The University of Texas ARLINGTON

Storage Structures Introduction
Chapter 8 (3rd edition)

Sharma Chakravarthy
UT Arlington
sharma@cse.uta.edu

Disks and Files

- DBMS stores information on ("hard") disks.
  - Known as HDDs in contrast to SSDs

- This has major implications for DBMS design!

  Remember impedance mismatch!

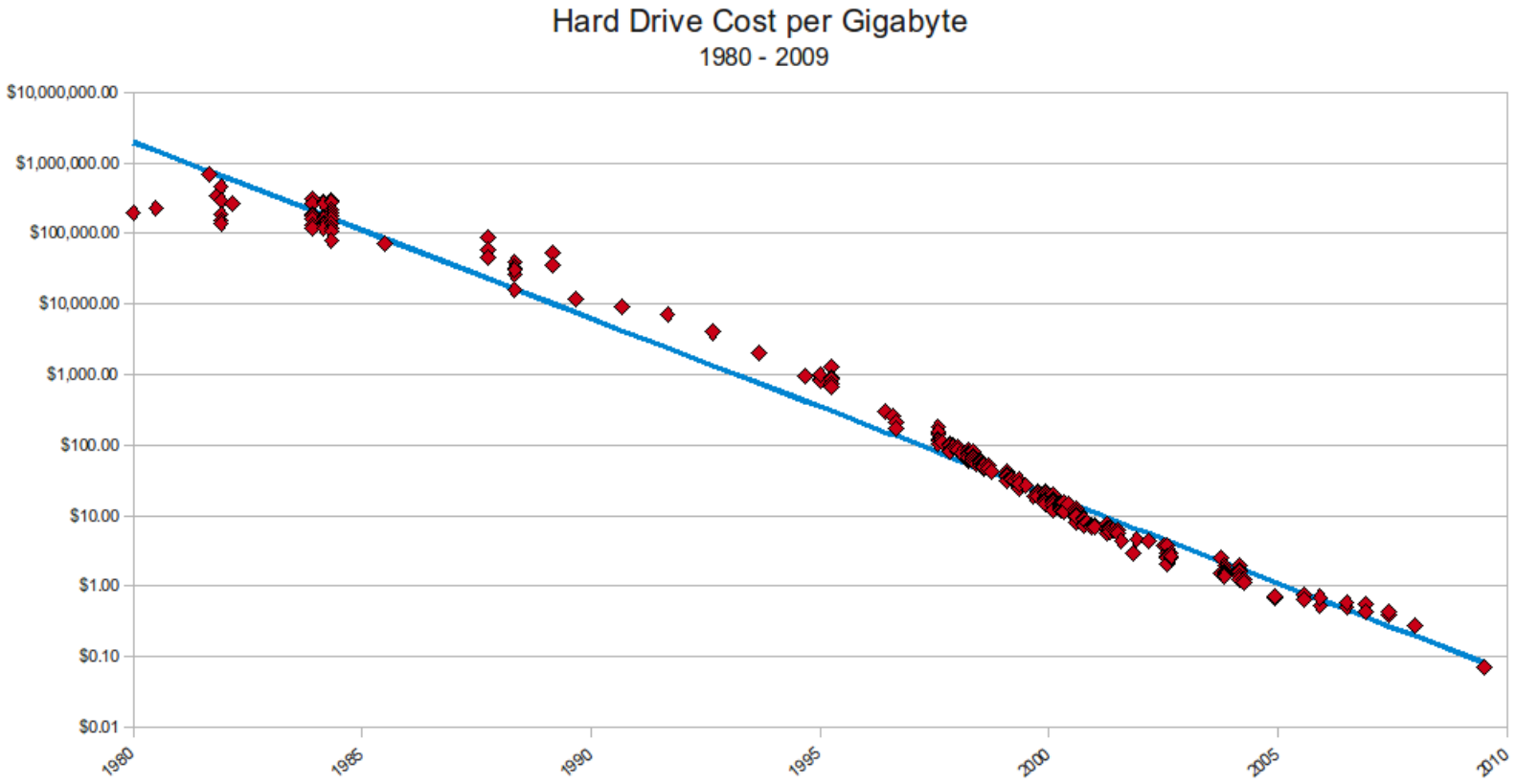
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.

  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store Everything in Main Memory?

- *Costs too much*.
  - 1 TB of RAM is $8000+
  - 1 TB of SSD for $150 portable SSD $250
  - 8 TB of HDD for $250
- *Main memory is volatile*. We want data to be saved (persistent) between runs. (Obviously!)
- Typical storage hierarchy:
  - Cache – most expensive (L1 and L2)
  - Main memory (RAM) for currently used data.
  - Solid state disks (SSD) -- New !! Still expensive
  - Disk for the main database (storage in Peta bytes).
  - Other storage devices (flash card, usb stick, …)
  - Tapes for archiving older versions of the data (tertiary storage).
  - DVD/CDROM and other devices – least expensive

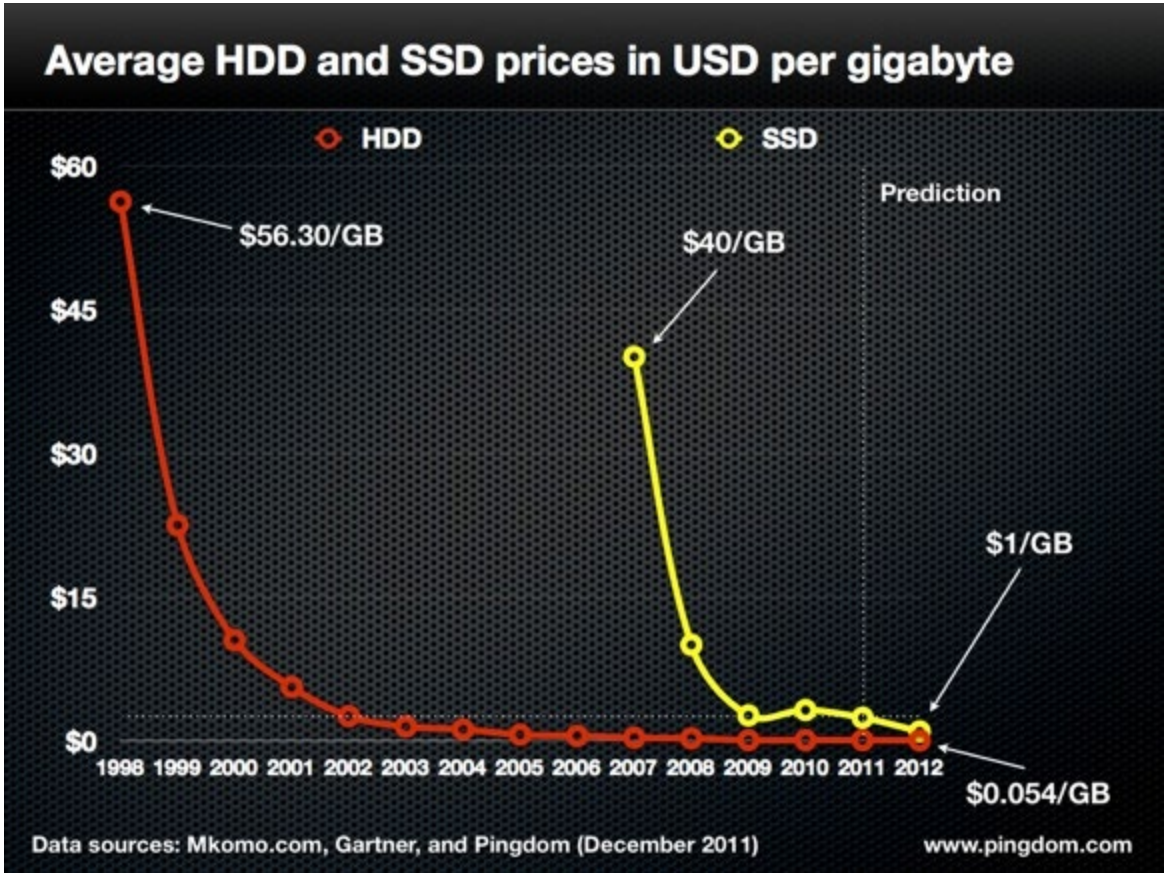
Over the last 30 years, space per unit cost has doubled roughly every 14 months (increasing by an order of magnitude every 48 months)



