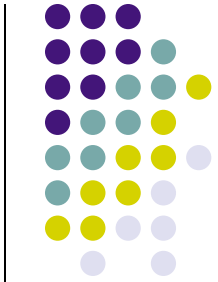
CMSC424: Storage and Indexes

Instructor: Amol Deshpande
amol@cs.umd.edu

Today's Class

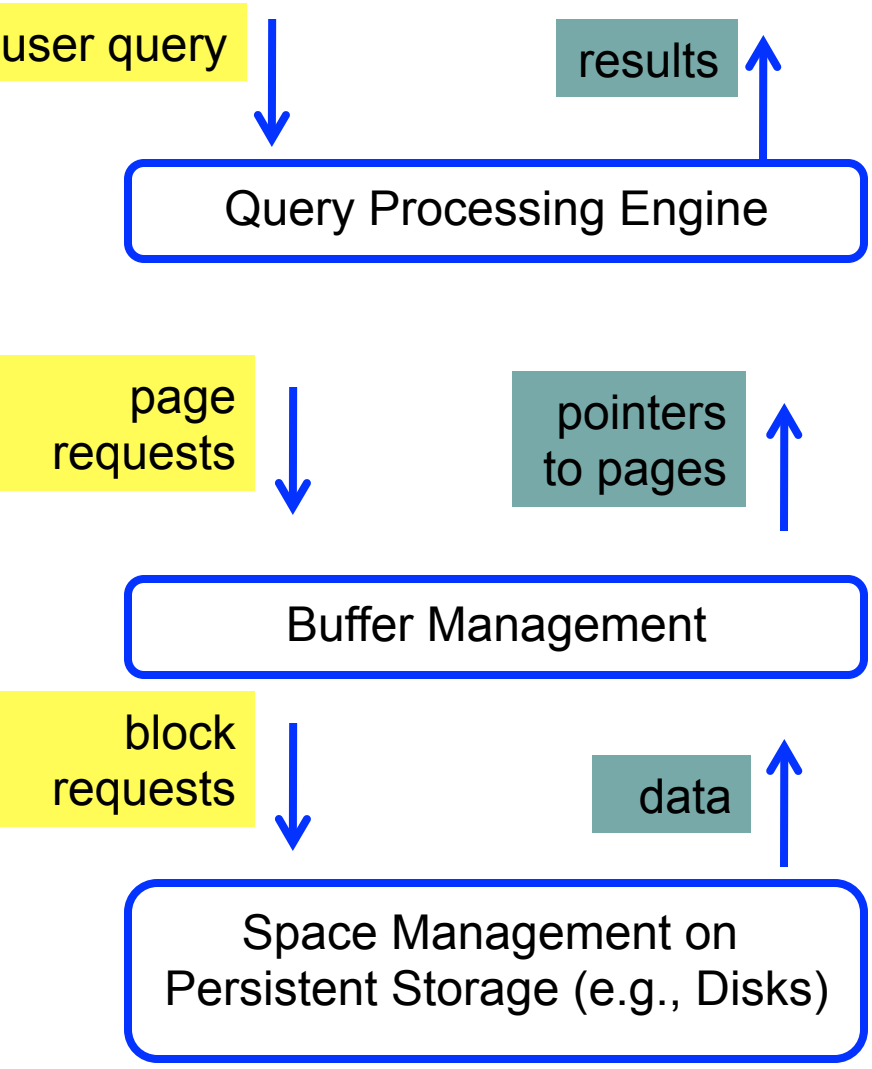
- Storage and Query Processing
  - Record storage; Indexes
- Other things
  - ELMS Dummy Assignment
    - Upload a PDF
  - Project 3: due this Friday
    - Make sure to go through the Notebook on EXPLAIN
  - No laptop use in class (without permission) !!

