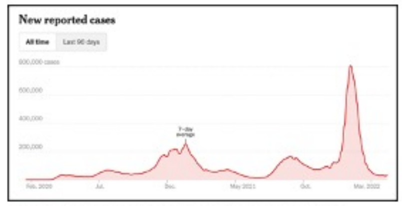
Help with Visualization Research!

MS and Undergrad student research, took Intro to Vis in the past. Hoping to submit to the VIS conference in a few weeks.

User study to understand what people remember when reading visualizations with captions

If interested in helping, take their survey!
https://www.surveymonkey.com/r/SD99FBR



United States has more than 800,000 new daily COVID cases in mid-January

Administrivia

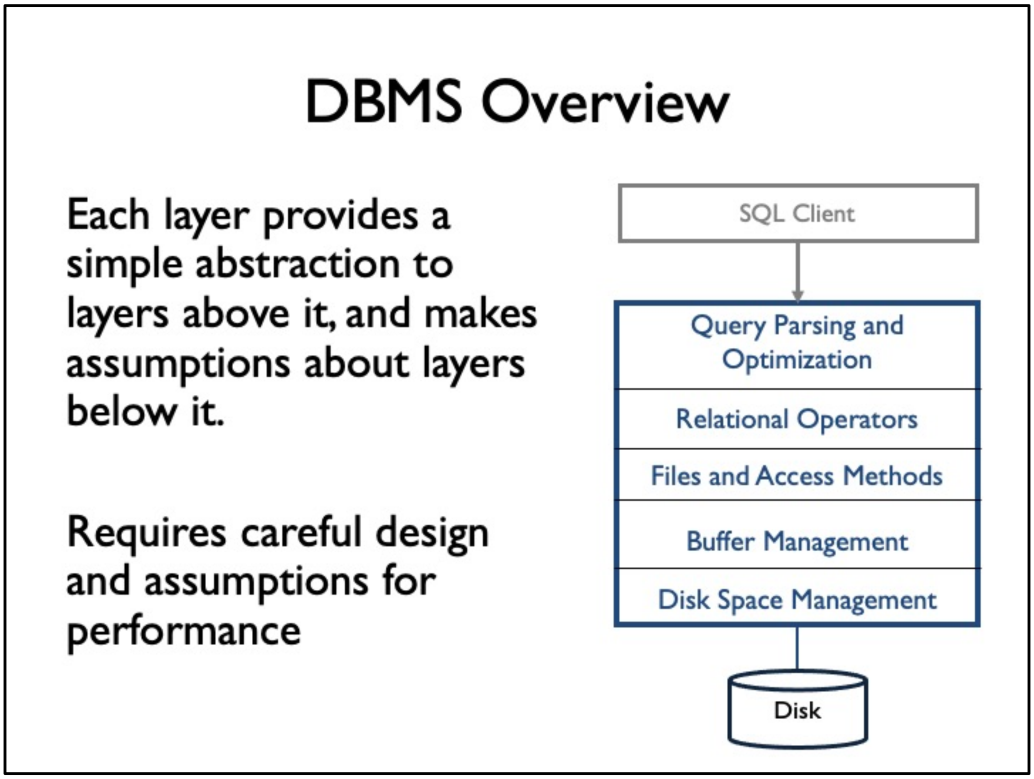
HW3 due

HW4 out: access costs and optimizer

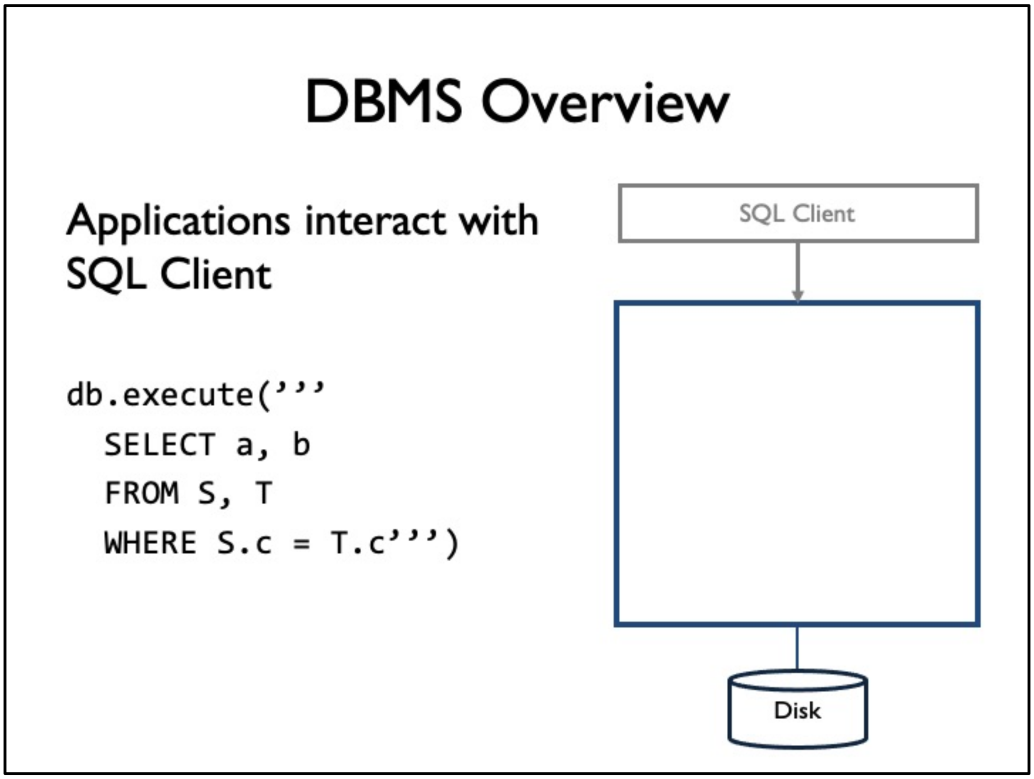
Project 2 out: more SQL!

L8
Disk, Storage, and Indexing

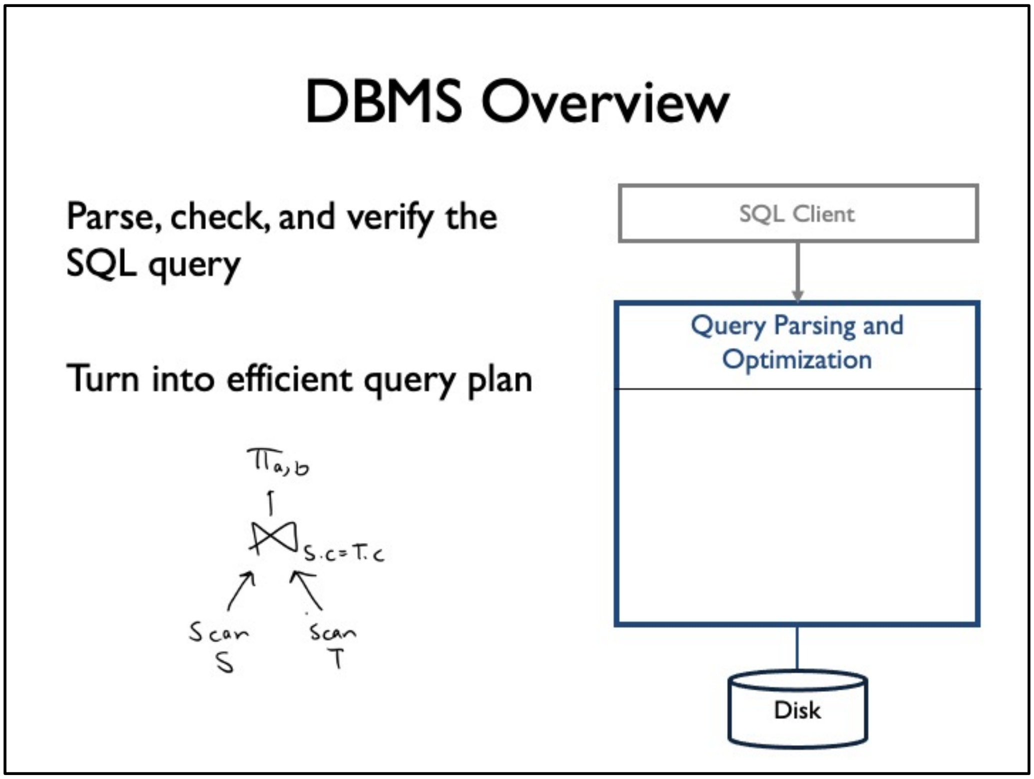
Eugene Wu



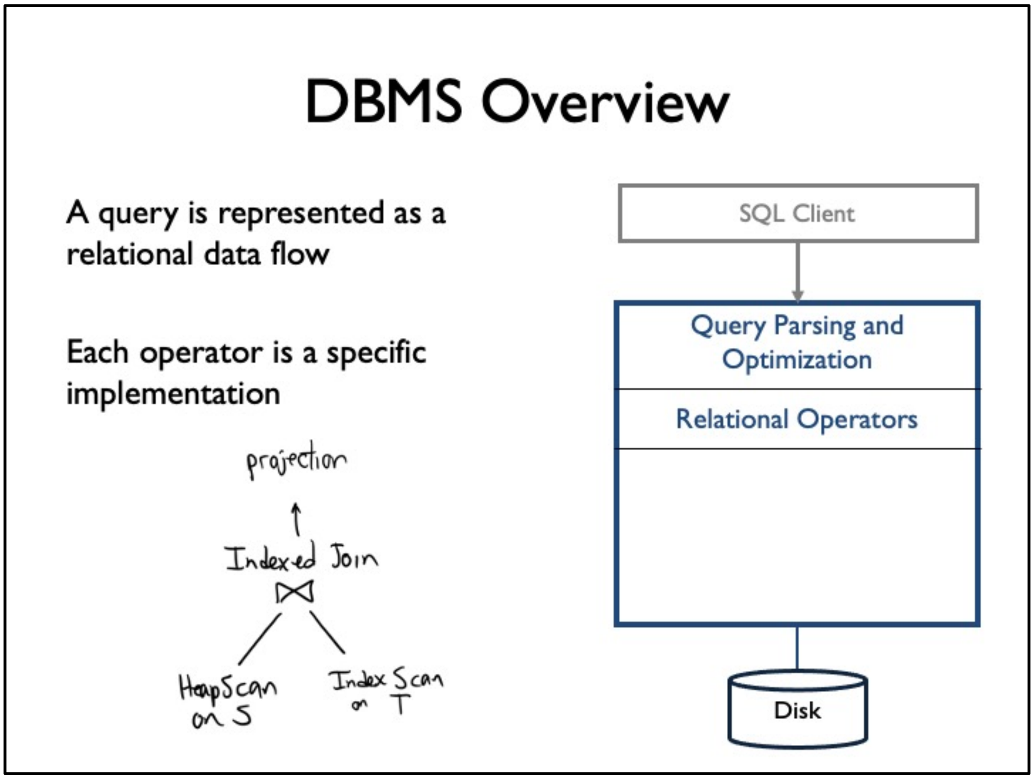
Classic architecture. Most large systems from Oracle, IBM etc use this. Optimized for disk.
Newer systems fro new hardware may change the organization, or tweak components, hwever these concepts are still there.



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.

