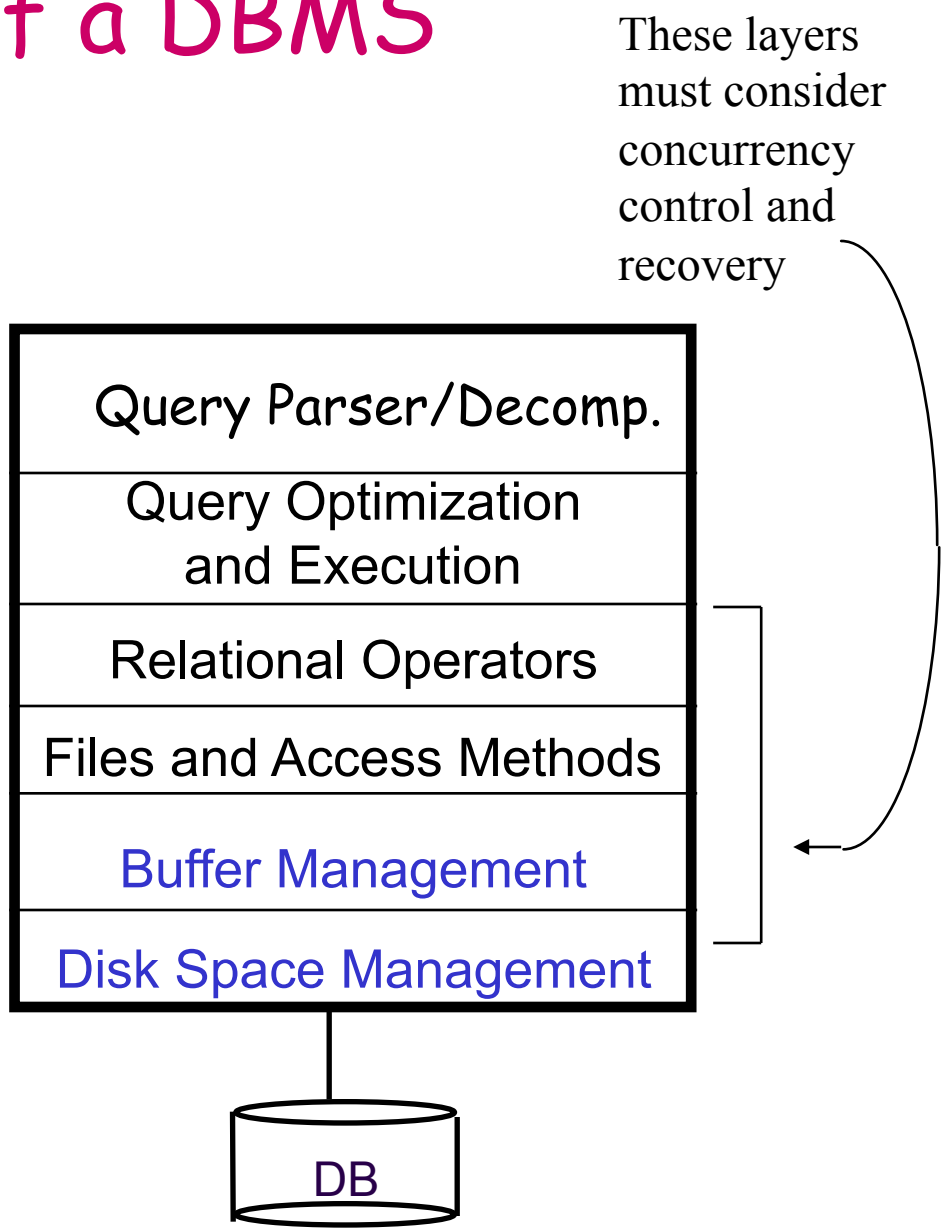
Structure of a DBMS

- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.

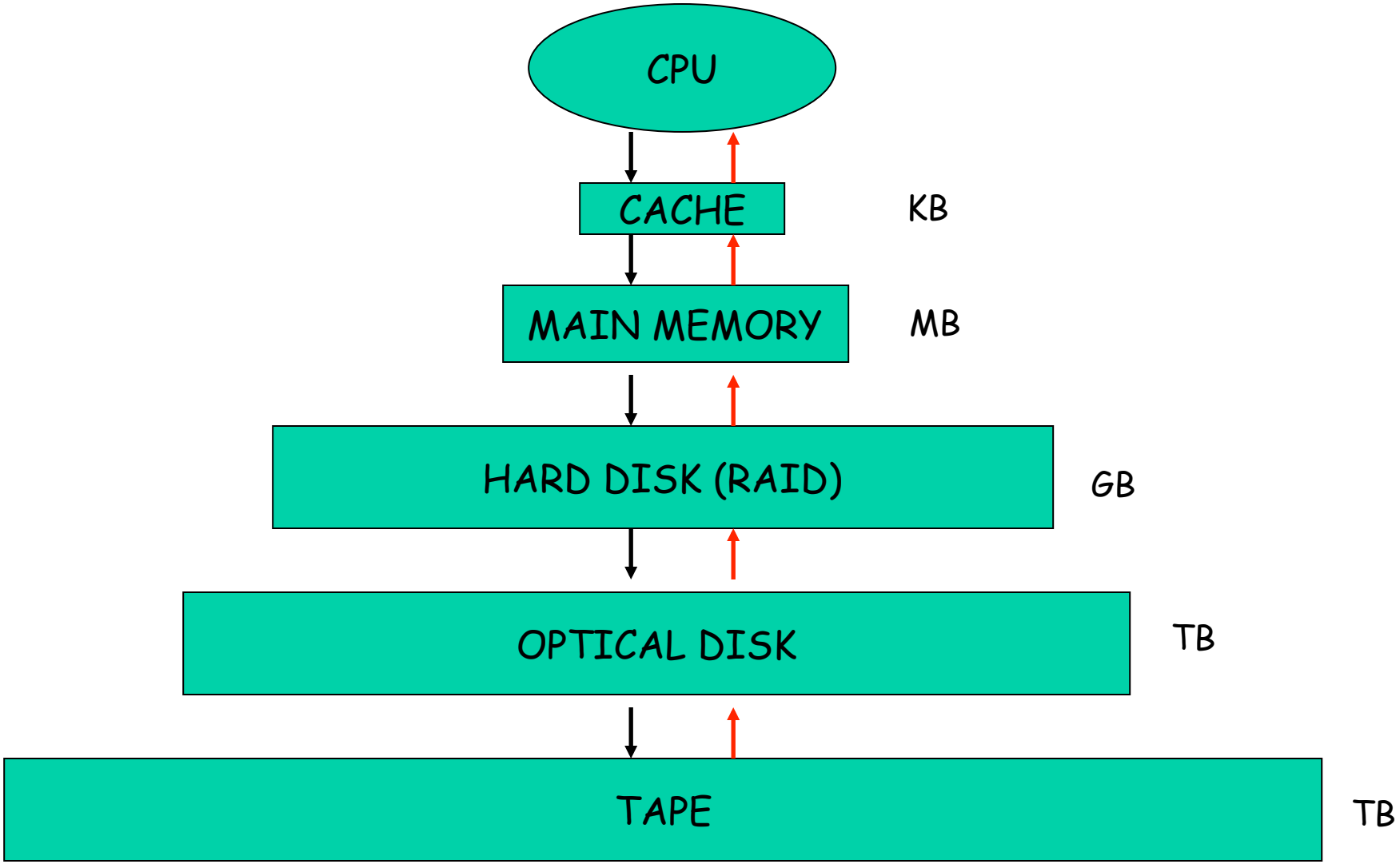


Data Storage

DBMS deals with a very large amount of data.

- How does a computer system store and manage very large volumes of data?
- What representations and data structures best support efficient manipulations of this data?

