L19
Disk, Storage, and Indexing

Example

Branch, Customer, banker Name, Office

BCNO

Name -> Branch, Office   N -> BO

Customer, Branch -> Name CB -> N

CB is the key (determines everything)

3NF

F^{min} = minimal cover of F

Run BCNF using F^{min}

for $X \rightarrow Y$ in F^{min} not in projection onto $R_1 \ldots R_N$

    create relation XY

BCNO   BC → N, N → BO

    NBO, CN using N → BO … oops create BCN

    NBO, CN, BCN … remove redundant CN relation

    NBO, BCN … BCN: BC is key; N → B violates BNCF

Branch, Customer, banker Name, Office

BC → N, N → BO

NBO key is N: determines all other fields

- Banker name determines branch, office
- NBO is in BCNF: All FDs are keys

BCN key?

- Maybe N? N determines B, but not C!
- Problem: A banker has multiple customers!
- Maybe CN? Customer could go to multiple branches!
- Example: (C: Alice, N: Evan, B: NYC), (Alice, Evan, SF)

Keys and Functional Dependencies

A key must determine all column values

Must be the left hand side (aka input or source) of FDs

BC → N, N → BO

NBO key is N: determines all other fields

- Banker name determines branch, office
- NBO is in BCNF: All FDs are keys

BC → N, N → BO

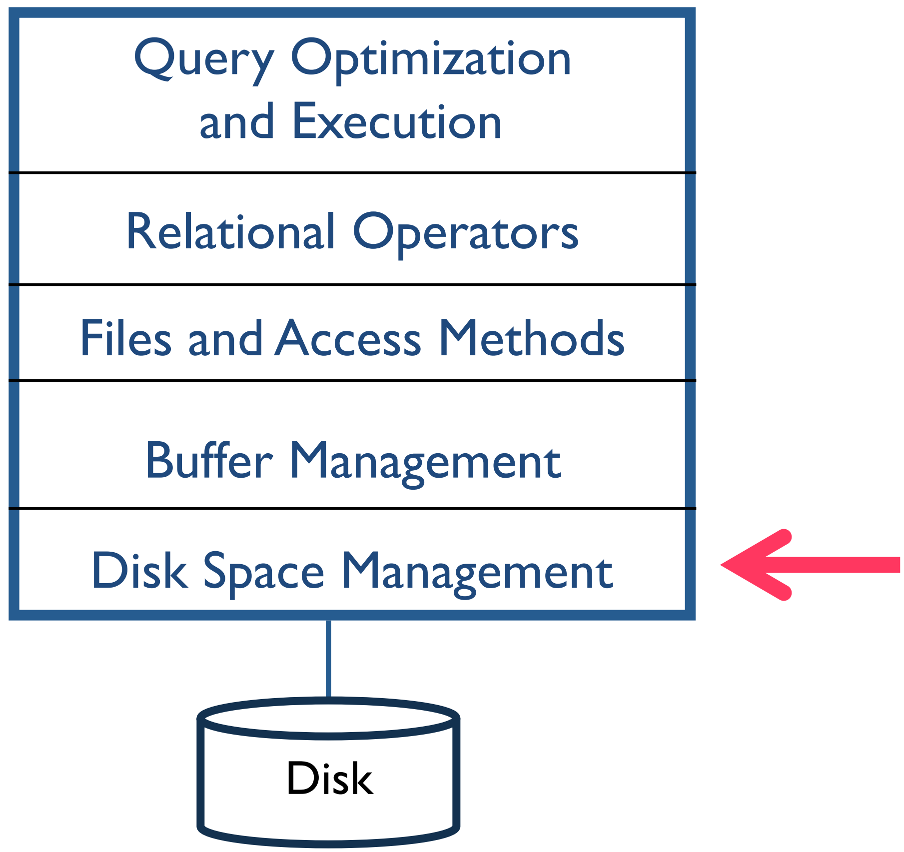
BCN key?

Maybe N? N determines B, but not C!

- Problem: A banker has multiple customers!
- Maybe CN? Customer could go to multiple branches!
- Problem: (C: Alice, N: Evan, B: NYC), (Alice, Evan, SF)

Must be BC; N → B violates BCNF; Okay for 3NF

Work from the bottom up



$ 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

$ Matters

Newegg enterprise $1000

- RAM: 64-96 GB
- SSD: 400-1000
- Disk: 24000

RAM for active data

Disk for main database

- secondary storage

Tapes for archive

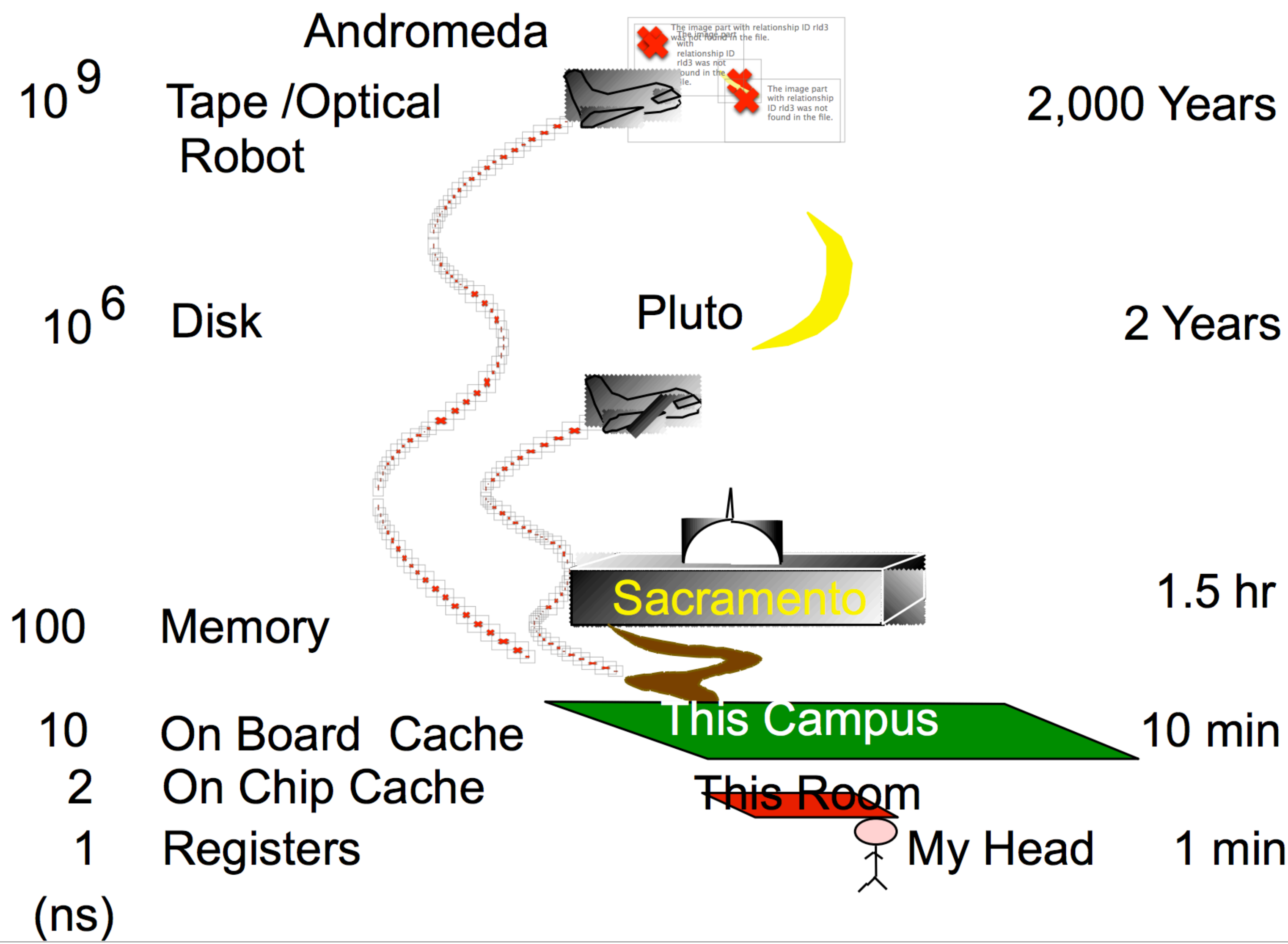
Registers	.3 ns
on chip cache	.5ns
off chip cache	7 ns
RAM	100 ns
SSD	.15 ms
Disk	5 ms

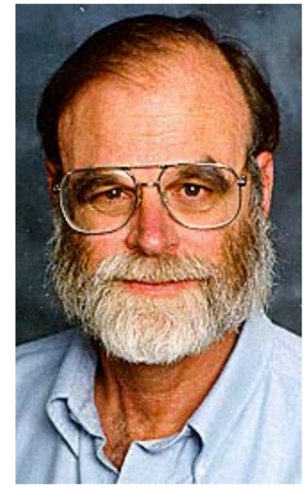
Interesting numbers

compress 1k bytes:	3000 ns
roundtrip in data center:	.5 ms

https://gist.github.com/jboner/2841832

Jim Gray's Storage Latency Analogy: How Far Away is the Data?

