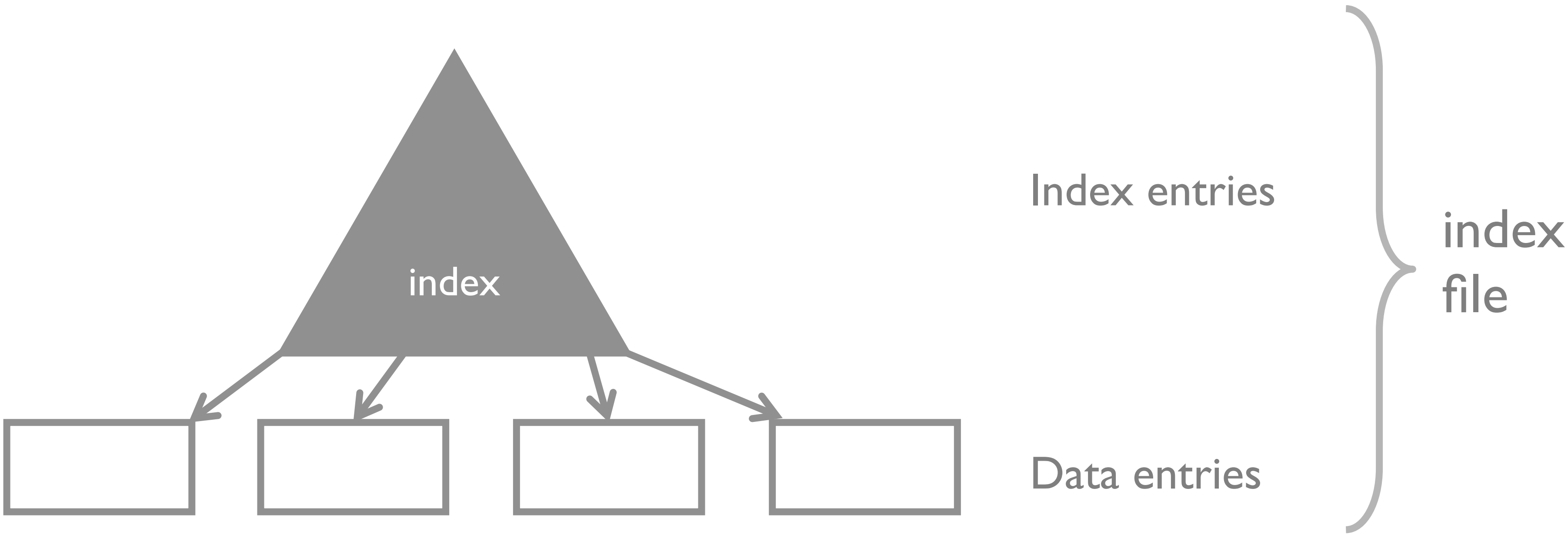
# L20
# Indexing Continued

# High level index structure




What is a data entry?

actual data record

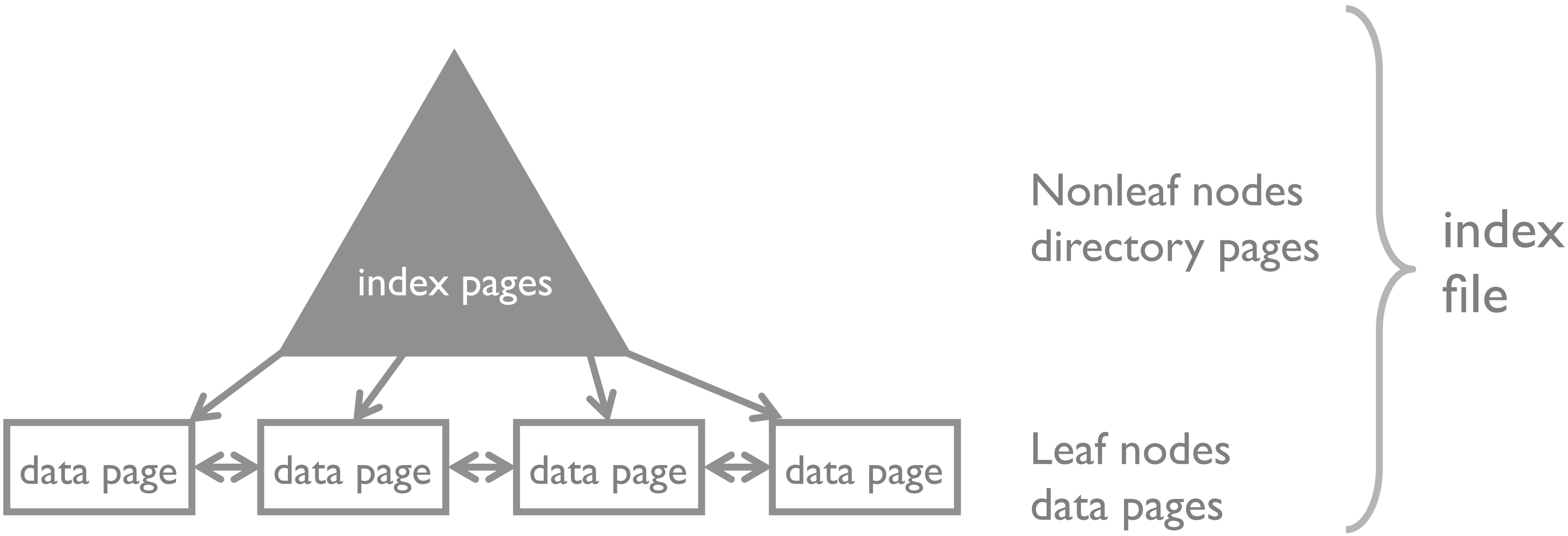
<search key value, rid>

Tradeoffs

directly access tuple.

