CSE544
Data Management

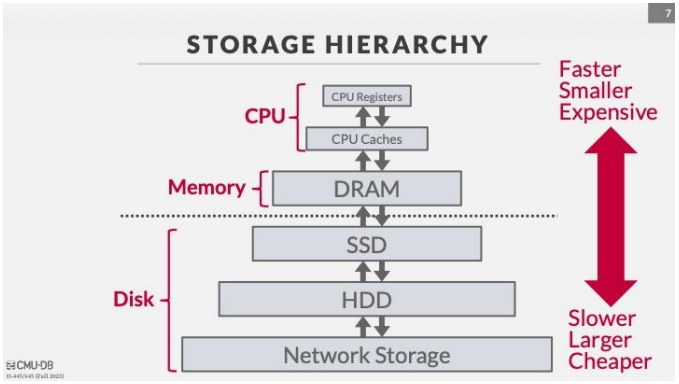
Lectures 8:
Page Format and Indexes

Announcements

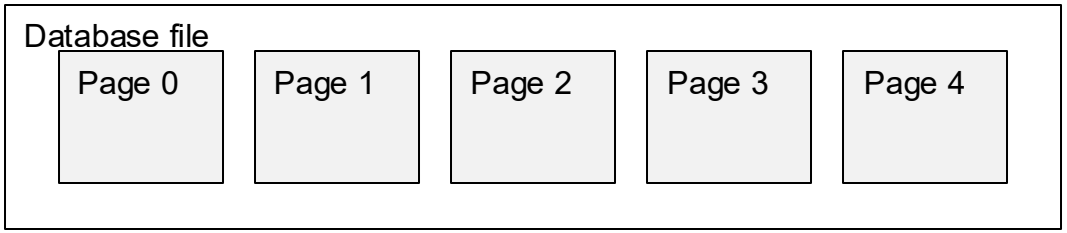
- HW1 is graded (thanks Moe!)
- HW2 due on Friday
- Project proposals due next Friday

Recap

- Storage hierarchy



- Disks
  - Unit of read/write = 1 block
  - Buffer pool: block/page/frame
- Files



Page Format

Page Format: Overview

- Row-oriented Format
  - A.k.a. N-ary storage model NSM
  - Each page contains several records
  - Records are laid out in the attribute order

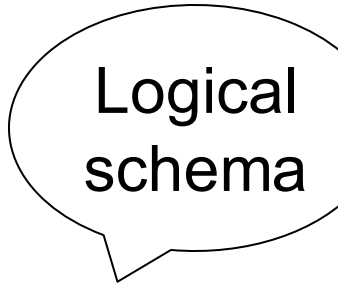
Page Format: Overview

- Row-oriented Format
  - A.k.a. N-ary storage model NSM
  - Each page contains several records
  - Records are laid out in the attribute order
- PAX
  - Each page contains several records, but grouped by their attribute

Page Format: Overview

- Row-oriented Format
  - A.k.a. N-ary storage model NSM
  - Each page contains several records
  - Records are laid out in the attribute order
- PAX
  - Each page contains several records, but grouped by their attribute
- Column-oriented Format
  - A.k.a. C-store
  - Each page contains values from only one attribute

Row-Oriented Storage



Product

Name	Price	Color
iPhone	599	gray
Jacket	129	blue
Pants	89	black
…		
…		
Bicycle	599	Red
…		

Row-Oriented Storage

Physical layout

Logical schema

Page 0

iPhone	599	gray
Jacket	129	blue
Pants	89	black
…		

Page 1

Bicycle	599	Red
…		
…		

…

Product

Name	Price	Color
iPhone	599	gray
Jacket	129	blue
Pants	89	black
…		
…		
Bicycle	599	Red
…		

Row-Oriented Storage

Physical layout

Logical schema

Page 0

iPhone	599	gray
Jacket	129	blue
Pants	89	black
…		

Page 1

Bicycle	599	Red
…		
…		

…

Product

Name	Price	Color
iPhone	599	gray
Jacket	129	blue
Pants	89	black
…		
…		
Bicycle	599	Red
…		

The schema stored separately, in the *database catalog*

Page Format

- One page contains several records
- Each record is uniquely identified by a Record ID (RID)
- How exactly do we store these records inside the page?

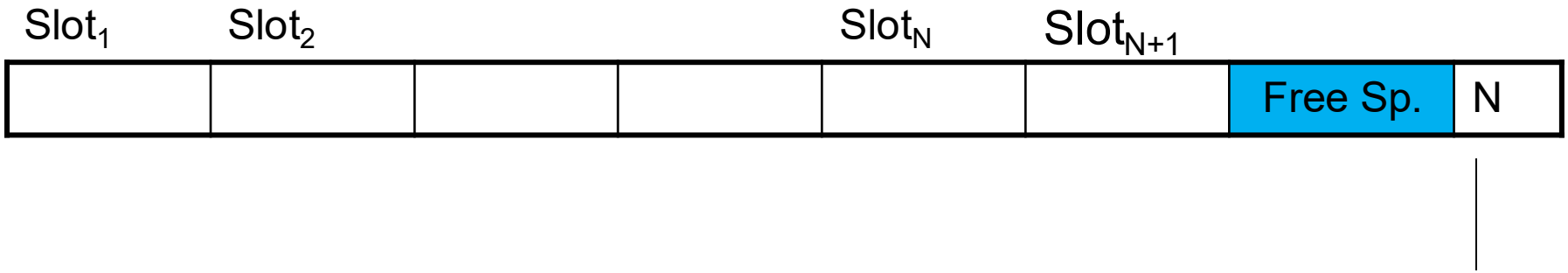
Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)

How do we insert a new record?

Page Format Approach 1

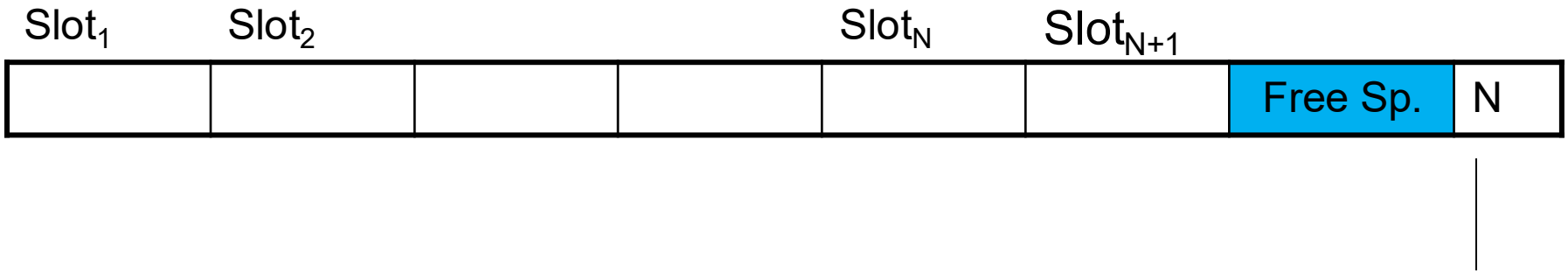
Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)



How do we insert a new record?

Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)

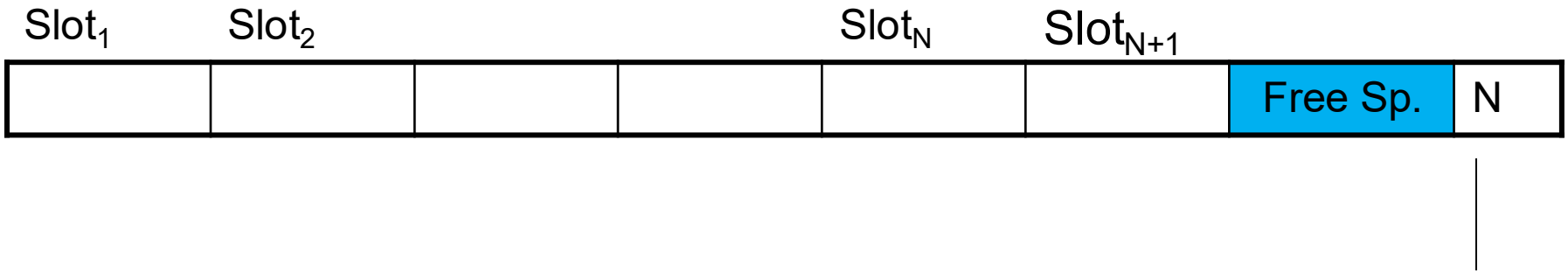


How do we insert a new record?

How do we delete a record?

Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)



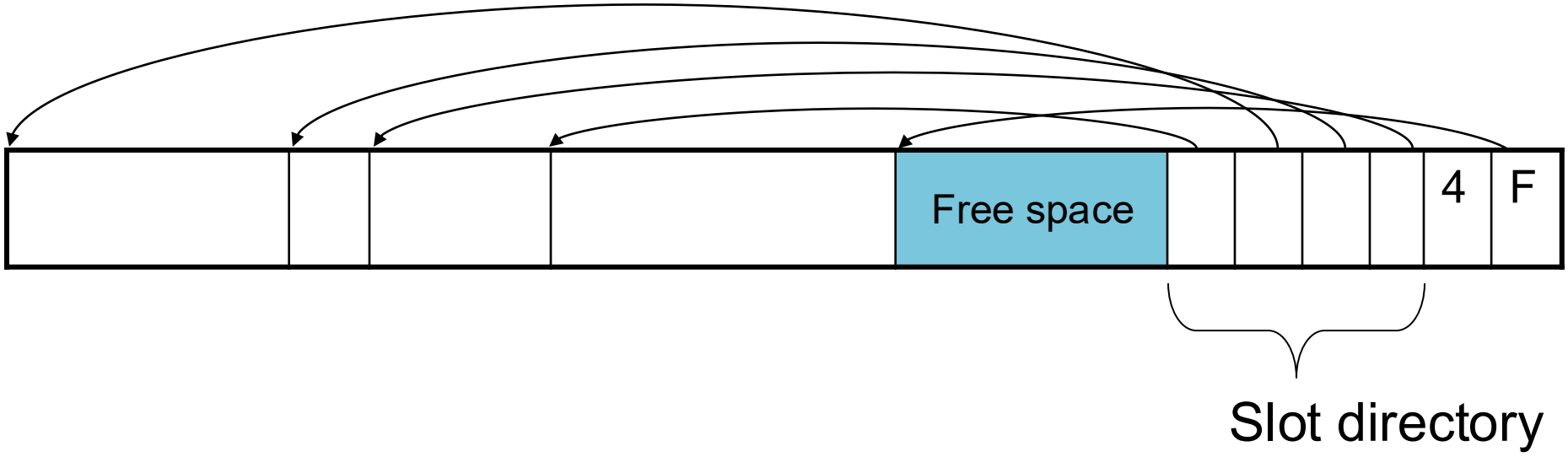
How do we insert a new record?

How do we delete a record? Cannot remove record (why?)

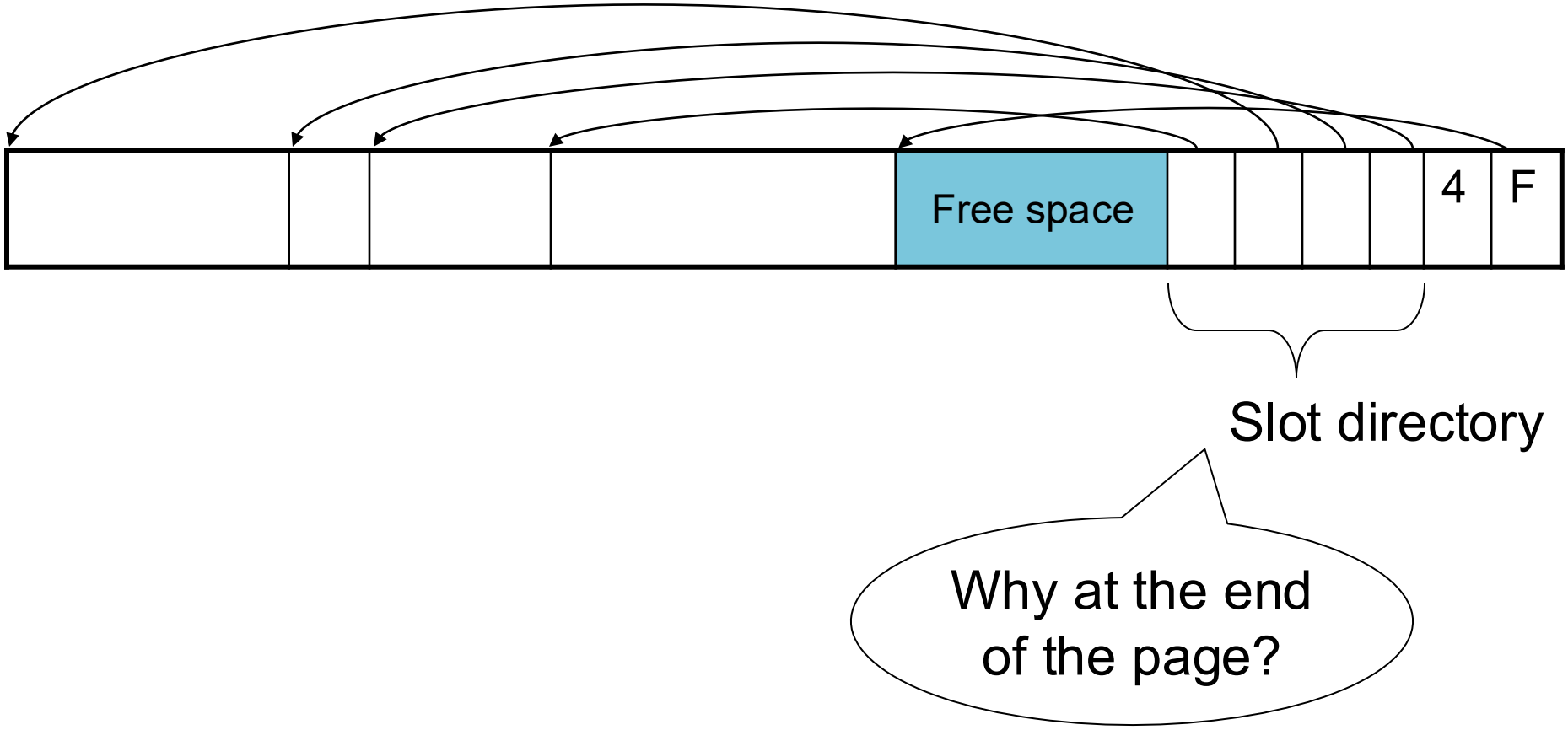
How do we handle variable-length records?

Page Format Approach 2

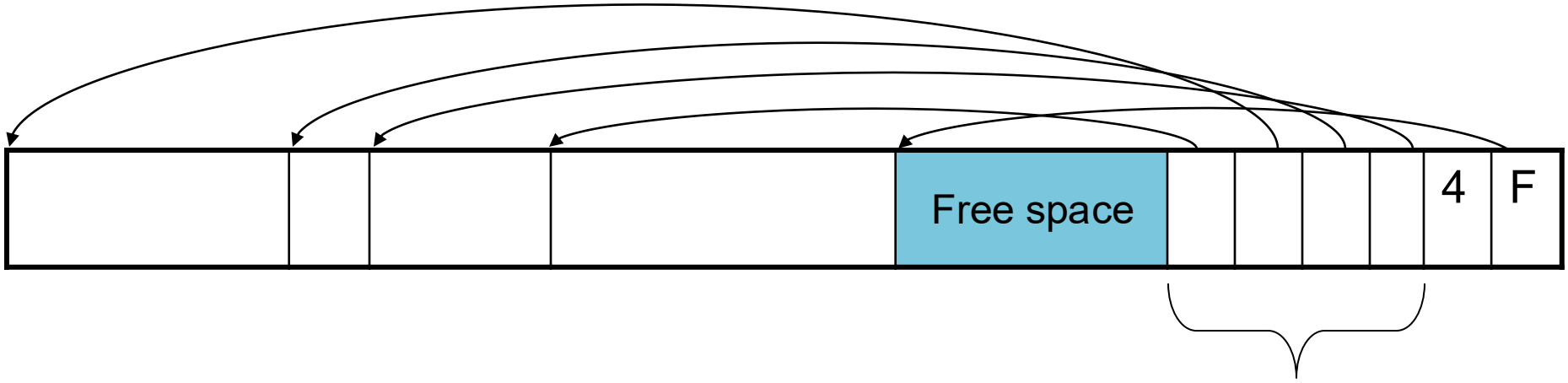
Page Format Approach 2



Page Format Approach 2



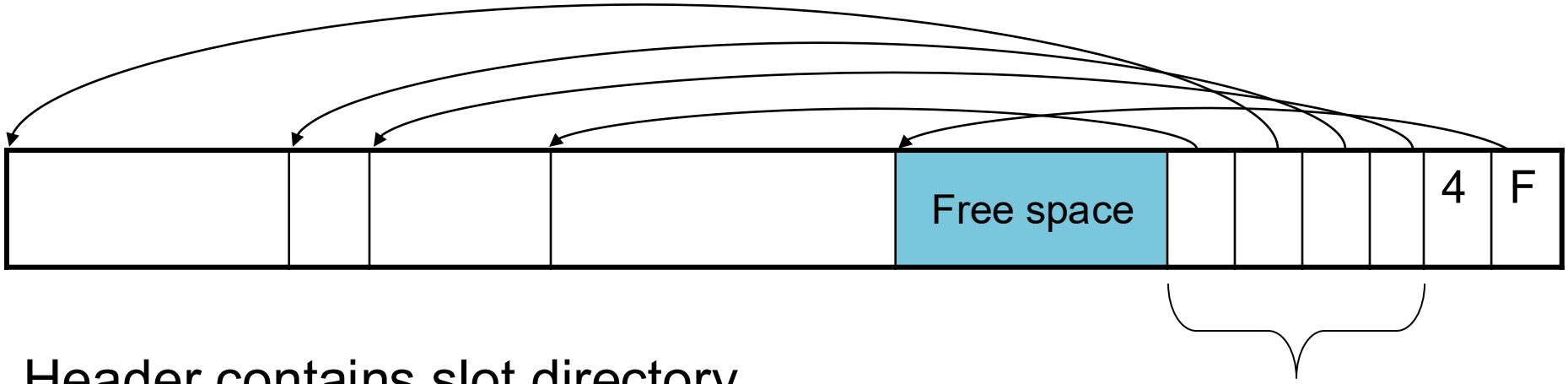
Page Format Approach 2



We can add a new slot when we insert a new record

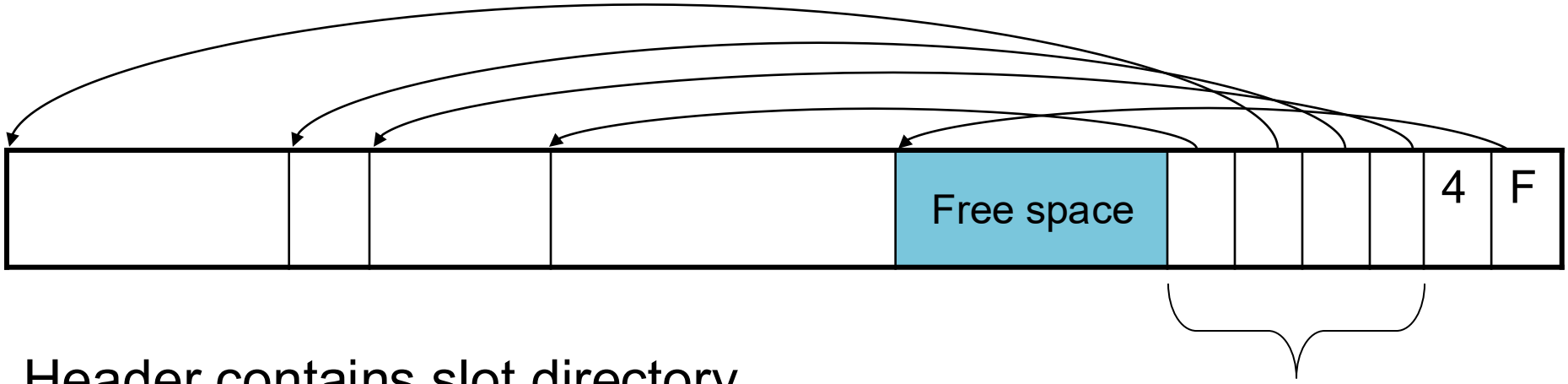
Why at the end of the page?

Page Format Approach 2



Header contains slot directory
+ Need to keep track of nb of slots
+ Also need to keep track of free space (F)

Page Format Approach 2



Header contains slot directory
+ Need to keep track of nb of slots
+ Also need to keep track of free space (F)

RID is (PageID, SlotID) combination
Variable-length records OK
Moving tuples inside page OK

Record Format

- One record contains several attributes
- How exactly do we store these attributes inside the record?

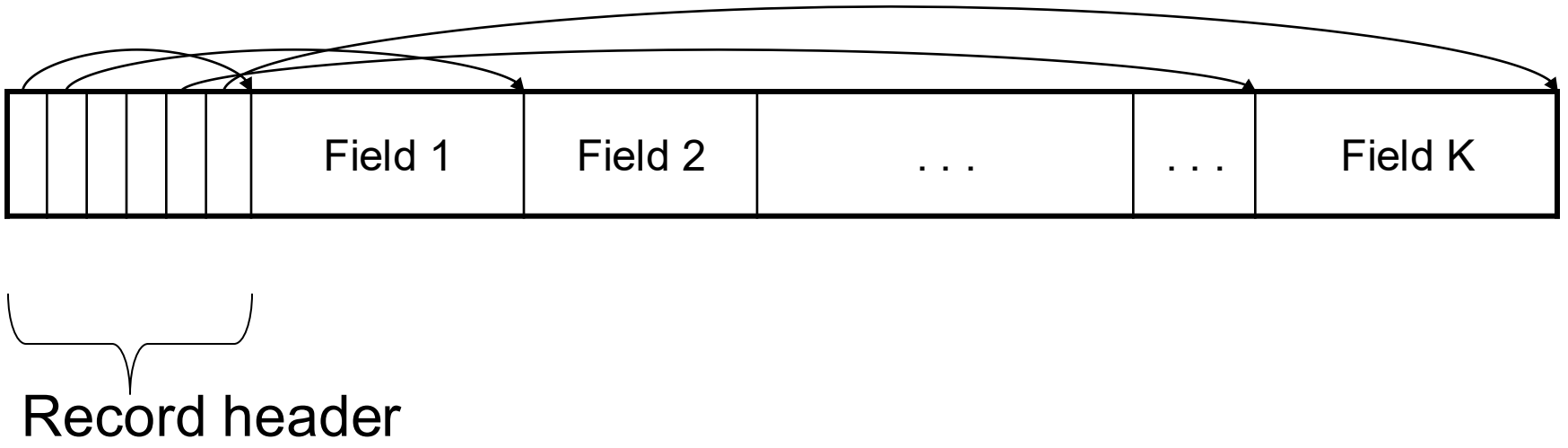
Record Formats

Fixed-length records => Each field has a fixed length (i.e., it has the same length in all the records)

Information about field lengths and types is in the catalog

Record Formats

Variable length records



Remark: NULLS require no space at all (why ?)

Row-Oriented: Summary

- One page: sequence of records
- One record: sequence of attributes

Column-oriented Storage

- Store each attribute in a different file

- Column 1: file0, file1, …
- Column 2: file10, file11, …

The Design and Implementation of Modern Column-Oriented Database Systems, Abadi et al., 2013

Column-oriented Storage

Row-based
(4 pages)

Page

A	1
A	2

A	2
A	2

B	2
B	4

C	4
C	4

Column-based
(4 pages)

A	1
A	2
A	2
A	2

B	2
B	4
C	4
C	4

Page

C-Store also avoids large tuple headers

From Row to Column Storage (Modern Designs)

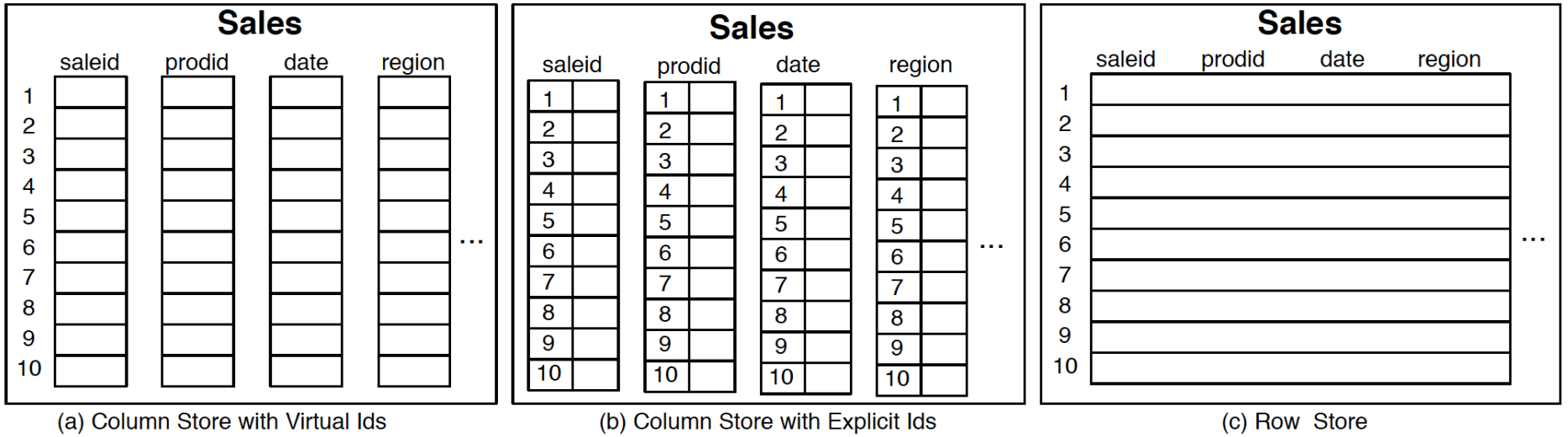


Figure 1.1: Physical layout of column-oriented vs row-oriented databases.

