CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu

Quick Announcements

- Posted notes on grading breakdown
- Videos and reading homework for next week will be posted today
- Will experiment with Zoom and/or Panopto quizzes to increase participation and interaction
- Feel free to send questions through Chat or Raise Hand

Spring 2020 – Online Instruction Plan

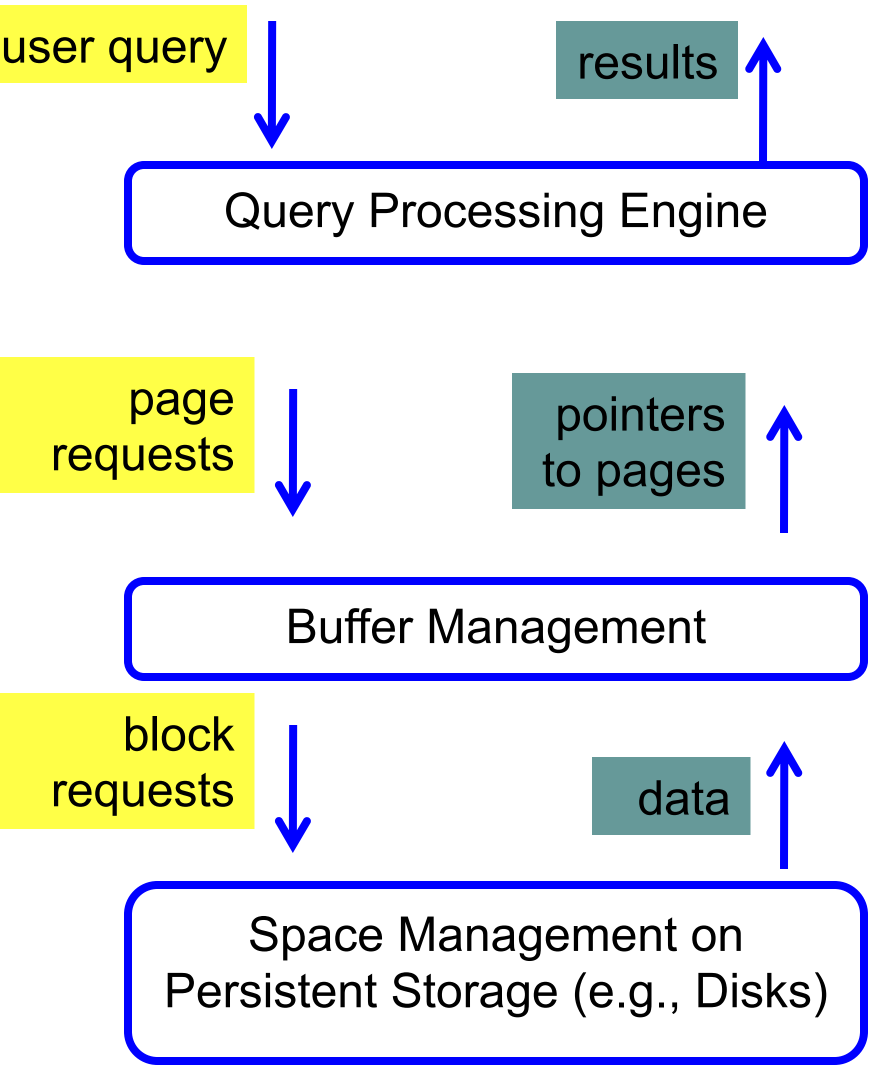
- Week 1 (March 30 – April 2):
  - File Organization and Overview of Indexes
  - B+-Trees
  - Hashing
  - Miscellaneous topics in Indexes
- Week 2: Query Processing
- Week 3: Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

Spring 2020 – Online Instruction Plan

- Reading homeworks based on the videos and book chapters
- Virtual Zoom/Webex Sessions during class time
  - Except March 30
- Tentative schedule below
  - Still trying to figure out the “Final” and overall grading breakdown

Reading Homeworks Due		Final		Projects Due			
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
March	30	31	1	2	3	4	5
April	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
	27	28	29	30	1	2	3
May	4	5	6	7	8	9	10
	11	12	13	14	15	16	17
	18						