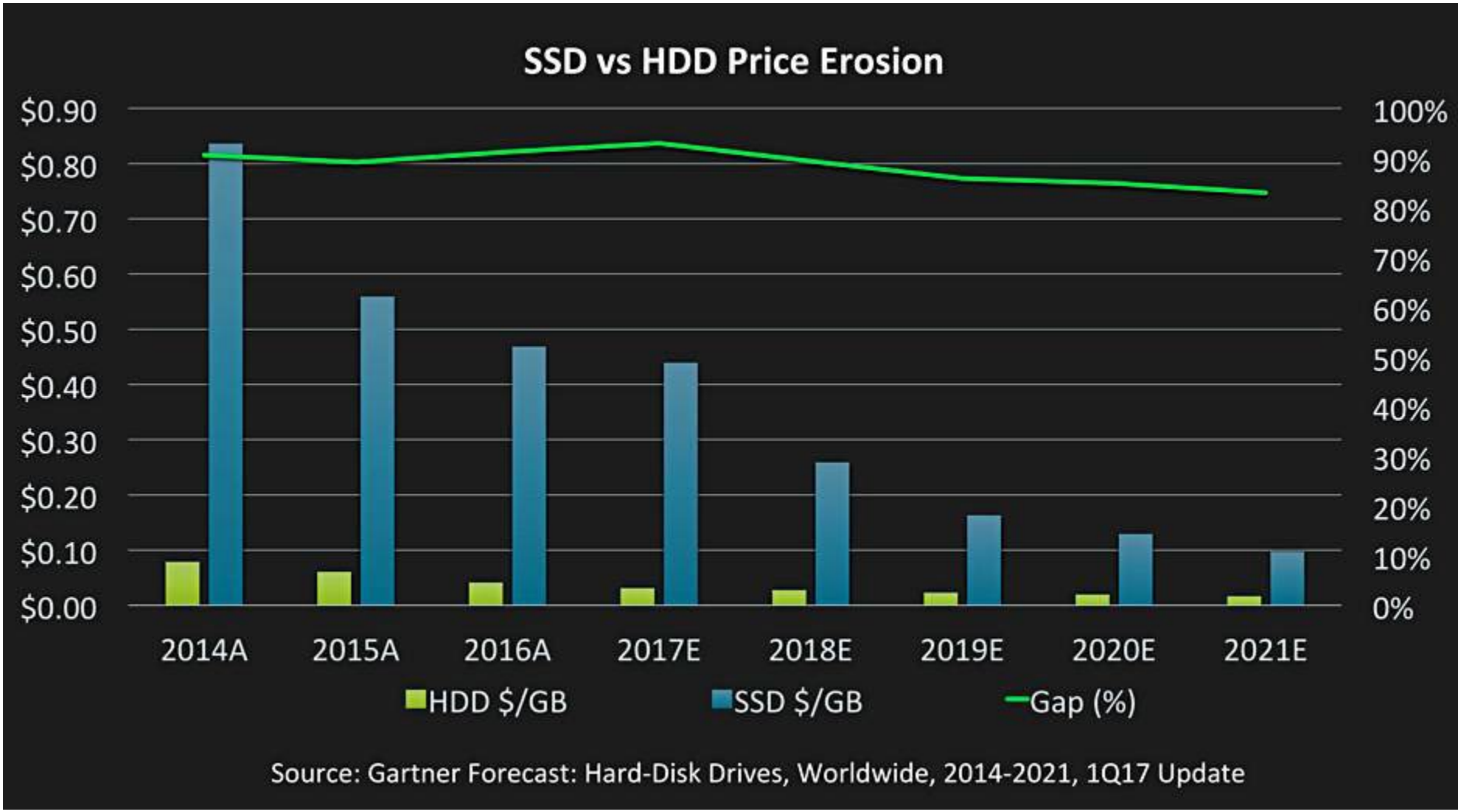
SSD Vs. HDD

☞HDD 3c/gb in 2014

☞SSD is .35c/gb in 2014



☞From: www.tomshardware.com


SSD vs HDD Price Erosion
$0.90
$0.80
$0.70
$0.60
$0.50
$0.40
$0.30
$0.20
$0.10
$0.00
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2014A
2015A
2016A
2017E
2018E
2019E
2020E
2021E
HDD $/GB
SSD $/GB
Gap (%)
Source: Gartner Forecast: Hard-Disk Drives, Worldwide, 2014-2021, 1Q17 Update

Disks

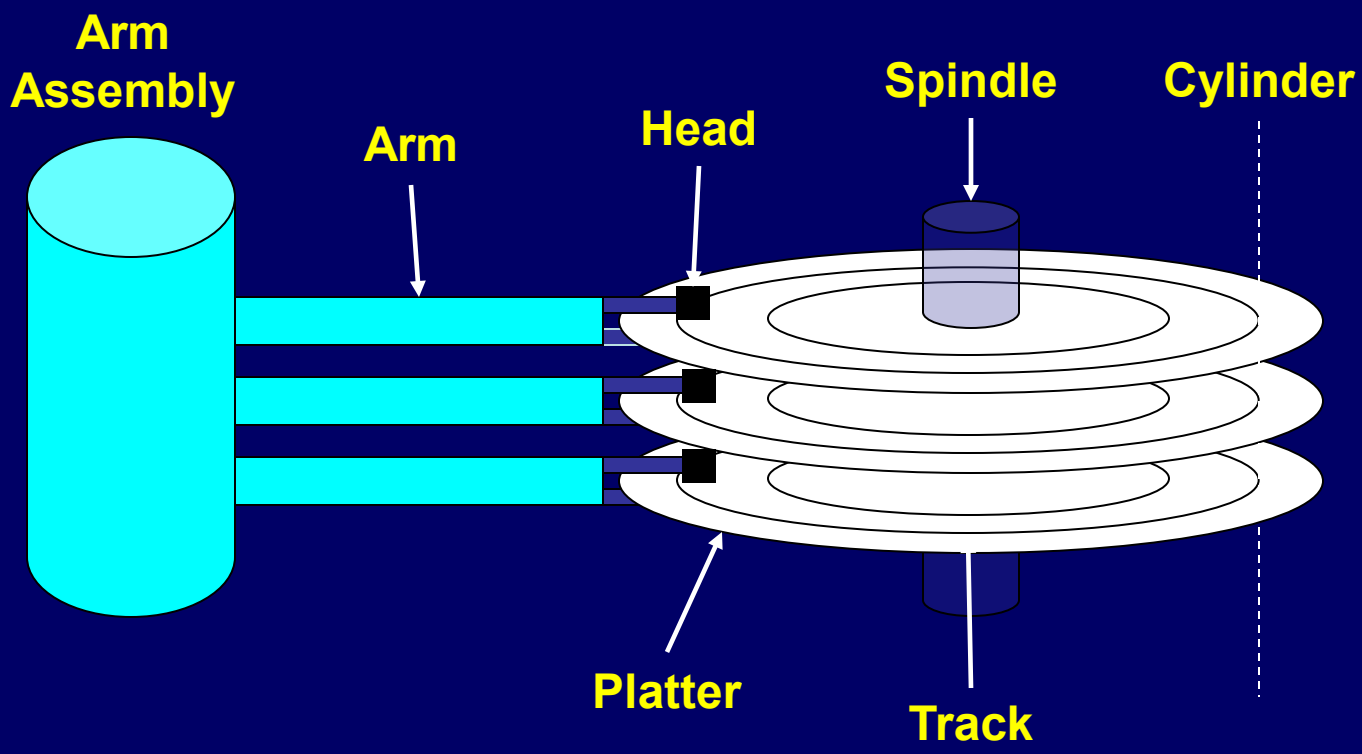
- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!