Basic tradeoffs:

- Reading all attributes of one records, v.s.
- Reading some attributes of many records

Column-oriented Storage

- Main idea:
  - Physical storage: complete vertical partition; each column stored separately: R.A, R.B, R.A
  - Logical schema: remains the same R(A,B,C)

- Main advantage:
  - Improved transfer rate: disk to memory, memory to CPU, better cache locality

Trade-Offs

- Row stores
  - Quick to update entire tuple (1 page IO)
  - Quick to access a single tuple
- Column stores
  - Avoid reading unnecessary columns
  - Better compression

Problem: needs an entire redesign of the DBMS

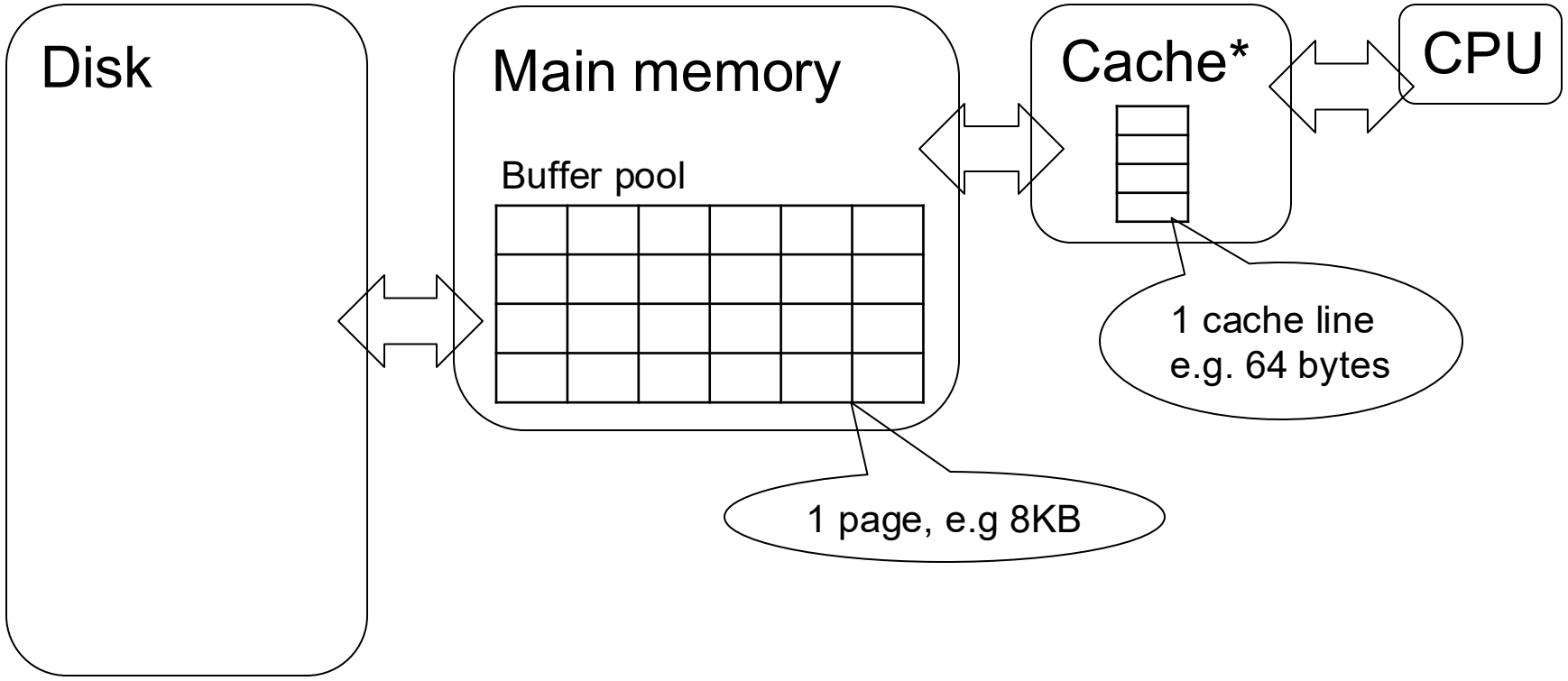
PAX

- As row-oriented: one page = set of records
- Inside the page: group data values by attribute, not by record
- This improves the L2-cache locality

Following slides are from Anastasia Ailamaki

Review: the Cache

Memory hierarchies:



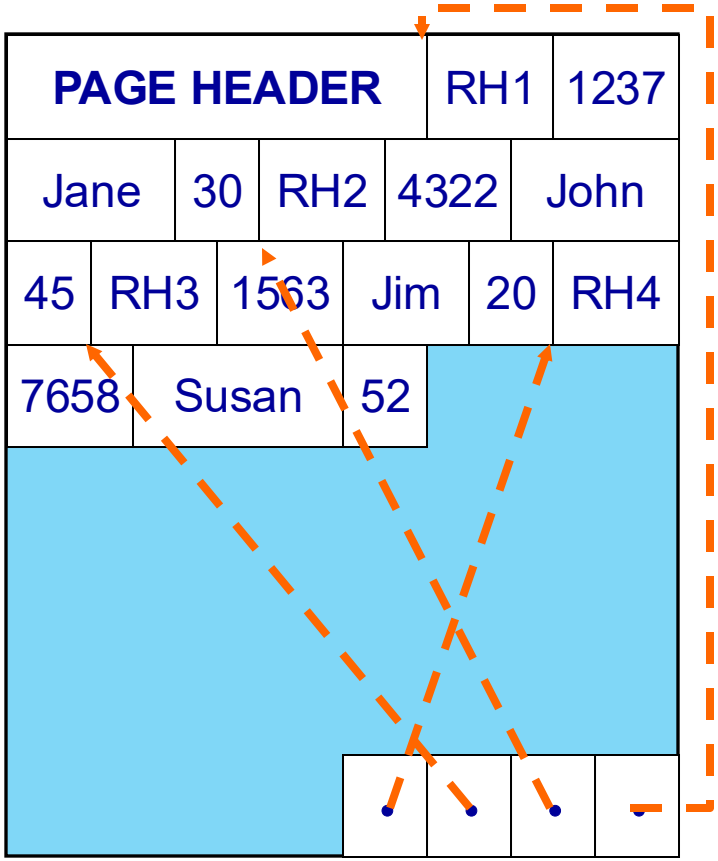
*aka CPU cache; several! L3, L2, L1 cache

Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

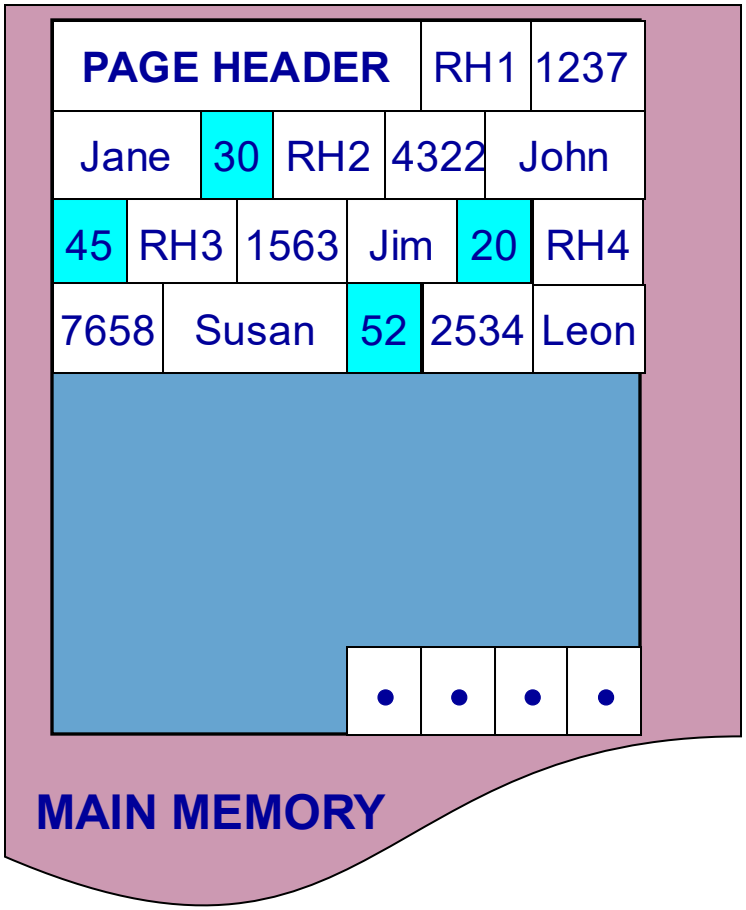
R

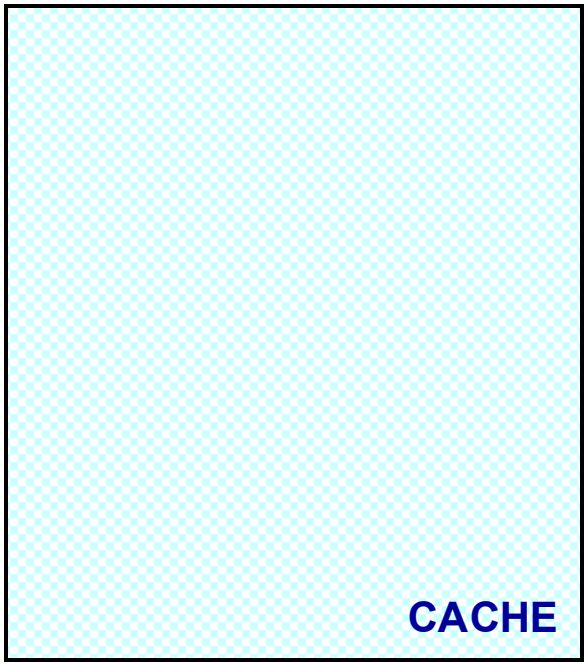
RID	SSN	Name	Age
1	1237	Jane	30
2	4322	John	45
3	1563	Jim	20
4	7658	Susan	52
5	2534	Leon	43
6	8791	Dan	37



- Records are stored sequentially
- Offsets to start of each record at end of page

