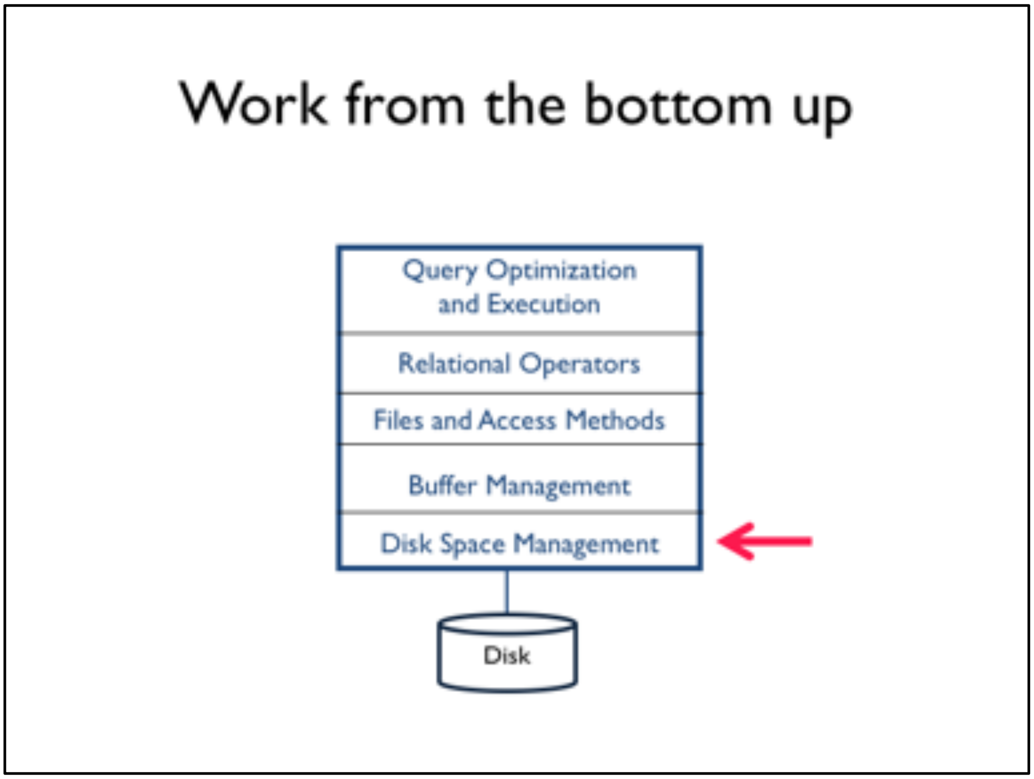
L8
Disk, Storage, and Indexing

Eugene Wu

Fall 2015



Classic architecture. Most large systems from Oracle, IBM etc use this. Optimized for disk.
Newer systems fro new hardware may change the organization, or tweak components, hwever these concepts are still there.

$ Matters

Why not store all in RAM?

- Costs too much
  - High-end Databases today ~Petabyte (1000TB) range.
  - ~60% cost of a production system is in the disks.
- Main memory not persistent
  - Obviously important if DB stops/crashes

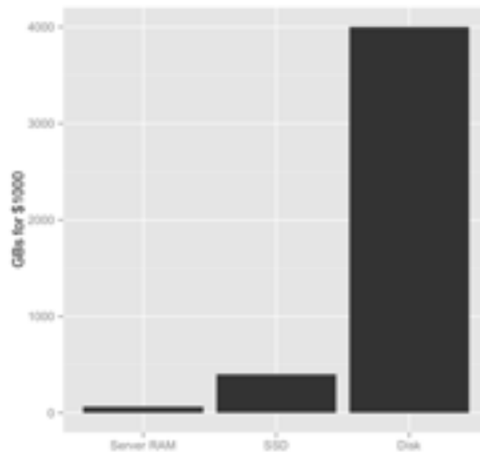
Some systems are *main-memory* DBMSes, topic for advanced DB course

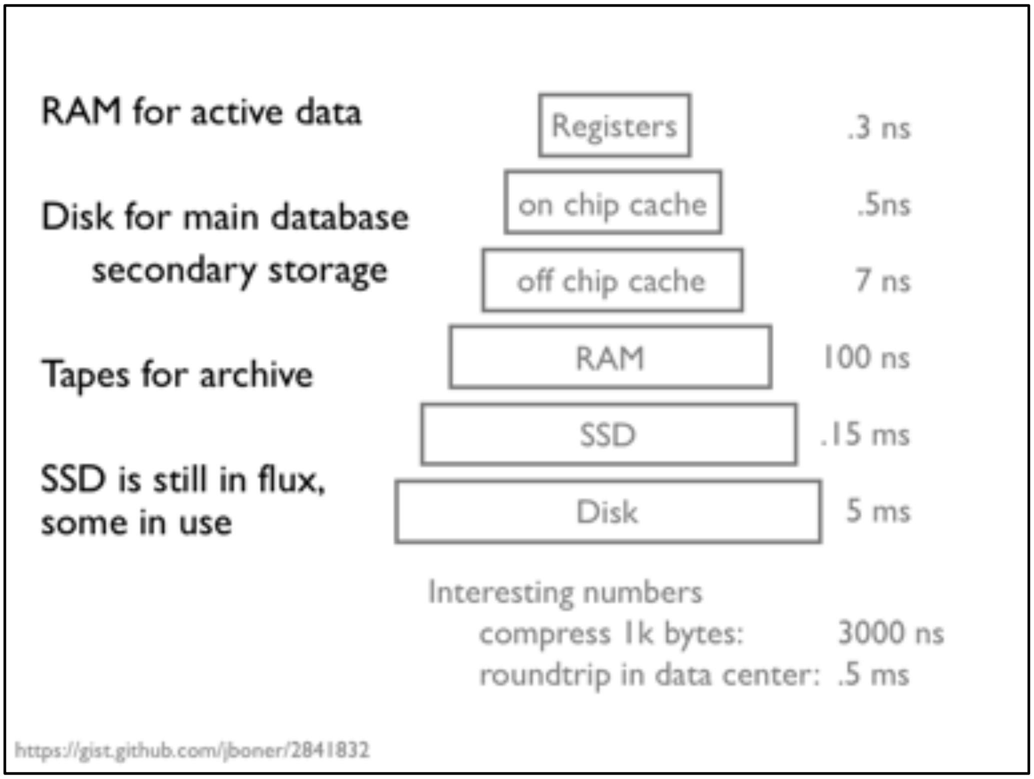
in many cases, youdon't have petabytes of data, and main memory or SSDs are practical.