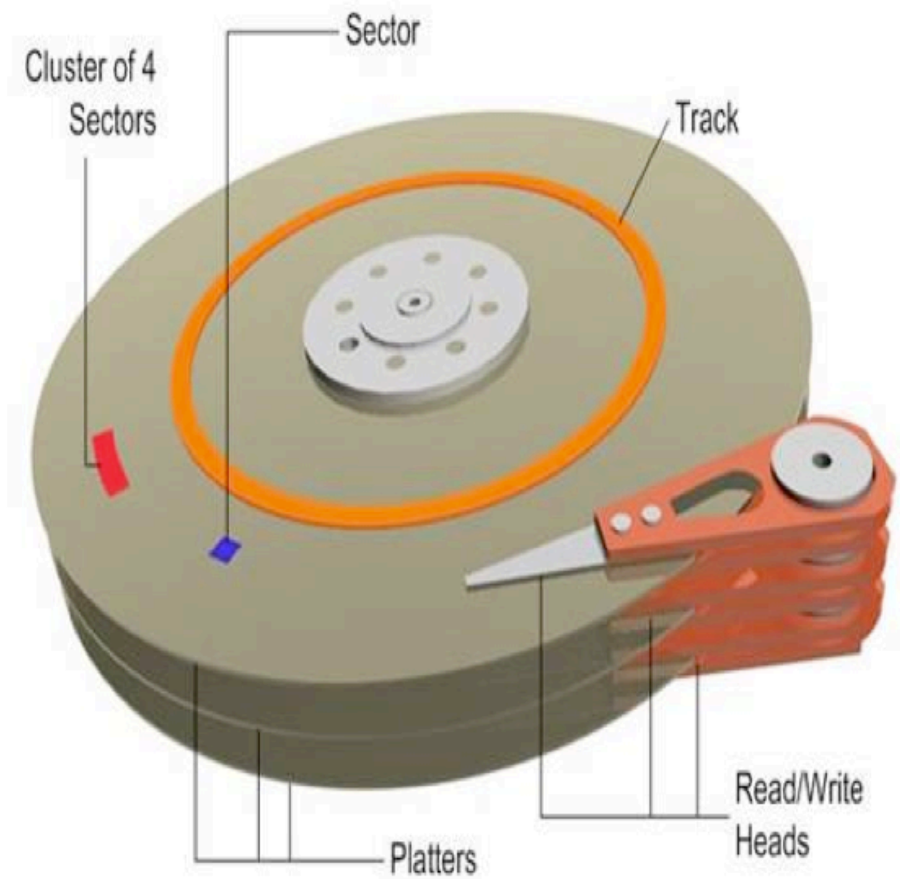




Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969!

Spin speed: ~7200 RPM

Arm moved in/out to position head over a track



Time to access (read or write) a disk block

seek time	2-4 msec avg (arm movement)
rotational delay	2-4 msec (based on rotation speed)
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

Pre-fetching

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

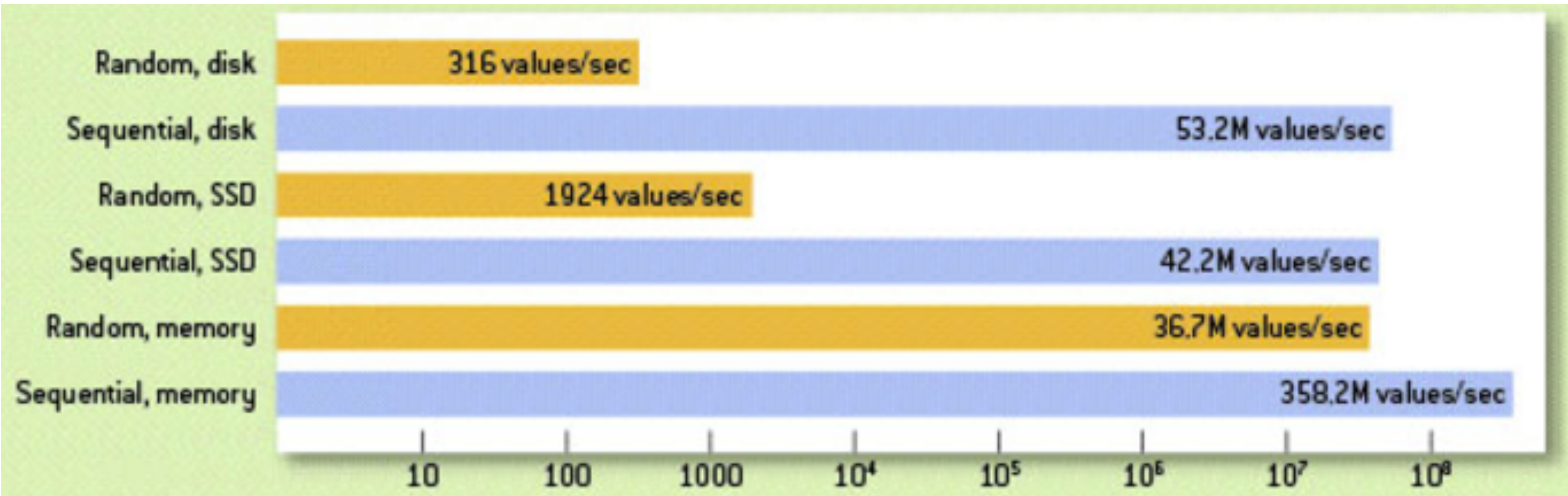
- >1 page/block at once

SSD versus Hard Drives

Disks are not dead!

	HDD WD Black 6 TB	SDD Samsung 850 Pro	SDD Factor
Sequential Throughput	214 MB/s	496 MB/s	2.3X
Random Throughput	0.5 MB/s	273 MB/s	546X
Random IO Latency	4800 us	8 us	600X
$/GB	$0.05	$0.46	0.1X

4 byte values read per second



5 orders of magnitude

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

Disk Space Management

VLDBs	SSDs: reduce variance
Small DBs	interesting data is small

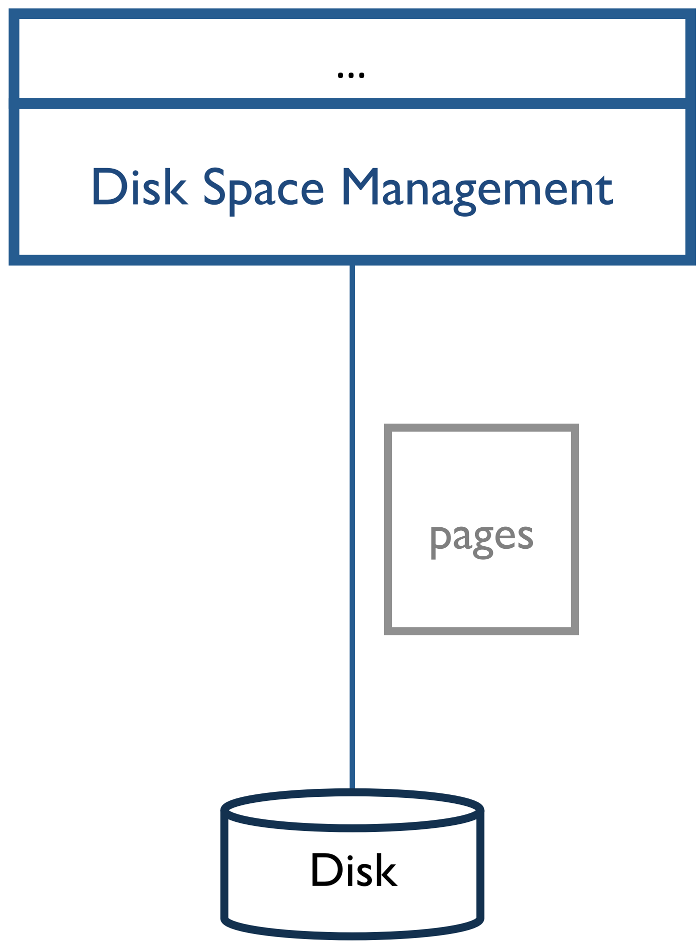
Huge data exists

Many interesting data is small

People will still worry about magnetic disk

May not care about it

Work from the bottom up



What is a page?

