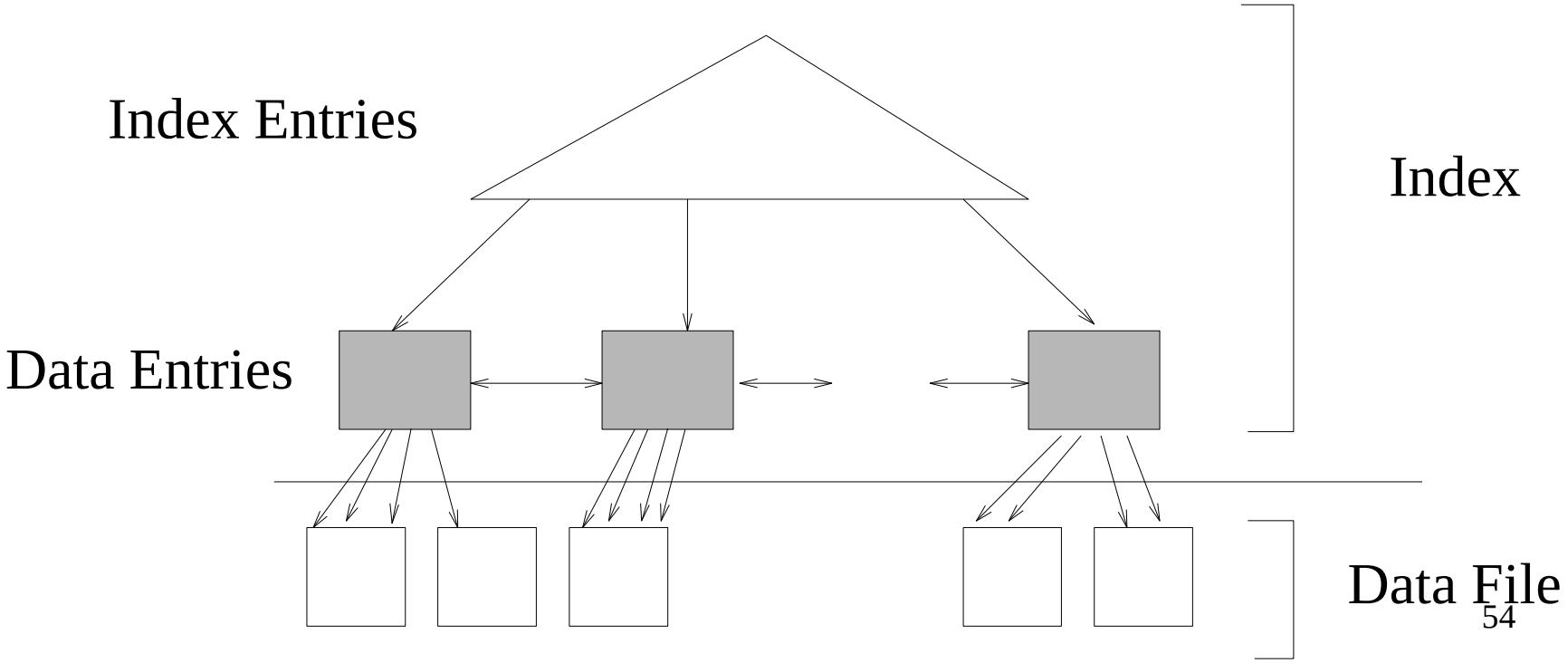
Index Classification

- *Primary* vs. *secondary*:
  - If search key contains primary key, then called **primary index**, otherwise it is called secondary.
  - *Unique* index: Search key contains a candidate key.
  - *Non-unique* indexes: can also be used to improve query performance by maintaining a sorted order of data values that are used frequently.