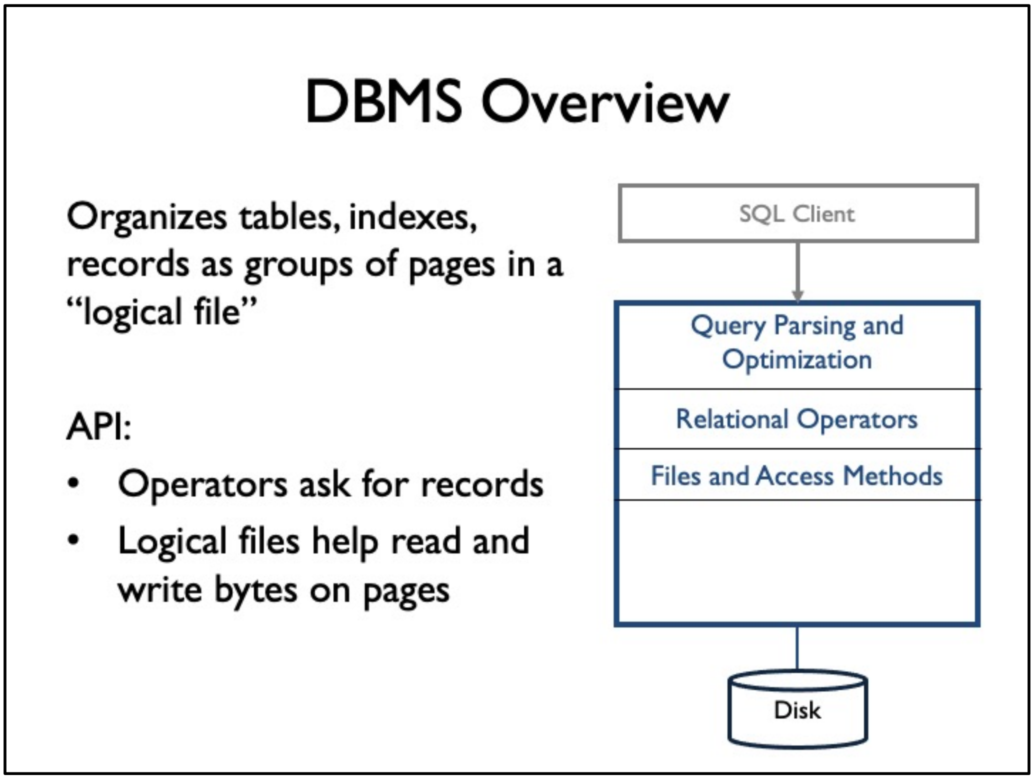


Classic architecture. Most large systems from Oracle, IBM etc use this. Optimized for disk.
Newer systems fro new hardware may change the organization, or tweak components, hwever these concepts are still there.



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.

What is a heapscan reading from? What about index scan?



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.

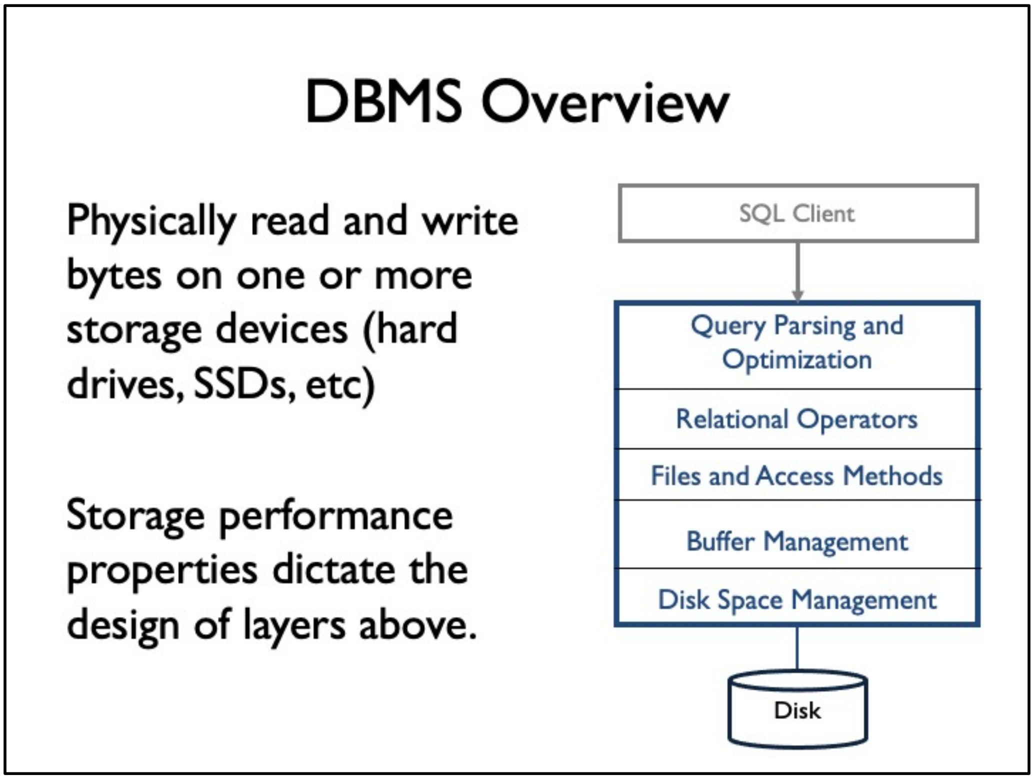
DBMS Overview

Not all pages can fit into RAM.

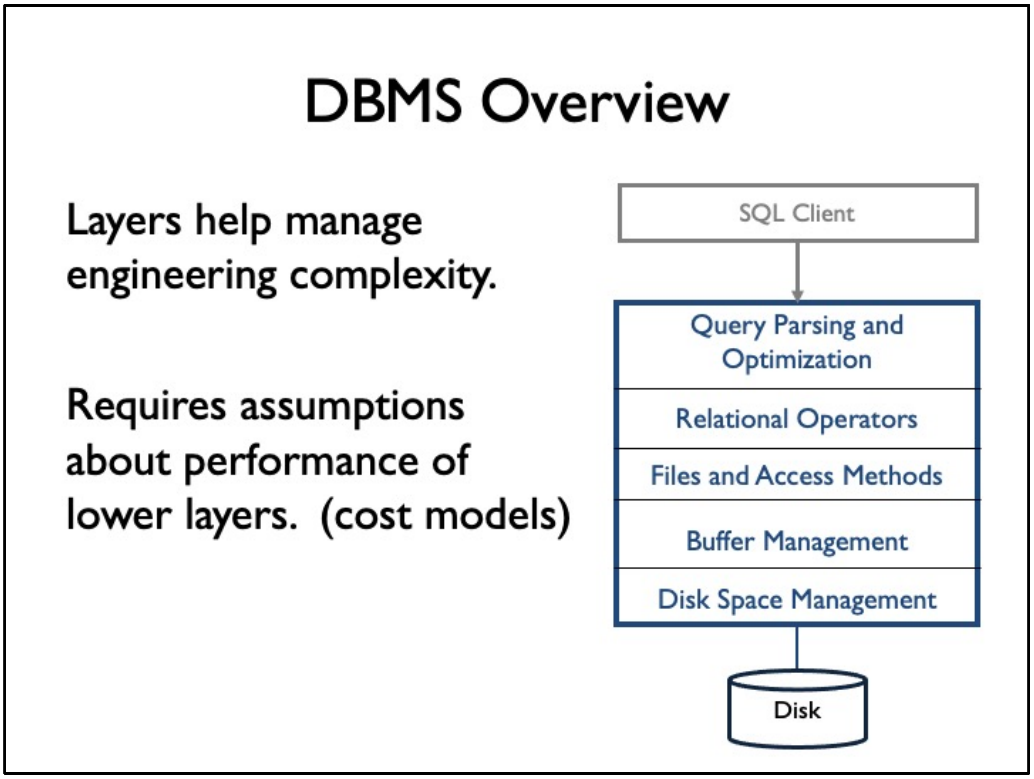
Buffer manager provides illusion that all pages are accessible.

Files simply ask for pages.

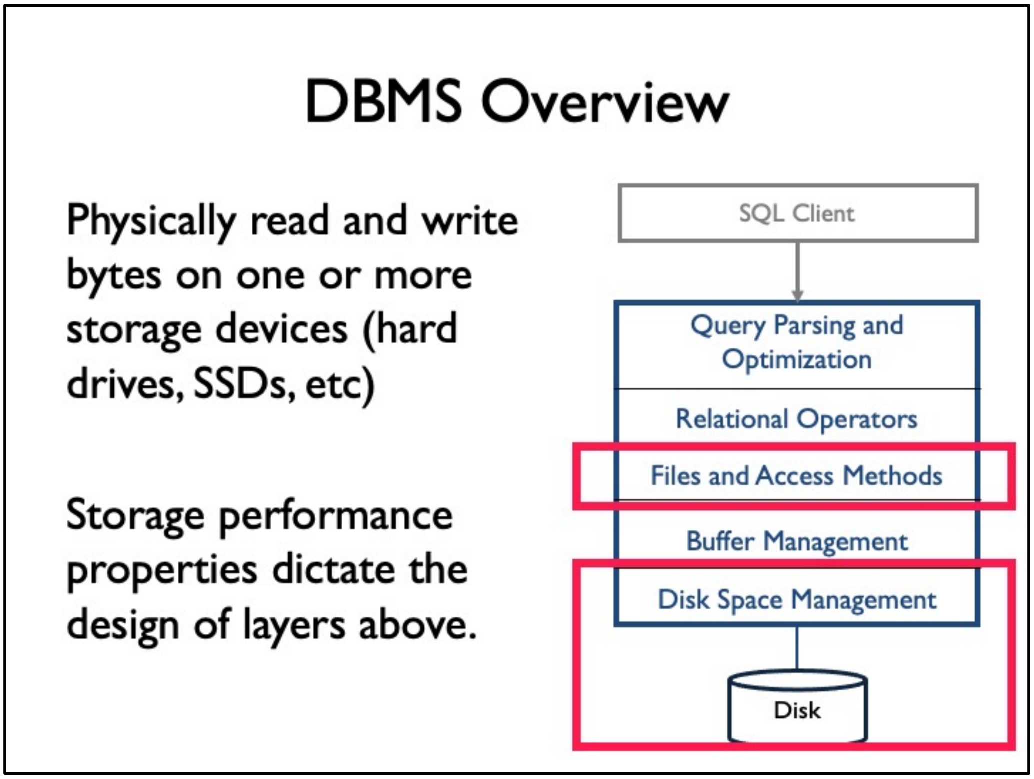
Classic architecture. Most large systems from Oracle, IBM etc use this. Optimized for disk.
Newer systems fro new hardware may change the organization, or tweak components, hwever these concepts are still there.



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.



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.



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.

Storage Devices

Hard Drives vs SSDs vs RAM

Locality (random vs sequential)

Reads vs Writes

Performance and Monetary Costs

Why not store all in RAM? $$$

Too much **$$$**

- High-end DBs: ~Petabyte (1000TB).
- SQL Hyperscale: 100TB+ (2018)
- Disks are ~60% cost of a production system

Main memory not persistent

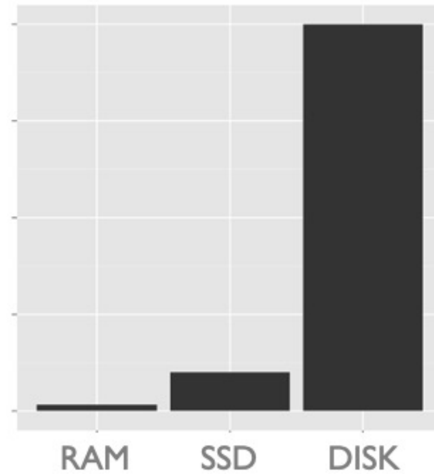
- Obviously important if DB stops/crashes

