L19: Indexing and Tuning

CS3200 Database design (sp18 s2)

https://course.ccs.neu.edu/cs3200sp18s2/

3/26/2018

Announcements!

- We are handing your exams back before and after class. Please pick it up in approximately alphabetical order
- Outline today
  - Exam 2 discussion
  - Indexing, B-trees

Exam 2

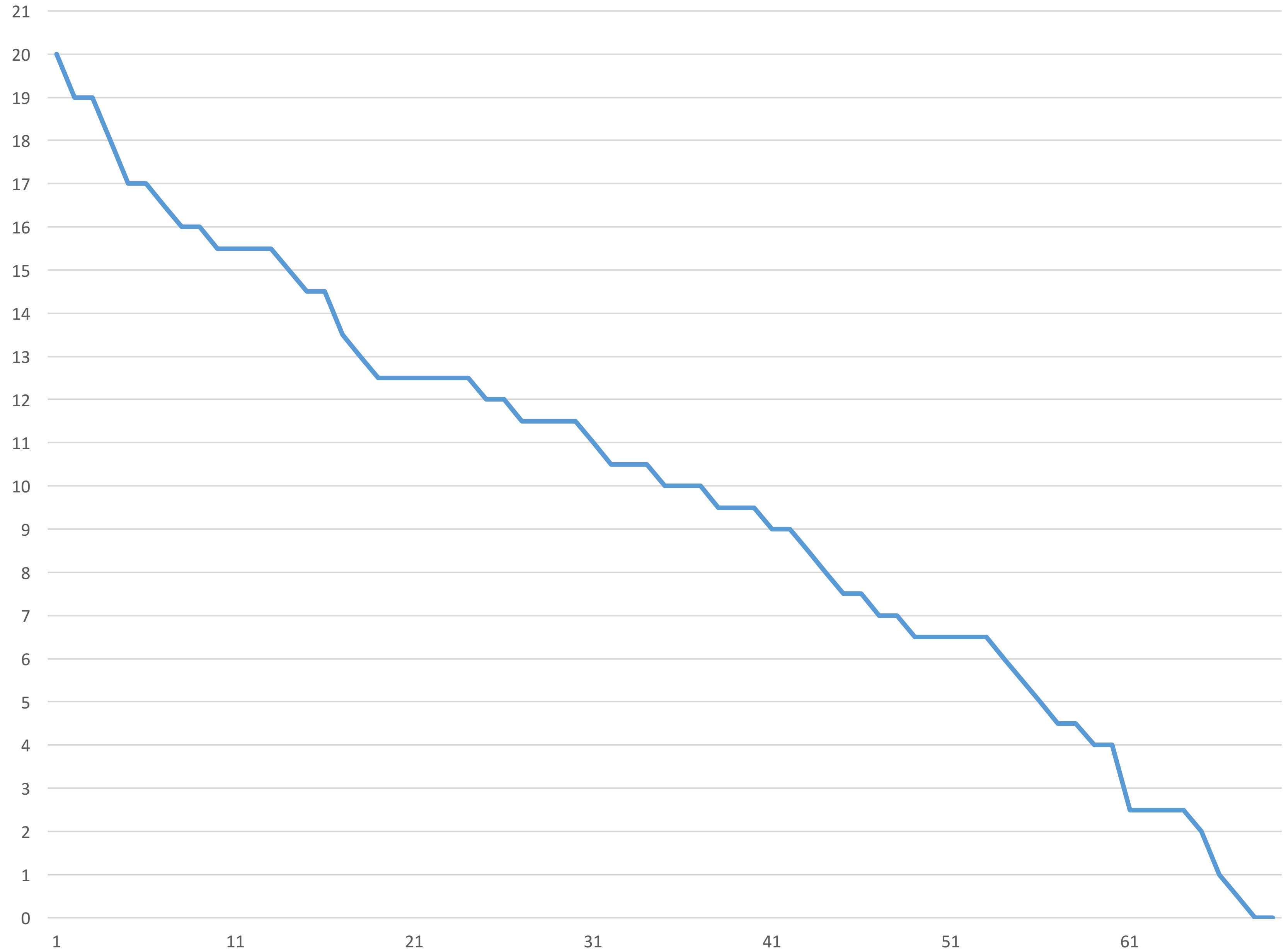
100% = 19

50% = 0

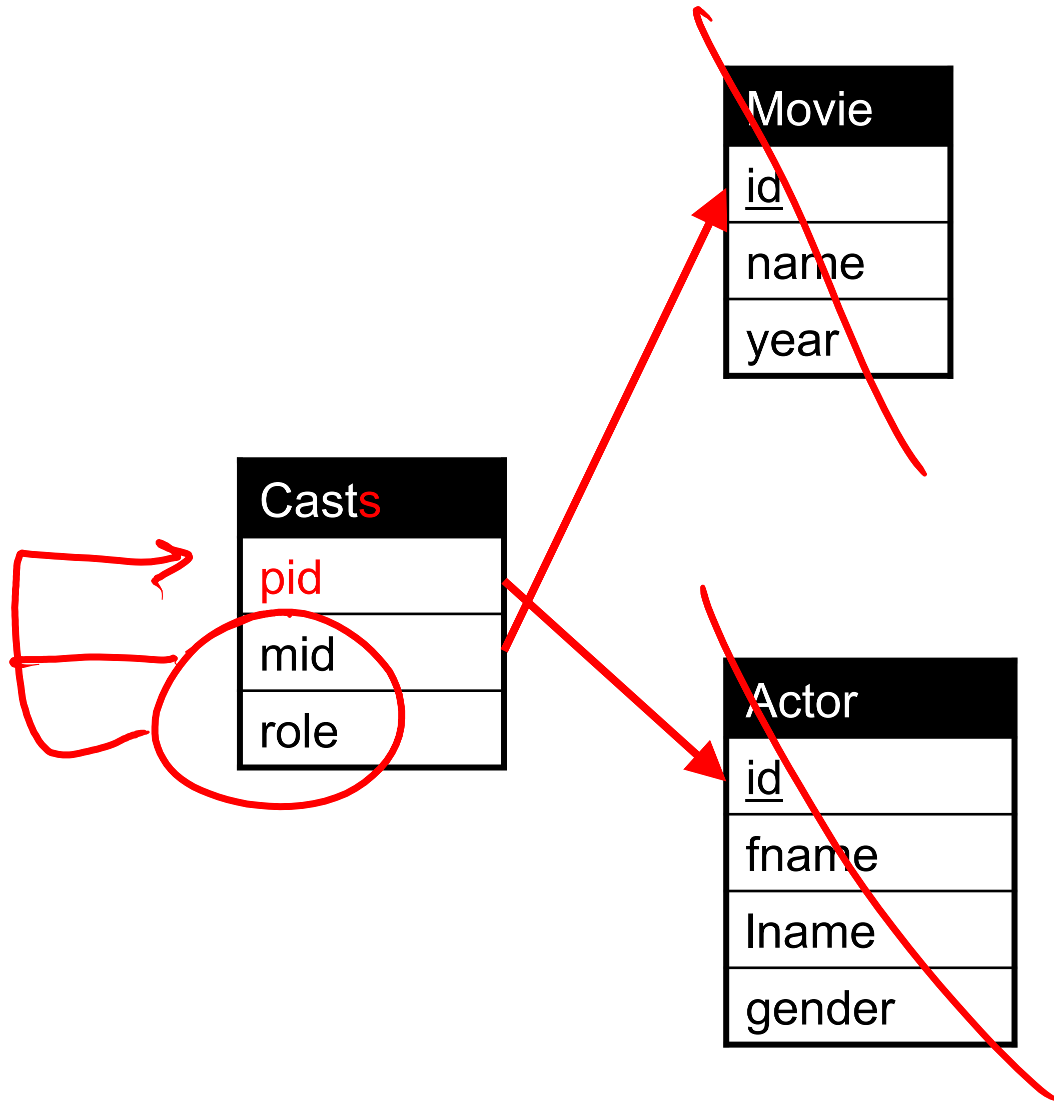
Exam 1

100% = 19

50% = 0



Exam2



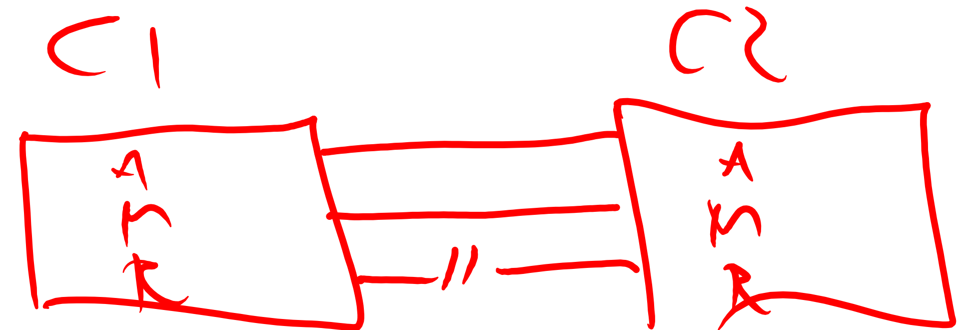
Find a minimal set of attributes in Casts that can serve as candidate key

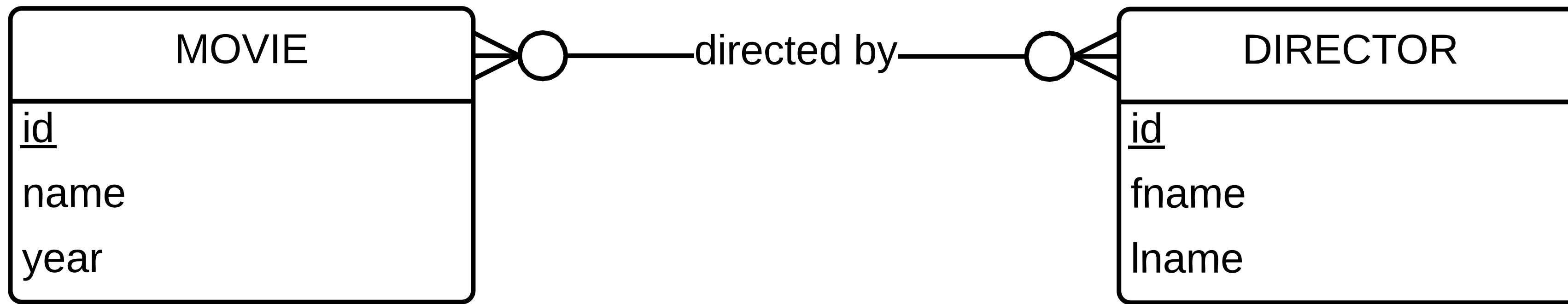
Assume that any FD that you observe in this example data set also holds in general.