Unit of transfer between storage and database

Typically fixed size

Small enough for one I/O to be fast

Big enough to not be wasteful

Usually a multiple of 4 kB

- Intel virtual memory hardware page size
- Modern disk sector size (minimum I/O size)

Random example sizes

SQLite: 1 kB

IBM DB2: 4 kB

Postgres: 8 kB pages

SQL Server: 8 kB

MySQL: 16 kB

MongoDB (Wired Tiger): 32 kB

Disk Space Management

Manages space on disk, IO, and caching

Sequential performance desirable

- hidden from rest of DBMS
- some algorithms assume sequential performance

Example Disk Space Interface

DiskInterface:

    readPage(page_id): data

    writePage(page_id, data)

    newPage(): page_id

    freePage(page_id)

Records and Files

Record: "application" storage unit

  e.g. a row in a table

Page: Collection of records

File: Collection of pages

  insert/delete/modify record

  get(record_id) a record

  scan all records

May be in multiple OS files spanning multiple disks

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

Unordered Heap Files

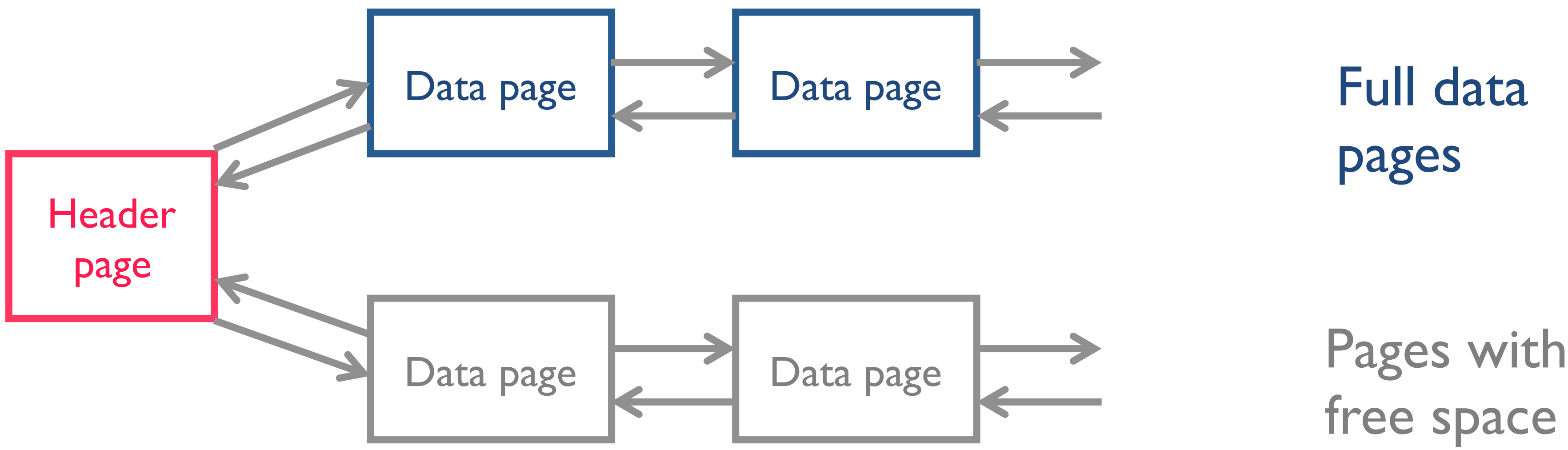
Unordered collection of records

Pages allocated as collection grows

Need to track:

- pages in file
- free space on pages
- records on page

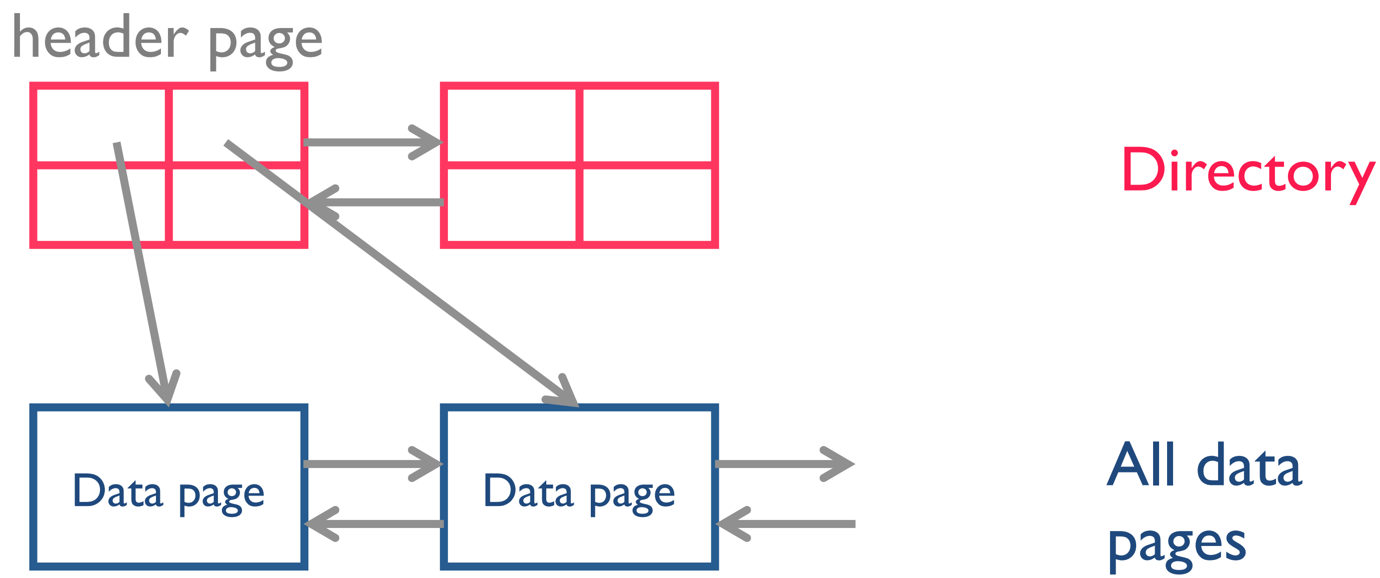
Potential Heap File Implementation



Header page location? Typically in the "catalog"

Data page = 2 pointers + data

Alternative: Use a directory



Directory entries track #free bytes on data pages

Directory is collection of pages

Indexes

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

Abraham Lincoln

Indexes

Heap files can get data by sequential scan

Queries use *qualifications* (predicates)

find students in "CS"

find students from CA

Indexes

file structures for value-based queries

B+-tree index (~1970s)