Hard Disk Drive (HDD) Components

- Rotating disks
- Arm assembly

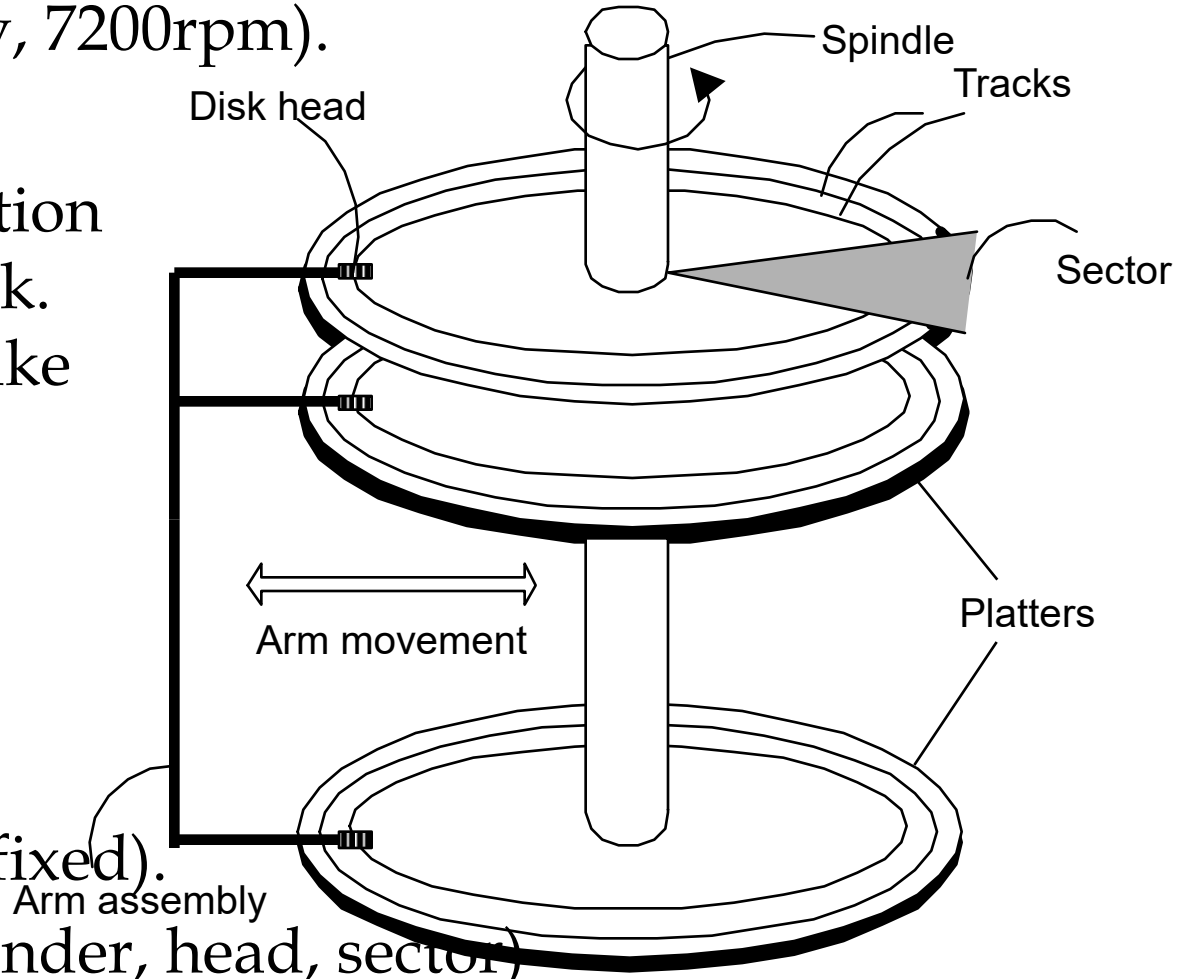
- Disk controller
- Cache
- Interface controller

HDD Organization



Components of a Disk

- The platters spin (say, 7200rpm).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
  - Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed).
- Addressing: CHS (cylinder, head, sector)



HDD Organization

- Typical configurations seen in disks today
  - Platter diameters: 3.7”, 3.3”, 2.6”
  - RPMs: 5400, 7200, 10000, 15000
    - 0.5-1% variation in the RPM during operation
  - Number of platters: 1-5
  - Mobile disks can be as small as 0.75”
- Power proportional to:
  - Tradeoff in the drive-design
- Read/write head
  - Reading – Faraday’s Law
  - Writing – Magnetic Induction
- Data-channel
  - Encoding/decoding of data to/from magnetic phase changes

Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
  - Buffer size (2 MB typical, 8 MB, …)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1 msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *Average seek time -- 9.1 msec*
  - *Average rotational delay -- 4.17 msec*
  - *transfer rate – 13MB/sec*
  - Seek from one track to next – 2.2 msec
  - Max. seek time 15 msec
- Disk access takes about 10 msec whereas accessing memory location takes about 60 nano secs !!
- Memory is more than a Million times faster!!