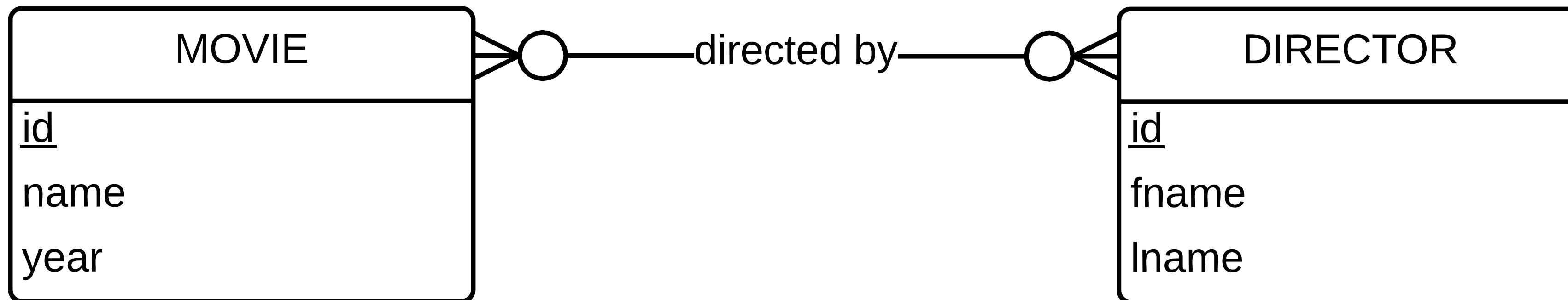
```
select mid, role, count(*)
from casts
group by mid, role;
```

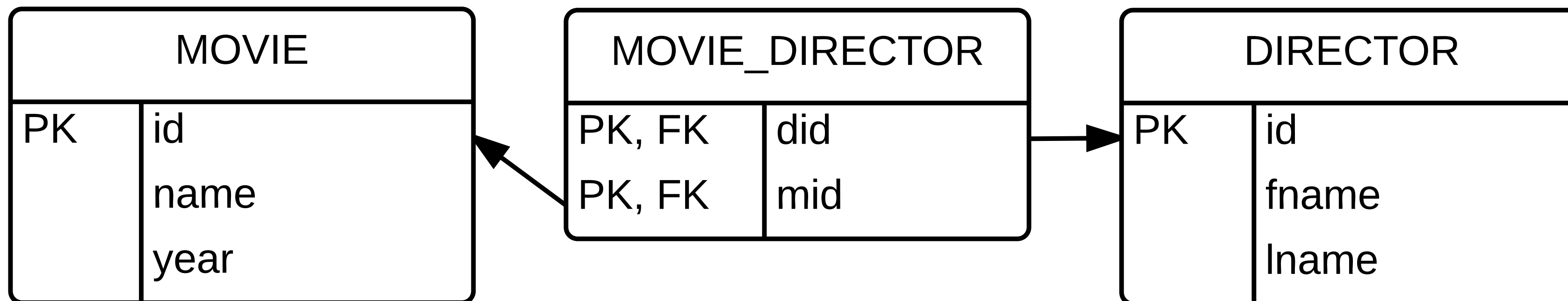
```
select count(*)
from casts c1, casts c2
where c1.mid = c2.mid
AND c1.aid = c2.aid
AND c1.role != c2.role;
```

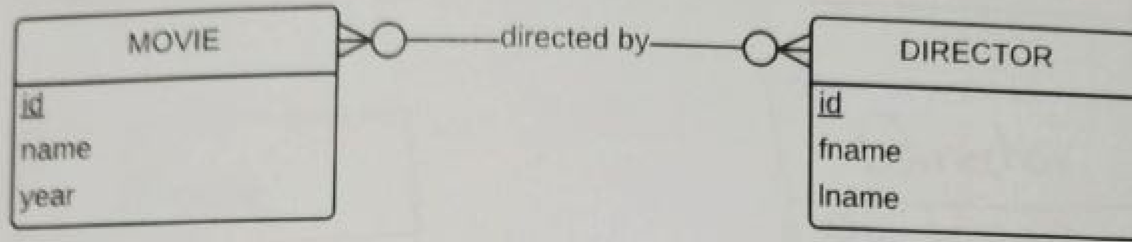
```
select mid, role, count(*)
from casts
group by mid, role
having count(*) > 1
```

