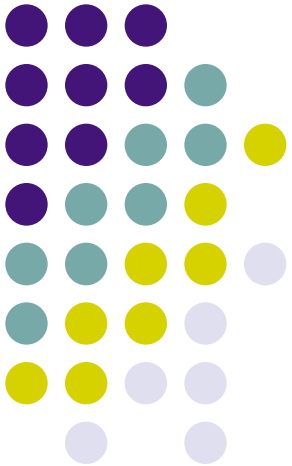
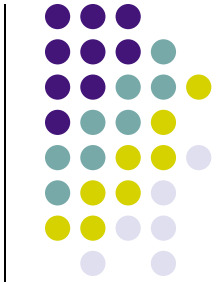
CMSC424: Database Design

Instructor: Amol Deshpande

amol@cs.umd.edu

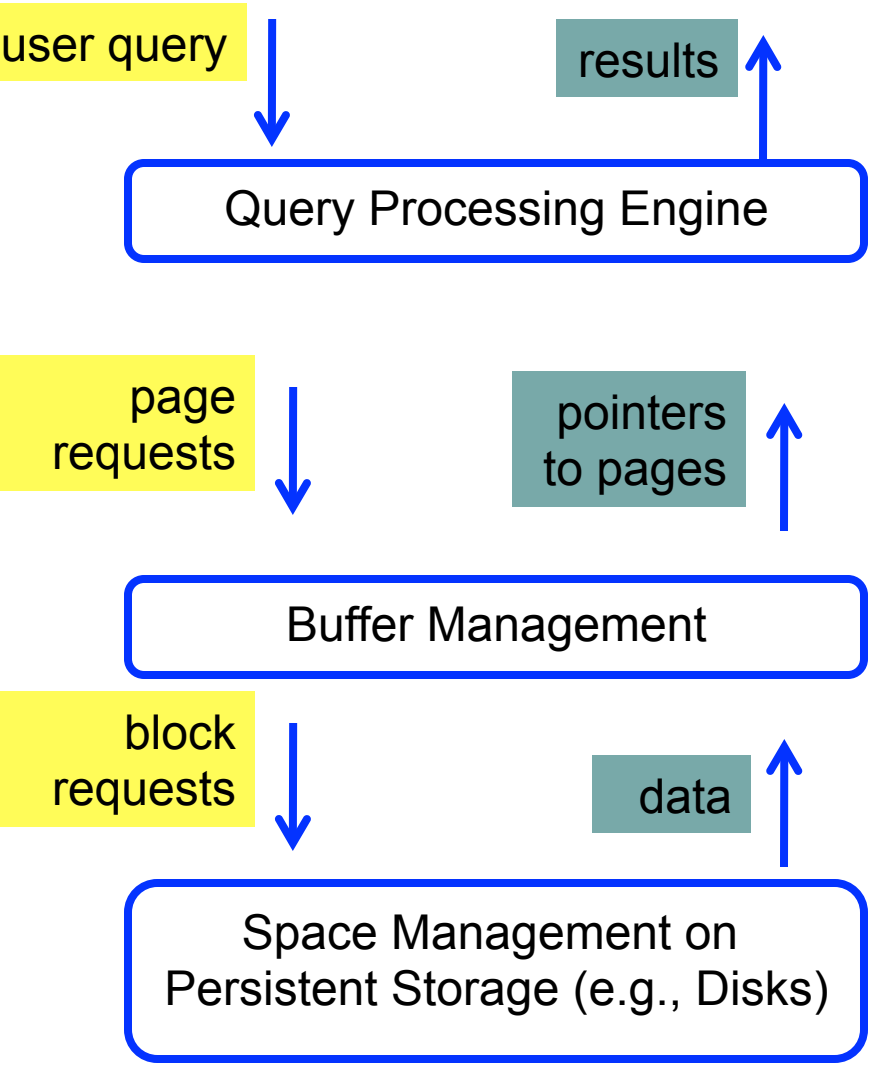


Databases



- 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

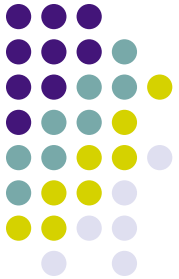


- Given a input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

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

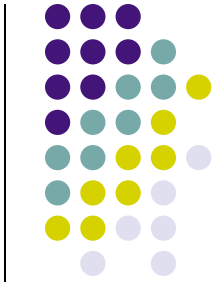
- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Outline



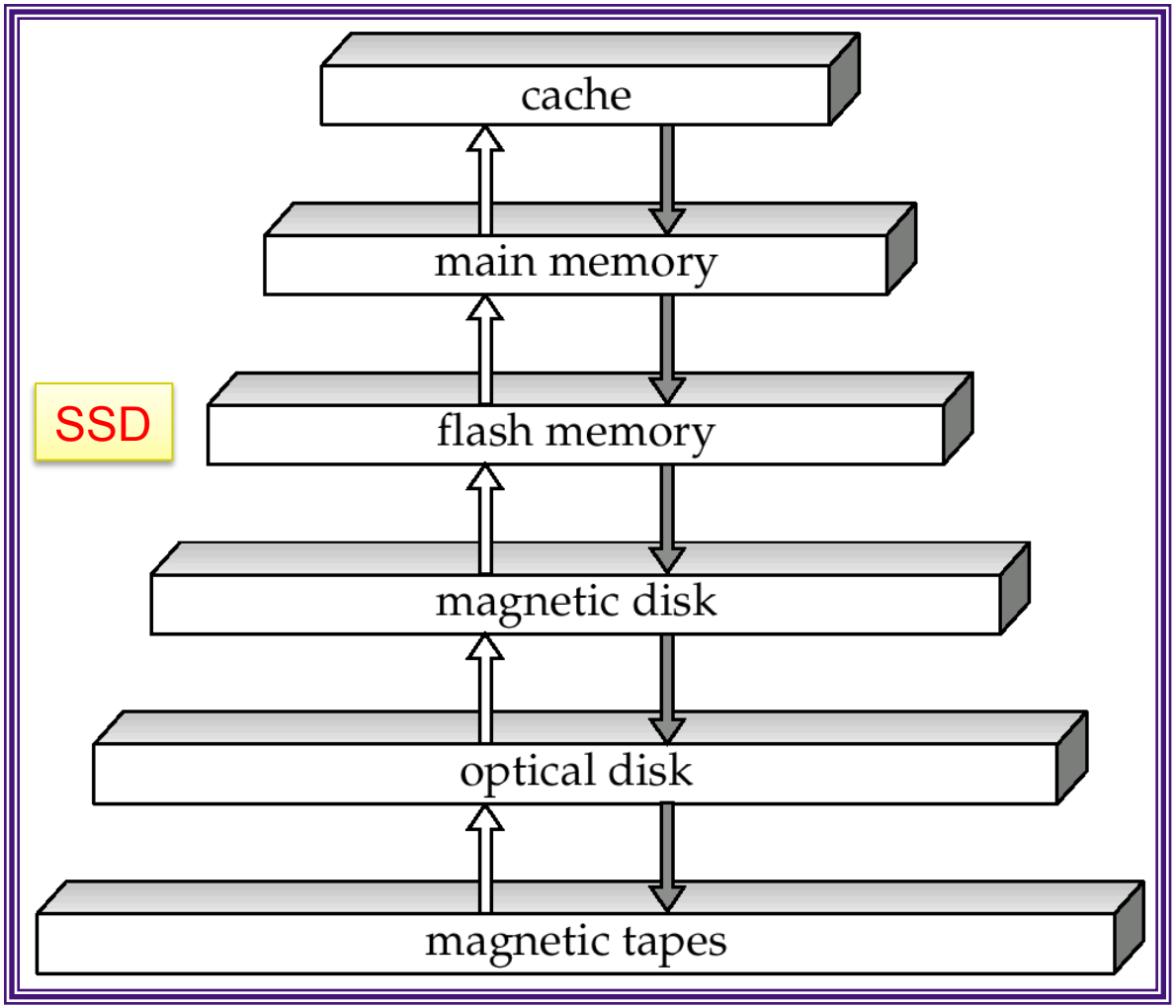
- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc.…

Storage Hierarchy

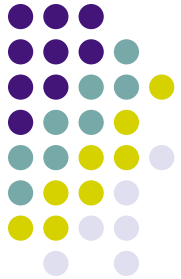


- Tradeoffs between speed and cost of access
- Volatile vs nonvolatile
  - Volatile: Loses contents when power switched off
- Sequential vs random access
  - Sequential: read the data contiguously
    - select * from employee
  - Random: read the data from anywhere at any time
    - select * from employee where name like '__a__b'
- Why care ?
  - Need to know how data is stored in order to optimize, to understand what's going on

Storage Hierarchy



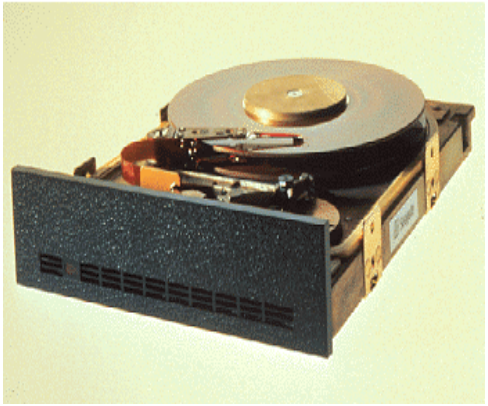
Outline



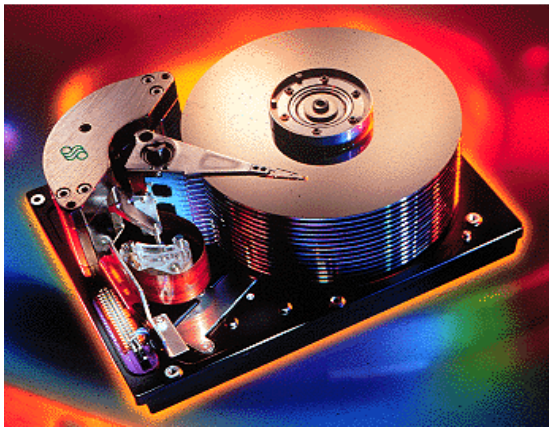
- Storage hierarchy
- Disks
- RAID
- File Organization
- Etc.…