$ Matters

Newegg enterprise $1000

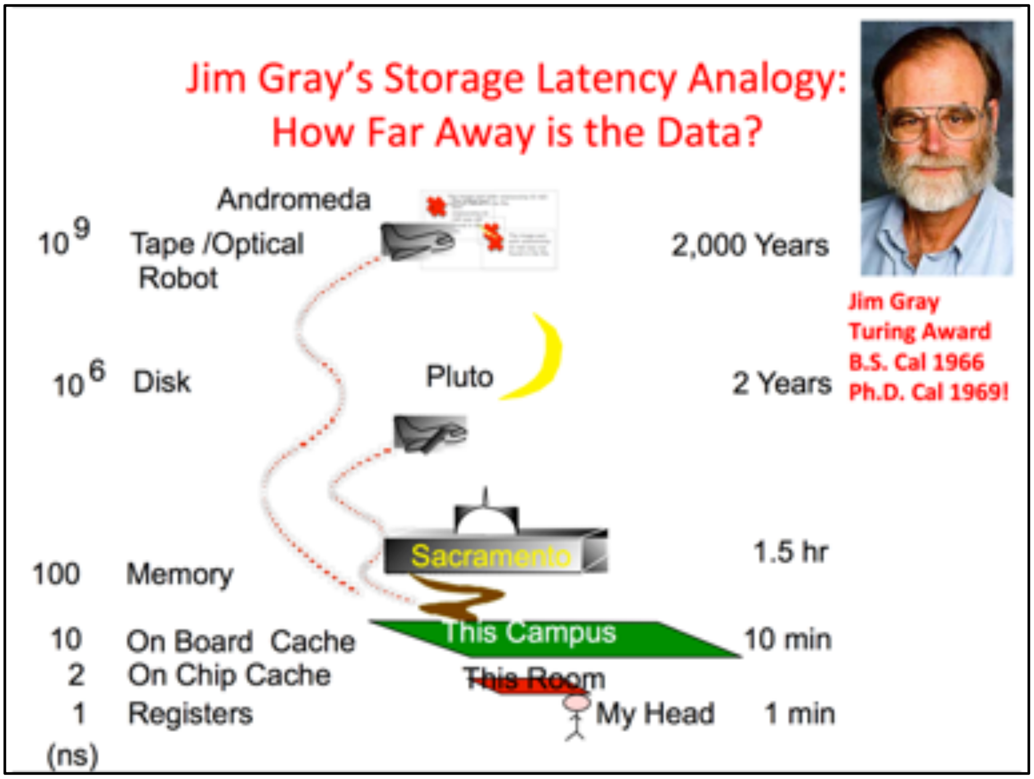
- RAM: 64
- SSD: 400
- Disk: 4000





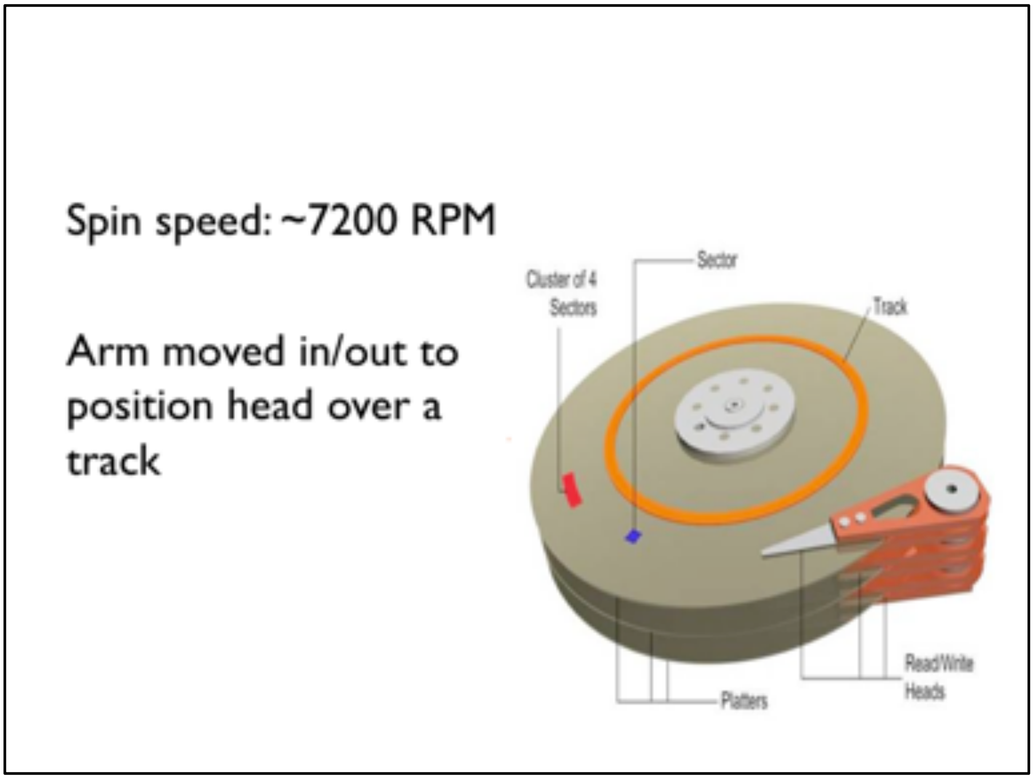
We'll focus on RAM and Disk and algorithms between the two.
It turns out what really matters is the performance ratio between the two
there are some algorithms specialized to how a disk works, but for most part the types of techniques DBs use between RAM and disk can be applied in for example chip cache and RAM, and indeed many techniques such as pre-fetching are commonly used in the Os as well

L2: 14x L1
Ram: 20x L2
compression: 0.003 milliseconds
Roundtrip in a data center: 0.5 ms: 10x faster than disk seek
disk: 50k times slower than RAM



Philly = sacramento
jim gray basically wrote the book on transaction processing, the ideas of transactions, ACID, data cube, 5 minute rule
5 minute rule: The 5-minute random rule: cache randomly accessed disk pages that are re-used every 5 minutes or less.
In 2000, Gray and Shenoy applied a similar calculation for web page caching and concluded that a browser should "cache web pages if there is any chance they will be re-referenced within their lifetime."[8]

go read his wikipedia page
https://en.m.wikipedia.org/wiki/Jim_Gray_(computer_scientist)



disk is a stack of these platters that are spinning just like a dj turn table – just 100x faster
the platters are coated with magnetic material that is used to flip bits between 1 and 0
the data is laid out in tracks – concentric circles like trees or music record
the track is split into tiny sectors or blocks of roughly 64kb, varies by manufacturer
think of a block like a page – similar to an OS page, usually OS pages are mulitple of disk blocks for nice properties

when want to read/write, youll near a little whirling sound, as the arm moves to position the head,

no random access, no pointers, no objects.

what's changed, has been the magnetic material on the surface of the platters, and encodings, etc, but the main thing, the physics, has not changed.
that's the only mechanical device in your computer!

API means need to
- READ: transfer page of data to disk from ram
- write: transfer page from disk to ram

Kinda slow. really slow

IS this the right api?

Time to access (read or write) a disk block

seek time	2-4 msec avg
rotational delay	2-4 msec
transfer time	0.3 msec/64kb page

Throughput

read	~150 MB/sec
write	~50 MB/sec

Key: reduce seek and rotational delays

- HW & SW approaches

Next block concept (in order of speed)

- blocks on same track
- blocks on same cylinder
- blocks on adjacent cylinder

Sequentially arrange files

- minimize seek and rotation latency

When sequentially scanning: Pre-fetch

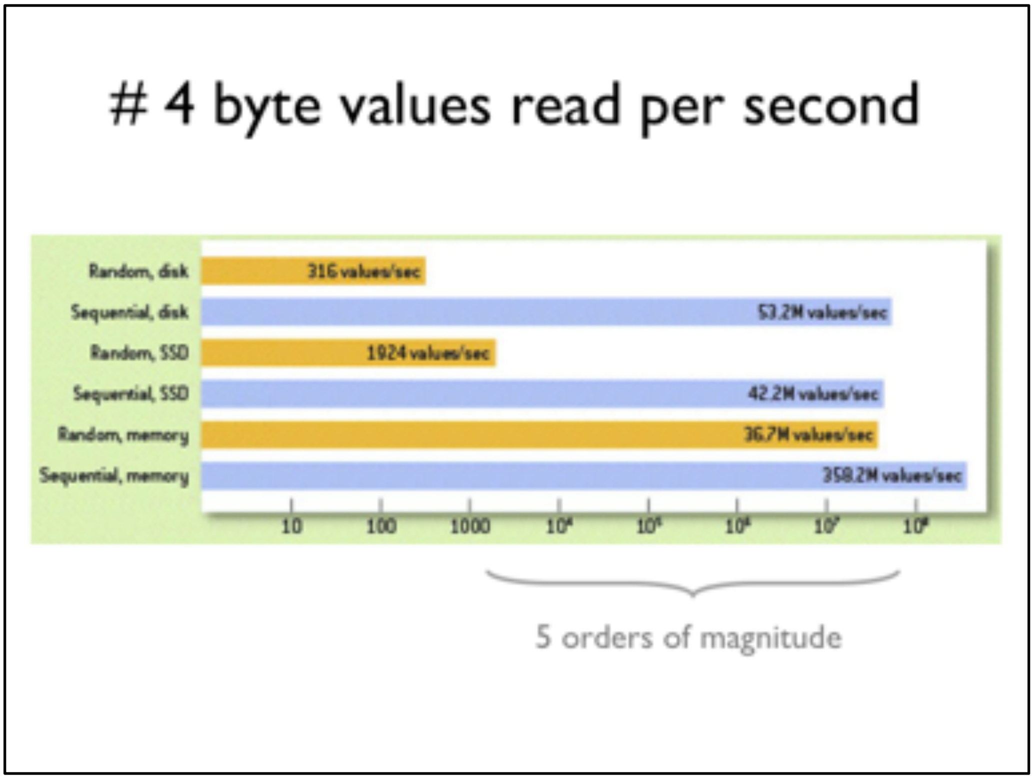
- >1 page/block at once

if CPU is going to have to wait in order for the data, you don’t want tit to waste resources, and one way to deal with it is while CPU is working on other data, to pre-fetch the next page or next pages

SSD maybe

- Fast changing, not yet stabilized
- Read small & fast
  - single read: 0.03ms
  - 4kb random reads: 500MB/sec
  - seq reads: 525MB/sec
- Write is slower for random
  - single write: 0.03ms
  - 4kb random writes: 120MB/sec
  - seq writes: 120MB/sec
- Write endurance limited
  - 2-3k cycle lifetimes
  - 6-10 months

need to replace



throughput is comparable between disk and SSD! The main difference is random access and latency

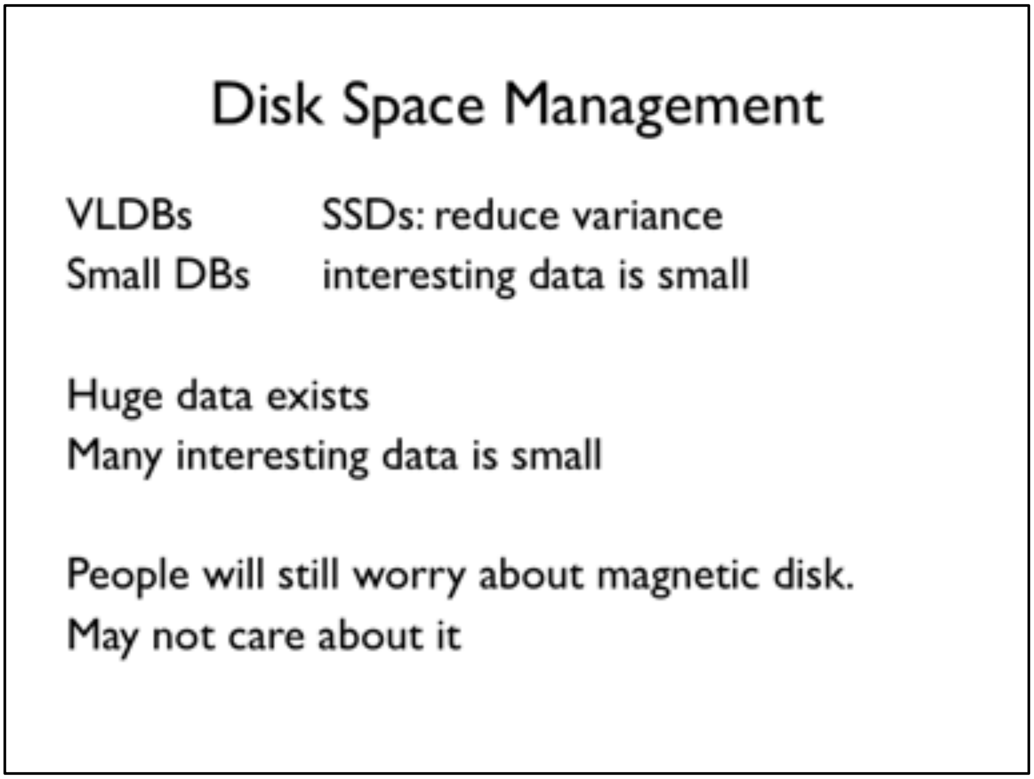
Pragmatics of Databases

Most databases are pretty small

- All global daily weather since 1929: 20GB
- 2000 US Census: 200GB
- 2009 english wikipedia: 14GB

