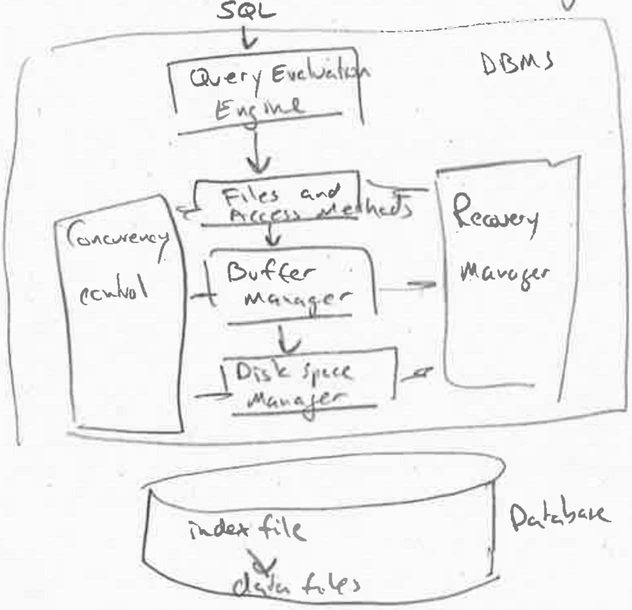
Tuesday 24/2/2015: Data Storage

Lecture Topics

I. Data on External Storage
II. File Organization
III. Cost Model
IV. Data Structures
V. Indexing

I. Data on External Storage



I.1 Disks

Disk is sequence of cylinders
cylinder is a sequence of tracks
track is a sequence of sectors

Each sector contains:
- Data area
- Header
- Error correcting information

A "block"
disk ≈ sequence of blocks

- All blocks are the same size
  512 B or 4096 B
- Must always read a block
  - seek time – moving arm
  - rotational delay – wait for block under head
  - transfer time – move data to/from disk
- Assume virtual memory page is same size as block

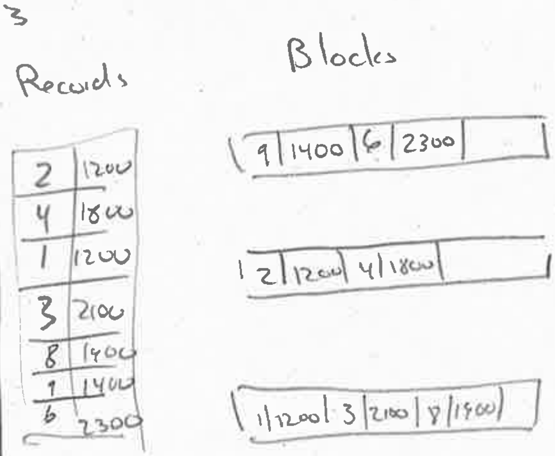
II Files

II.1 File is a sequence of records
records are fixed or variable size
(logical view)

- (Physical view) File is a sequence of blocks (fixed size, not contiguous)
- Easy to find: first, last, next, prev block

II.2: Assumptions
- fixed size records
  - No record is in more than 1 block
- several records per block
- "left over" space at end of block

II.3



II.4 To answer a query:
- Read all blocks into RAM
- Get relevant data from blocks
- Additional Processing to answer

Q: How to make this fast?

Tuesday 24/2/2015 Data Storage Pg. 2

III Cost Model

III.1

In the book:

B number of data pages

R: Number of Records per page

D: Average time to read/write page

In this lecture:

- Read or Write block = 1 unit of time
- Processing RAM = free
- Ignore caching

Justification:

- Disk IO > CPU
- Don't want to model disk contiguity
- Do not want to model cache slots

Goal => minimize number of block accesses

Heuristic -> make each block read/write as "useful" as possible

Implications:

if you know where E#2 and E#4 are

- data structure cost model = 2 (RAM accesses)
- database cost model = 1 (block access)

#2, #9

- data structure = 2
- database = 2

III.2 Operations

- Scan
- Equality
- Range
- Insert
- Delete

Tools: (1) File organization
(2) Indexes (structure showing where records are)

(1) = when you read a block, many useful records
(2) = know where the blocks are

- Maintaining F.O. and index is not free
- Extreme cases: only read vs only write

only read → many indexes & file organization

only write → no indexes or organization

IV Data Structures

- Heap (unsorted sequence. different from data structure "heap" and process "heap")
- Sorted sequence
- Hashing
- 2-3 tree

IV.1 Heap

Find: 'O(n)' operations

Delete: O(n) operations -> maybe compacted?