1956
IBM RAMAC
24” platters
100,000 characters each
5 million characters

1979
SEAGATE
5MB



1998
SEAGATE
47GB



2006
Western Digital
500GB
Weight (max. g): 600g

Latest:

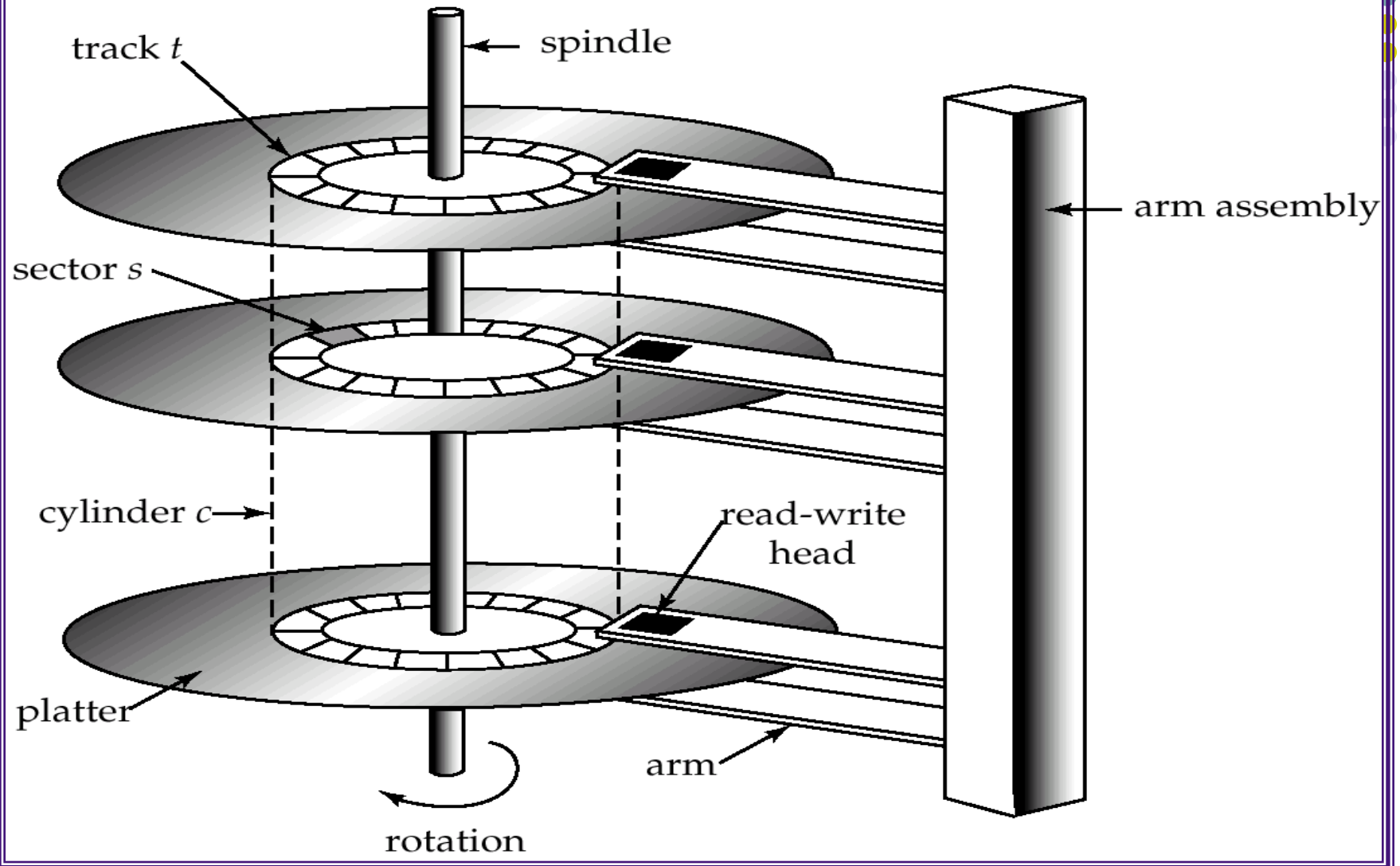
Single hard drive:

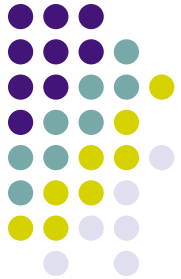
- Seagate Barracuda 7200.10 SATA
- 750 GB
- 7200 rpm
- weight: 720g
- Uses “perpendicular recording”

Microdrives

IBM 1 GB

Toshiba 80GB

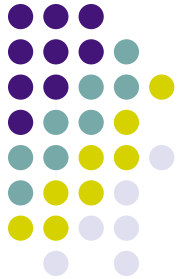

track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation



"Typical" Values

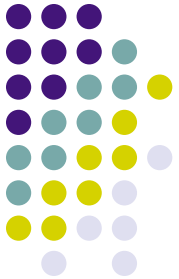
Diameter:	1 inch → 15 inches
Cylinders:	100 → 2000
Surfaces:	1 or 2
(Tracks/cyl)	2 (floppies) → 30
Sector Size:	512B → 50K
Capacity →	360 KB to 2TB (as of Feb 2010)
Rotations per minute (rpm) →	5400 to 15000

Accessing Data



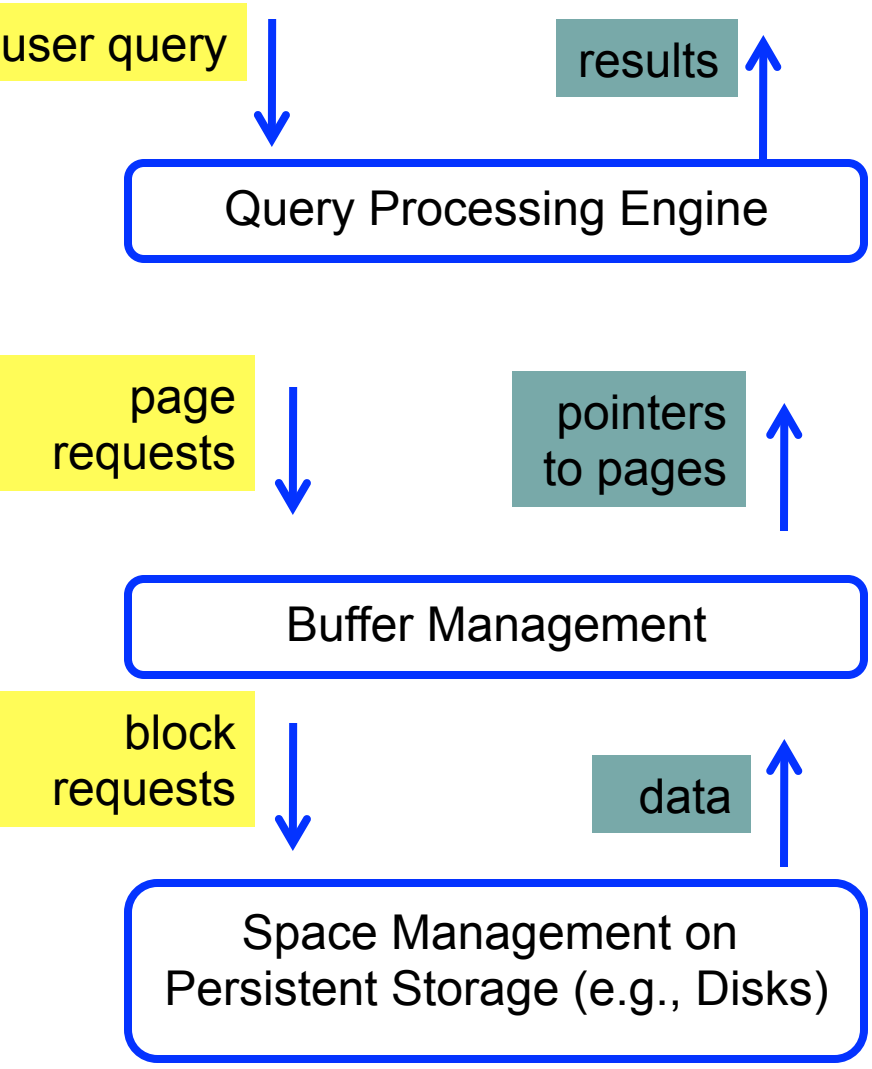
- Accessing a sector
  - Time to *seek* to the track (seek time)
    - average 4 to 10ms
  - + Waiting for the sector to get under the head (rotational latency)
    - average 4 to 11ms
  - + Time to transfer the data (transfer time)
    - very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 x 512bytes = 50 KB/s
- Data transfer rates
  - Rate at which data can be transferred (w/o any seeks)
  - 30-50MB/s to up to 200MB/s (Compare to above)
    - Seeks are bad !

Outline



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

Query Processing/Storage

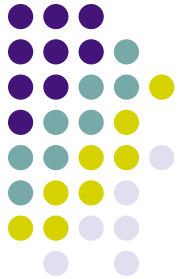


- Given a input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

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

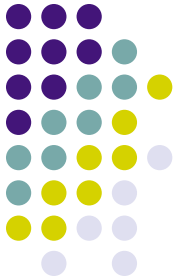
- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