Hash index

Overview! Details in 4112

Indexes

Defined wrt a *search key*

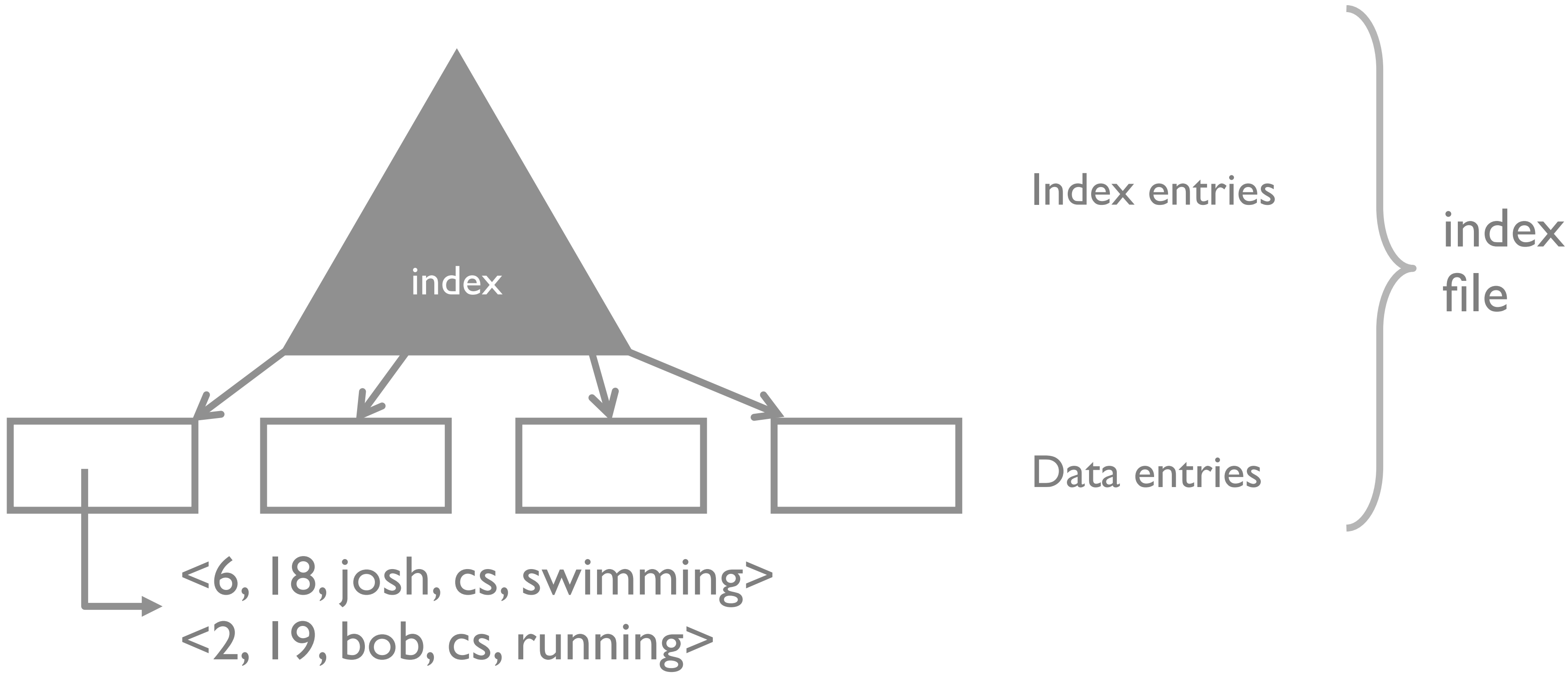
different than a candidate key!

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

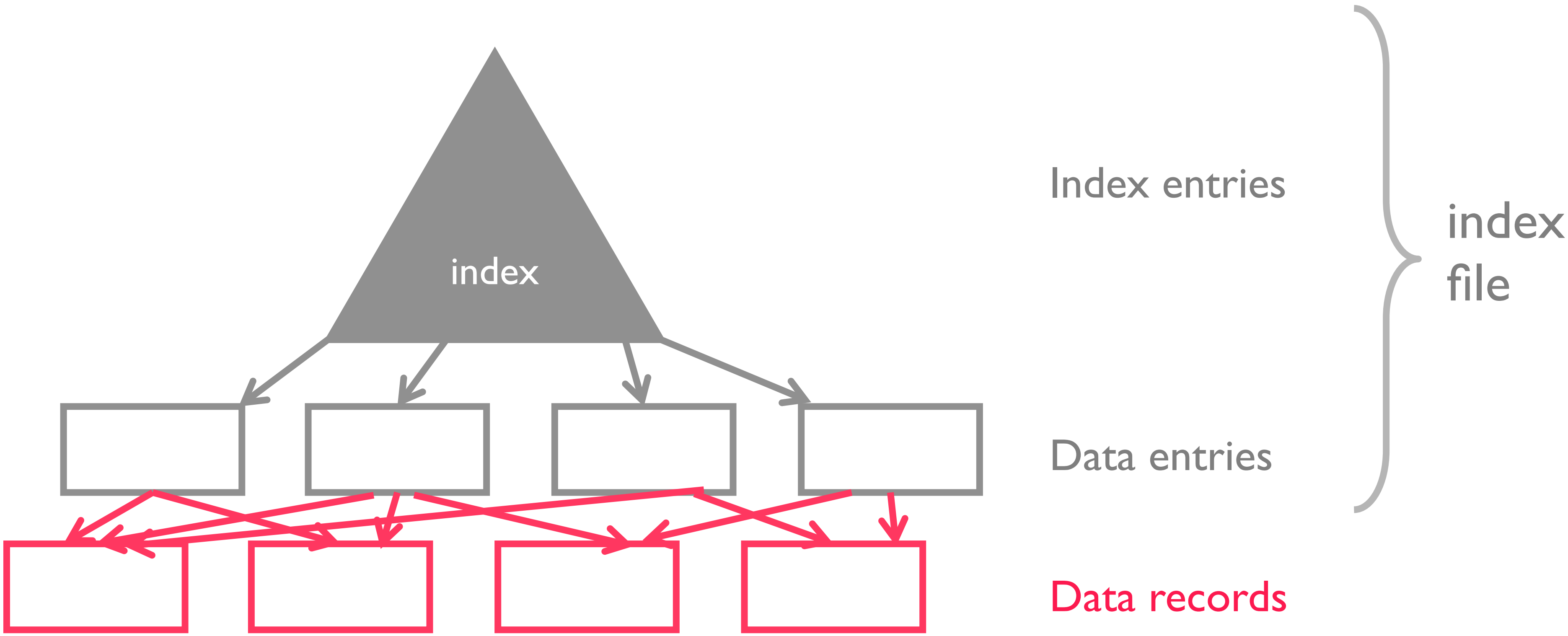
High level (Primary) index structure



What is a data entry?

actual data record

High level (Secondary) index structure



What is a data entry?