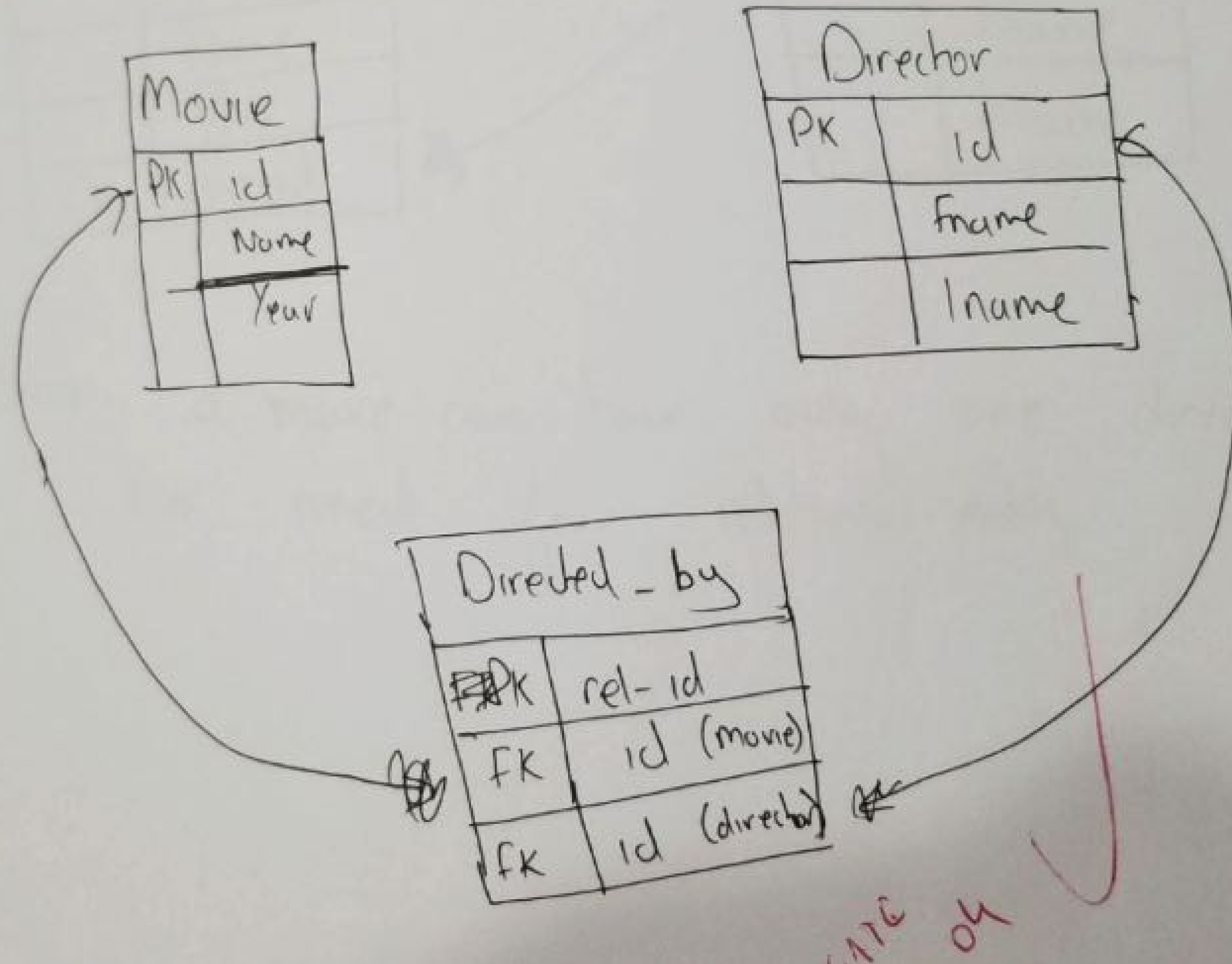


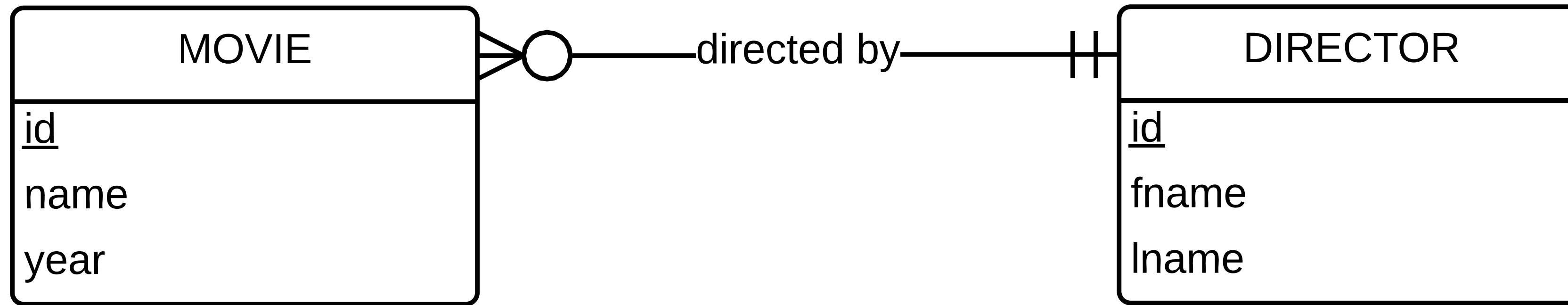

MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname

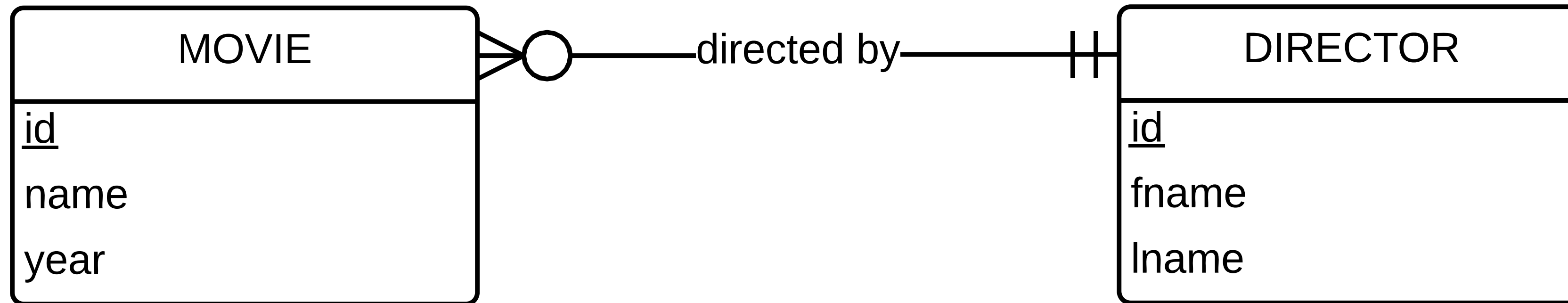

MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname

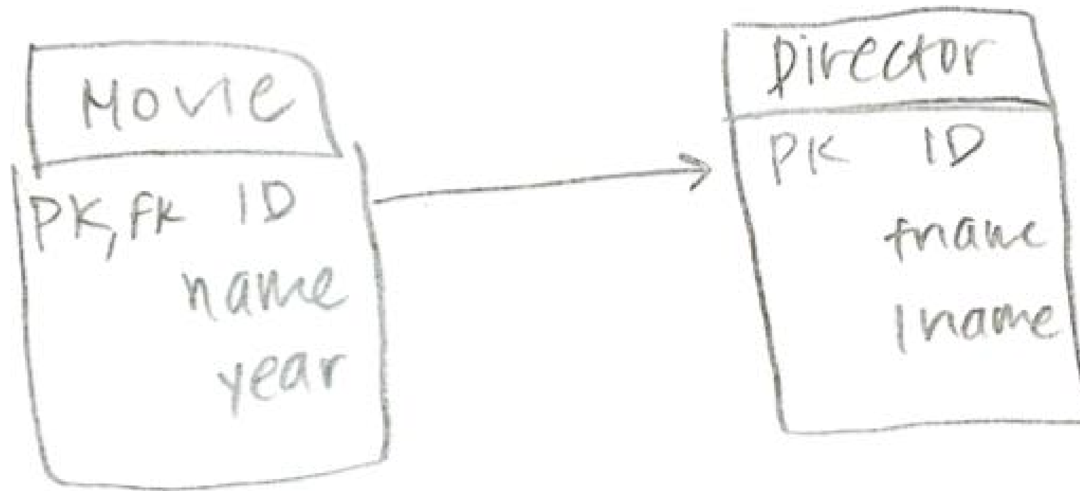

MOVIE
PK
id
name
year
MOVIE_DIRECTOR
PK, FK
did
PK, FK
mid
DIRECTOR
PK
id
fname
lname

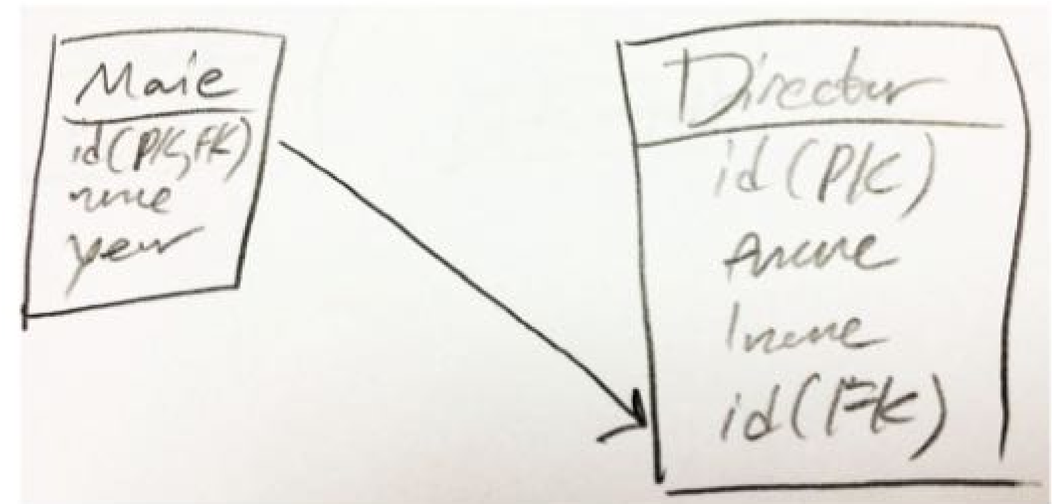

MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname

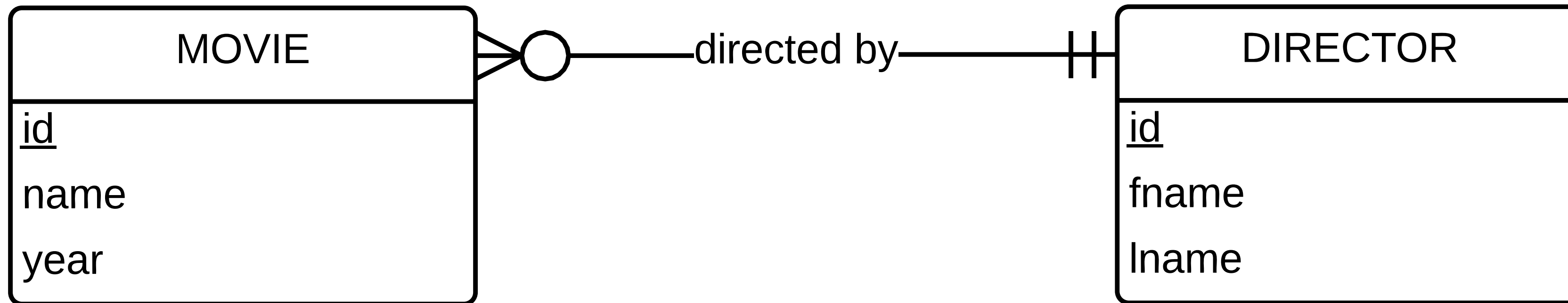

Movie
PK
id
Name
Year
Director
PK
id
fname
lname
Directed_by
PK
rel-id
FK
id (movie)
FK
id (director)

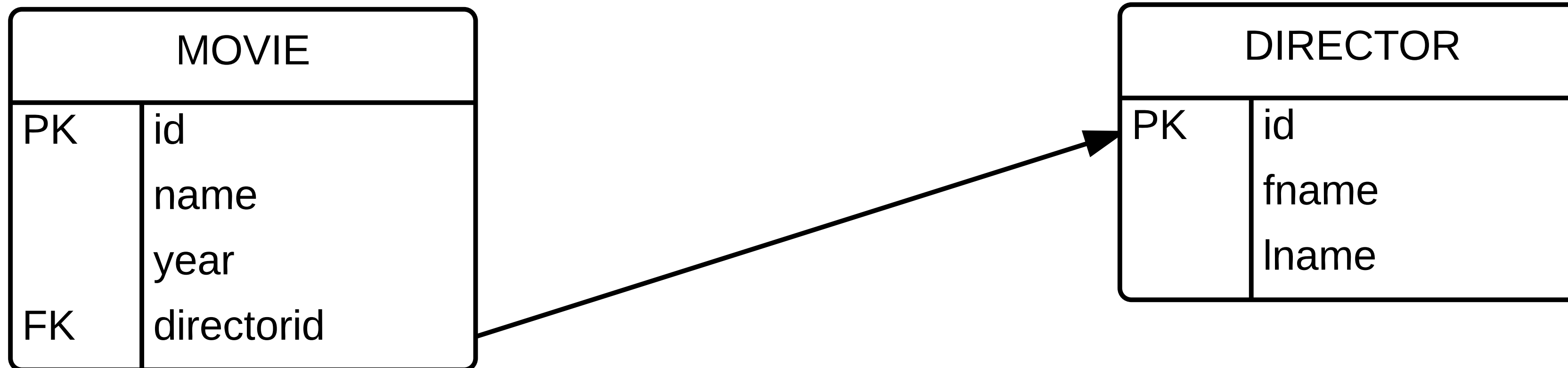

MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname


MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname


Movie
PK,FK ID
name
year
Director
PK ID
fname
lname


id(PK,FK)
year
Director
id (PK)
id(FK)


MOVIE
id
name
year
directed by
DIRECTOR
id
fname
lname

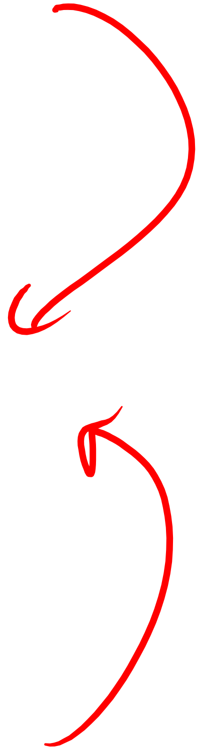

MOVIE
PK
id
name
year
FK
directorid
DIRECTOR
PK
id
fname
lname

Transform the ERD into the appropriate schema

NURSE
nurseID
name
DOB

date_assigned

CARE CENTER
centerID
location



NURSE

PK	nurseID
	name
	DOB

CARECENTER

PK	centerID
	location
FK	nurseID
	date_assigned

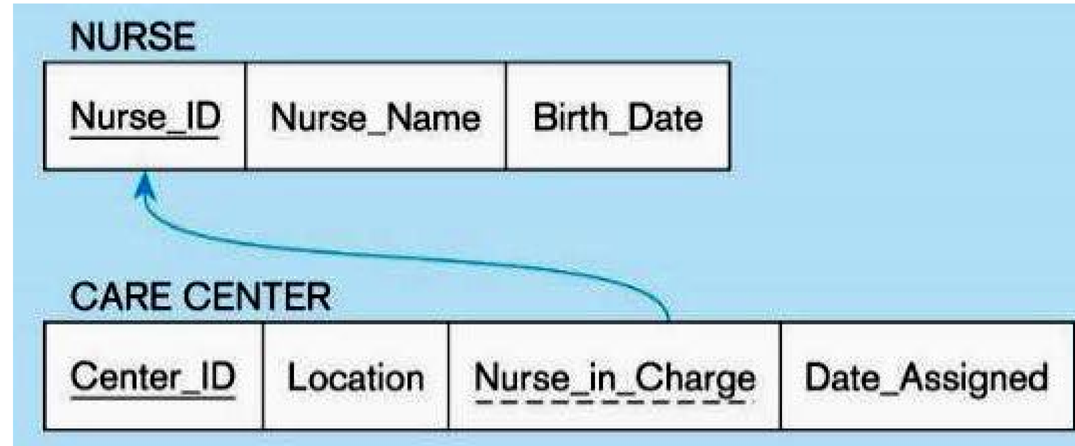
NURSE
nurseID
name
DOB

date_assigned

CARE CENTER
centerID
location

Slide 242

Nurses: Instance



Nurse

nid	name	birth_date
1	Alice	1/1/1980
2	Bob	1/1/1970
3	Clarissa	1/1/1975
4	Dora	1/1/1972