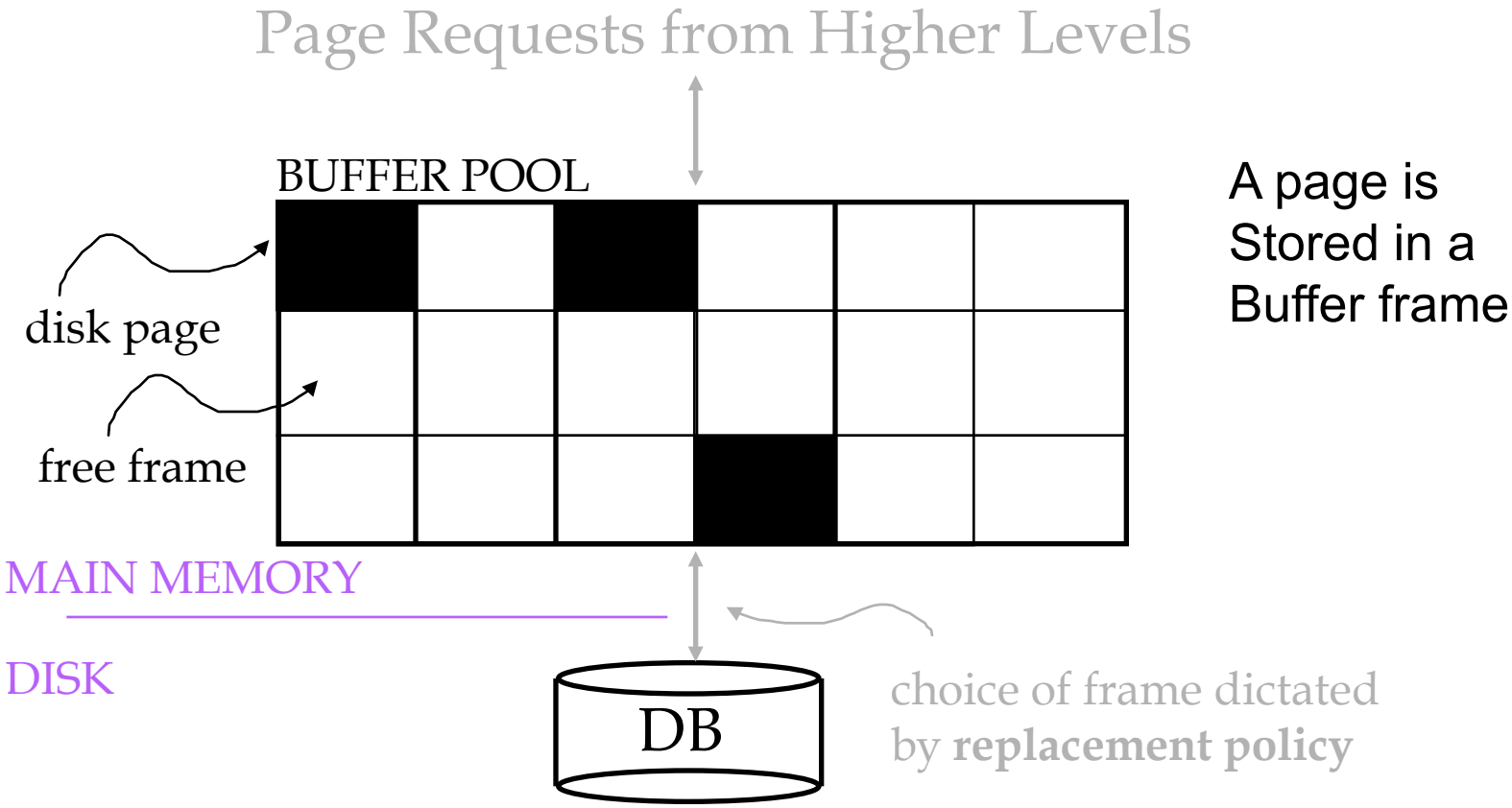
Arranging Pages on Disk

- `*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed.

Buffer Management in a DBMS



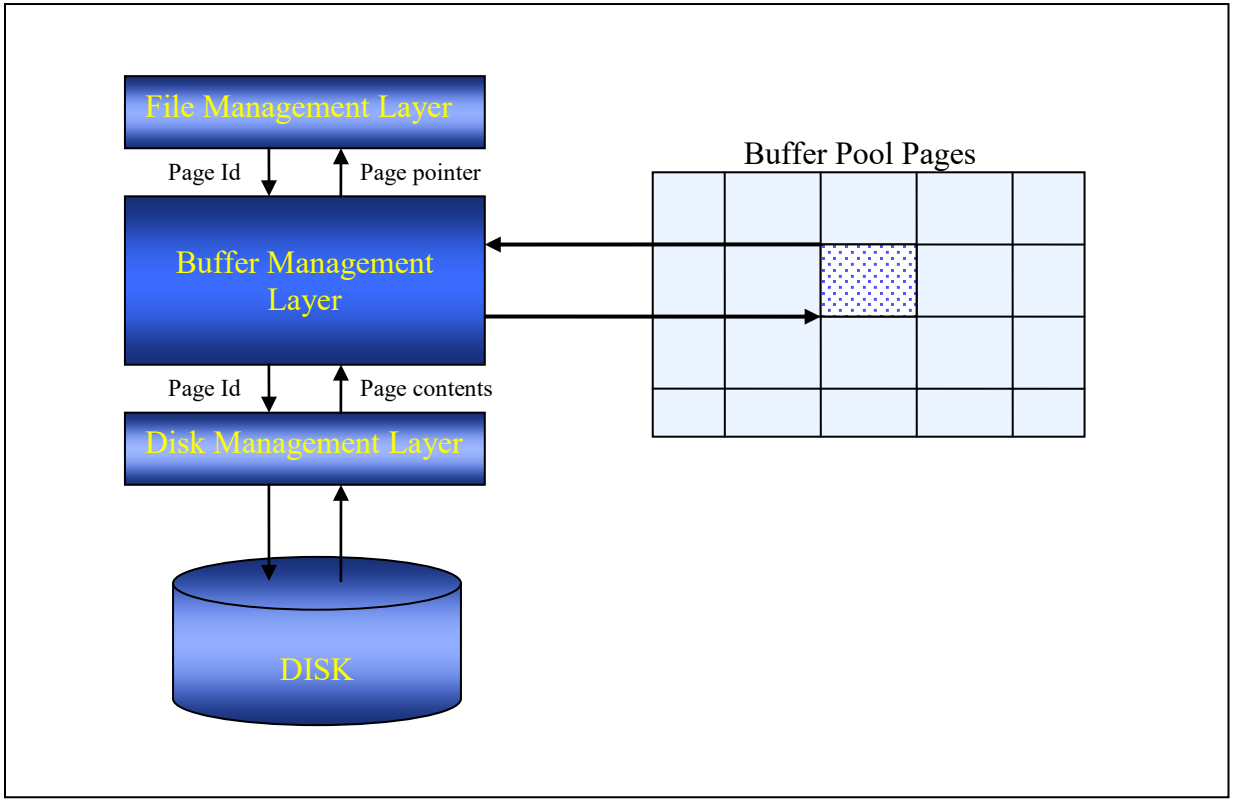
A page is Stored in a Buffer frame

- *Data must be in RAM for DBMS to operate on it!*
- *Mapping of <frame#, pageid> pairs is maintained.*