All keys are indexed. All indexes are not necessarily keys

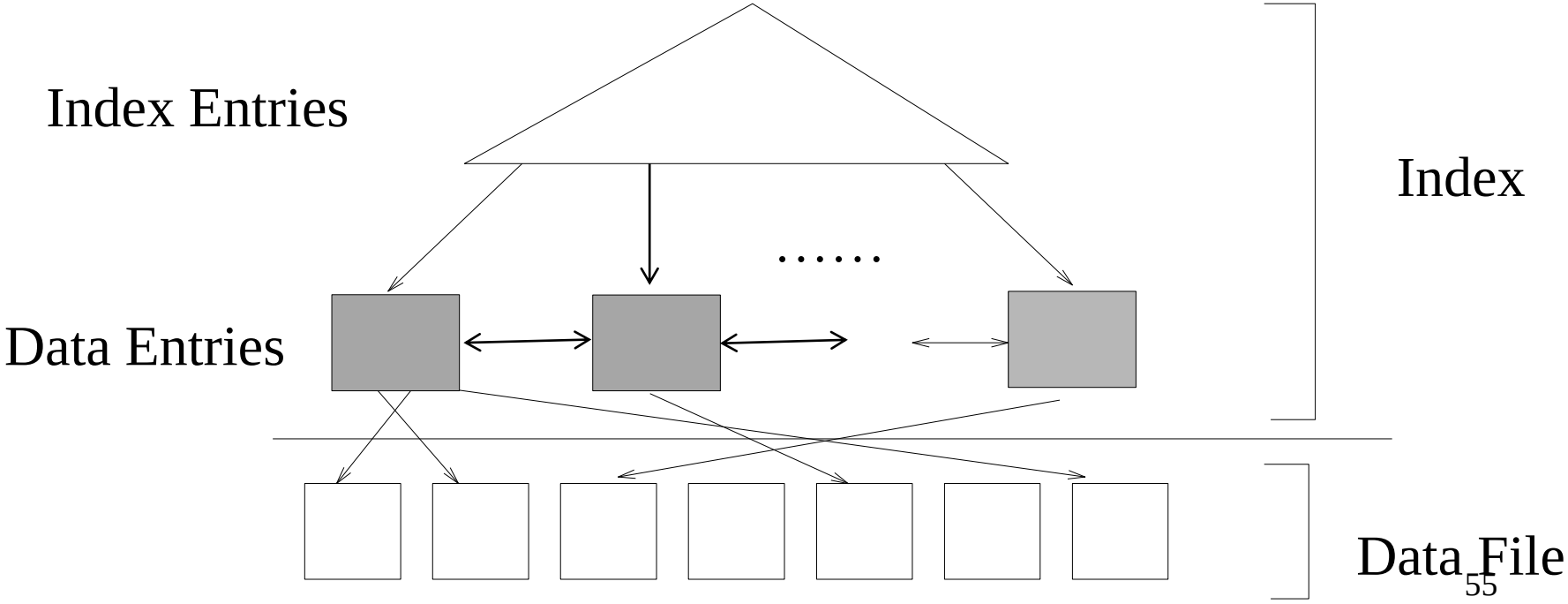
Clustered Index

- A *Clustered index* is a type of index in which table records are physically reordered to match the index.
- A clustered index can *tells you the order of the records on disk*



Non-clustered Index

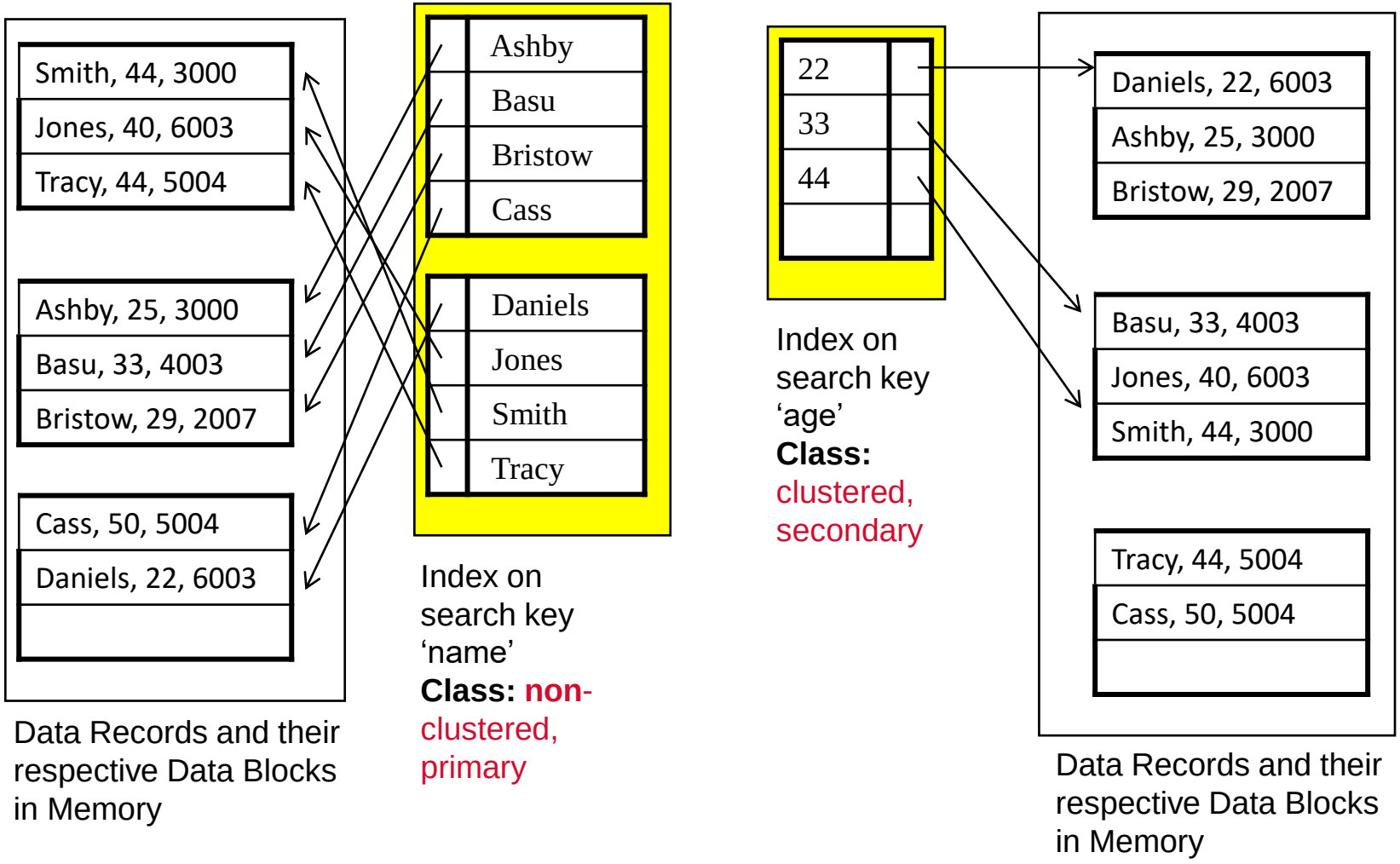
- Clustered indexes are relatively expensive to maintain
- A data file can be clustered on at most one search key.