Carecenter

cid	location	nurseid@	date_assigned
1	Boston	1	1/1/2016
2	New York	3	3/1/2016

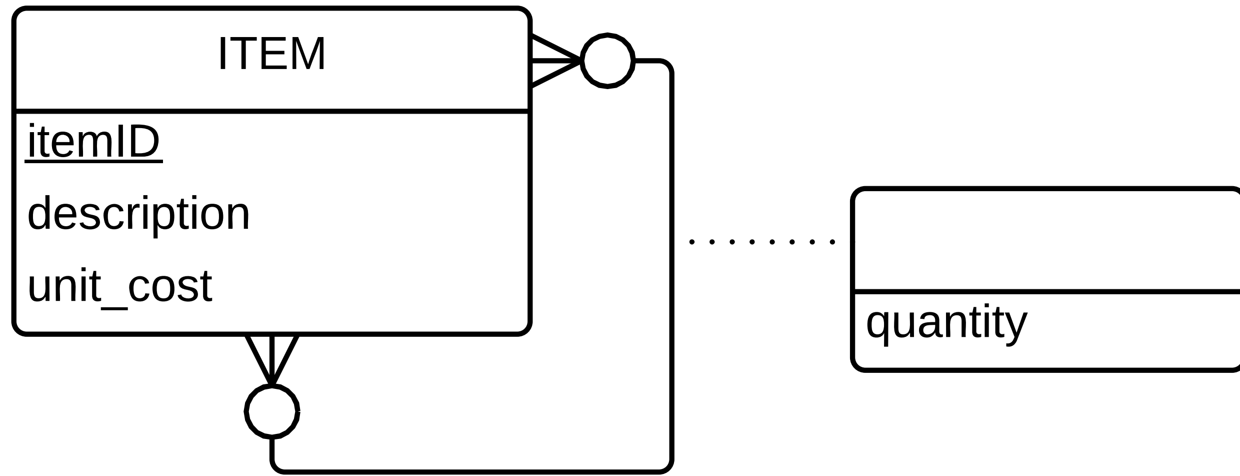
Nurse

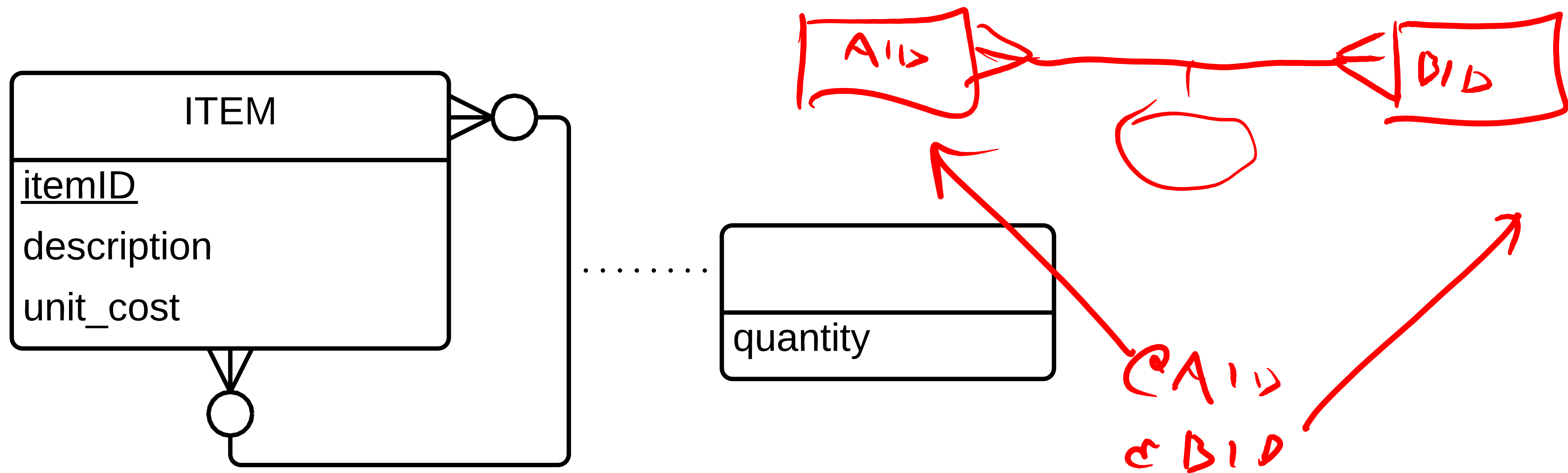
nid	name	birth_date	carecenter@	date_assigned
1	Alice	1/1/1980	1	1/1/2016
2	Bob	1/1/1970	NULL	NULL
3	Clarissa	1/1/1975	3	3/1/2016
4	Dora	1/1/1972	NULL	NULL

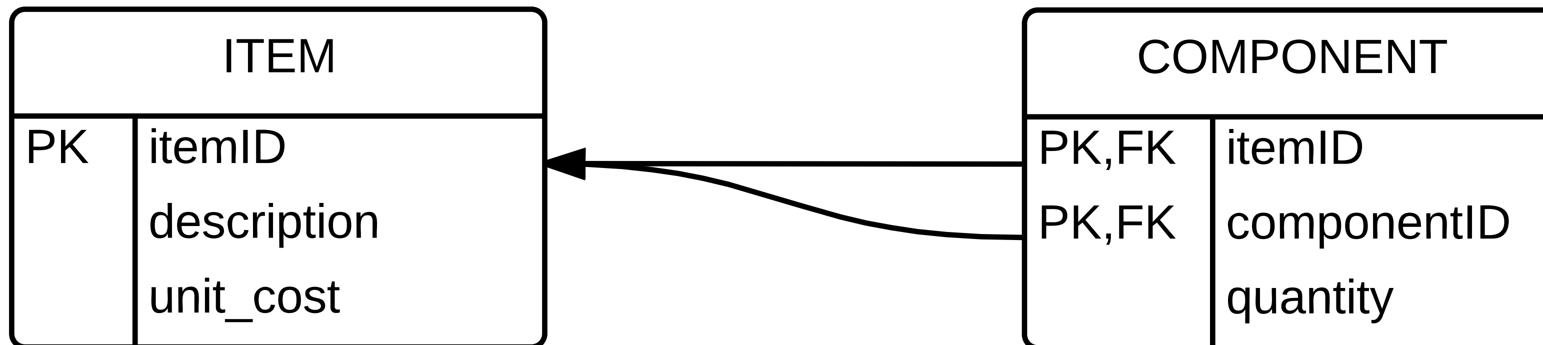
Carecenter

cid	location
1	Boston
2	New York

Slide 243

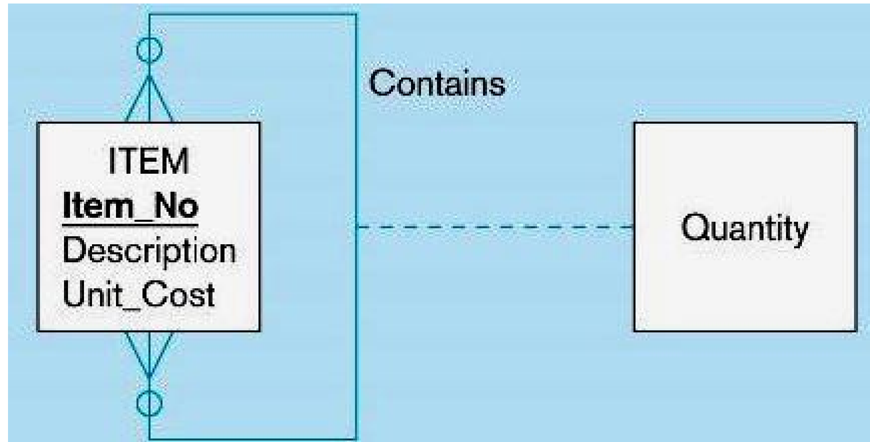

ITEM
itemID
description
unit_cost
quantity


ITEM
itemID
description
unit_cost
quantity


ITEM
PK
itemID
description
unit_cost
COMPONENT
PK,FK
itemID
PK,FK
componentID
quantity

2) Mapping a Unary M:N Relationship

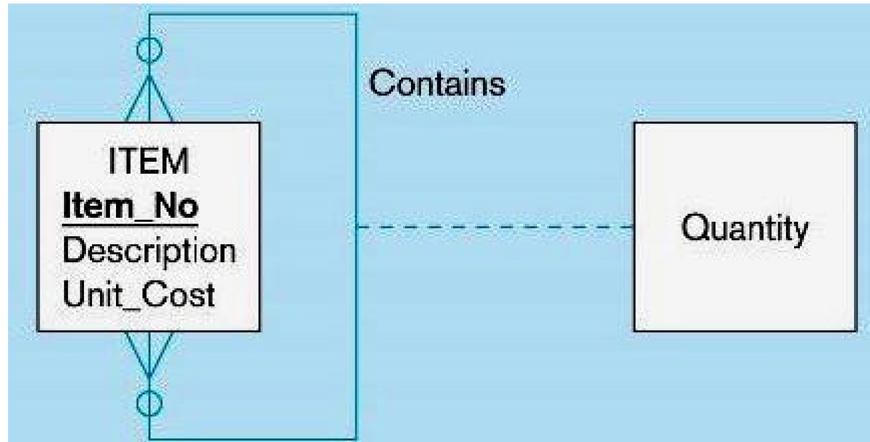
Bill-of-materials relationships (M:N)



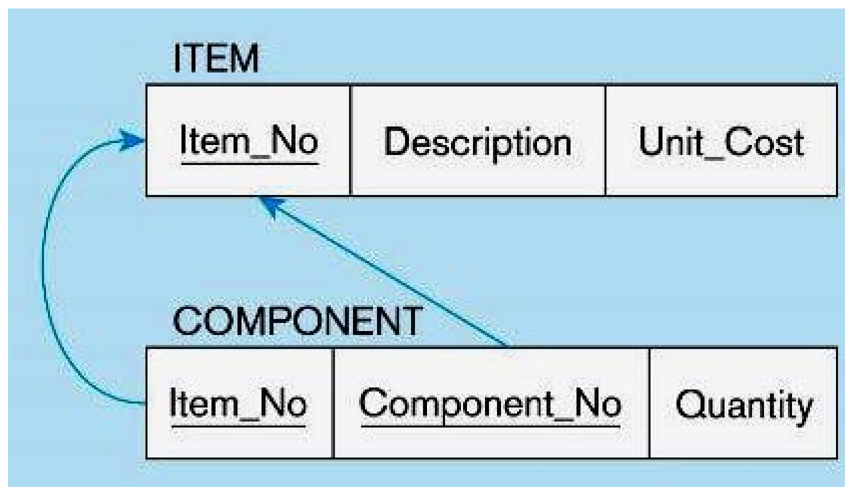
Slide 249

2) Mapping a Unary M:N Relationship

Bill-of-materials relationships (M:N)