Data sizes grow faster than moore's law

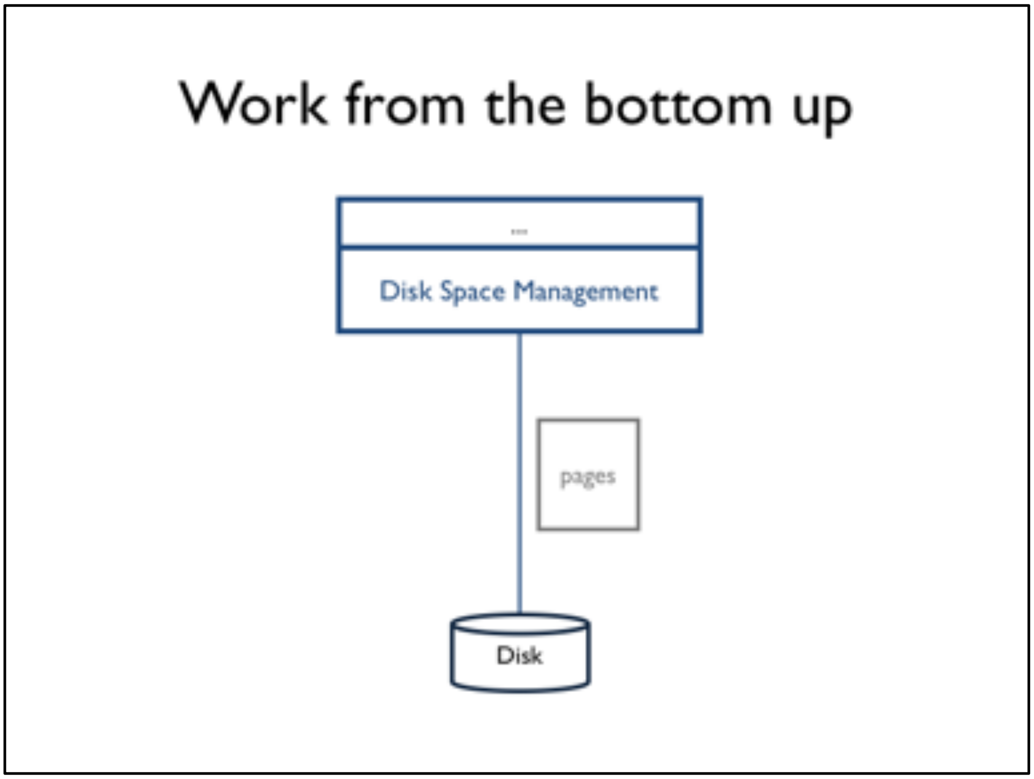
when would you have cloud scale databases
if in sciences, or in practice – small numbero f machines, or a big desktop makes sense



At this scale, if you're talking a huge amount of data, then SSDs primarily reduce latency variance
For small databases, doesn't matter too much.

FB haystack – all in memory
lots and lots of variety

This slide seems somewhat bi-polar. Tries to illustrate the range of what matters



All of this is very complicated – and we DONT want to deal with sectors, or tracks, or platters.
So the abstraction use to communicate with the disk is in pages. We say we want to write or read a set of pages, and the disk controller will help manage that request.

Disk Space Management

Lowest layer of DBMS, manages space on disk

Low level IO interface:
- allocate/deallocate a page
- read/write page

Sequential performance desirable
- try to ensure sequential pages are sequential on disk
- hidden from rest of DBMS
- but algorithms may assume sequential performance

could imagine directly managing the hardware, but then you need to talk to different physical devices and deal with drivers etc.
A huge amount of operating system code is for dealing with and providing drivers for a wide range of hardware devices, so best let OS manage that and give us a file abstraction

usually, we allocate a huge amount of space on disk – usually allocated sequentially, and once we have that, use file API to read write blocks, with the understanding that the file is on disk
Higher level don't have guarantees that things will be sequential, BUT if we know things are sequential we can use better algorithms

our operations will be at the level of pages

Files

Pages are IO interface
Higher levels work on records and files (of records)

File: collection of pages
- insert/delete/modify record
- get(record_id) a record
- scan all records

Page: collection of records
- typically *fixed size* (8kb in PostgreSQL)

May be stored in multiple OS files spanning multiple disks

Think File == Table

need way of mapping records to pages to files

abstraction is pages, and we read and write pages
note that it's a COLLECTION. no ordering
no assumptions of WHERE the pages live, we don't care.

contraist with unix file API
- stream of bytes
- there's an ordering
- DB File is unordered

We'll have different types of files, with different organizations that make certain types of record access patterns faster or slower
Fancier files provide additional access methods for e.g., looking up records by value rather than record id

Units that we'll care about

Ignore CPU cost
Ignore RAM cost

B # *data* pages on disk for relation
R # records per data page
D avg time to read/write data page to/from disk

Simplifies life when computing costs
Very rough approximation, but OK for now
    ignores prefetching, bulk writes/reads, CPU/RAM

we'll talk about non-data pages that arepart of the index

ultimately this will all be important for talking about performance tradeoffs of different ways to physically represent a file, so we need some performance modeling. ignoring a lot of details including seek times, etc could always add that in

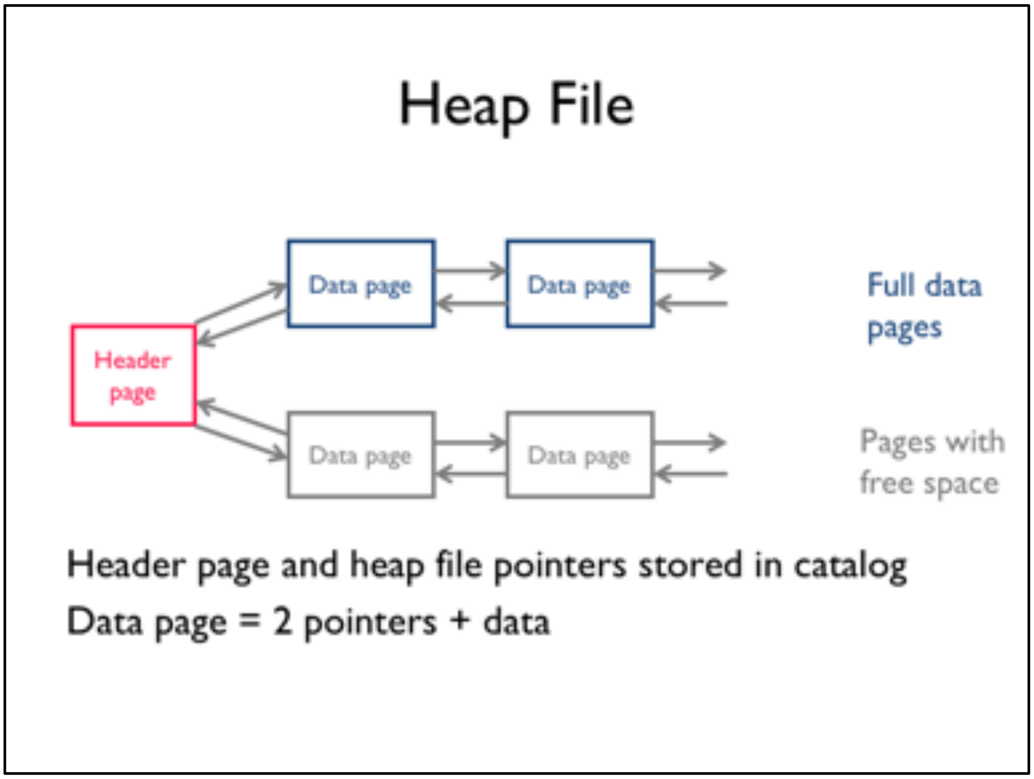
Given the above, how long does it take to read the entire relation?
How many recordsd are in the relation?

Unordered Heap Files

Collection of records (no order)

As add/rm records, pages de/allocated

To support record level ops, need to track:
- pages in file
- free space on pages
- records on page

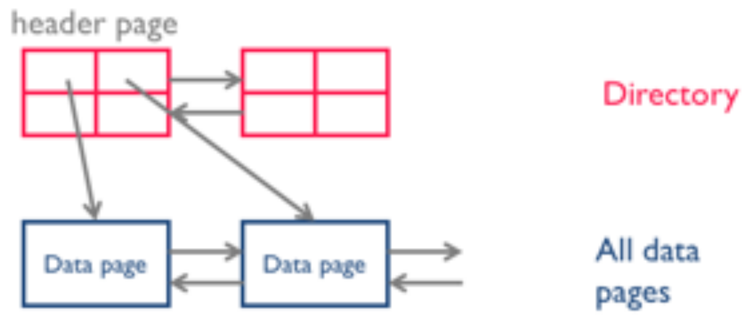


header page (directory) with two doubly linked lists, of full pages, and not full pages
location of header page stored in a database catalog (somewhere special)

what's a pointer on the disk? pointer? no. sector of the track etc? Nope. OS will give us a block number (disk block ID)

mwhat's bad about this? what's this good for?
which pages have how much free space? We don't know. Need to walk through free space linked list
how to find records? will need to scan all of the pages unless we know something more.