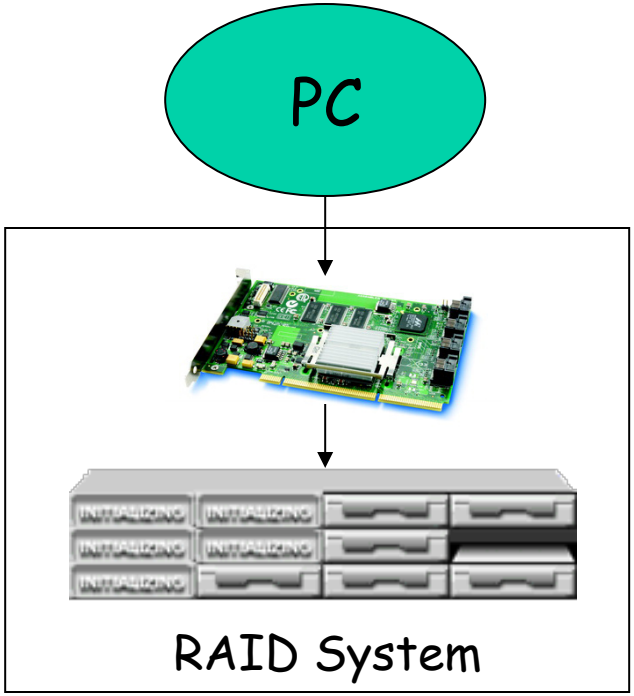
Storage Hierarchy



Redundant Array Inexpensive Disk

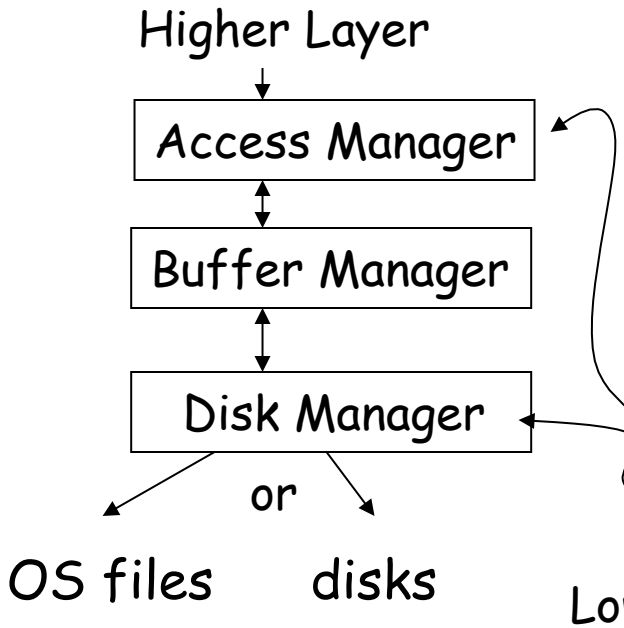
RAID: A number of disks is organized and appears to be a single one to the OS

- Aggregate disk capacity
- Increase I/O throughput
- Fault-tolerant (hot swapping)



Storing databases/tables/records

- Each database contains a number of tables
- Each table contains a number of records
- Each record has a unique identifier called a *record id*, or *rid*.
  - Records can be stored in files based on the underlying OS.
  - Records can be stored directly to disk blocks bypassing file systems (raw devices).



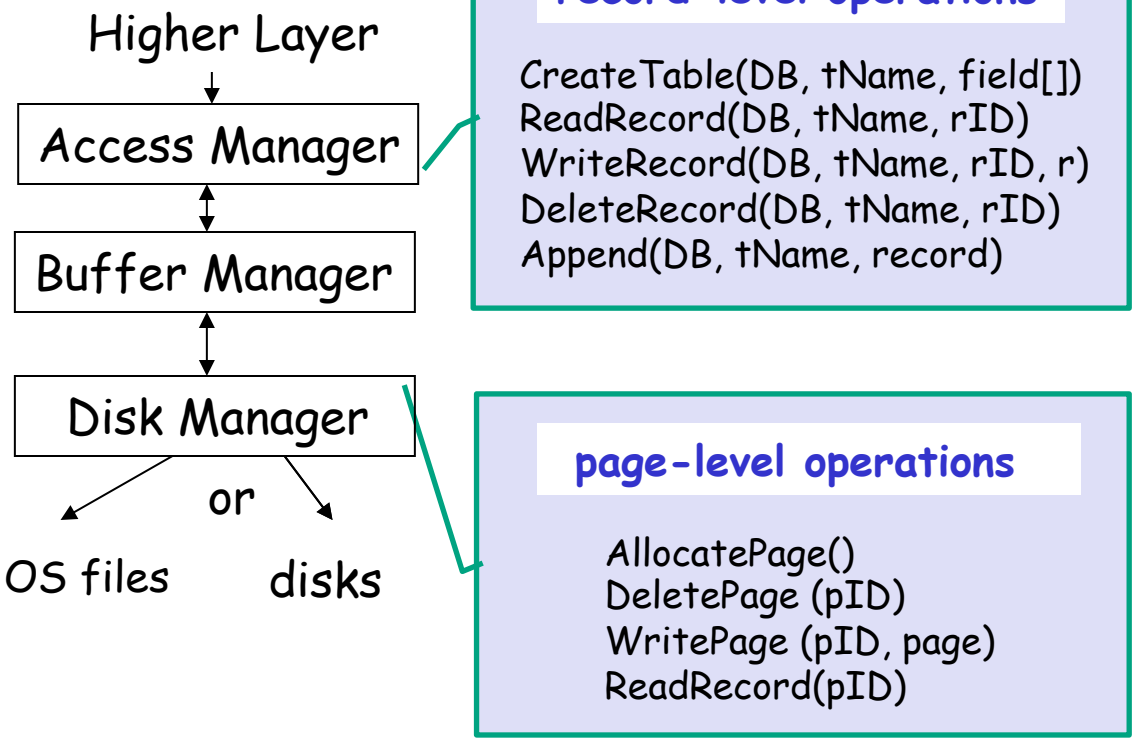
Why raw devices?

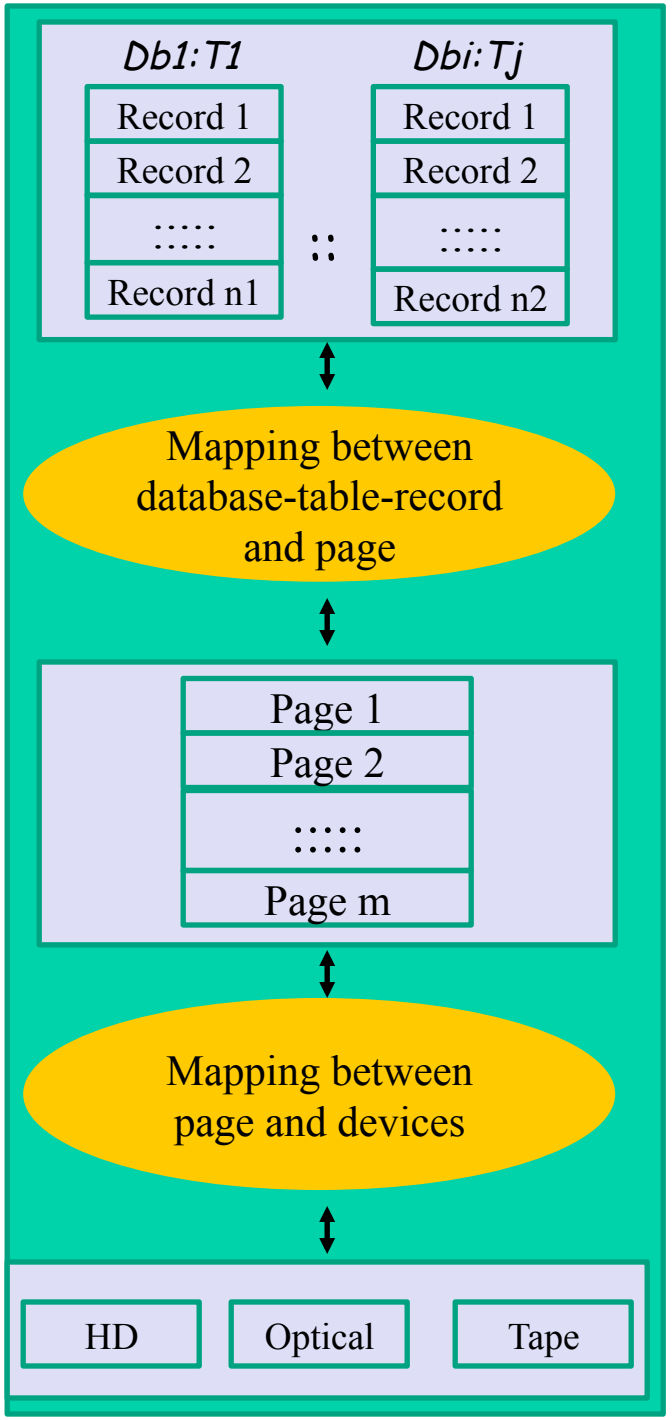
- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks.
- Performance

Arrange records in a page.