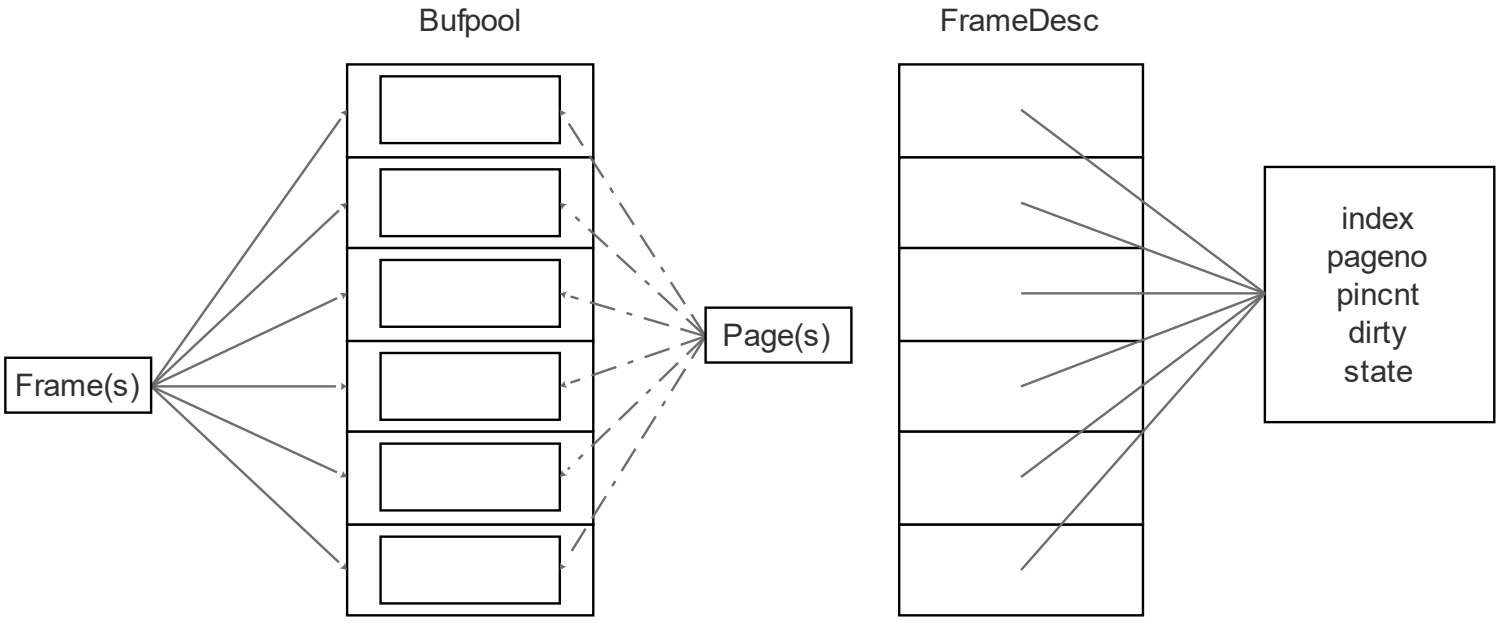
Buffer manager

- The buffer manager reads disk pages into a main memory page as needed.
- The buffer manager is used by access methods, heap files, and relational operators to read / write /allocate / de-allocate pages.
- When a page is requested, the buffer manager brings it in and pins it.
- Replacement policies can be changed at compile time
  - An abstract class Replacer is used for this purpose
  - This class is extended to change replacement policies (e.g., Clock, LRU, MRU)

Buffer manager



Design



- Hashmap (pagemap), maps current page numbers to frames; used for efficient lookups

Buffer Manager Implementation

- Public BufMgr(int numbufs) //constructor
  - Initialize your buffer manager layer with buffer pool, frame table and page map and replacer policy
  - Buffer Pool: The actual array of page objects
  - Frame table: Array of frame descriptors, each containing the pin count and dirty status
  - Page map: Maps current page numbers to frame numbers

Buffer Manager Implementation (2)

- Public pageId newPage(Page firstpg, int run_size):
  - Allocates a set of new pages
  - Try to pin that page in the buffer, if the page is available in some frames
  - You can use allocate_page() function from disk manager layer
- Public void freePage(PageId firstid):
  - This method deallocates a page from disk after removing it from the buffer, if present
  - You can use deallocate_page function from disk manager layer.

Buffer Manager Implementation (3)

- Public void pinPage(PageId pageno, Page page, boolean skipRead):
  - Check if the required page is in the buffer and already pinned, if so , increment pin count of the page
  - Else the page is not buffer pool, read the page from disk using read_page. **Note,** here you may need to use the replacement policy here if there are no free pages. This may involve writing out a replacement candidate out to disk for which you may use write_page (if the dirty bit was set)
  - Skipread has two values
    - PIN_MEMCPY : Copy a given page to buffer pool
    - PIN_DISKIO : Read a disk page to buffer pool

Buffer Manager Implementation (4)

- Public void unpinPage (PageId pageno, boolean dirty)
  - Unpin a page when it is no more needed by decreasing its pin count
  - Check whether the page is in the buffer manager
  - Check whether the page is pinned or not
  - if not, then decrement the pin count, update frame descriptor and notify the replacer

Buffer Manager Implementation (5)

- Public void flushPage(pageId pageno):
  - Writes a page in the buffer pool to disk, if the page is dirty
- Public void flushAllPages():
  - Writes all the dirty pages from the buffer pool to disk
- Public int getNumBuffers():
  - Return the total number of buffer frames
- Public int getNumpinned():
  - Return the total number of unpinned buffer frames

When a Page is Requested ...

- If requested page is not in buffer pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin (increment pin_count)* the page and return its address.
- If there is no frame to choose, then the buffer is full. *When is the buffer full*?

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

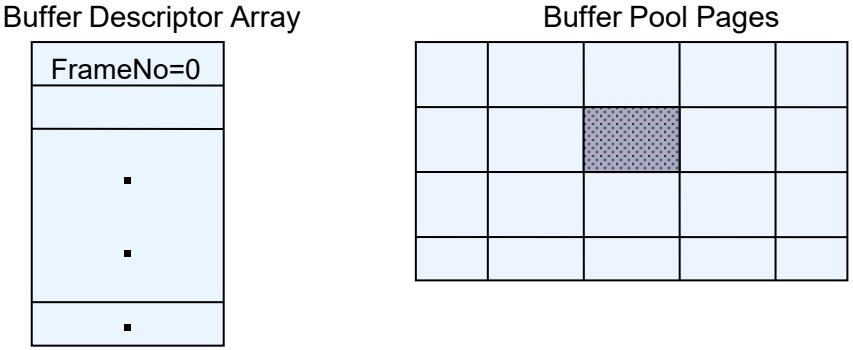
Design

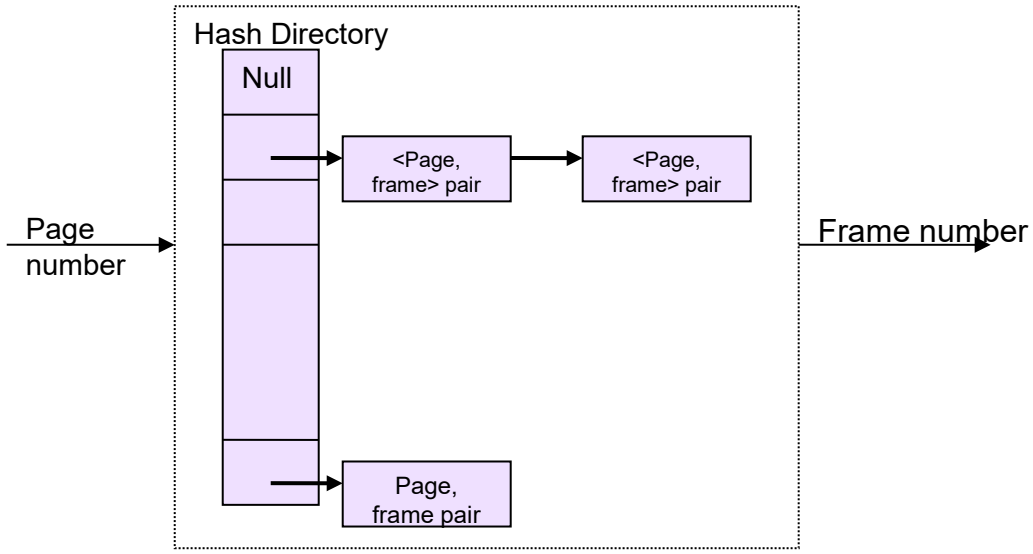
Initially

Page_num = INVALID_ PAGE

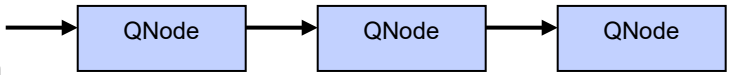
pin_count = 0

Dirty = FALSE





Queue Implementation for LRU, MRU and CLOCK



Data structures used in the buffer manager

Design (Contd.)