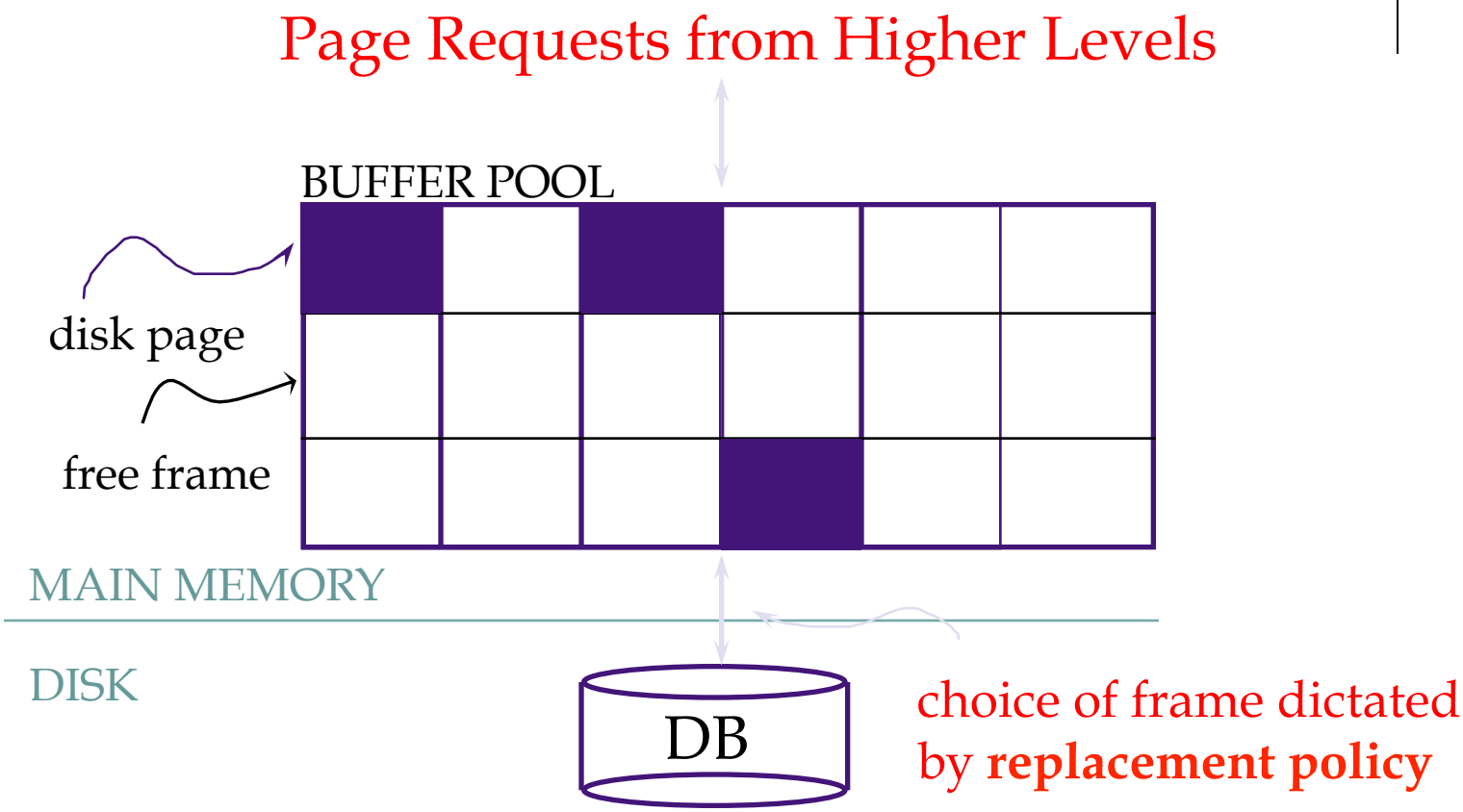
Buffer Manager



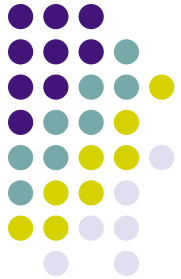
- When the QP wants a block, it asks the “buffer manager”
  - The block must be in memory to operate upon
- Buffer manager:
  - If block already in memory: return a pointer to it
  - If not:
    - Evict a current page
      - Either write it to temporary storage,
      - or write it back to its original 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



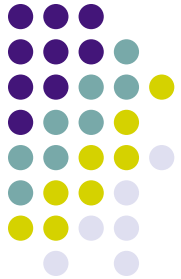


Buffer Manager



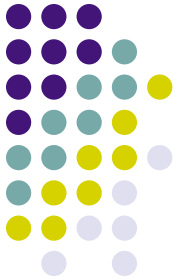
- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - Why ?
  - Clock ?
    - An efficient implementation of LRU

Buffer Manager



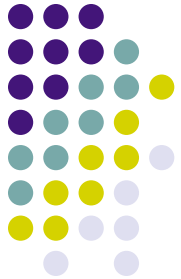
- *Pinning* a block
  - Not allowed to write back to the disk
- *Force-output (force-write)*
  - Force the contents of a block to be written to disk
- *Order the writes*
  - This block must be written to disk before this block
- Critical for fault tolerant guarantees
  - Otherwise the database has no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

Outline



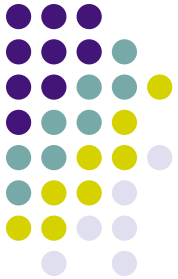
- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc.…

File Organization



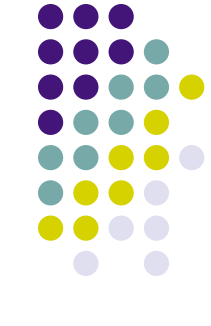
- How are the relations mapped to the disk blocks ?
  - Use a standard file system ?
    - High-end systems have their own OS/file systems
    - OS interferes more than helps in many cases
  - Mapping of relations to file ?
    - One-to-one ?
    - Advantages in storing multiple relations clustered together
  - 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



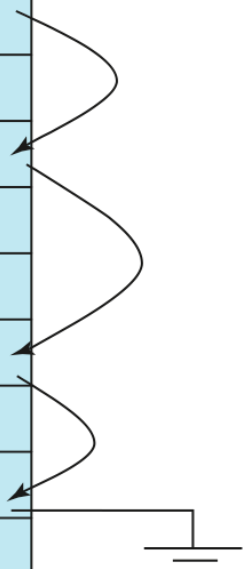
- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: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

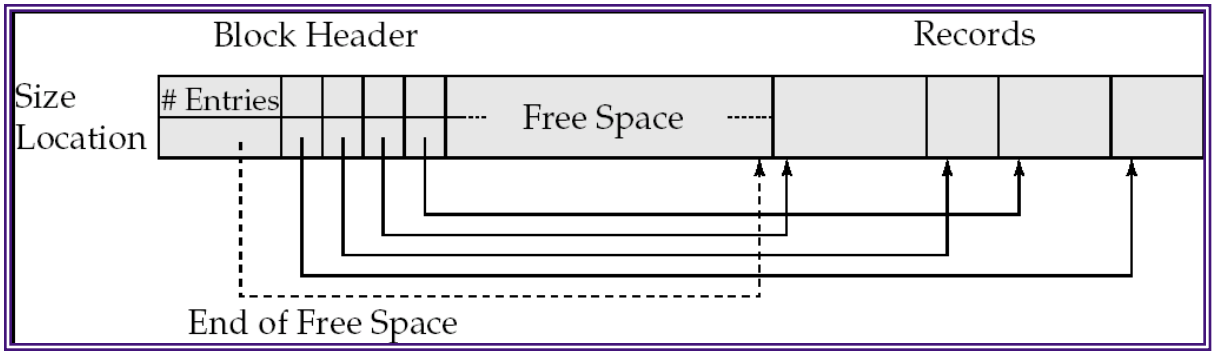
- Deleting: using “free lists”

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



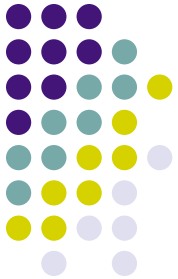
Variable-length Records

Slotted page structure



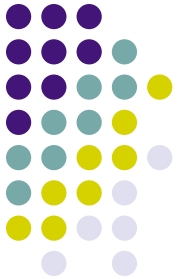
- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

File Organization



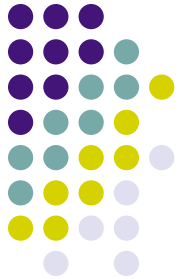
- Which block of a file should a record go to ?
  - Anywhere ?
    - How to search for “SSN = 123” ?
    - Called “heap” organization
  - Sorted by SSN ?
    - Called “sequential” organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a “hash” key
    - Called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

Sequential File Organization



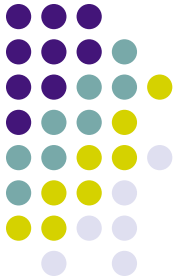
- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become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

Sequential File Organization



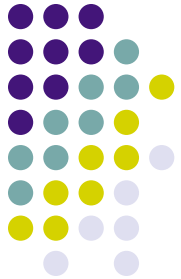
- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account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