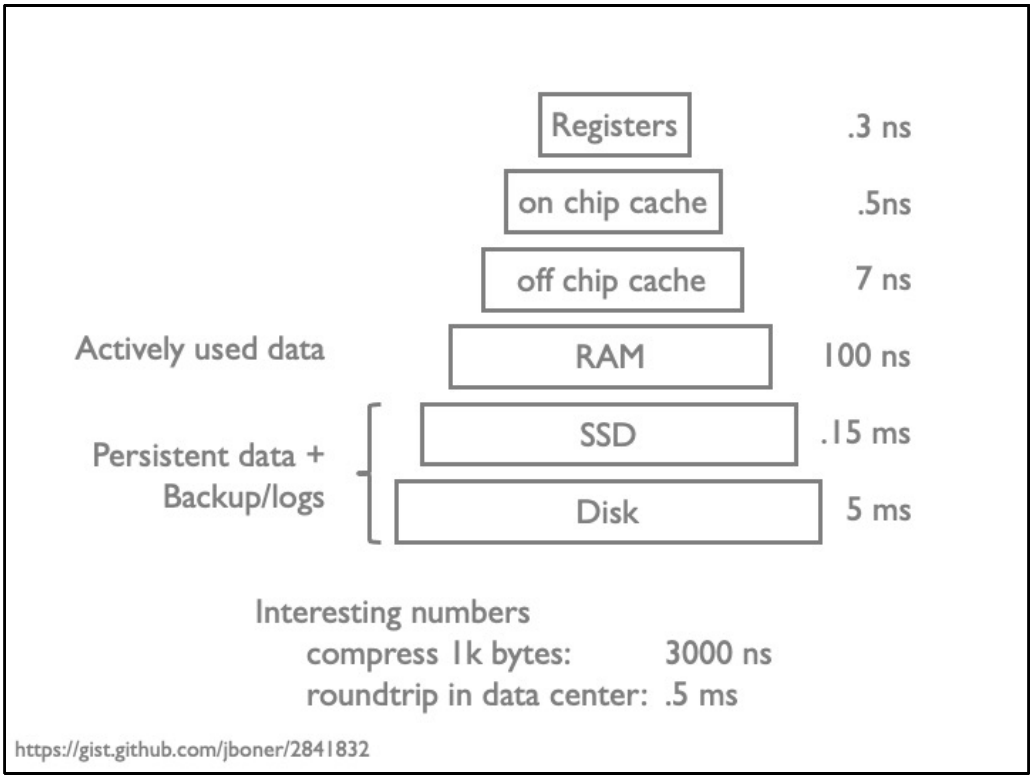
main-memory DBMSes discussed in advanced DB course

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

$ Matters

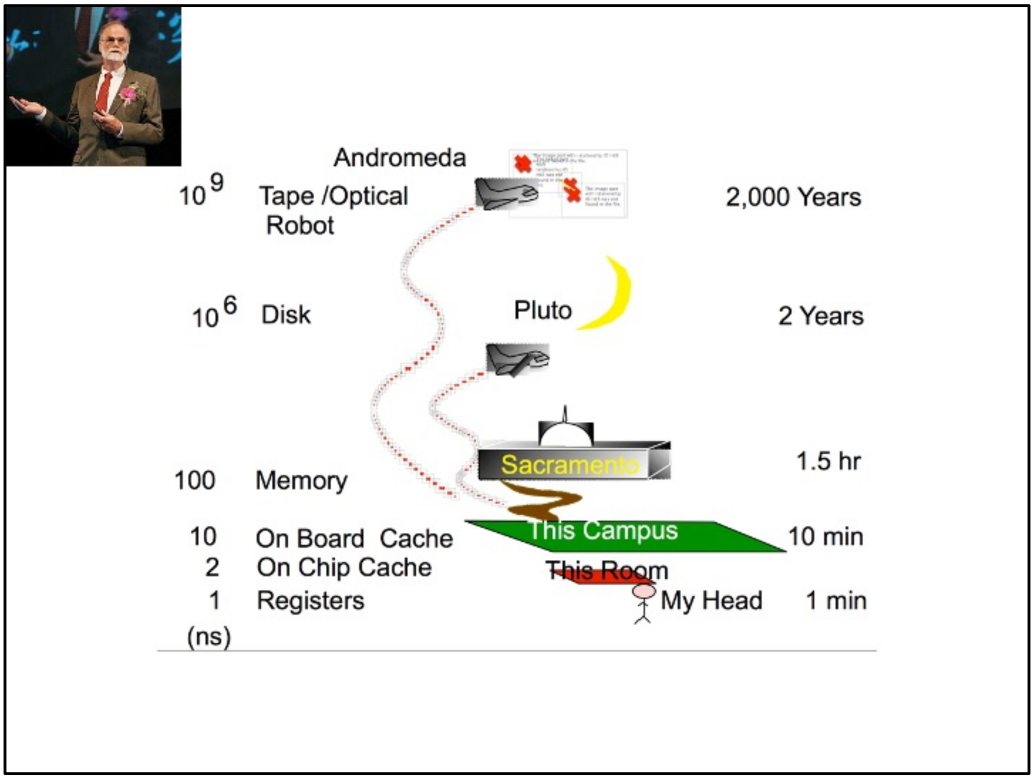
Newegg enterprise $1000
- RAM: 0.08TB
- SSD: ~2TB (25x)
- Disk: ~40TB (500x)





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]

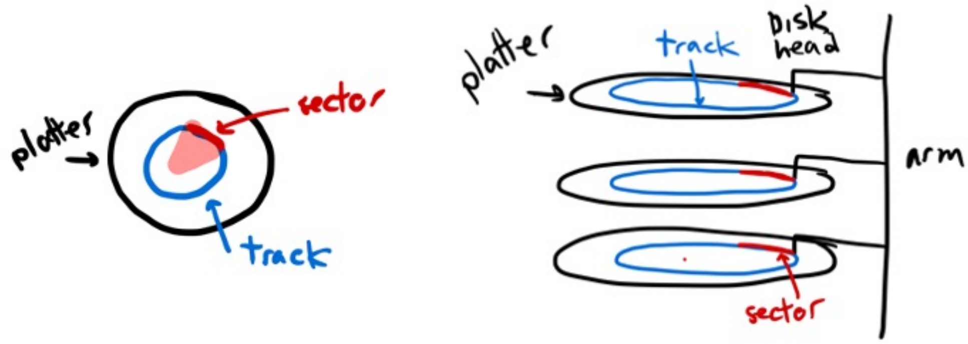
go read his wikipedia page
https://en.m.wikipedia.org/wiki/Jim_Gray_(computer_scientist)

Spin speed: ~7200-15K RPM.   10K RPM ~ 6ms/rotation

Sector: multiples of pages.

More sectors further from center

Terminology: block = page in this class



https://www.youtube.com/watch?v=BIB49F6ExkQ

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?

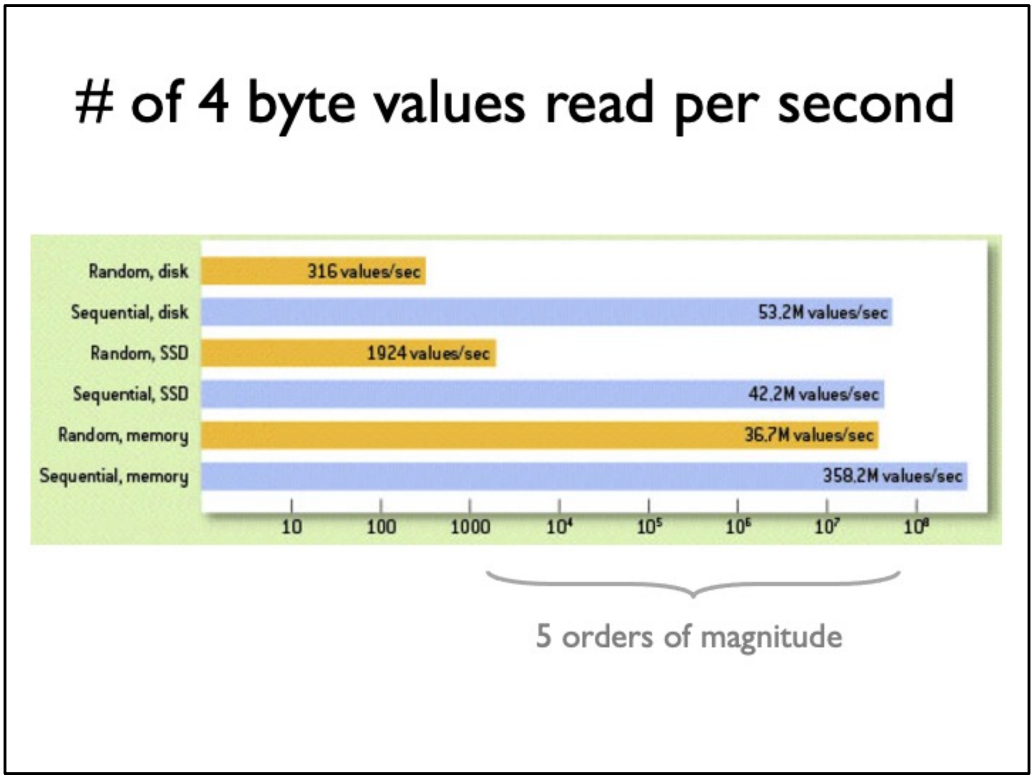
Time to access (read or write) a disk block

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

Throughput

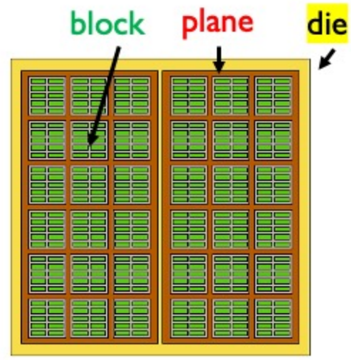
read	~150 MB/sec
write	~50 MB/sec

Key: reduce seek and rotational delays
- HW & SW approaches



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

SSDs



NAND memory

Small reads: 4-8k

Big writes: 1-2MB

2-3k writes before failure

wear leveling: distribute writes around

Write amplification: changing 4 bytes writes a 1-2MB chunk!

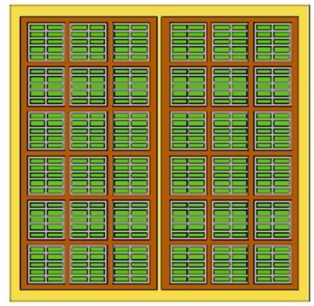
Need to think about wear, garbage collection, writing in bulk

https://www.anandtech.com/show/2738/5

SSDs

Reads fast

- single read time: 0.03ms
- random reads: 500MB/s
- sequential reads: 525MB/s



Writes less predictable

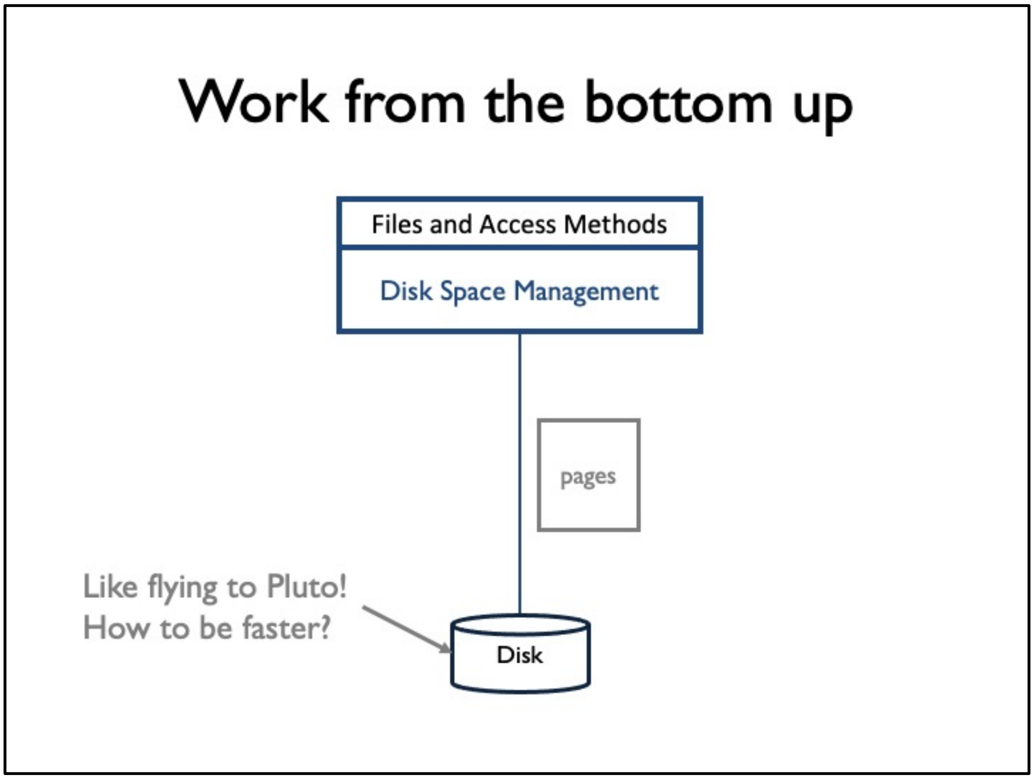
- single write time: 0.03ms
- random writes: 120MB/s
- sequential writes: 480MB/s

What's Best? Depends on Application

Small databases: SSD/RAM
- All global daily weather since 1929: 20GB
- 2000 US Census: 200GB
- 2009 english wikipedia: 14GB
- Easily fits on an SSD or in RAM

Very Big databases: Disk
- Sensors easily generate TBs of data/day
- Boeing 787 generates ½ TB per flight
- Disk has best cost-capacity ratio
- SSDs help reduce read variance

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



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.