actual data record

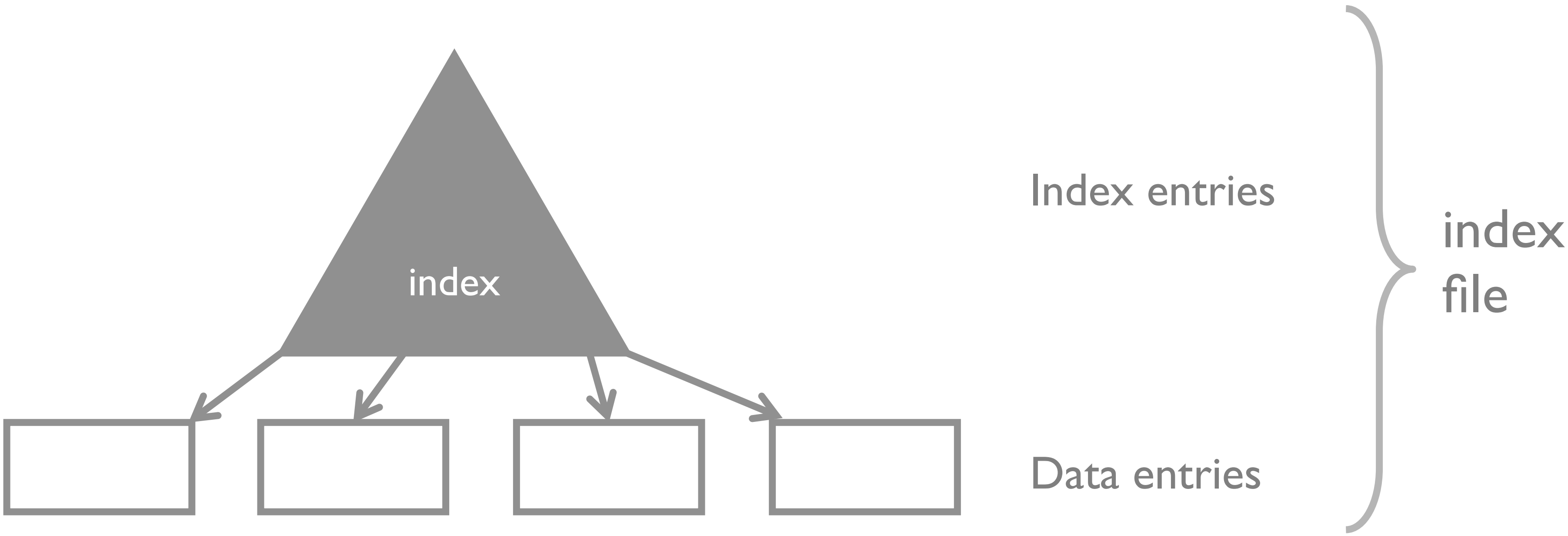
<search key value, rid>

Tradeoffs

directly access tuple.

compact, fixed size entries

High level index structure



What is a data entry?

actual data record

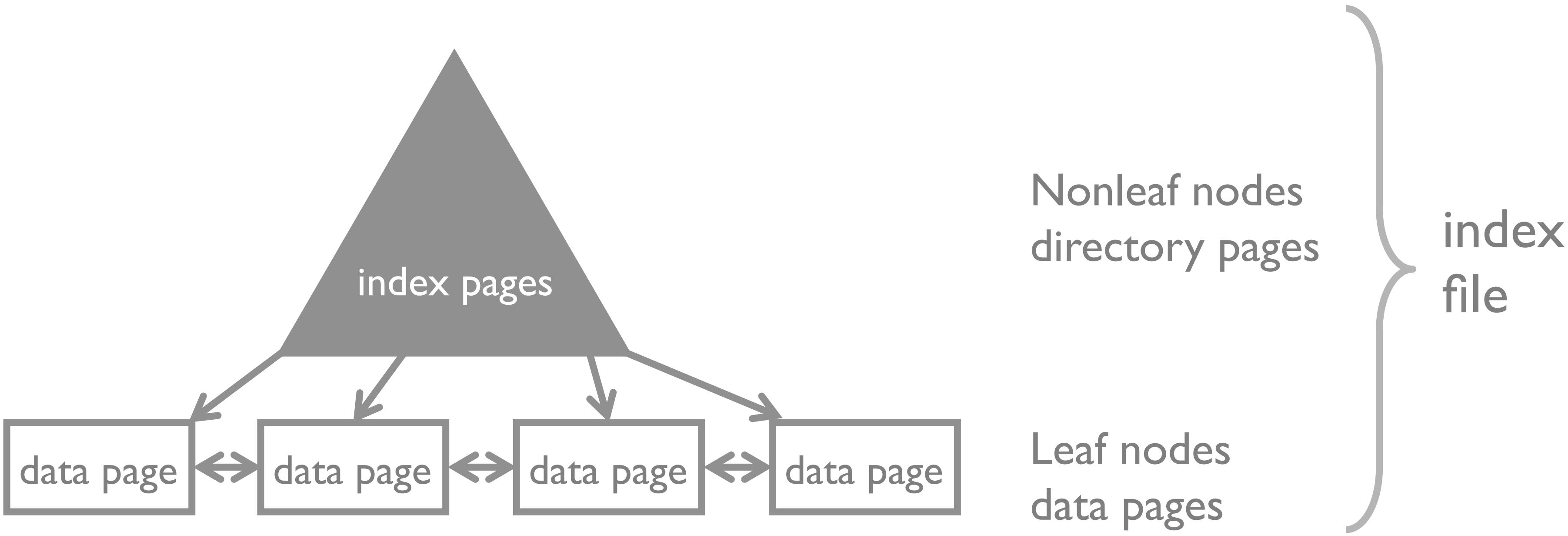
<search key value, rid>

Tradeoffs

directly access tuple.

compact, fixed size entries

B+ Tree Index

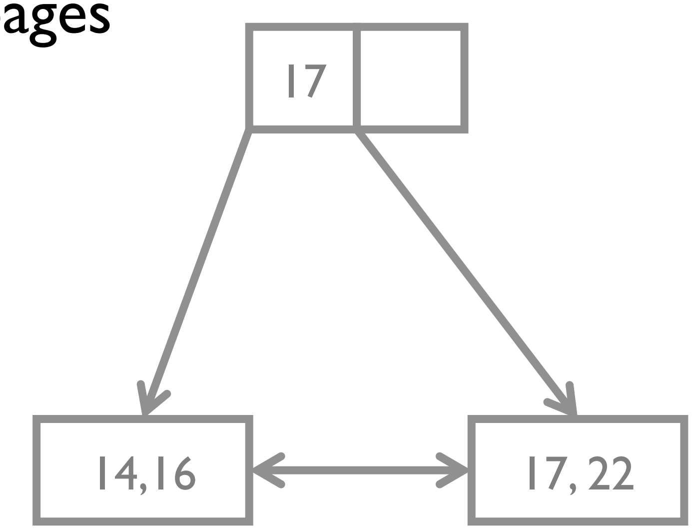


Node = Page
Equality and range queries
Self balancing
Leaf nodes are connected
Disk optimized

B+ Tree on (age)

Non-leaf directory pages
m index entries
m+1 pointers

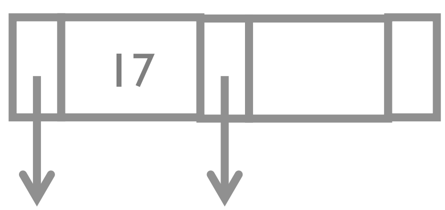
Leaf Data pages
data entries/tuples



index & data
page contents
are in order

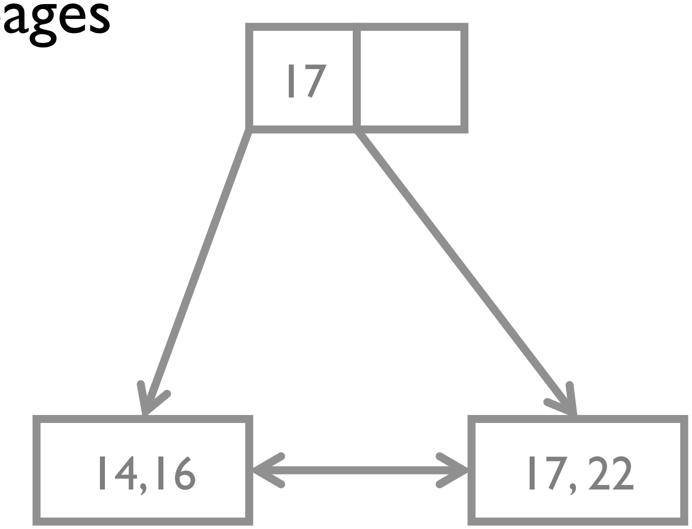
Query: SELECT * WHERE age= 14

directory page



Index Only Queries: B+ Tree on (age)