Predicate Evaluation using NSM



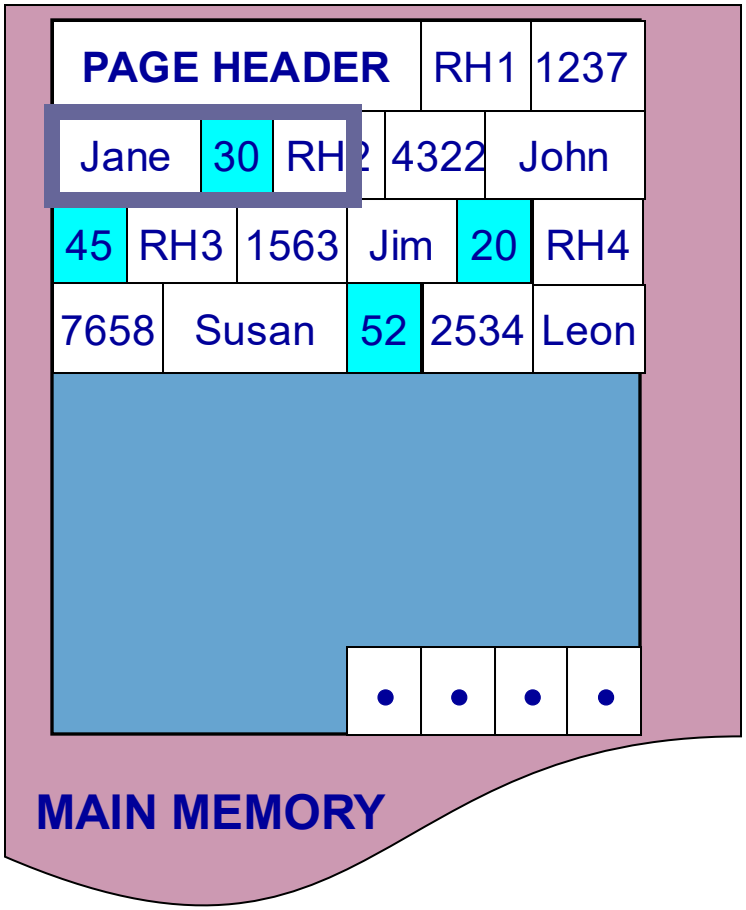


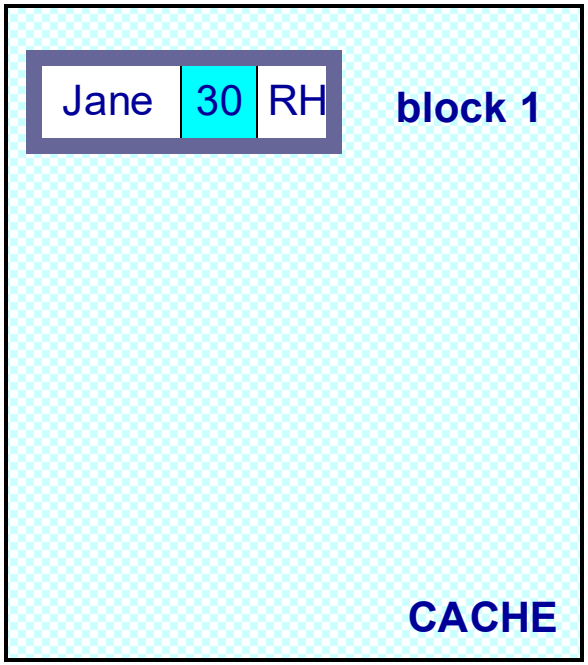

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using NSM



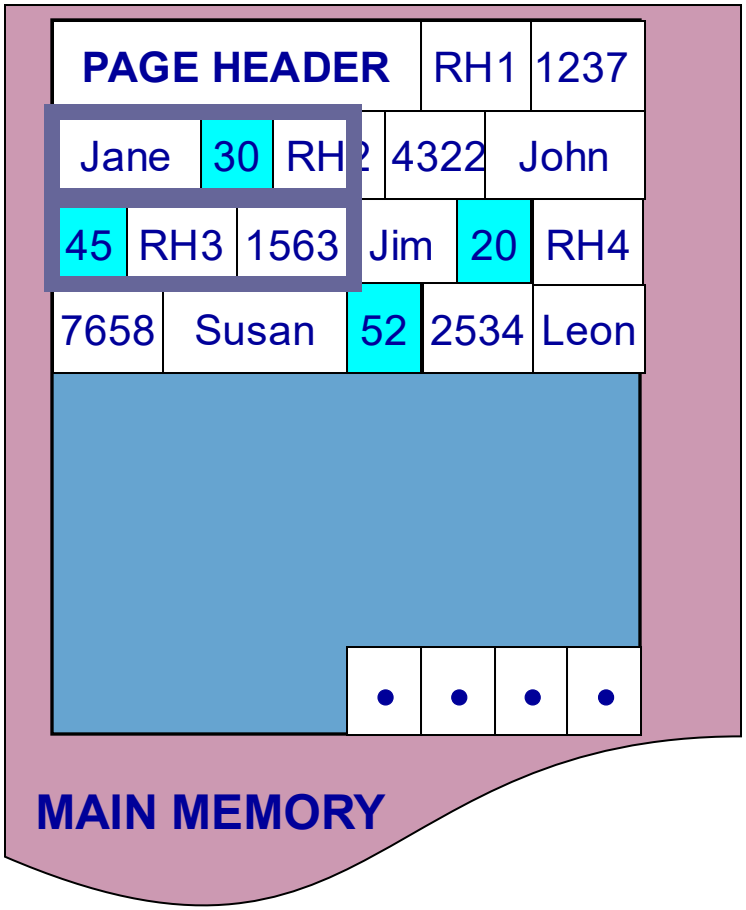


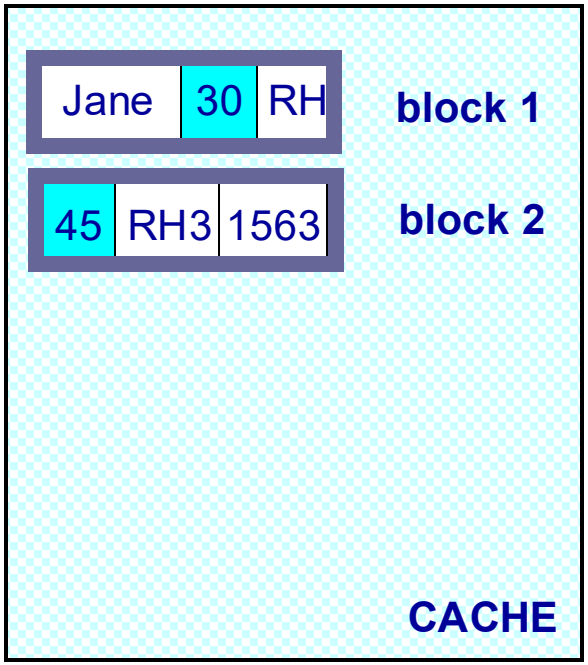

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using NSM



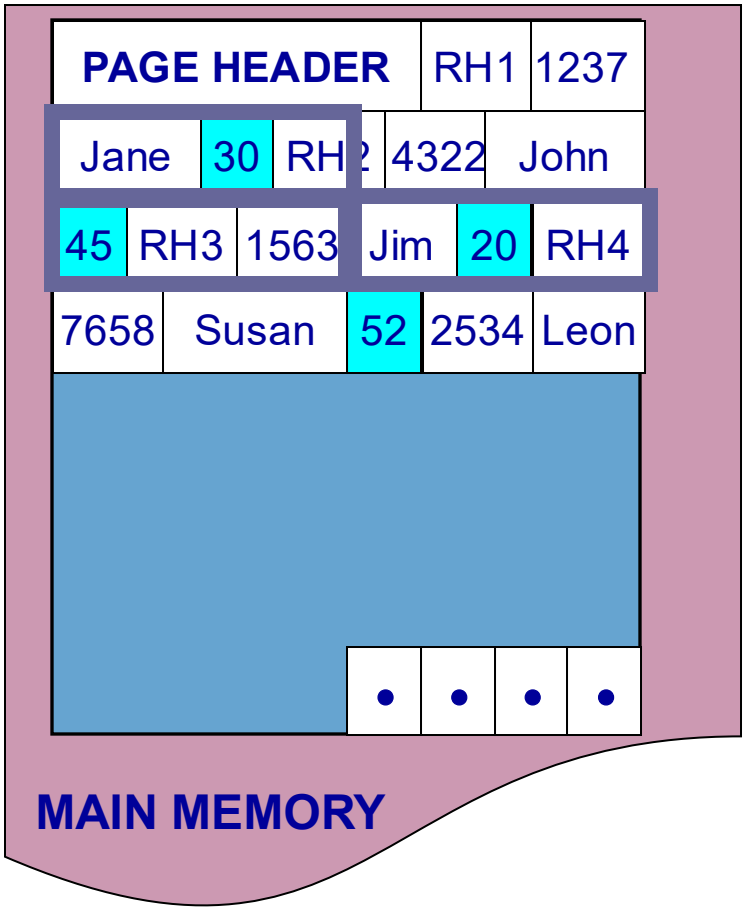


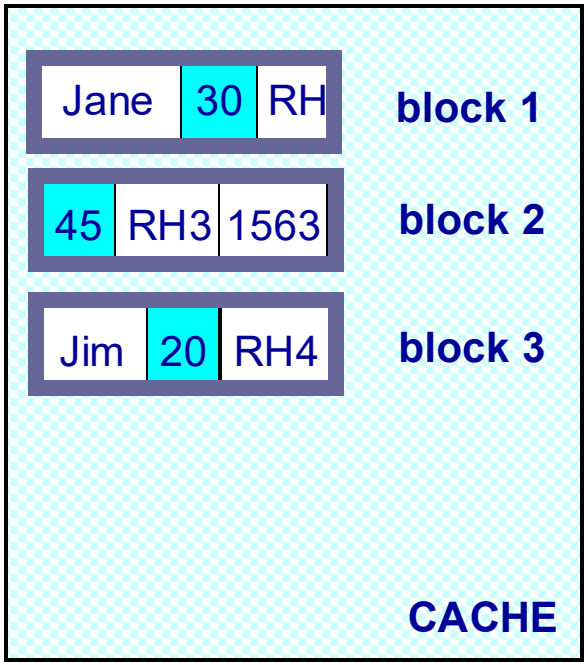

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using NSM



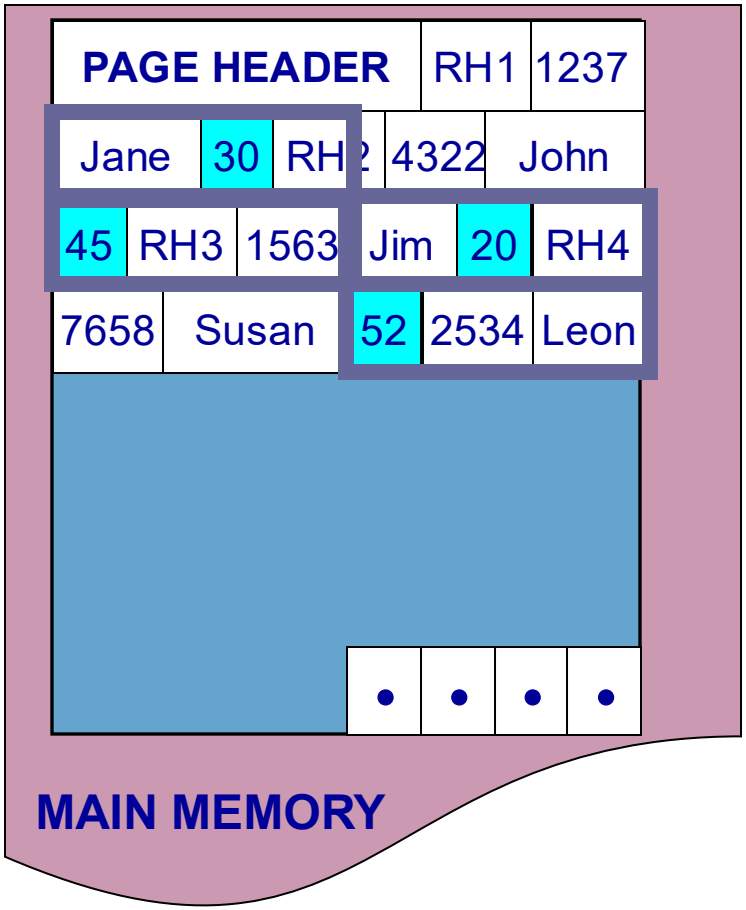


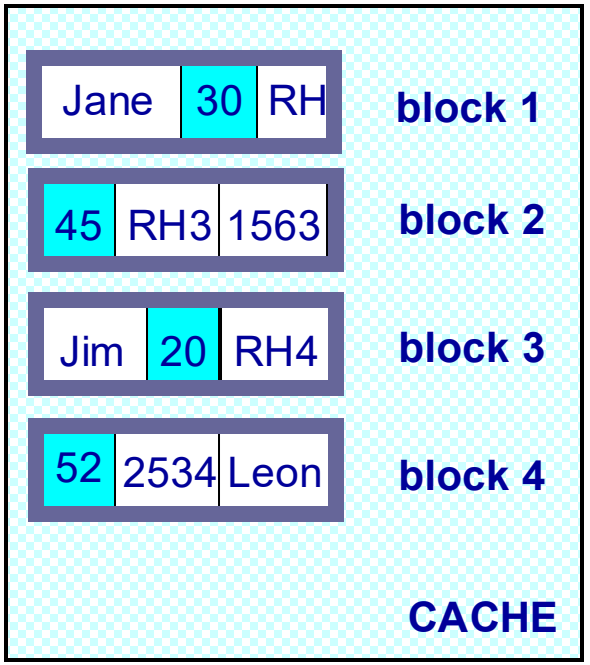

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using NSM






```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Need New Data Page Layout

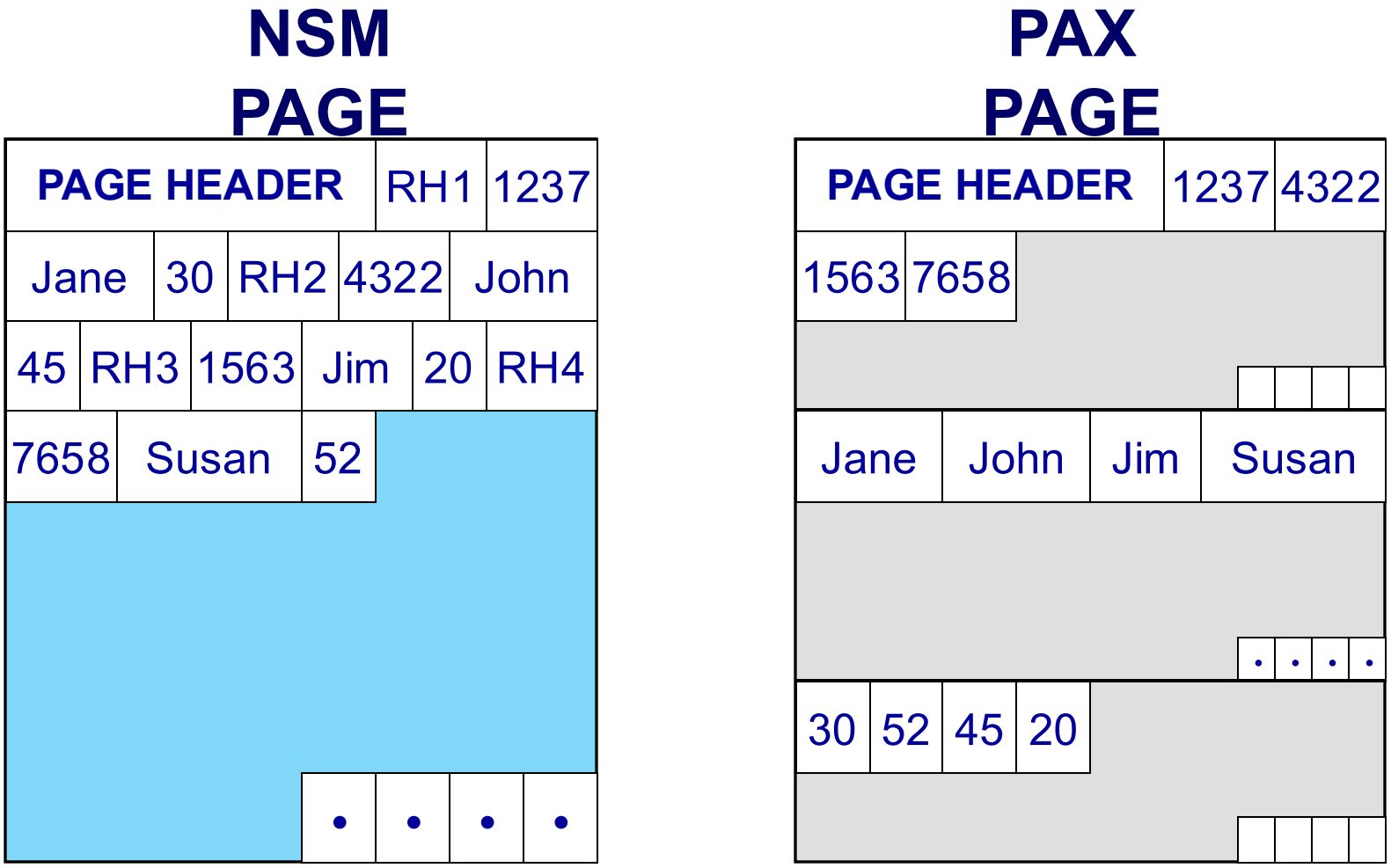
- Eliminates unnecessary memory accesses
- Improves inter-record locality
- Keeps a record's fields together
- Does not affect I/O performance

and, most importantly, is…

low-implementation-cost, high-impact

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

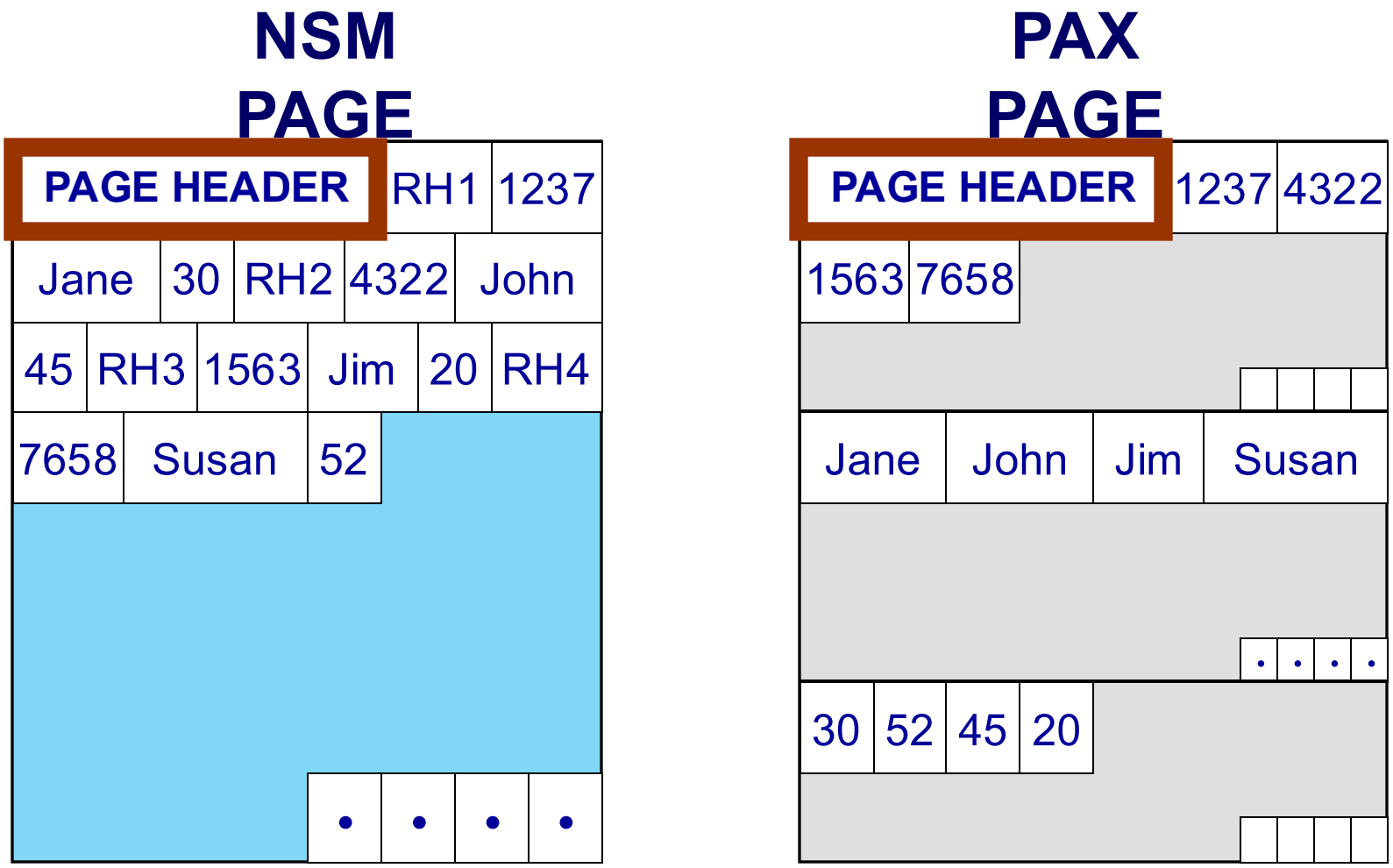
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

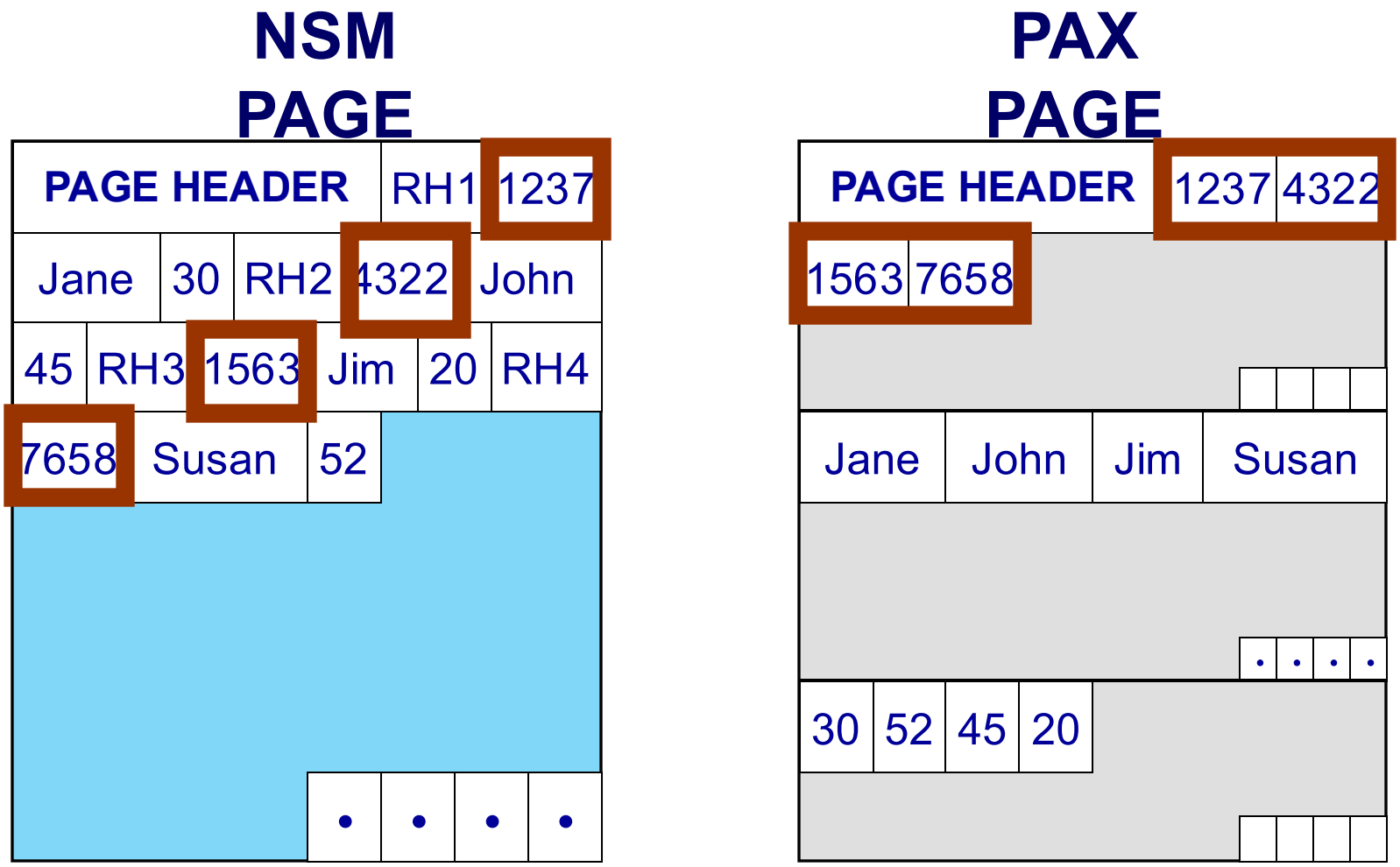
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

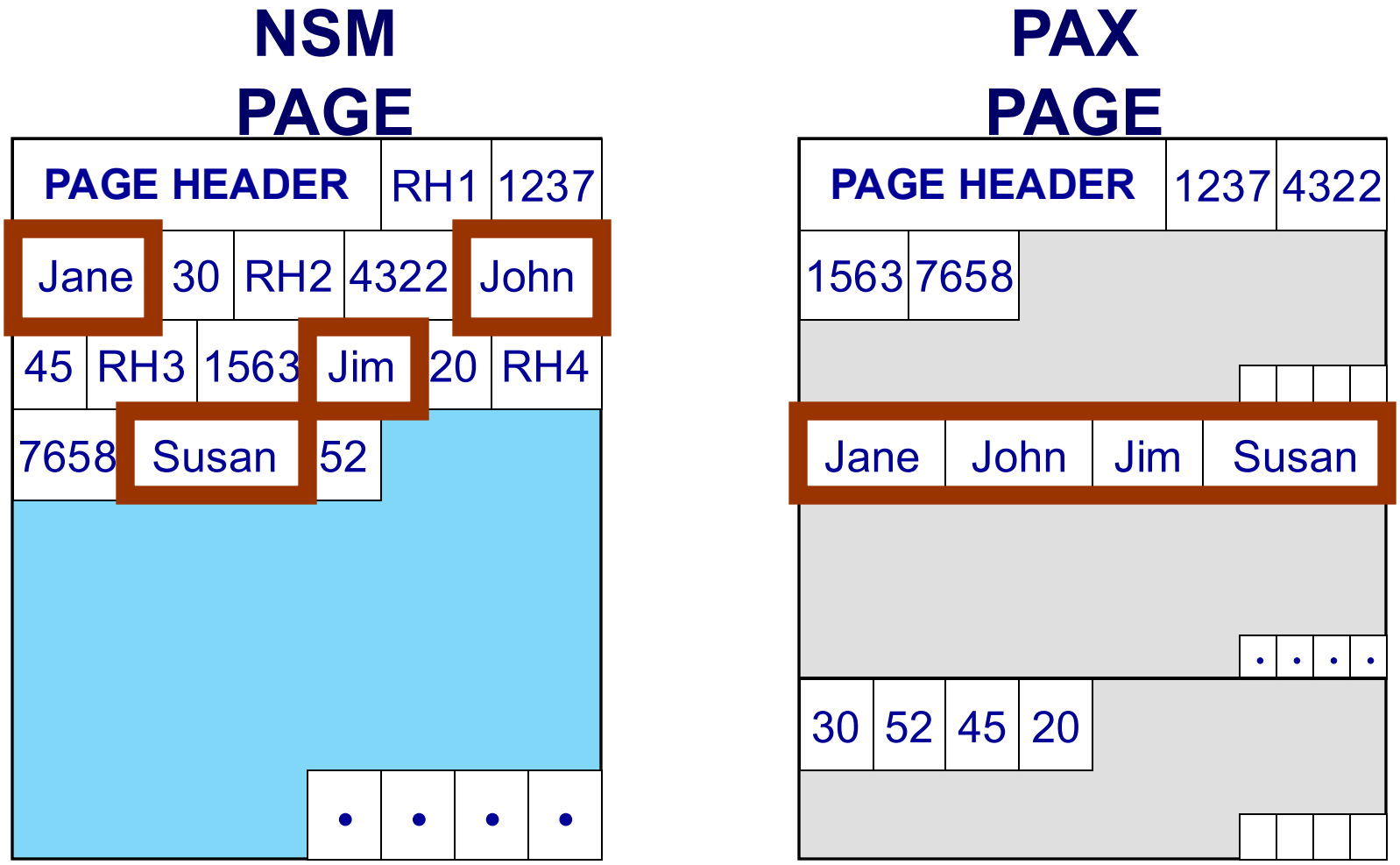
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

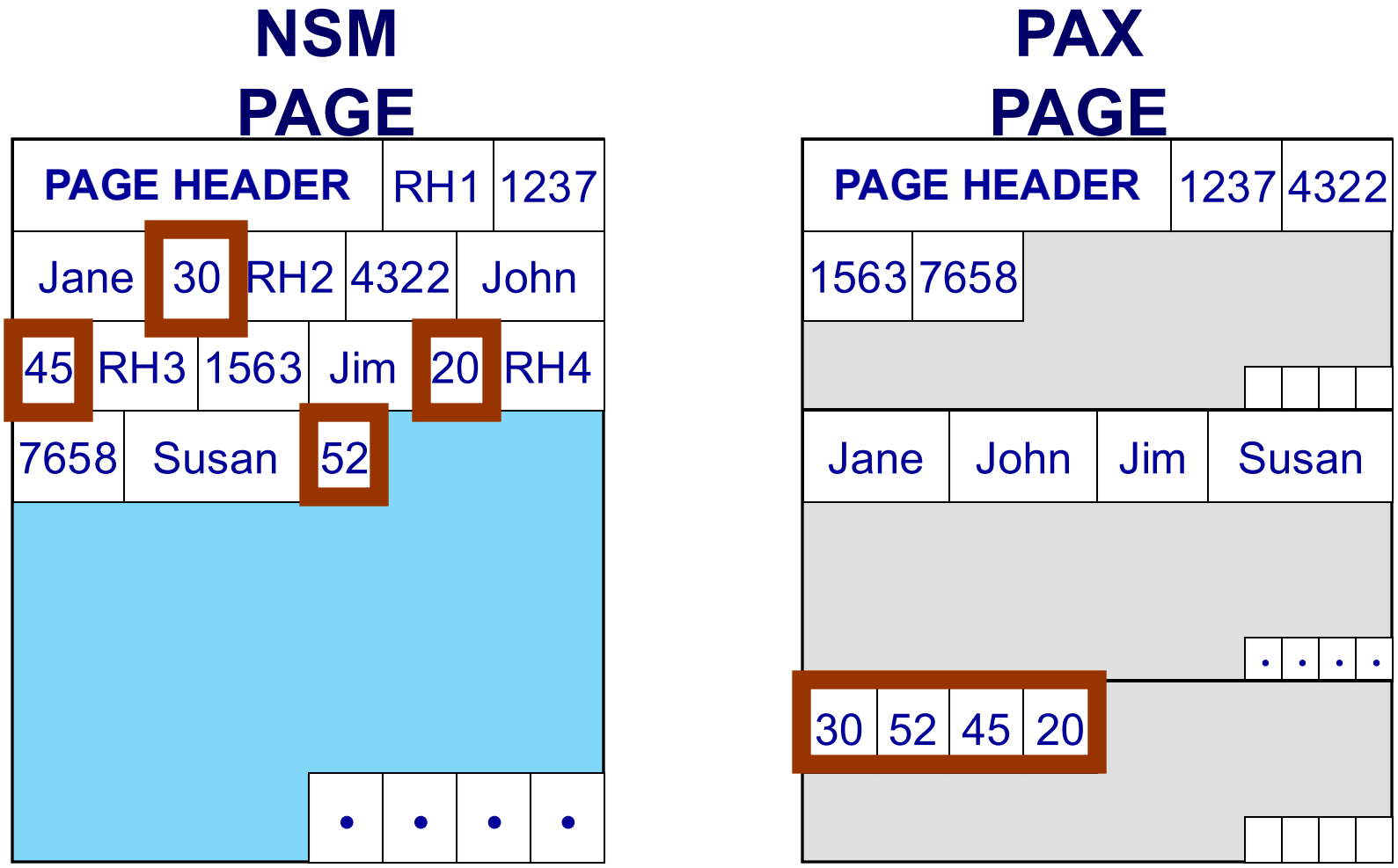
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

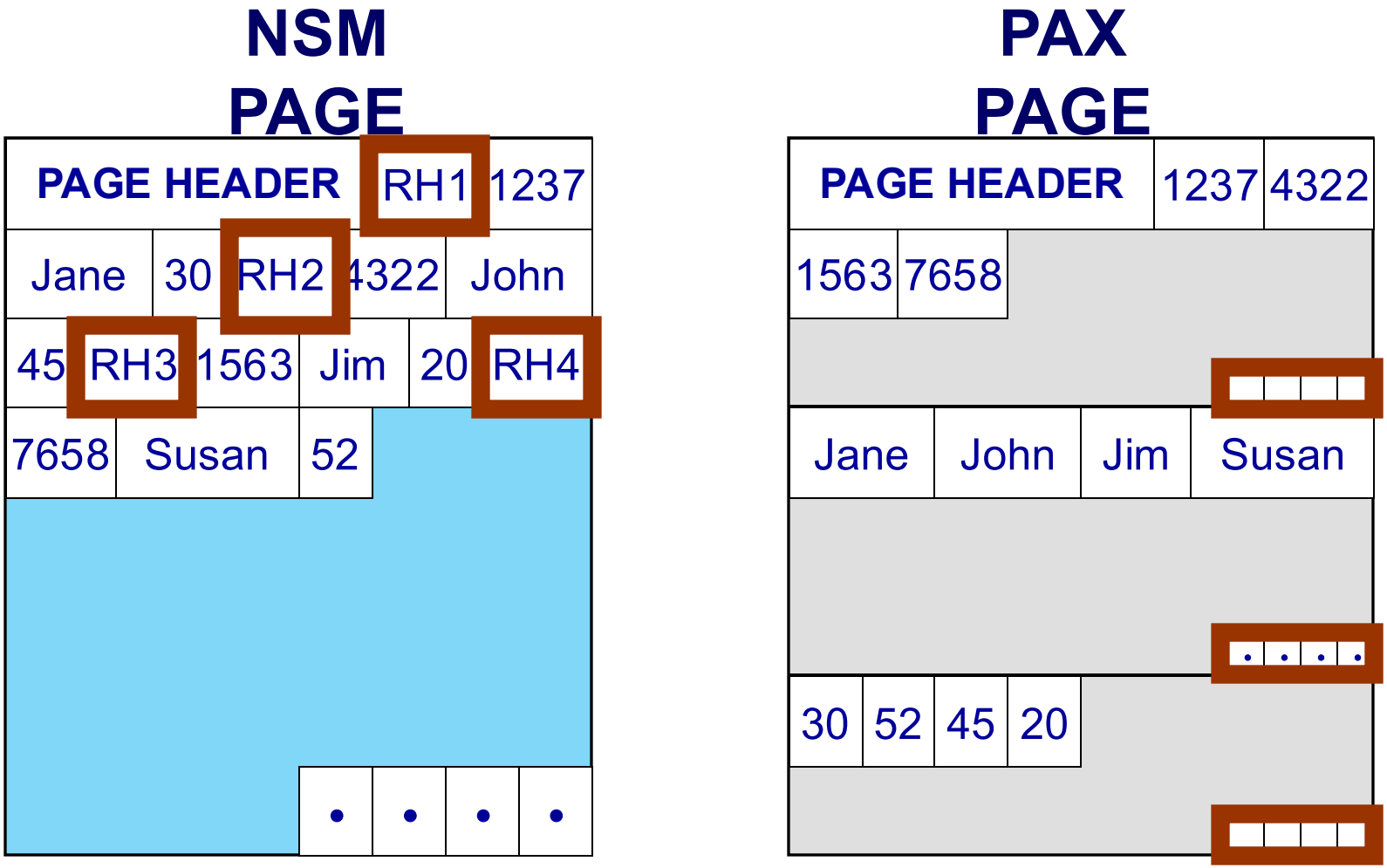
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

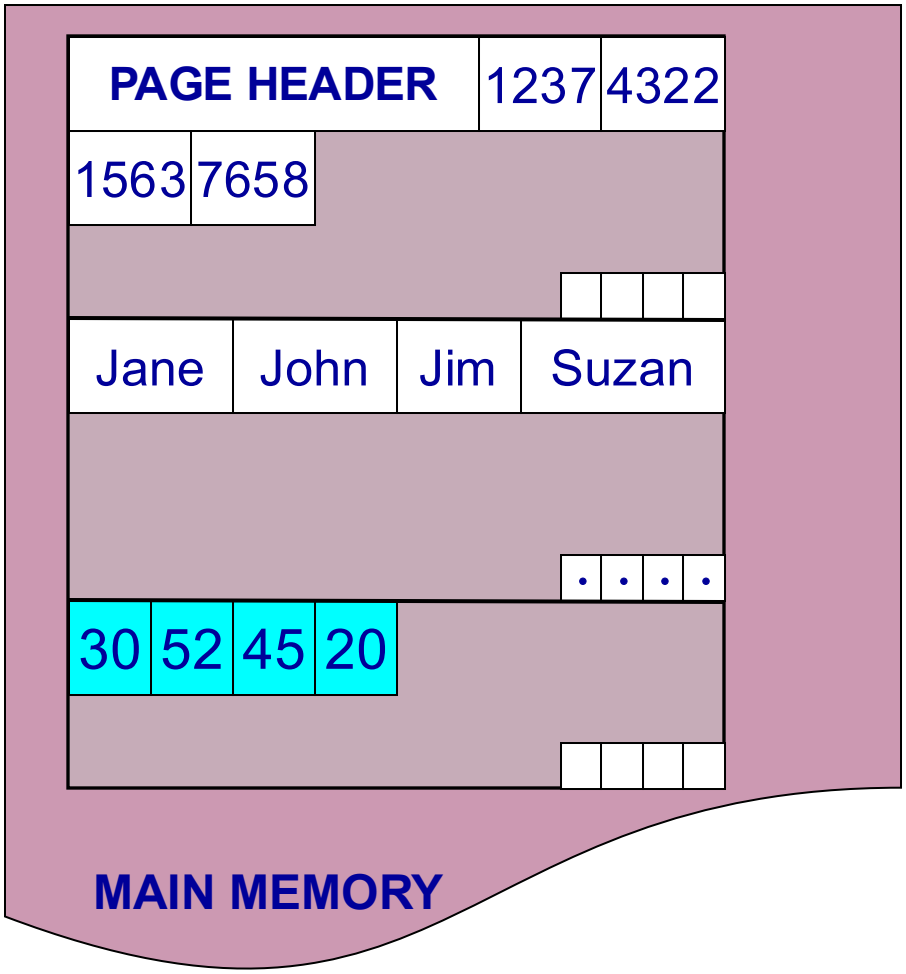
Partition Attributes Across (PAX)

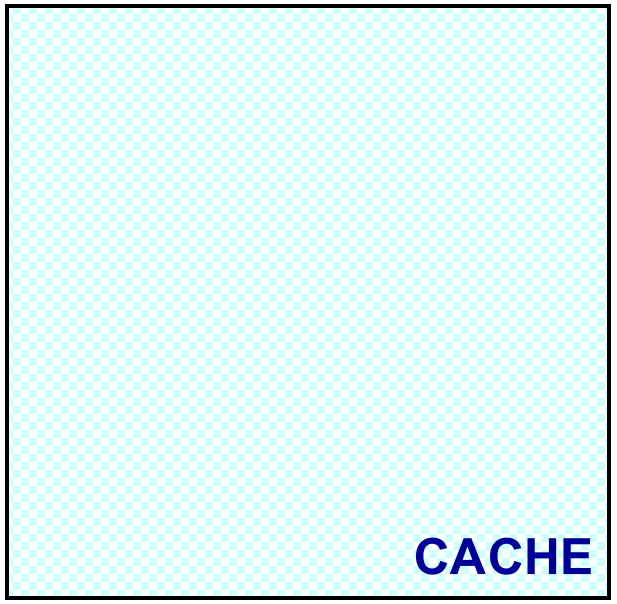


Partition data *within* the page for spatial locality

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using PAX



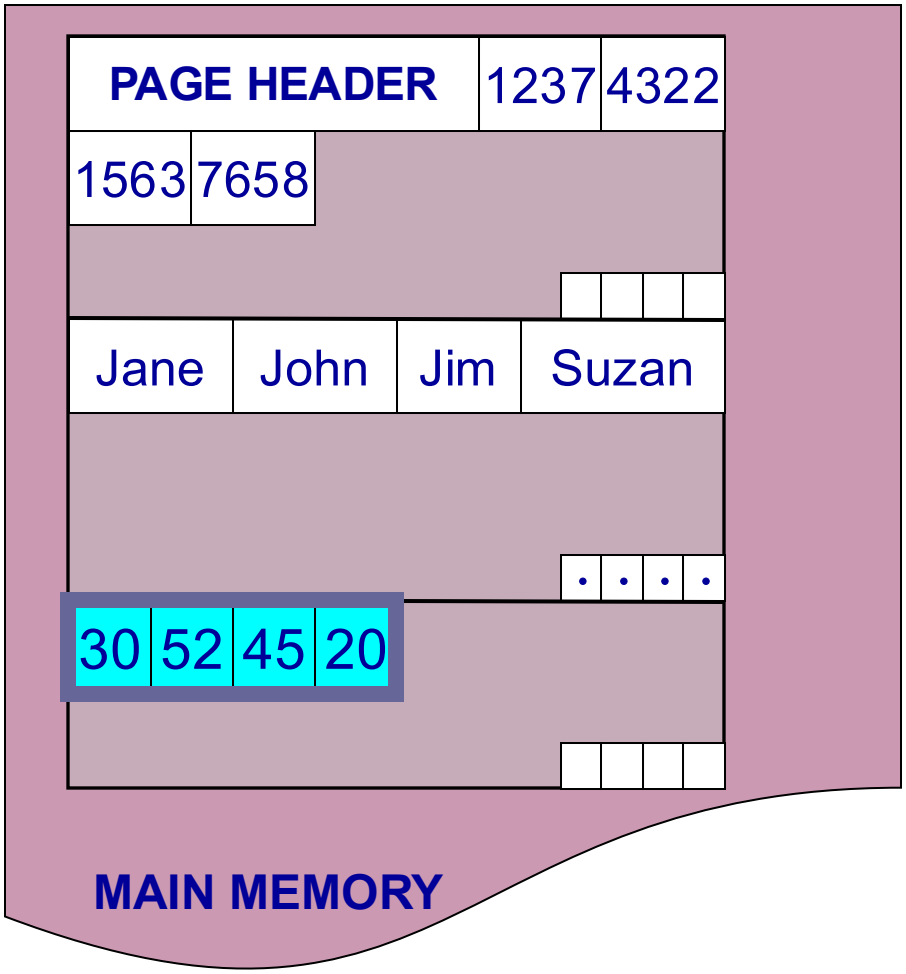


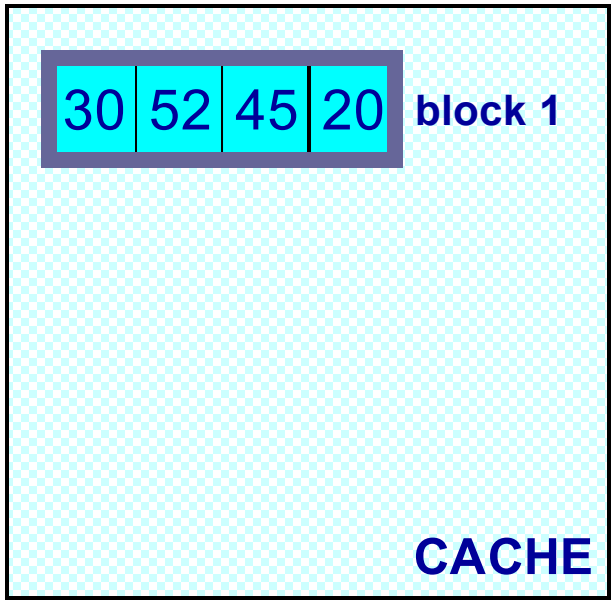