Index Guidelines

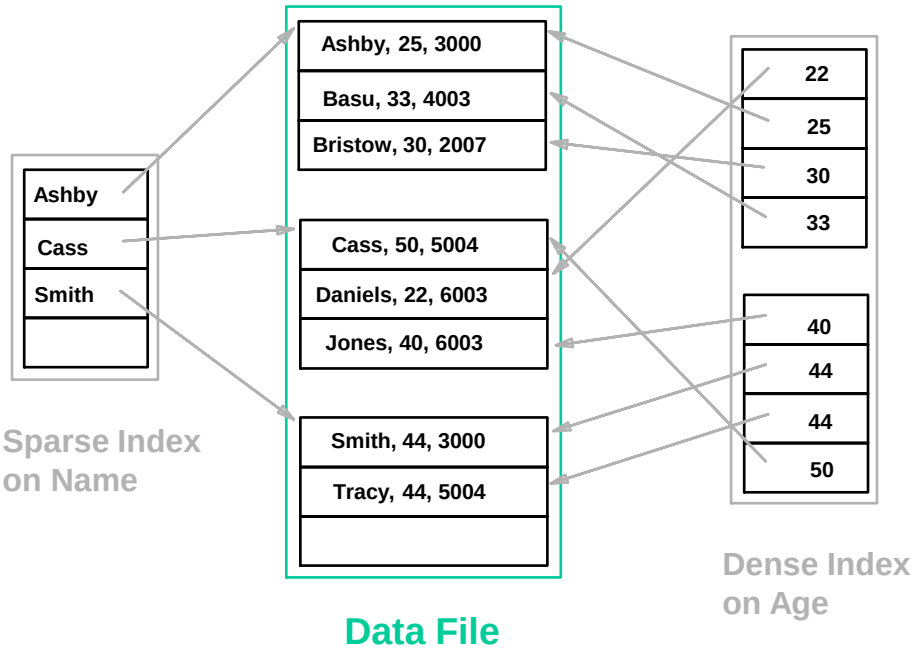
- *There can be only one clustered index per table*. However, you can create multiple non-clustered indexes on a single table.
  - With a clustered index the rows are stored physically on the disk in the same order as the index. Therefore, there can be only one clustered index.
- Indexes can make queries go faster, but…
  - Don't forget, the actual index requires disk space
  - Updating the index is significantly more time consuming. You must factor in the cost of **updates**

Examples of Index – Approach 1



Index Classification (Contd.)

- *Dense* vs. *Sparse indexes*:
  - If there is at least one data entry per search key value (in some data record), then the index is dense.
  - Sparse indexes are smaller
- Note:
  - Some useful optimizations are based on dense indexes
  - Every sparse index is clustered



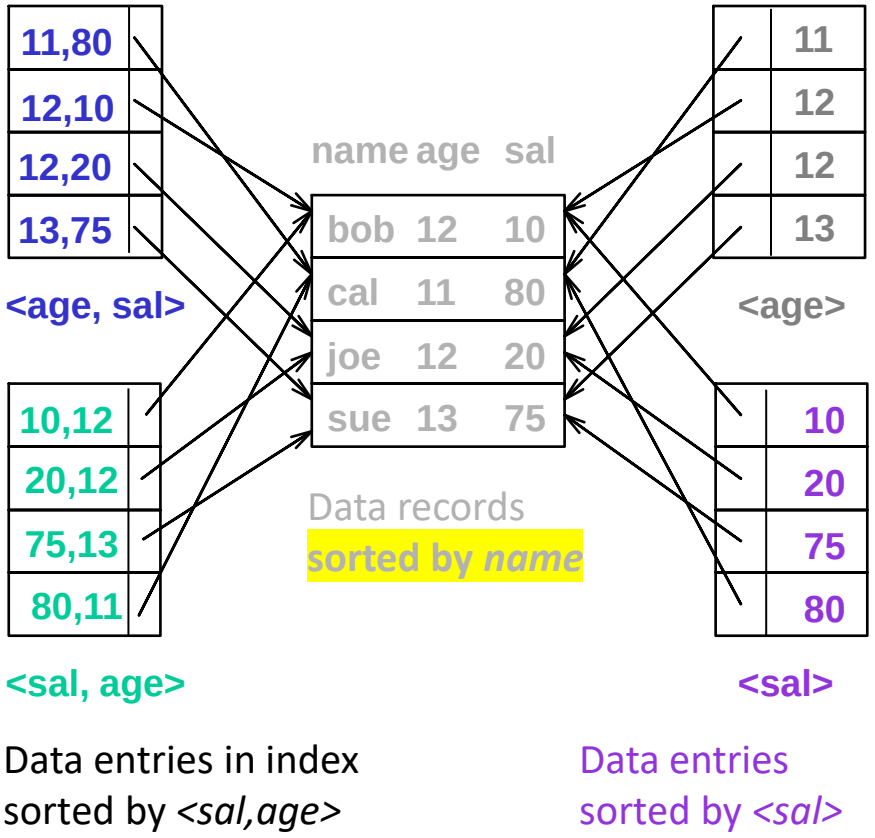
Index Classification (Contd.)