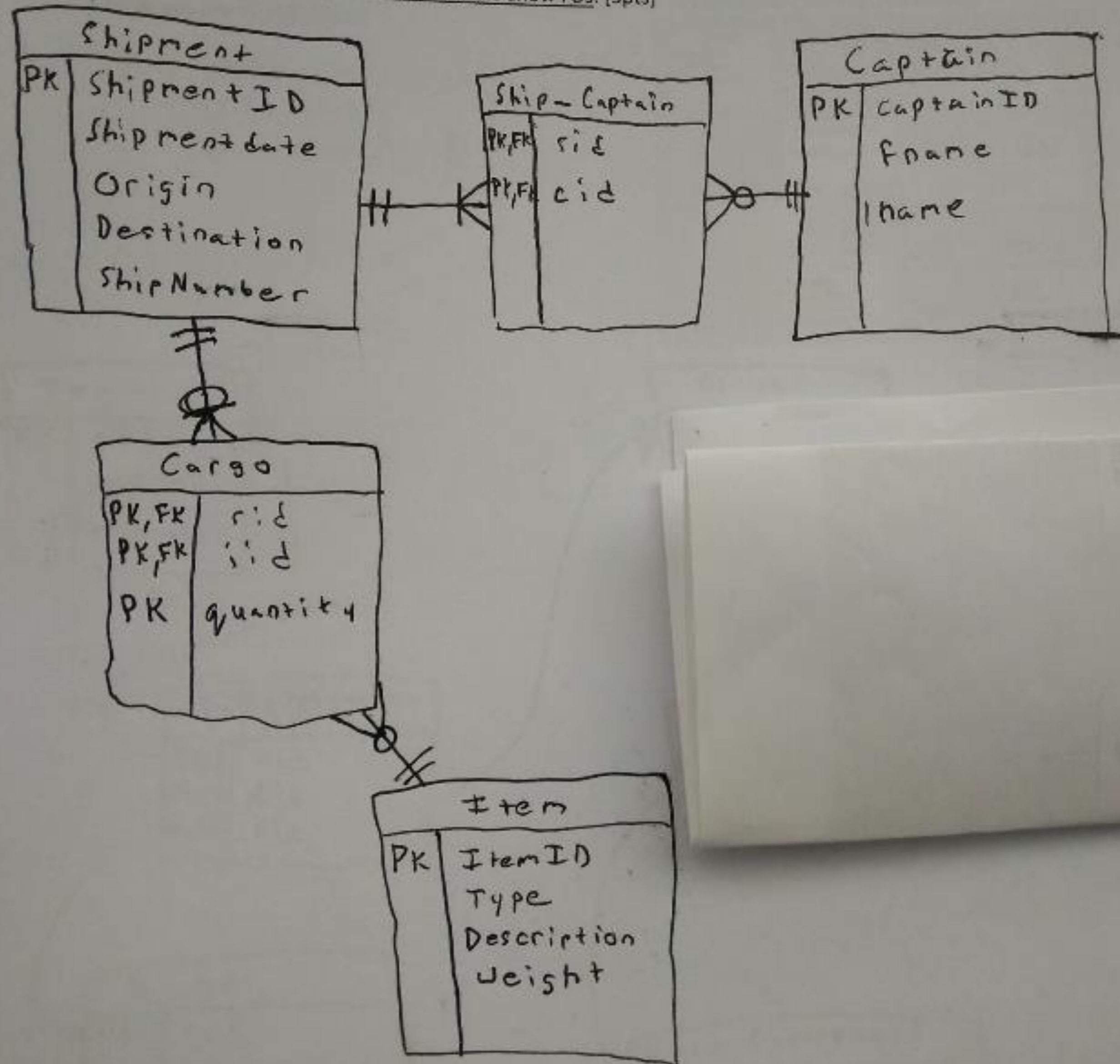
ITEM and COMPONENT relations



Create Two relations:

- One for the entity type
- One for an associative relation in which the primary key has two attributes, both taken from the primary key of the entity

Slide 250



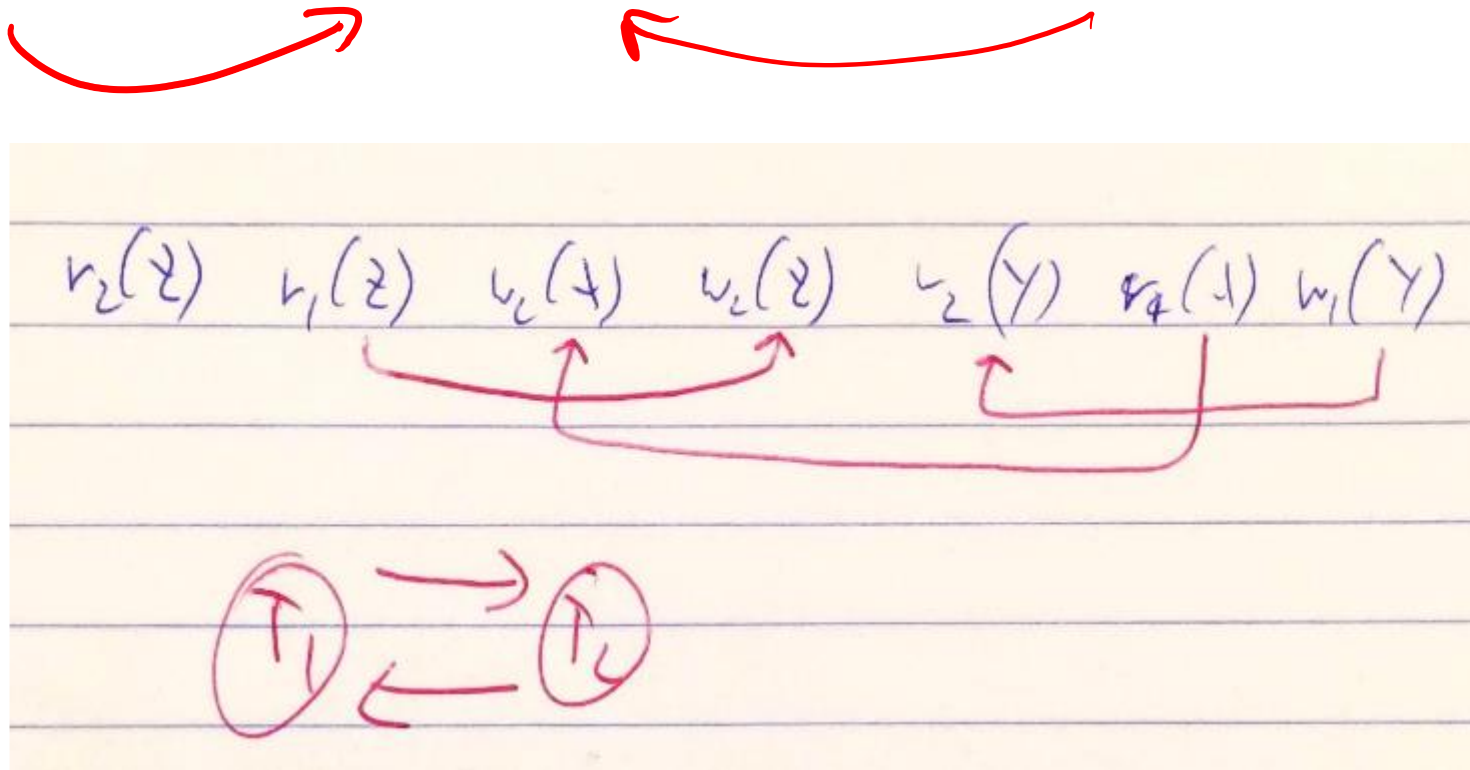
(b) Decompose your previous relation into a set of 3NF relations. Draw a relational schema for your 3NF relations and include the referential integrity constraints PKs and FKs. In contrast to question part (a), you do not need to show FDs. [3pts]
Shipment
PK
ShipmentID
Shipment date
Origin
Destination
ShipNumber
Ship_Captain
PK,FK
sid
PK,FK
cid
Captain
PK
CaptainID
fname
lname
Cargo
PK,FK
rid
PK,FK
iid
PK
quantity
Item
PK
ItemID
Type
Description
weight

Is the following schedule serializable? In order to answer the question, first draw the conflict graph.

$r_2(Z)$, $r_1(Z)$, $w_2(X)$, $w_2(Z)$, $w_2(Y)$, c_2, $r_1(X)$, $w_1(Y)$, c_1

Is the following schedule serializable? In order to answer the question, first draw the conflict graph.

$r_2(Z),\ r_1(Z),\ w_2(X),\ w_2(Z),\ w_2(Y),\ c_2,\ r_1(X),\ w_1(Y),\ c_1$



Back to indexing …

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is <u>covering</u> *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

High-level Categories of Index Types

- B-Trees (covered next)
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - We will look at a variant called B+ Trees

- Hash Tables
  - There are variants of this basic structure to deal with IO
  - Called linear or extendible hashing- IO aware!

The data structures we present here are "IO aware"

Real difference between structures: costs of ops *determines which index you pick and why*

Activity-41.ipynb

2. B+ Trees

What we will learn about next

- B+ Trees: Basics
- B+ Trees: Design & Cost
- Clustered Indexes

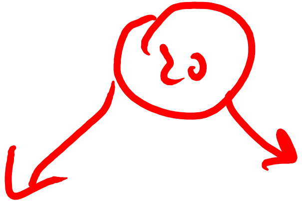
B+ Trees

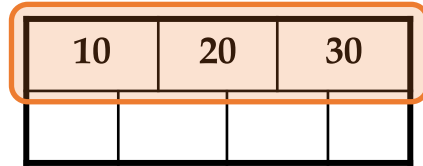
- Search trees
  - B does not mean binary!

- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)

- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree Basics



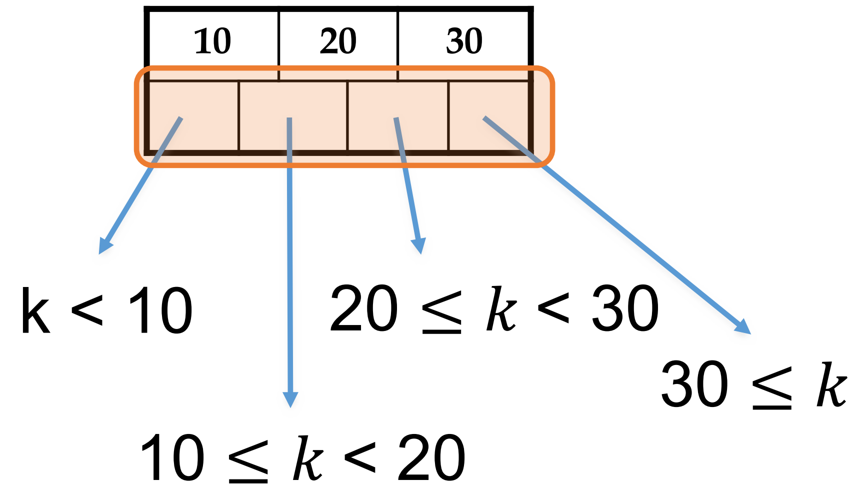


Parameter ***d*** = the (minimum) degree (=min number of keys)

Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****,

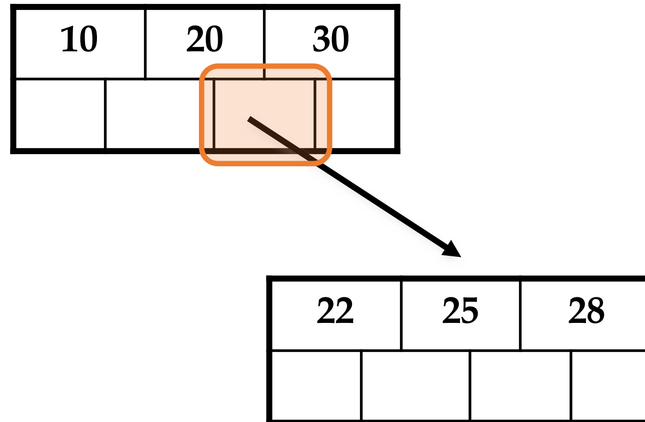
except for root node, which can have between **1 and 2d keys*

B+ Tree Basics



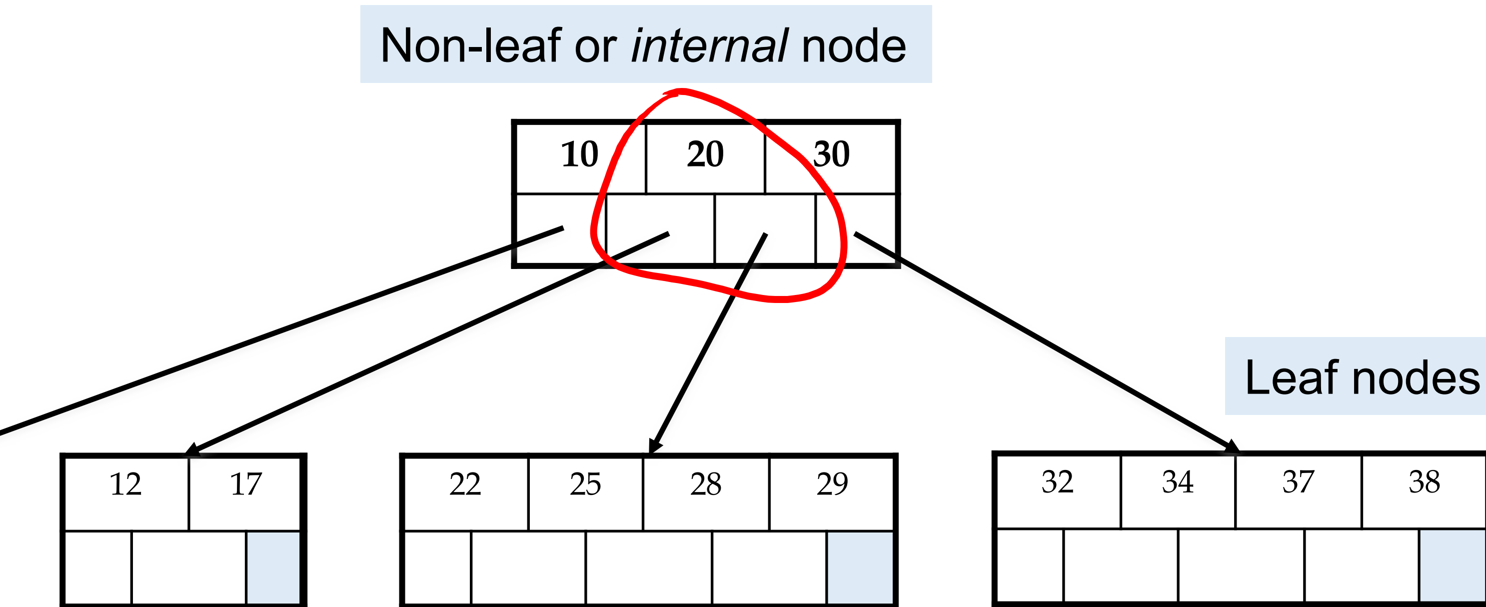
The *n* keys in a node define *n+1* ranges (branching factor)

B+ Tree Basics



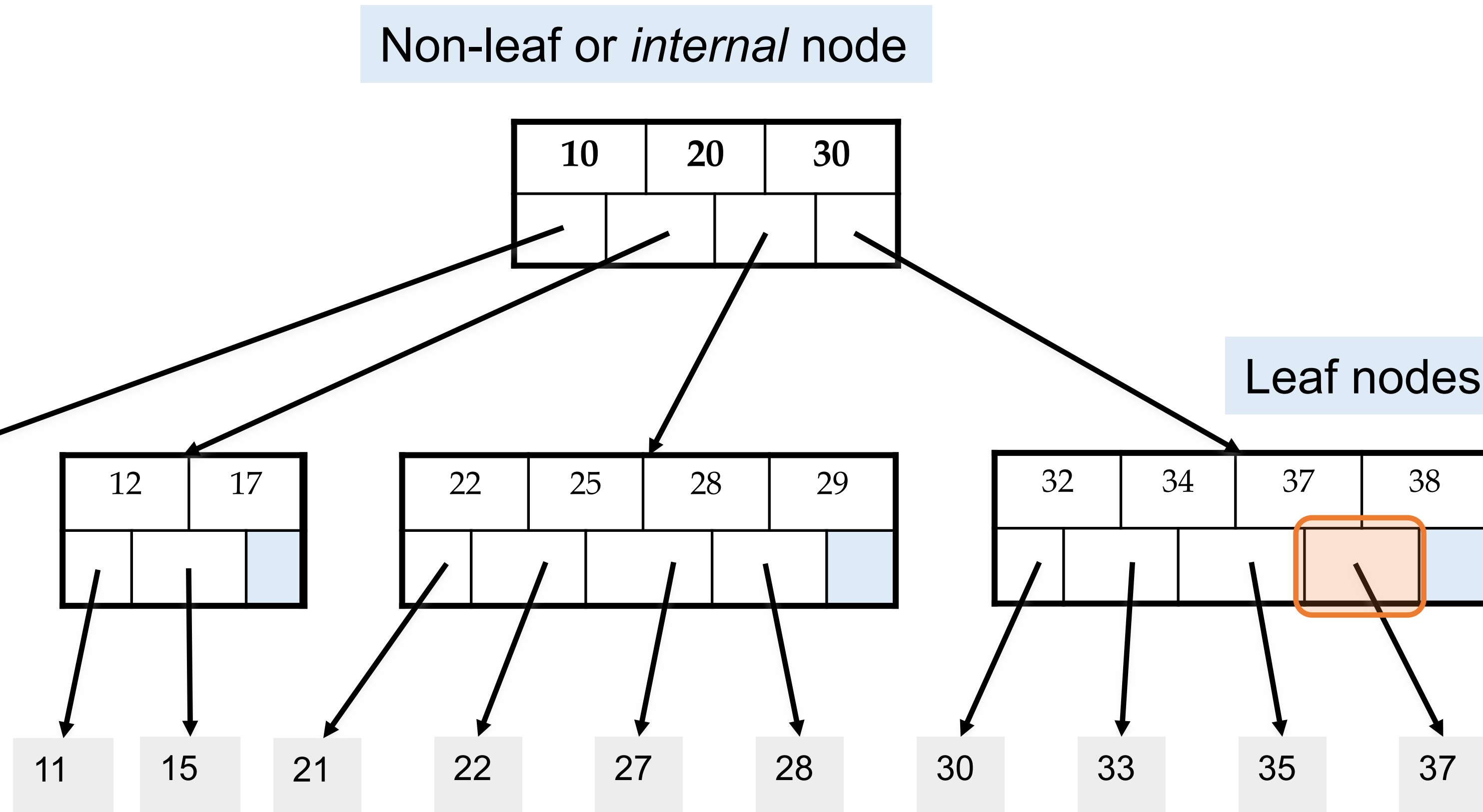
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

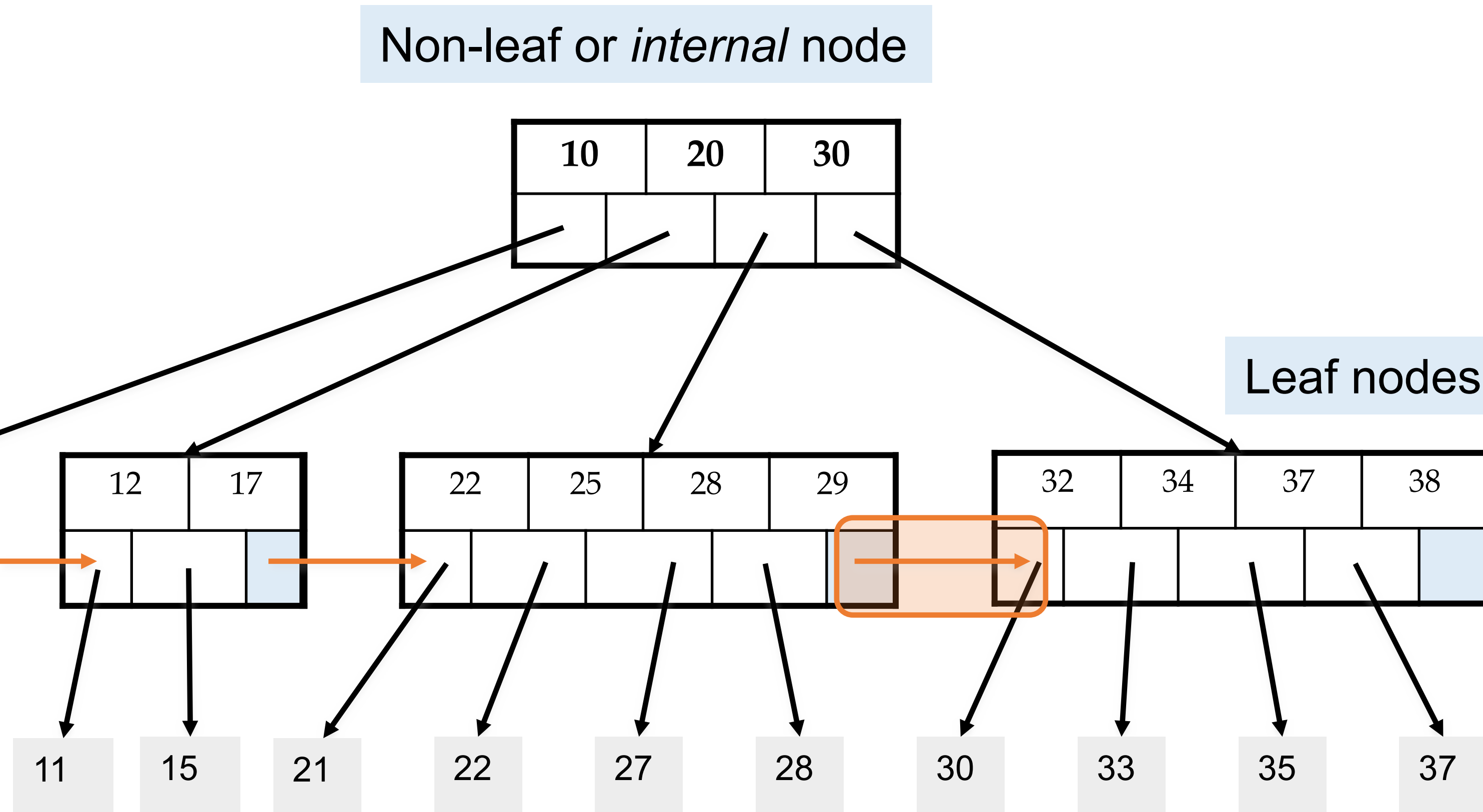
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