```
select name
from  R
where age > 50
```

Fewer cache misses, low reconstruction cost

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Predicate Evaluation using PAX

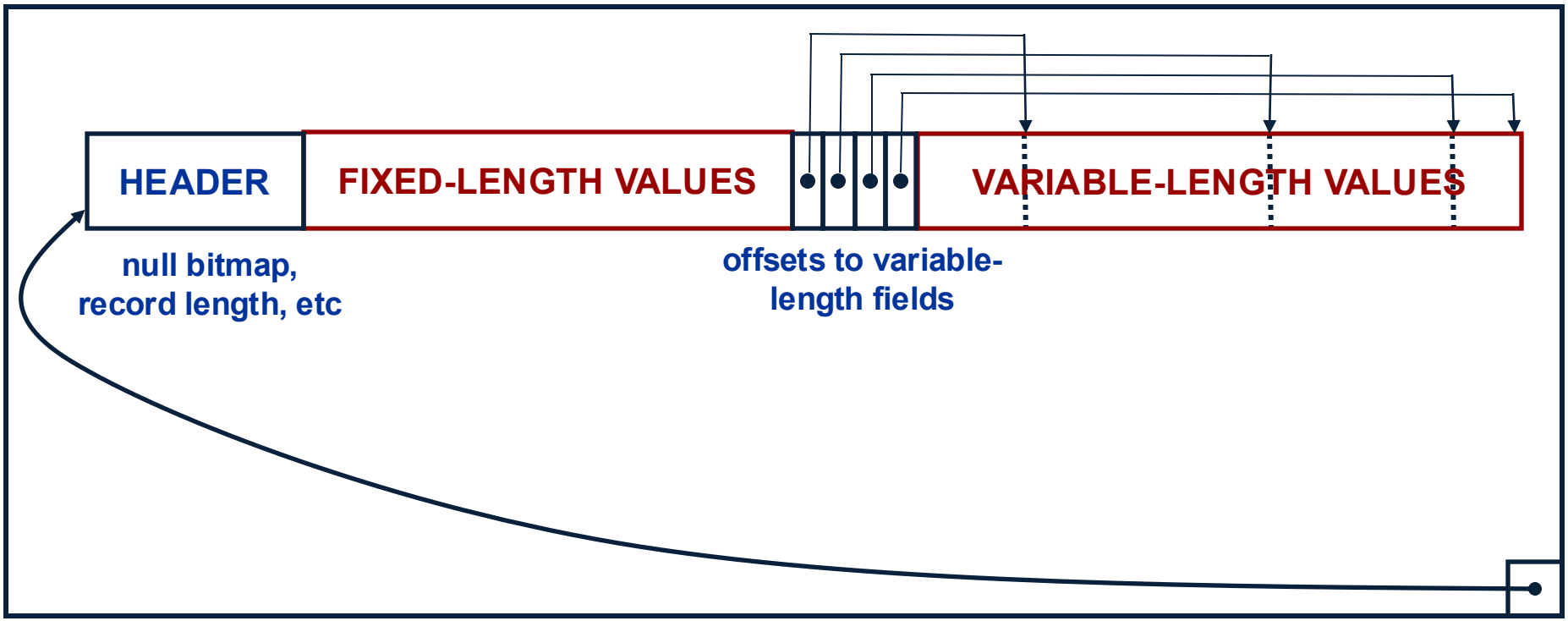