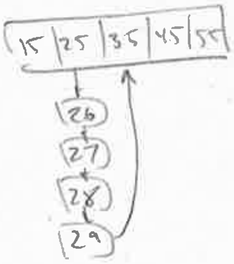
Insert: O(1) or O(N)

IV.2 Sorted Sequence

Find: O(log N) binary search

Delete: O(log N) or O(log N + N)
Find value, and compact

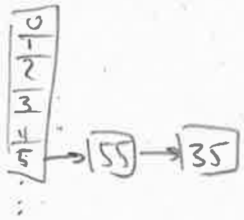
Insert: O(log N) or O(log N + N)
Find and push to tail



24/2/2015 Data Storage Pg 3

IV.3 Hashing

- Pick a B bigger than N
- function h
  $h: I \to B$
- bucket directory



- Assume computing "h" is free

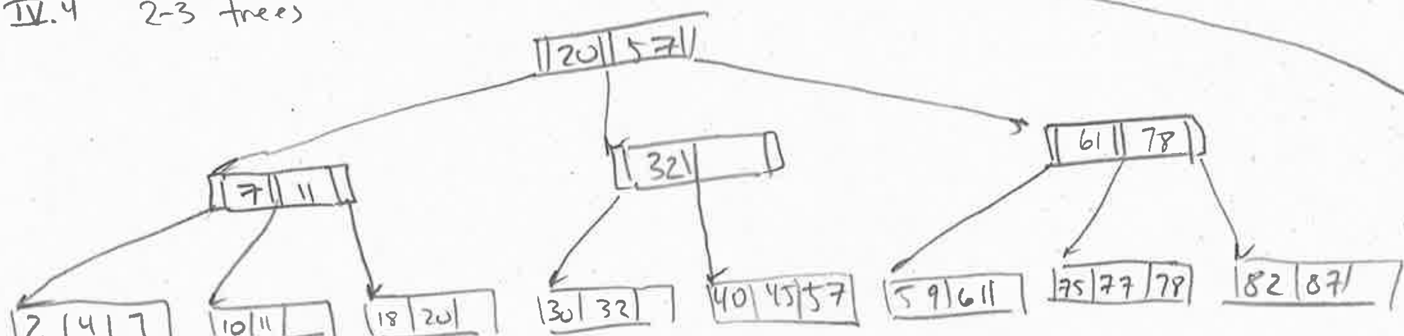
Find: $O(1)$ or $O(N+1)$

Insert: $O(1)$ or $O(N+1)$

- prohem if B is too small, need to "grow" and rehash
  - can "amortize" this cost

Delete: $O(1)$ or $O(N+1)$

IV.4 2-3 trees



- Rooted (has a root) and directed (order of children matter)
- All paths from root to leaves are the same
- For each child of a node, there is an index value
- For non-leaf, indx indicates the largest value of the leaf in the sib tree
- Each leaf has 2 or 3 values

- May need to "restructure" when you insert or delete
- restructuring is linear in the number of levels of the tree $\approx O(\log_3 N)$ or $O(\log_2 N)$

Find: $O(\log N)$

Insert: $O(\log N)$

Delete: $O(\log N)$

IV.5 Which to use?

- if large N, use hash or 2-3
- if many "range", use 2-3
- if not many range, use hash

24/2/2015 Data Storage Pg. 4

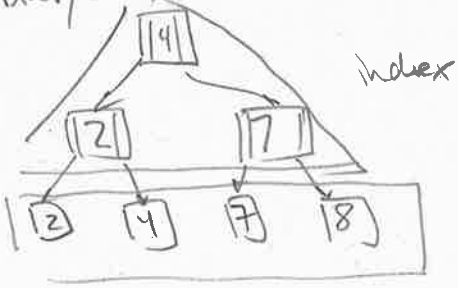
I Indexing

- Data file of blocks
  - block of records
  - Each record has a "key"
- Index file
  - records of the form (key, Block address)
  - the B points to the block of the file that contains k
  - Not saying that there is an index record for every k

= binary search tree

=> data, not organized in any way

Binary search tree:



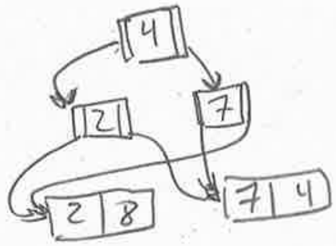
Dense index – for every record (key) in the database, there is a pointer in the index to the block containing it.

Sparse index – opposite of dense

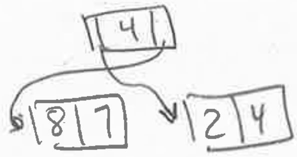
=> remember, pointers point to blocks, not records. Once the block is in RAM, it can be found quickly

clustered – a file can be "fully" sorted, without many records between blocks

unclustered – opposite



dense and unclustered



Sparse and clustered

Best Case Scenario:

Clustered + Sparse

Clustered => lots of "related" records

Sparse => efficient to find a block

Example: easy to get all records with a value greater than 4

Summary:

Sparse + unclustered = Bad, cannot find records easily

Dense + clustered = unnecessarily large index

Dense + unclustered = good

Sparse + clustered = best