Review: Query Processing/Storage

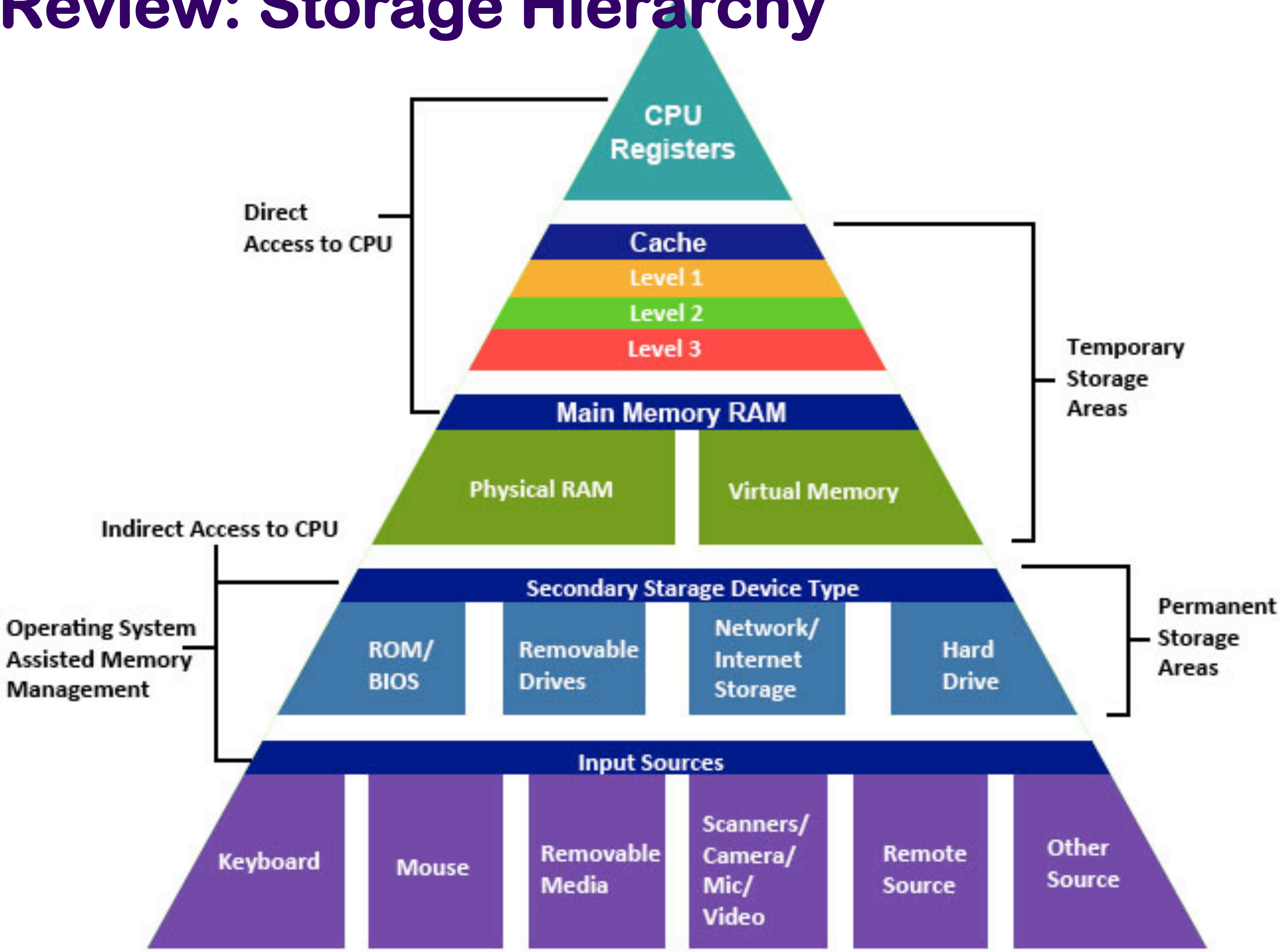


- Given a input user query, decide how to “execute” it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