compact, fixed size entries

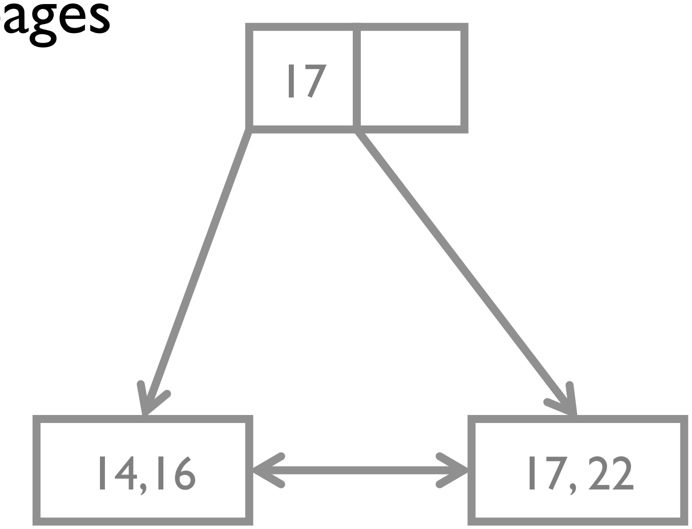
# B+ Tree Index




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

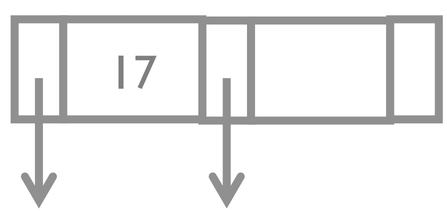



index & data
page contents
are in order

**Leaf Data pages**
data entries/tuples

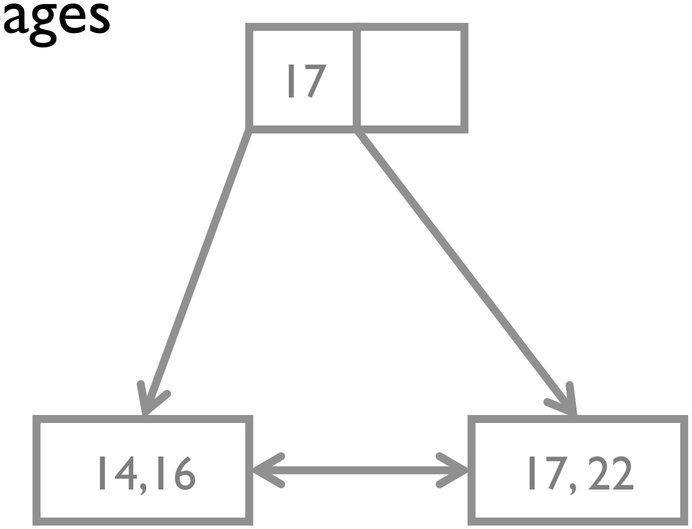
Query: SELECT * WHERE age= 14

directory page



# Index Only Queries: B+ Tree on (age)

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

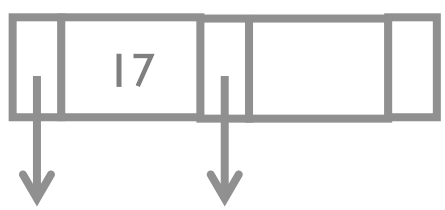



index & data
page contents
are in order

**Leaf Data pages**
data entries/tuples

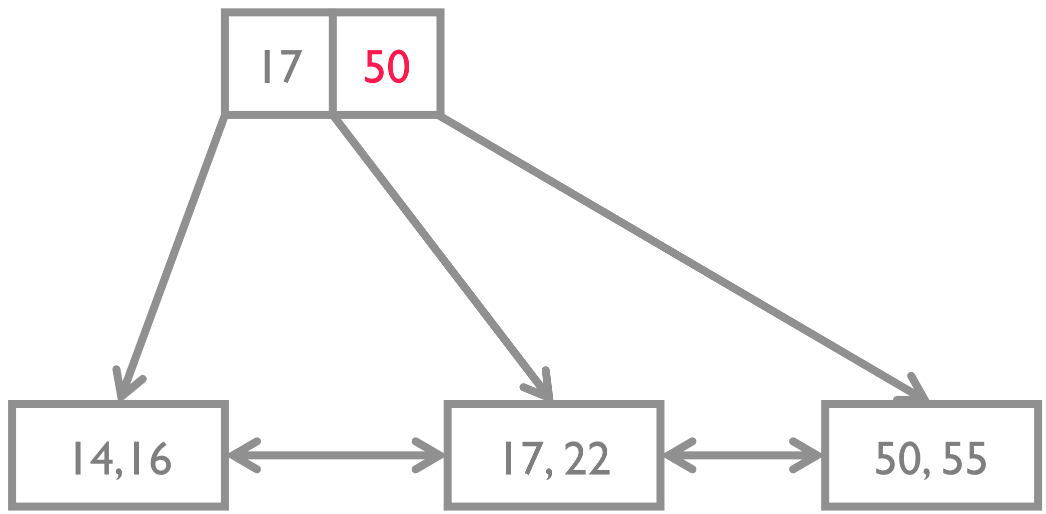
Query: SELECT age WHERE age = 14
(index only!)

directory page



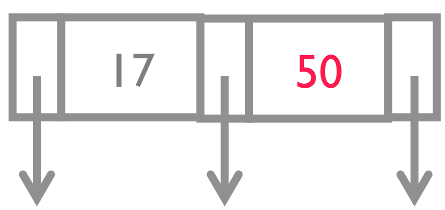
# B+ Tree on (age)