Outline



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

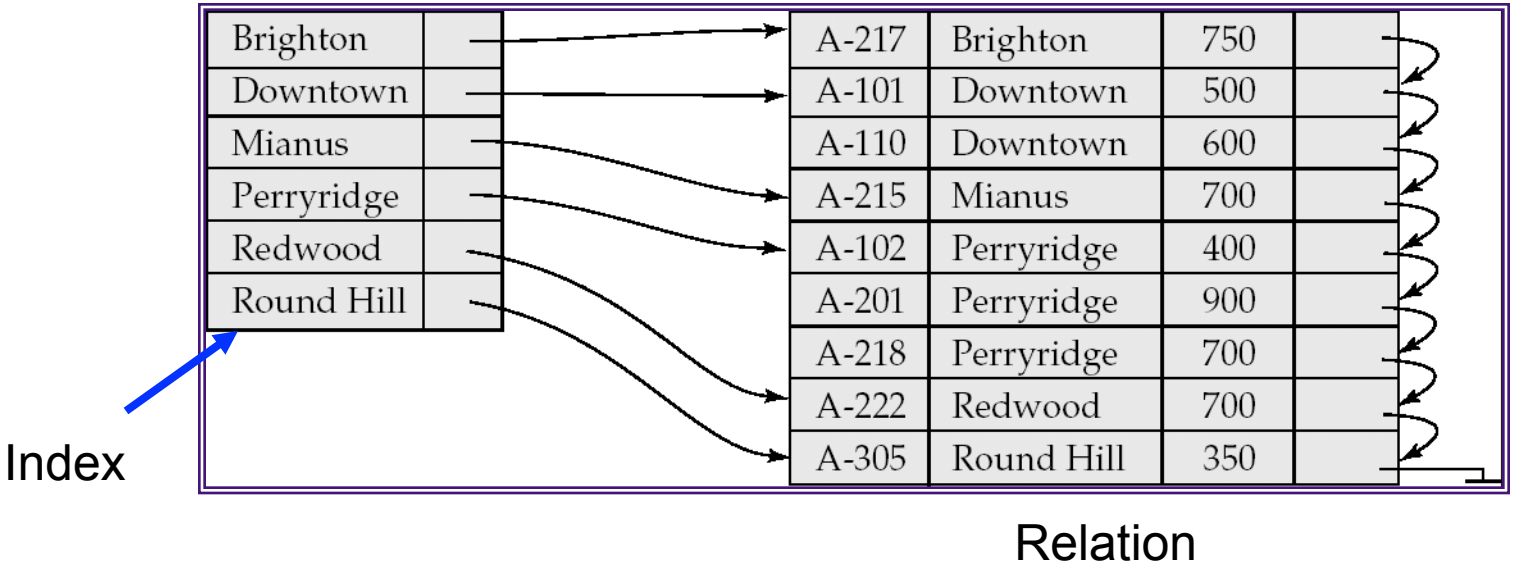
Index



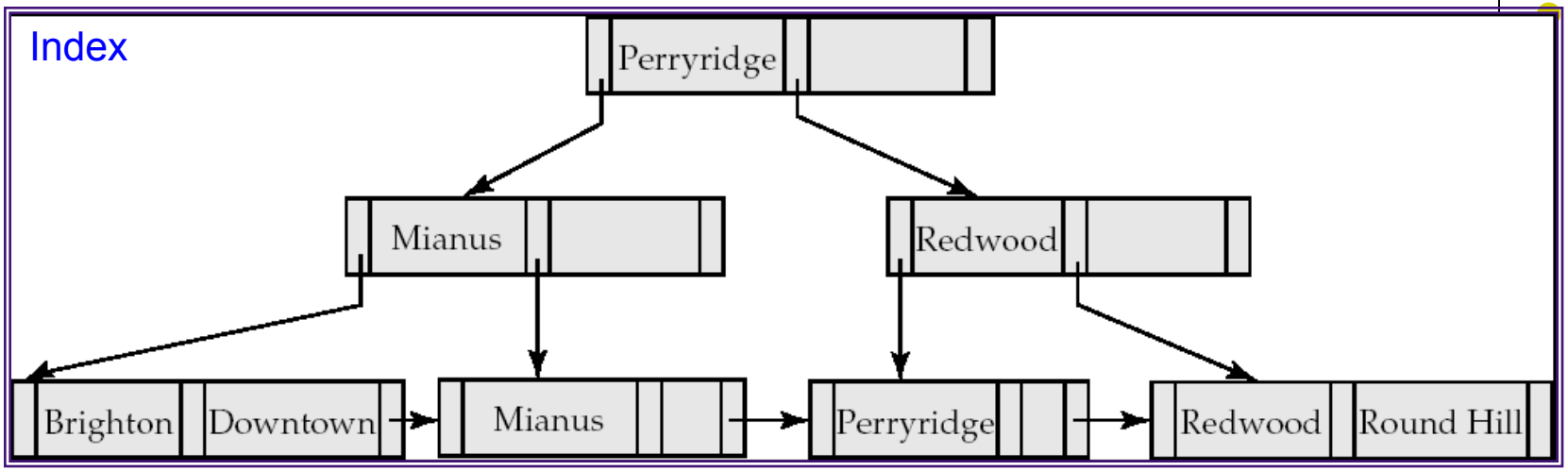
- A data structure for efficient search through large databaess
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- Search key:
  - Attribute 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

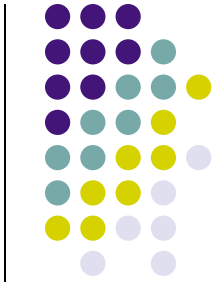
- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation



Example B+-Tree Index



B+-Tree Node Structure

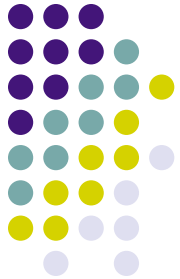


- Typical node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Properties of B+-Trees

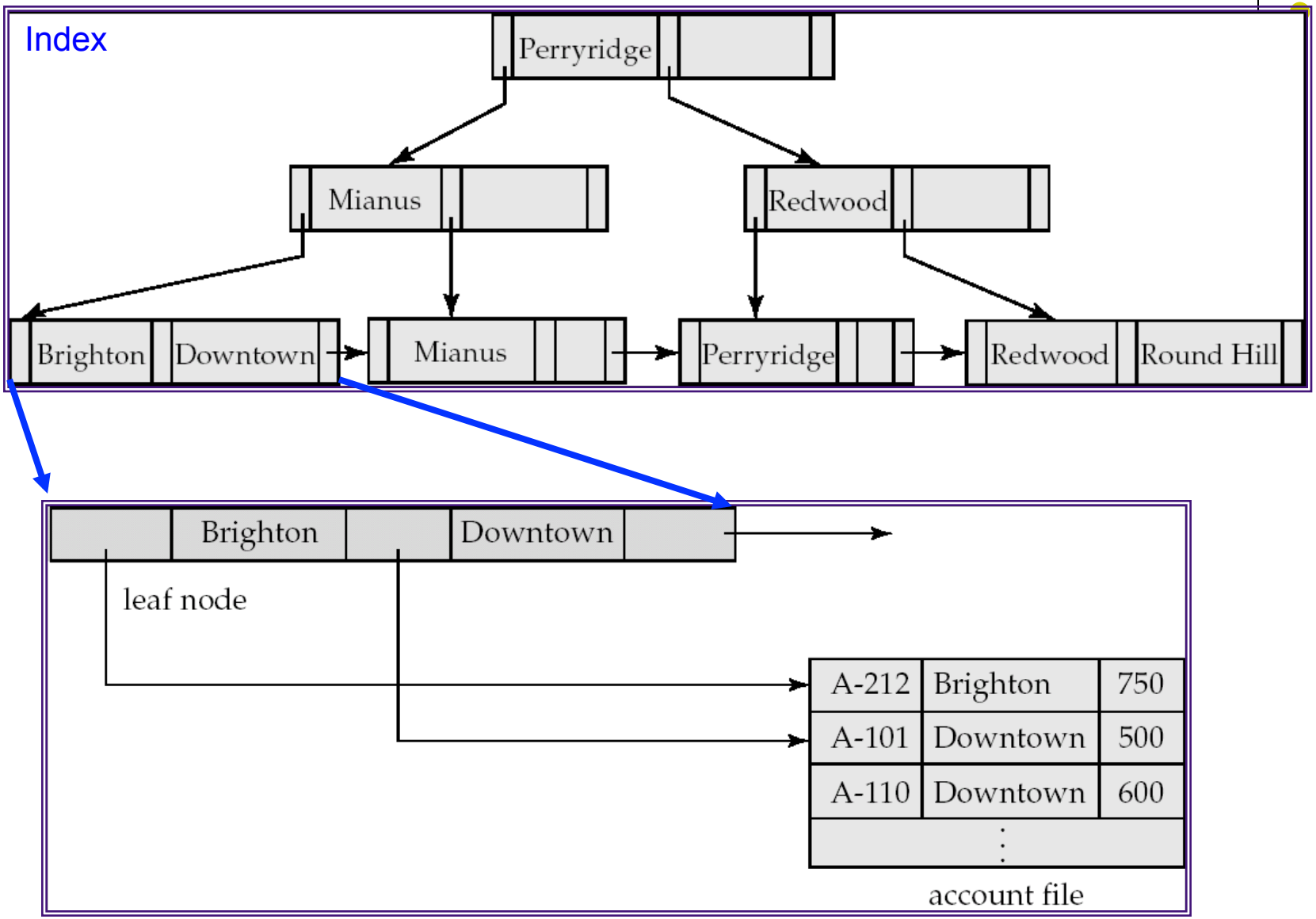


- It is balanced
  - Every path from the root to a leaf is same length

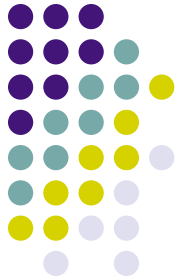
- Leaf nodes (at the bottom)
  - *P1* contains the pointers to tuple(s) with key *K1*
  - …
  - *Pn* is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