The buffer manager maintains a buffer pool to store the pages fetched from the disk. A buffer descriptor table is maintained for each frame in the buffer pool to store the characteristics of that specific frame. The buffer descriptor stores information about whether the specified frame is dirty/clean and its pin count.

The buffer manager implements a hash function to map page numbers to frame numbers. This is needed when the upper layer sends a page number and the buffer manager must resolve which frame that page resides in, if at all the page is in the buffer at that time.

The buffer manager implements functions to pin and unpin pages, create new pages, free exiting pages and flush pages to the disk

More on Buffer Management

- Requestor of page must unpin it, and indicate whether page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later in module 2.)
- The buffer is full when app pin counts of ALL pages are non-zero!
  - What happens then?

Buffer Replacement Policy

- If multiple frames have pin count as zero, which one do you choose?
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, most-recently-used (MRU). FIFO, Random, 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 is less than # pages in file means each page request causes an actual I/O. MRU much better in this situation (but not in all situations, of course).

Project 1

- Each team will be asked to implement one of the 2 replacement policies: LRU, MRU, FIFO, CLOCK, or random
- All of you should be familiar with LRU and MRU (from OS class)
- Clock replacement policy has less overhead
  - Chooses a frame in a circular order (1 to n)
  - *Current* is a clock hand moving across the clock face
  - Each frame has a *reference* bit, turned on when pin count becomes 0
  - Read the rest from the book
- What policies to implement by each team is given in the param file

Buffer Hit Ratios

- A replacement policy is good if requests for a page is already in the buffer pool
- Buffer hit ratio can be computed by keeping track of hits (already in the buffer pool) to the number of times the page is loaded to the buffer pool.
- BHR: count the number of page loads (not considered a hit) and the number of times it was a hit (that is page was already in the buffer pool). Output both numbers and the ratio
- Higher the ratio, better is the replacement policy
- Also, keep track of the loads and hits for each page and output the top 5/10 pages with respect to hits
- As part of project 1, you will implement 2 replacement policies and compute BHR and top 5/10 pages with respect to hits by adding code to the given Buffer manager (BufMgr class)
- You will have to modify the print method
  printBhrAndRefCount in the buffer manager

DBMS vs. OS File (Buffer)

- OS does disk space & buffer management as well. So, why not let OS manage these tasks?
  - Purpose is the same in both. Use files (code mainly) larger than memory available
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.
  - Use multiple replacement policies on differet parts of the buffer pool

DBMS vs. OS File System

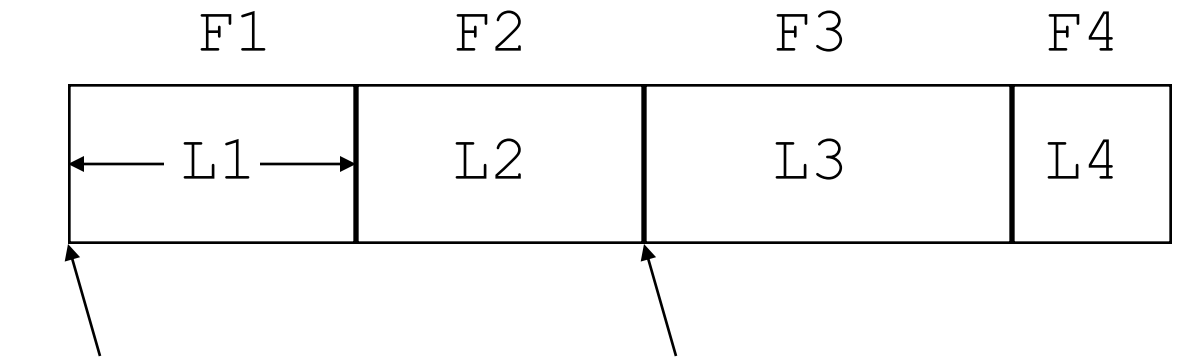
- The above are not provided by the OS
- A DBMS can also predict buffer usage better than OS and do prefetching of pages!

- Other limitations:

- Differences in OS support: portability issues

- Some limitations, e.g., files can't span disks.
- Earlier files sizes was also an issue

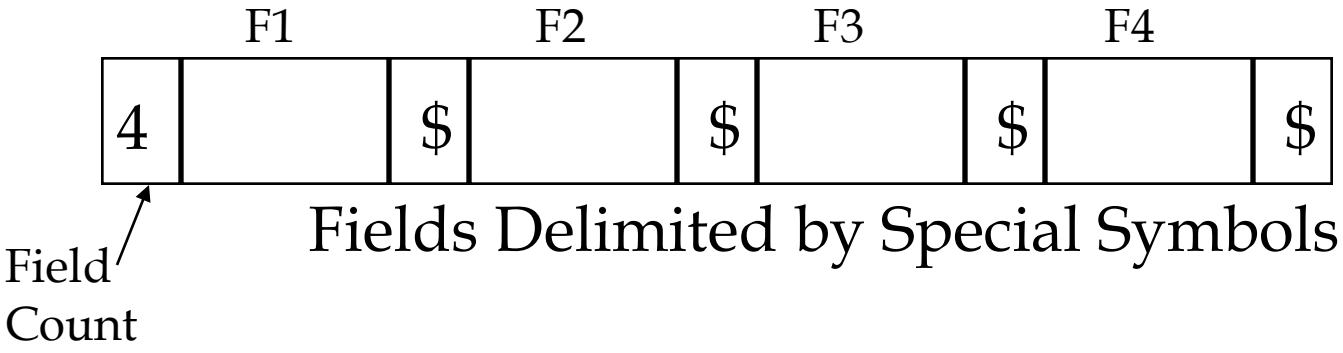
Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*

Record Formats: Variable Length

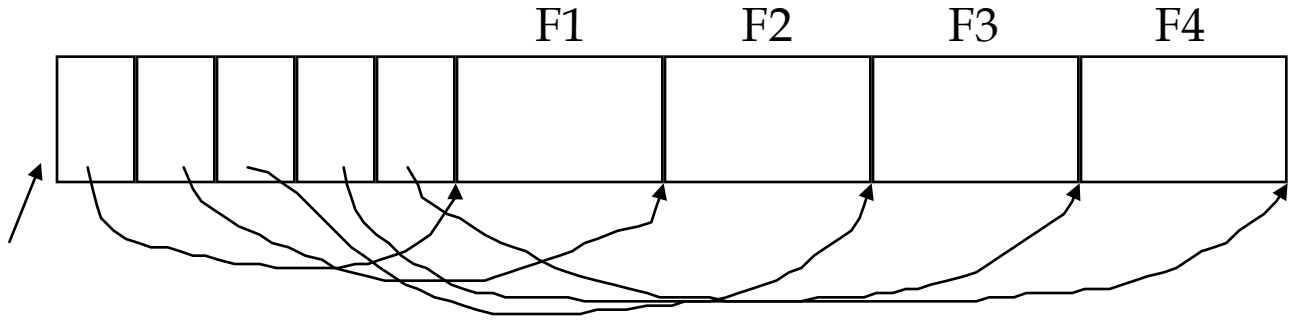
- Two alternative formats (# fields is fixed):