Databases



- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

Query Processing/Storage

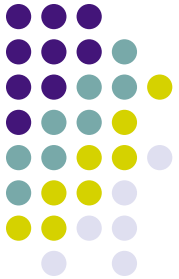


- Given a input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

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

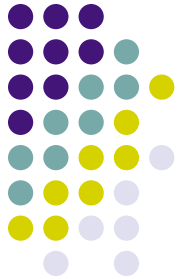
- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Outline



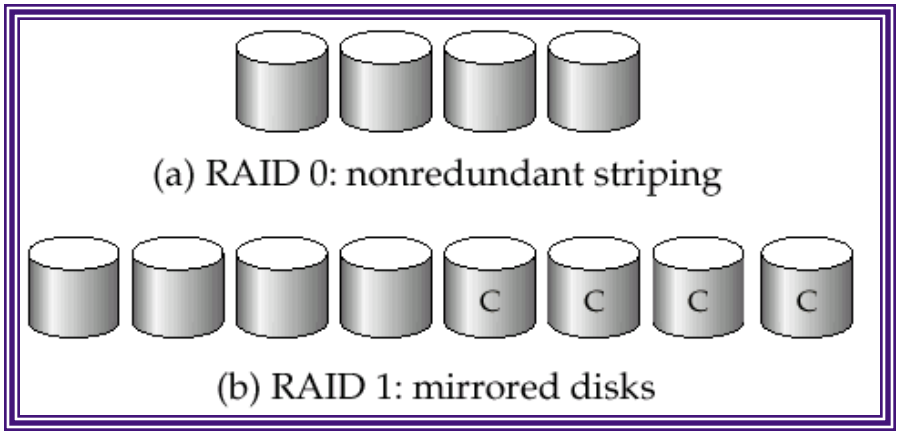
- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc.…

RAID



- Redundant array of independent disks
- Goal:
  - Disks are very cheap
  - Failures are very costly
  - Use “extra” disks to ensure reliability
    - If one disk goes down, the data still survives
  - Also allows faster access to data
- Many raid “levels”
  - Different reliability and performance properties

RAID Levels



(a) No redundancy.

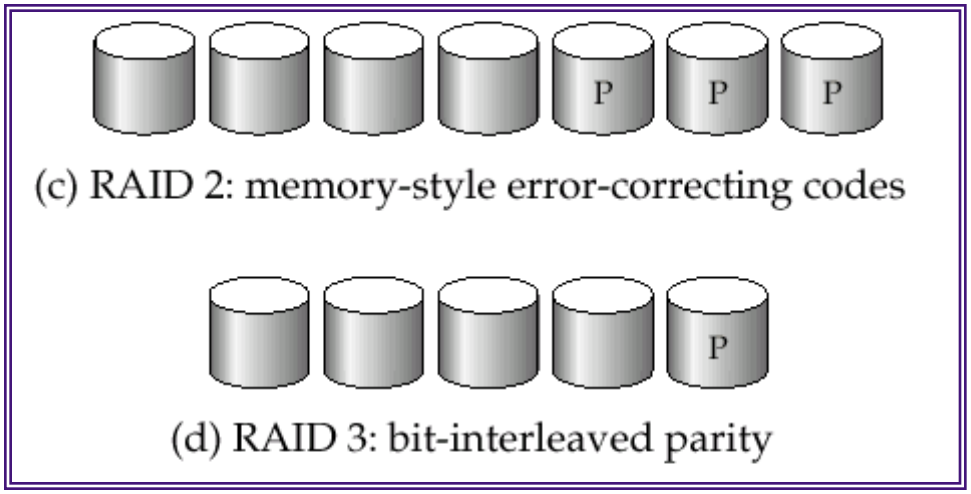
(b) Make a copy of the disks.

If one disk goes down, we have a copy.

Reads: Can go to either disk, so higher data rate possible.

Writes: Need to write to both disks.

RAID Levels



(c) Memory-style Error Correcting

Keep extra bits around so we can reconstruct.

Superceeded by below.

(d) One disk contains “parity” for the main data disks.

Can handle a single disk failure.

Little overhead (only 25% in the above case).

RAID Level 5

- Distributed parity “blocks” instead of bits
- Subsumes Level 4
- Normal operation:
  - “Read” directly from the disk. Uses all 5 disks
  - “Write”: Need to read and update the parity block
    - To update 9 to 9’
      - read 9 and P2
      - compute P2’ = P2 *xor* 9 *xor* 9’
      - write 9’ and P2’

(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

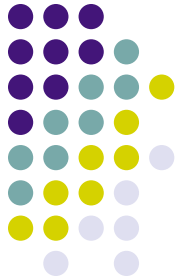
RAID Level 5

- Failure operation (disk 3 has failed)
  - "Read block 0": Read it directly from disk 2
  - "Read block 1" (which is on disk 3)
    - Read P0, 0, 2, 3 and compute 1 = P0 *xor* 0 *xor* 2 *xor* 3
  - "Write":
    - To update 9 to 9'
      - read 9 and P2
        - Oh… P2 is on disk 3
        - So no need to update it
      - Write 9'