Example B+-Tree Index



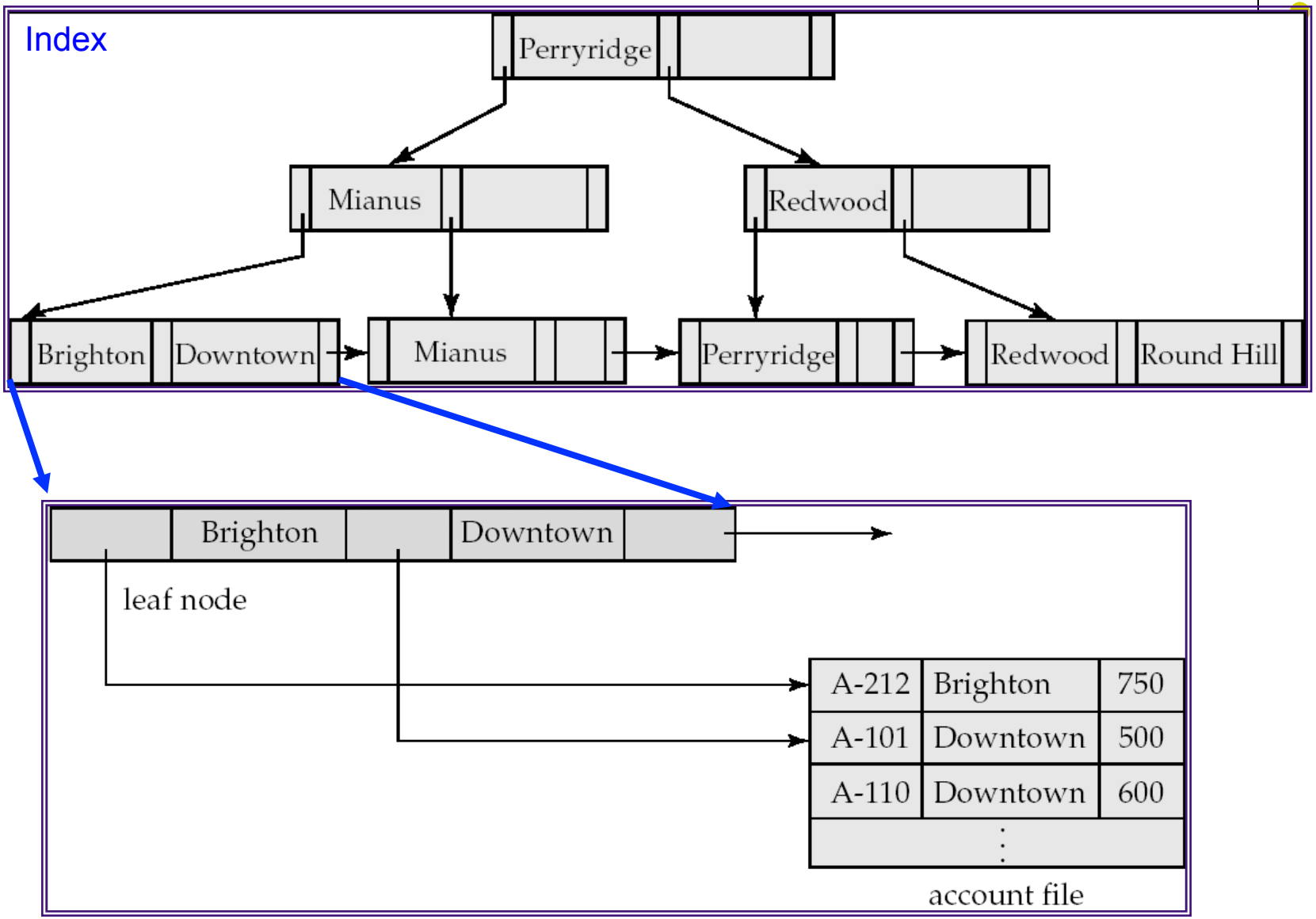
Properties



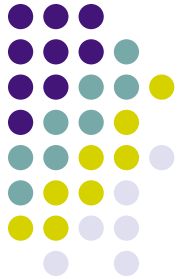
- Interior nodes

  - All tuples in the subtree pointed to by *P1,* have search key < *K1*
  - To find a tuple with key *K1'* < *K1,* follow *P1*
  - *…*
  - Finally, search keys in the tuples contained in the subtree pointed to by *Pn,* are all larger than *Kn-1*
  - Must contain at least n/2 entries (unless root)

Example B+-Tree Index

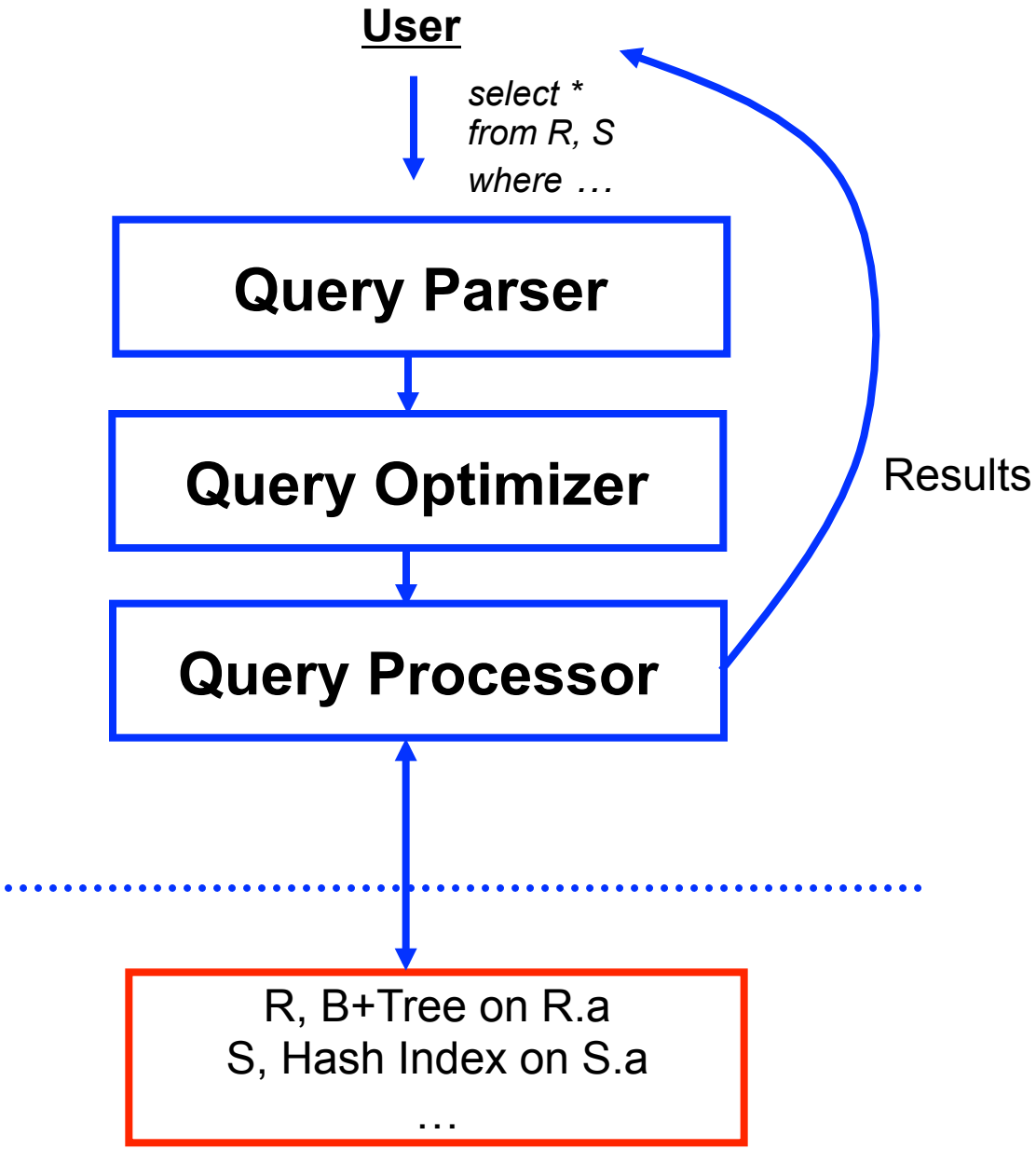


Query Processing



- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Overview

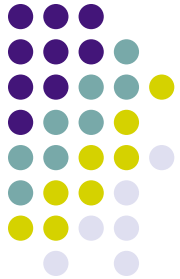


Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

Read the data from the files
Do the query processing
joins, selections, aggregates
…

"Cost"

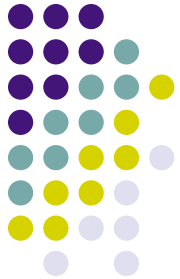


- Complicated to compute
- We will focus on disk:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $$b * t_T + S * t_S$$

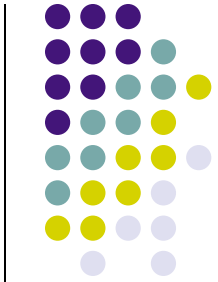
  - Measured in *seconds*

Query Processing



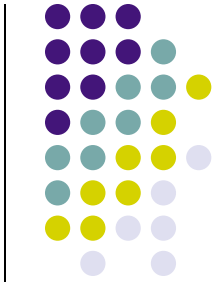
- Overview
- Buffer Manager
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Selection Operation



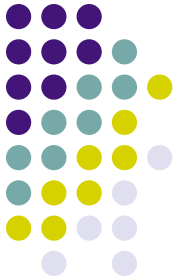
- select * from person where SSN = “123”
- Option 1: Sequential Scan
  - Read the relation start to end and look for “123”
    - Can always be used (not true for the other options)
  - Cost ?
    - *Let b_r = Number of relation blocks*
    - Then:
      - 1 seek and br block transfers
    - So:
      - $t_S + b_r * t_T$ *sec*
    - *Improvements:*
      - *If SSN is a key, then can stop when found*
        - *So on average, $b_r/2$ blocks accessed*

Selection Operation



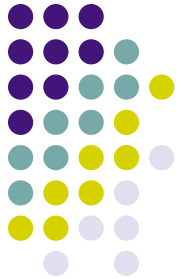
- select * from person where SSN = “123”
- Option 2 : Binary Search:
  - Pre-condition:
    - *The relation is sorted on SSN*
    - *Selection condition is an equality*
      - E.g. can’t apply to “*Name like ‘%424%’*”
  - Do binary search
    - Cost of finding the *first* tuple that matches
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
      - All I/Os are random, so need a seek for all
        - The last few are closeby, but we ignore such small effects
  - Not quite: What if 10000 tuples match the condition ?
    - Incurs additional cost

Selection Operation



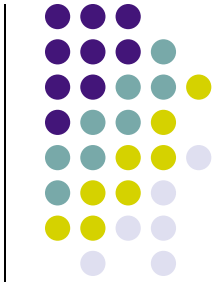
- select * from person where SSN = “123”
- Option 3 : Use Index
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all pointers using the index

Query Processing



- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Join



- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*

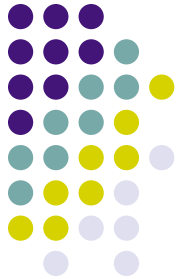
- Option 1: Nested-loops

  for each tuple r in R

  for each tuple s in S

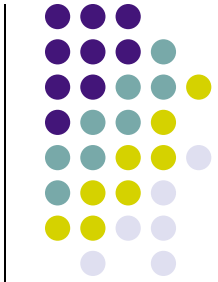
  check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called *outer relation*
- S called *inner relation*

Nested-loops Join



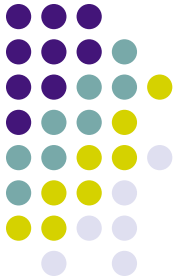
- Cost ? Depends on the actual values of parameters, especially memory
- b_r, b_s → *Number of blocks of R and S*
- n_r, n_s → *Number of tuples of R and S*
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: b_r
    - For each *R* tuple, must scan *S*: $n_r * b_s$
  - Seeks ?
    - $n_r + b_r$

Nested-loops Join



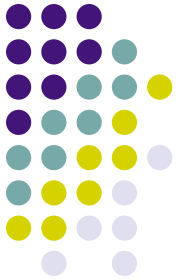
- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$
- Example:
  - Number of records -- *R:* $n_r = 10{,}000$, *S:* $n_s = 5000$
  - Number of blocks -- *R:* $b_r = 400$, *S:* $b_s = 100$
- Then:
  - blocks transferred: 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
- What if we were to switch R and S ?
  - 2,000,100 block transfers, 5100 seeks
- Matters

Nested-loops Join



- Case 2: *S* fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: *2*
- Example:
  - Number of records -- *R:* $n_r = 10{,}000$, *S:* $n_s = 5000$
  - Number of blocks -- *R:* $b_r = 400$, *S:* $b_s = 100$
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
- This is orders of magnitude difference

Hash Join



- Case 1: Smaller relation *(S)* fits in memory
- Nested-loops join:

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a

- Cost: $b_r + b_s$ transfers, 2 seeks
- The inner loop is not exactly cheap (high CPU cost)

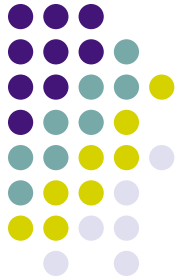
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

Hash Join



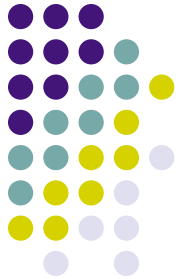
- Case 1: Smaller relation *(S)* fits in memory
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

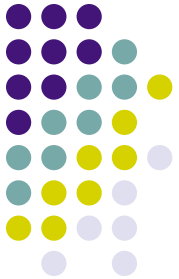
  use the hash index on S to find tuples such that S.a = r.a
- Cost: $b_r + b_s$ transfers, 2 seeks (unchanged)
- Why good ?
  - CPU cost is much better (even though we don't care about it too much)
  - Performs much better than nested-loops join when *S* doesn't fit in memory (next)

Hash Join



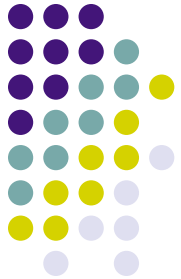
- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory

Hash Join



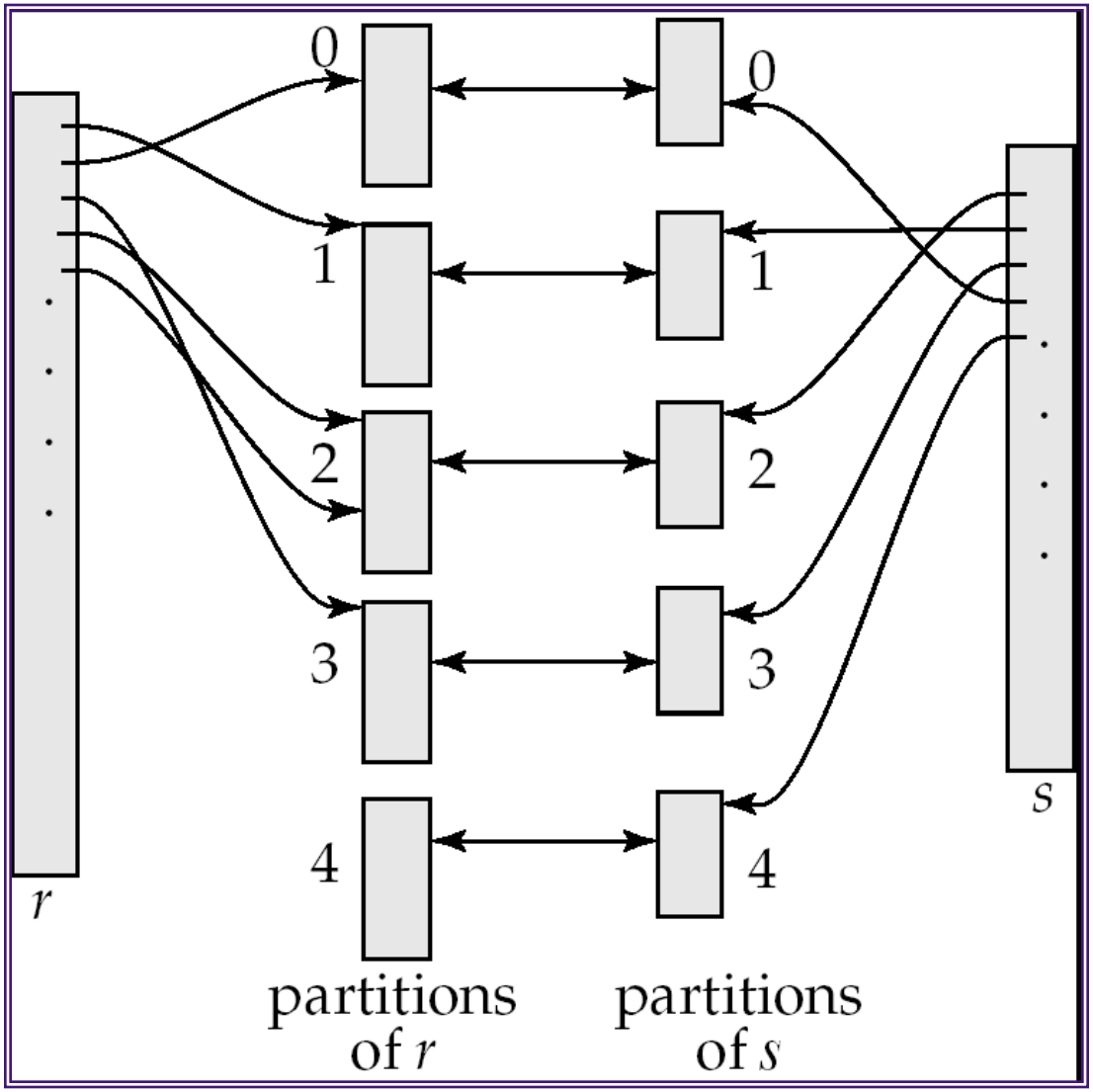
- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

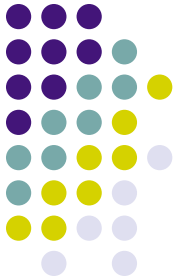


- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and bulid a hash index on it (S_i fits in memory)
  - Read R_i block by block, and use the hash index to find matches.
- Cost ?
  - $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
    - Where b_b is the size of each output buffer
  - Much better than Nested-loops join under the same conditions

Hash Join



Merge-Join (Sort-merge join)

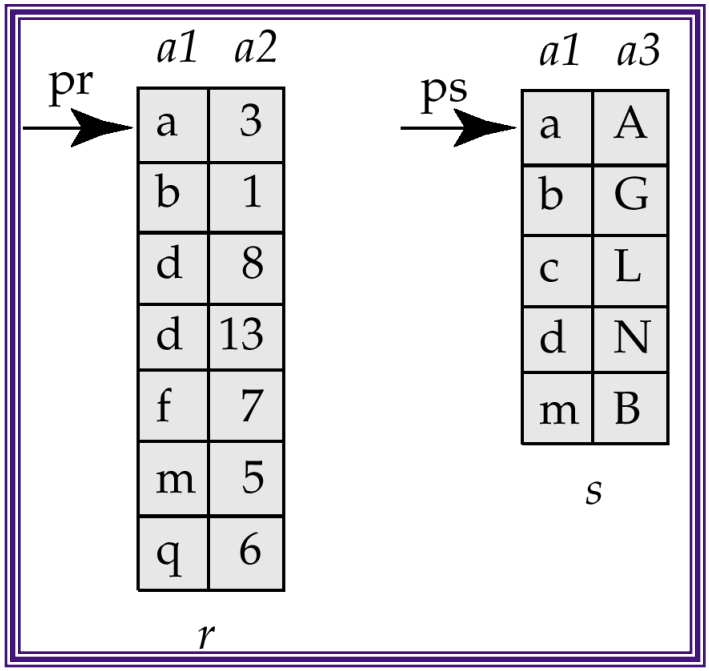


- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called “sort-merge join” sometimes

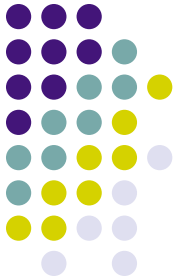
*select **
from r, s
where r.a1 = s.a1

Step:

1. *Compare the tuples at pr and ps*
2. *Move pointers down the list*
   - *Depending on the join condition*
3. *Repeat*

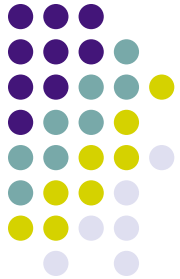


Merge-Join (Sort-merge join)



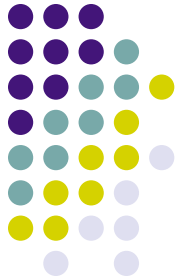
- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

Joins: Summary



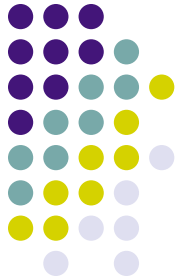
- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Hybrid hash join
  - An optimization on hash join that is always implemented
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

Query Processing



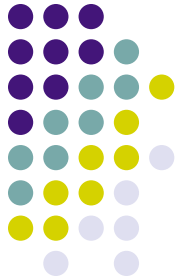
- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Sorting



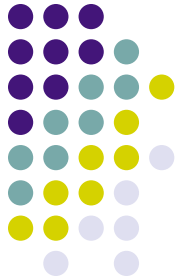
- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough…
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