Non-leaf directory pages
m index entries
m+1 pointers

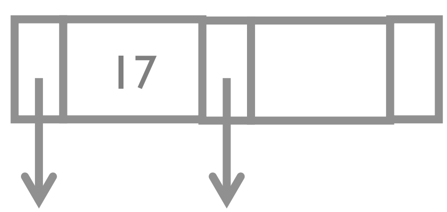


index & data
page contents
are in order

Leaf Data pages
data entries/tuples

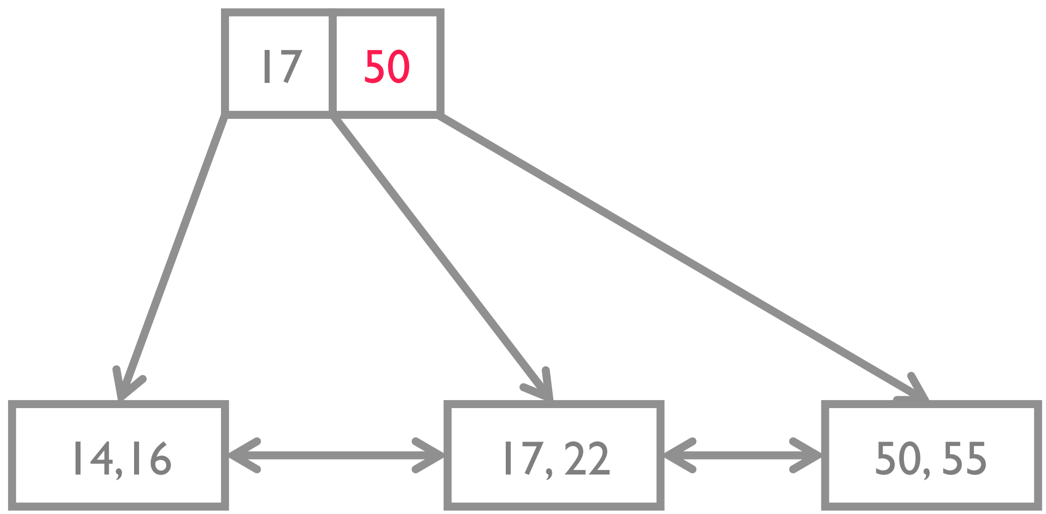
Query: SELECT age WHERE age = 14
(index only!)

directory page



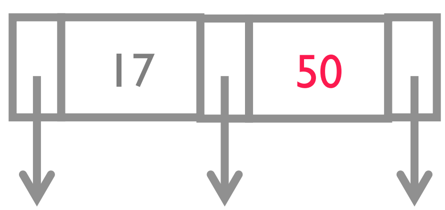
B+ Tree on (age)

Note: 50 not a data entry

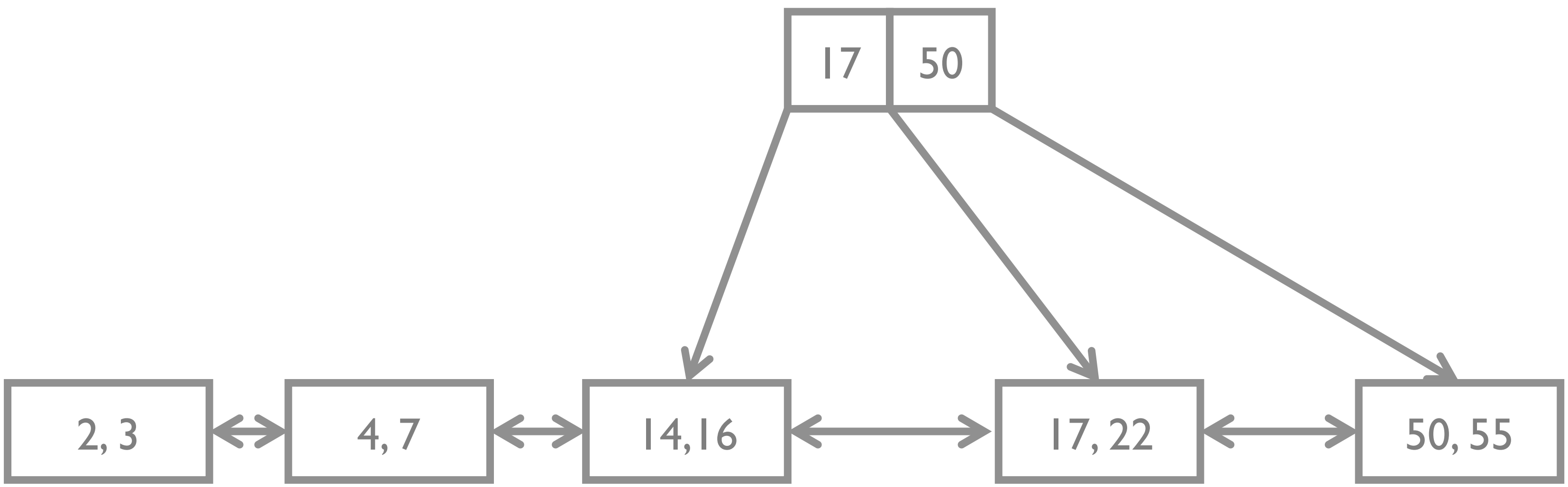


Query: SELECT * WHERE age = 50

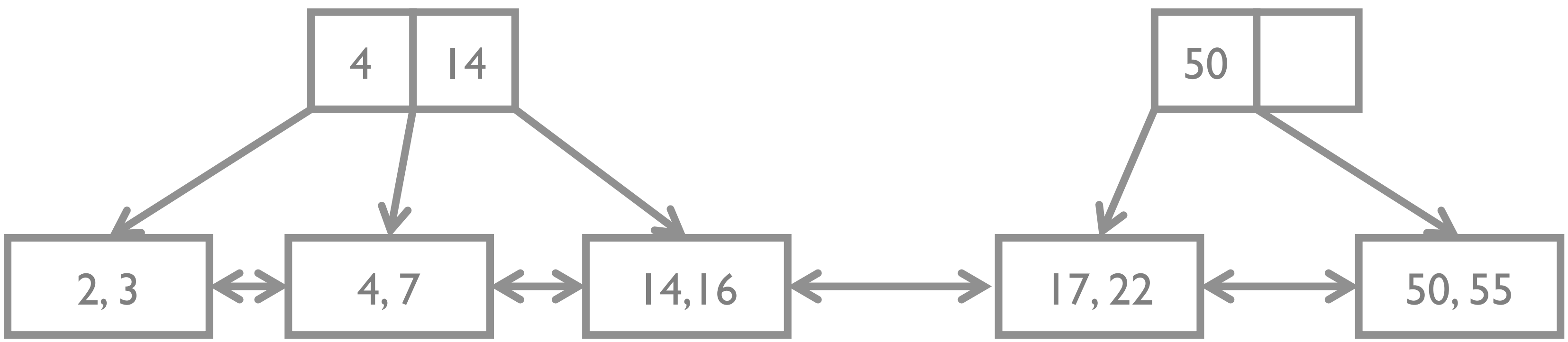
directory page



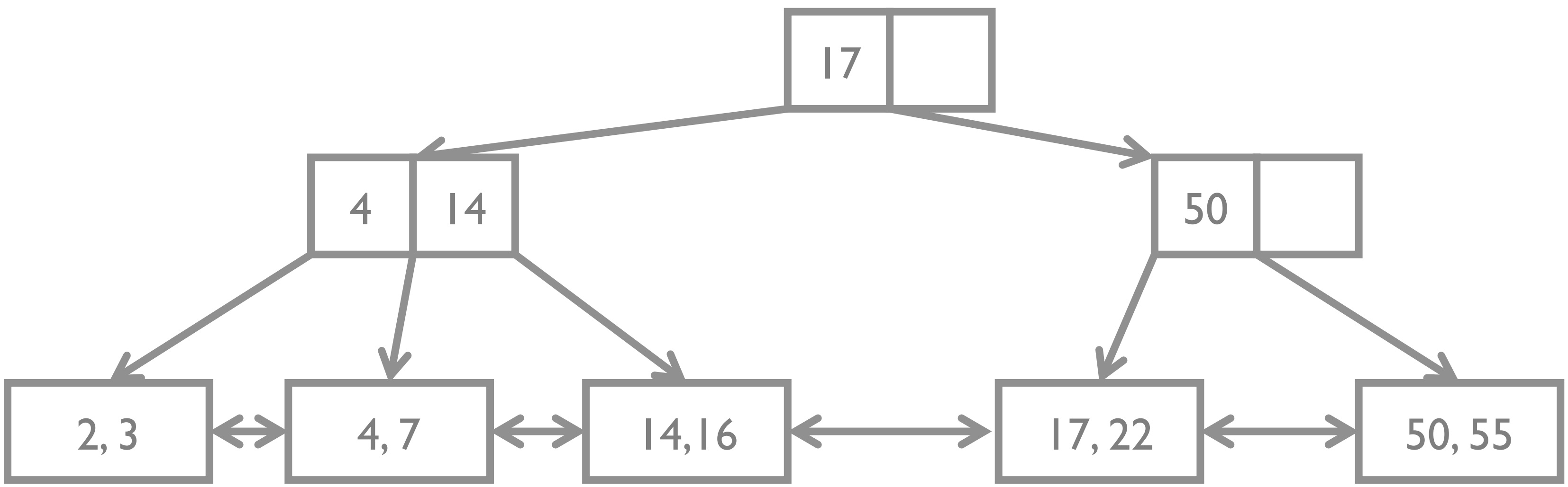
B+ Tree on (age)