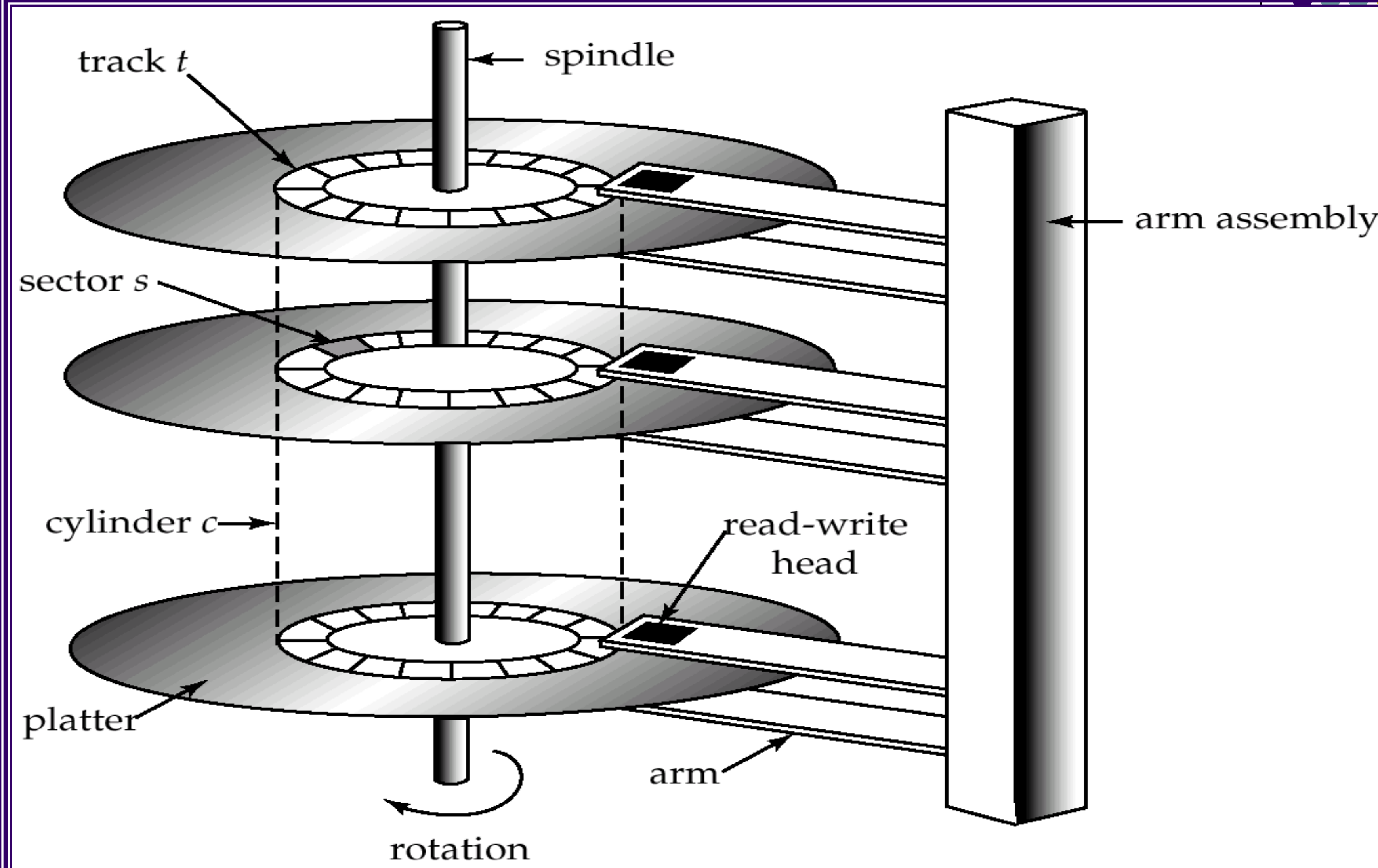
- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Review: Storage Hierarchy



source: http://cse1.net/recaps/4-memory.html

Review: Disks



We focus on "disks" for the rest of the semester, but everything applies to SSDs as well.

File Organization & Indexes Overview

- Book Chapters
  - 10.5, 10.6, 11.1, 11.2
- Key topics:
  - What are different ways the tuples mapped to disk blocks?
  - What are the pros and cons of the different approaches to map tuples to blocks?
  - How an “index” helps efficiently find tuples that satisfy a condition?
  - What are key characteristics of indexes?