Lowest level in DBMS software architecture

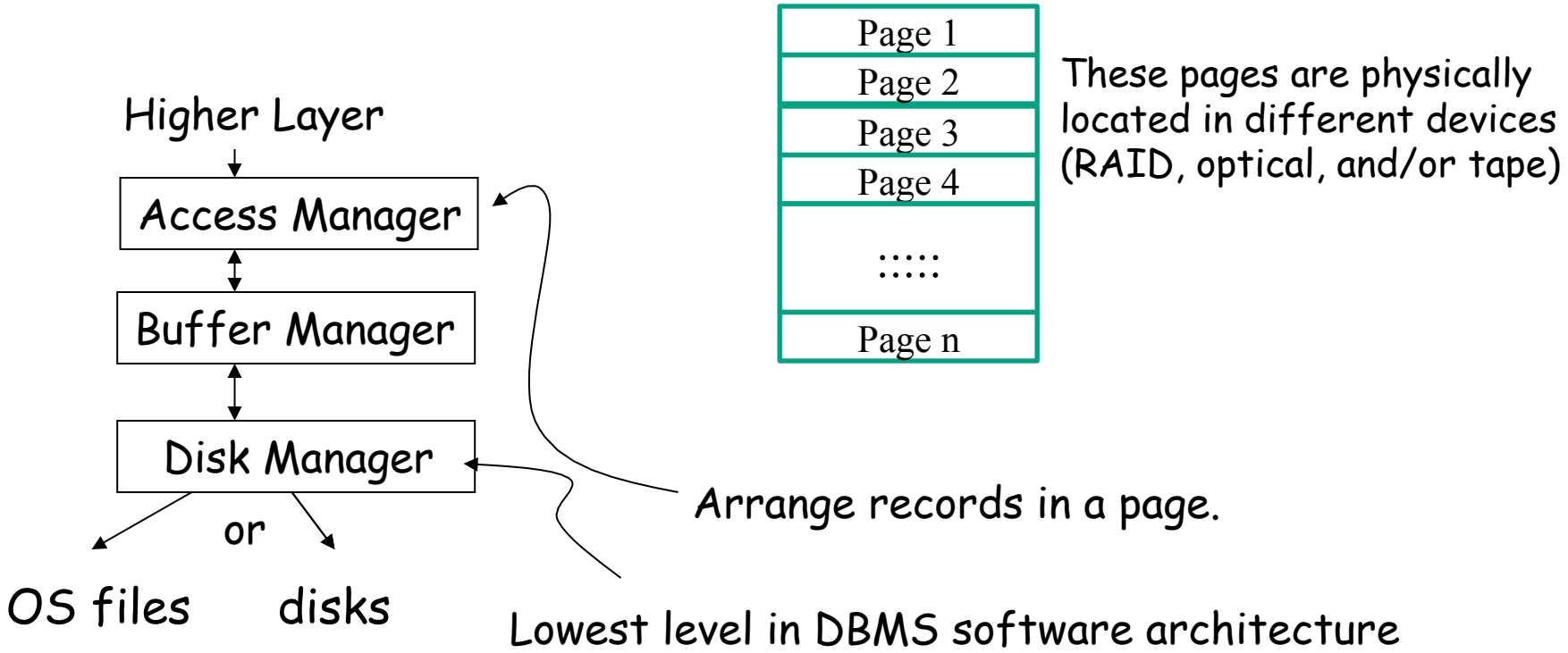
An overview at 30000-feet High



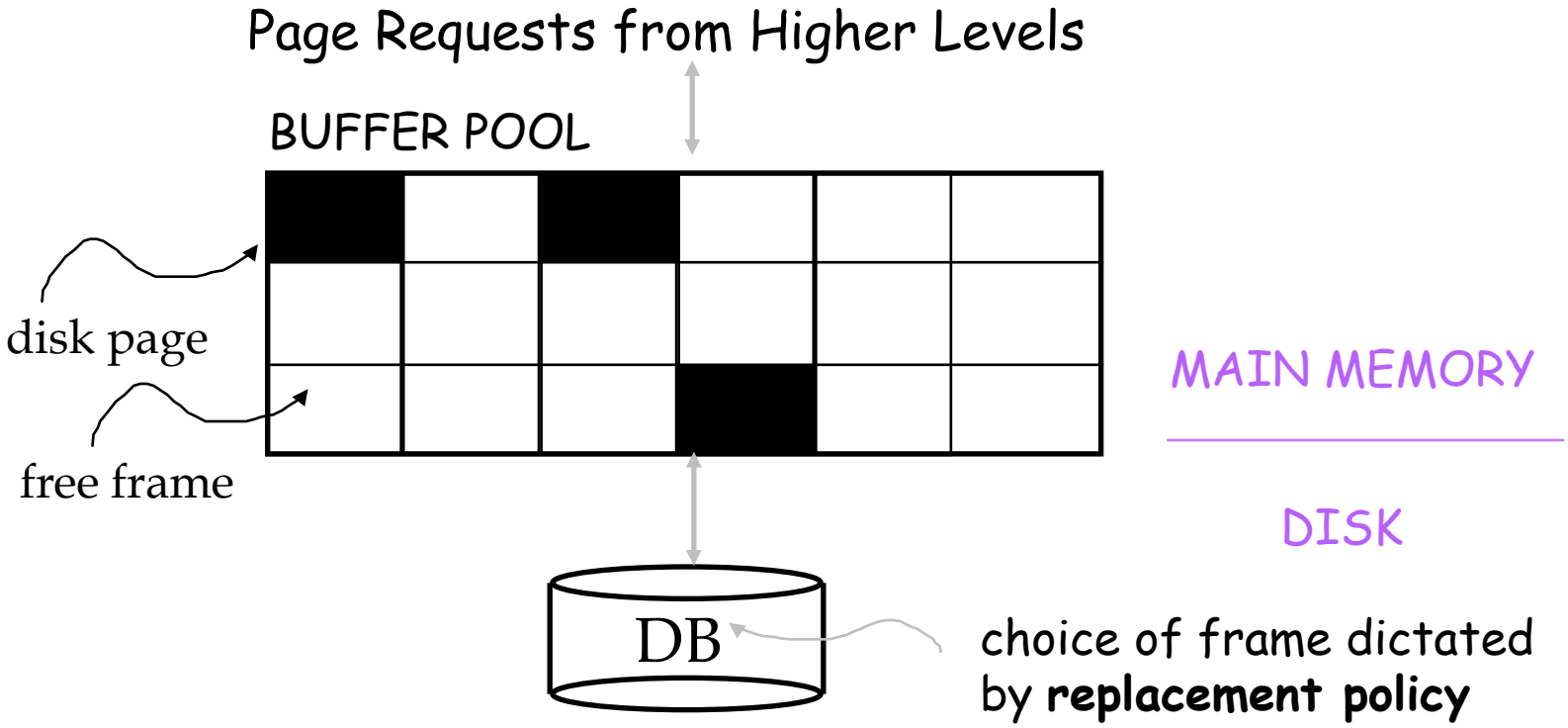


Disk Manager

- Higher levels call upon this layer to:
  - allocate/de-allocate a page on disk
  - read/write a page (or a block or a unit of disk retrieval)



Buffer Manager



Size of a frame equal to size of a disk page
Two variables associated with each frame/page:

- Pin_count: Number of current users of the page
- Dirty: whether the page has been modified since it has been brought into the buffer pool.

Buffer Management
When a Page is Requested ...

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

- *Pin* the page and return its address to the requester.

  Pinning: Incrementing a pin_count.
  Unpinning: Release the page and the pin_count is decremented.

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

Buffer Replacement Policy

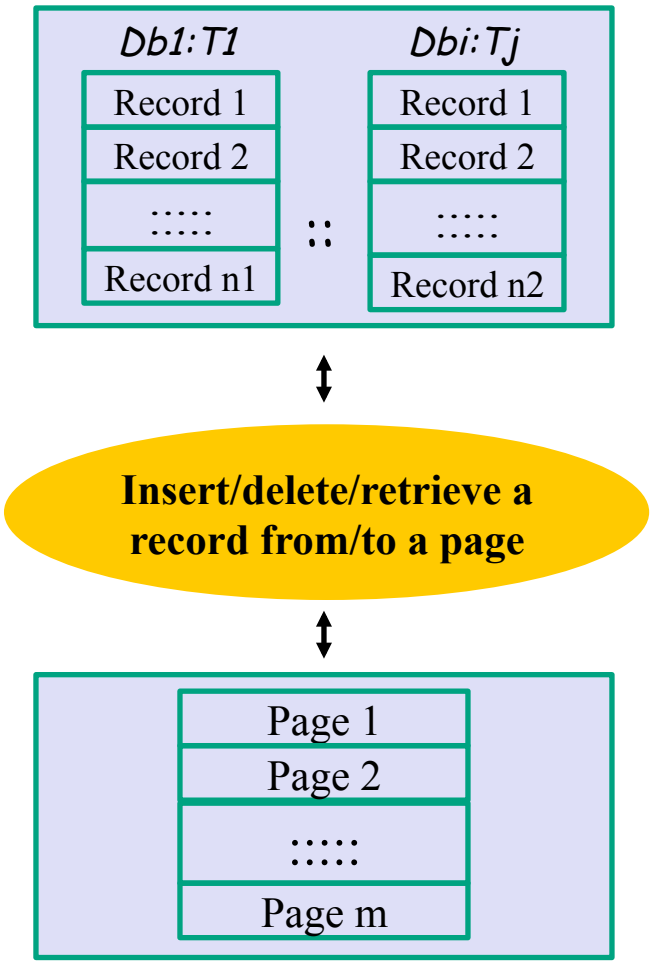
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), First-In-First-Out (FIFO), Most-recently-used (MRU) etc.