Note: 50 not a data entry

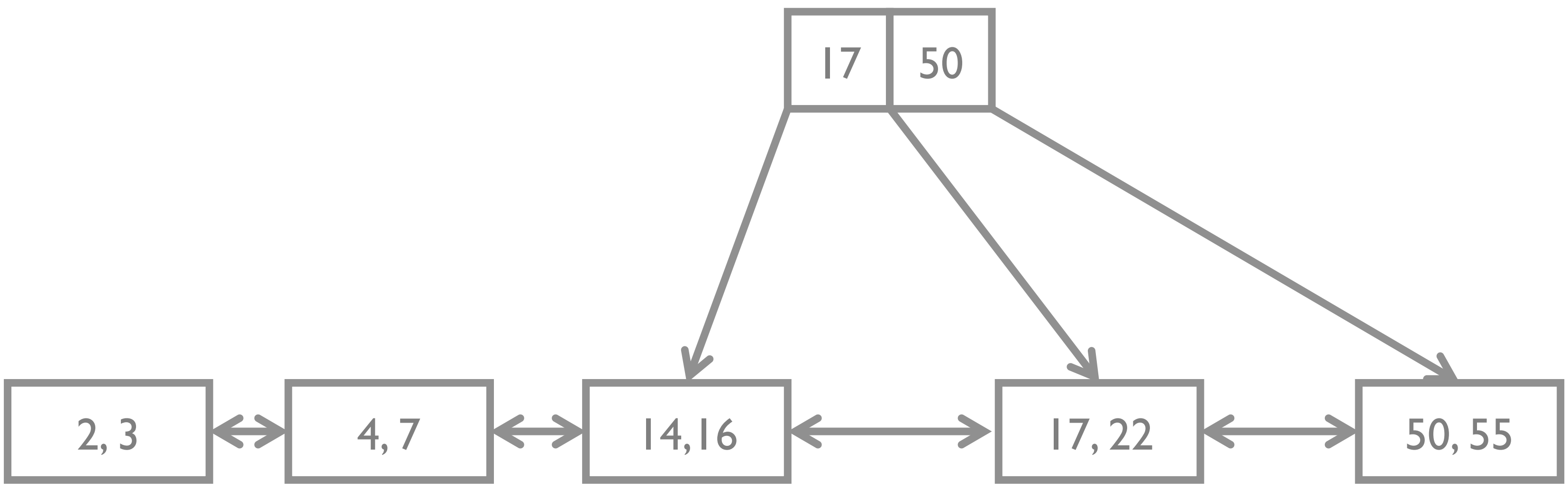



Query: SELECT * WHERE age = 50

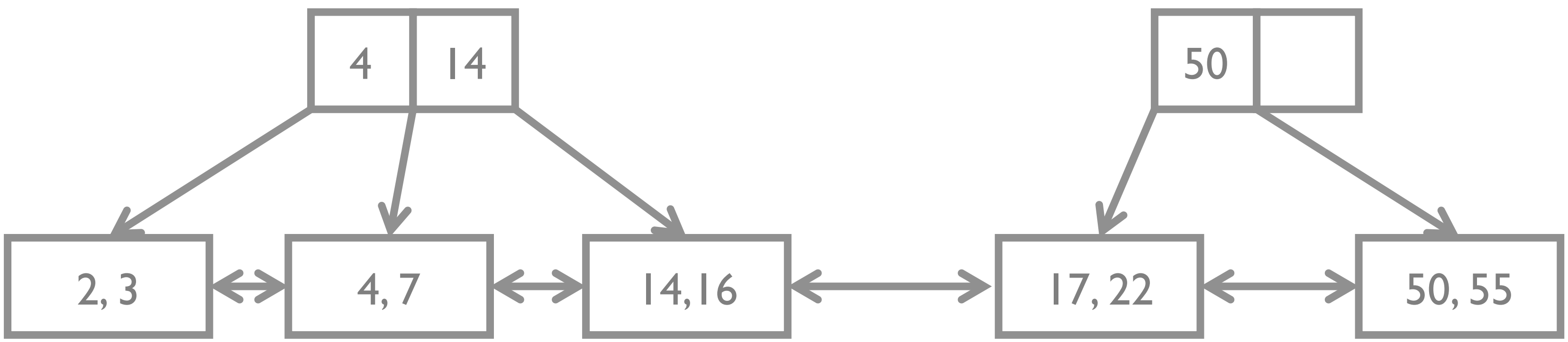
directory page



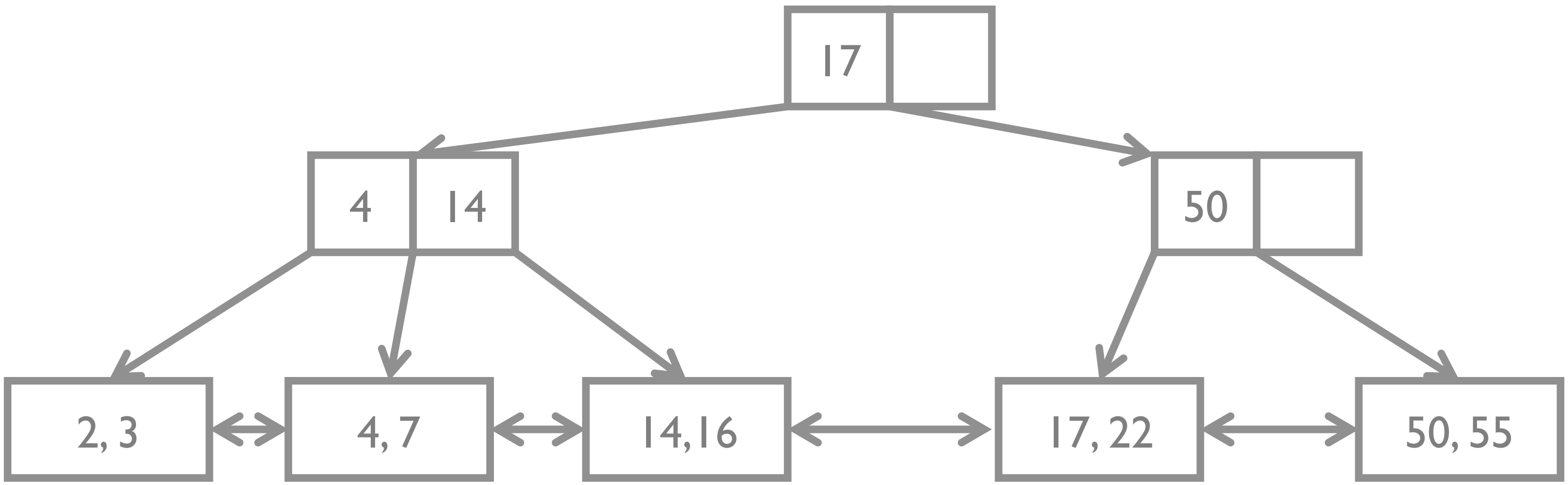
# B+ Tree on (age)