Mapping Tuples to Disk Blocks

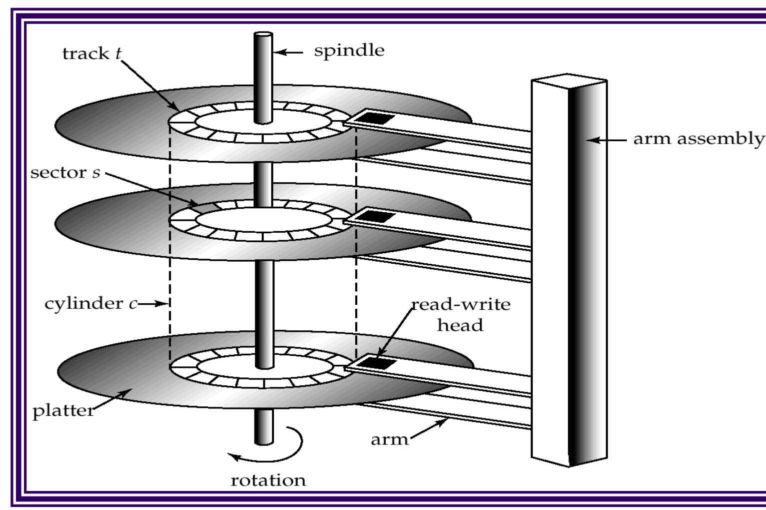
ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
838[illegible]	[illegible]	[illegible]	[illegible]	[illegible]aylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

- Very important implications on performance
- Quite a few different ways to do this

?



File Organization

- Requirements and Performance Goals:
  - Allow insertion/deletions of tuples/records in relations
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
  - Fetch all tuples from a specific relation (scans)
    - Faster if they are all sequential/in contiguous blocks
  - Allow building of "indexes"
    - Auxiliary data structures maintained on disks and in memory for faster retrieval
  - And so on…

File System or Not

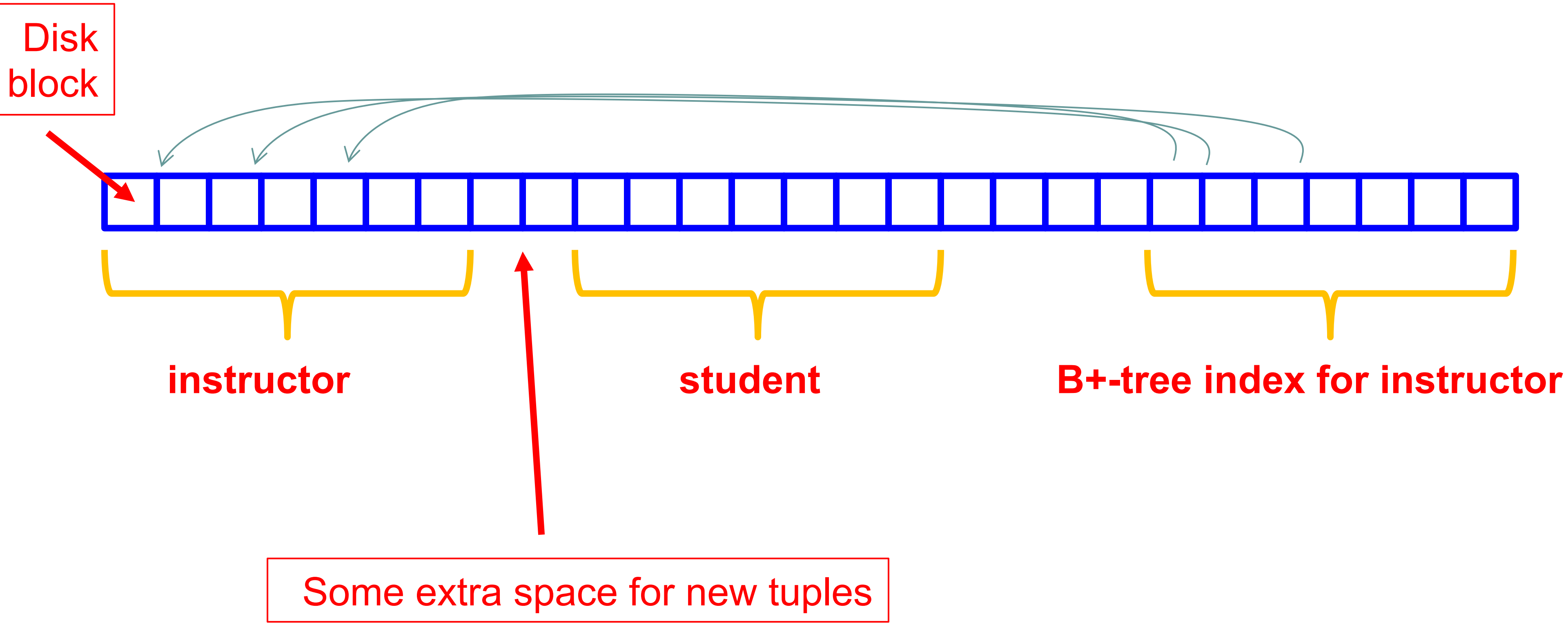
- Option 1: Use OS File System
  - File systems are a standard abstraction provided by Operating Systems (OS) for managing data
  - Major Con: Databases don't have as much control over the physical placement any more --- OS controls that
    - E.g., Say DBMS maps a relation to a "file"
    - No guarantee that the file will be "continguous" on the disk
    - OS may spread it across the disk, and won't even tell the DBMS
- Option 2: DBMS directly works with the disk or uses a lightweight/custom OS
  - Increasingly uncommon – most DBMSs today run on top of OSes (e.g., PostgreSQL on your laptop, or on linux VMs in the cloud, or on a distributed HDFS)