External sort-merge



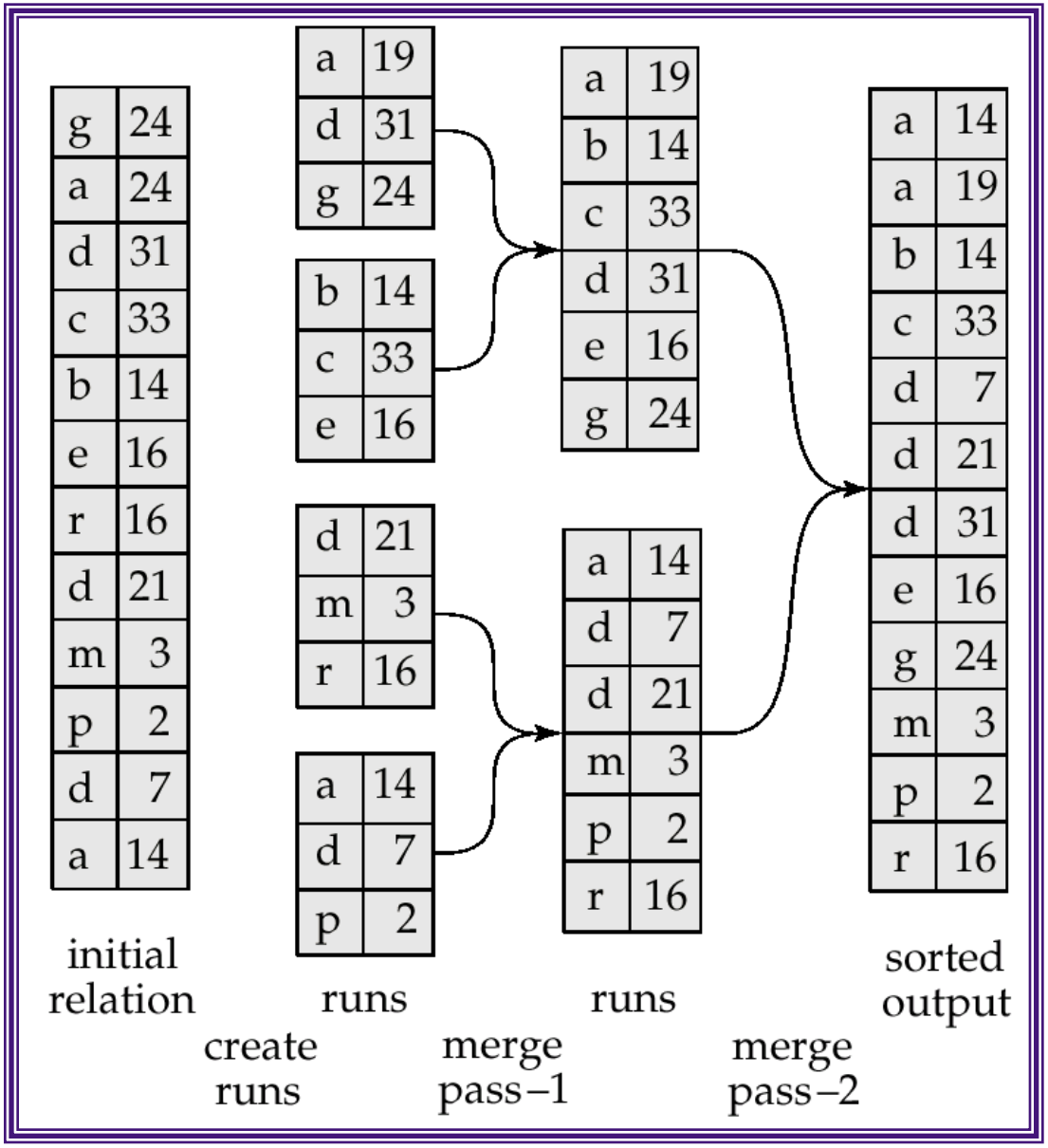
- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this “N” times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*

External sort-merge

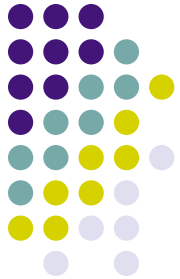


- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M ?*
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000 blocks = 4MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

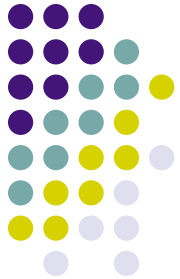


- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk accesses for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

  Thus total number of disk accesses for external sorting:

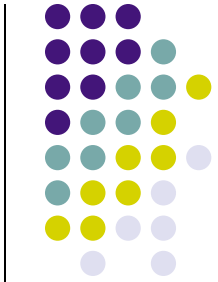
  $$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

Query Processing



- Overview
- Selection operation
- Join operators
- Other operators
- Putting it all together…
- Sorting

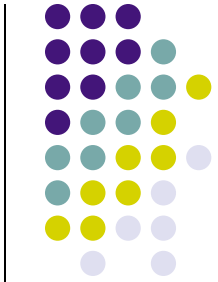
Group By and Aggregation



select a, count(b)
from R
group by a;

- Hash-based algorithm
- Steps:
  - Create a hash table on *a,* and keep the *count(b) so far*
  - Read *R* tuples one by one
  - For a new *R tuple, "r"*
    - Check if *r.a* exists in the hash table
    - If yes, increment the count
    - If not, insert a new value

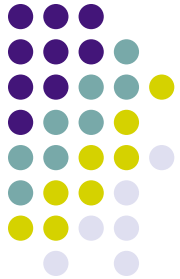
Group By and Aggregation



select a, count(b)
from R
group by a;

- Sort-based algorithm
- Steps:
  - Sort *R* on *a*
  - Now all tuples in a single group are contigous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

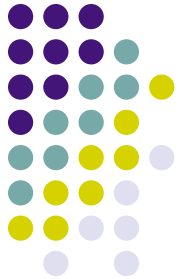
Group By and Aggregation



select a, AGGR(b) from R group by a;

- sum(), count(), min(), max(): only need to maintain one value per group
  - Called “distributive”
- average() : need to maintain the “sum” and “count” per group
  - Called “algebraic”
- stddev(): algebraic, but need to maintain some more state
- median(): can do efficiently with sort, but need two passes (called “holistic”)
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- count(distinct b): must do duplicate elimination before the count

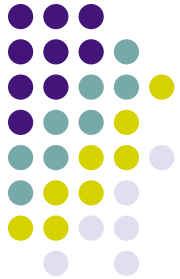
Duplicate Elimination



select distinct a
from R ;

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev = r*
    - else
      - Skip *r*

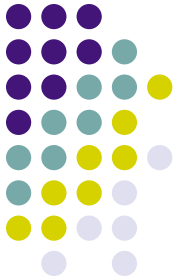
Set operations



*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"union"*: append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based algorithm*

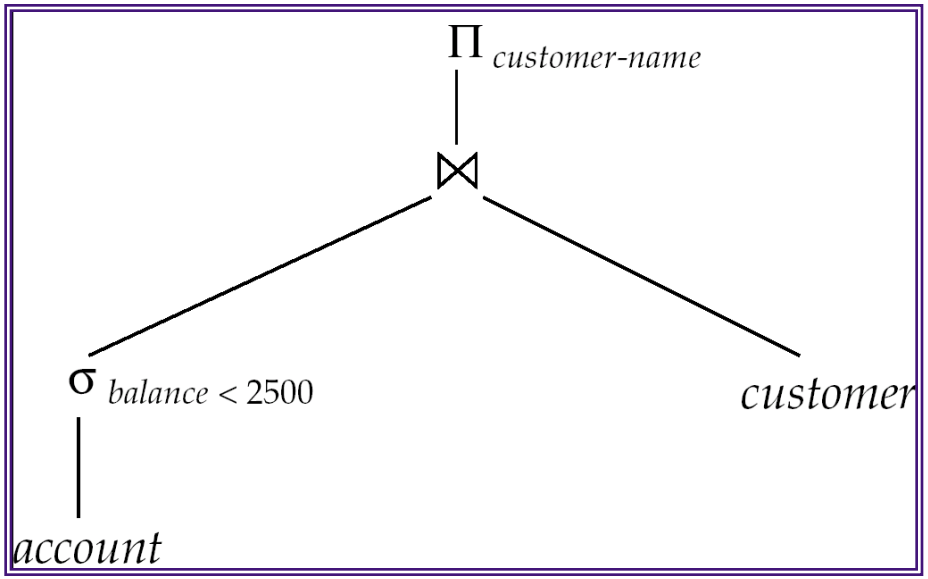
Query Processing



- Overview
- Selection operation
- Join operators
- Other operators
- Putting it all together…
- Sorting

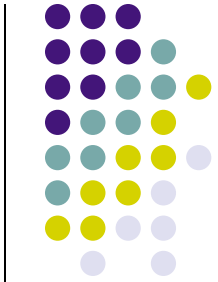
Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500



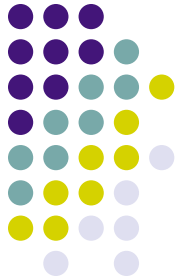
- Two options:
  - Materialization
  - Pipelining

Evaluation of Expressions



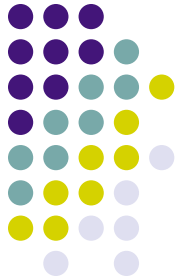
- Materialization
  - Evaluate each expression separately
    - Store its result on disk in *temporary relations*
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about

Materialization



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

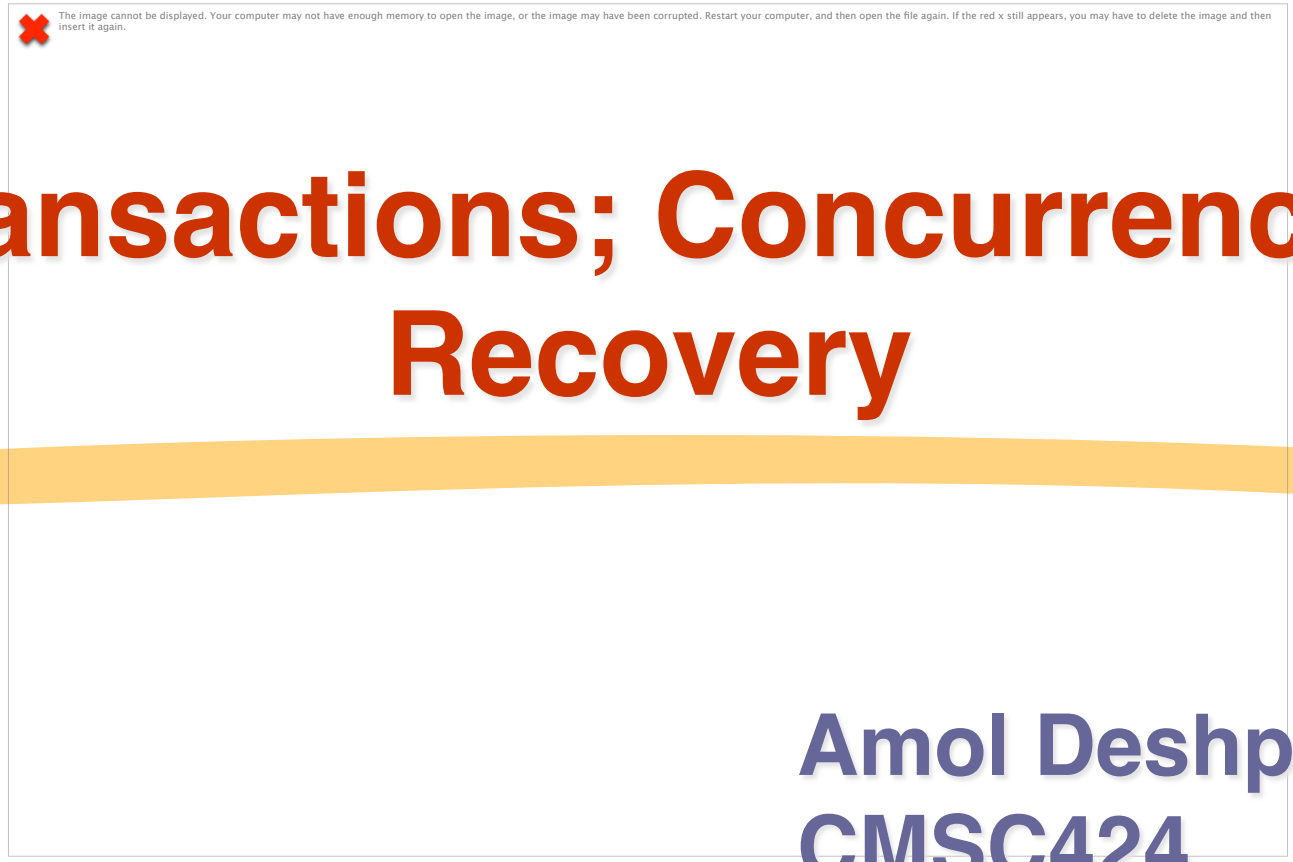
Pipelining



- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- A "blocking" operation: An operation that has to consume entire input before it starts producing output tuples