Strategies for Fast Data Access
(think about going to the store)

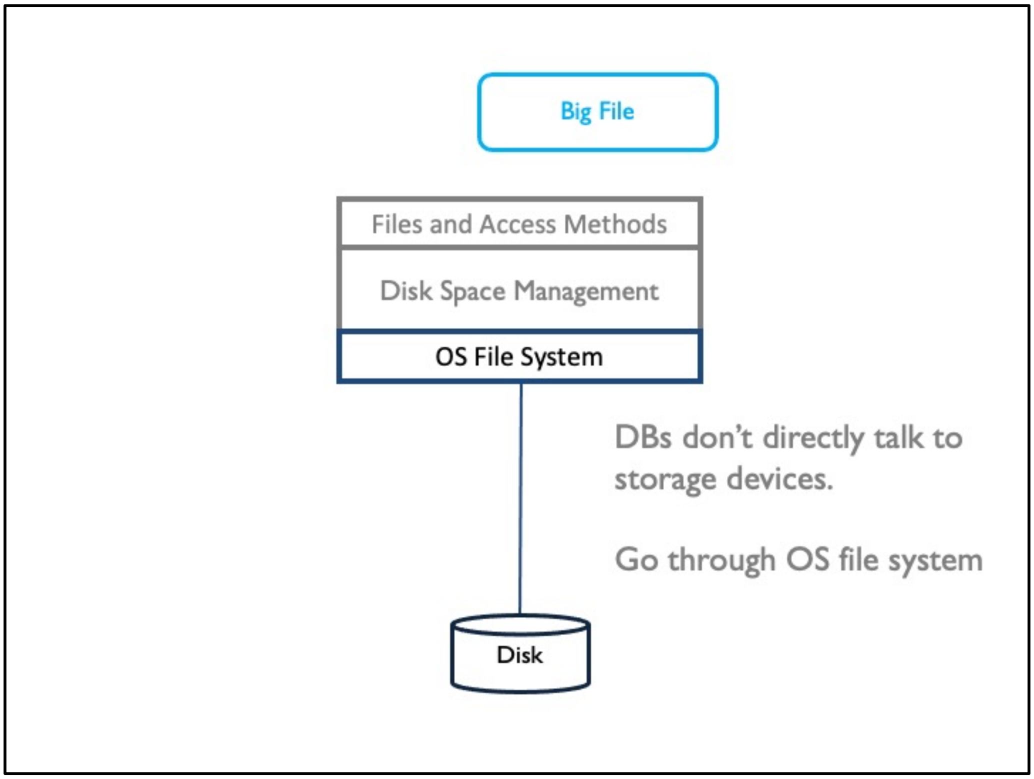
Big difference between random & sequential access

- Optimize for sequential accesses

Amortize sequentially read & write big chunks of bytes
Cache popular blocks
Pre-fetch what you will need later

API

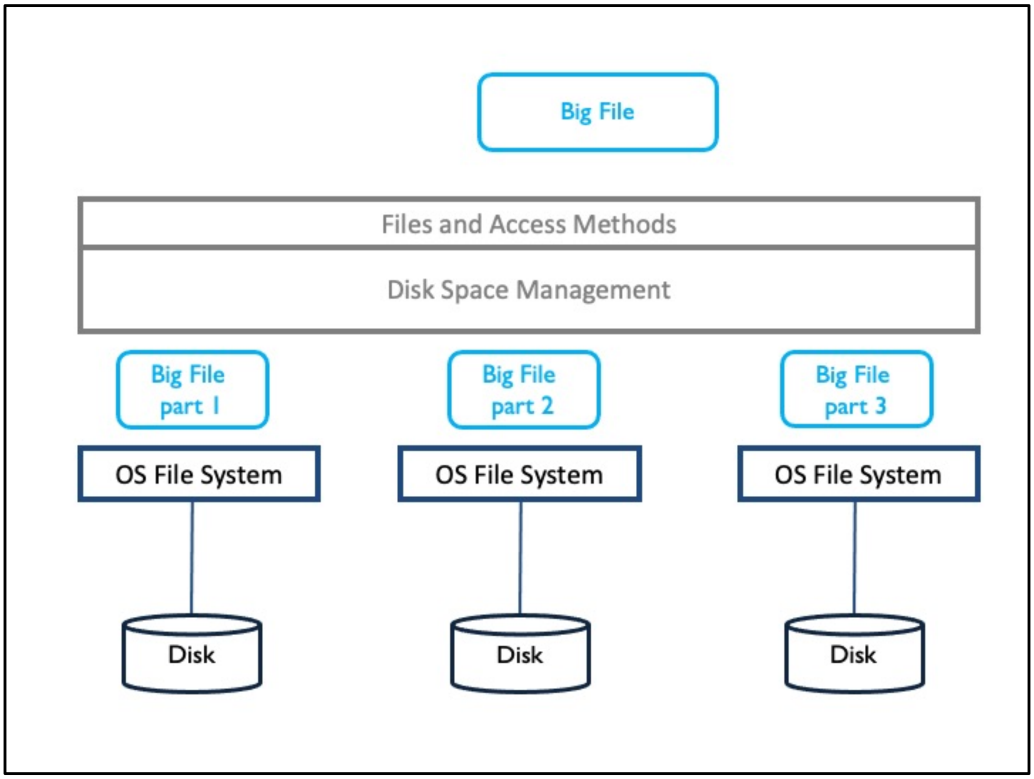
- read/write page
- read/write sequential pages
- notion of "next" page (upper layers can assume next is faster)



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

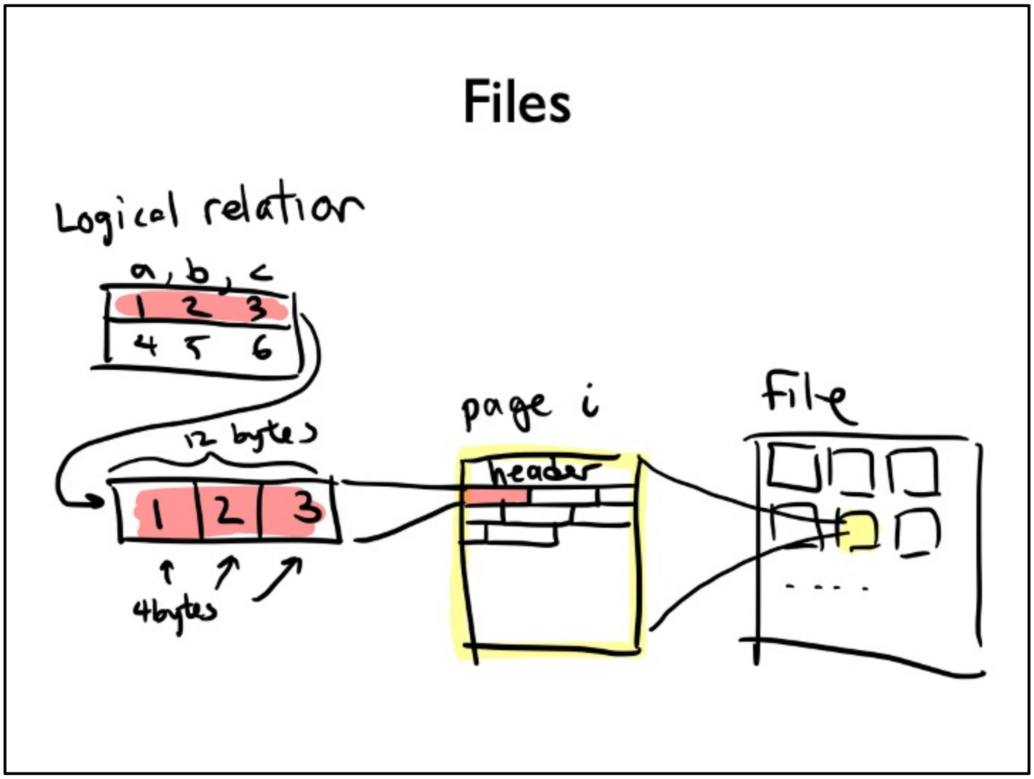
our operations will be at the level of pages



This abstraction is nice, because we are not tied to a single file system. Can partition data across multiple file systems as well, and none of our files and access methods and higher layers in the DBMS need to care.

PostgreSQL database stored in files

```
ewu@dyn-160-39-248-197 ~/d/p/base> cd 33883/
ewu@dyn-160-39-248-197 ~/d/p/b/33883> ls
112           12526         2601          2612          2659          2696          2840_vm       3456_fsm      3602
113           12528         2601_fsm      2612_fsm      2660          2699          2841          3456_vm       3602_fsm
1247          12529         2601_vm       2612_vm       2661          2701          2995          3466          3602_vm
1247_fsm      12529_fsm     2602          2613          2662          2702          2995_vm       3466_vm       3603
1247_vm       12529_vm      2602_fsm      2613_vm       2663          2703          2996          3467          3603_fsm
1249          12531         2602_vm       2615          2664          2704          3079          3468          3603_vm
1249_fsm      12533         2603          2615_fsm      2665          2753          3079_fsm      3501          3604
1249_vm       12534         2603_fsm      2615_vm       2666          2753_fsm      3079_vm       3501_vm       3605
12504         12536         2603_vm       2616          2667          2753_vm       3080          3502          3606
12504_fsm     12538         2604          2616_fsm      2668          2754          3081          3503          3607
12504_vm      1255          2604_vm       2616_vm       2669          2755          3085          3534          3608
12506         1255_fsm      2605          2617          2670          2756          3118          3541          3609
12508         1255_vm       2605_fsm      2617_fsm      2673          2757          3118_vm       3541_fsm      3712
12509         1259          2605_vm       2617_vm       2674          2830          3119          3541_vm       3764
12509_fsm     1259_fsm      2606          2618          2675          2830_vm       3164          3542          3764_fsm
12509_vm      1259_vm       2606_fsm      2618_fsm      2678          2831          3256          3574          3764_vm
12511         1417          2606_vm       2618_vm       2679          2832          3256_vm       3575          3766
12513         1417_vm       2607          2619          2680          2832_vm       3257          3576          3767
12514         1418          2607_fsm      2619_fsm      2681          2833          3258          3576_vm       548
12514_fsm     1418_vm       2607_vm       2619_vm       2682          2834          3394          3596          549
12514_vm      174           2608          2620          2683          2834_vm       3394_fsm      3596_vm       826
12516         175           2608_fsm      2620_vm       2684          2835          3394_vm       3597          826_vm
12518         2187          2608_vm       2650          2685          2836          3395          3598          827
12519         2328          2609          2651          2686          2836_vm       34002         3598_vm       828
12519_fsm     2328_vm       2609_fsm      2652          2687          2837          34004         3599          PG_VERSION
12519_vm      2336          2609_vm       2653          2688          2838          34004_fsm     3600          pg_filenode.map
12521         2336_vm       2610          2654          2689          2838_fsm      34004_vm      3600_fsm      pg_internal.init
12523         2337          2610_fsm      2655          2690          2838_vm       34008         3600_vm
12524         2600          2610_vm       2656          2691          2839          34031         3601
12524_fsm     2600_fsm      2611          2657          2692          2840          3455          3601_fsm
12524_vm      2600_vm       2611_vm       2658          2693          2840_fsm      3456          3601_vm
ewu@dyn-160-39-248-197 ~/d/p/b/33883>
```



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

Files

Higher layers want to talk in terms of records, and files of records

File: collection of pages
Minimum API:
  insert/delete/modify record
  lookup record_id
  scan all records

Page: collection of records
  typically *fixed page sizes* (8 or 64kb in PostgreSQL)

These are logical.
Different page organizations in a file have different access costs

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
OK to not be exactly correct

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?