# B+ Tree on (age)

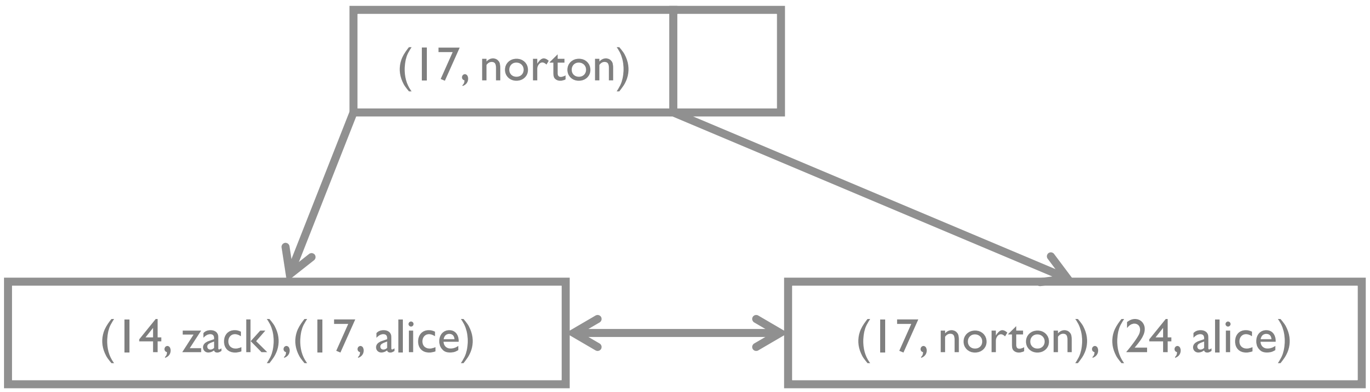


# B+ Tree on (age)




Query: SELECT * WHERE age > 15

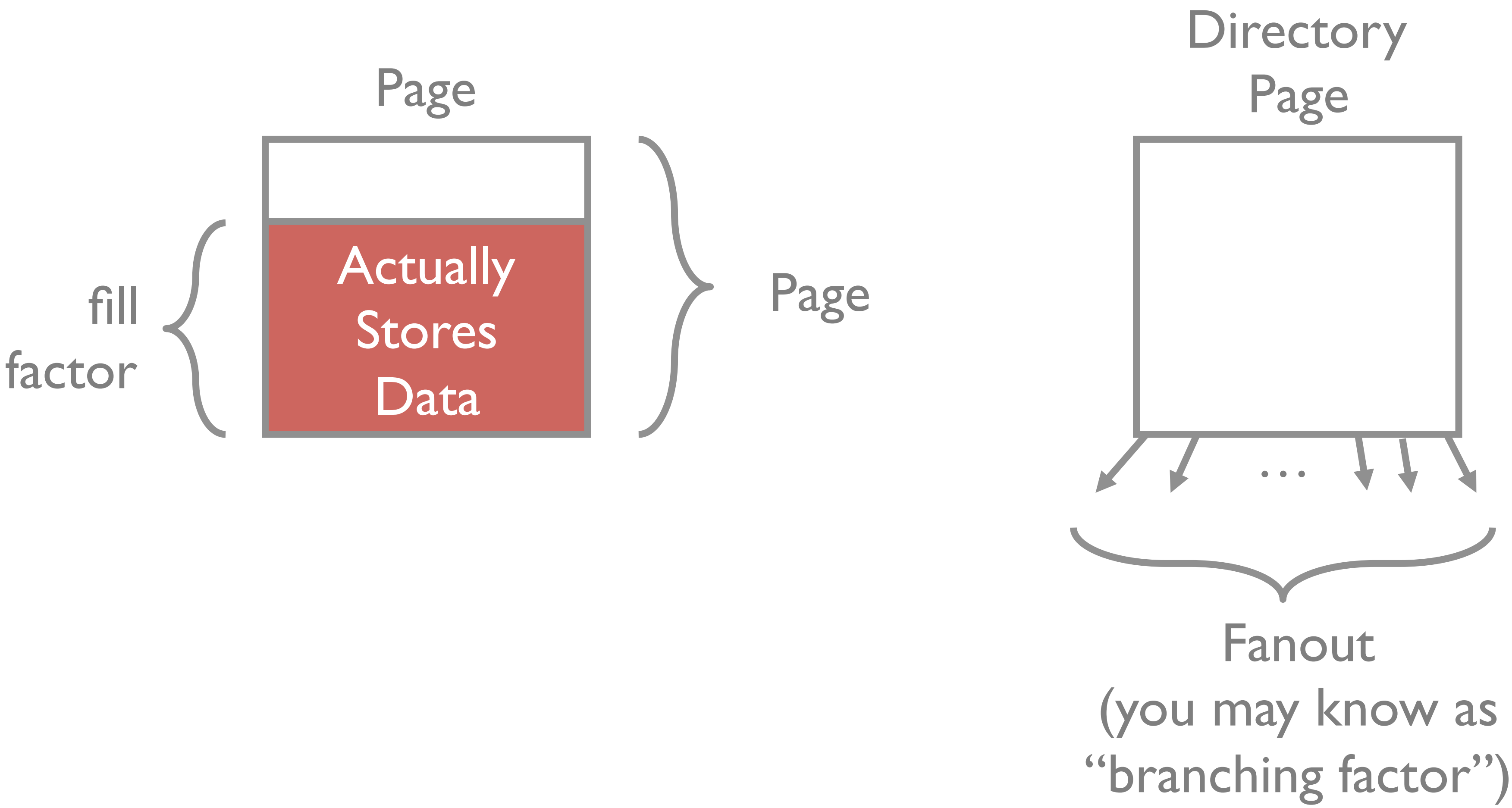
# B+ Tree on (age, name)