Use a directory



Directory entries track #free bytes on data pages

Directory is collection of pages

header pages connected together, a linked list of pointers.
each entry of directory has a pointer to a data page and how much free data, so scan directory instead of data

lots of pointers in a header page, so should be pretty small
usually good enough if using this approach

Administrivia

Project 1 evaluations this week
(very important for your grade)

HW3 has been out

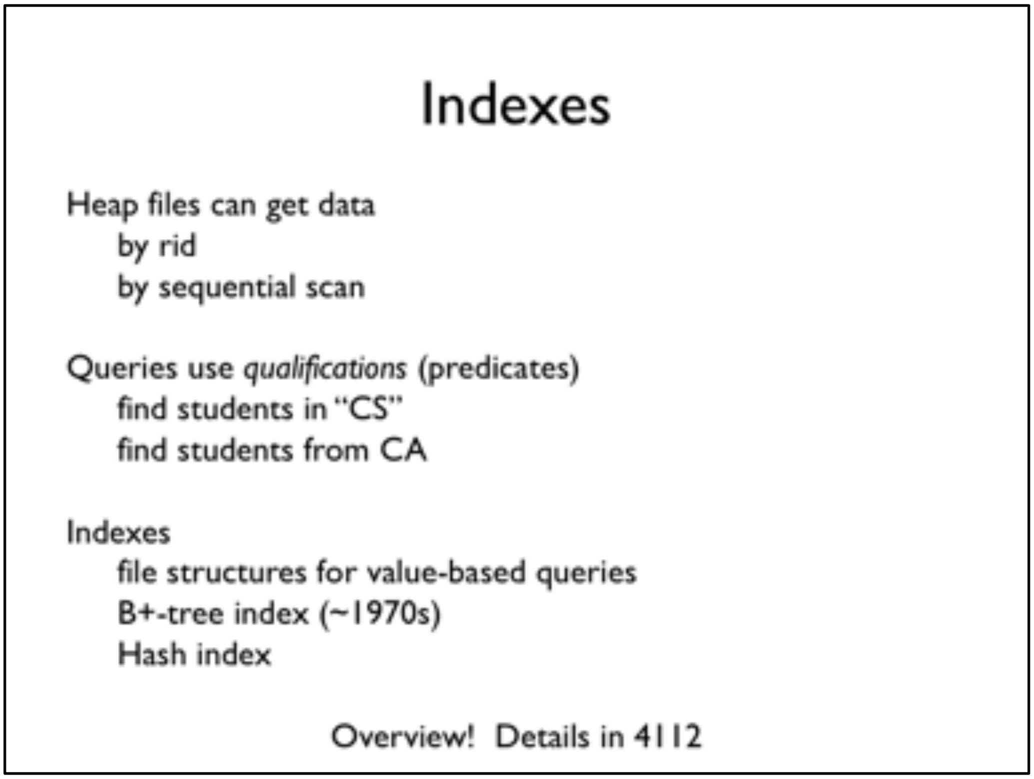
Project 2 destined to be out on Wednesday

HW4 next Monday

Indexes

"If I had eight hours to chop down a tree,
I'd spend six sharpening my ax."

Abraham Lincoln



How would we find a record by rid? scan through the linked list until we find it. Expectation is ½ of all data pages

Keep in mind, indexes are designed to make things faster – with tradeoffs about what types of accesses they speed up.
It is common to use up more space to build indices than for the actual data.

In all of this, we'll be setting up to be able to compare the query costs of using each type of access method

Indexes

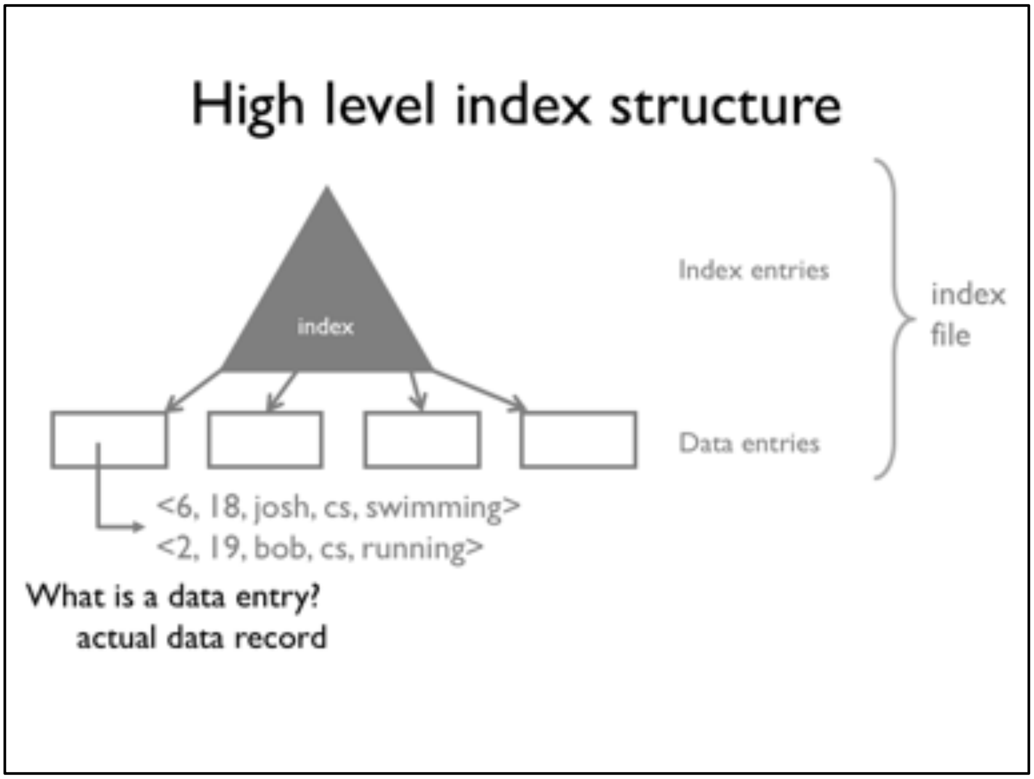
Defined wrt a *search key*

no relation to candidate keys!

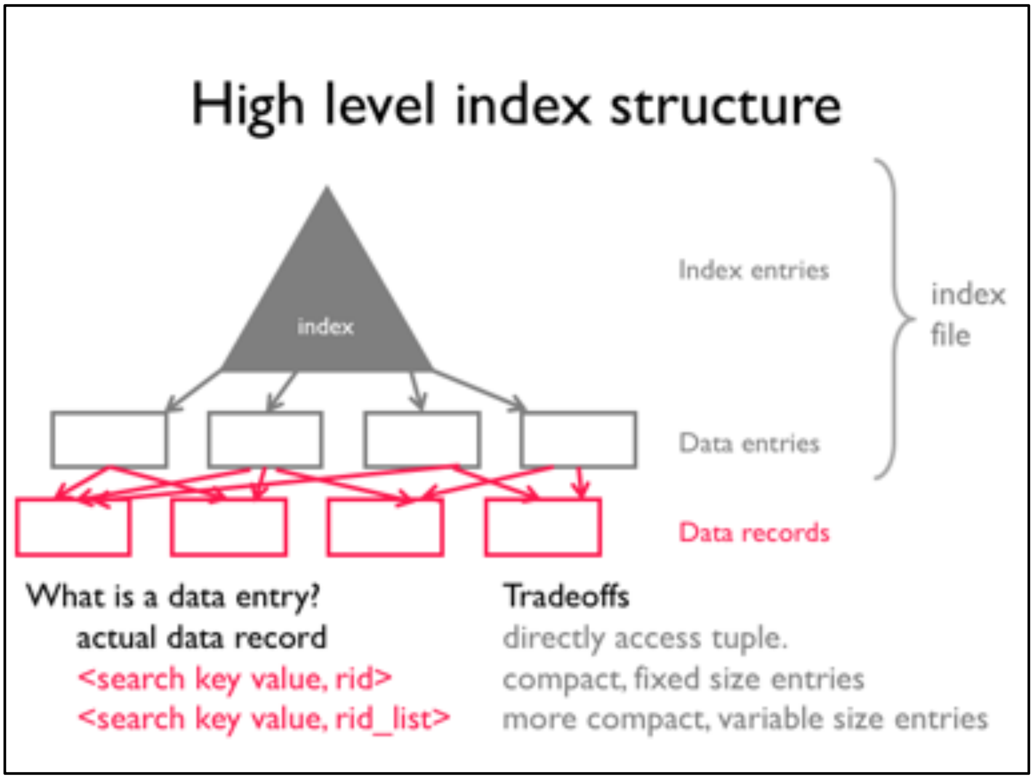
Faster access for WHERE clauses w/ search key

```
CREATE INDEX idx1 ON users USING btree (sid)
CREATE INDEX idx2 ON users USING hash  (sid)
CREATE INDEX idx3 ON users USING btree (age,name)
```

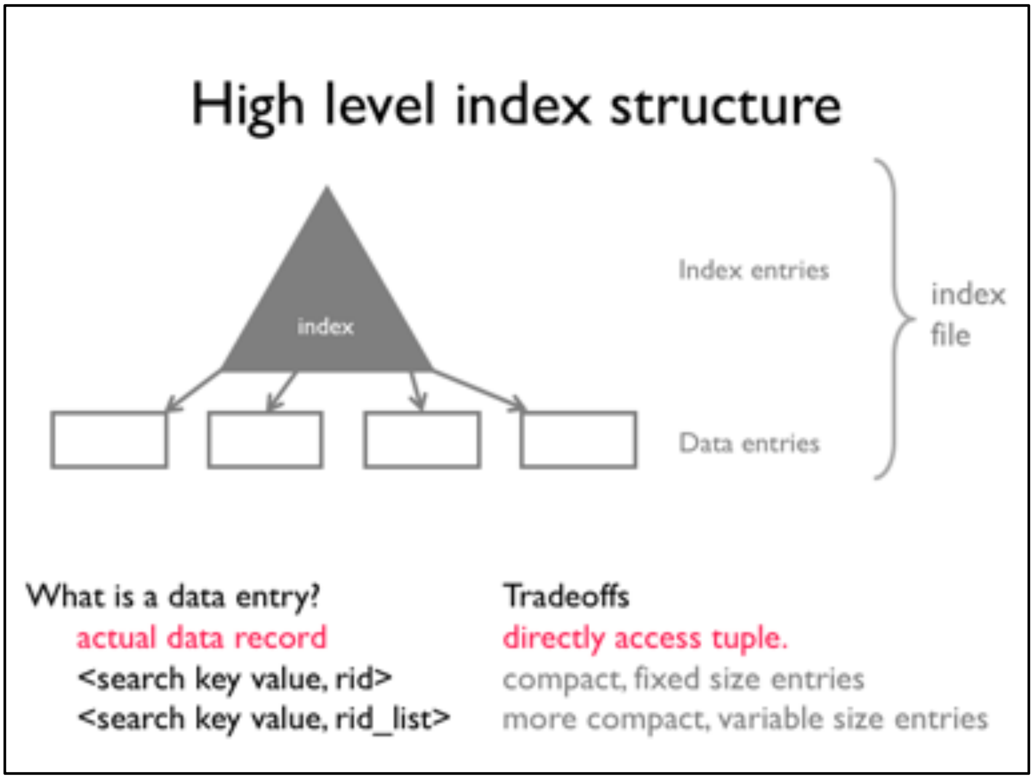
You will play around with indexes in HW4