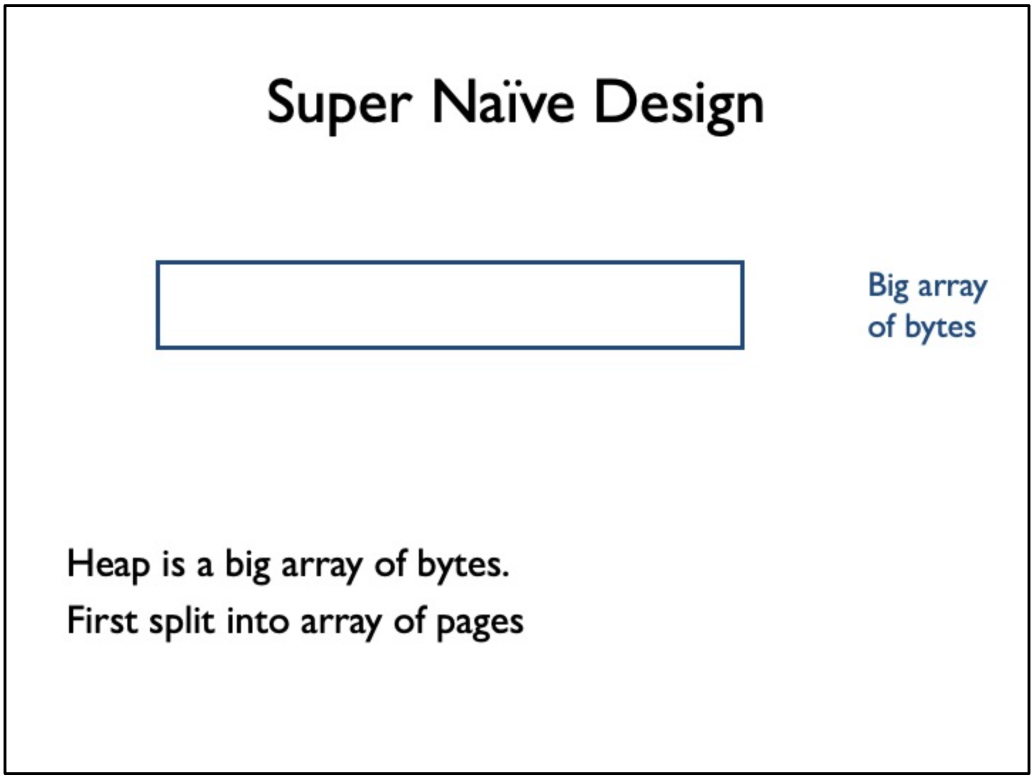
Unordered Heap Files

Collection of records (no order)

As we add records, pages allocated
As we remove records, pages removed

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

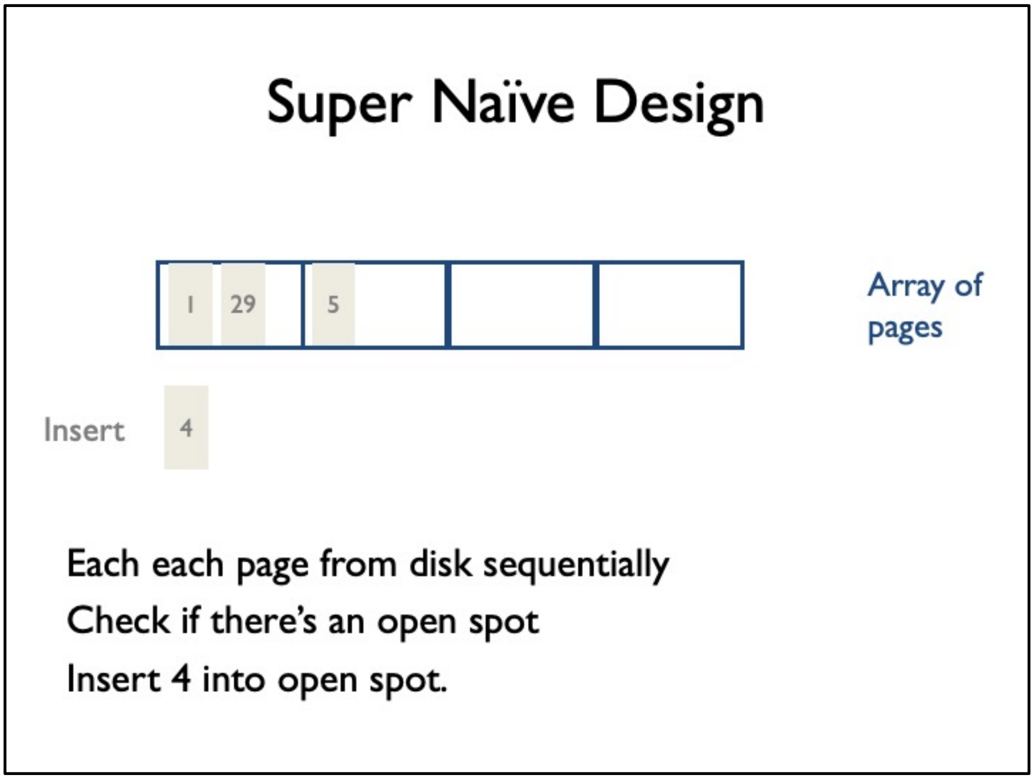
Ok, let's design that



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)

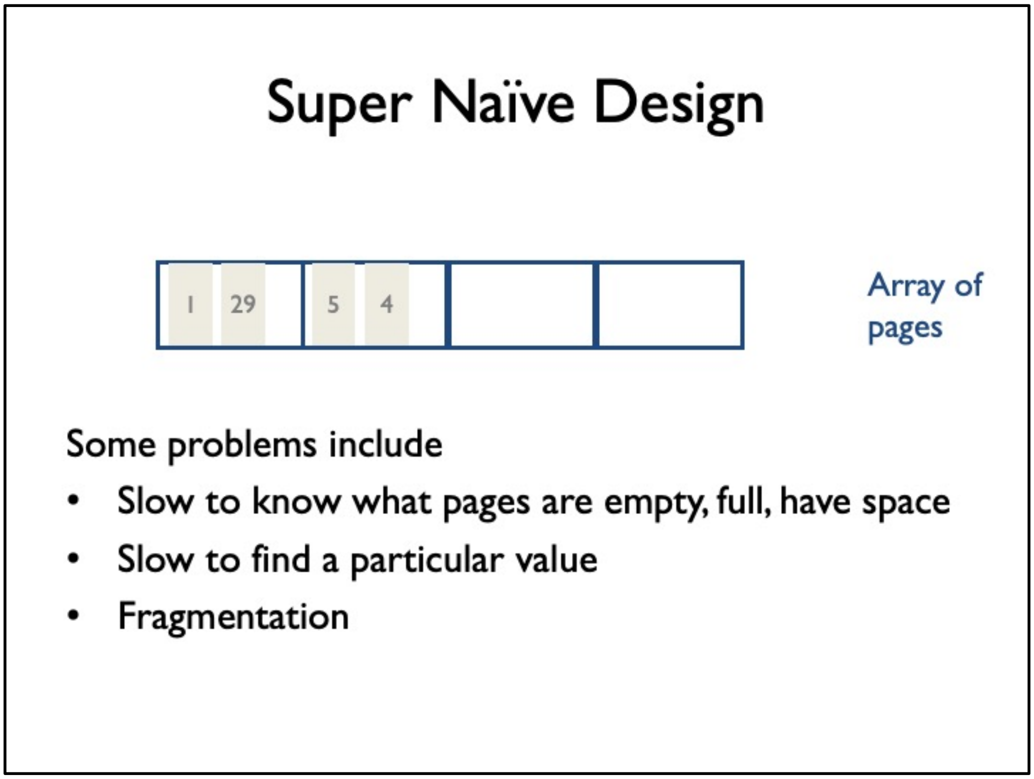
mwhat's bad about this? what's this good for?
which pages have how much free space? We don't know. Need to walk through free space linked list
how to find records? will need to scan all of the pages unless we know something more.



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)

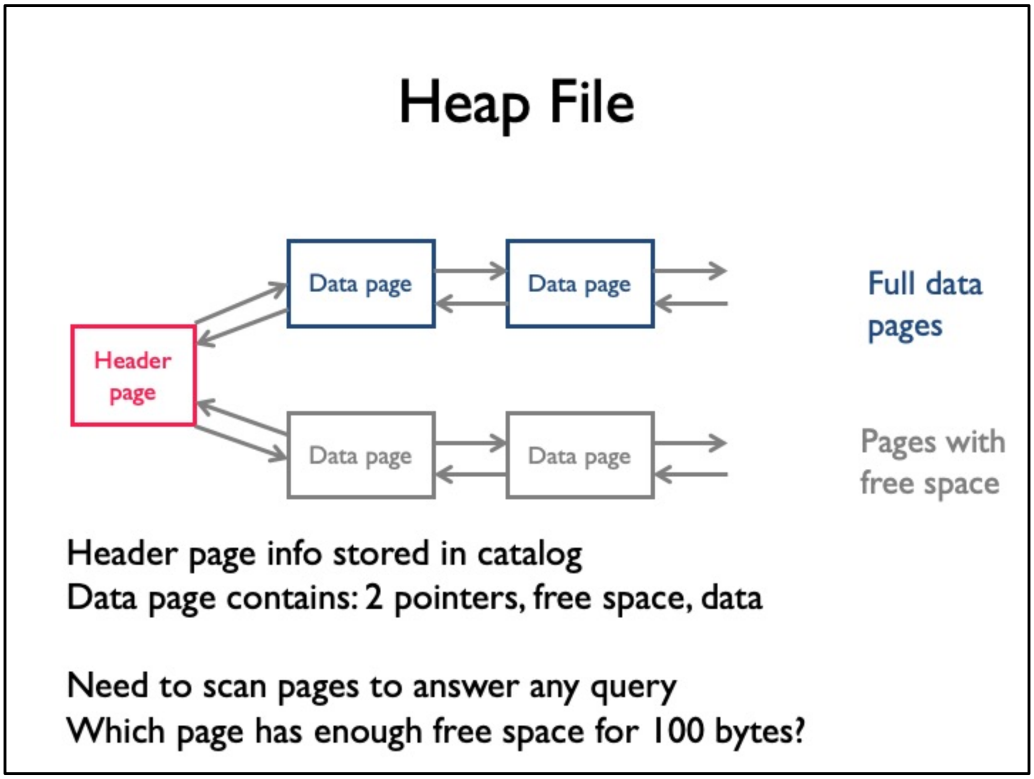
mwhat's bad about this? what's this good for?
which pages have how much free space? We don't know. Need to walk through free space linked list
how to find records? will need to scan all of the pages unless we know something more.



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.