- Policy can have big impact on # of I/O's; depends on the *access pattern.*

- *Sequential flooding*: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

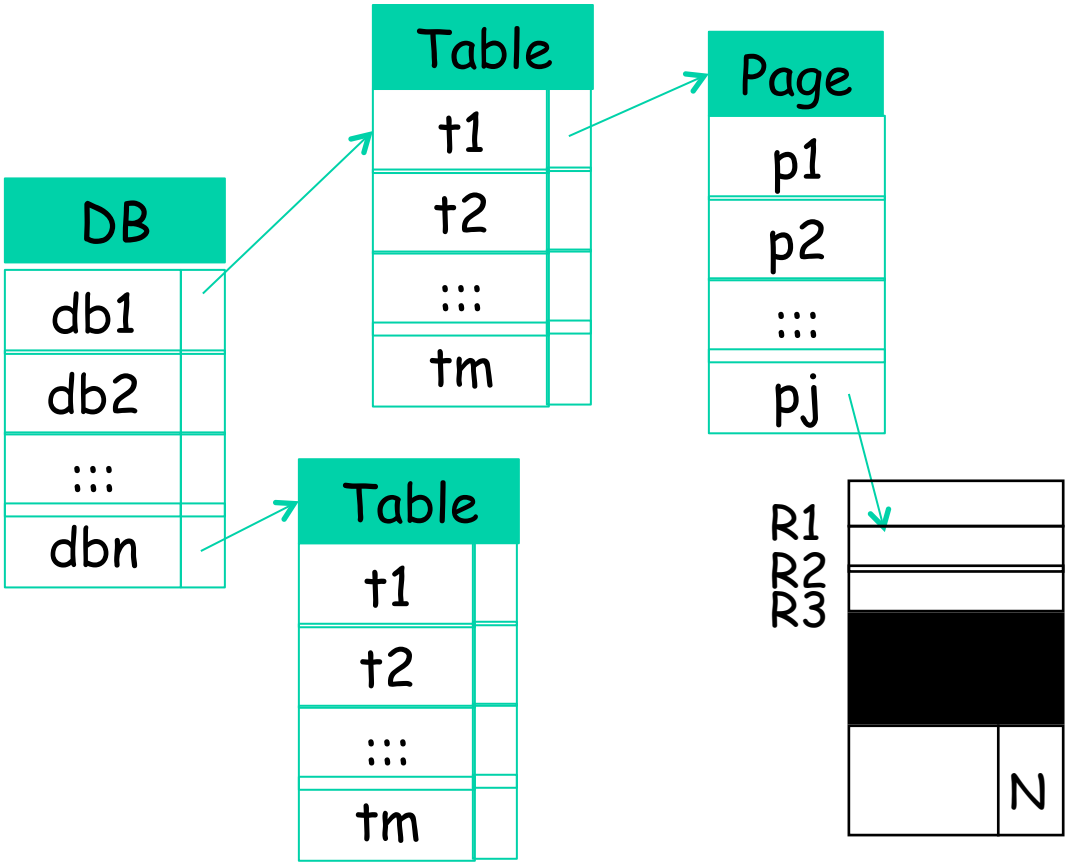
Access Manager

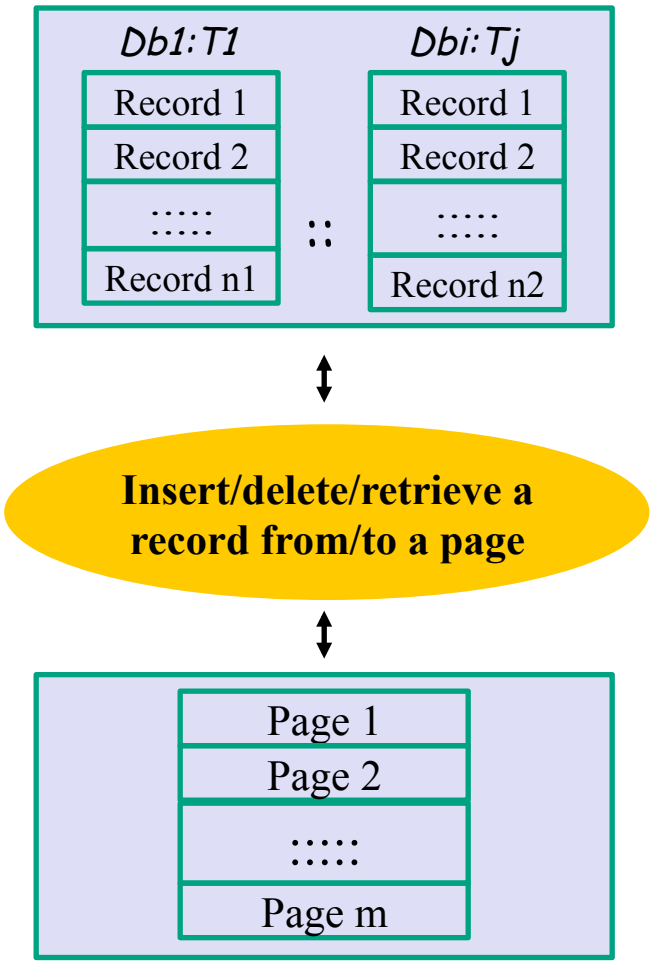
- Arrange records into a page
- Retrieve records from a page
- What to concern
  - Mapping between record and page
  - Record formats
  - Page formats
  - File formats