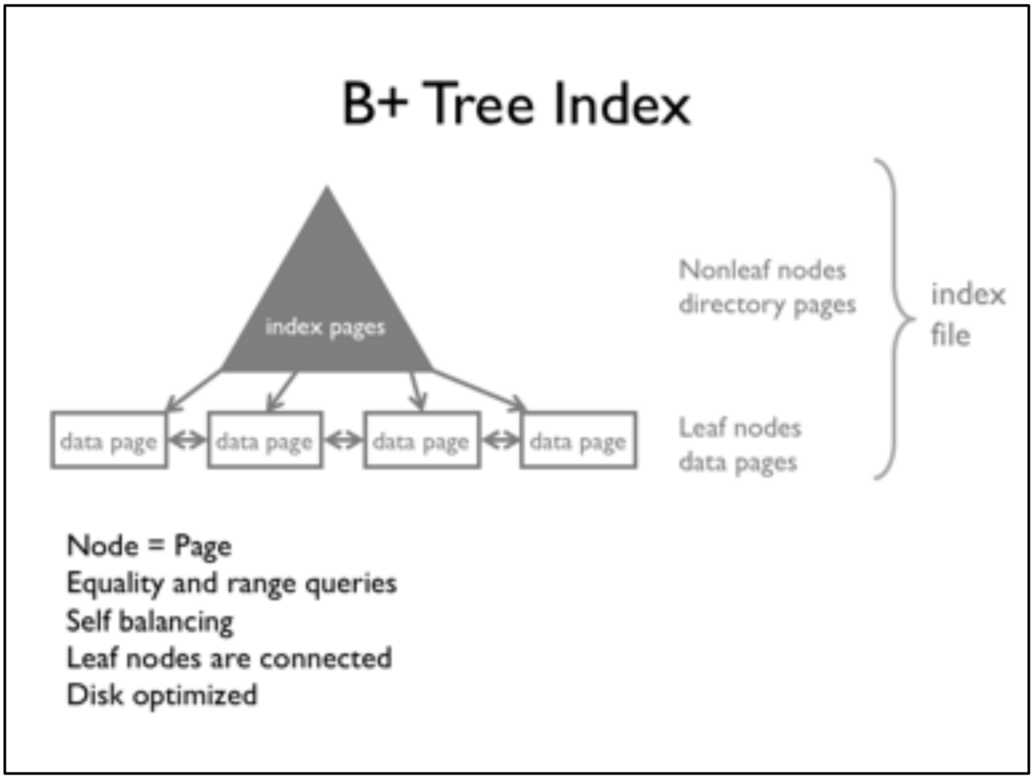
an index is a data structure that organizes data records ON DISK to optimize certain types of retrievals via search keys
given a predicate on a search key, it routes use to one or more data entries



alternatively, instead of storing everything, just store the search key value and a pointer to the record
The red boxes at the bottom is a heap file or some other place where the records are actually stored, and each data entry points to a location in the heap file.



We will store the actual data record in our examples



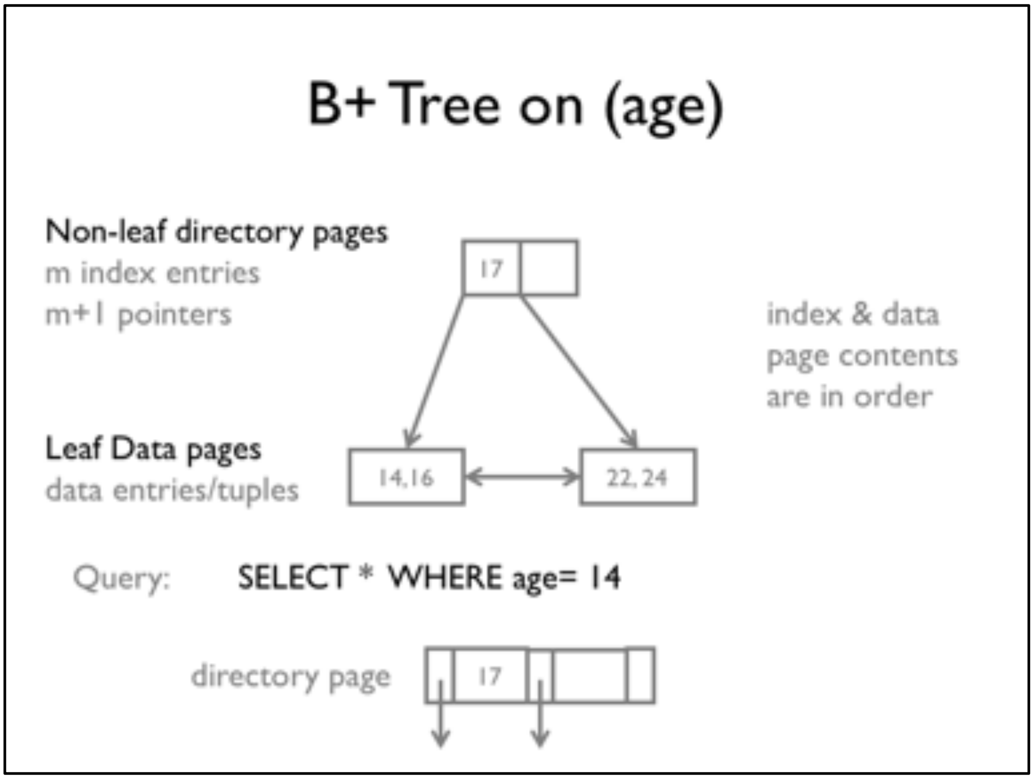
We saw that the directory for the heap file can reduce the cost of certain operations.
- What if we allowed multiple levels of directories?
- And kept them in sorted order on the values?

In contrast to traditional binary search trees, where each node is a single value, B+ tree nodes are pages that contain multiple values. This serves to increase the throughput when reading tree nodes.

In fact, many "key value" stores like mongo and berkeley DB are persistent B+Tree data structures
Workhorse of most DBMSes

Consists of an index structure for directing the search algorithm
along with data entries as the leaf nodes that contain the actual data
(same data pages as in unordered heap files)
This entire structure is composed of pages – index heap file

Terminology:
height: is wrt the index entires. So a tree of height 1 includes a single root node and data entries



Example of a b+ tree indexed on val (val is the *index key)*

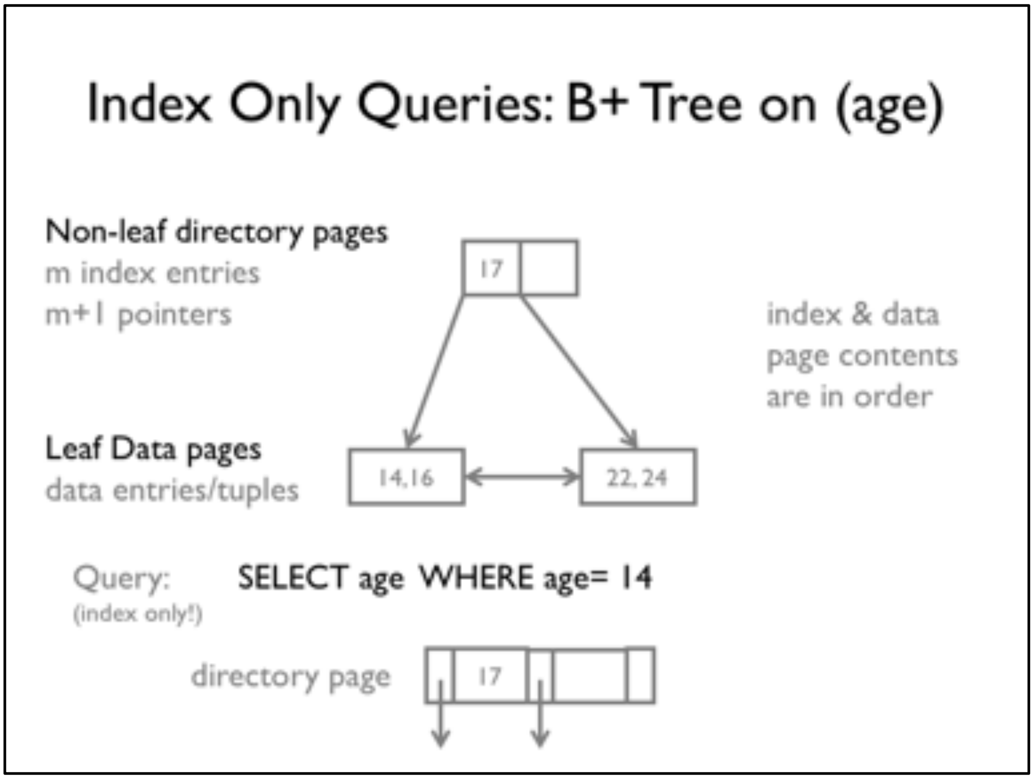
Each non-leaf node is like a directory:

- sorted list of values.
- index entry = age value to compare against when searching the tree
- the left and right side of each non-null index entry are pointers to the child nodes

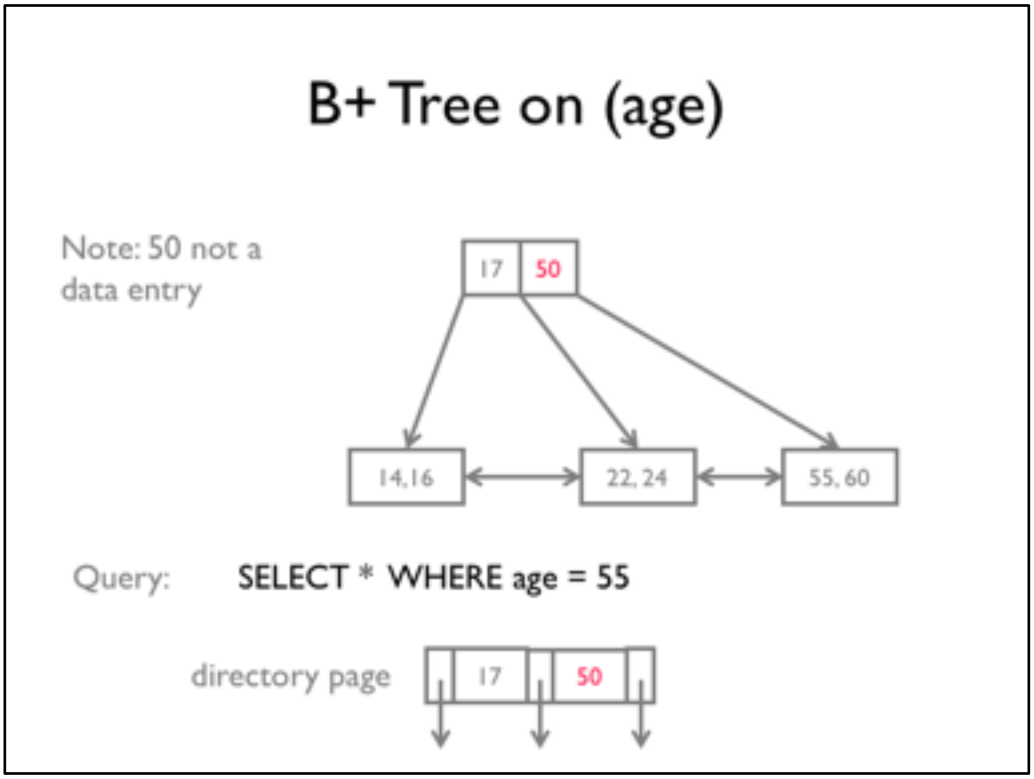
Here, I only show the age value of the tuples stored in the data pages

Unlike a binary tree, interior nodes are only a directory – don't store data. data is all in the leaves

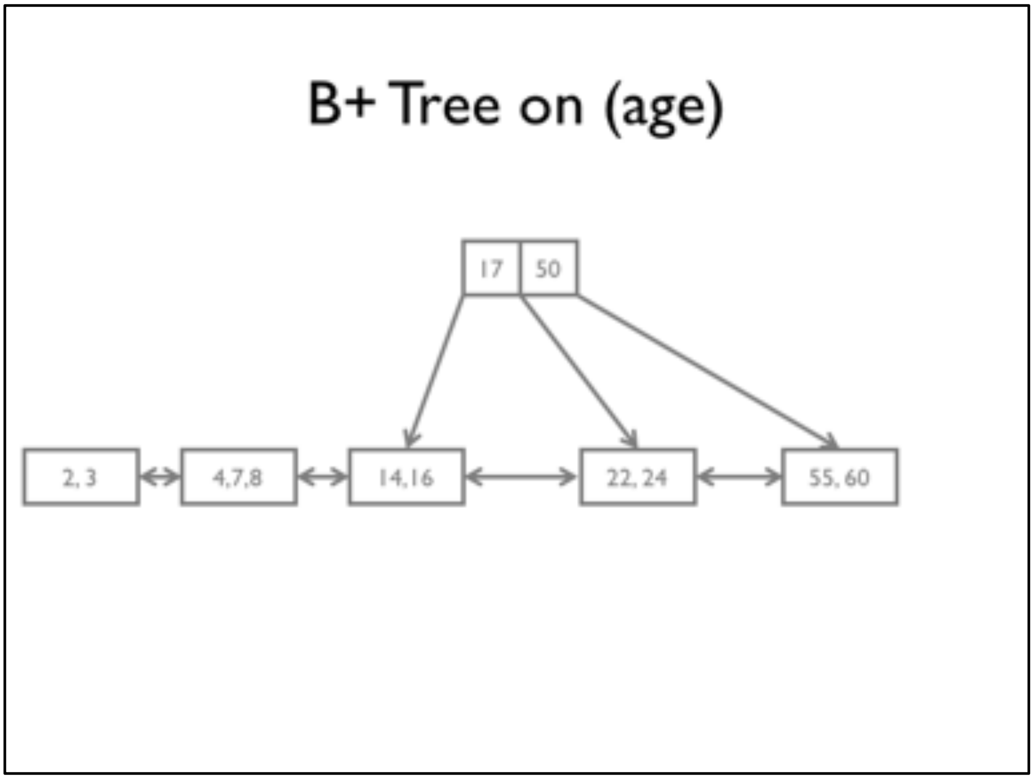
How to search? Load index page and do binary search. Can do it since loaded in memory.



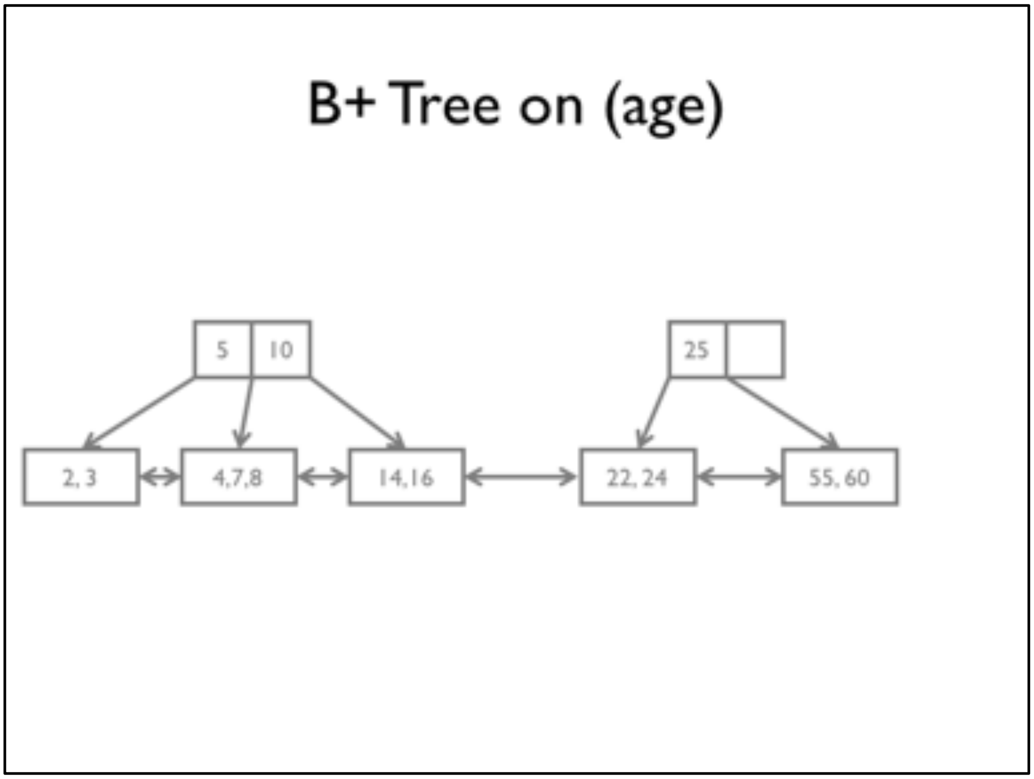
Note that if the data entries are <age value, rid>, then a query that projects the search key can be *index only*



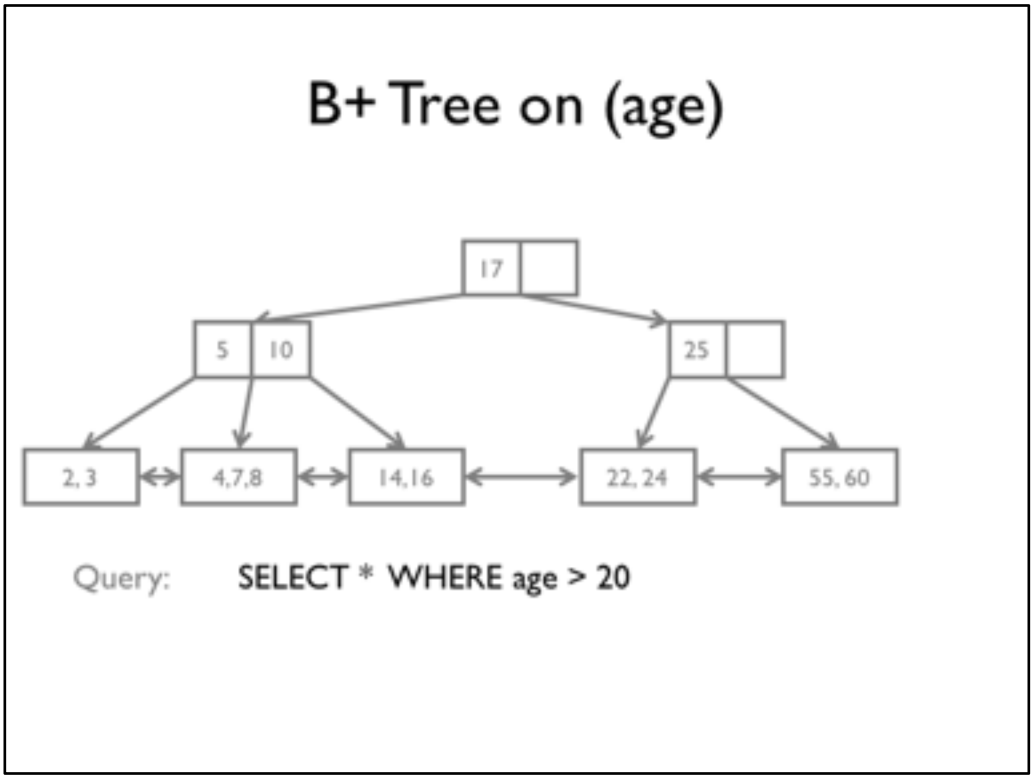
Unlike a binary tree, leaves are only a directory – don't store data. data is all in the leaves