```
select name
from  R
where age > 50
```

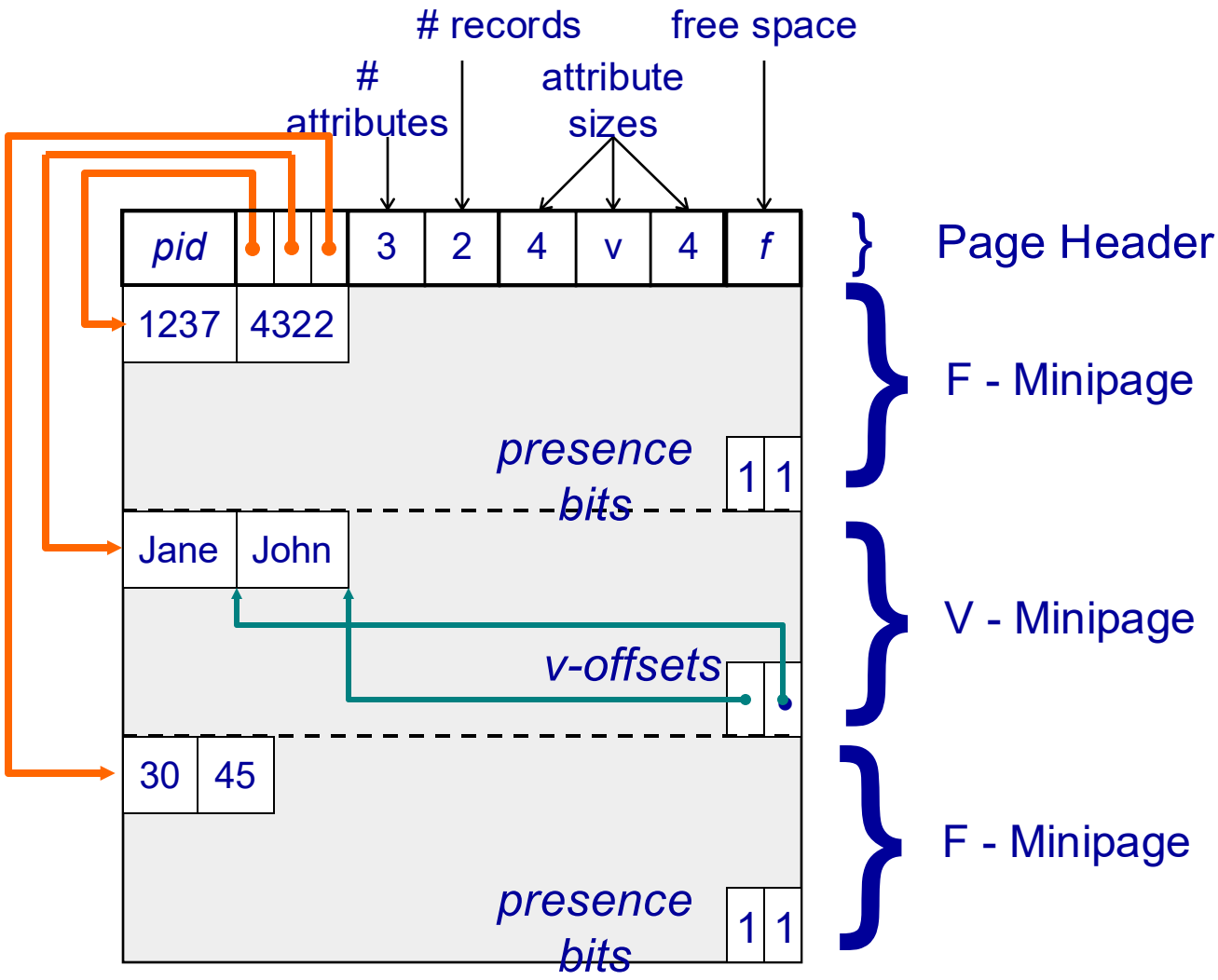
Fewer cache misses, low reconstruction cost

A Real NSM Record



NSM: All fields of record stored together + slots

PAX: Detailed Design



PAX: Group fields + amortizes record headers

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

PAX - Summary

- Improves processor cache locality
- Does not affect I/O behavior
  - Same disk accesses for NSM or PAX storage
  - No need to change the buffer manager

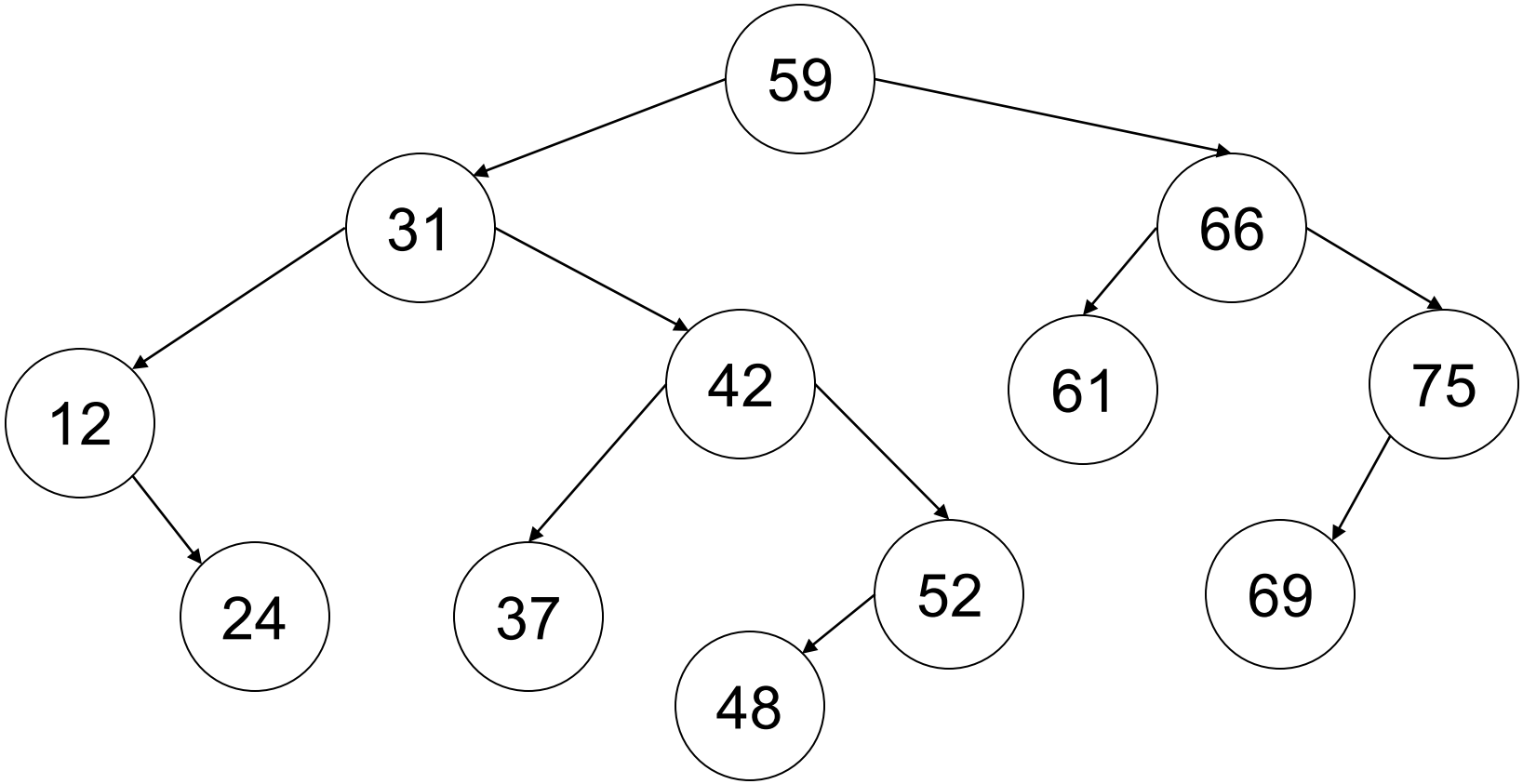
- Today:
  - Most (all?) commercial engines use a PAX layout of the disk
  - Beyond disk: Snowflake partitions tables horizontally into files, then uses column-store inside each file (hence, PAX)

Indexes: B+ Trees

Index

- An index is a file that allows direct access to the data file based on an attribute value
- The attribute is called a *key*; does no need to be a key in the table
- Index entries:
  (Key, RID) or (Key, List(RID)) pairs
- Index data structures: B+ trees, hash tables

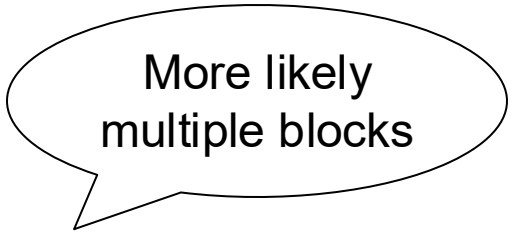
Review: Binary Search Tree



B+ Trees Properties

- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

B+ Trees

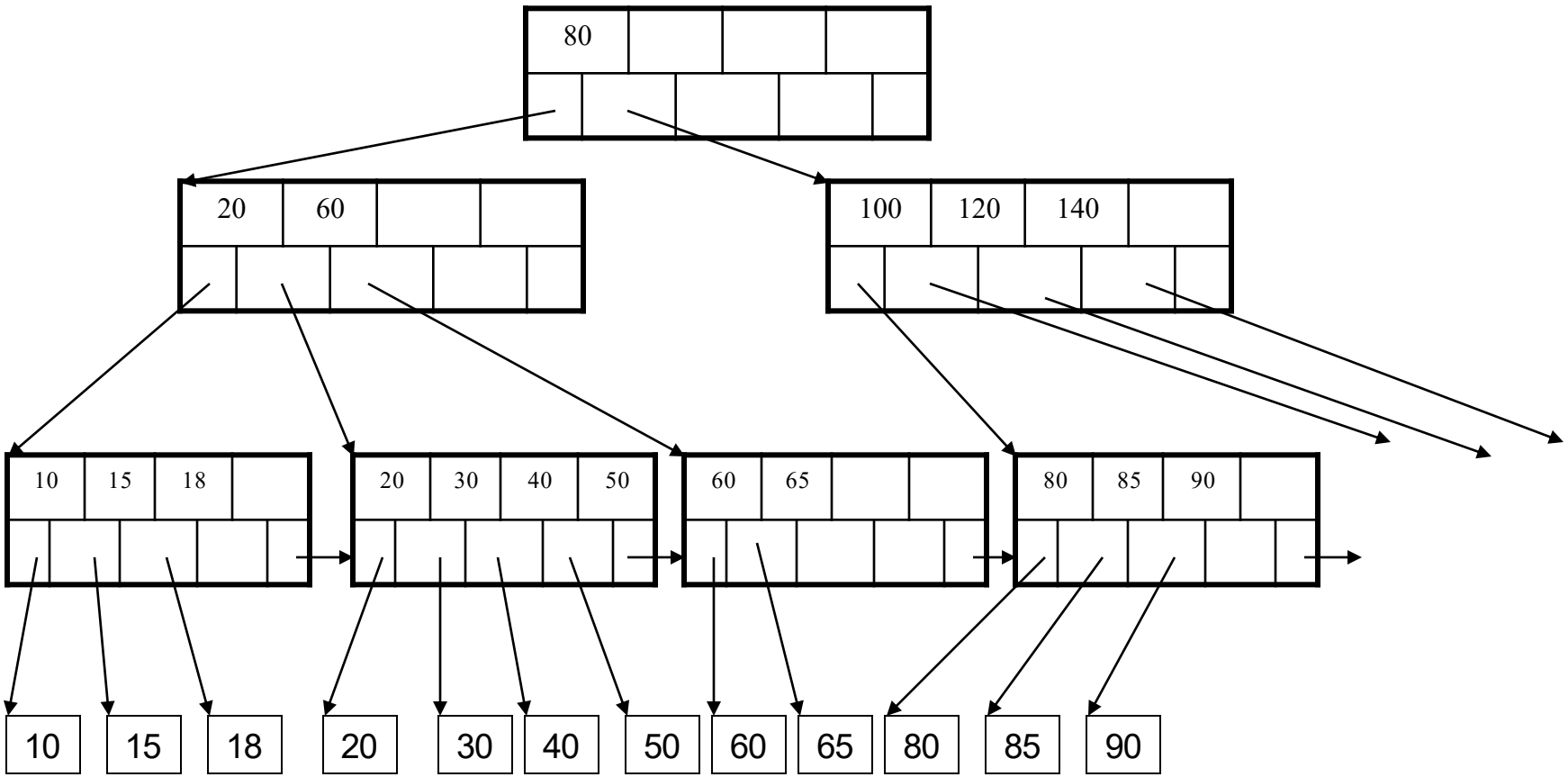


- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Store multiple keys and children
  - Read 1 page, use many keys

- Idea in B+ Trees
  - Keys are stored only on leaves
  - Internal nodes used only for guiding
  - Leaves are linked in a list, for range queries

B+ Tree Example

d = 2

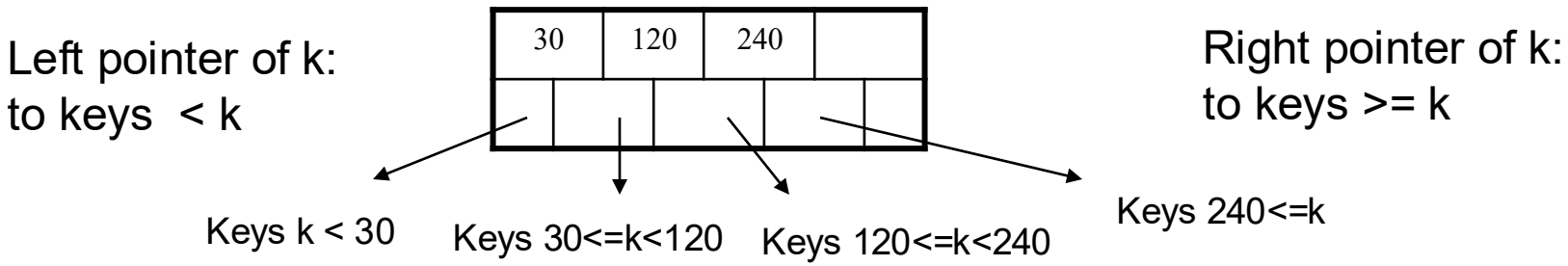


B+ Trees Details

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)

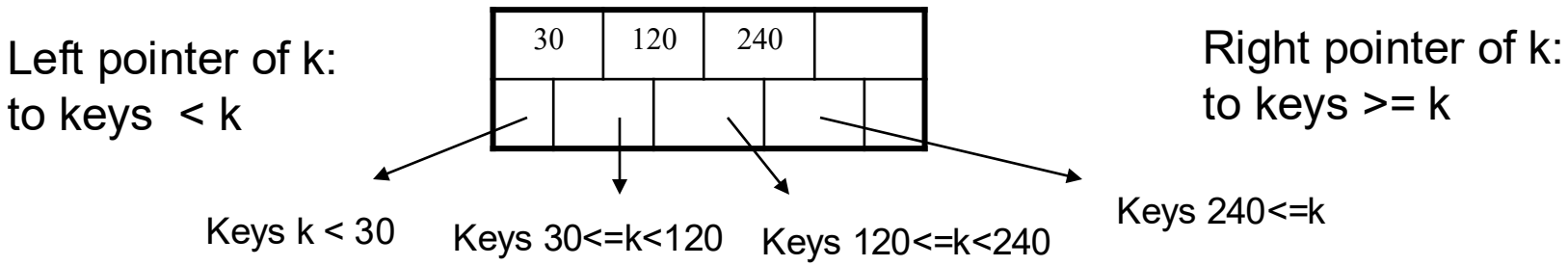
B+ Trees Details

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

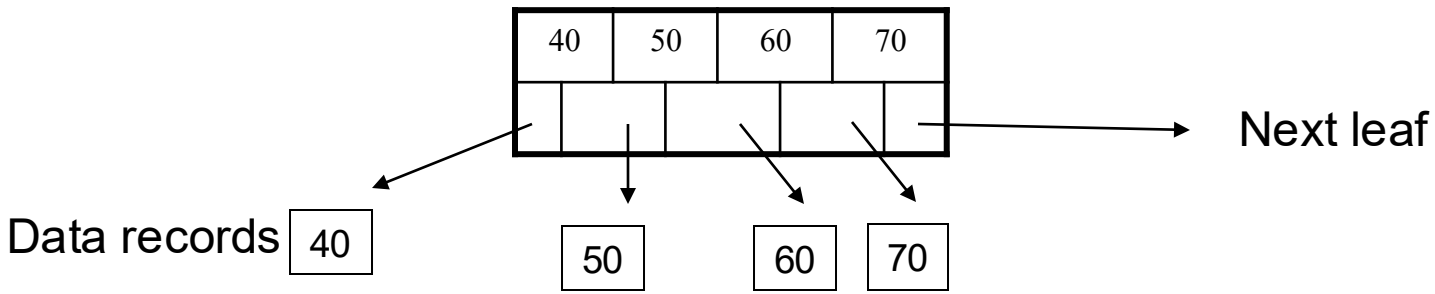


B+ Trees Details

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**



- Each leaf has **d <= m <= 2d keys**:



Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$

Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N = 30/7 = 4$

Search time

- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N = 30/7 = 4$
- Search time better than log N??

Search time

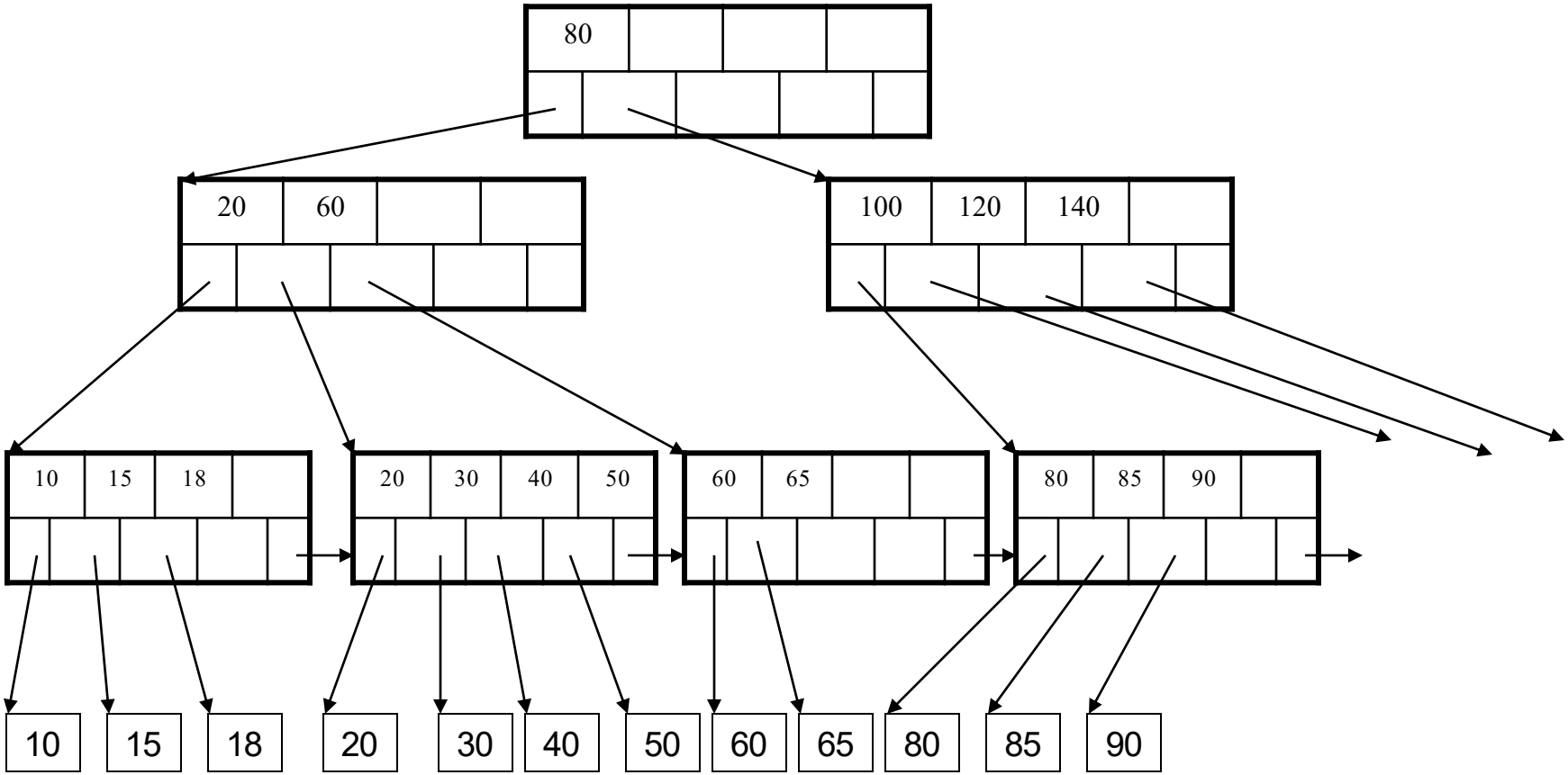
- Every B+ tree is perfectly balanced
  - Assume each node has m children:
  - Depth=$\log_m N$
- Ex. N = 1,000,000,000
  - Binary search tree: log N = 30
  - B+ tree, assume m=128: $\log_m N = 30/7 = 4$
- Search time better than log N??
  - No: binary search in a node takes log m
  - Search time = $\log_m N$ * log m = log N

B+ Trees in Practice

- Typical order: 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
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