- Finding *i'th* field requires scan of record

Record Formats: Variable Length

☞Two alternative formats (# fields is fixed):



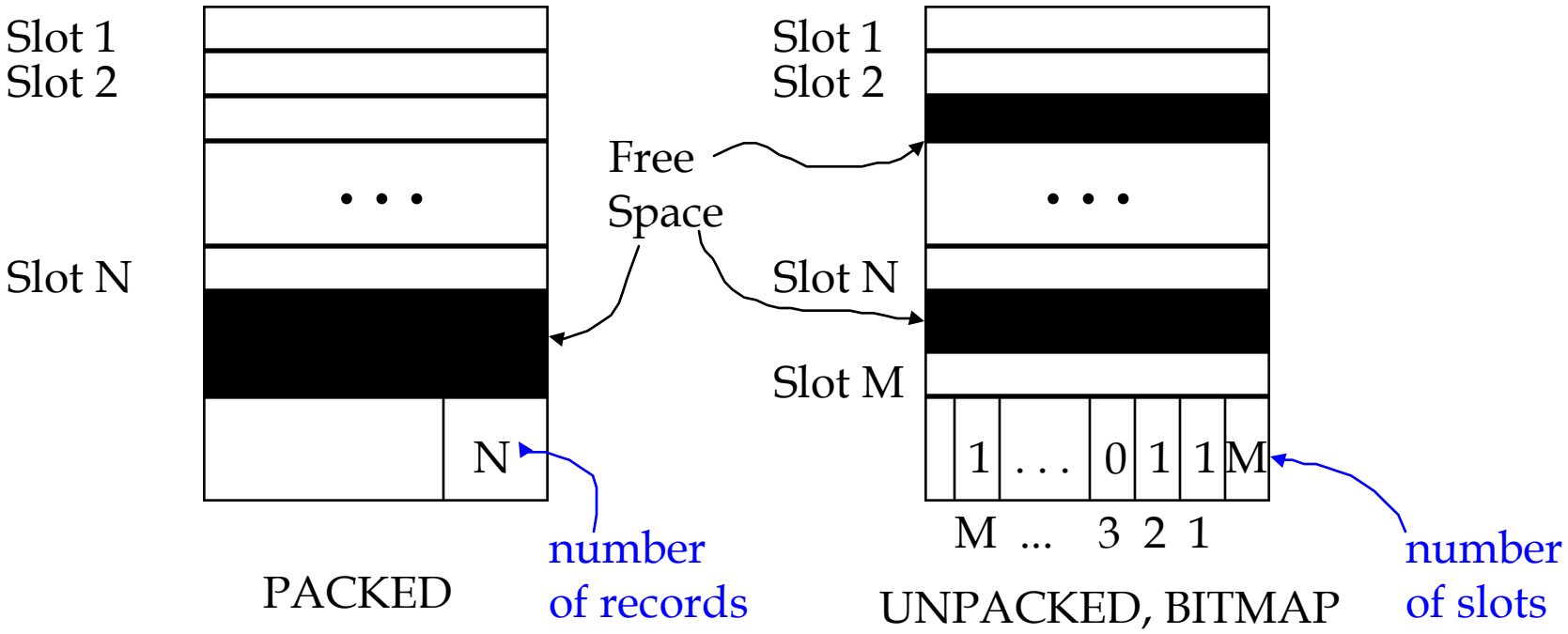
Array of Field Offsets

* Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Subtle Issues: Variable Length

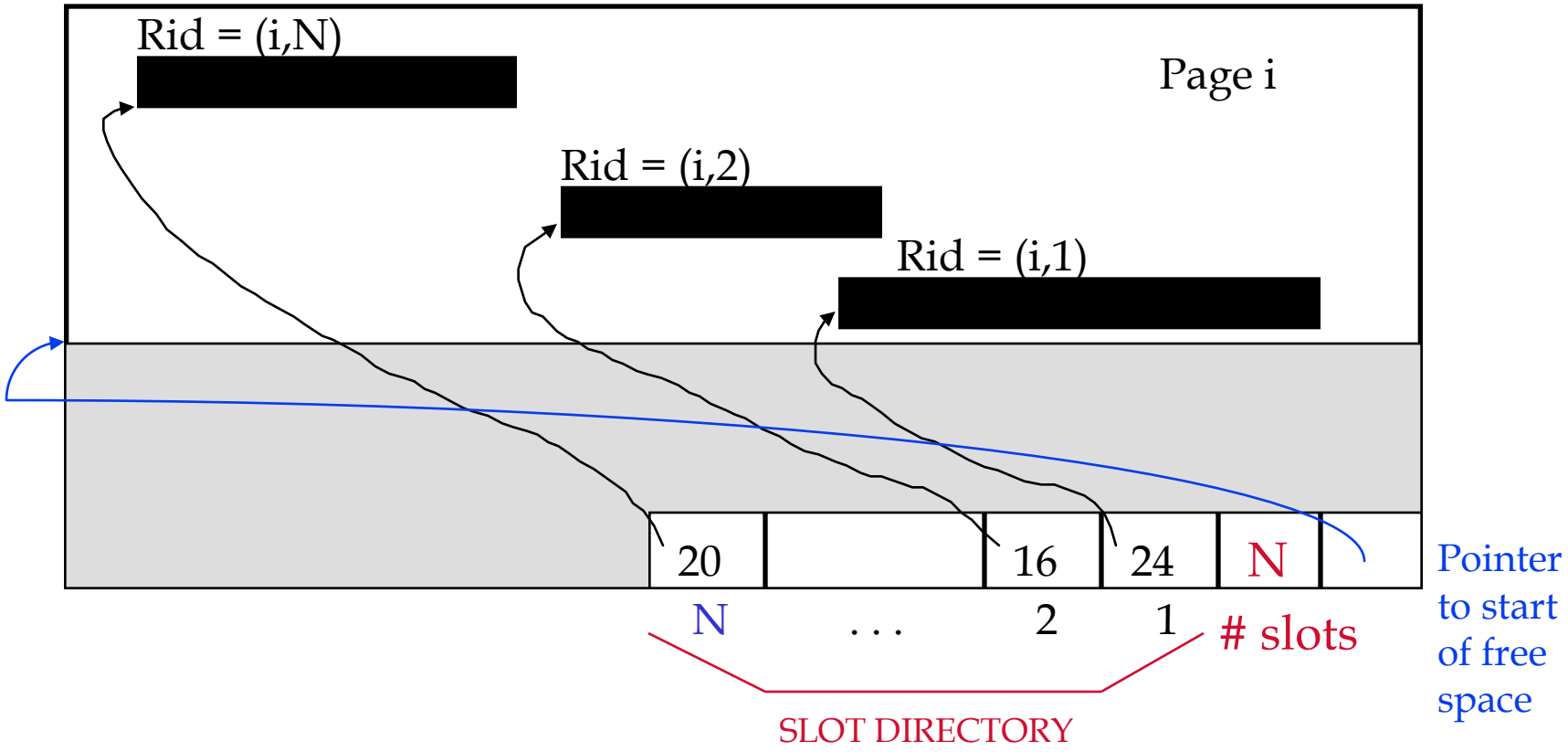
- What about modification? Growth may involve moving fields
- What if a record does not fit into the space remaining on a page? (rids and forwarding address)
- A record grows to occupy more than one page! (chained smaller records) – spanning records