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

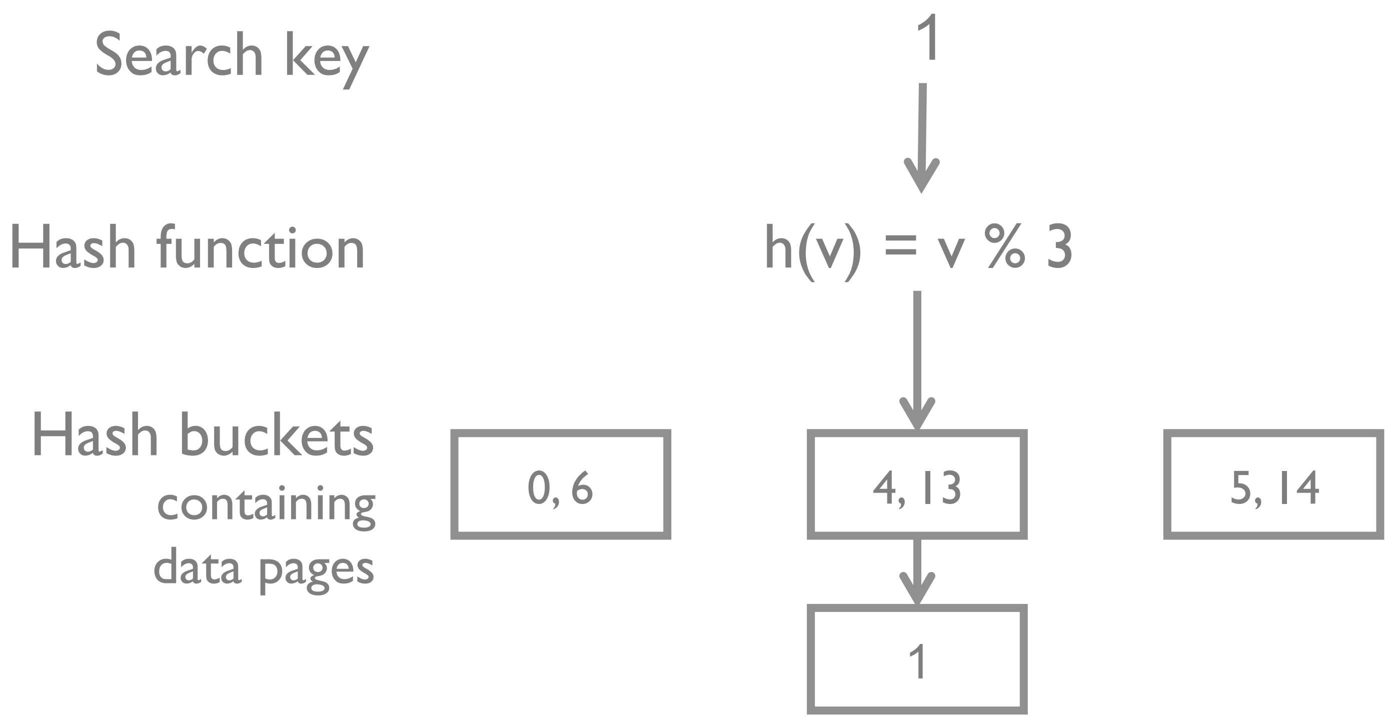
# Hash Index on age

Hash function h(v) = v % 3

Hash buckets containing data pages

| | | |
|---|---|---|
| 0, 6 | 4, 13 | 5, 14 |

# INSERT Hash Index on age

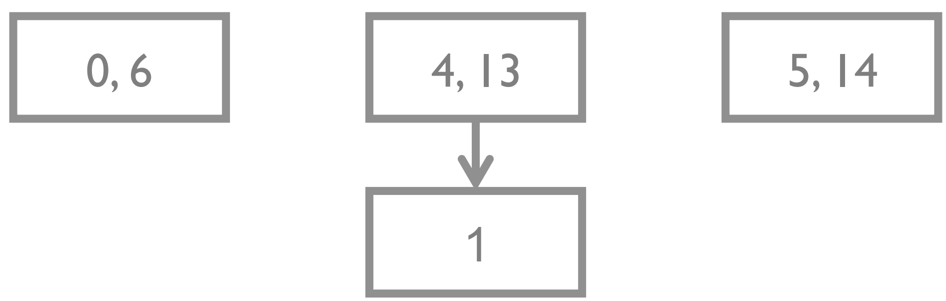


# INSERT Hash Index on age

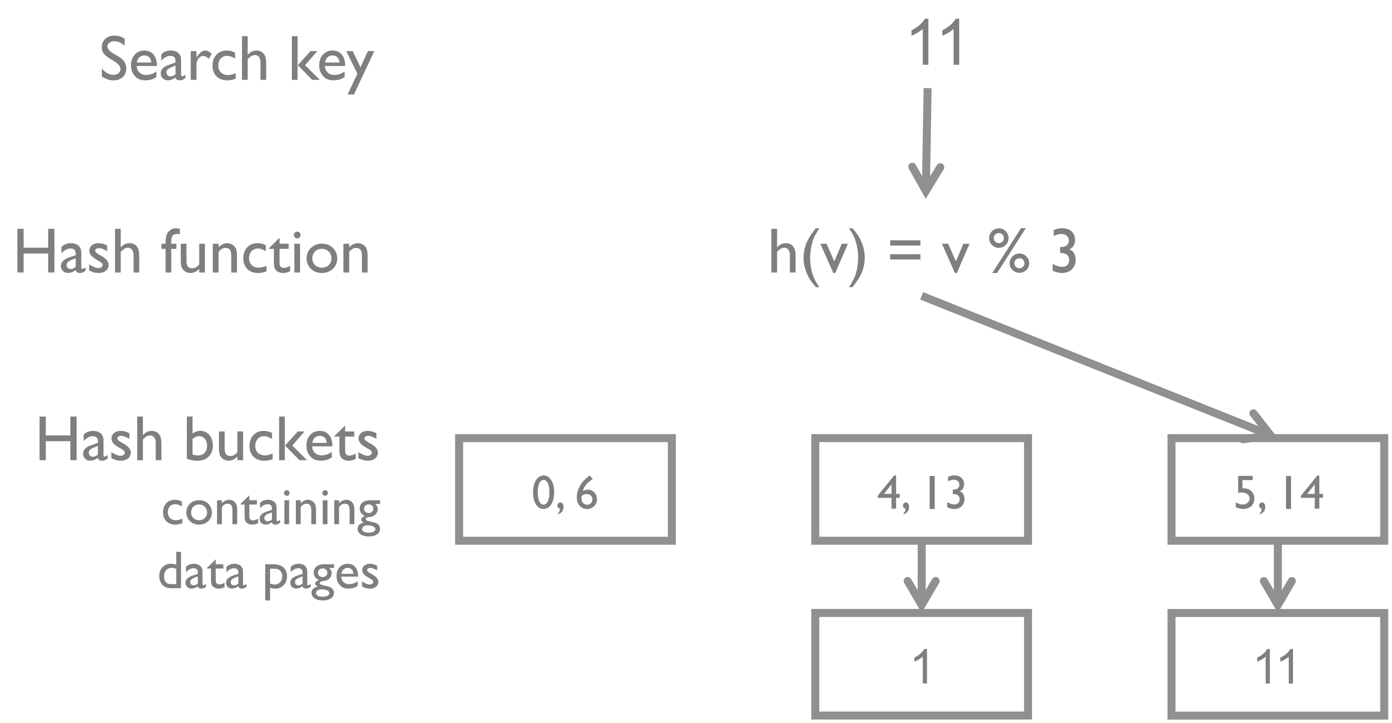
Search key 11

Hash function h(v) = v % 3

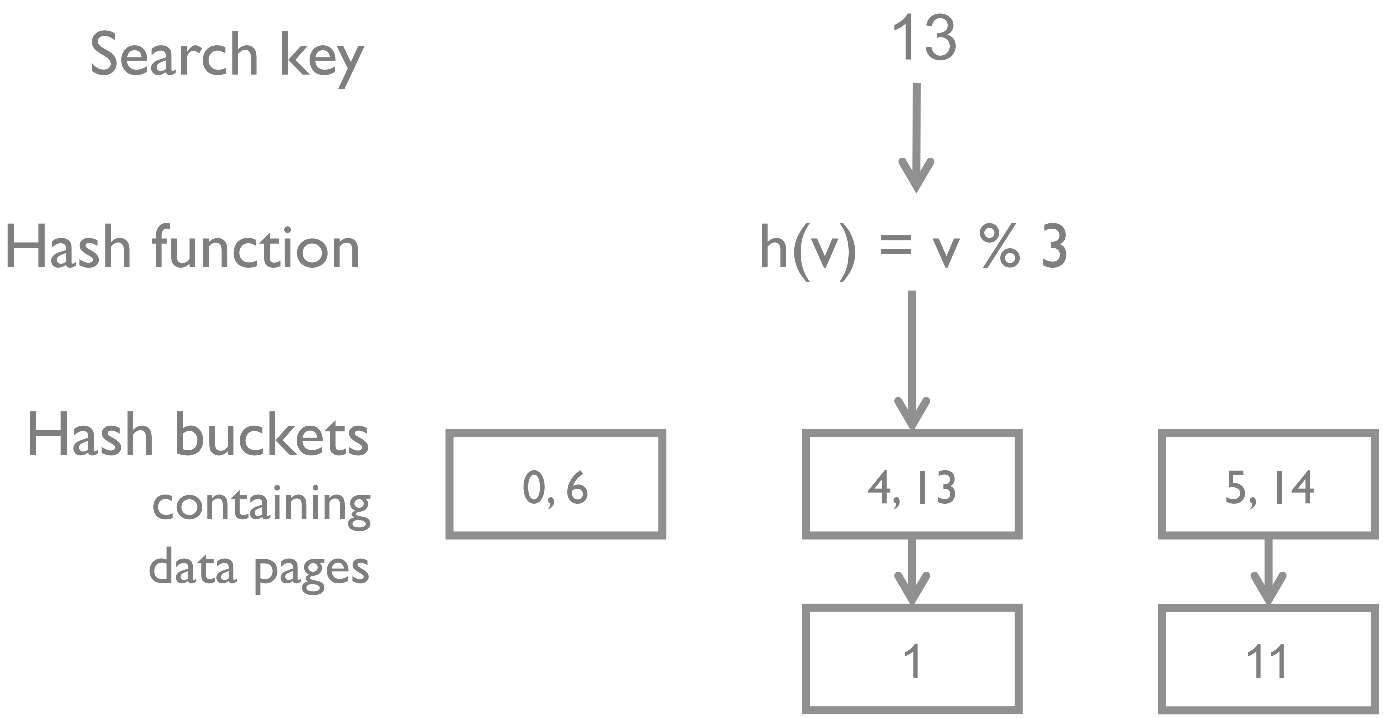
Hash buckets containing data pages



# 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

# L20
# Query Execution & Optimization

# Steps for a New Application

Requirements

what are you going to build?

Conceptual Database Design

pen-and-pencil description

Logical Design

formal database schema

Schema Refinement:

fix potential problems, normalization

Physical Database Design

optimize for speed/storage

Optimization

App/Security Design

prevent security problems

# Recall

Relational algebra

equivalence: multiple stmts for same query

some statements (much) faster than others

Which is faster?

a. $\sigma_{v=1}(R \times T)$

b. $\sigma_{v=1}(R) \times T$

$|R| = |T| = 10$ pages. 100? 1M?