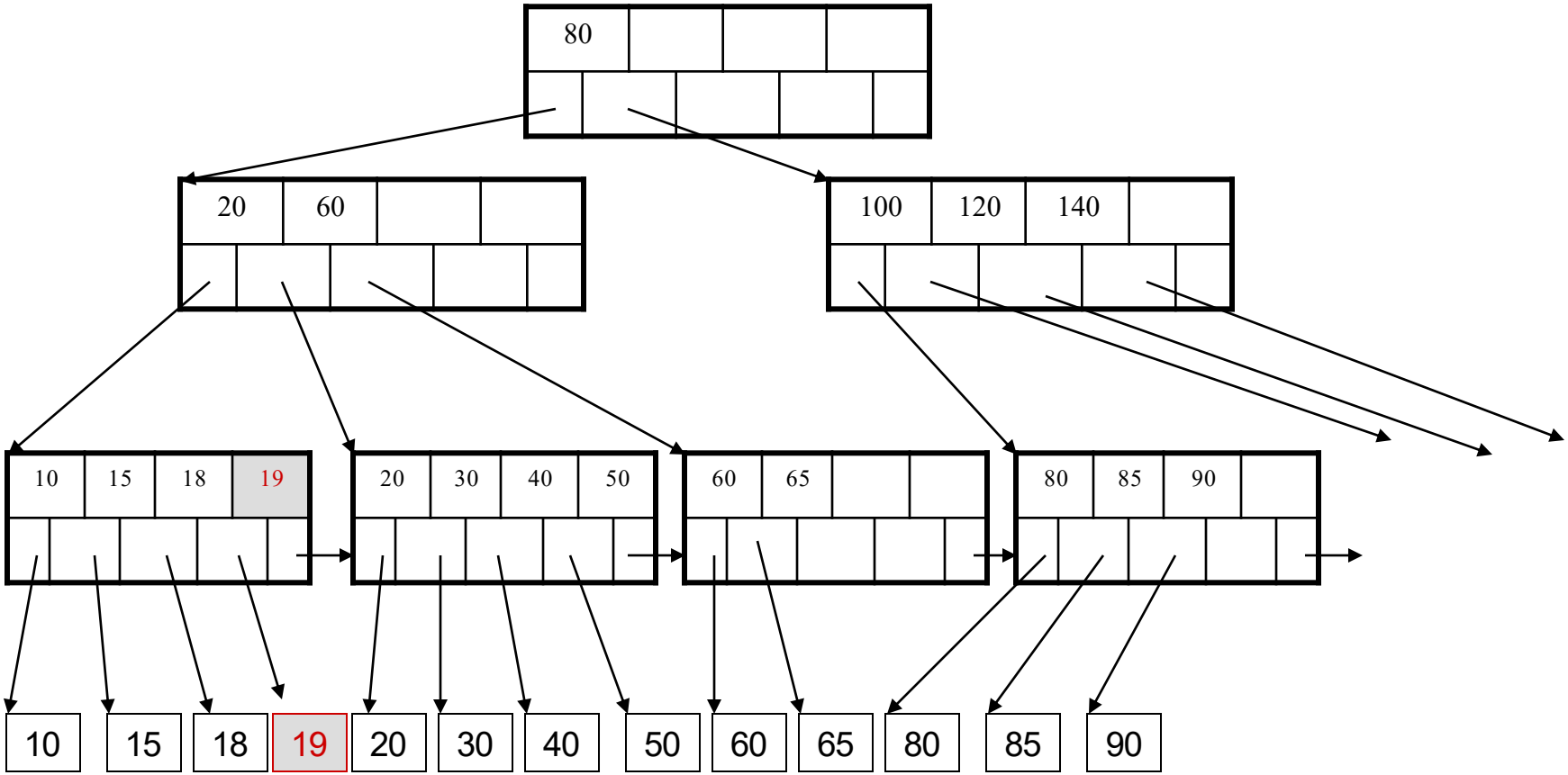
Insertion in a B+ Tree

Insert K=19



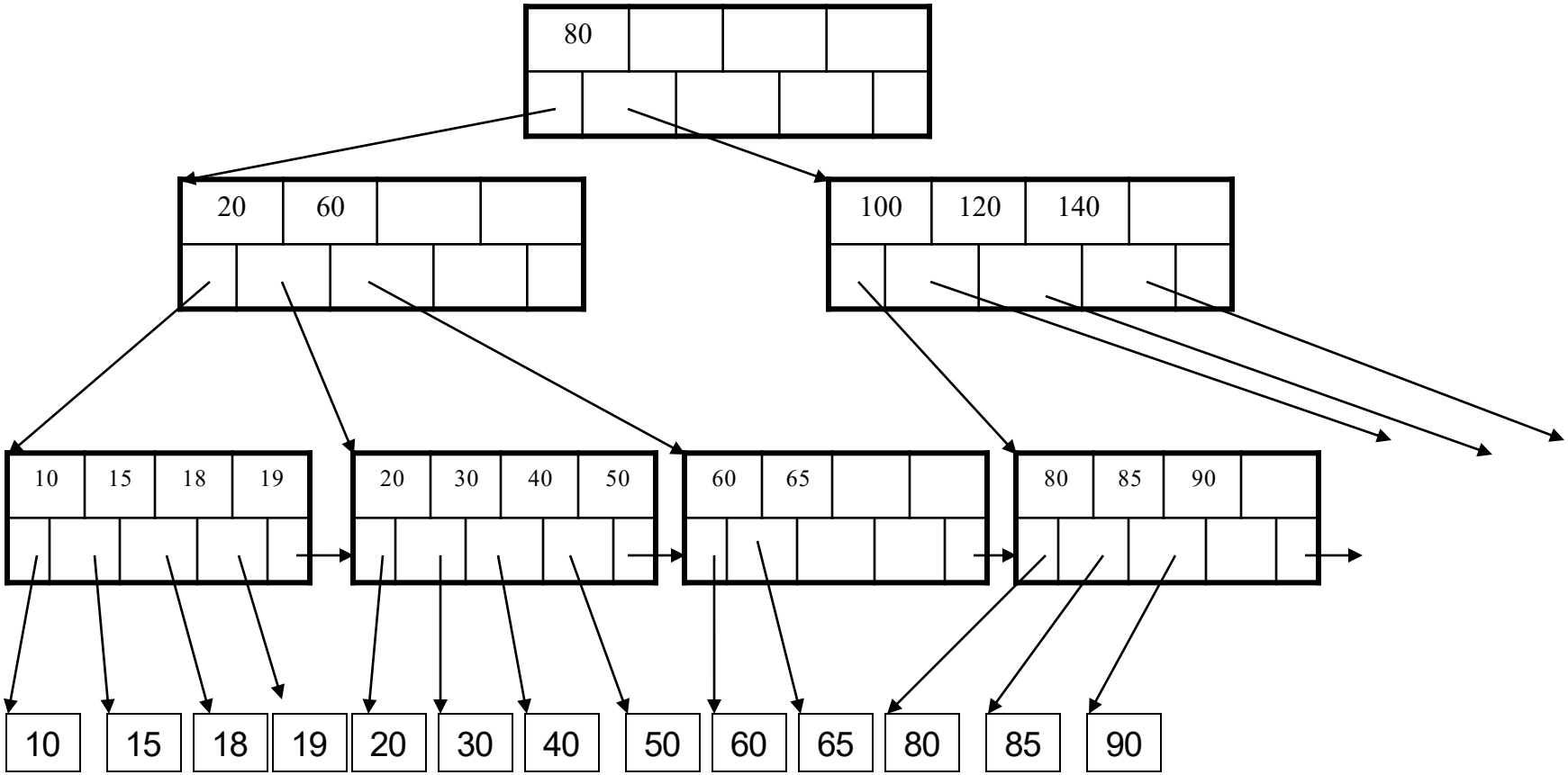
Insertion in a B+ Tree

After insertion



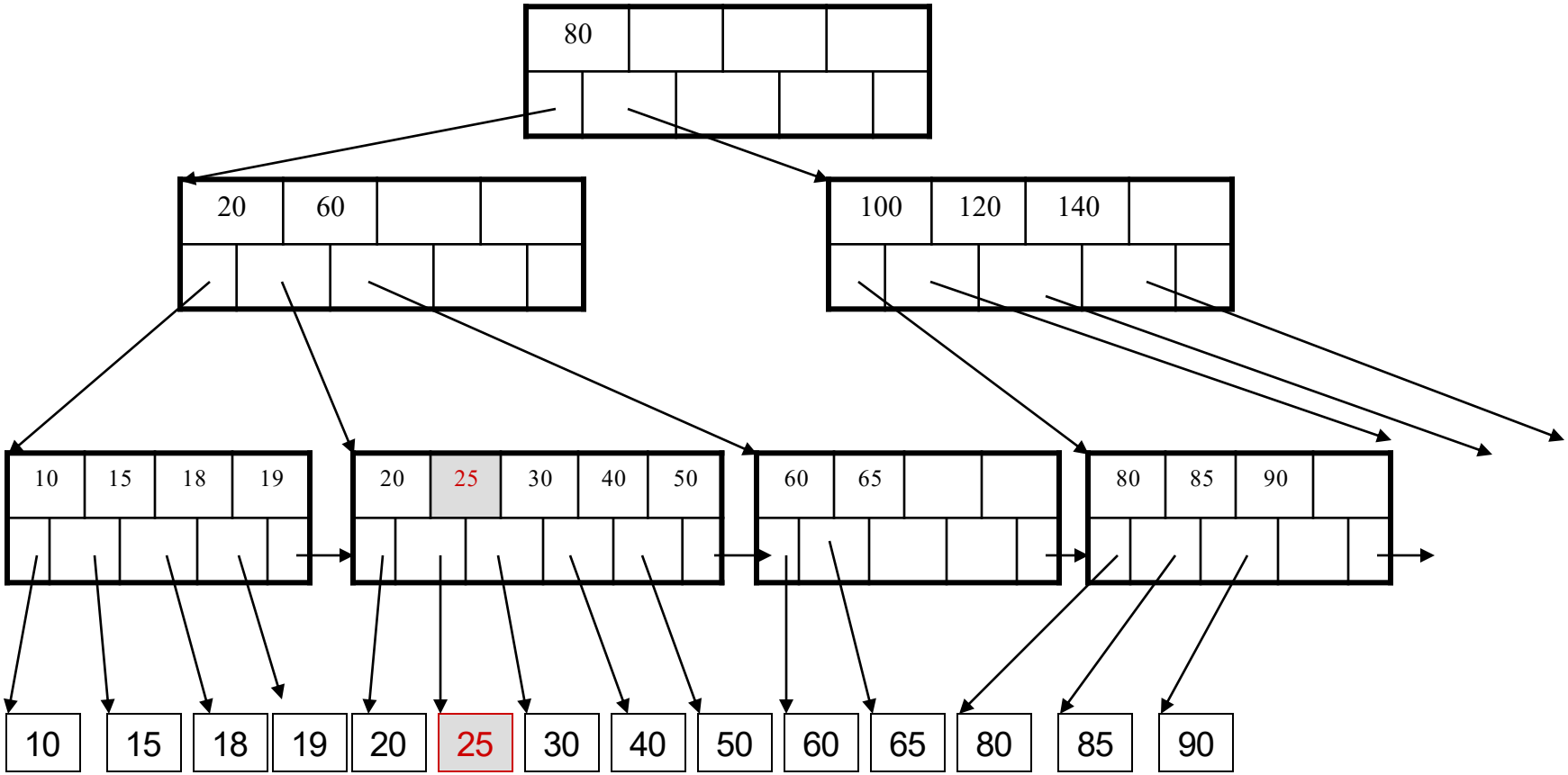
Insertion in a B+ Tree

Now insert 25



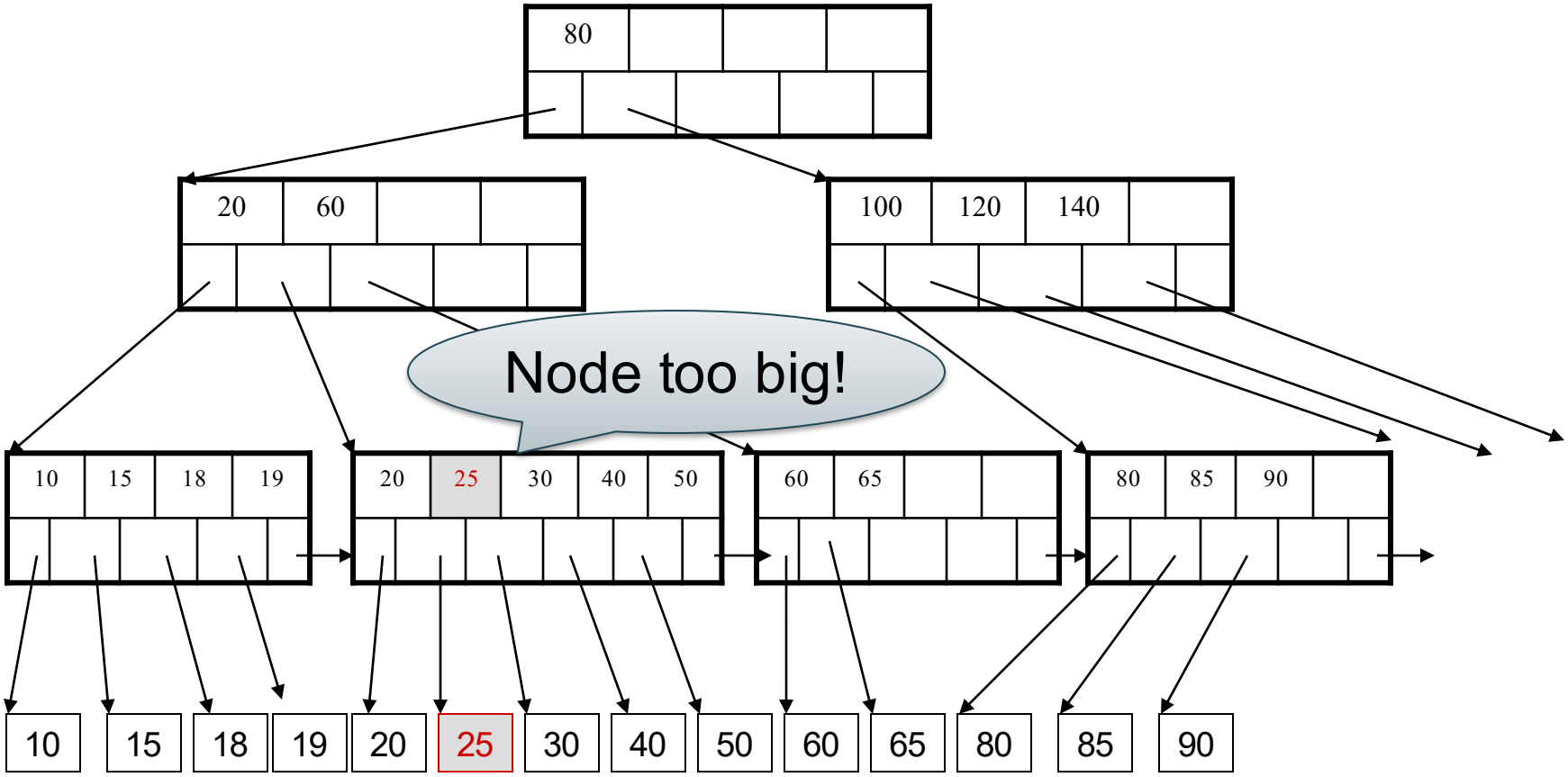
Insertion in a B+ Tree

After insertion



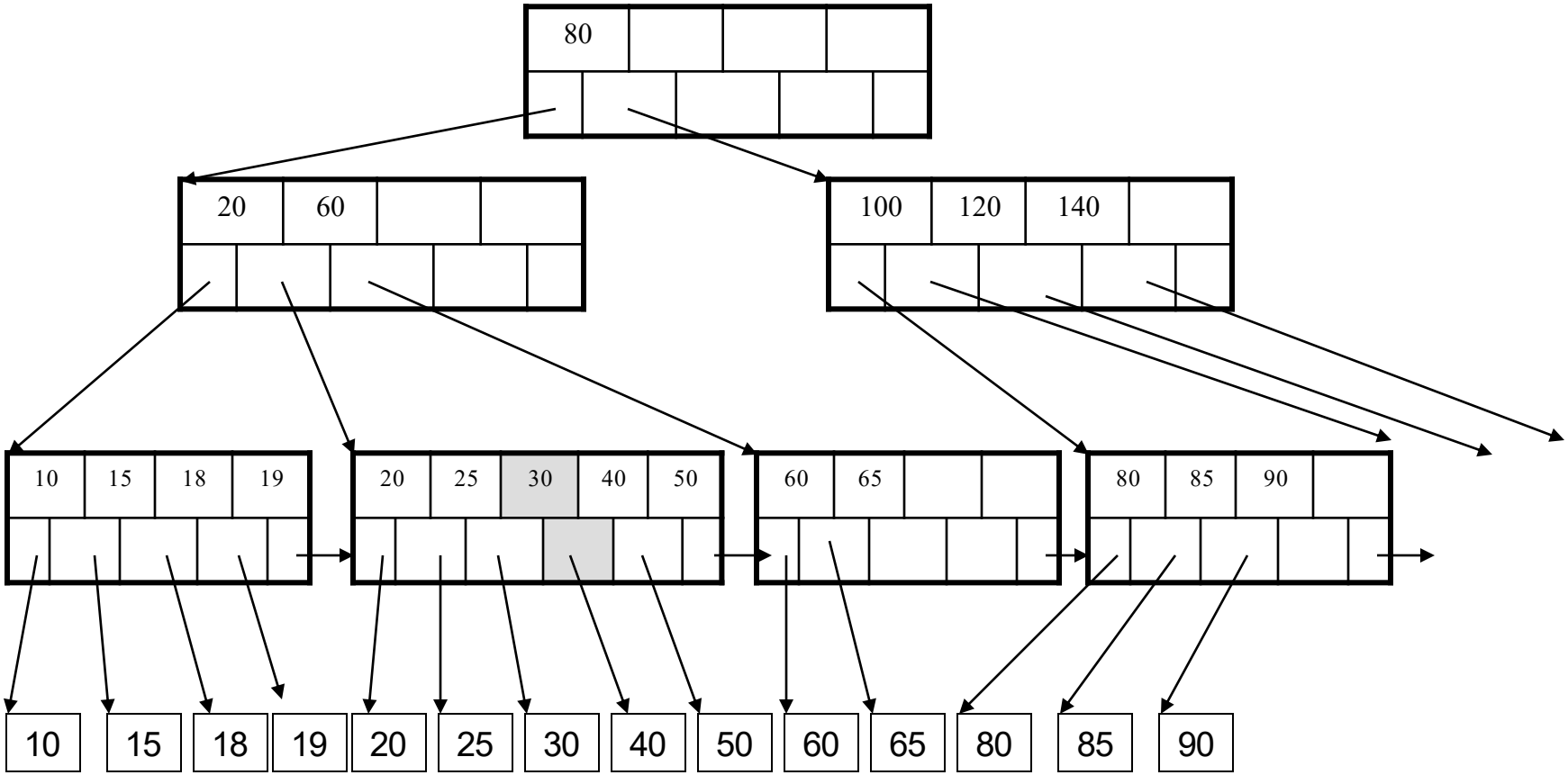
Insertion in a B+ Tree

After insertion



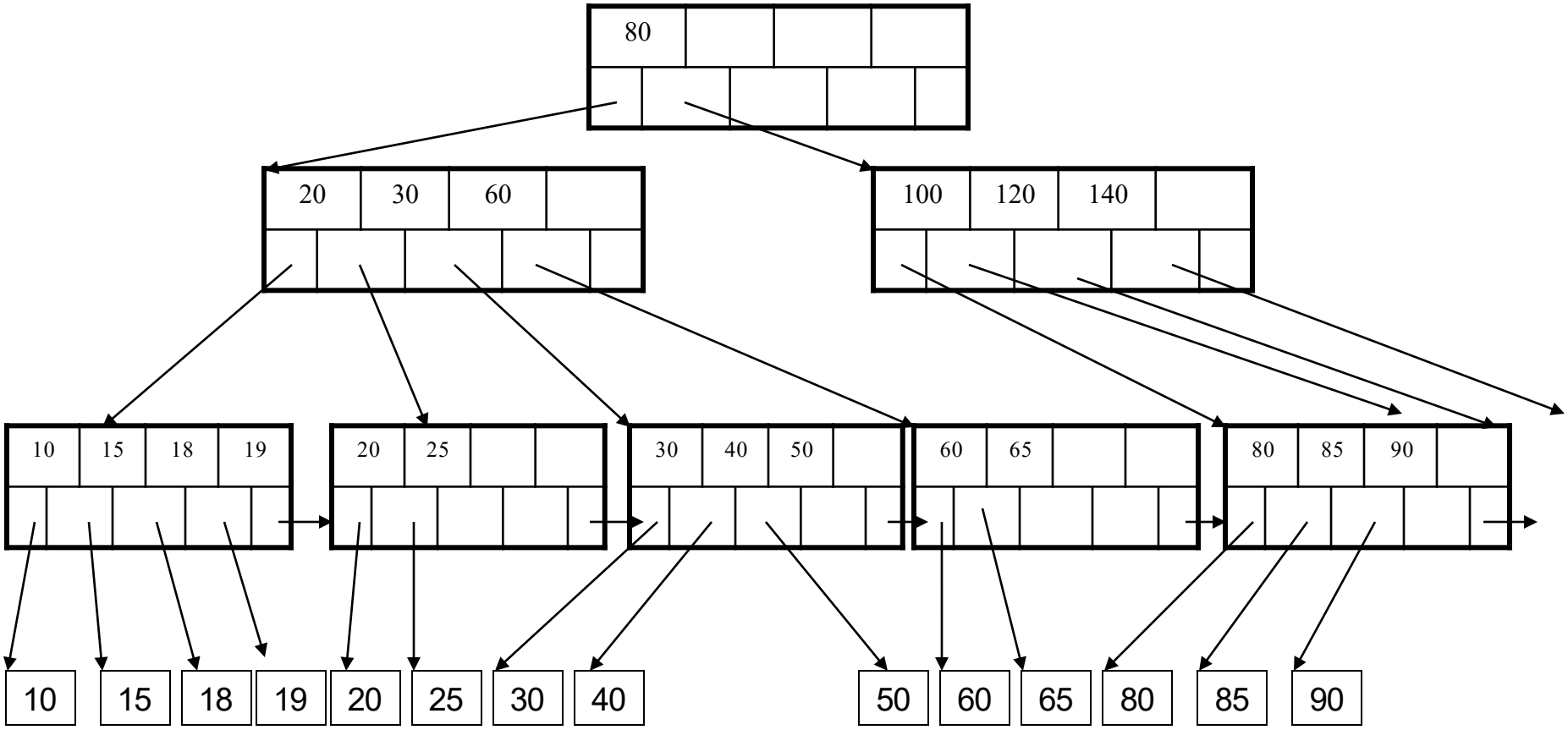
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split

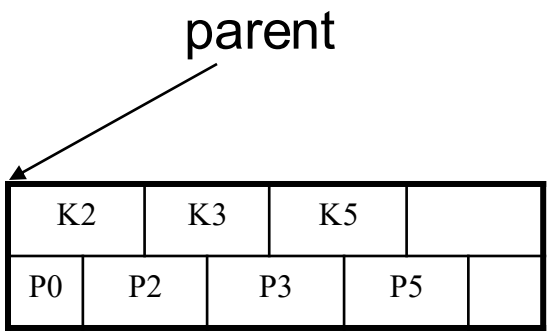


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

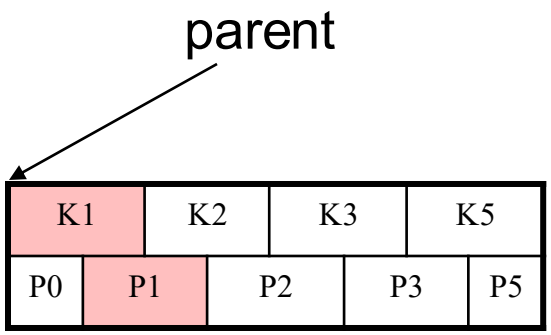


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

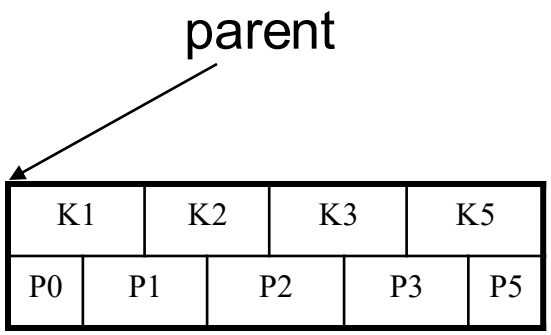


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k4

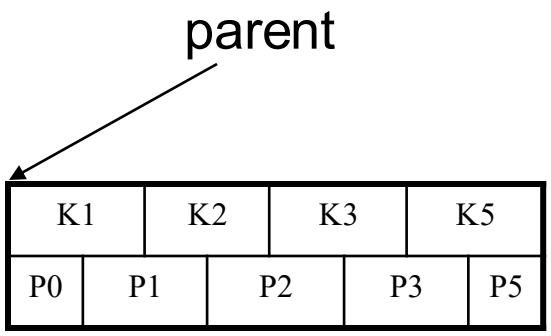


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

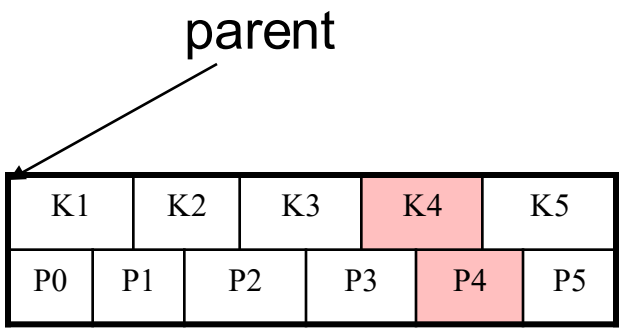


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

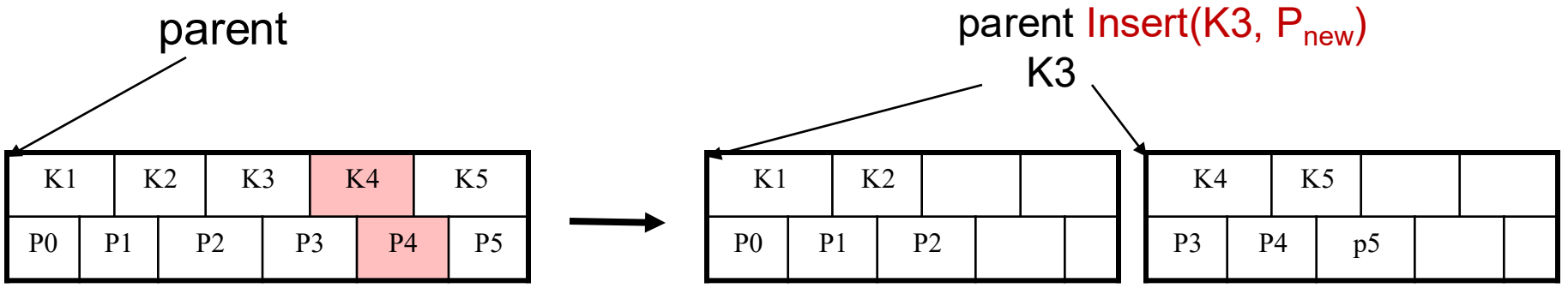


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

Insert k4

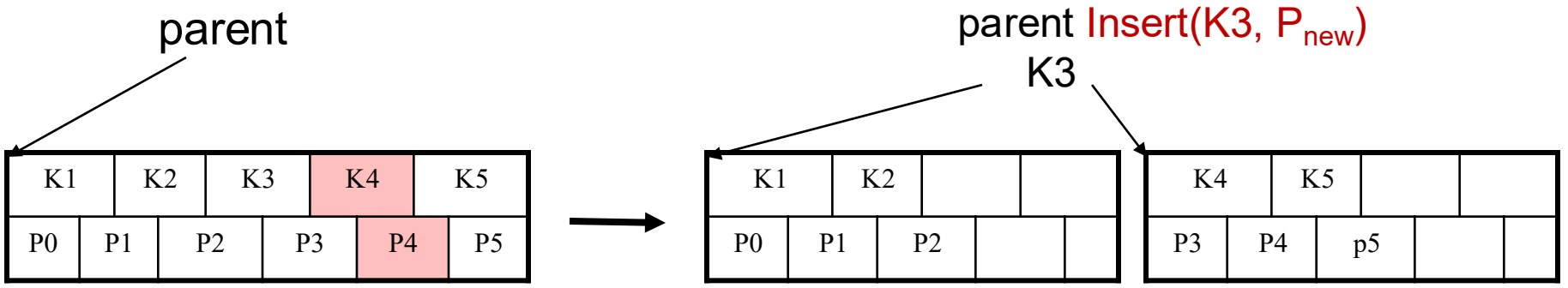


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

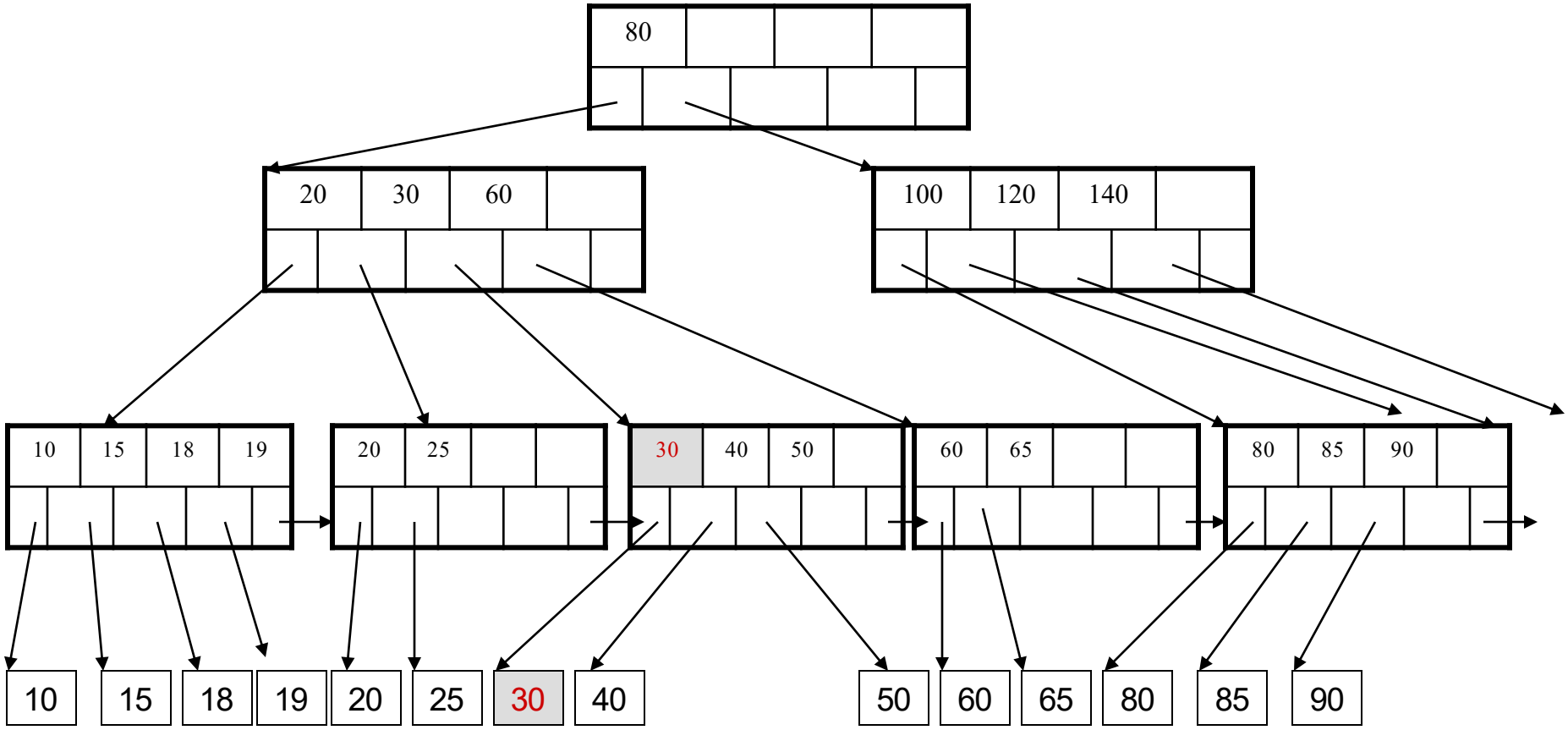
Insert k4



- If leaf, also keep K3 in right node
- When root splits, new root has 1 key only

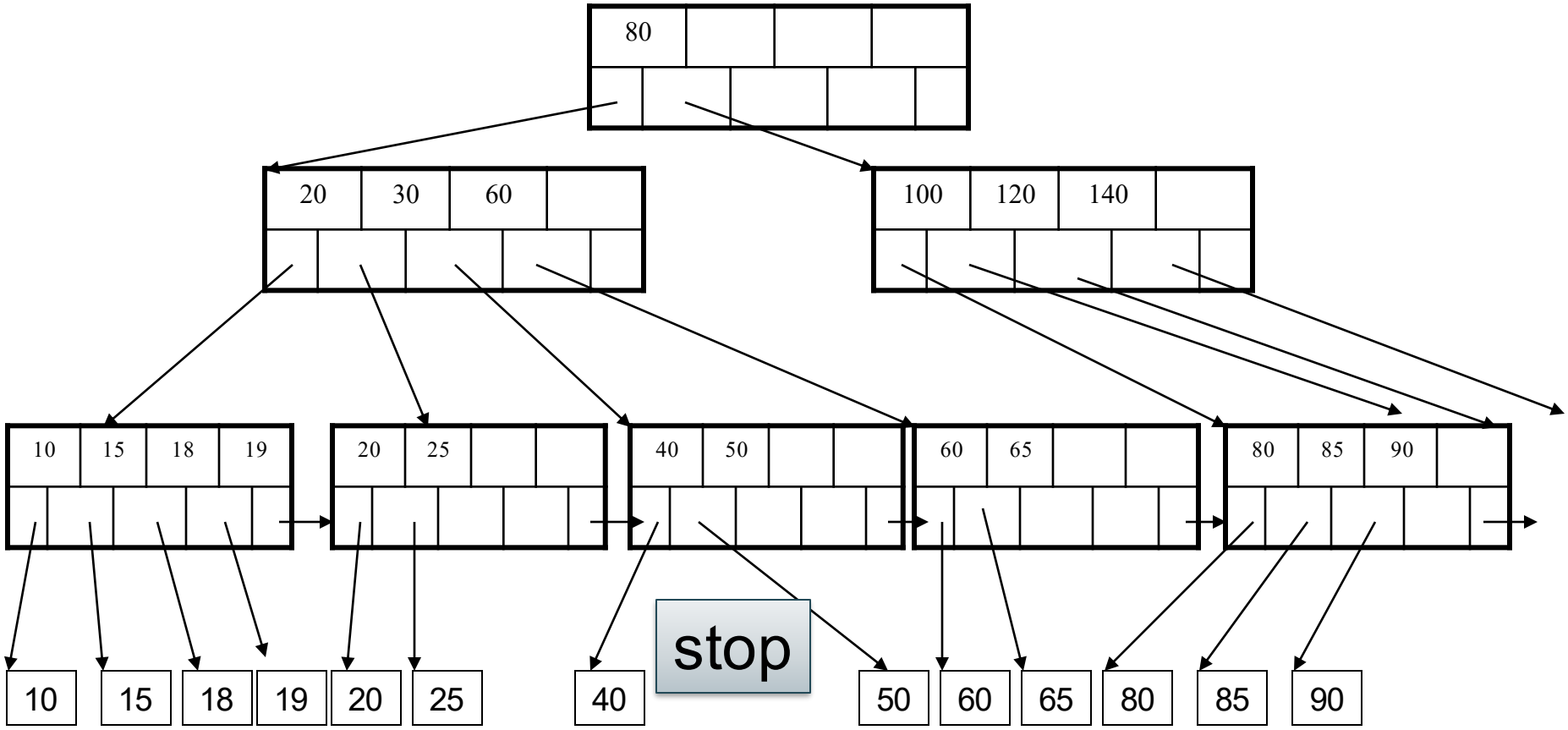
Deletion from a B+ Tree

Delete 30



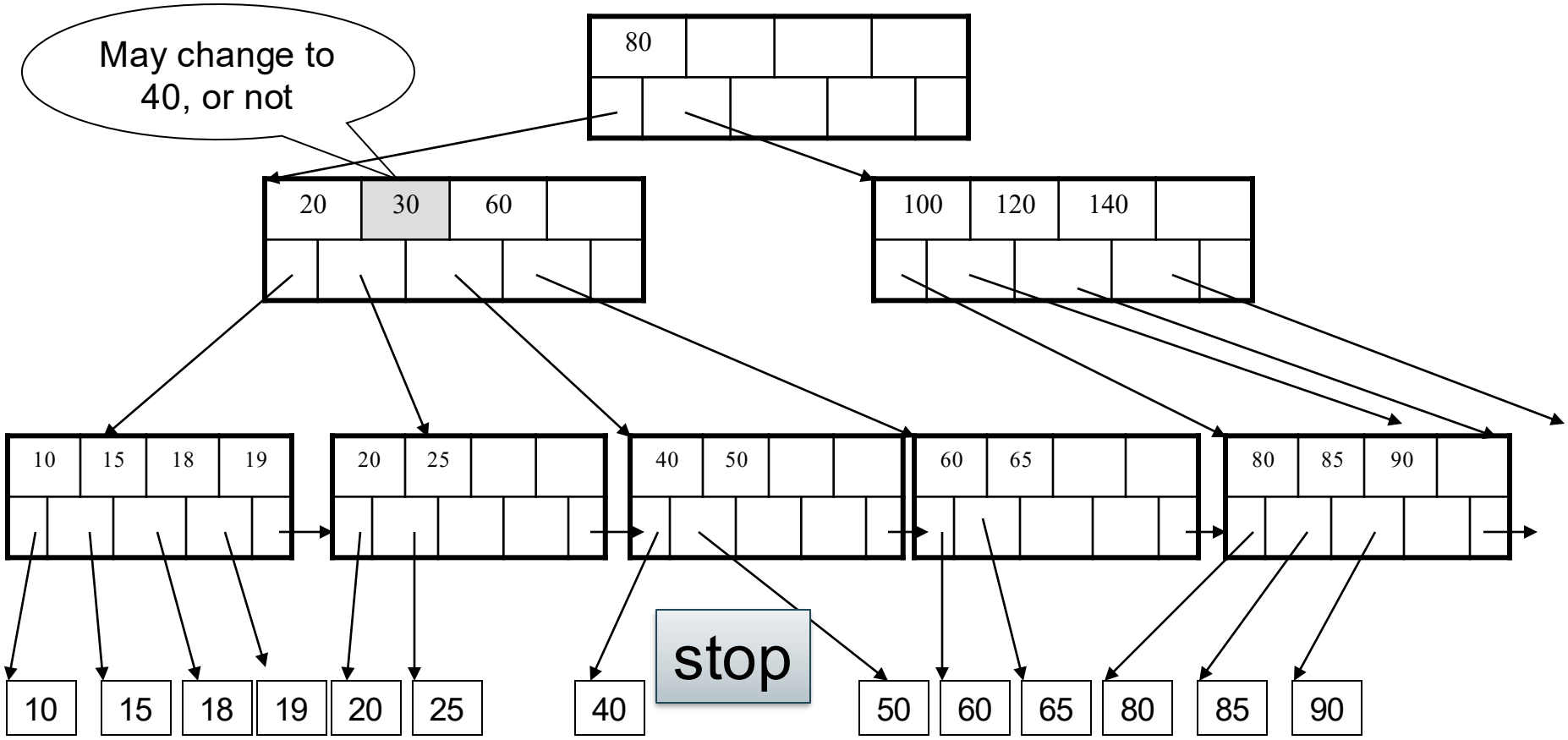
Deletion from a B+ Tree

After deleting 30



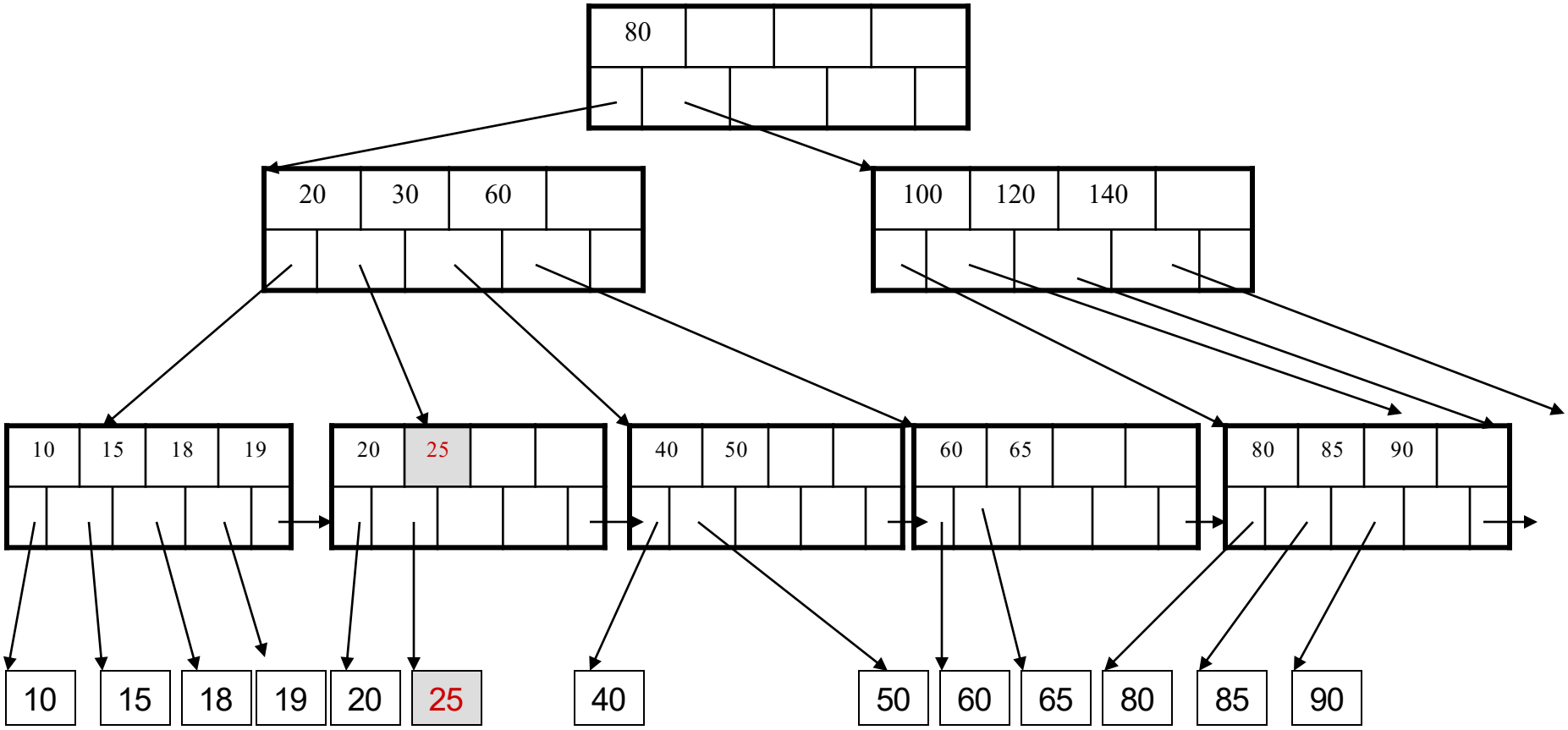
Deletion from a B+ Tree

After deleting 30



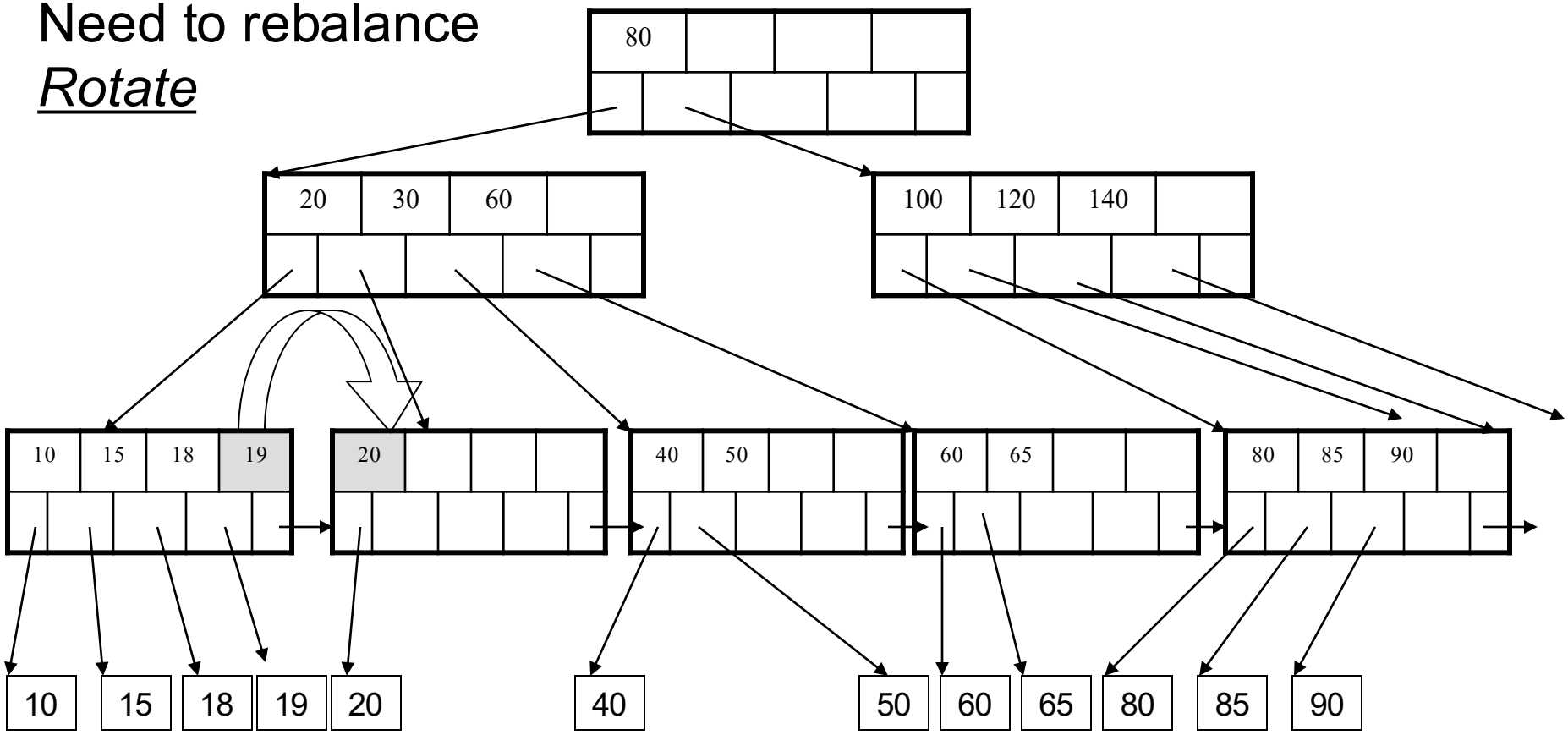
Deletion from a B+ Tree

Now delete 25

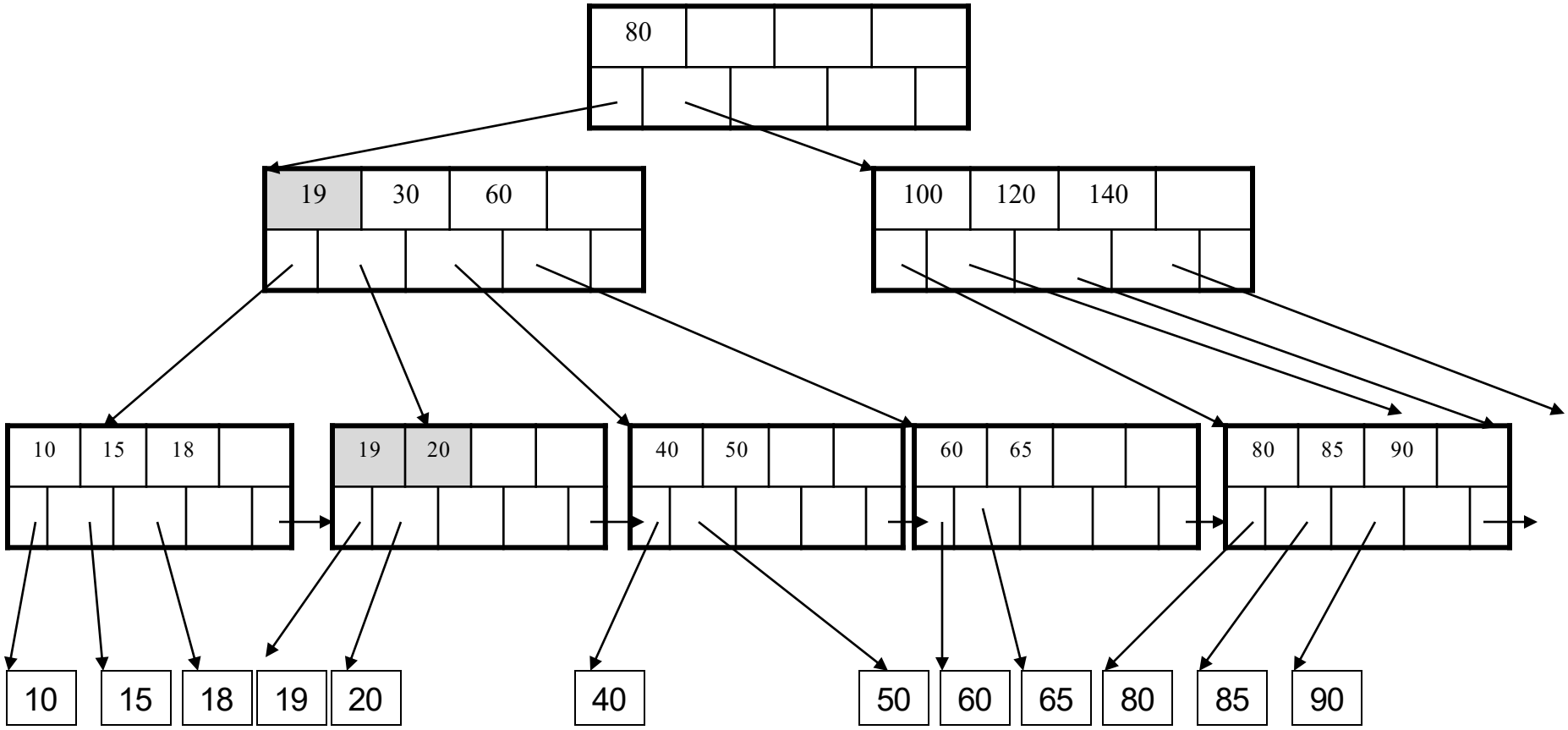


Deletion from a B+ Tree

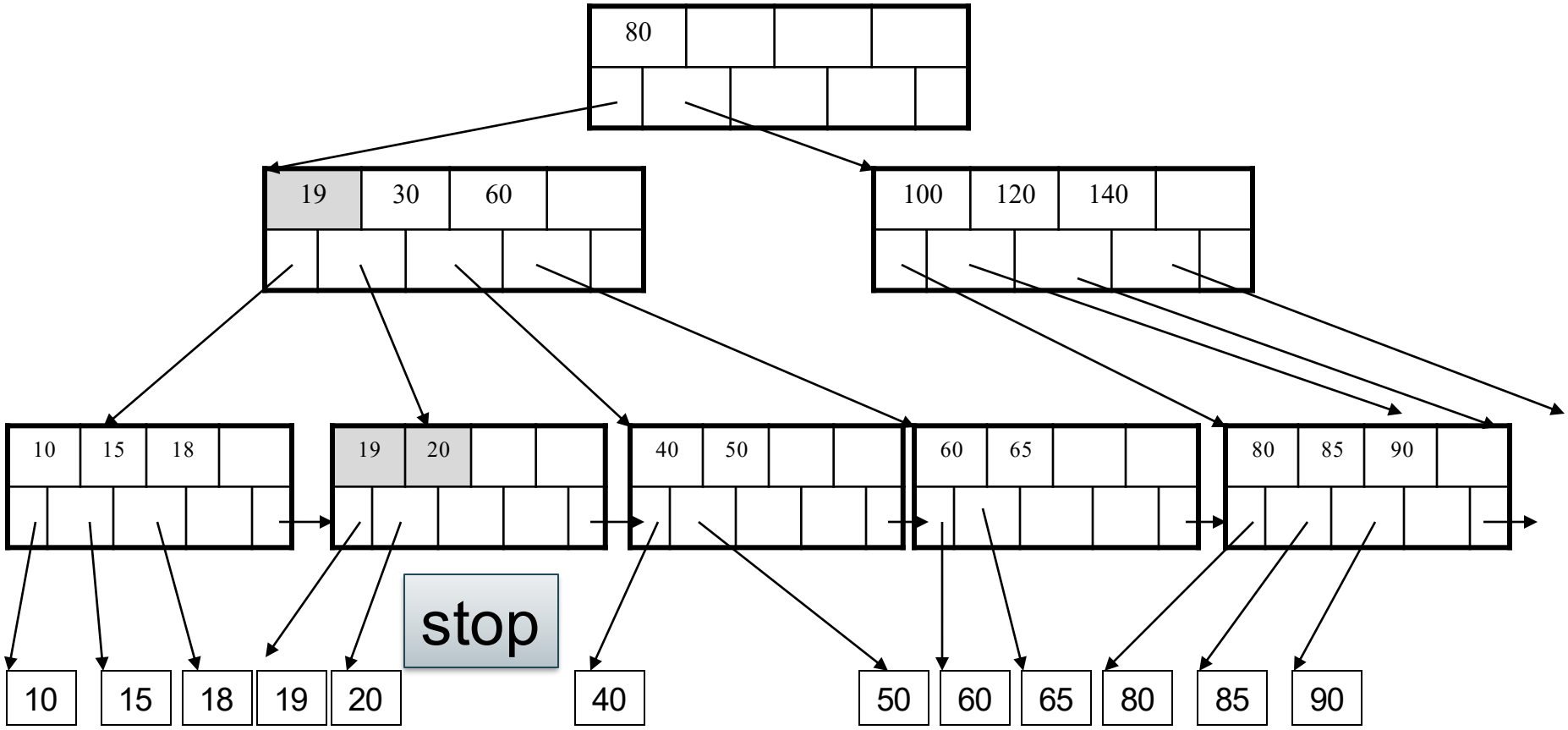
After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree

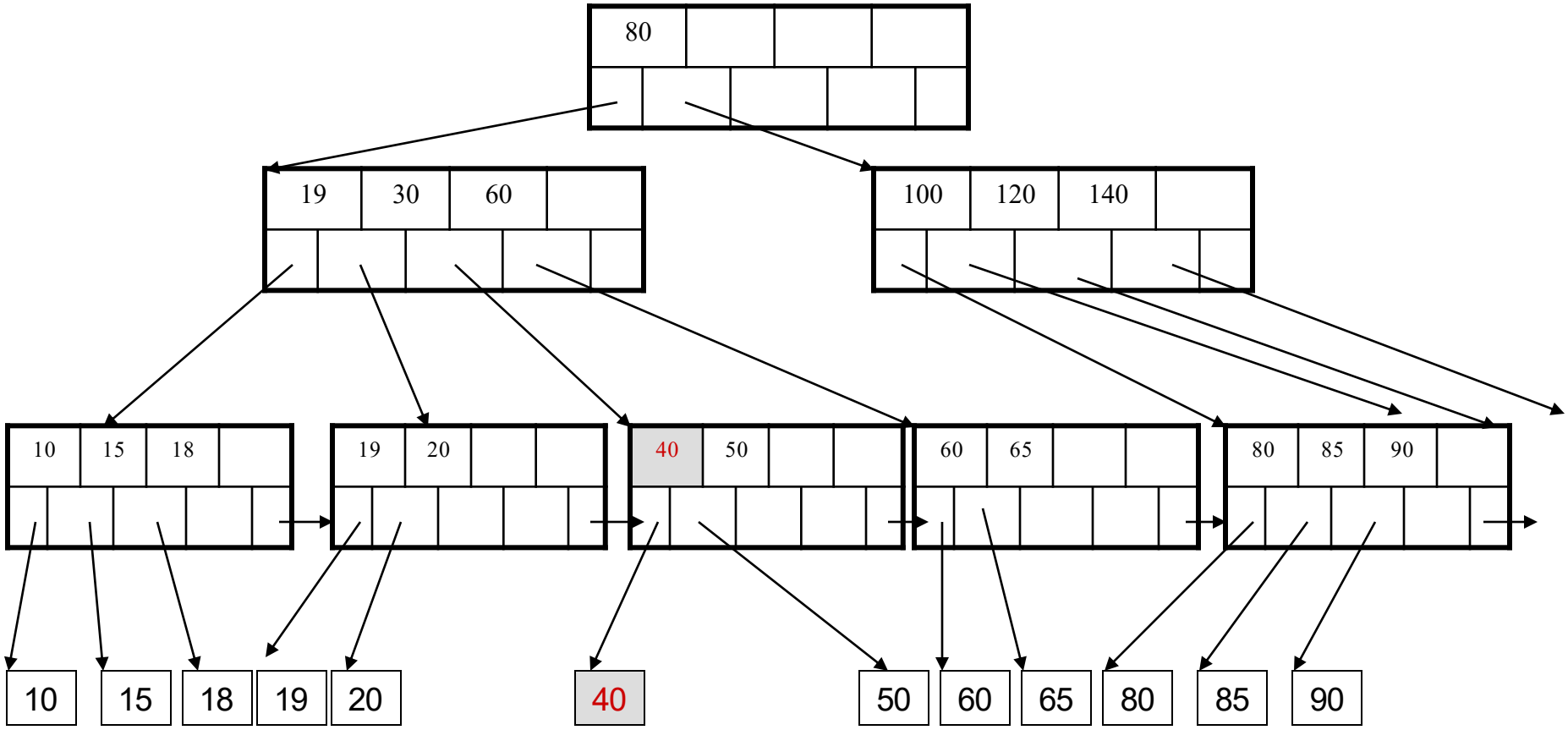


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

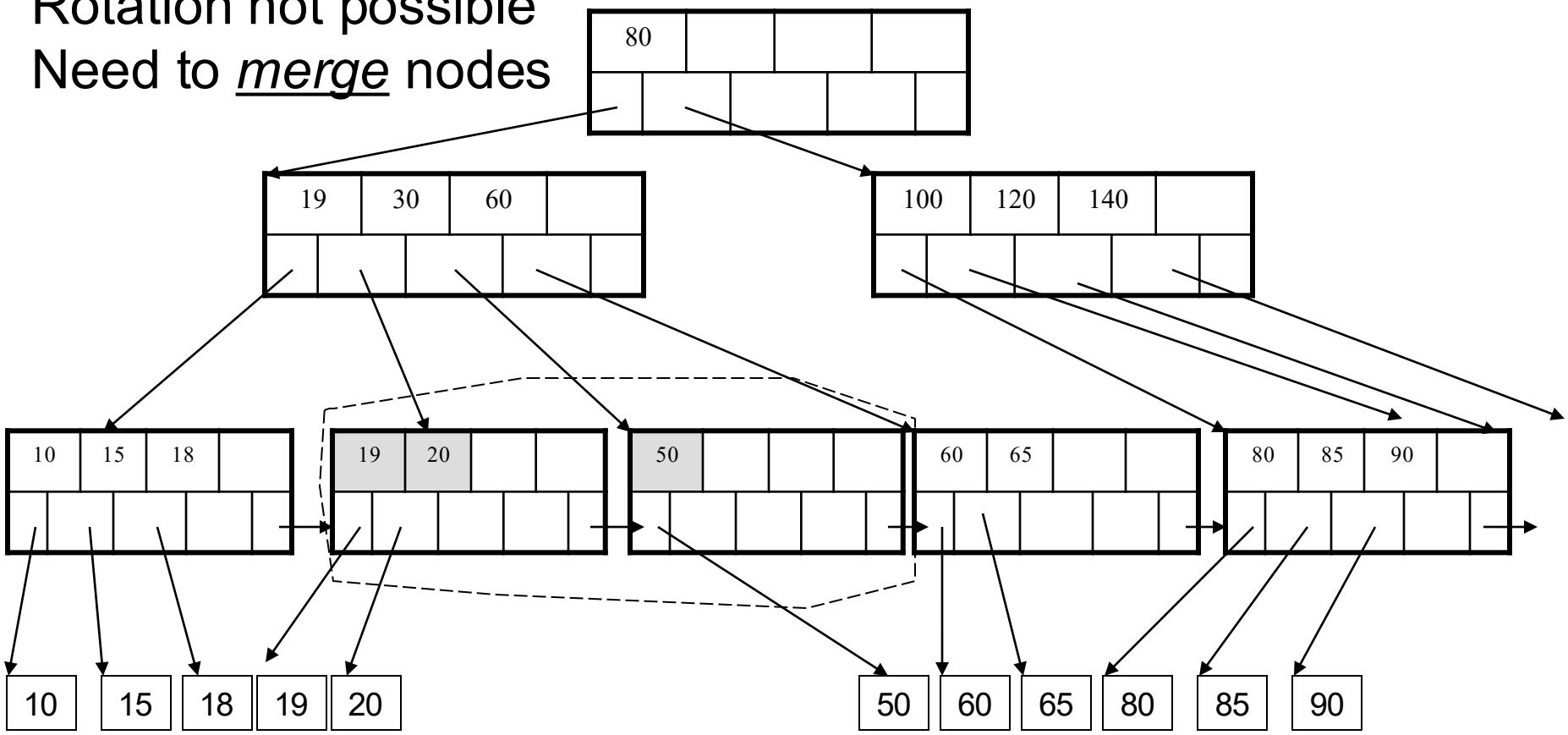


Deletion from a B+ Tree

After deleting 40

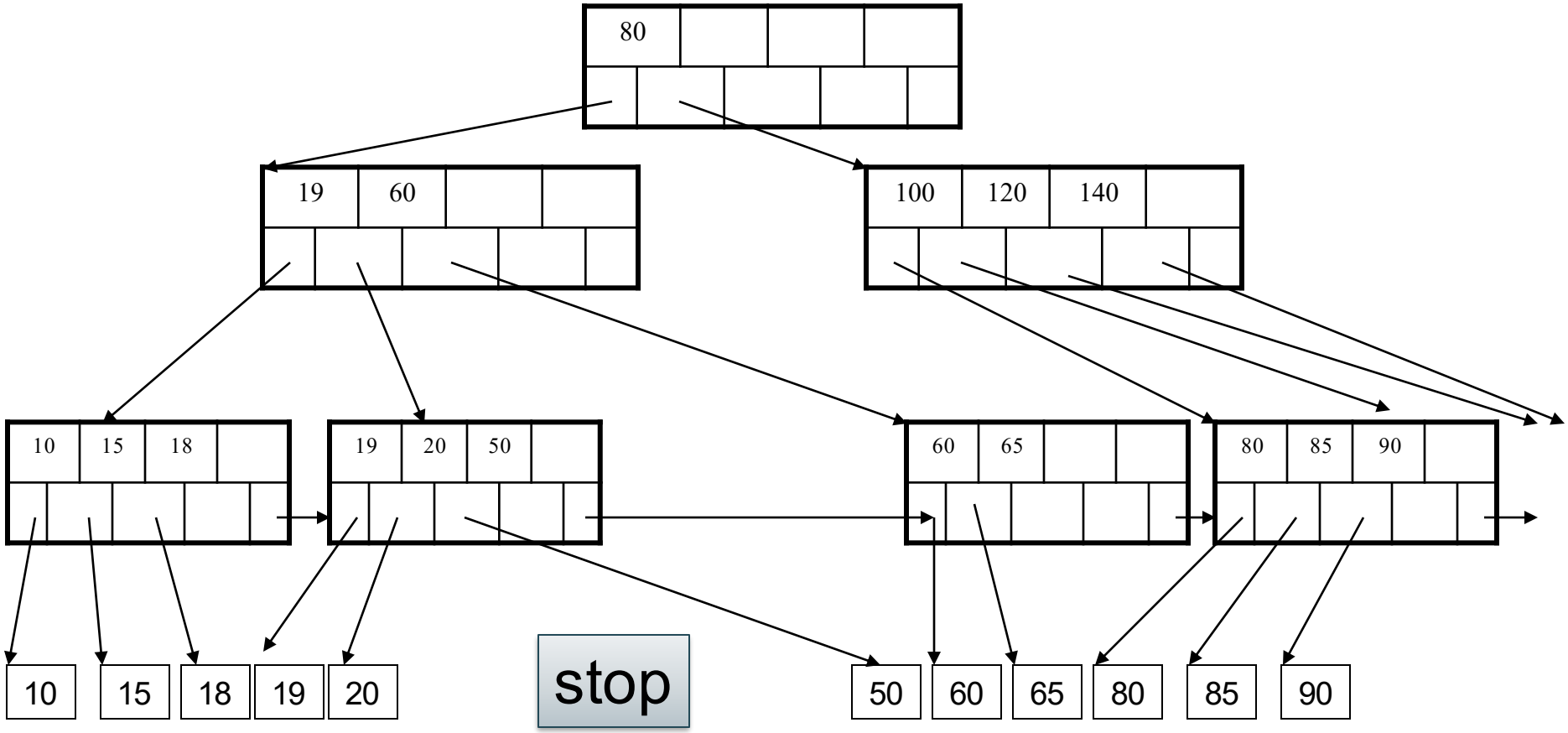
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



Deletion: Summary

Delete (K, P)

- Delete K, P from the leaf
- If capacity ≥ min: **Stop**
- If neighbor capacity > min: rotate and **Stop**
- Merge with a neighbor P' (choose right or left) and steal a key K' from parent
  - Parent: Delete (K', P')

B+ Tree Summary

B-trees

- Node as large as one block
- Always perfectly balanced

B+ -trees

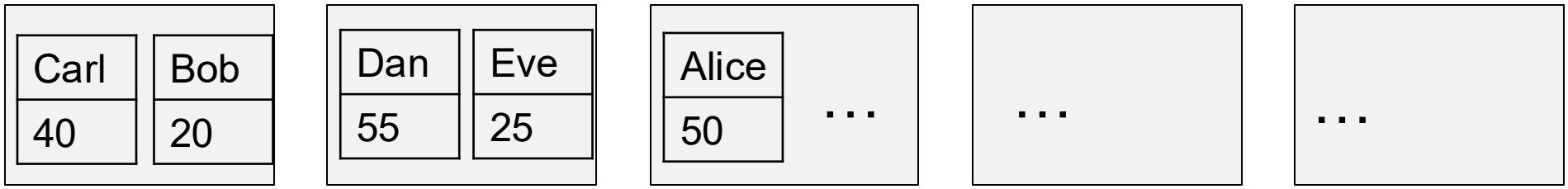
- Leaves: contain all (Key,RID) pairs
- Internal nodes: keys for navigation only
- Leaves are linked, for range searches

Index in SQL