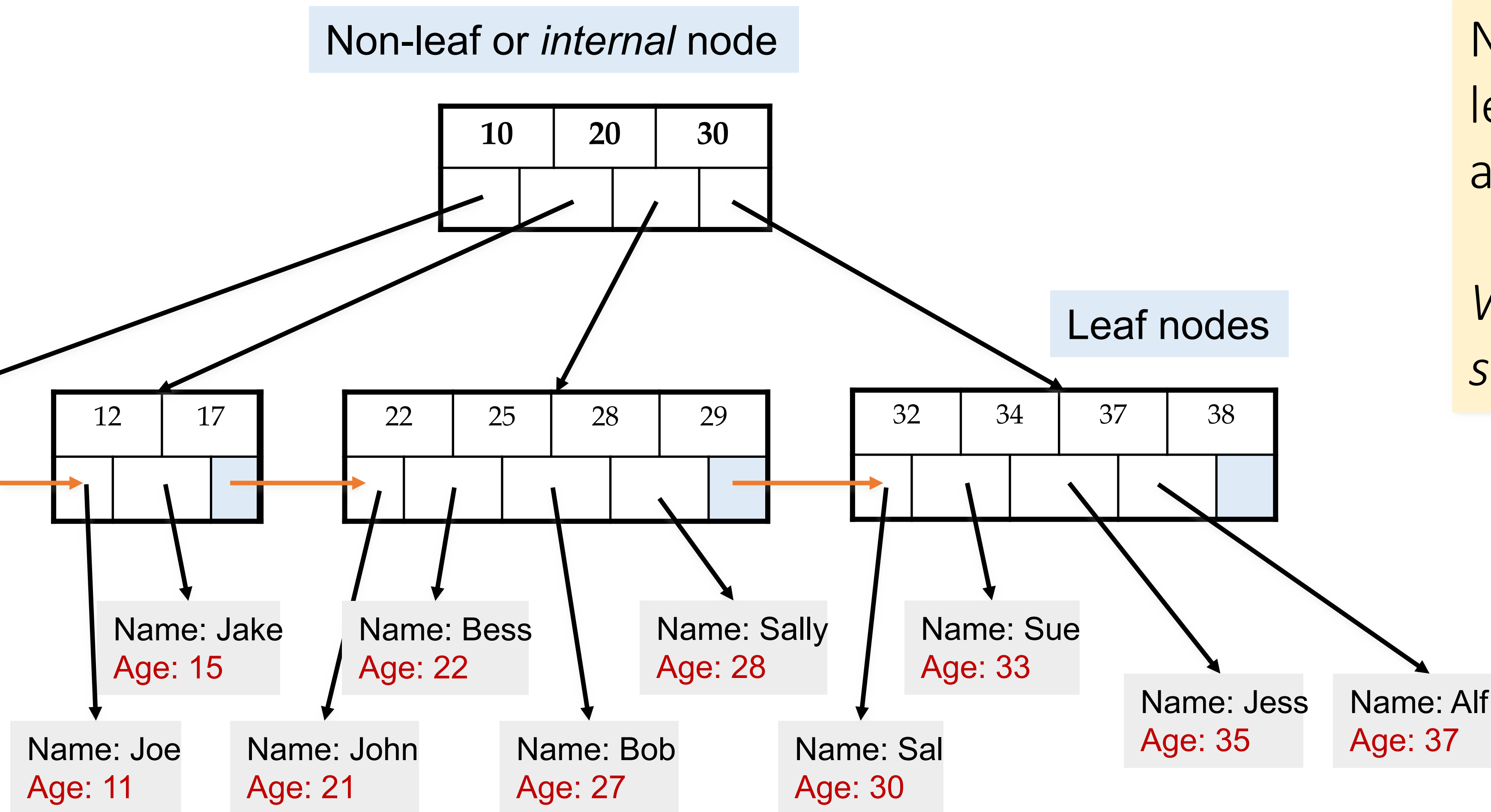


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf
- For range queries:
  - As above
  - Then sequential traversal

```sql
SELECT name
FROM   people
WHERE  age = 25
```

```sql
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

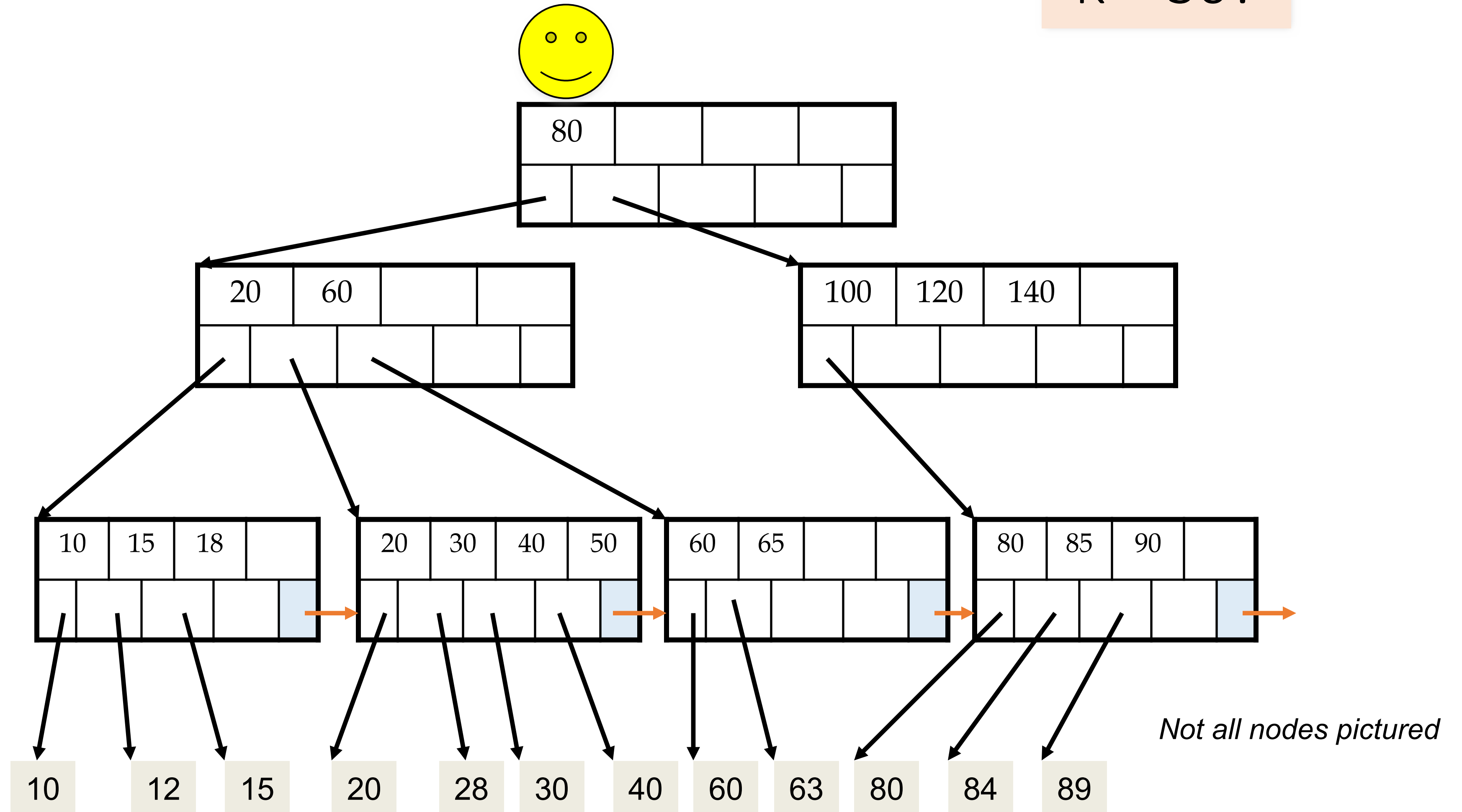
B+ Tree Exact Search Animation

30 < 80

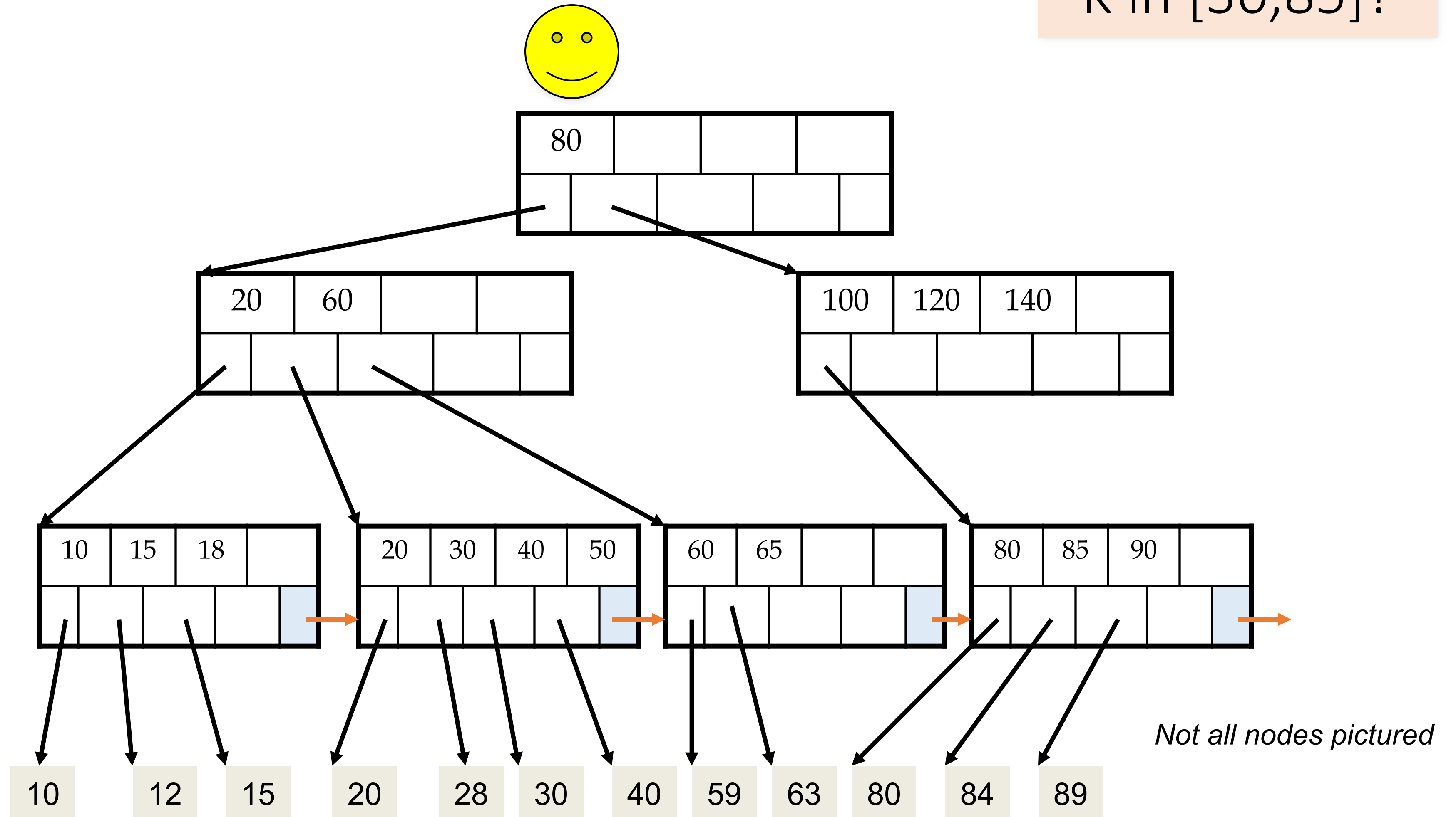
30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation



30 < 80

30 in [20,60)

30 in [30,40)

To the data!

B+ Tree Design

- How large is d?

  D = 2d

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes (=2^12)

- We want each node to fit on a single block/page
  - How large is d?

$4D + 8(D+1) \leq 4096$

B+ Tree Design

- How large is d?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes (= 2^{12})

NB: Oracle allows 64K = 2^{16} byte blocks
$\rightarrow d \leq 2730$

- We want each node to fit on a single block/page
  - $2d \times 4 + (2d+1) \times 8 \leq 4096 \rightarrow d \leq 170$

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have high fanout (between d+1 and 2d+1)

- This means that the depth of the tree is small → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!

- A TB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)
  - $(2*2730 + 1)^h = 2^{40} \rightarrow h = 4$

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133

- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records

- Top levels of tree sit in the buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

<u>Fill-factor</u> is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is in [d+1, 2d+1] (we'll assume it's constant for our cost model…)
  - ***N*** = the total number of pages we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (h) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have M available buffer pages, by the same logic:
  - We can store $\boldsymbol{L_M}$ levels of the B+ Tree in memory
  - where $\boldsymbol{L_M}$ is the number of levels such that the sum of all the levels' nodes fit in the buffer:
    - $M \geq 1 + f + \cdots + f^{L_M - 1} = \sum_{l=0}^{L_M - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_M + 1$

where $M \geq \sum_{l=0}^{L_M - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers

- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as “Cost(OUT)”

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_M + Cost(OUT)$

where $M \geq \sum_{l=0}^{L_M - 1} f^l$

Fast Insertions & Self-Balancing

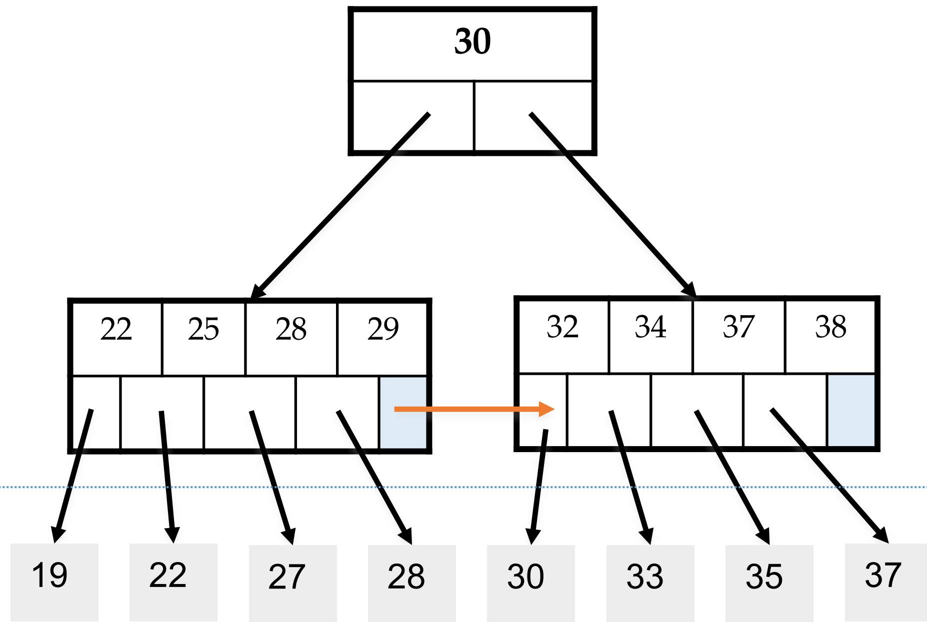
- The B+ Tree insertion algorithm has several attractive qualities (though we won't have time to go into details, this time):
  - ~ Same cost as exact search
  - Self-balancing: B+ Tree remains balanced (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

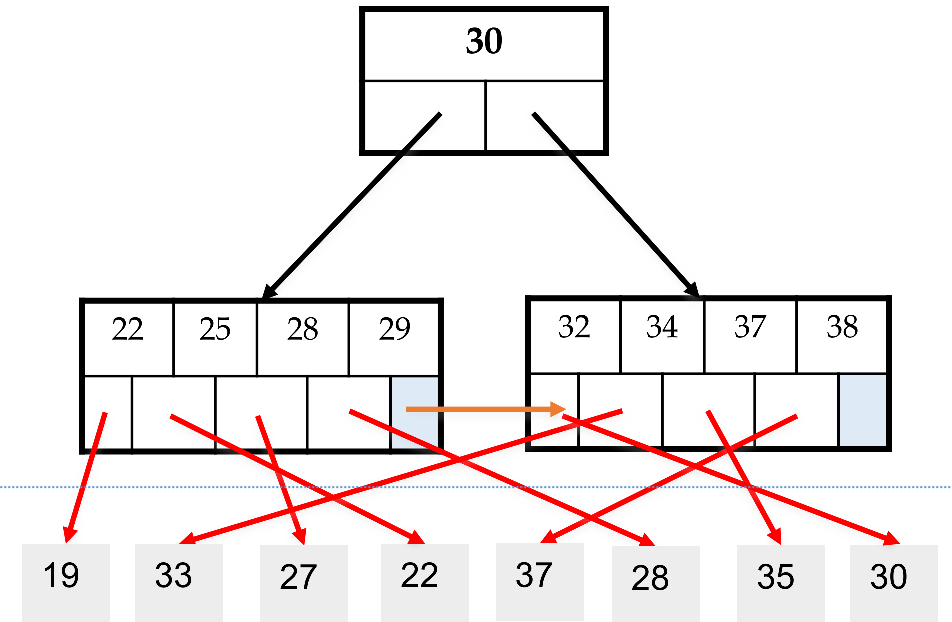
Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Index Entries

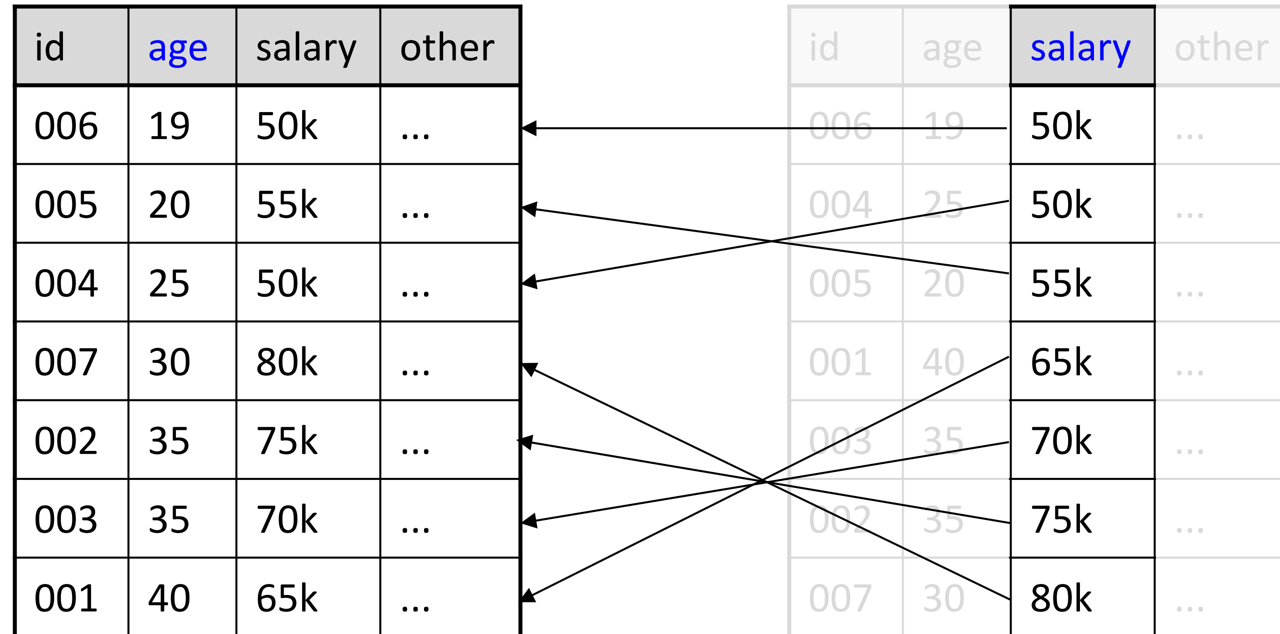


Data Records

Clustered

Unclustered

Recap: High-level overview: indexes



id	age	salary	other
006	19	50k	…
005	20	55k	…
004	25	50k	…
007	30	80k	…
002	35	75k	…
003	35	70k	…
001	40	65k	…

id	age	salary	other
006	19	50k	…
004	25	50k	…
005	20	55k	…
001	40	65k	…
003	35	70k	…
002	35	75k	…
007	30	80k	…

data file = index file
clustered (primary) index

index file
unclustered (secondary) index

Clustered vs. Unclustered Index

- Recall that for a disk with block access, sequential IO is much faster than random IO

- For exact search, no difference between clustered / unclustered

- For range search over R values: difference between 1 random IO + R sequential IO, and R random IO:
  - A random IO costs ~ 10ms on solid state drives (sequential much much faster)
  - For R = 100,000 records- difference between ~10ms and ~17min!

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An IO aware algorithm!

- We create indexes over tables in order to support fast (exact and range) search and insertion over multiple search keys

- B+ Trees are one index data structure which support very fast exact and range search & insertion via high fanout
  - Clustered vs. unclustered makes a big difference for range queries too