(f) RAID 5: block-interleaved distributed parity

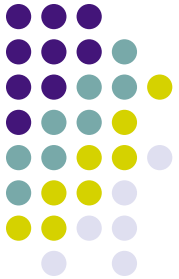
P0	0		2	3
4	P1		6	7
8	9		10	11
12	13		P3	15
16	17		19	P4

Choosing a RAID level



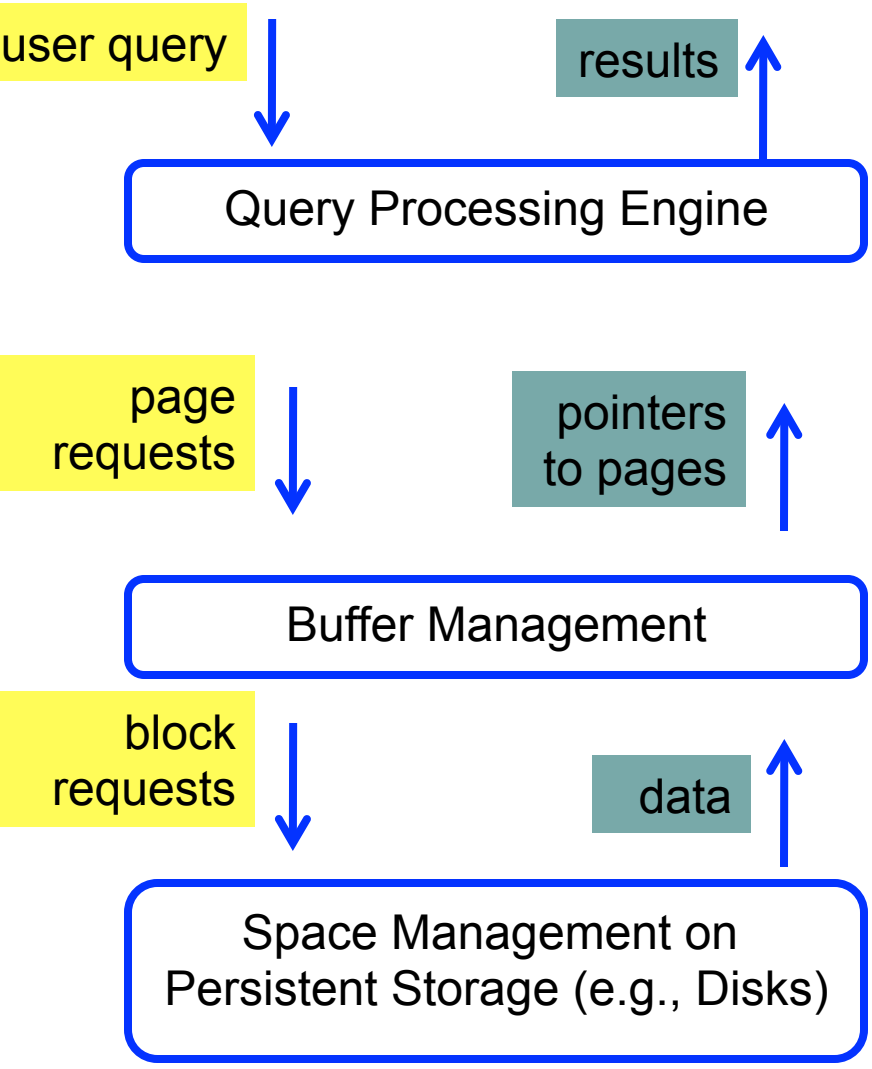
- Main choice between RAID 1 and RAID 5
- Level 1 better write performance than level 5
  - Level 5: 2 block reads and 2 block writes to write a single block
  - Level 1: only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 5 lower storage cost
  - Level 1 60% more disks
  - Level 5 is preferred for applications with low update rate, and large amounts of data