Through a File System

- Option 1: Allocate a single "file" on the disk, and treat it as a contiguous sequence of blocks
  - This is what PostgreSQL does
  - The blocks may not actually be contiguous on disk
- Option 2: A different file per relation
  - Some of the simpler DBMS use this approach
- Either way: we have a set of relations mapped to a set of blocks on disk

Assumptions for Now

- Each relation stored separately on a separate set of blocks
  - Assumed to be contiguous
- Each “index” maintained in a separate set of blocks
  - Assumed to be contiguous



Within block: Fixed Length Records

- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: Rearrange
  - Option 2: Keep a *free list* and use for next insert

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

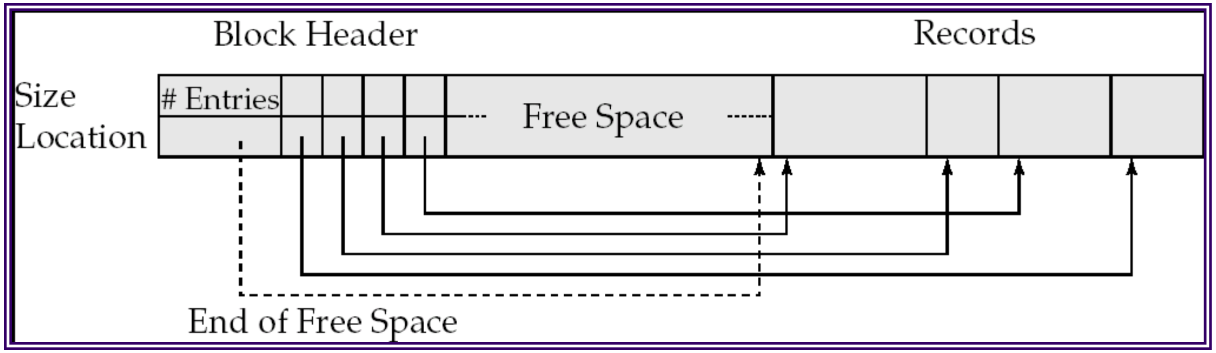
Within block: Fixed Length Records

- Deleting: using "free lists"

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
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

Within block: Variable-length Records

Slotted page/block structure



- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

Across Blocks of a Relation

- Which block should a record go to ?
  - Anywhere ?
    - How to search for “SSN = 123” ?
    - Called “heap” organization
  - Sorted by SSN ?
    - Called “sequential” organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a “hash” key
    - Called “hashing” organization
    - Store the record with SSN = x in the block number x%1000
    - Why ?

Sequential File Organization

- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while

Sequential File Organization

- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account information
    - < 4 requests satisfied per second

Index

- A data structure for efficient search through large databaess
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes


EXIT
FACTS ON

Ordered Indexes

- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation

Index

10101	
32343	
76766	

Relation

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Primary *dense* Index

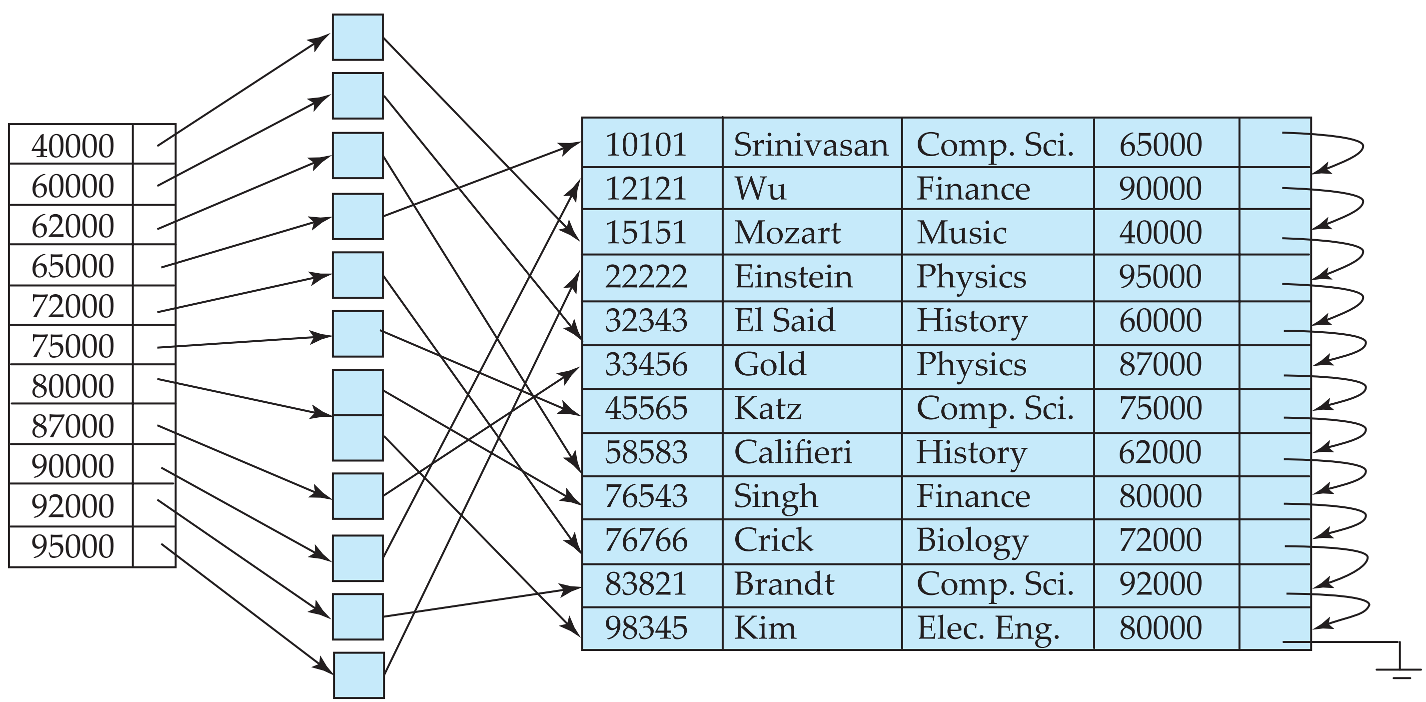
- Every key must appear in the index
- Index becomes pretty large, but can often avoid having to go to the relation
  - E.g., select * from instructor where ID = 10000
    - Not found in the index, so can return immediately