Transactions; Concurrency; Recovery

Amol Deshpande
CMSC424

Overview

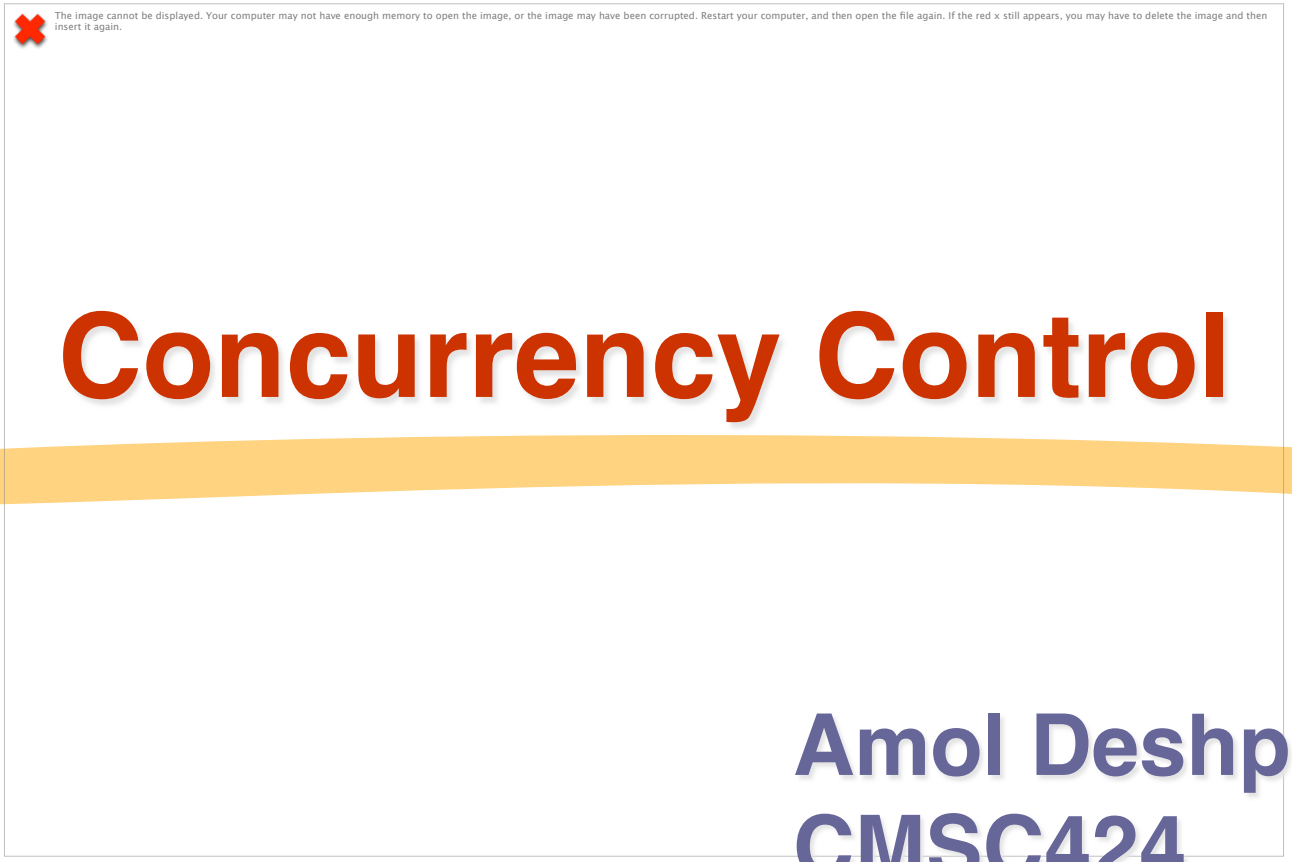
- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost
- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

How does..

- .. this relate to *queries* that we discussed ?
  - Queries don't update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing total balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries
- TPC-C vs TPC-H

Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - BUT… **disks are durable. They might stop, but data is not lost.**
    - For now.
  - Disks only guarantee *atomic sector writes,* nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES



Concurrency Control

Amol Deshpande
CMSC424

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - *Shared* (S) locks (also called *read locks)*
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (also called *write locks)*
    - Obtained for updating a data item

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
read(B)	read(A)
B ←B-50	read(B)
write(B)	display(A+B)
read(A)	
A ←A + 50	
write(A)	

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
B ←B-50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
lock-X(A)	unlock(B)
read(A)	display(A+B)
A ←A + 50	
write(A)	
unlock(A)	

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
  - Depends

T2 lock type	T1 lock type	Should allow ?
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible,* grant the lock, otherwise T1 waits in a *queue.*

Other CC Schemes: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of “optimistic concurrency control”
- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated

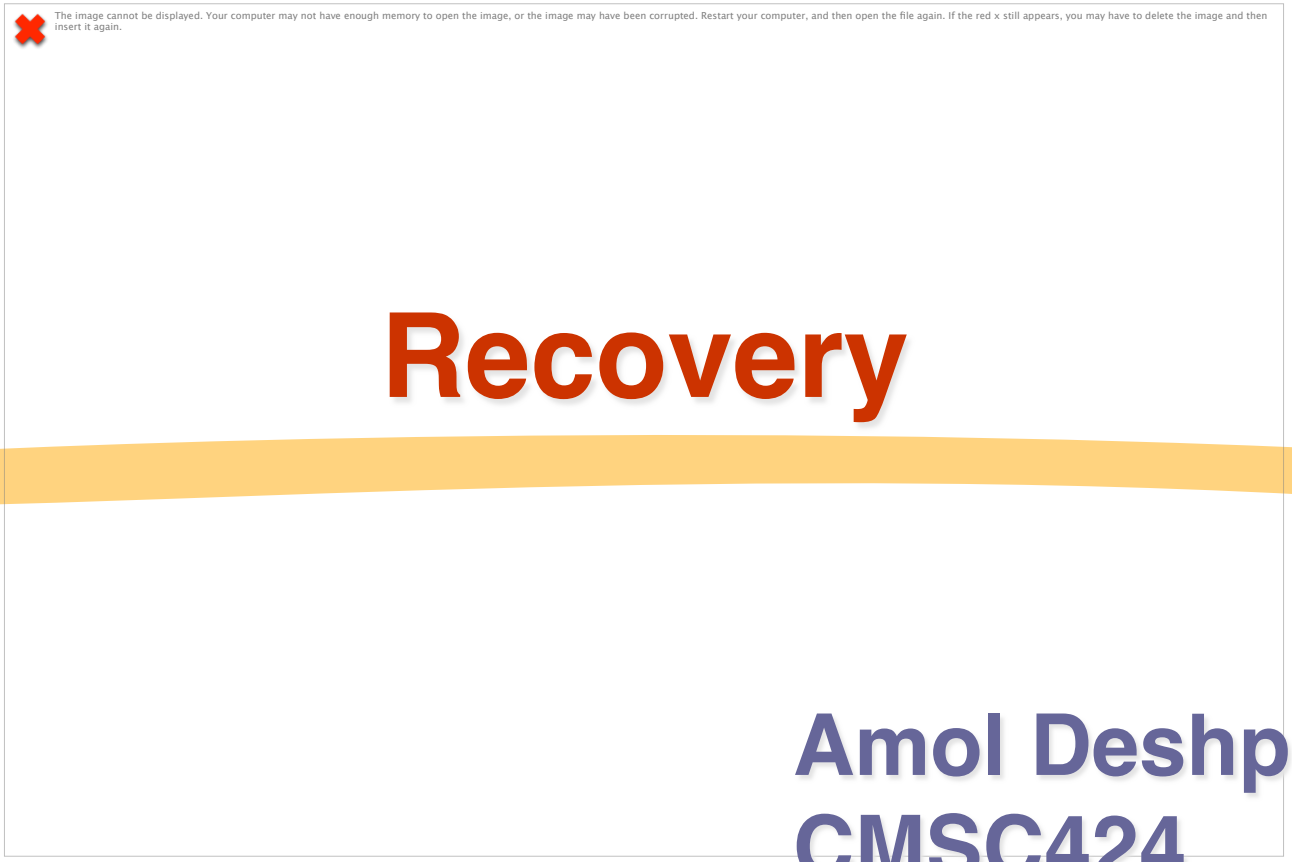
Other CC Schemes: Snapshot Isolation

- Read queries:
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

Other CC Schemes: Snapshot Isolation

- Advantages:
  - Read query don't block at all, and run very fast
  - As long as conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols

- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation



Recovery

Amol Deshpande
CMSC424

Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability ?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
      - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.

- Essentially similar solutions

Reasons for crashes

- Transaction failures
  - Logical errors, deadlocks
- System crash
  - Power failures, operating system bugs etc
- Disk failure
  - Head crashes; ***for now we will assume***
    - ***STABLE STORAGE:*** ***Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data***

Log-based Recovery

- Most commonly used recovery method
- Intuitively, a log is a record of everything the database system does
- For every operation done by the database, a *log record* is generated and stored *typically on a different (log) disk*
- *<T1, START>*
- *<T2, COMMIT>*
- *<T2, ABORT>*
- *<T1, A, 100, 200>*
  - T1 modified A; old value = 100, new value = 200