Outline



- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes…

Query Processing/Storage

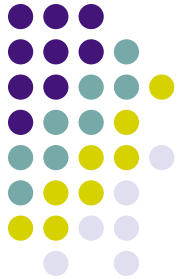


- Given a input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

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

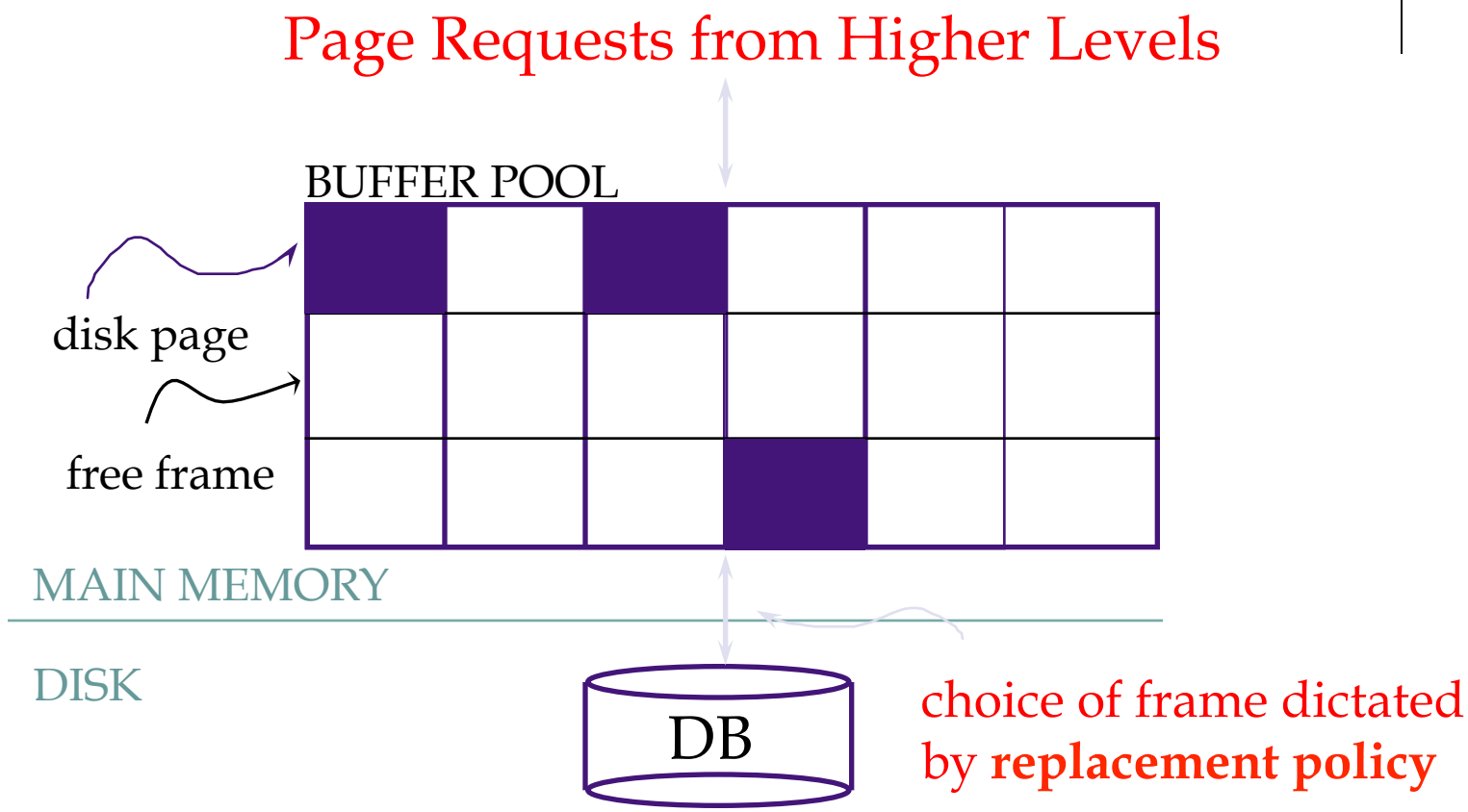
- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Buffer Manager

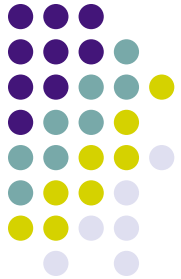


- When the QP wants a block, it asks the “buffer manager”
  - The block must be in memory to operate upon
- Buffer manager:
  - If block already in memory: return a pointer to it
  - If not:
    - Evict a current page
      - Either write it to temporary storage,
      - or write it back to its original location,
      - or just throw it away (if it was read from disk, and not modified)
    - and make a request to the storage subsystem to fetch it