```
create table Person(name text, age int)
```

Index in SQL

```
create table Person(name text, age int)
```

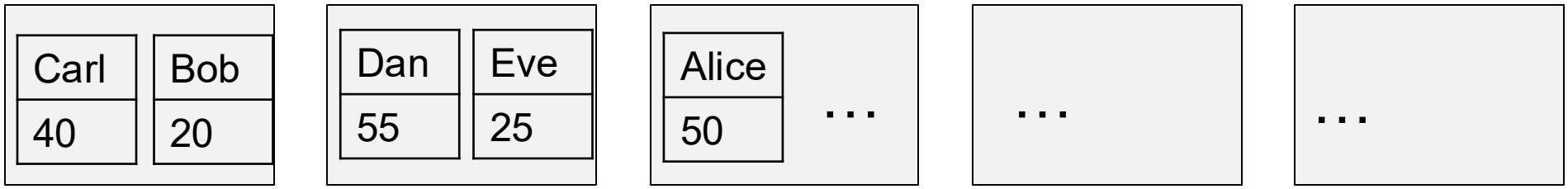


Person data file

Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

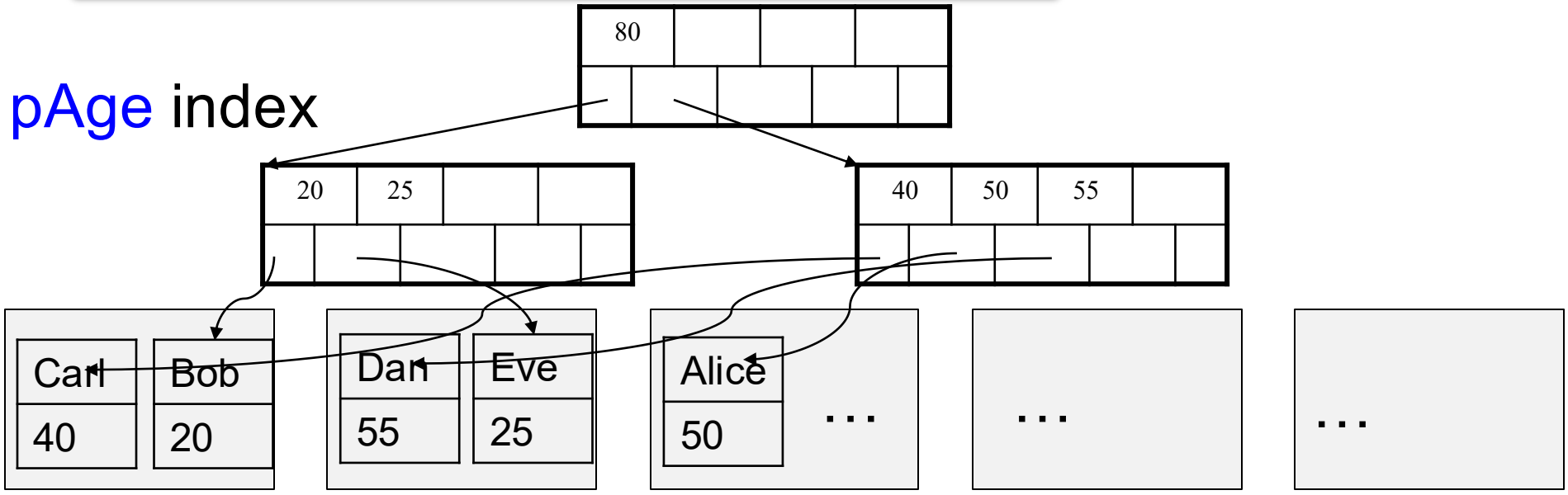


Person data file

Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

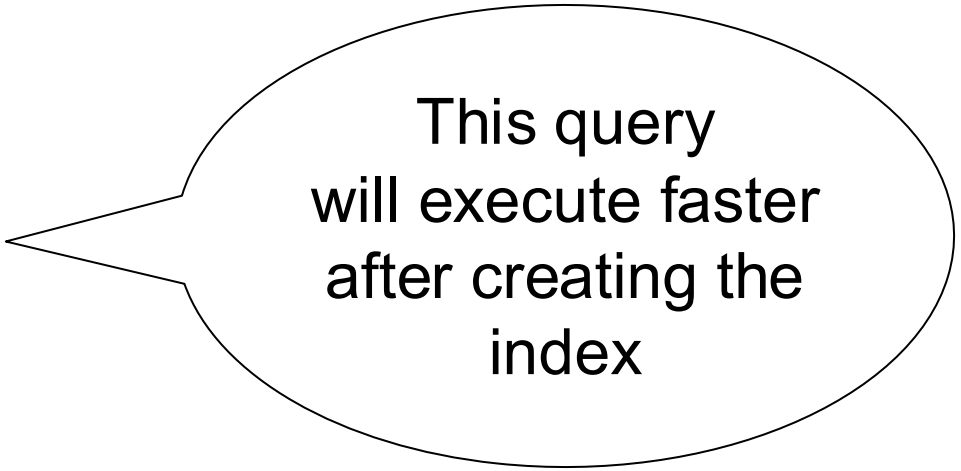


Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