B+ Tree on (age)

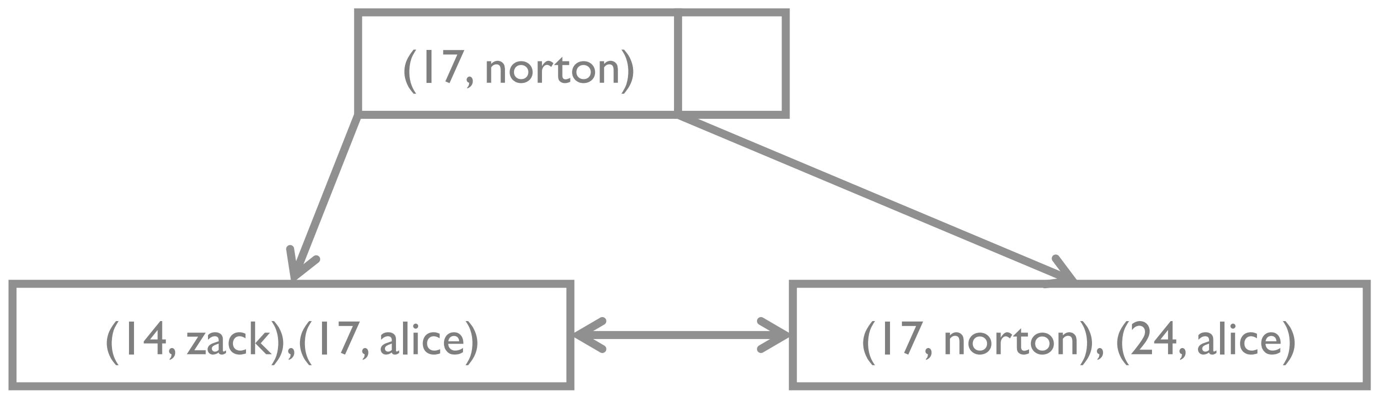


B+ Tree on (age)



Query: SELECT * WHERE age > 15

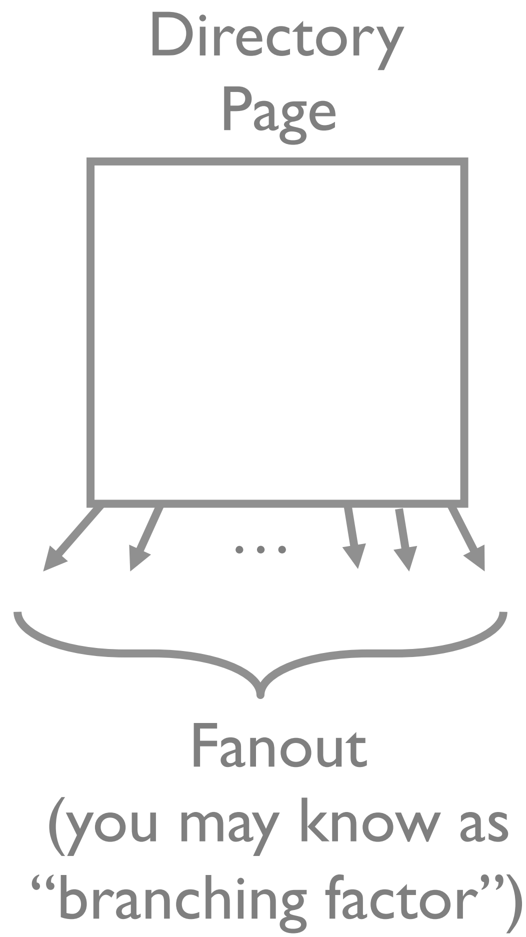
B+ Tree on (age, name)