Buffer Manager

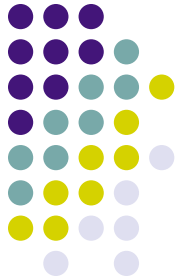


Buffer Manager



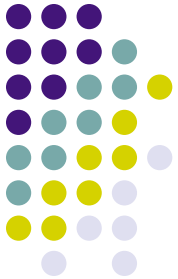
- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - Why ?
  - Clock ?
    - An efficient implementation of LRU

Buffer Manager



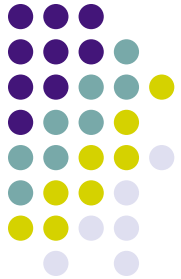
- *Pinning* a block
  - Not allowed to write back to the disk
- *Force-output (force-write)*
  - Force the contents of a block to be written to disk
- *Order the writes*
  - This block must be written to disk before this block
- Critical for fault tolerant guarantees
  - Otherwise the database has no control over whats on disk and whats not on disk

Outline



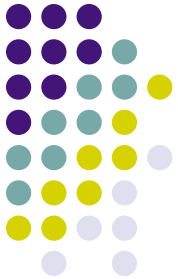
- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc.…

File Organization



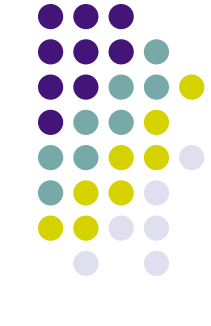
- How are the relations mapped to the disk blocks ?
  - Use a standard file system ?
    - High-end systems have their own OS/file systems
    - OS interferes more than helps in many cases
  - Mapping of relations to file ?
    - One-to-one ?
    - Advantages in storing multiple relations clustered together
  - A *file* is essentially a *collection of disk blocks*
    - How are the tuples mapped to the disk blocks ?
    - How are they stored within each block

File Organization



- Goals:
  - Allow insertion/deletions of tuples/records
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
- Simplest case
  - Each relation is mapped to a file
  - A file contains a sequence of records
  - Each record corresponds to a logical tuple
- Next:
  - How are tuples/records stored within a block ?

Fixed Length Records



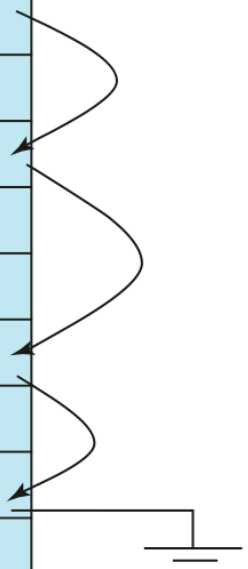
- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: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

Fixed Length Records

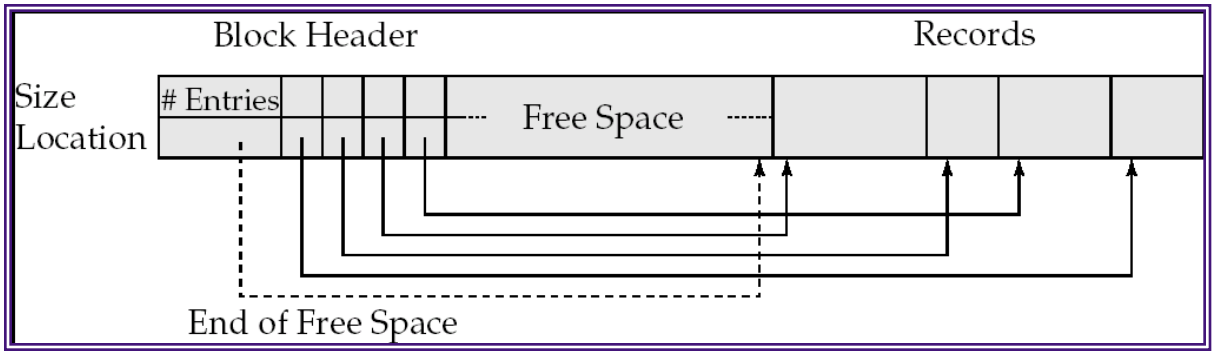
- Deleting: using “free lists”