If we add more data, let's say we have 2 additional pages of data, then this directory page is full and we can't add more pointers, and so we need to split it up in order to index the new pages



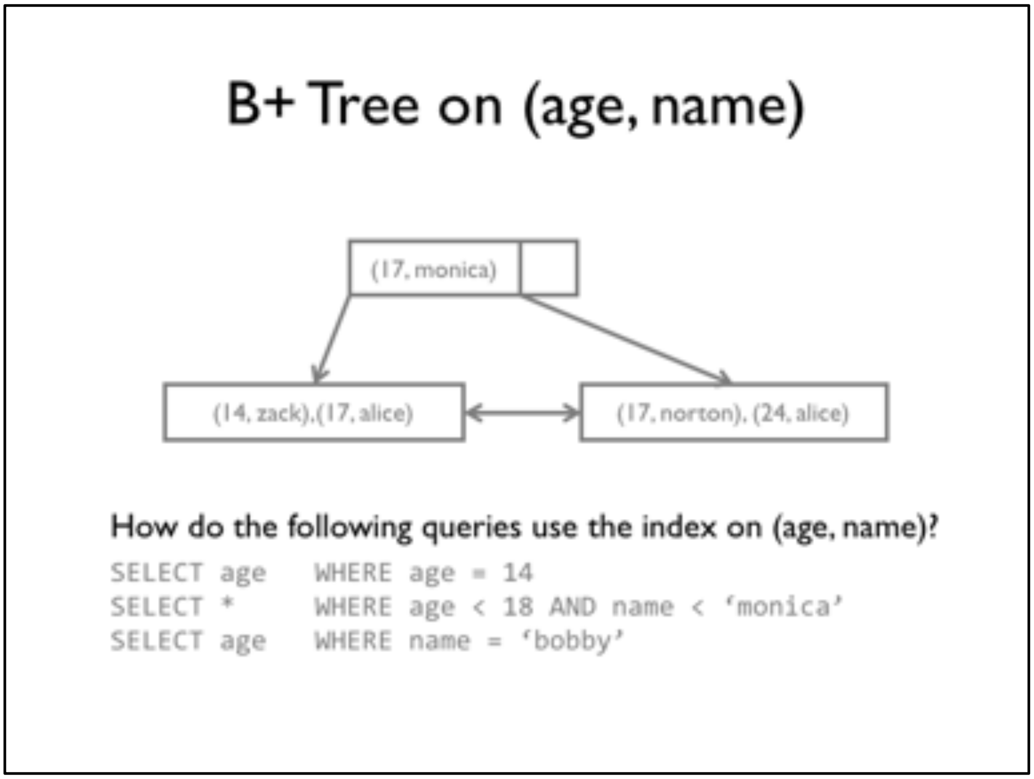
We might spit it up to index the pages this way, but this is not a tree notice that the index entry 50 has disappeared. This is because the indexentries don't contain data.



The details of how we do this don't matter, but it's self balancing, so any combination of inserts, updates, and deletes end up with a balanced tree (meaning the left and right children are roughly the same amount of data

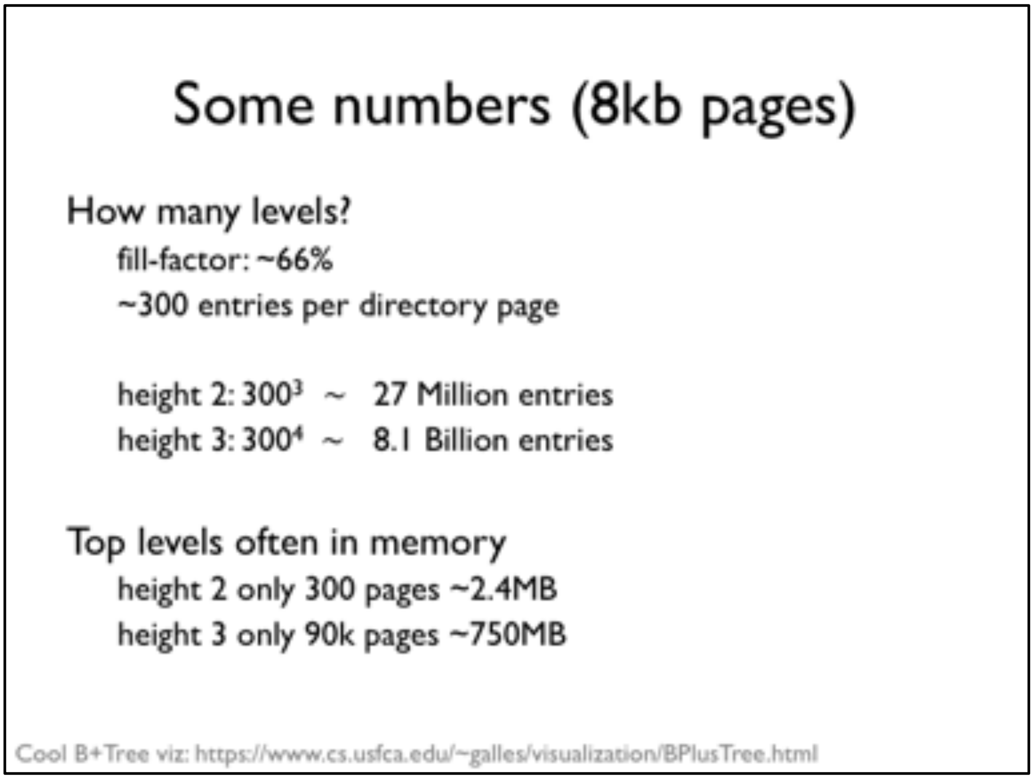
typically hundred(s) of items in a page

What's a benefit of the doubly linked list at the bottom of the b+tree?
It supports range queries as well. Here we go to the page with 20 (or smallest number larger than 20) and scan along the leaf pages



Composite key
Note that (17, alice) < (17, monica) even though both have 17.

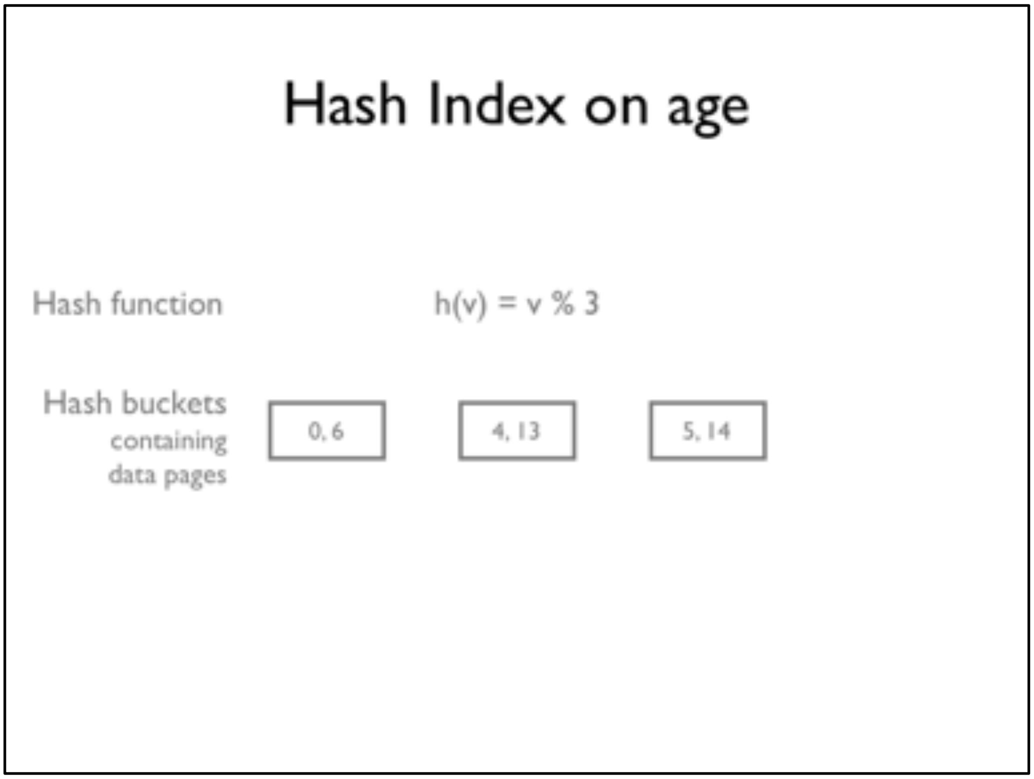
Q1: index only, use first part of the index key



8 kb pages, integer entries and integer pointers (8 + 8 bytes) = 500 entries in a directory
60% fill factor is 300 entires