# unique values in R = 1. 100? 1M? ← selectivity!

# Overview of Query Optimization

SQL → query plan

How plans are executed

Some implementations of operators

Cost estimation of a plan

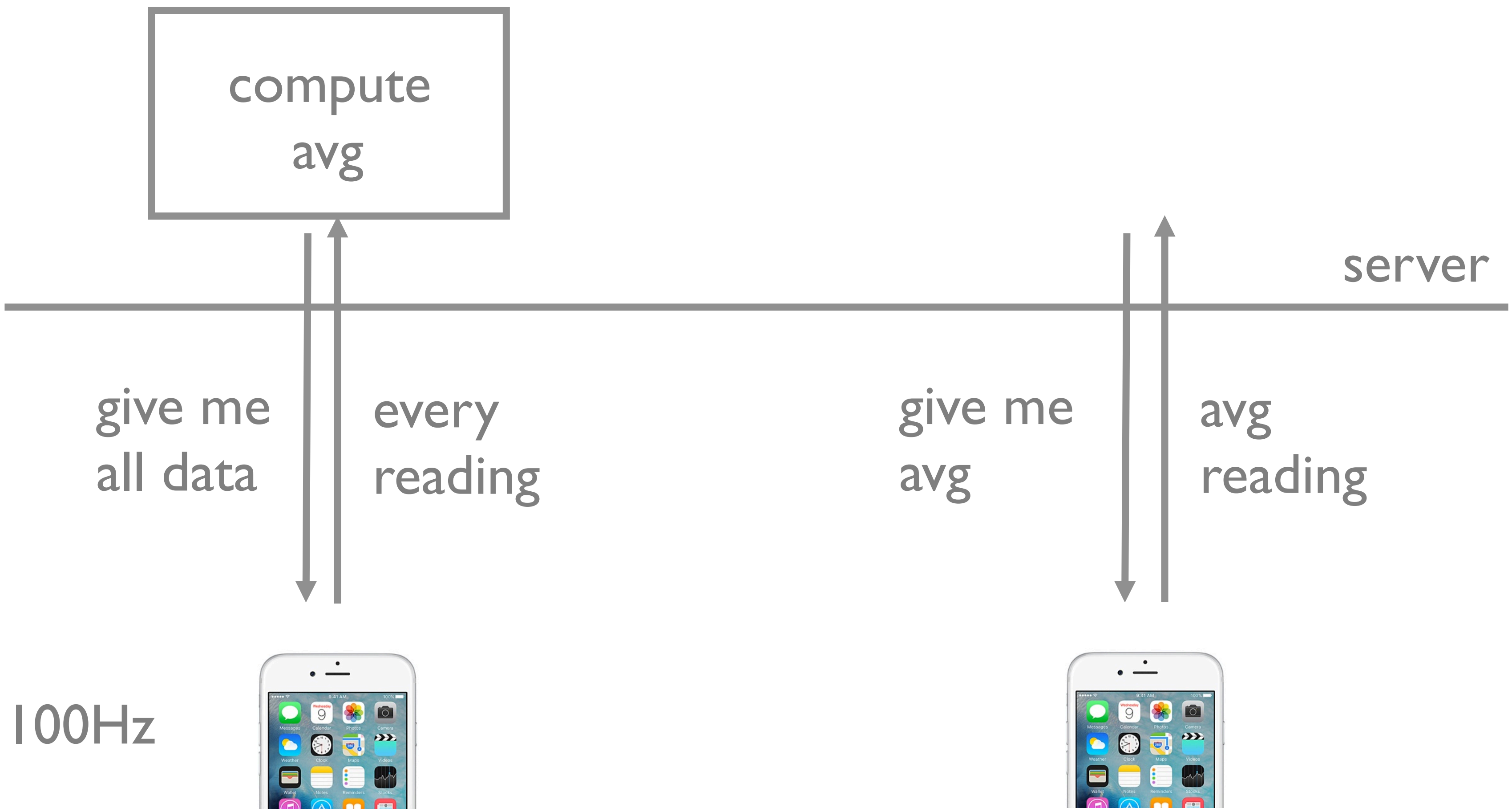
- Selectivity

System R dynamic programming

All ideas from System R's "Selinger Optimizer" 1979

# iPhones as a database

“avg acceleration over the past hour”



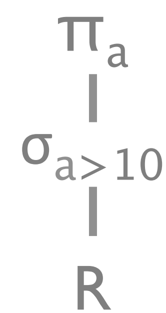
# SQL → Query Plan

SELECT a FROM R

$\pi_a(R)$

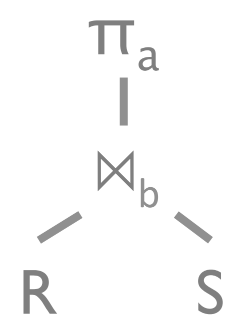

SELECT a FROM R
WHERE a > 10

$\pi_a(\sigma_{a>10}(R))$




SELECT a
FROM R JOIN S
ON R.b = S.b

$\pi_a(\bowtie_b(R,S))$



# Query Evaluation

Push vs Pull?

Push
- Operators are input-driven
- As operator (say reading input table) gets data, push it to parent operator.

Pull
- Operators are demand-driven
- If parent says "give me next result", then do the work

Are cursors push or pull?

# Query Evaluation

Naïve execution (operator at a time)

- read R
- filter a>10 and write out
- read and project a
- Cost: B + M + M

SELECT a
FROM R
WHERE a > 10

$$\pi_a \\ | \\ \sigma_{a>10} \\ | \\ R$$

B # *data* pages

M # pages matched in WHERE clause

Could we do better?

# Query Evaluation

Pipelined exec (tuple/page at a time)

- read first page of R, pass to $\sigma$
- filter a > 10 and pass to $\pi$
- project a
- (all operators run concurrently)
- Cost: B

Note: can't pipeline some operators!

e.g., sort, some joins, aggregates

why?

SELECT a
FROM R
WHERE a > 10

$$\begin{array}{c} \pi_a \\ | \\ \sigma_{a>10} \\ | \\ R \end{array}$$

B # *data* pages

M # pages matched in WHERE clause

# Query Evaluation

What if R is indexed?