Page Formats: Fixed Length Records (physically numbered slots)



* *Record id (rid) or tuple id (tid) = <page id, slot #>. In the first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records (logically numbered slots)



* *Can move records on page without changing rid; so, attractive for fixed-length records too.*
* Record cannot be removed from the directory! Why?
* Can be compacted without changing rid

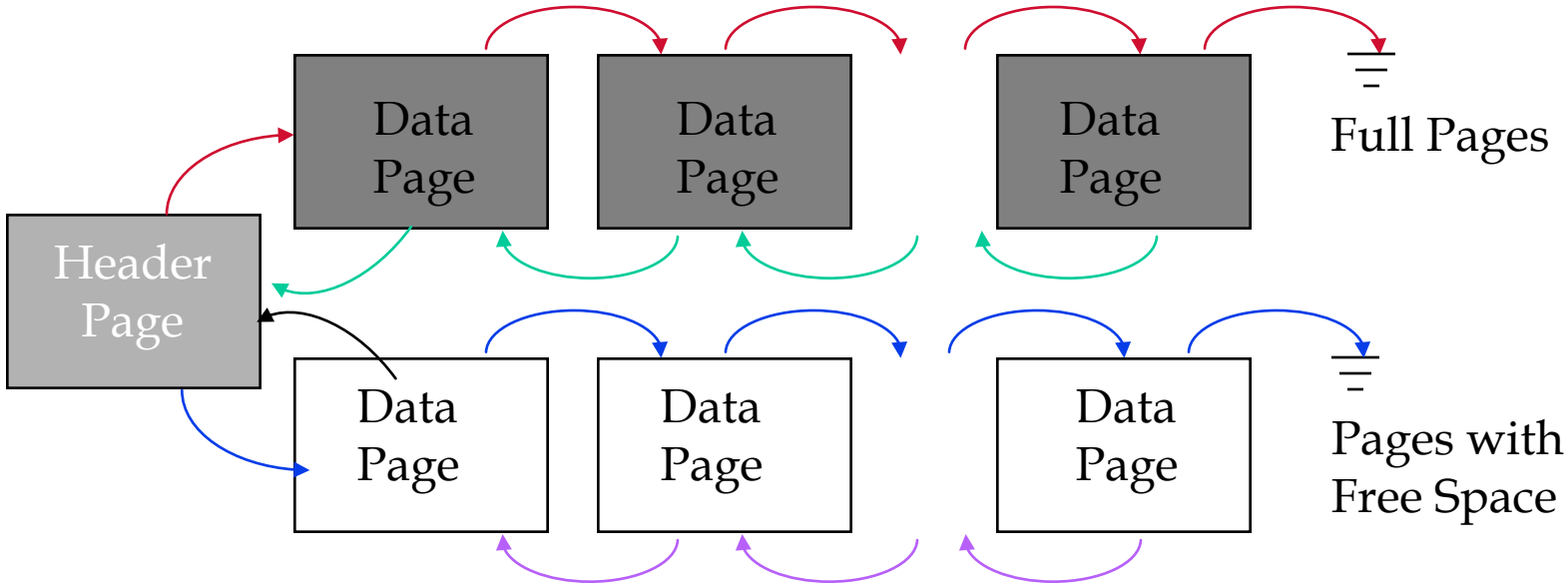
Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify opeartions
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

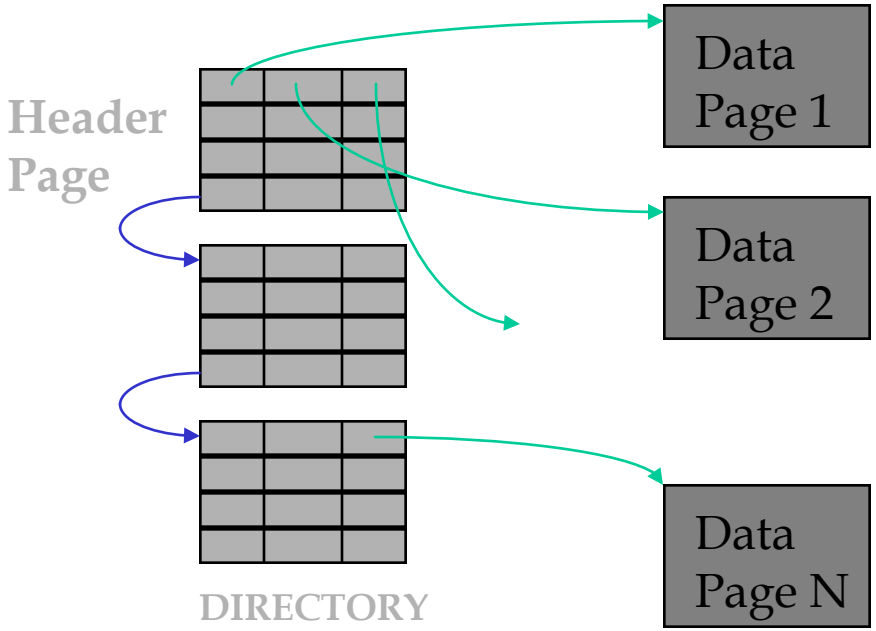
- Simplest file structure contains records in no particular order.
- As a file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `disk pointers' plus data.
- Disadvantage for variable length records! (all pages will be on the free list)

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages!*

Indexes

- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with gpa > 3.0
- Indexes are file structures that enable us to answer such value-based queries efficiently.

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

* *Catalogs are themselves*
* *stored as relations!*

Attr_Cat(attr_name, rel_name, type, position)

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Summary

- Disks provide cheap, non-volatile storage.
  - Random access, but cost depends on location of page on disk; important to arrange data sequentially to minimize *seek* and *rotation* delays.
- Buffer manager brings pages into RAM.
  - Page stays in RAM until released by requestor.
  - Written to disk when frame chosen for replacement (which is sometime after requestor releases the page).
  - Choice of frame to replace based on *replacement policy (LRU, MRU, FIFO, CLOCK. …).*
  - Tries to *pre-fetch* several pages at a time.

Summary (Contd.)

- DBMS vs. OS File Support
  - DBMS needs features not found in many OSs, e.g., forcing a page to disk, controlling the order of page writes to disk, files spanning disks, ability to control pre-fetching and page replacement policy based on predictable access patterns, etc.
- Variable length record format with field offset directory offers support for direct access to i'th field and null values.
- Slotted page format supports variable length records and allows records to move on page.

Summary (Contd.)

- File layer keeps track of pages in a file, and supports abstraction of a collection of records.
  - Pages with free space identified using linked list or directory structure (similar to how pages in file are kept track of).
- Indexes support efficient retrieval of records based on the values in some fields.
- Catalog relations store information about relations, indexes and views. (*Information that is common to all records in a given collection.*)

Thank You !