Record-Page Mapping

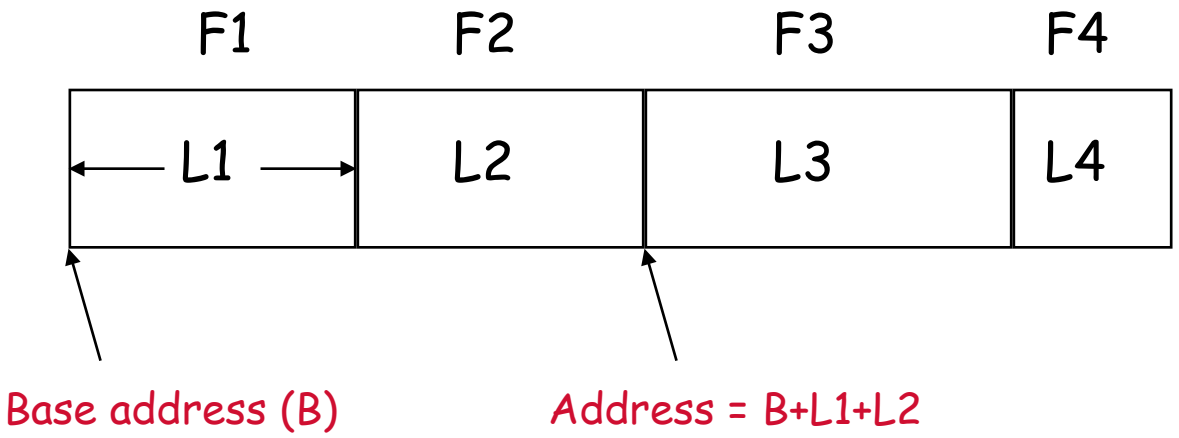
- Maintain a mapping table





Record Formats: Fixed Length

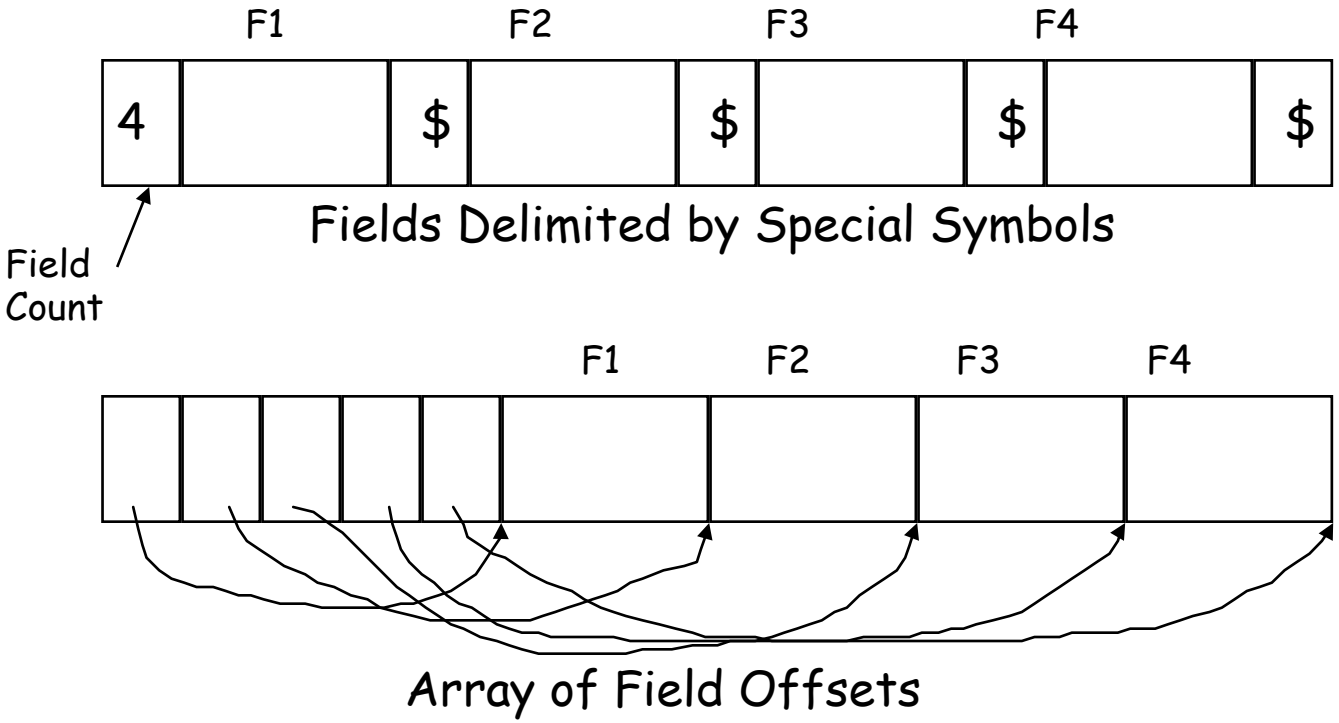
Name: char (40)
Address: char (100)
Phone: char (10)
Email: char (100)



- Information about field types and lengths is stored in *system catalogs.*
  - Li: Size of field i in bytes

Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



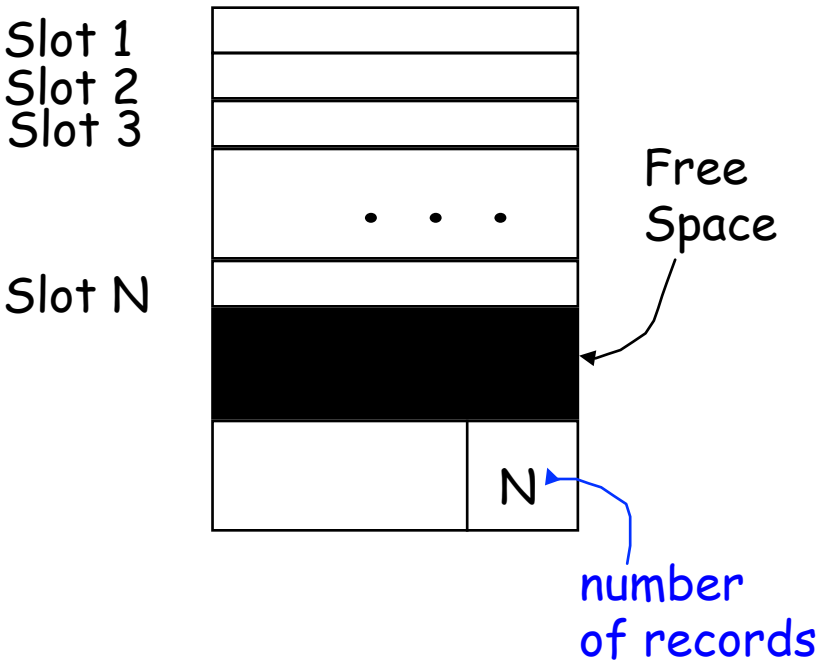
Can be used for fixed length fields Or Variable length fields

VARCHAR
BLOB

Second offers direct access to i'th field, efficient storage of *nulls*; small directory overhead.

Page Formats: Fixed Length Records

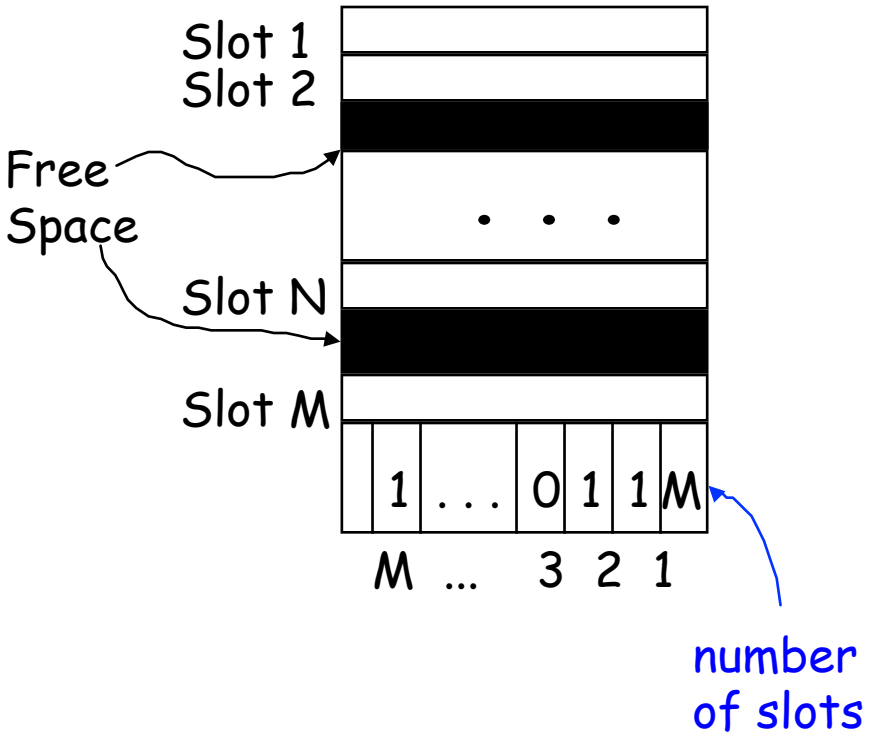
Solution 1: PACKED



- Each slot holds one record
- Record the number of records (total = PageSize/RecordSize)
- The free slot starts from N+1
- When appending a new record, allocate slot N+1, then update N ++
- When deleting a record at slot i, move the last record to the slot i. If sorted, all records after slot i must be moved up