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



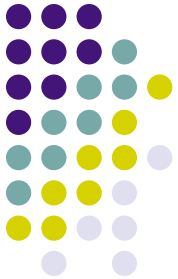
Variable-length Records

Slotted page structure



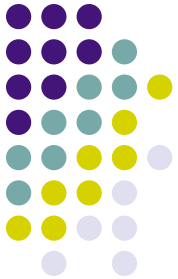
- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

File Organization



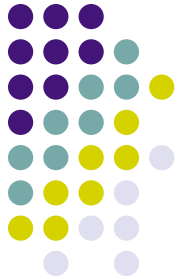
- Which block of a file should a record go to ?
  - Anywhere ?
    - How to search for “SSN = 123” ?
    - Called “heap” organization
  - Sorted by SSN ?
    - Called “sequential” organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a “hash” key
    - Called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

Sequential File Organization



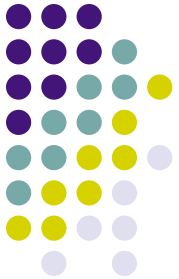
- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become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

Sequential File Organization



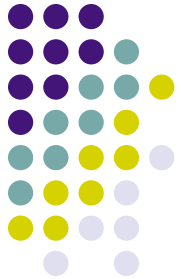
- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account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

Outline



- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- Etc…

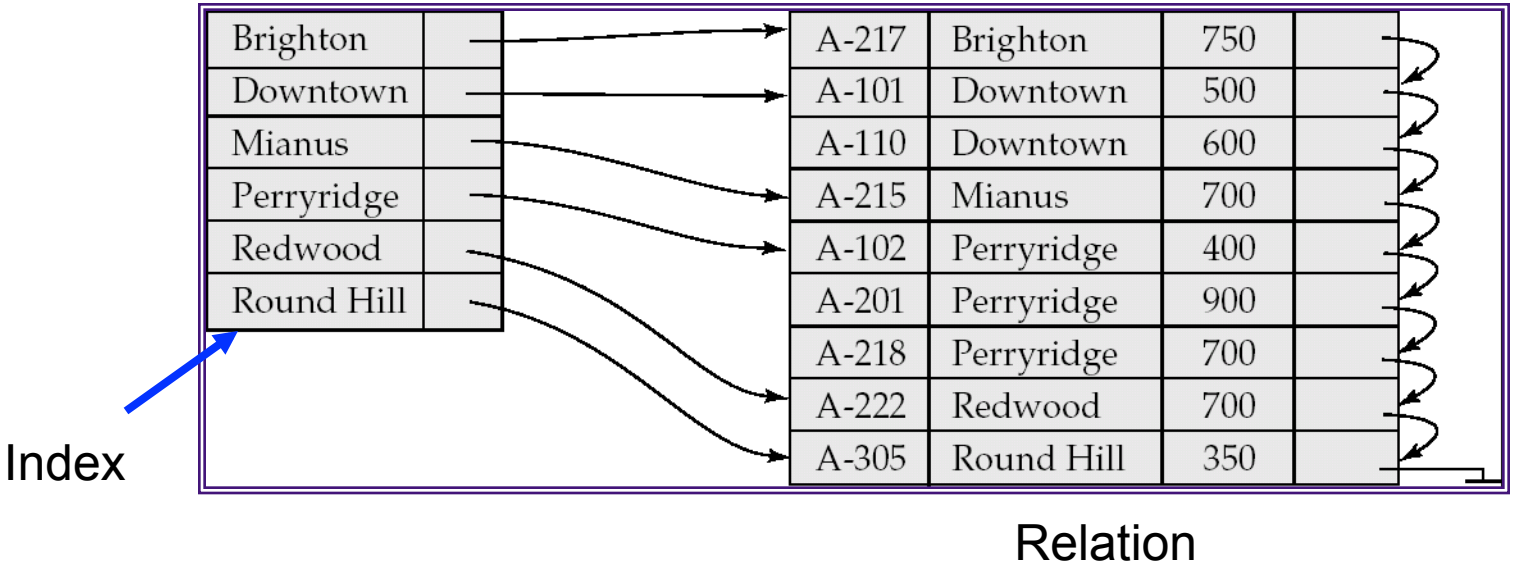
Index



- 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

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



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