- Hash index
  - Not appropriate
- B+Tree index
  - use a>10 to find initial data page
  - scan leaf data pages
  - Cost: $\log_F B + M$

SELECT a
FROM R
WHERE a > 10

$\pi_a$
|
$\sigma_{a>10}$
|
R

B # *data* pages

M # pages matched in WHERE clause

# Access Paths

Choice of how to access input data is called the Access Path

- file scan or
- index + matching condition (e.g., a > 10)

# Access Paths

Sequential Scan

doesn't accept any matching conditions

Hash index search key <a,b,c>

accepts conjunction of equality conditions on *all* search keys

e.g., a=1 and b = 5 and c = 5

will (a = 1 and b = 5) work? why?

Tree index search key <a,b,c>

accepts conjunction of terms of *prefix* of search keys

typically best with equality on all but last column

e.g., a = 1 and b = 5 and c < 5

will (a = 1 and b > 5) work?

will (a > 1 and c > 9) work?

# How to pick Access Paths?

Selectivity

ratio of # outputs satisfying predicates vs # inputs

0.01 means 1 output tuple for every 100 input tuples

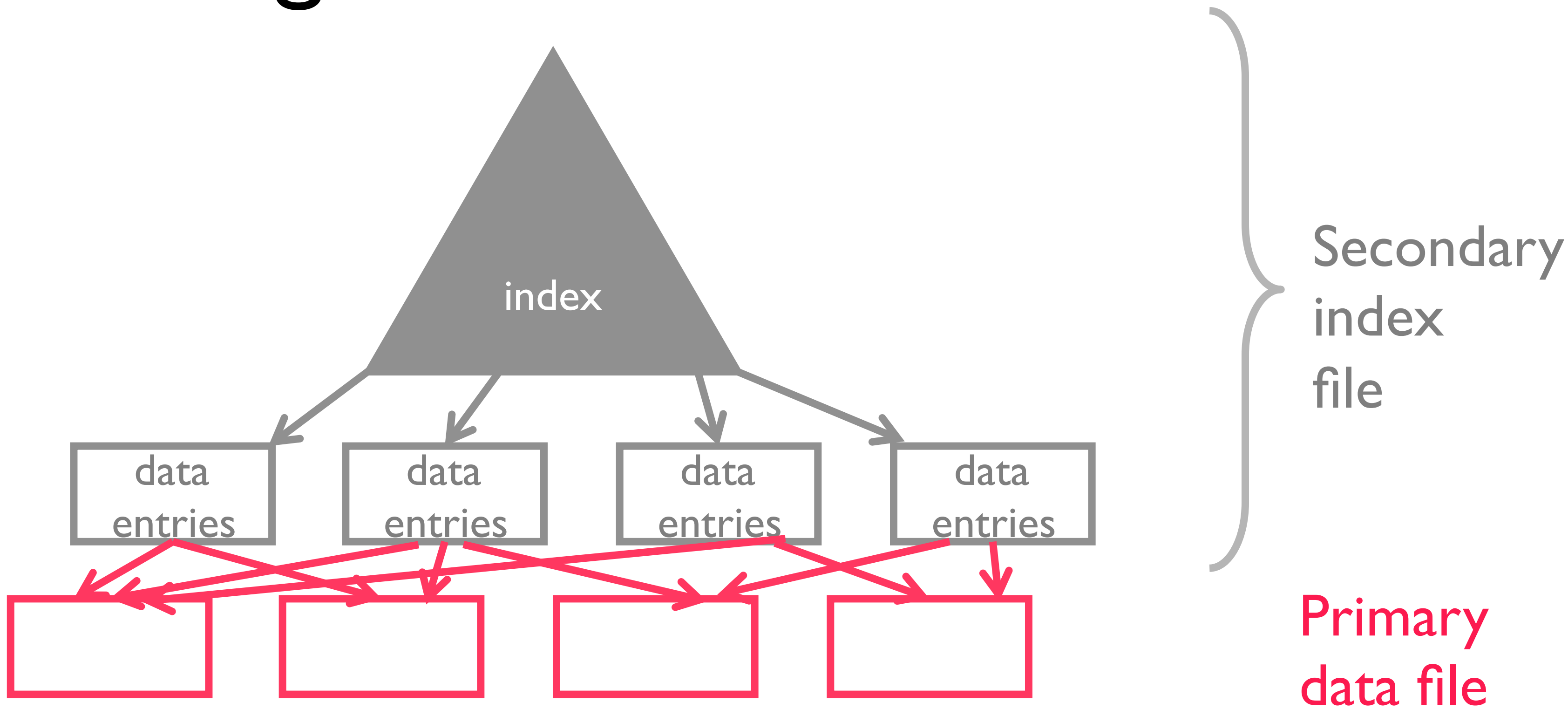
Assume

attribute selectivity is independent

if selectivity(a=1) = 0.1, selectivity(b>3) = 0.6

selectivity(a=1 and b>3) = 0.1*0.6 = 0.06

# High level index structure




What is a data entry?

- actual data record
- <search key value, rid>
- <search key value, rid_list>

# How to pick Access Paths?

Hash index on <a, b, c>

a = 1, b = 1, c = 1 how to estimate selectivity?

1. pre-compute attribute statistics by scanning data

   e.g., a has 100 values, b has 200 values, c has 1 value

   selectivity = 1 / (100 * 200 * 1)

2. How many distinct values does hash index have?

   e.g., 1000 distinct values in hash index

3. make a number up

   "default estimate" is the fancy term

# System Catalog Keeps Statistics

System R

| | |
|---|---|
| NCARD | "relation cardinality” # tuples in relation |
| TCARD | # pages relation occupies |
| ICARD | # keys (distinct values) in index |
| NINDX | pages occupied by index |

min and max keys in indexes

Statistics were expensive in 1979!

Super elegant: catalog stored in relations too!

# What Optimization Options Do We Have?

Access Path ✔

Predicate push-down

Join implementation

Join ordering

In general, depends on operator implementations. So let's take a look