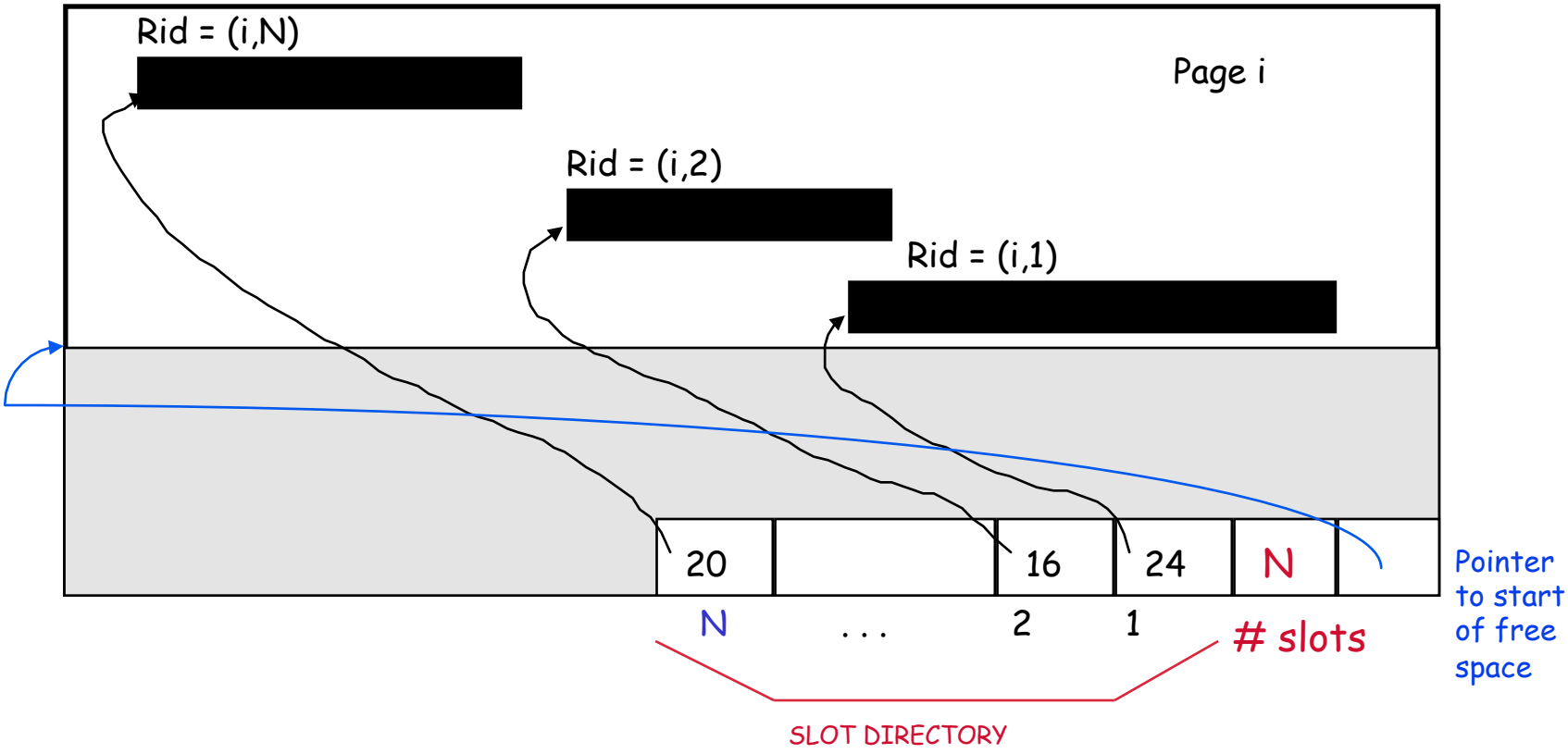
Page Formats: Fixed Length Records

Solution 1: UNPACKED



- Each slot holds one record
- Total number of slots is PageSize/RecordSize
- Need a bitmap to record if a slot is occupied or not
- If bit[i]==1, slot i is occupied
- When inserting a record, search the bitmap to find a bit that is 0, then allocate the corresponding slot for the record
- When deleting a record, simply reset the corresponding bit

Page Formats: Variable Length Records



☛ Can move records on page without changing rid; so, attractive for fixed-length records too.

File Formats and Operation Costs

- Format
  - Heap File: Suitable when typical access is a file scan retrieving all records.
  - Sorted File: Best if records must be retrieved in some order, or only a `range' of records is needed.
  - Hashed File: Good for equality selections.

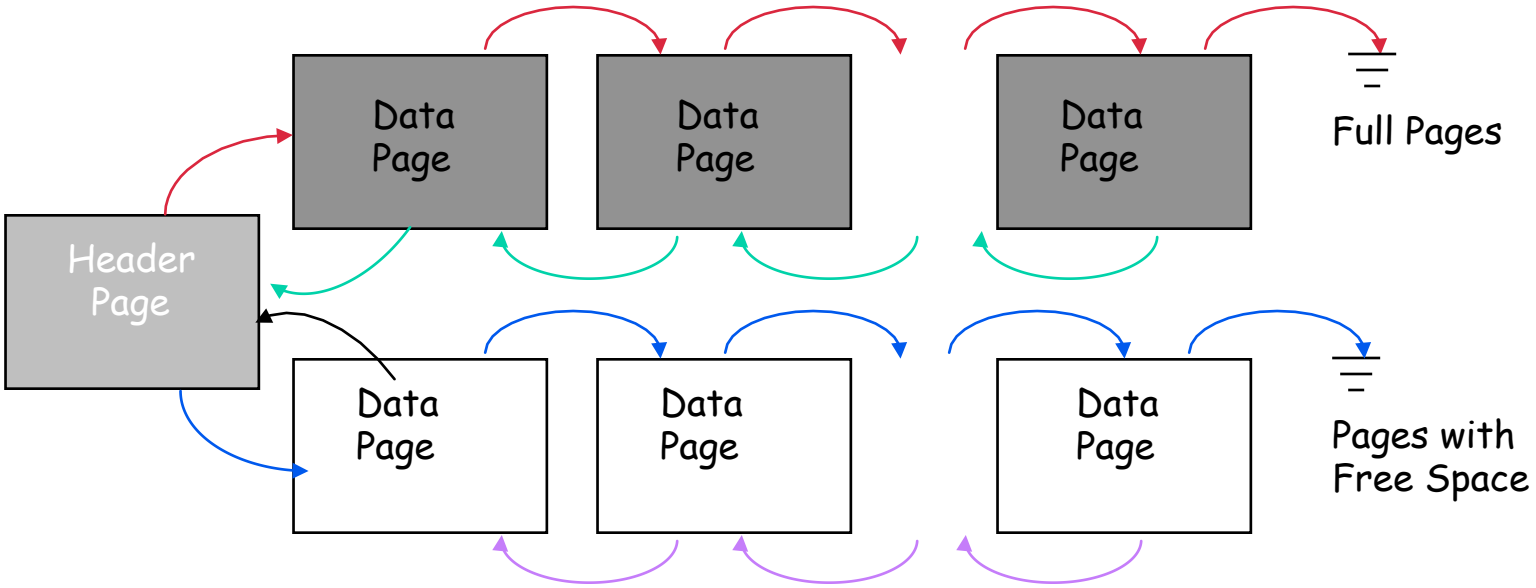
- Cost factors (we ignore CPU costs, for simplicity)
  - **P:** The number of data pages
  - **R:** Number of records per page
  - **D:** (Average) time to read or write disk page

  - Measuring number of page I/O's ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.

Heap Files

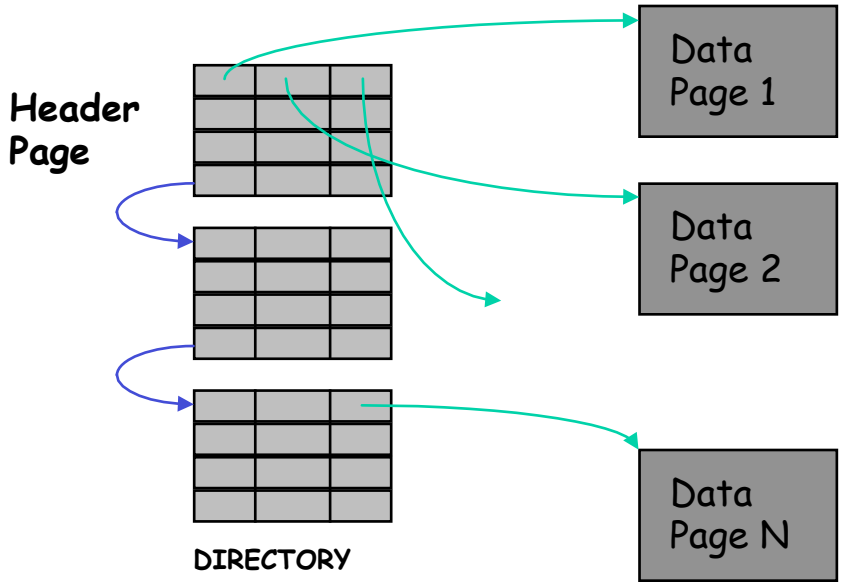
- The data in a heap file is not ordered.
  - How to find a page that has some free space
  - How to find the free space inside a page
    - Depend on record format, i.e., fixed-length or variable length
- Two types of implementations
  - Link-based
  - Directory-based

Heap File Implemented as a List



- To insert a record, one searches the pages with free space and find the one that has sufficient space
  - Many pages may contain some tiny free space
  - A long list of pages may have to loaded in order to find an appropriate one

Heap File Using a Page Directory



- The directory is a collection of pages;
  - Each page contains a number of entry
  - Each entry contains a pointer linking to the page and a variable recording the free space
- The number of directory pages is much smaller than that of data pages

Heap Files and Associated Costs

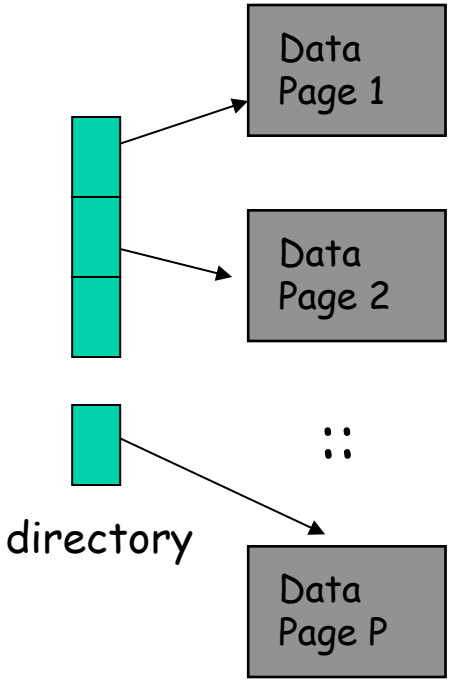
Scan: P*D

Search with equality selection: If a selection is based on a candidate key, on average, we must scan half the file, assuming that the record exists 0.5*P*D.