- *Composite Search Keys*: Search on a combination of fields.
  - **Supports Equality query**:
    - Field has a fixed target value
    - E.g. age=20 and sal =75 (wrt <sal,age> index)
  - **Range query**:
    - At least one field value without a fixed value
    - E.g. age = 20; or age=20 and sal > 10

Data records sorted by name

Data entries in index sorted by search key to support range queries

Examples of composite key indexes using lexicographic order
Each sorted by search keys



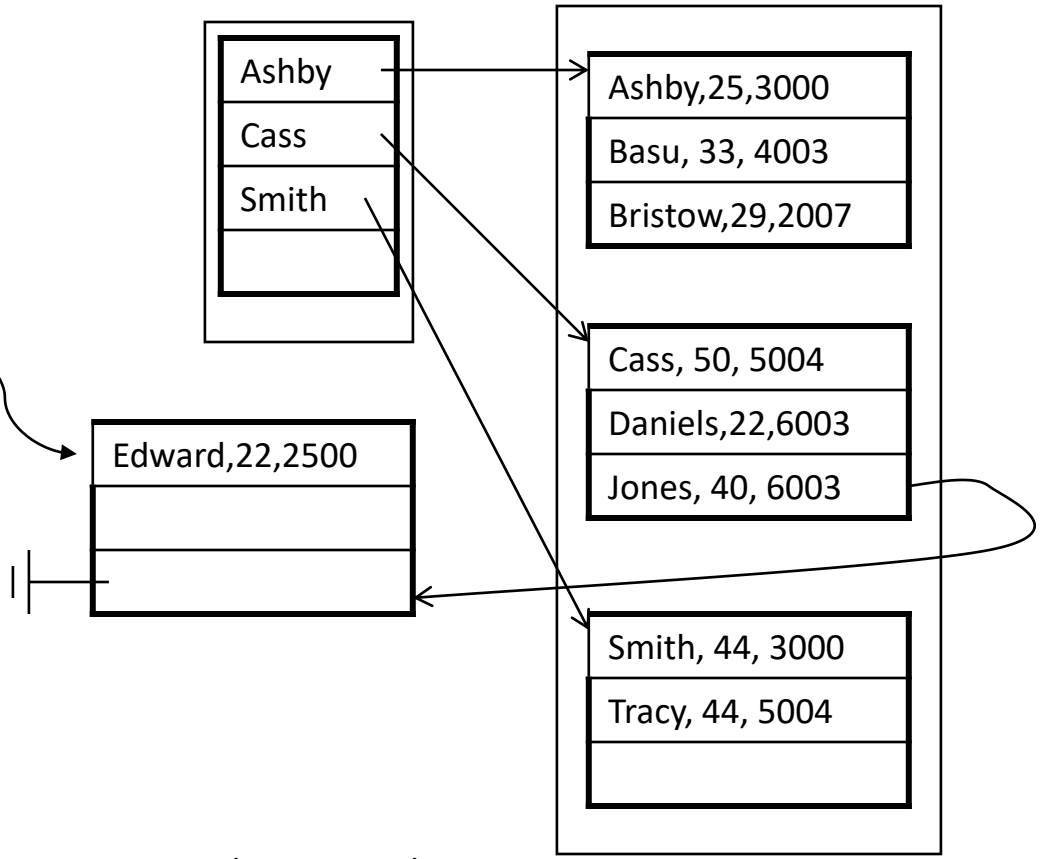
To Build Index (search key k)

- Non-clustered (must be dense):
  - Primary: each data entry k* points to the single record that contains k
  - Secondary: each data entry k* points to all the records that contain k
- Clustered (primary/ secondary):
  - Sort both data file and index file on the search key
  - Each data entry k* points to the ***first*** record that contains k
  - Note: overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)

Overflow Page

Insert record:
(Edward, 22, 2500)

We must use an
overflow page

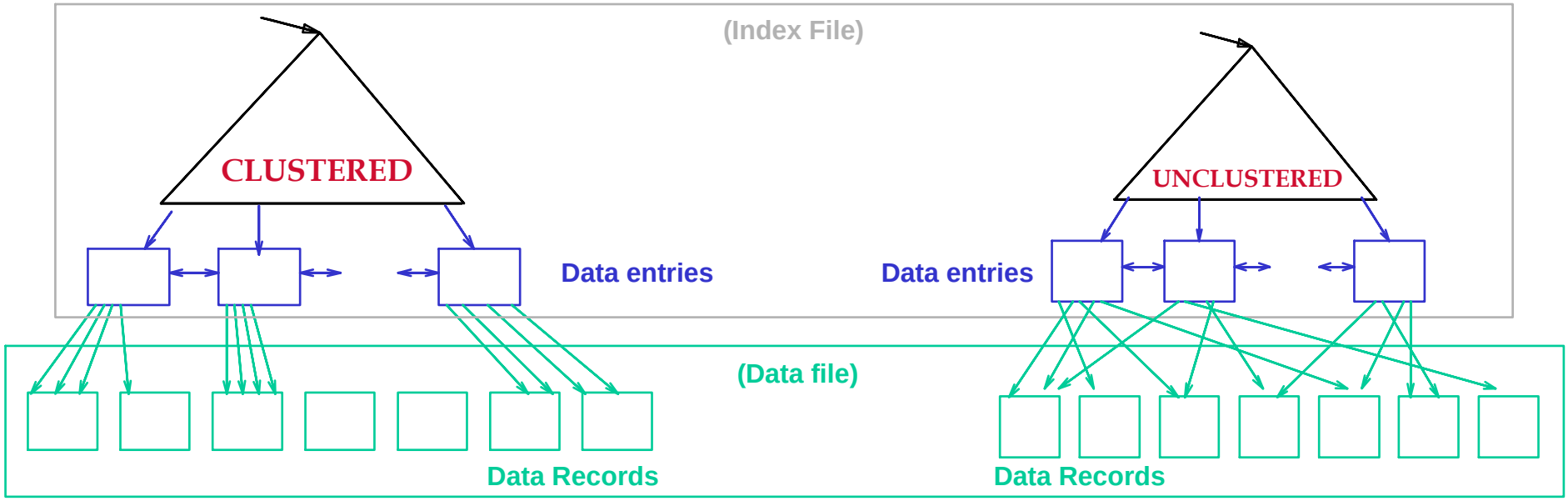


Index on search
Key 'name'
Class: clustered
primary

Sorted file on
name field

More on Index Structures

- Sometime the index file itself may be too large to be accessed efficiently
  - View the index file as a data file, and then build an additional index on this “data” file
- This idea can be applied repeatedly! Giving us a index structure like a tree!