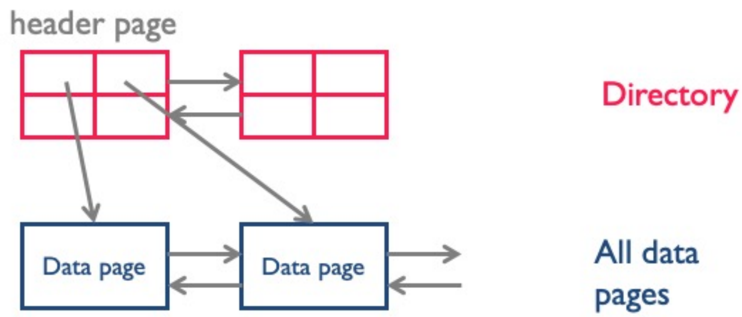


header page (directory) with two doubly linked lists, of full pages, and not full pages
location of header page stored in a database catalog (somewhere special)

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)

what'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

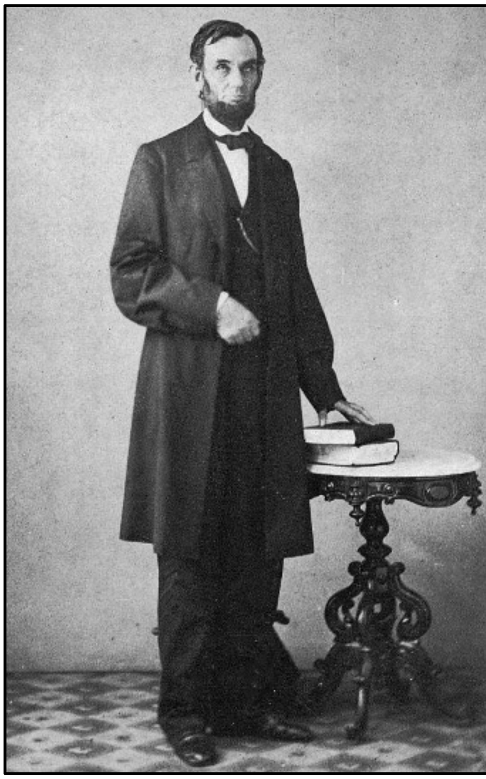


Directory entries track # free bytes on data pages

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



Indexes

"If I had 8 hours to
chop down a tree,
I'd spend 6 sharpening
my ax."

Abraham Lincoln

Indexes

Heap files answer any query via a sequential scan, but…

Queries use *qualifications* (predicates)

find students where class = "CS"

find students with age > 10

Indexes: file structures for value-based queries

B+-tree index (~1970s)

Hash index

Overview! Details in 4112

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 with respect to a *search key*

don't confuse with candidate keys!!