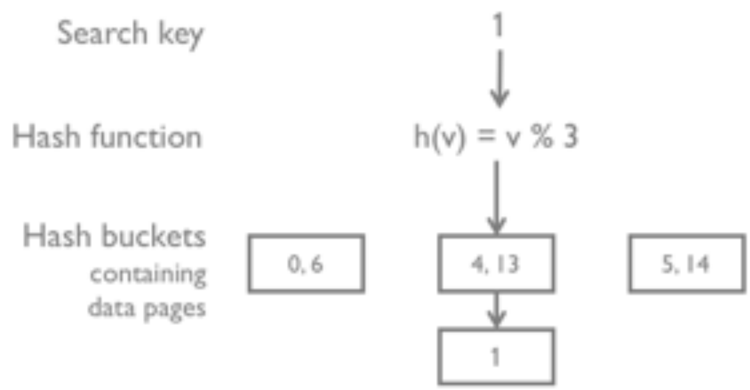
Recall that we the DBMS allocates a big chunk of memory to use for itself.
Given some standard caching policy, what are the changes the root node will be in memory?
Well it's accessed on every lookup, so it's likely in memory
What about the next level? Doesn't take much space, and probably accessed frequently, so also in memory.
So for a 27M entry table, only 1 IO to access the data page. Pretty good!
Logarithmic data structures are good when the constants are large (e.g., fanout)

Height = length of path from root to leaf
when height = 2, why do we use 300^3? recall height is just the index entry pages
level 3 == root (level 1) -> level 2 -> level 3 -> data entry



hash indexes: single or multiple hash functions. Array of data pages. compute hash using hash function to get data page and insert into it. If data page is full, add an overflow page

INSERT Hash Index on age



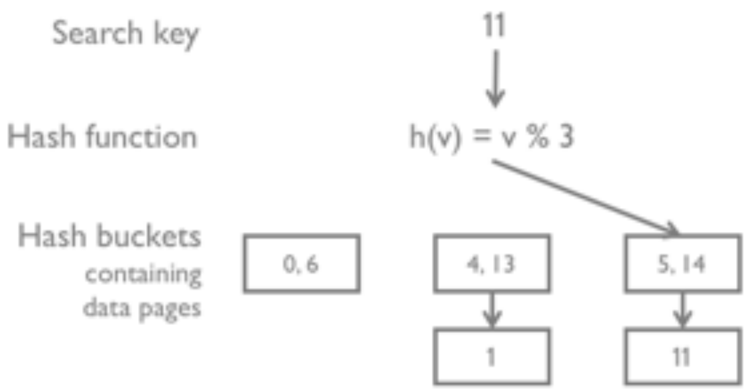
INSERT Hash Index on age

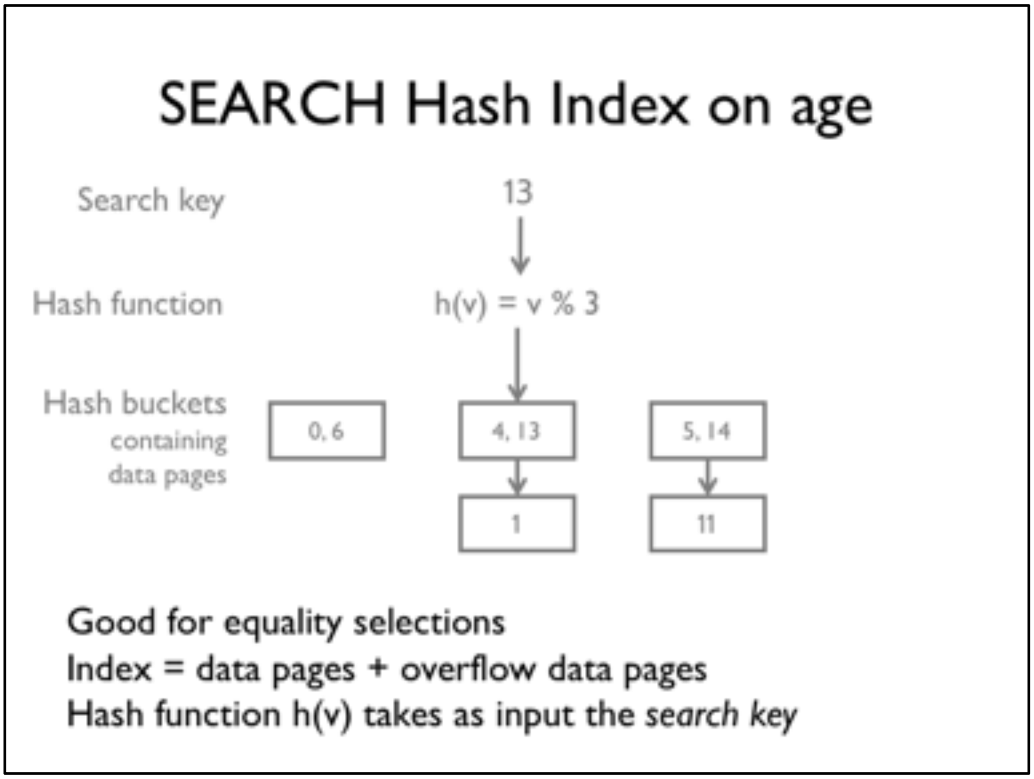
Search key 11

Hash function $h(v) = v \% 3$

Hash buckets containing data pages

INSERT Hash Index on age





In terms of metadata, how is this different than B+ trees?
What types of queries is this good for compared to B+ trees?

Costs

Three file types

- Heap, B+ Tree, Hash

Operations we care about

Scan all data	SELECT * FROM R
Equality	SELECT * FROM R WHERE x = 1
Range	SELECT * FROM R WHERE x > 10 and x < 50
Insert record	
Delete record	

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything				
Equality				
Range				
Insert				
Delete				

B # *data* pages
D time to read/write page
M # pages in range query

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD			
Equality	0.5BD			
Range	BD			
Insert	2D			
Delete	Search + D			

Heap File

equality on a key. How many results?

B # *data* pages
D time to read/write page
M # pages in range query

B: total number of data pages in table
M: if doing a range query, we are fetching M pages

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD		
Equality	0.5BD	$D(\log_2 B)$		
Range	BD	$D(\log_2 B + M)$		
Insert	2D	Search + BD		
Delete	Search + D	Search + BD		

Heap File

equality on a key. How many results?

Sorted File

files compacted after deletion

B # *data* pages

D time to read/write page

M # pages in range query

we assume that the heap is sorted on the query predicate attribute, otherwise it's as good as an unordered heap

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD	1.2BD	
Equality	0.5BD	$D(\log_2 B)$	$D(\log_{80} B + 1)$	
Range	BD	$D(\log_2 B + M)$	$D(\log_{80} B + M)$	
Insert	2D	Search + BD	$D(\log_{80} B)$	
Delete	Search + D	Search + BD	$D(\log_{80} B)$	

Heap File

equality on a key. How many results?

Sorted File

files compacted after deletion

B+ Tree

100 entries/directory page

80% fill factor

B # *data* pages

D time to read/write page

M # pages in range query

why does scanning take 1.2BD? (see the assumptions for B+Tree in the slide)

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD	1.2BD	1.2BD
Equality	0.5BD	$D(\log_2 B)$	$D(\log_{80} B + 1)$	D
Range	BD	$D(\log_2 B + M)$	$D(\log_{80} B + M)$	1.2BD
Insert	2D	Search + BD	$D(\log_{80} B)$	2D
Delete	Search + D	Search + BD	$D(\log_{80} B)$	2D

Heap File

equality on a key. How many results?

Sorted File

files compacted after deletion

B+ Tree

100 entries/directory page

80% fill factor

Hash index

no overflow

80% fill factor

B # *data* pages

D time to read/write page

M # pages in range query

can you even perform range query with a hash index?

why is hash 1.2BD for range query? don't know the exact domain of the values!

$1 < x < 10$

try 1, 2, 3, 4, … 10?

what about 1.5?

How to pick?

Depends on your queries (workload)

- Which relations?
- Which attributes?
- Which types of predicates (=, <,>)
- *Selectivity*
- Insert/delete/update queries? how many?

selectinivy
- why wouldn't you use hash index for a range query?
- what is equality but selectivity?
- 0.1 selectivity = will return ~10% of the tuples

if all of your queries are inserts, then a heap file may make the most sense
if all of your queries are primary key equality accesses, then hash table may be a g
ood idea

How to choose indexes?

Considerations

- which relations should have indexes?
- on what attributes?
- how many indexes?
- what type of index (hash/tree)?

called Physical database design problem

which relations are we accessing? Are they already fast? Or are they slow and an index would help?
(amount of improvement to queries on relation)
attributes: recall that b+tree or hash depend on the search key
Composite search key or single attribute search key?

Naïve Algorithm

```
get query workload
group queries by type
for each query type in order of importance
   calculate best cost using current indexes
   if new index IDX will further reduce cost
      create IDX
```

Why not create every index?

- update queries slowed down (upkeep costs)
- takes up space

workload

in many databases, the index sizes can often be much much larger than the actualy data, so that queries go faster.
What if you don’t use update queries?

High level guidelines

Check the WHERE clauses

- attributes in WHERE are search/index keys
- equality predicate → hash index
- range predicate → tree index

Multi-attribute search keys supported

- order of attributes matters for range queries
- may enable queries that don't look at data pages (*index-only*)

didn't talk about index-only

Summary

Design depends on economics, access cost ratios
Disk still dominant wrt cost/capacity ratio
Many physical layouts for files
- same APIs, difference performance
- remember physical independence

Indexes
- Structures to speed up read queries
- Multiple indexes possible
- Decision depends on workload

Things to Know

- How a hard drive works and its major performance characteristics
- The storage hierarchy and rough performance differences between RAM, SSD, Hard drives
- What files, pages, and records are, and how they are different than the UNIX model
- Heap File data structure
- B+ tree and Hash indexes
- Performance characteristics of different file organizations