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’
```

Terminology



Some numbers (8kb pages)

How many levels?

- fill-factor: ~66%
- ~300 entries per directory page

- height 2: $300^3 \sim$ 27 Million entries
- height 3: $300^4 \sim$ 8.1 Billion entries

Top levels often in memory

- height 2 only 300 pages ~2.4MB
- height 3 only 90k pages ~750MB

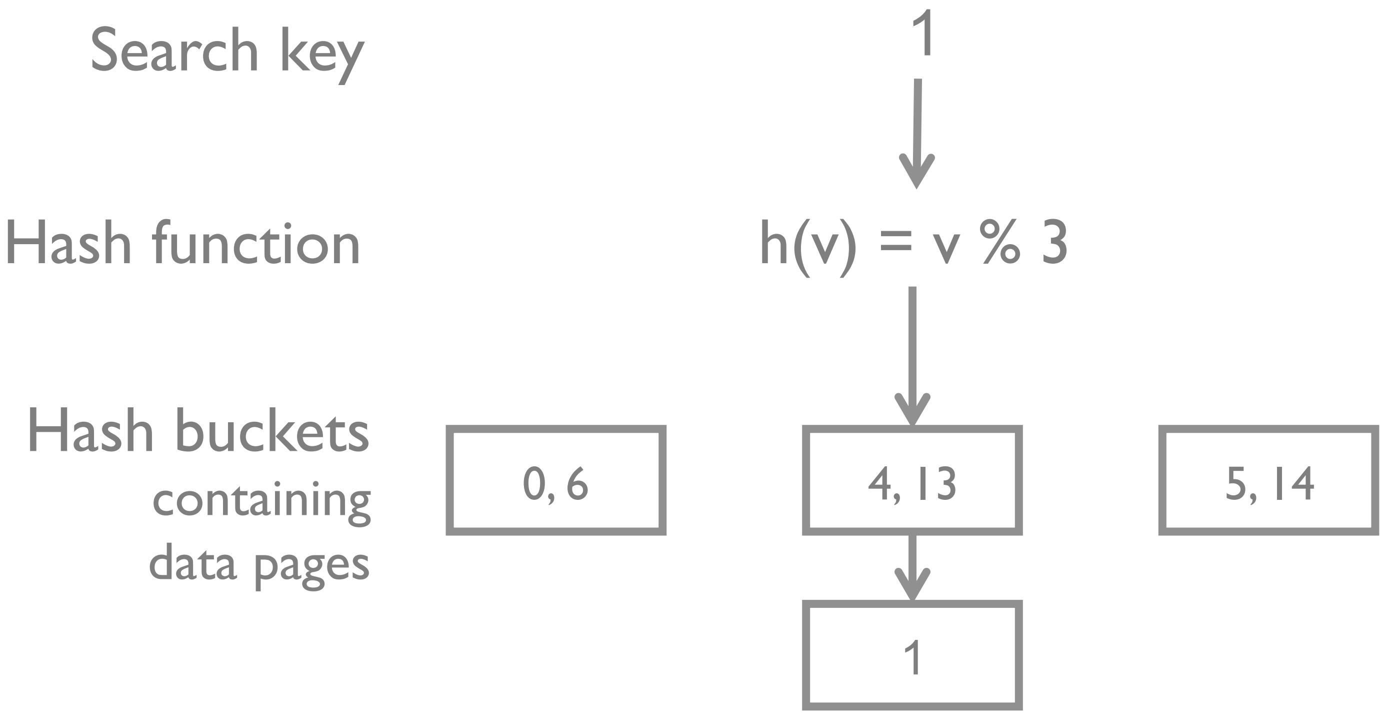
Cool B+Tree viz: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Hash Index on age

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

Hash buckets containing data pages	0, 6	4, 13	5, 14

INSERT Hash Index on age

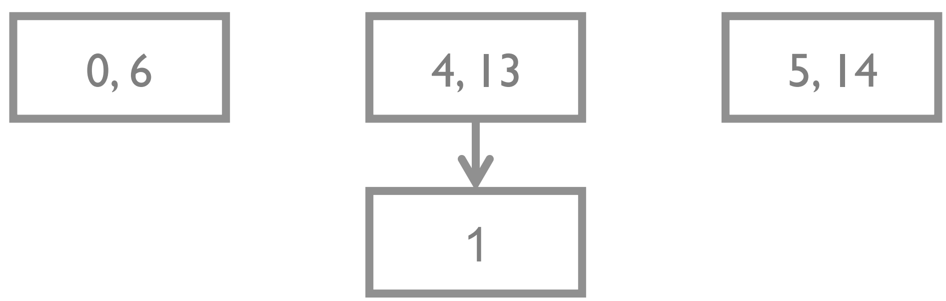


INSERT Hash Index on age

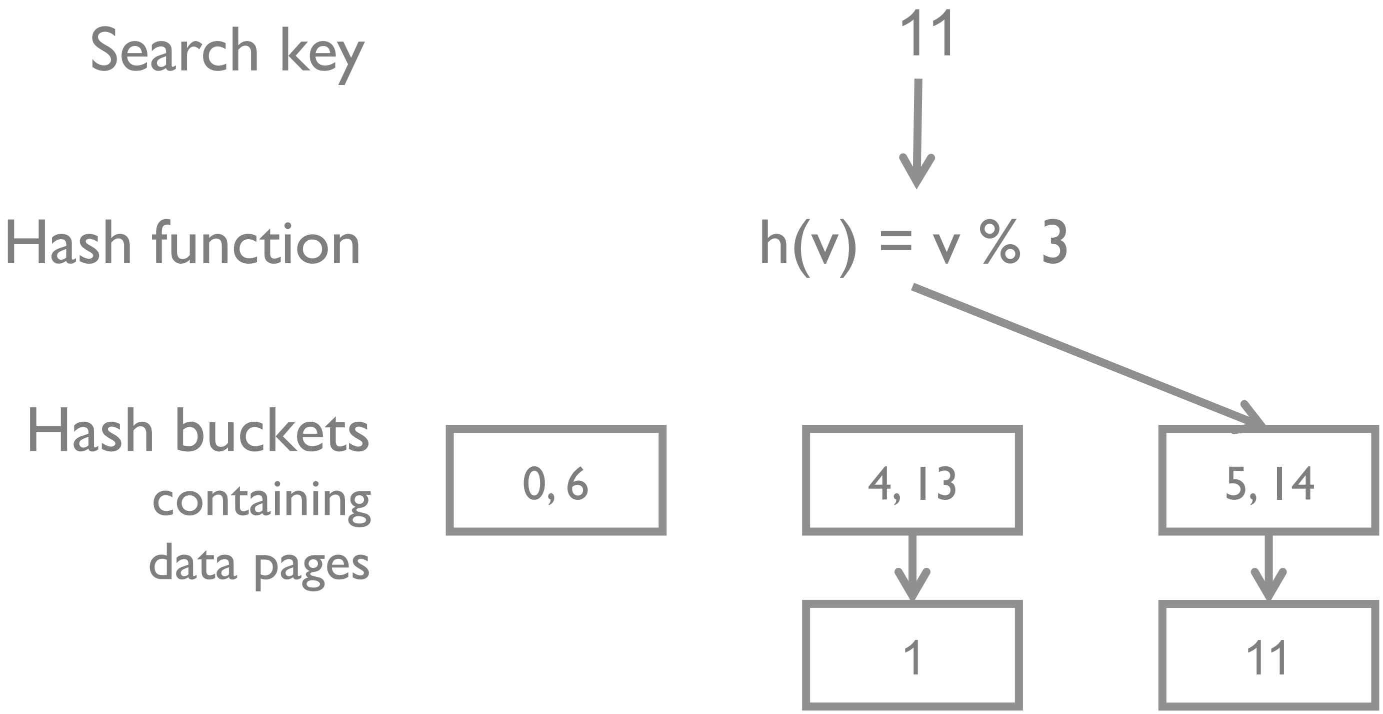
Search key: 11

Hash function: h(v) = v % 3

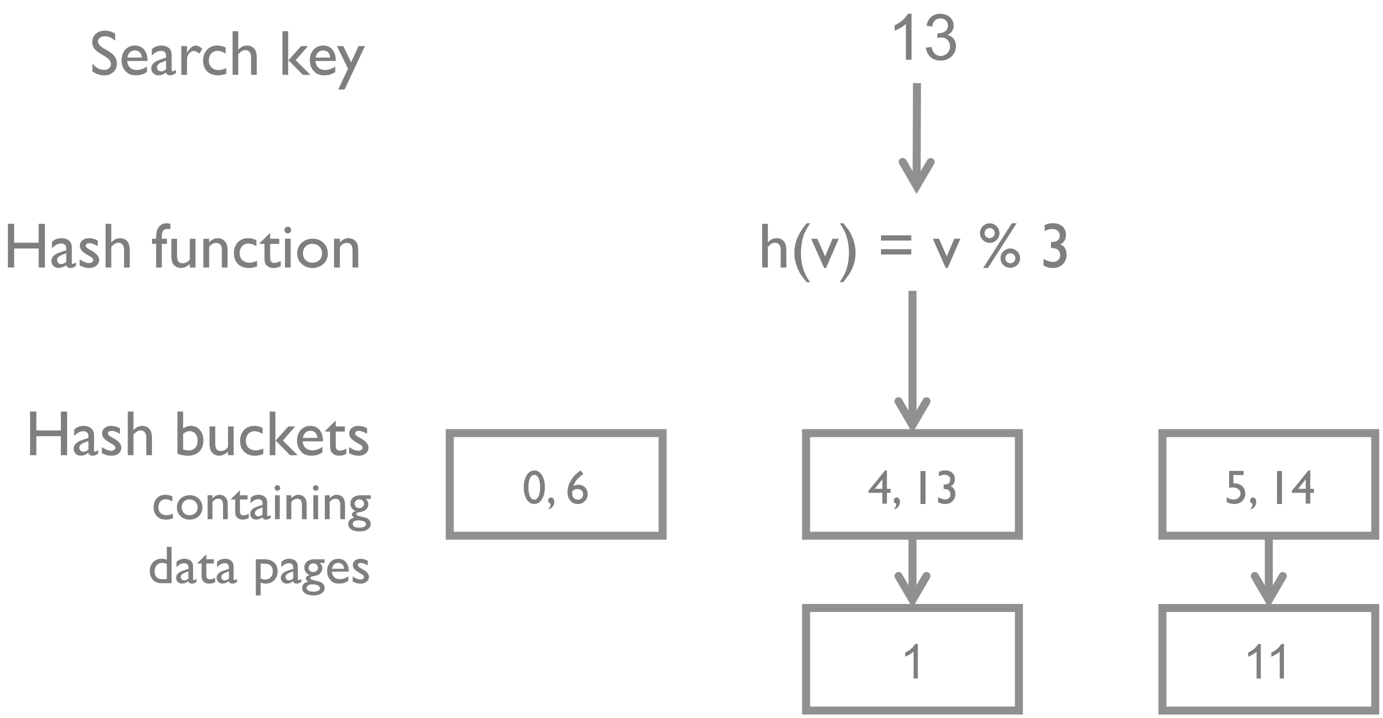
Hash buckets containing data pages



INSERT Hash Index on age



SEARCH Hash Index on age



Good for equality selections
Index = data pages + overflow data pages
Hash function h(v) takes as input the *search key*

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 10 < x 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 (avg)			
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

	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

	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

	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

no overflow

80% fill factor

B # *data* pages

D time to read/write page

M # pages in range query

How to pick?

Depends on your queries (workload)

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

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?

- updates are slower: upkeep costs
- takes up space

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*)

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