Search with range selection: The entire file must be scanned. The cost is P*D.

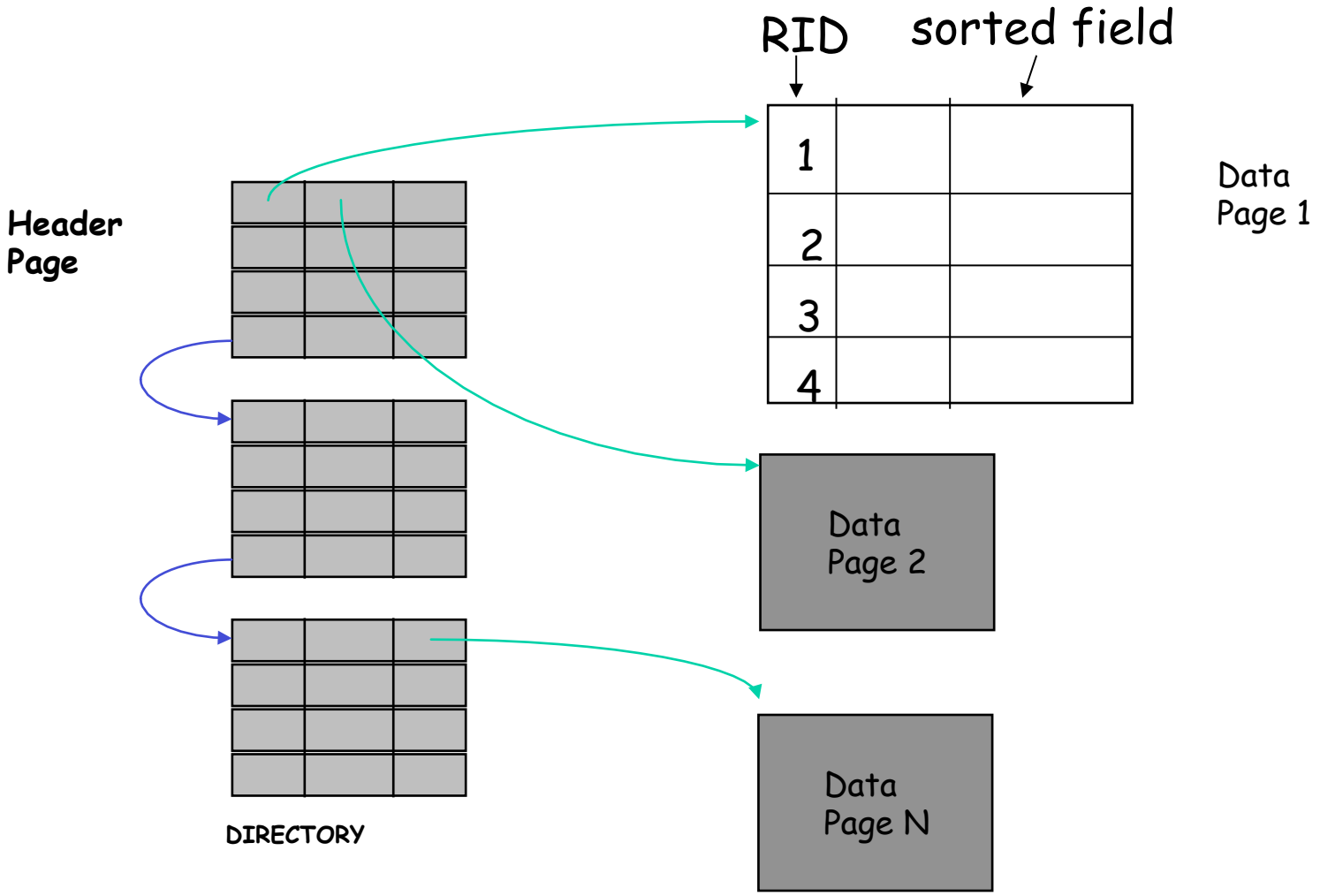
Insert: Assume that records are always inserted at the end of the file. We fetch the last page in the file, add the record, and write the page back. The cost is 2D.

Delete:. The cost also depends on the number of qualifying records. The cost is search cost + D.



P: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

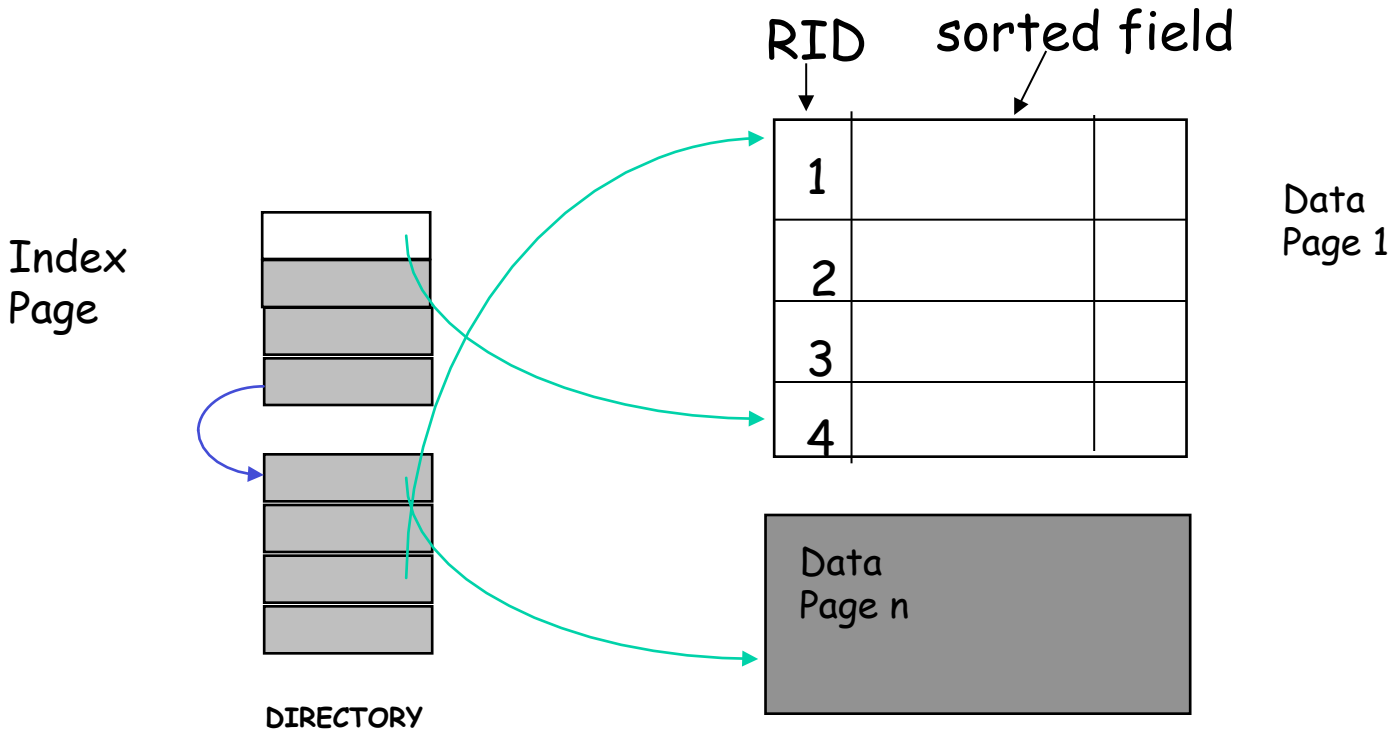
Sorted File



Sort records directly

- Expensive when inserting a record

Sorted File



- Keep the sorted field using index page
  - Each entry points to a record
    - May contain the value of the sorted field
    - May contain a valid bit for deleting operation
  - The entries are sorted according to the sorted field
  - Inserting a record just need to reorganize the index page
    - The size is much smaller

Sorted Files and Associated Costs

Scan: P*D

Search with equality selection: Assume that the selection is specified on the field by which the file is sorted. The cost is D * log P assuming that the sorted file is stored sequentially.