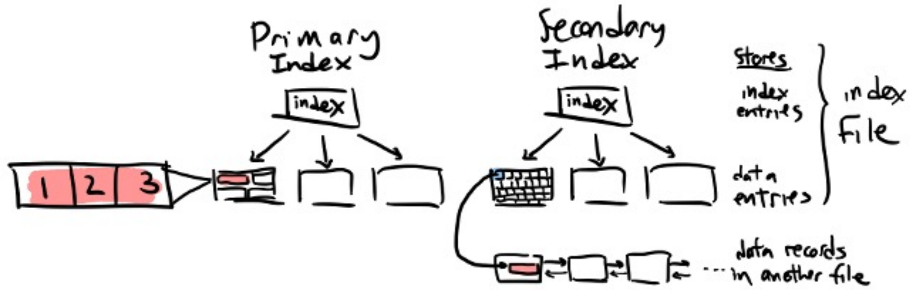
Faster access for WHERE clauses w/ search key

```
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

https://www.postgresql.org/docs/11/sql-createindex.html

Primary vs Secondary Index Files



Primary

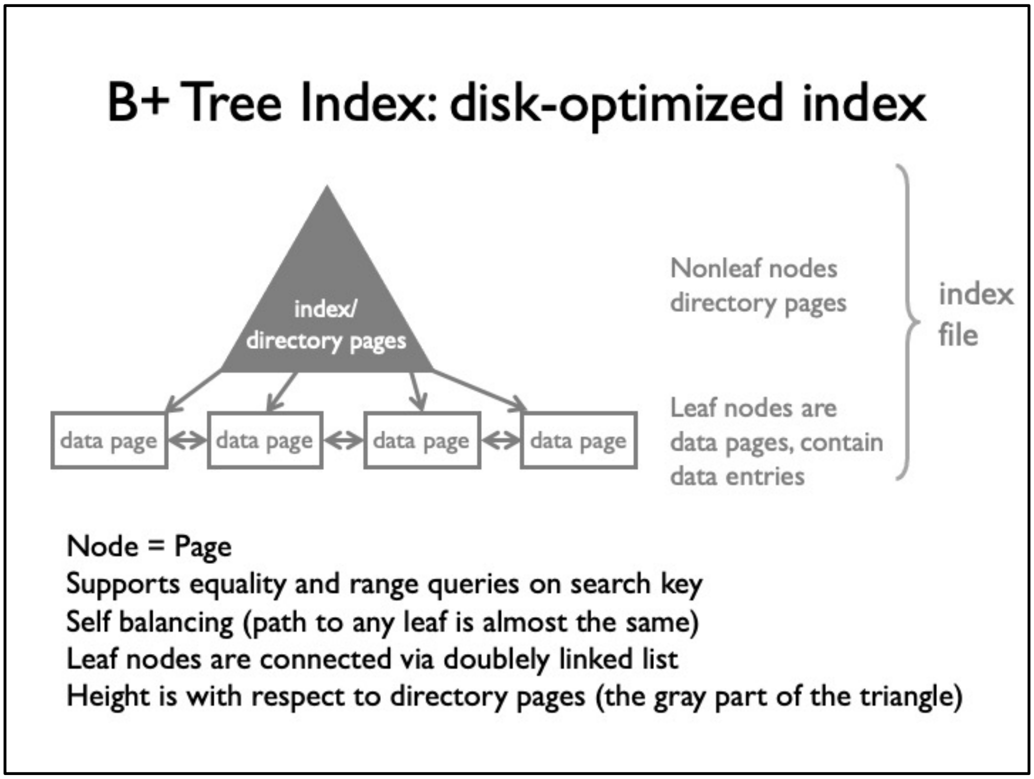
Data entries contain actual tuples

Pros: directly access data.

Secondary

Data entries contain <search key val, rid> pointers into another file

Pros: index is more compact



We saw that the directory for the heap file can reduce the cost of certain operations.

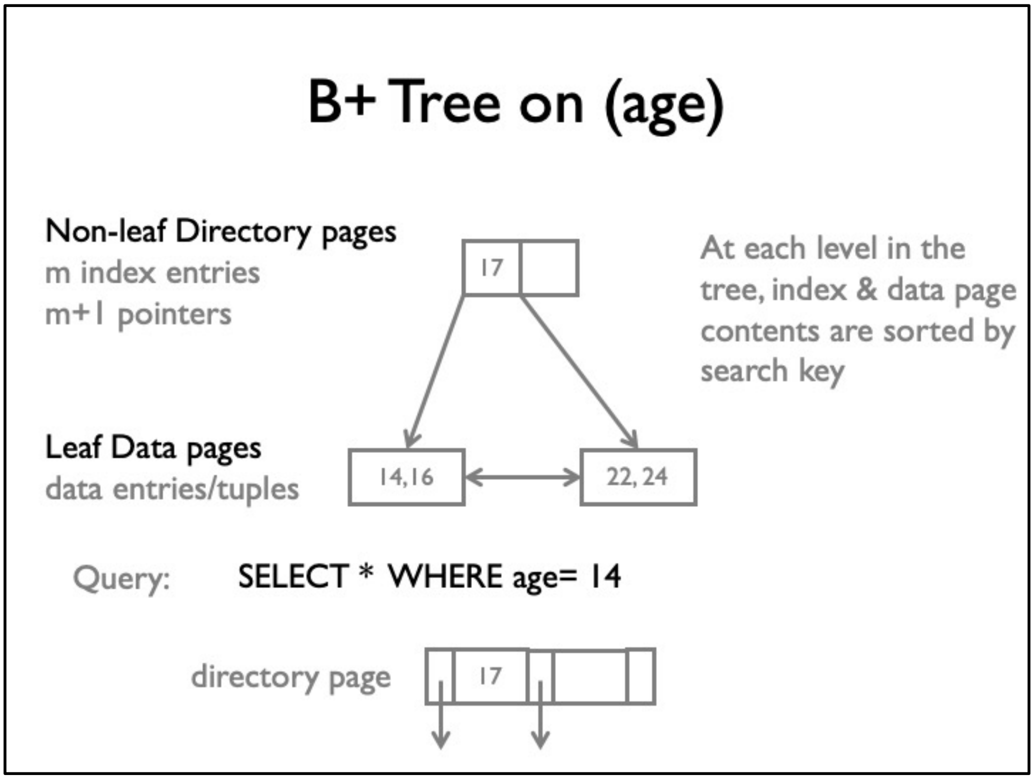
- What if we allowed multiple levels of directories?
- And kept them in sorted order on the values?

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

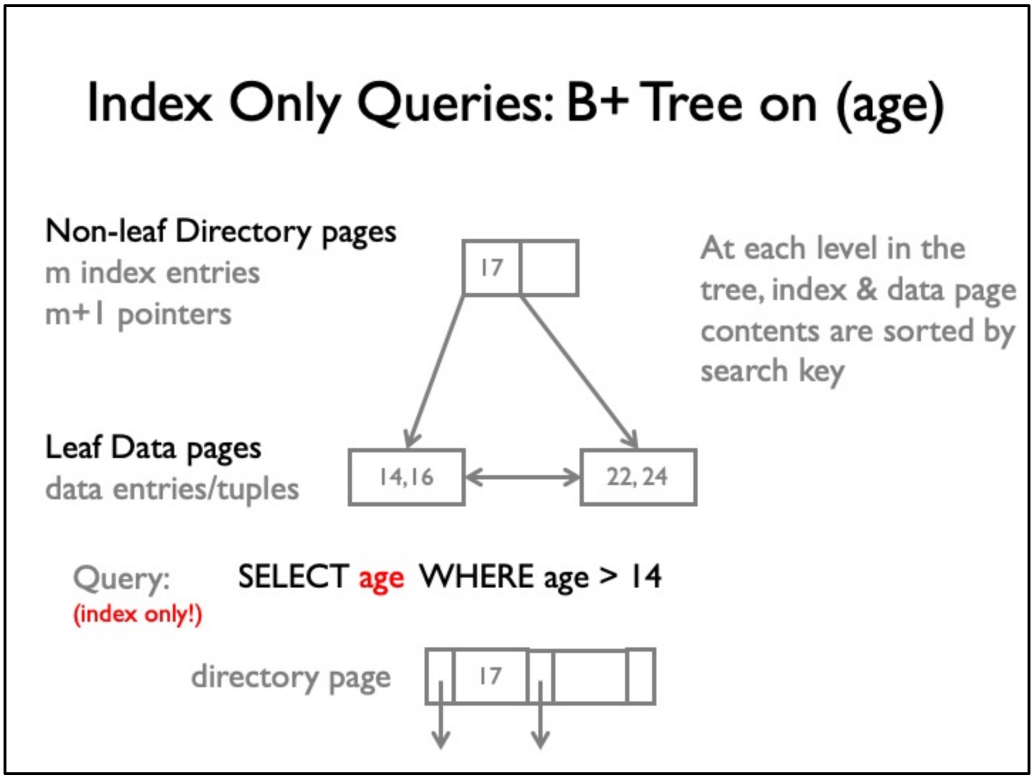


Example of a b+ tree indexed on val (val is the *index key)*
Each non-leaf node is like a directory:
- sorted list of values.
- index entry = age value to compare against when searching the tree
- the left and right side of each non-null index entry are pointers to the child nodes
- this means there are N keys and N+1 pointers in a directory page

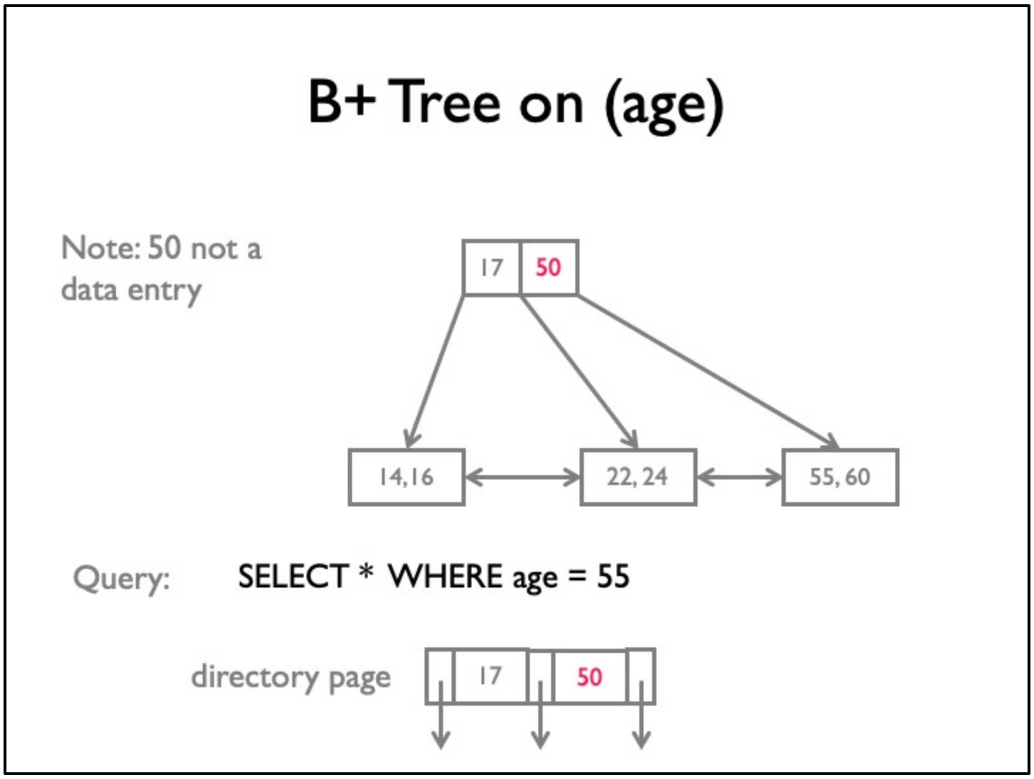
Here, I only show the age value of the tuples stored in the data pages

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

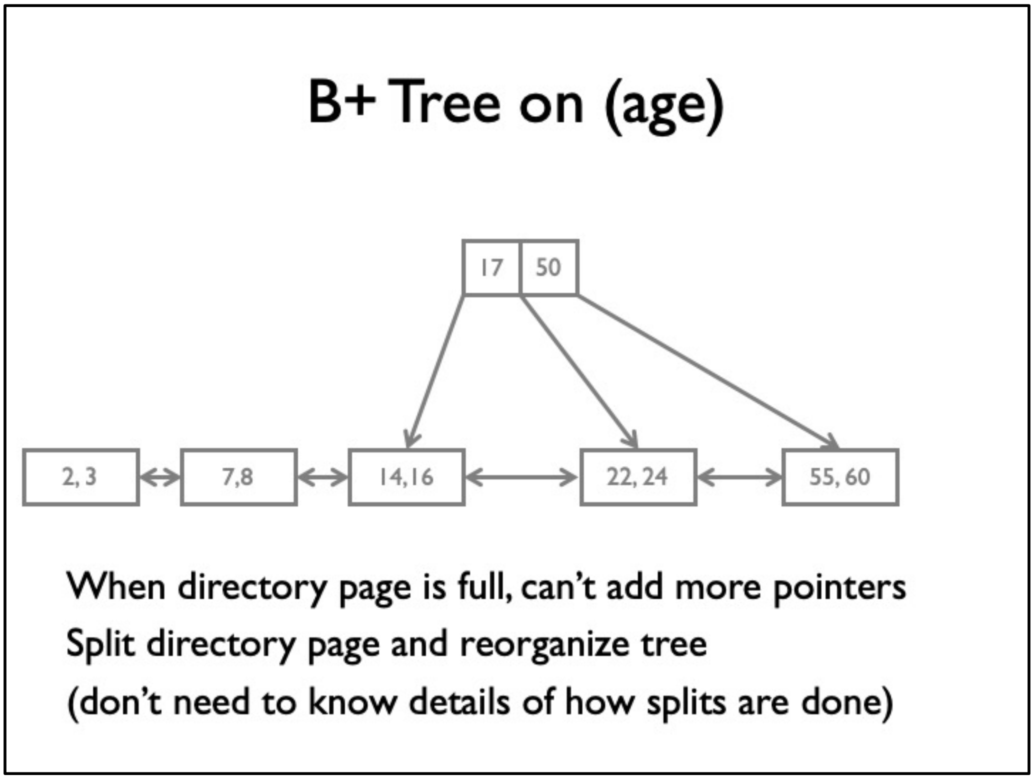
How to search? Load index page and do binary search. Can do it since loaded in memory.



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, nonleaves 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

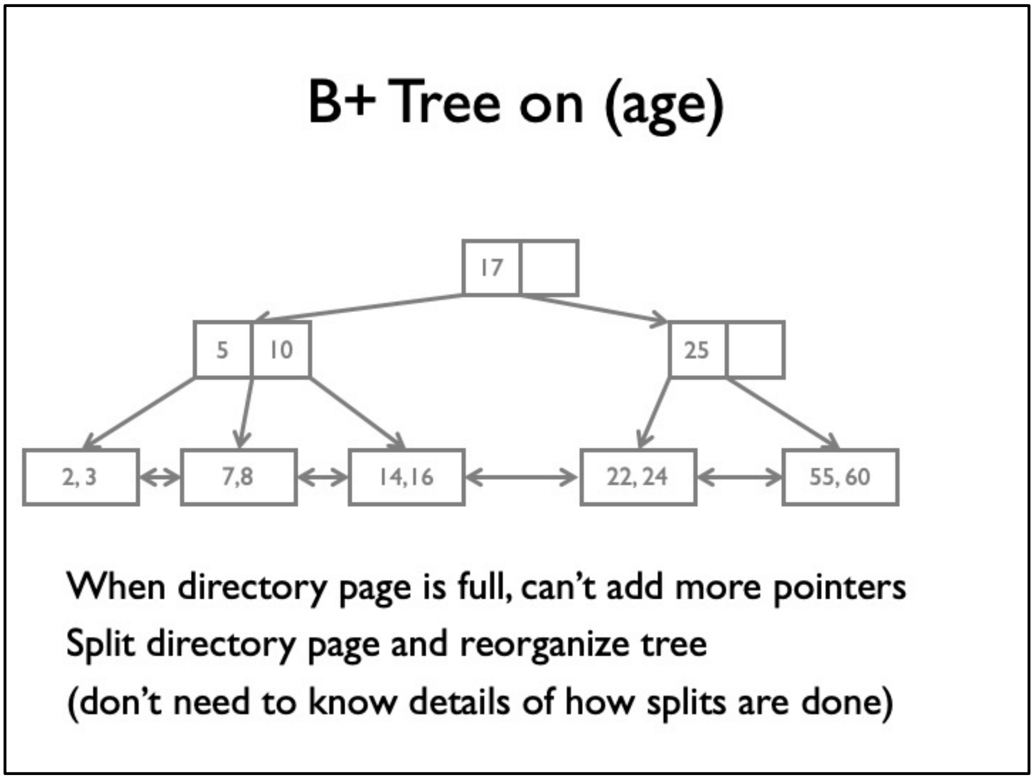
B+ Tree on (age)

When directory page is full, can’t add more pointers

Split directory page and reorganize tree

(don’t need to know details of how splits are done)

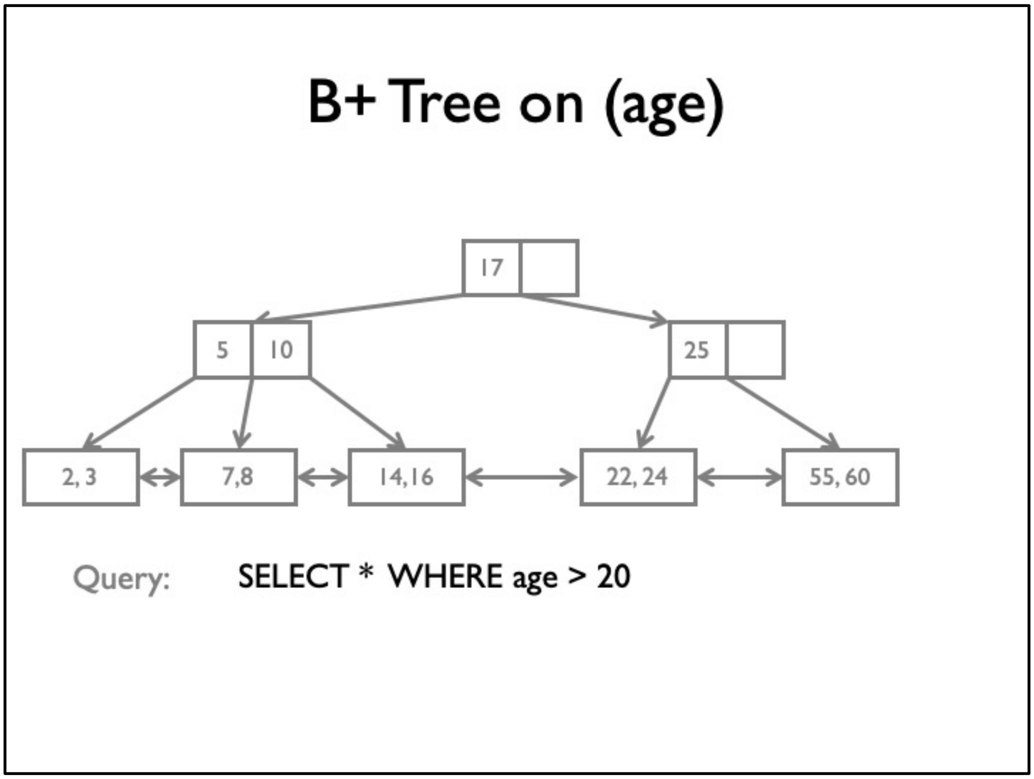
We might spit it up to index the pages this way, but this is not a tree
notice that the index entry 50 has disappeared. This is because the indexentries don’t contain data.



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

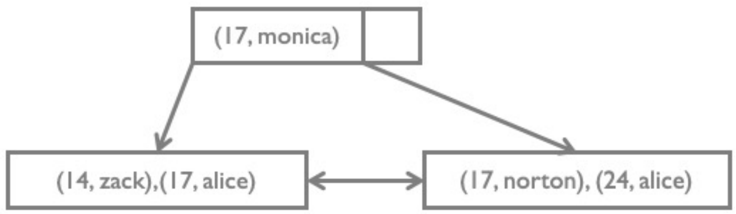


The details of how we do this don't matter, but it's self balancing, so any combination of inserts, updates, and deletes end up with a balanced tree (meaning the left and right children are roughly the same amount of data

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 Keys: B+ Tree on (age, name)



How do the following queries use the index on (age, name)?

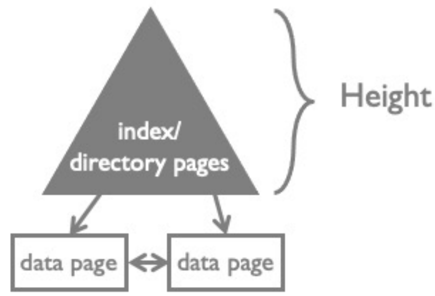
```
SELECT age   WHERE age = 14
SELECT *     WHERE age < 18 AND name < 'monica'
SELECT age   WHERE name = 'bobby'
```

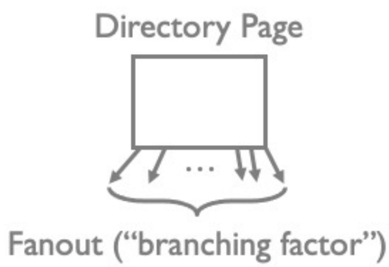
Composite key
Note that (17, alice) < (17, monica) even though both have 17.

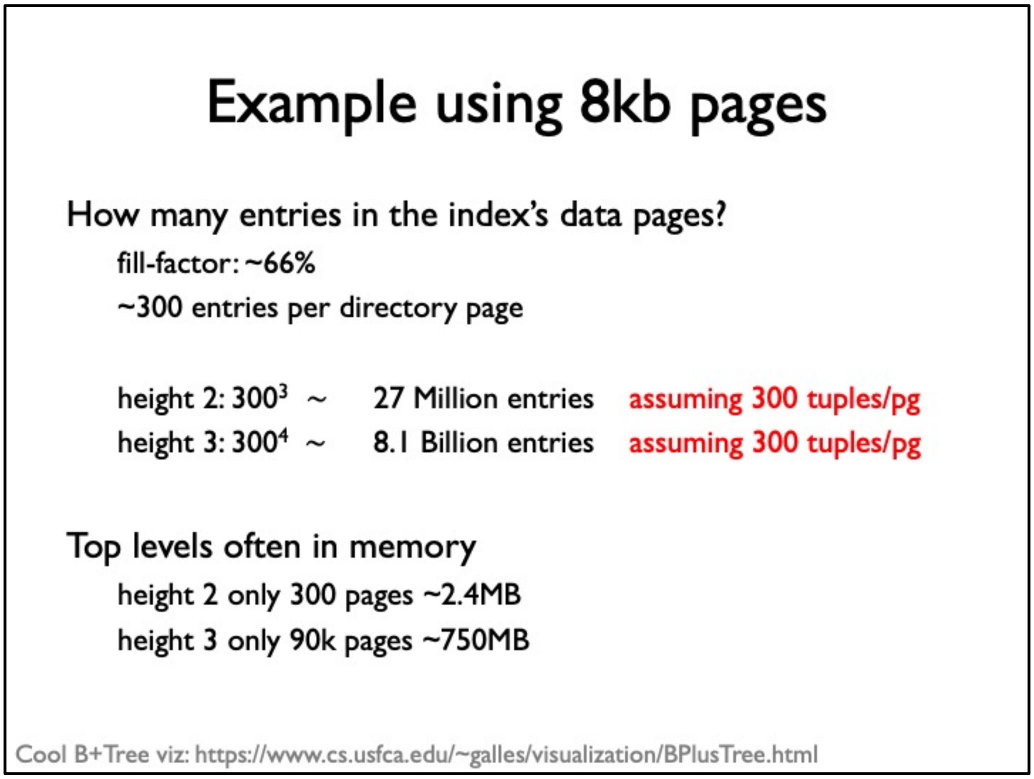
Q1: index only, use first part of the index key

Terminology

Keep free space for fast inserts
Reduce "fill factor" parameter







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

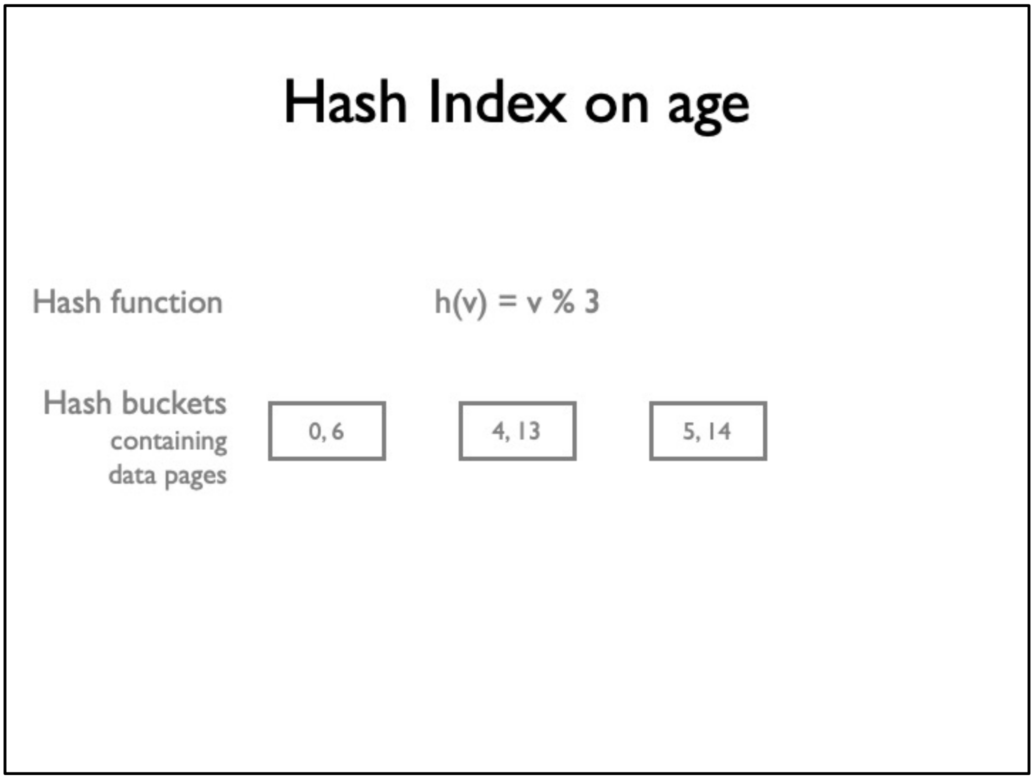
Recall that we the DBMS allocates a big chunk of memory to use for itself.
Given some standard caching policy, what are the changes the root node will be in memory?