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

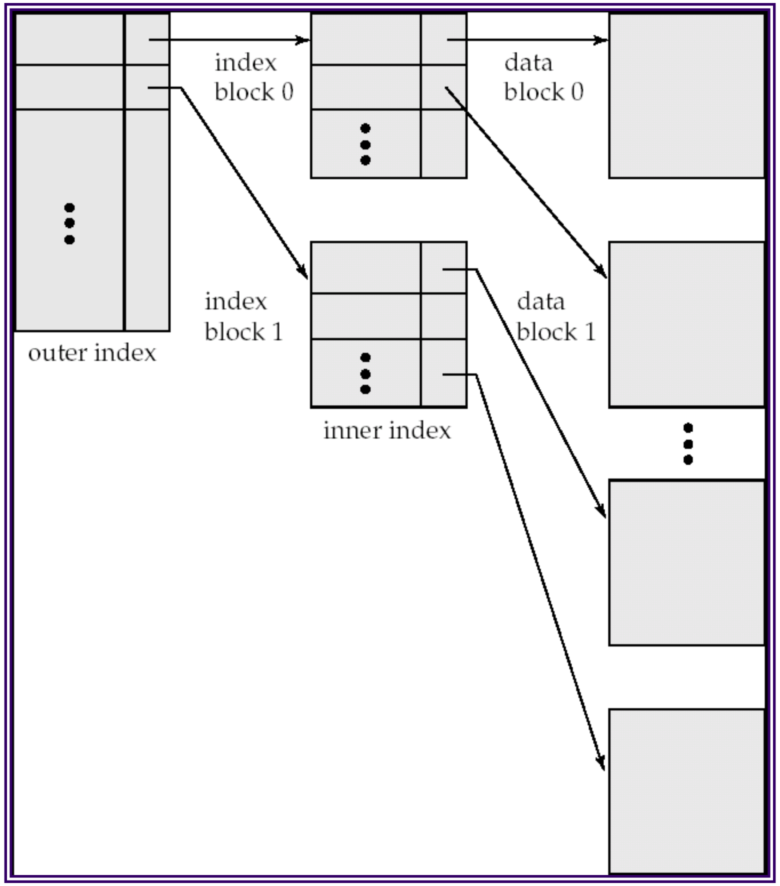
Secondary Index

- Relation sorted on *ID*
- But we want an index on *salary*
- Must be dense
  - Every search key must appear in the index



Multi-level Indexes

- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



Multi-level Indexes

- How do you search through a multi-level index ?

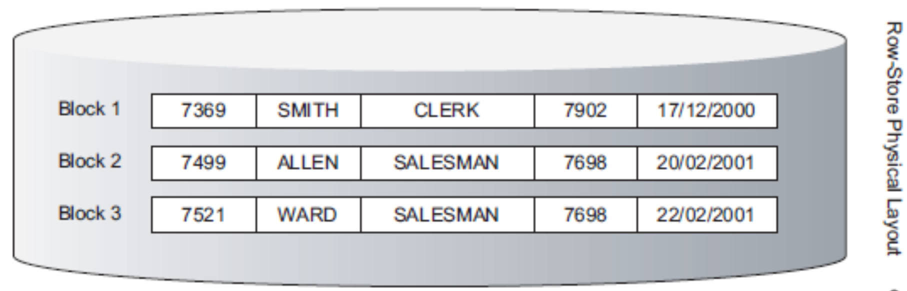
- What about keeping the index up-to-date ?
  - Tuple insertions and deletions
    - This is a static structure
    - Need overflow pages to deal with insertions
  - Works well if no inserts/deletes
  - Not so good when inserts and deletes are common

Next

- Different ways to build more efficient indexes
  - B+-Tree indexes
  - Hashing-based indexes

Advanced Topics

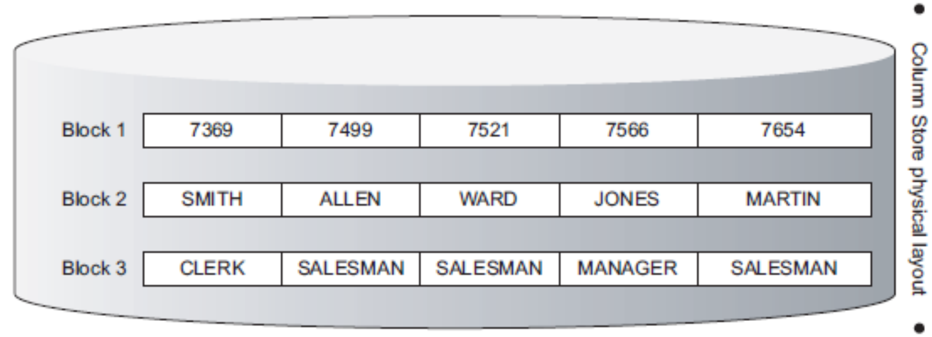
- Row vs columnar representation:
  - We are largely focused on row representation
  - Column-based organization much more efficient for queries
    - But are not as efficient to update
  - Used by most modern warehouses



Row Database stores row values together

Logical Schema

EmpNo	EName	Job	Mgr	HireDate
7369	SMITH	CLERK	7902	17/12/1980
7499	ALLEN	SALESMAN	7698	20/02/1981
7521	WARD	SALESMAN	7698	22/02/1981
7566	JONES	MANAGER	7839	2/04/1981
7654	MARTIN	SALESMAN	7698	28/09/1981
7698	BLAKE	MANAGER	7839	1/05/1981
7782	CLARK	MANAGER	7839	9/06/1981



Column Database stores column values together

Advanced Topics

- Data Storage Formats used in ”big data” world
  - Parquet, Avro, and many others

- Sophisticated on-disk and in-memory representations for maintaining very large volumes of data as ”files”
  - That can be emailed, shared, interpreted by many different programs

- Typically tend to be ”column-oriented”
  - Are not designed to be easy to update (by and large)

- Lot of work in recent years on this