Search with range selection:

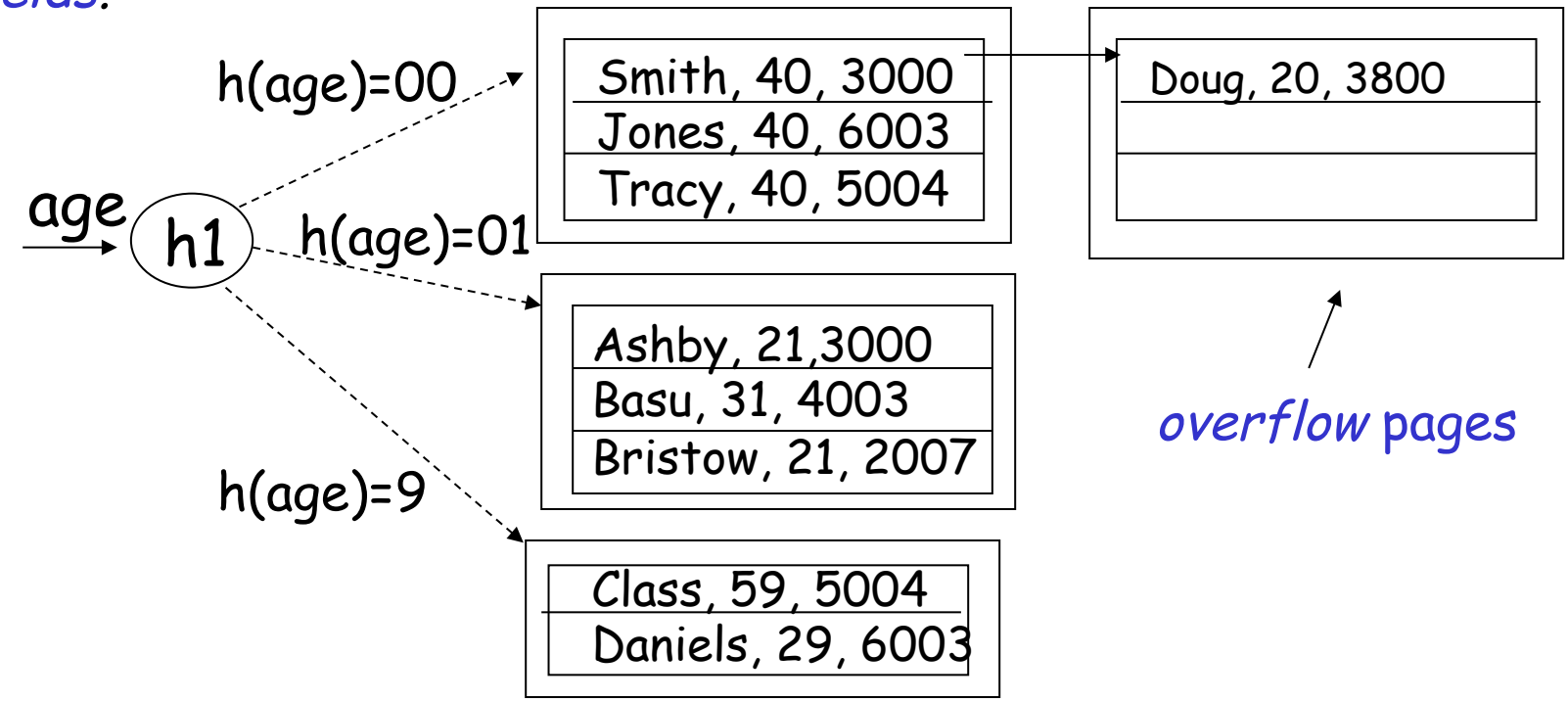
D * log P + cost of retrieving qualified records.

Insert: Search cost + 2*0.5*P*D; the assumption is that the inserted record belongs in the middle of the file.

Delete: Search cost + 2*0.5*P*D; the assumption is that we need to pack the file and the record to be deleted is in the middle of the file.

Hashed files

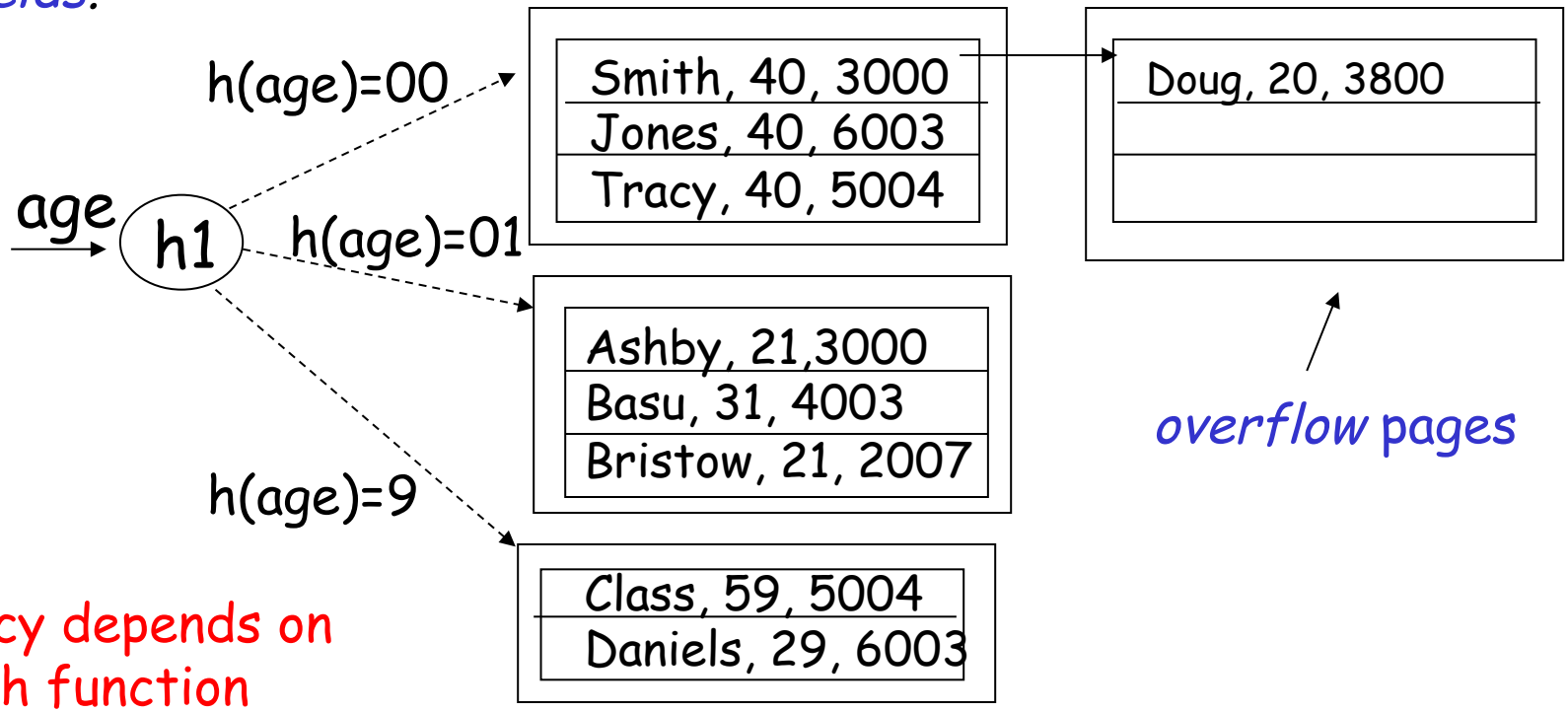
- File is a collection of *buckets.*
- Bucket = *primary* page plus zero or more *overflow* pages.
- Hashing function **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at only some of the fields of *r*, called the *search fields*.



File hashed on age

Hashed files

- File 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 only some of the fields of *r*, called the *search fields*.



File hashed on age

Efficiency depends on
- Hash function
- Data skew factor

Hashed Files and Associated Costs

Assume that there is no overflow page.

Scan: 1.25*P*D if pages are kept at 80% occupancy

Search with equality selection:D

Search with range selection: 1.25*P*D

Insert: Search cost + D = 2D

Delete: Search cost + D = 2D

Cost Comparison

	Heap File	Sorted File	Hashed File
Scan all records	BD	BD	1.25 BD
Equality Search	0.5 BD	D log $_2$B	D
Range Search	BD	D (log $_2$B + # of pages with matches)	1.25 BD
Insert	2D	Search + BD	2D
Delete	Search + D	Search + BD	2D

☛ Several assumptions underlie these (rough) estimates!