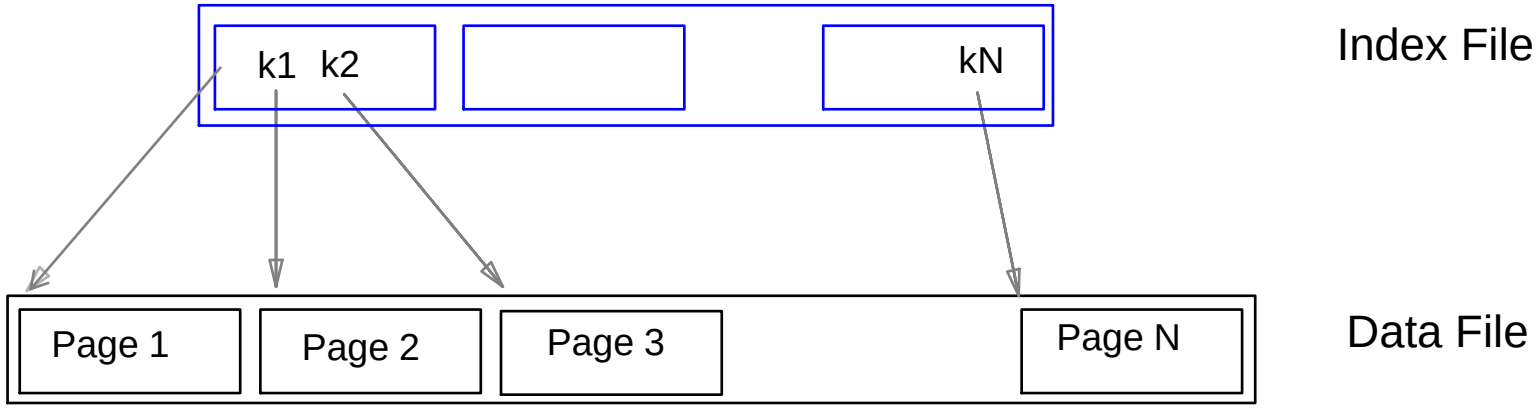
Tree-Structured Indexing

Michael Yu

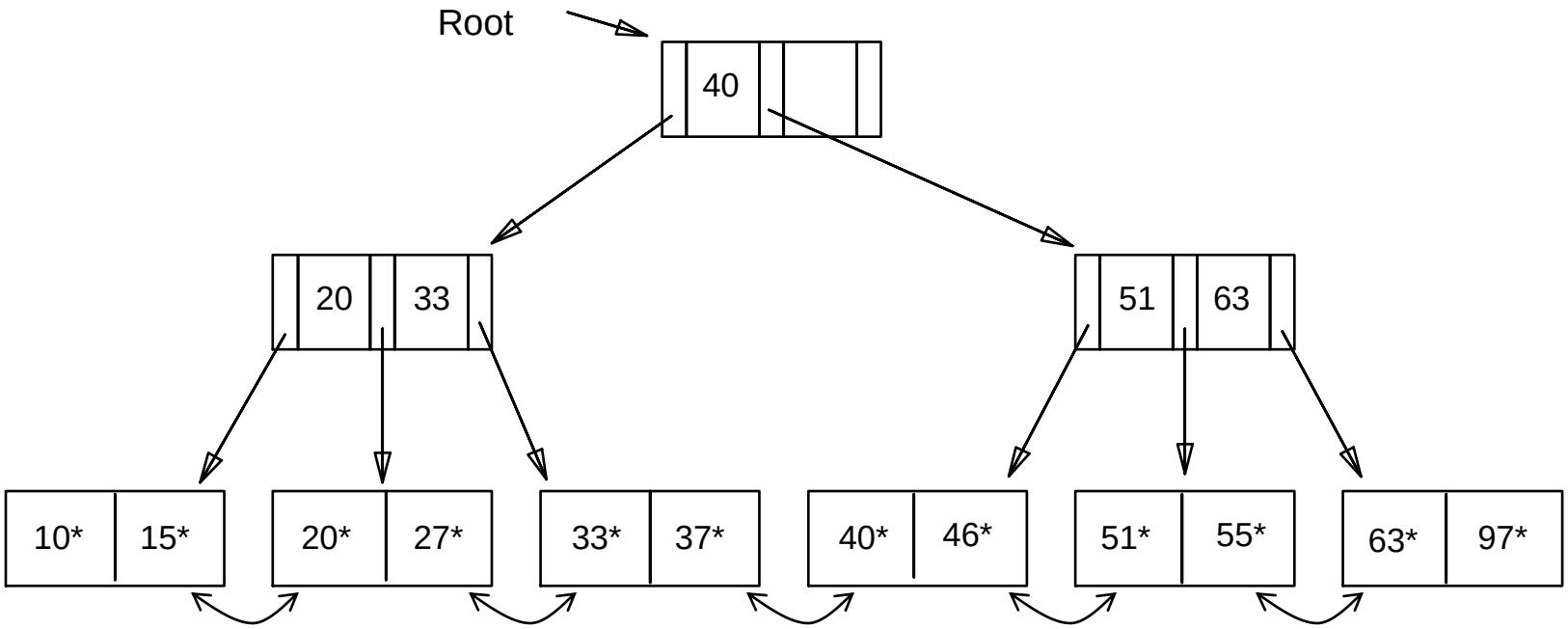
2024-2025 T1

Range Searches

- "Find all students with 3.0 < gpa < 3.5"
  - If records are sorted via gpa. We can do a binary search to find first such student, then scan to find others.
- Limitation: Cost of binary search can be quite high
  - Motivation idea: Create an 'index' file, and *do binary search on (much smaller) index file!*