Well it's accessed on every lookup, so it's likely in memory
What about the next level? Doesn't take much space, and probably accessed frequently, 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

For Real?

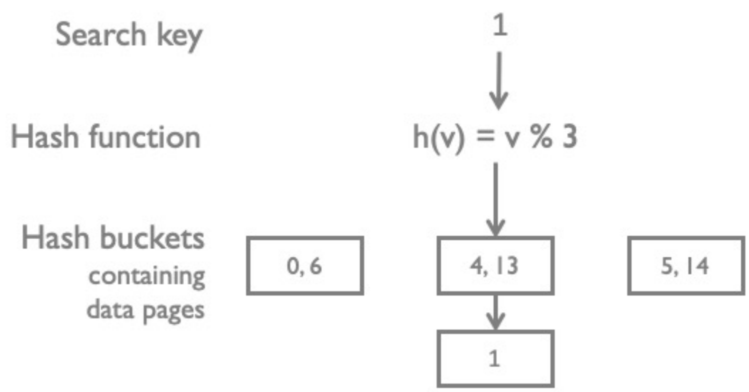
problem. We have run TB range databases on Raspberry PI with 4GB of RAM and hammered that in benchmarks (far exceeding the memory capacity). The interesting thing here is that the B+Tree nature means that the upper tiers of the tree were already in memory, so we mostly ended up with a single page fault per request.

https://ayende.com/blog/196161-C/re-are-you-sure-you-want-to-use-mmap-in-your-database-management-system



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



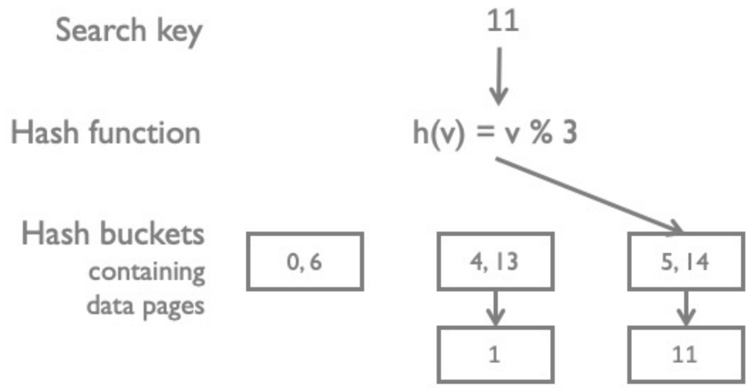
INSERT Hash Index on age

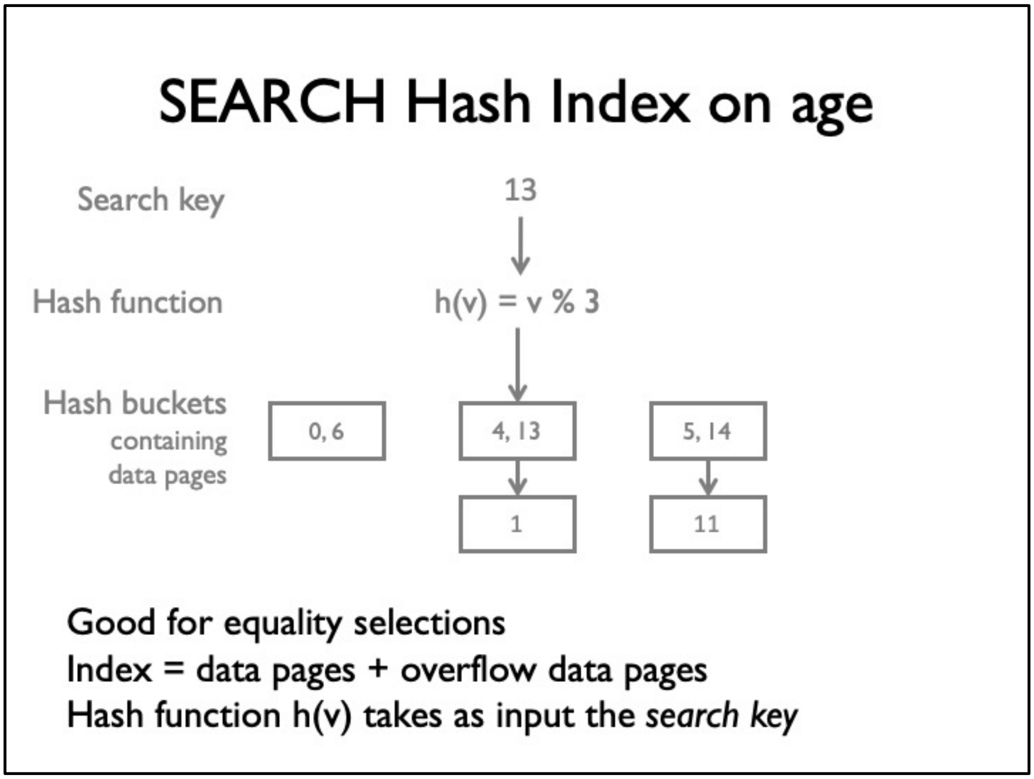
Search key 11

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
- Indexes can be primary or secondary

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 tuple	
Delete tuple	DELETE WHERE …

	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	1.25BD	
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	$D(\log_{80} B + 2)$	
Delete	Search + D	Search + BD	$D(\log_{80} B + 2)$	

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	1.25BD	1.25BD
Equality	0.5BD	$D(\log_2 B)$	$D(\log_{80} B + 1)$	D
Range	BD	$D(\log_2 B + M)$	$D(\log_{80} B + M)$	1.25BD
Insert	2D	Search + BD	$D(\log_{80} B + 2)$	2D
Delete	Search + D	Search + BD	$D(\log_{80} B + 2)$	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
- 80% fill factor when computing scans
- assumes no overflow when computing equality/insert/delete.

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?

Where do B, D, M come from?

Estimated from more basic info

Assuming (we will be clear about this)

- fanout = number of directory entries
- pointer in secondary index same size as directory entry

Given:

- p: page size
- r: record size
- d: directory entry size
- f: fill factor
- n: # records

Estimate for primary and secondary index:

- size
- height
- access cost

Primary B+ Index

Page Size	p = 100	records/page	p / r = 10
Record Size	r = 10	direntries/page	p / d = 20
Dir entry Size	d = 5	fanout:	20
Fill Factor	f = 100%	# data pages	n/(p/r) = 800
# Records	n = 8000	height	$\log_{20}800 = 3$

Cost to look up a single record is
3 for directory pages + 1 data page

Secondary B+ Index

Page Size	p = 100	records/page	p / r = 10
Record Size	r = 10	direntries/page	p / d = 20
Dir entry Size	d = 5	fanout:	20
Fill Factor	f = 100%	# data pages	n/(p/d) = 400
# Records	n = 8000	height	2

Cost to look up a single record is
2 for directory pages + 1 data page + 1 pointer lookup

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 or tree 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 & differences between RAM, SSD, Hard drives

What files, pages, and records are, and how different than UNIX model

Heap File data structure

B+ tree and Hash indexes

Performance characteristics of different file organizations

Given statistics, figure out directory size, index height, access cost