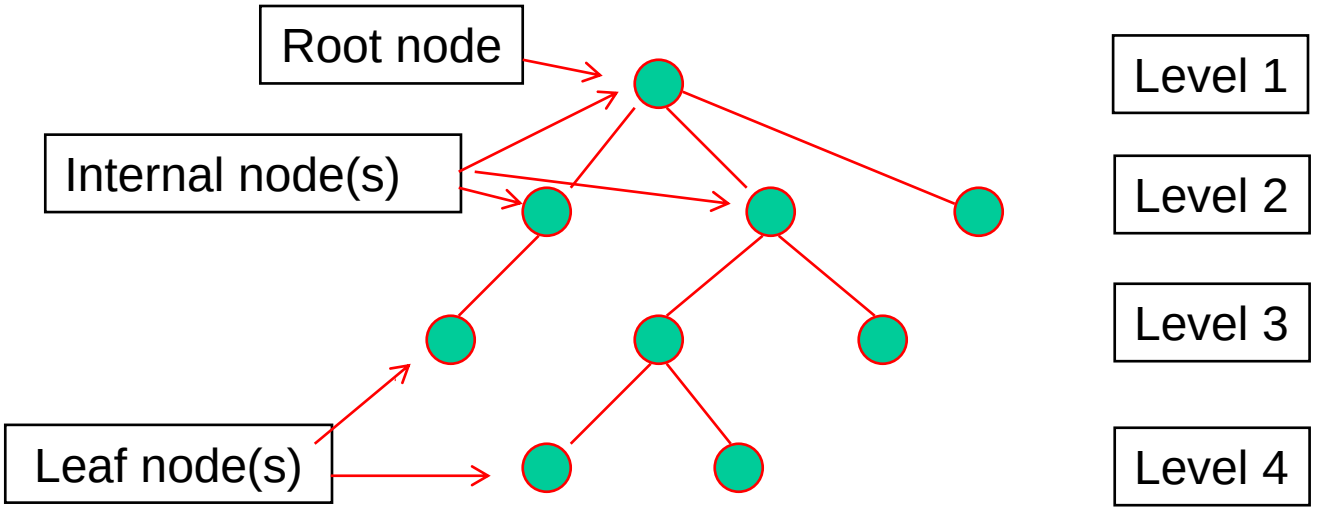


Example of B+ tree



B+ Tree: Most Widely Used Index

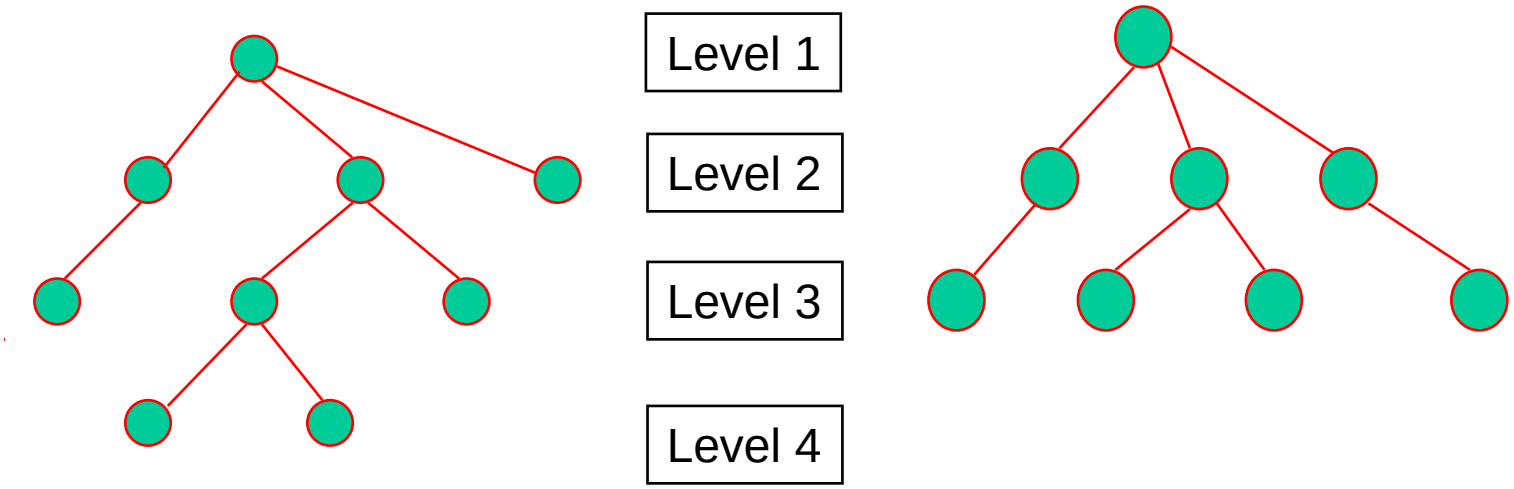
- B+ Tree is a tree (See below for concept)



If a higher level node is connected to a lower level node, then the higher level node is called a parent (grandparent, ancestor, etc.) of the lower level node

B+ Tree: Most Widely Used Index

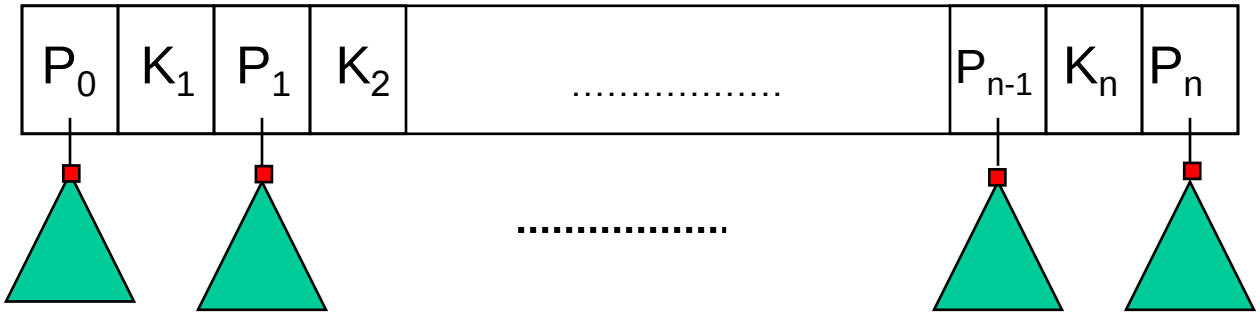
- B+ trees are all Balanced trees: all leaves at the same **level/height**
  - Height of a node: its distance to the root
  - Level of a node: height + 1



B+ Tree: Most Widely Used Index

- Structure of a B+ tree:
  - It is ***balanced:*** all leaf nodes at the same level
  - It's nodes has a special structure

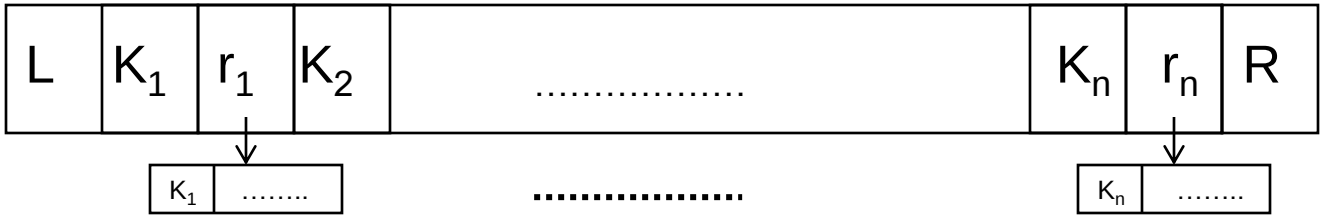
B+ tree: Internal nodes



Each P_i is a pointer to a child node, each K_i is a search key value
Pointers outnumber search key values by *exactly one.*

- **Requirements:**
  - **$K_1 < K_2 < \ldots < K_n$**
  - **If the node is not the root, we require $d \leq n \leq 2d$ where d is a pre-determined value for this B+ tree, called its *order***
  - **If the node is the root, we require $1 \leq n \leq 2d$**
  - **For any search key value K in the subtree pointed by P_i,**
    - **If $P_i = P_0$, we require $K < K_1$**
    - **If $P_i = P_1, \ldots, P_{n-1}$, we require $K_i \leq K < K_{i+1}$**
    - **If $P_i = P_n$, we require $K_n \leq K$**

B+ tree: Leaf node



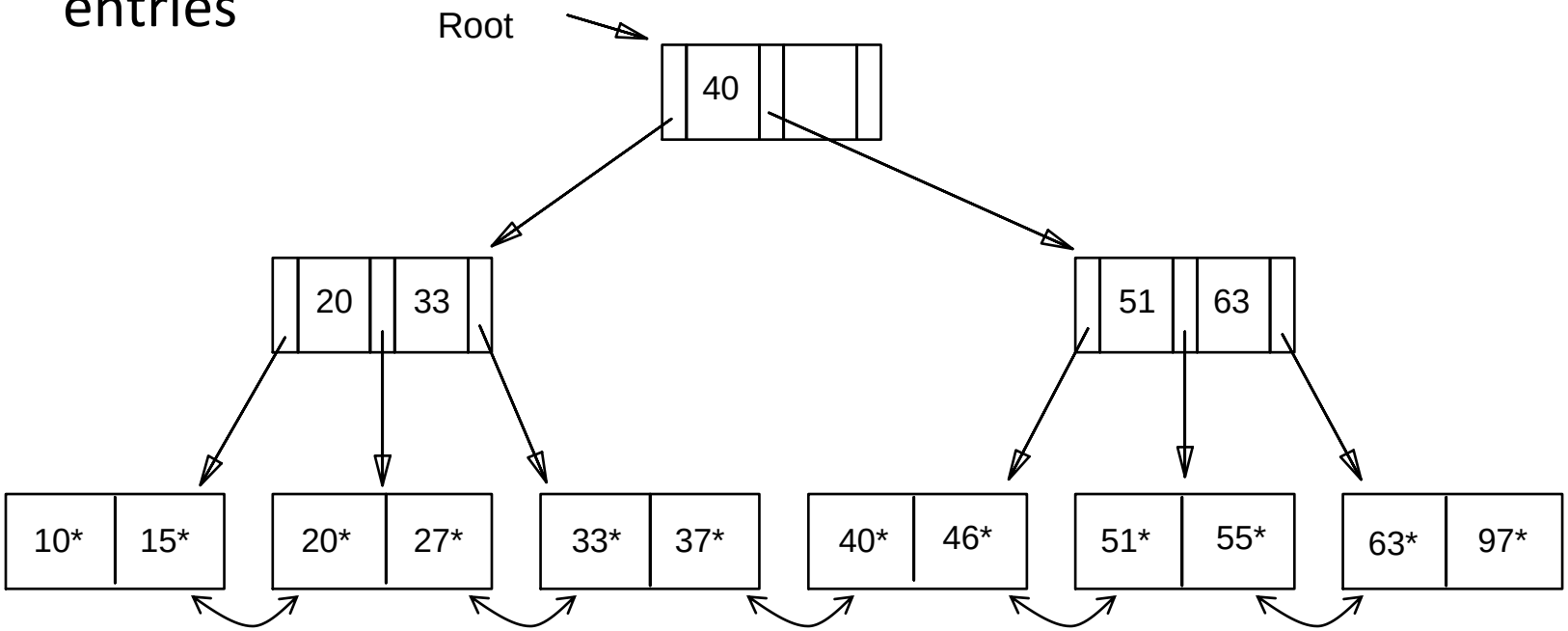
- Each r_i is a pointer to a record that contains search key value K_i …
- L points to the left neighbor, and R points to the right neighbor
- $K_1 < K_2 < \ldots < K_n$
- We require $d \leq n \leq 2d$ where d is the order of this B+ tree
- We will use K_i* for the pair K_i, r_i and omit L and R for simplicity

Order of B+ Tree

- **The number d is the order of a B+ tree**.
- Each node (with the exception of the root node) must have $d \leq x \leq 2d$ entries
  - However, it's technically possible for leaf nodes to temporarily end up with < d entries immediately after you delete data, we do not consider this specific case for now.

Example: A B+ tree with order of 1

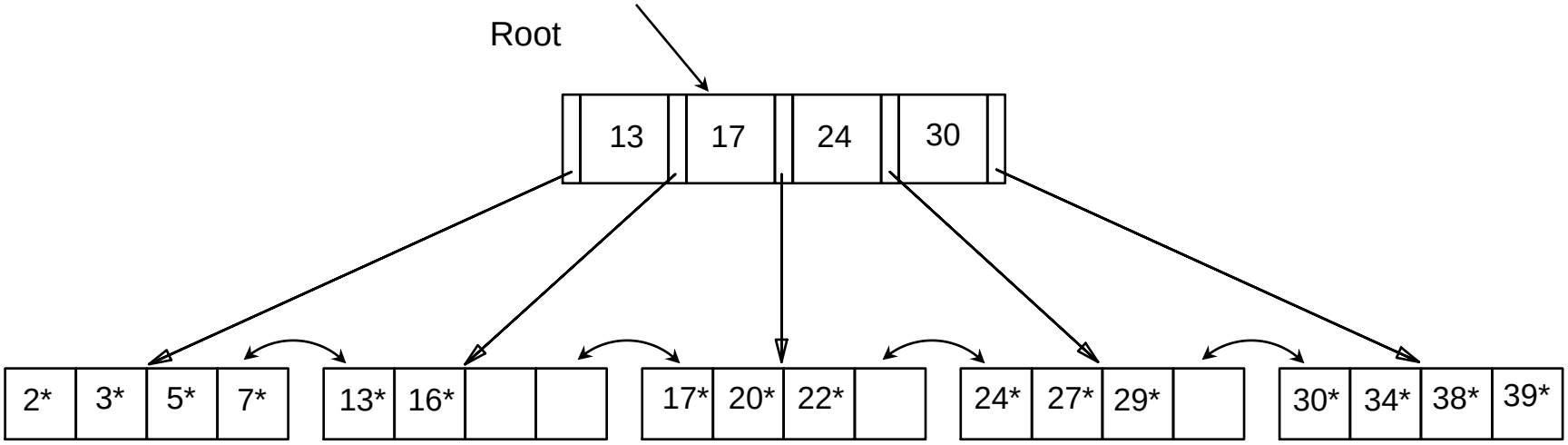
- Each node must hold at least 1 entry, and at most 2 entries



- Given search key values 27, 51, 64, how to find the rids?
  - Search begins at the root, and key comparisons direct it to a leaf

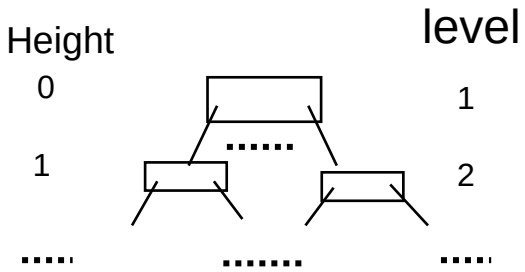
Example: a B+ tree with order 2

- Search for 5*, 15*, all data entries >= 24* ...
- The last one is a range search, we need to do the sequential scan, starting from the first leaf containing a value >= 24.



Cost for searching a value in B+ tree

- In general nodes are blocks/**pages**
- Let **H** be the height of the B+ tree: need to read **H+1** pages to reach a leaf node
- Let **F** be the (average) number of pointers in a node (for internal node, called *fanout factor*)
  - Level 1 = 1 page = F^0 page
  - Level 2 = F pages = F^1 pages
  - Level 3 = Fx F pages = F^2 pages
  - Level H+1 = …….. = F^H pages (i.e., leaf nodes)
- Suppose there are D data entries. So there are D/(F-1) leaf nodes
  - D/(F-1) = F^H. That is, H = $\log_F(\frac{D}{F-1})$



If the fanout factor is F, we usually assume a leaf node stores F-1 data entries.

B+ Trees in Practice

- Typically, a node is a page
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133 (i.e, # of pointers in internal node)
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
- Suppose there are 1,000,000,000 data entries.
  - $H = \log_{133}(1000000000/132) < 4$
  - The cost is 5 pages read