```
select *
from Person
where age = 44
```



Index in SQL

```
create table Person(name text, age int)
```

```
create index pAge on Person(age)
```

```
select *
from Person
where age = 44
```

```
select *
from Person x, Car y
where x.age = y.age
```

A join will not benefit from this index

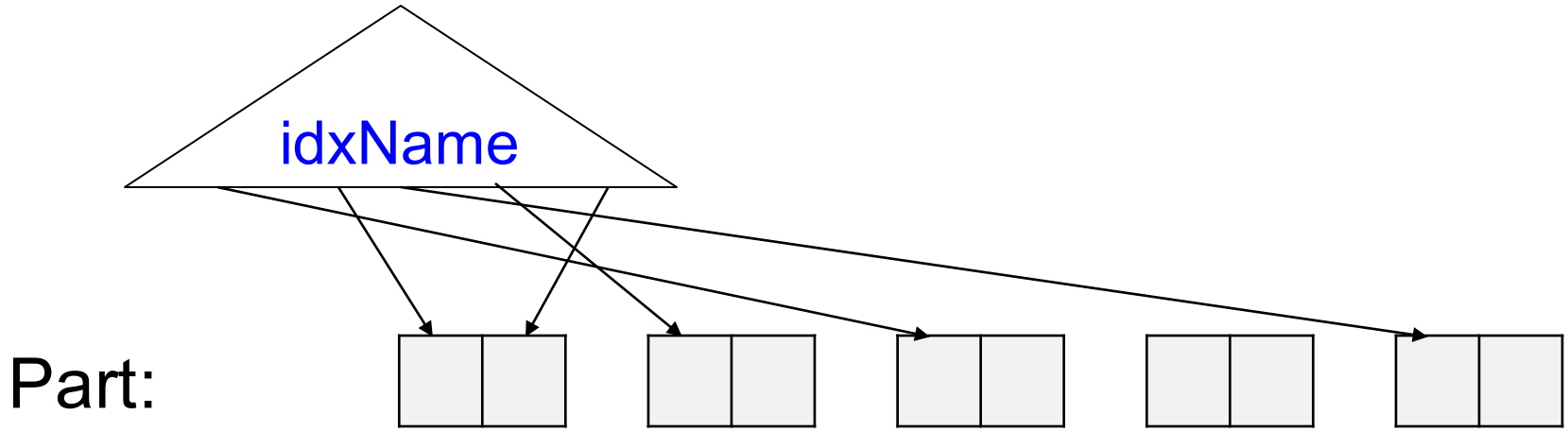
Discussion

- We can create many indexes for a table
- One query can only use one index
- Each update to the table requires updating all indices

Part(pno,pname,psize,pcolor)

Create Index

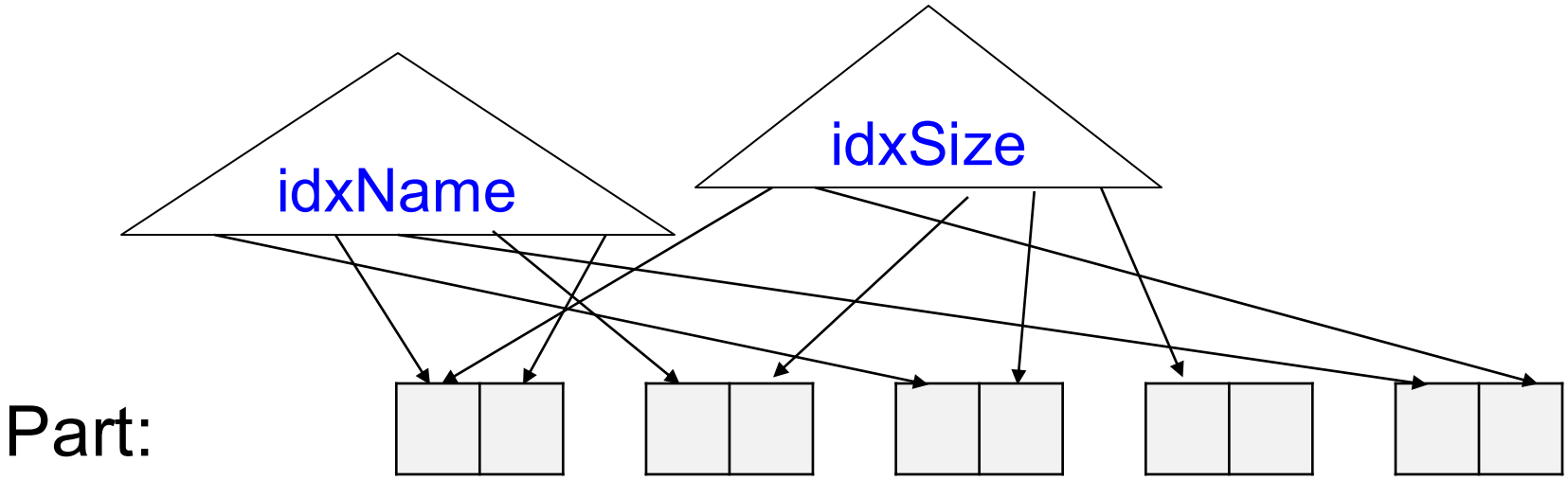
```
create index idxName on Part(pname);
```



Part(pno,pname,psize,pcolor)

Create Index

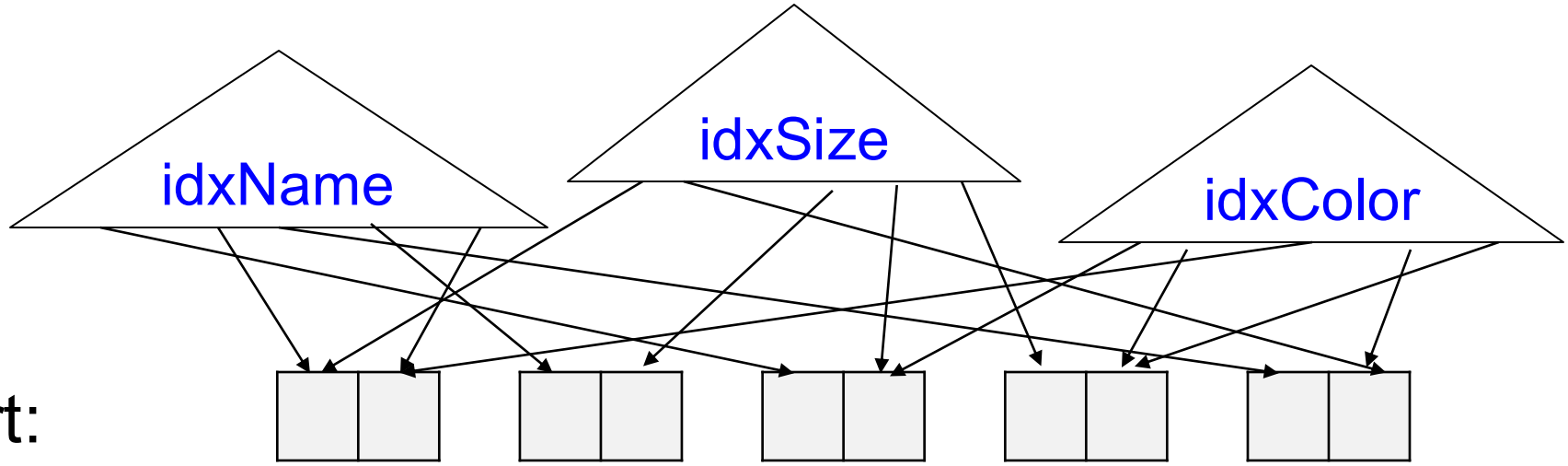
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
```



Part(pno,pname,psize,pcolor)

Create Index

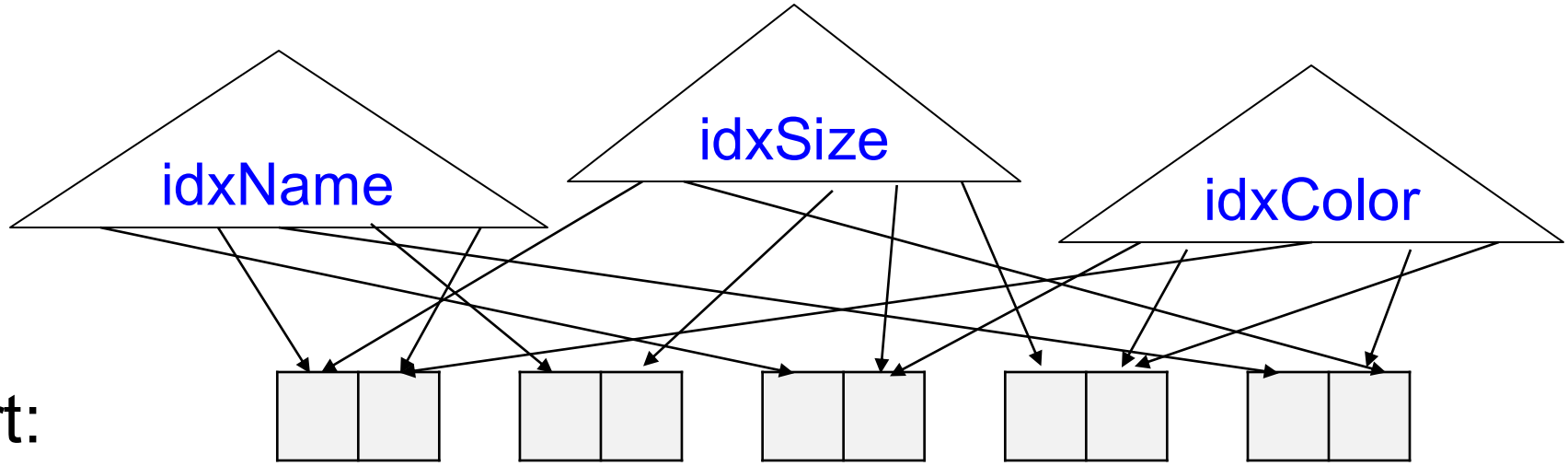
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
create index idxColor on Part(pcolor);
```



Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

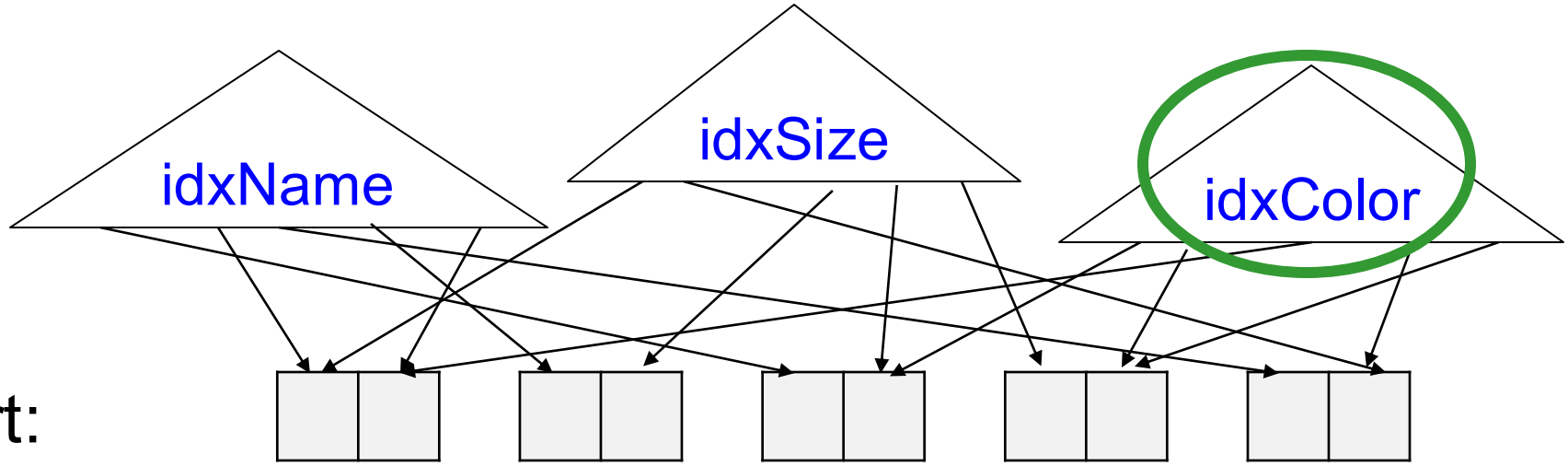


Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

Use idxColor

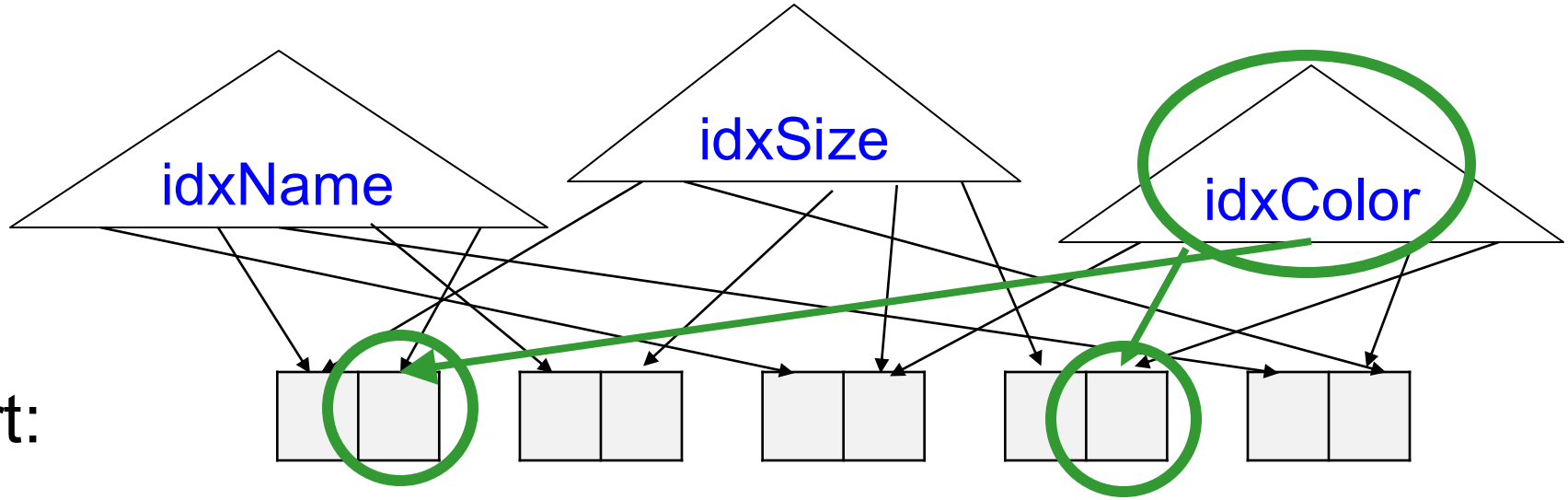


Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

Use idxColor

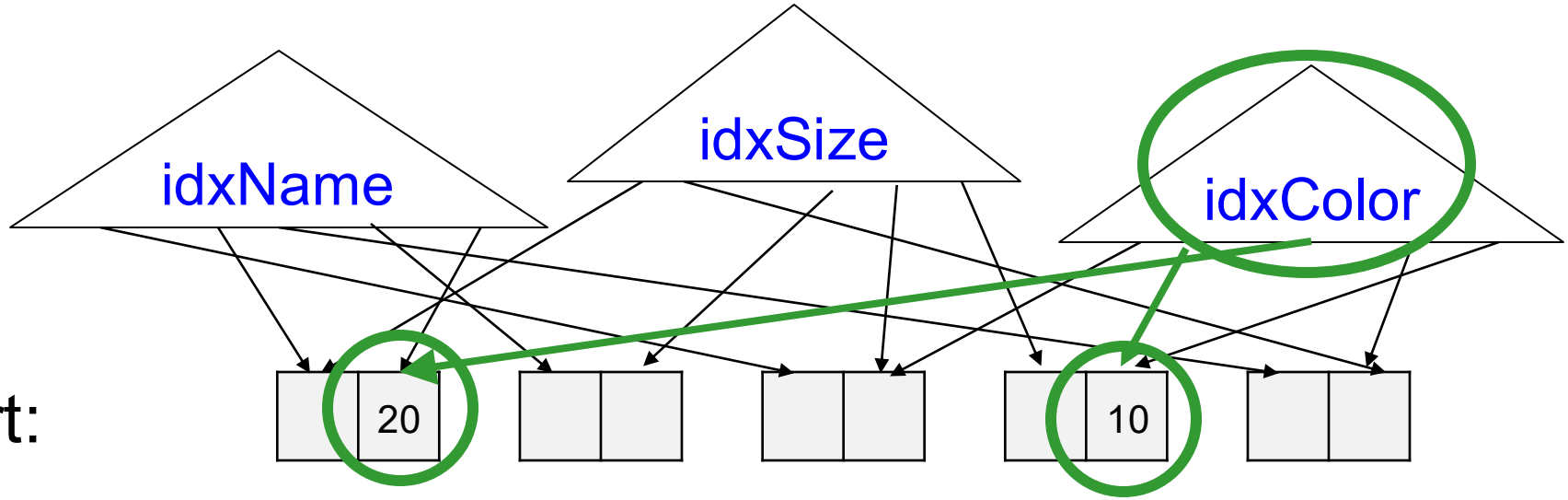


Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

Use idxColor



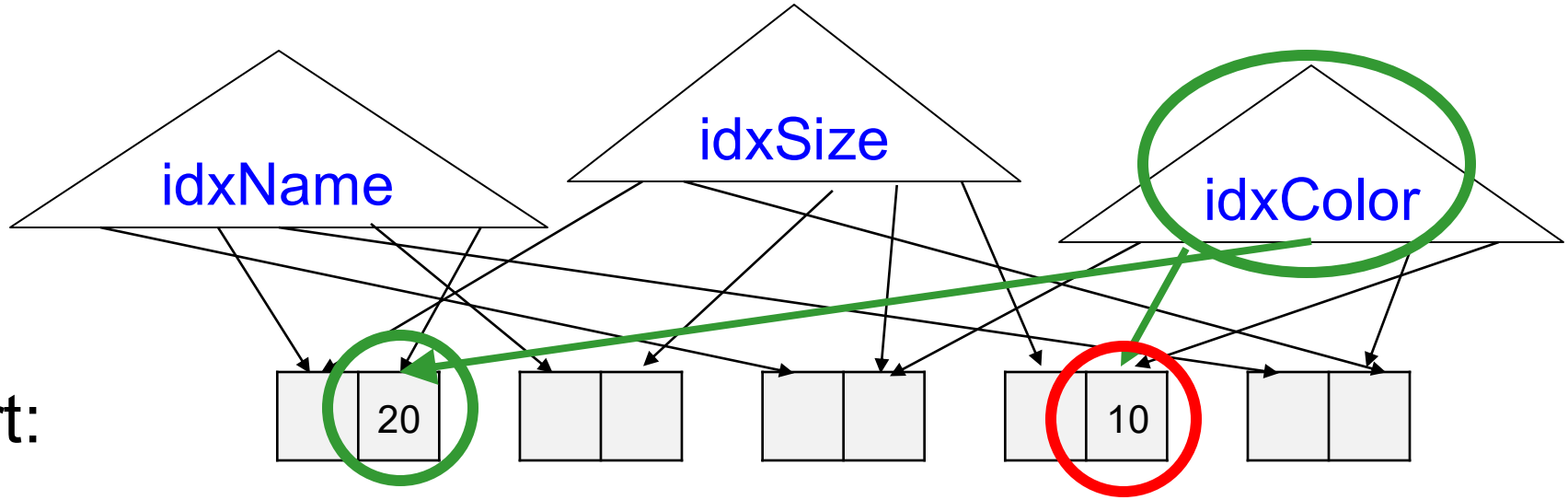
Read all green parts, return those with psize=10

Part(pno,pname,psize,pcolor)

Create Index

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

Use idxColor



Read all green parts, return those with psize=10

Discussion

- In general there can be multiple indexes available to compute a where-condition:
  - Attr1=val1 and Attr2=val2 and …

Discussion

- In general there can be multiple indexes available to compute a where-condition:
  - Attr1=val1 and Attr2=val2 and …
- Optimizer needs to choose one, then iterate over the answers and filter out the other conditions

Discussion

- In general there can be multiple indexes available to compute a where-condition:
  - Attr1=val1 and Attr2=val2 and …
- Optimizer needs to choose one, then iterate over the answers and filter out the other conditions
- Problem is called access path selection

Multi-attribute Index

- We can create an index on multiple attributes: A, B, C, …
- Values are concatenated, then sorted
- Attribute order matters:
  - Create index on R(A,B,C)
  - Create index on R(C,A,B)

Part(pno,pname,psize,pcolor)

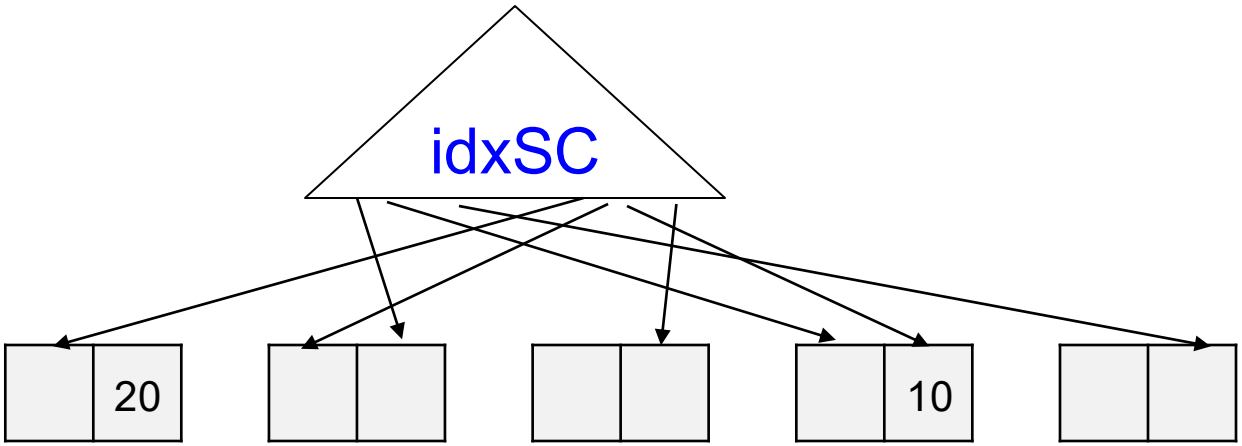
Multi-attribute Index

```
create index idxSC on Part(psize, pcolor);
```

Part(pno,pname,psize,pcolor)

Multi-attribute Index

```
create index idxSC on Part(psize, pcolor);
```

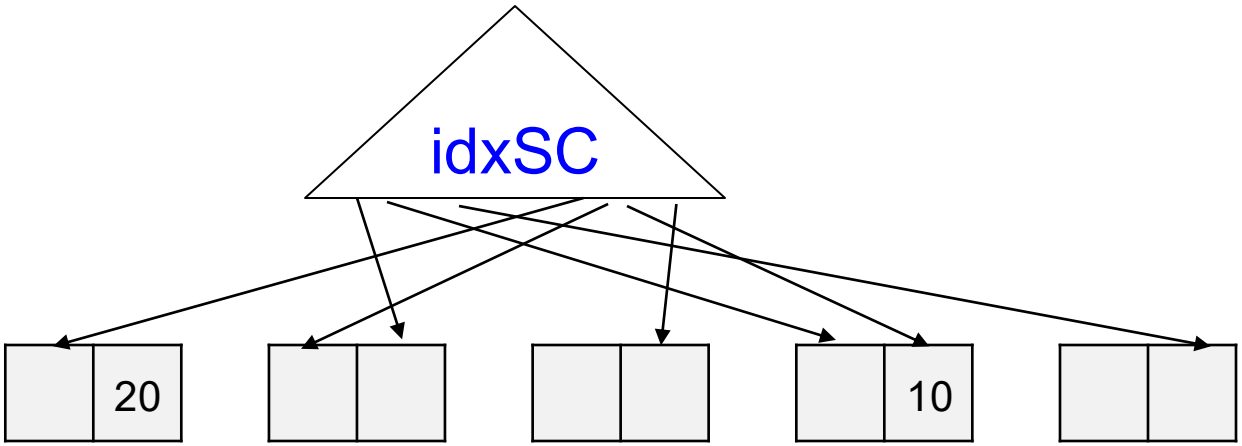


Part(pno,pname,psize,pcolor)

Multi-attribute Index

```
create index idxSC on Part(psize, pcolor);
```

```
select *
from Part
where psize = 10 and pcolor = ‘green’
```

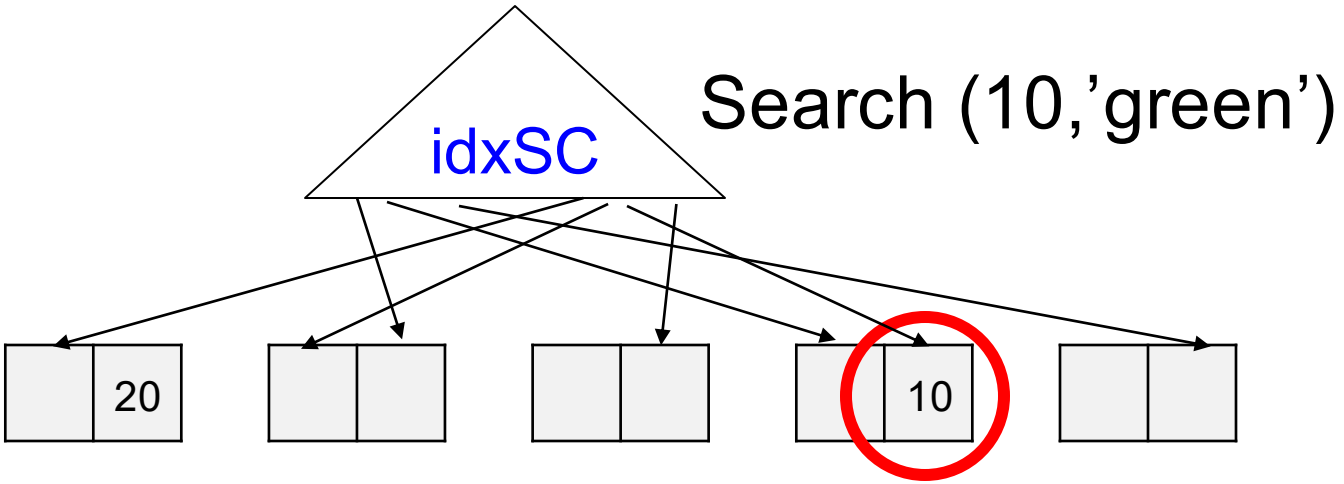


Part(pno,pname,psize,pcolor)

Multi-attribute Index

```
create index idxSC on Part(psize, pcolor);
```

```
select *
from Part
where psize = 10 and pcolor = 'green'
```

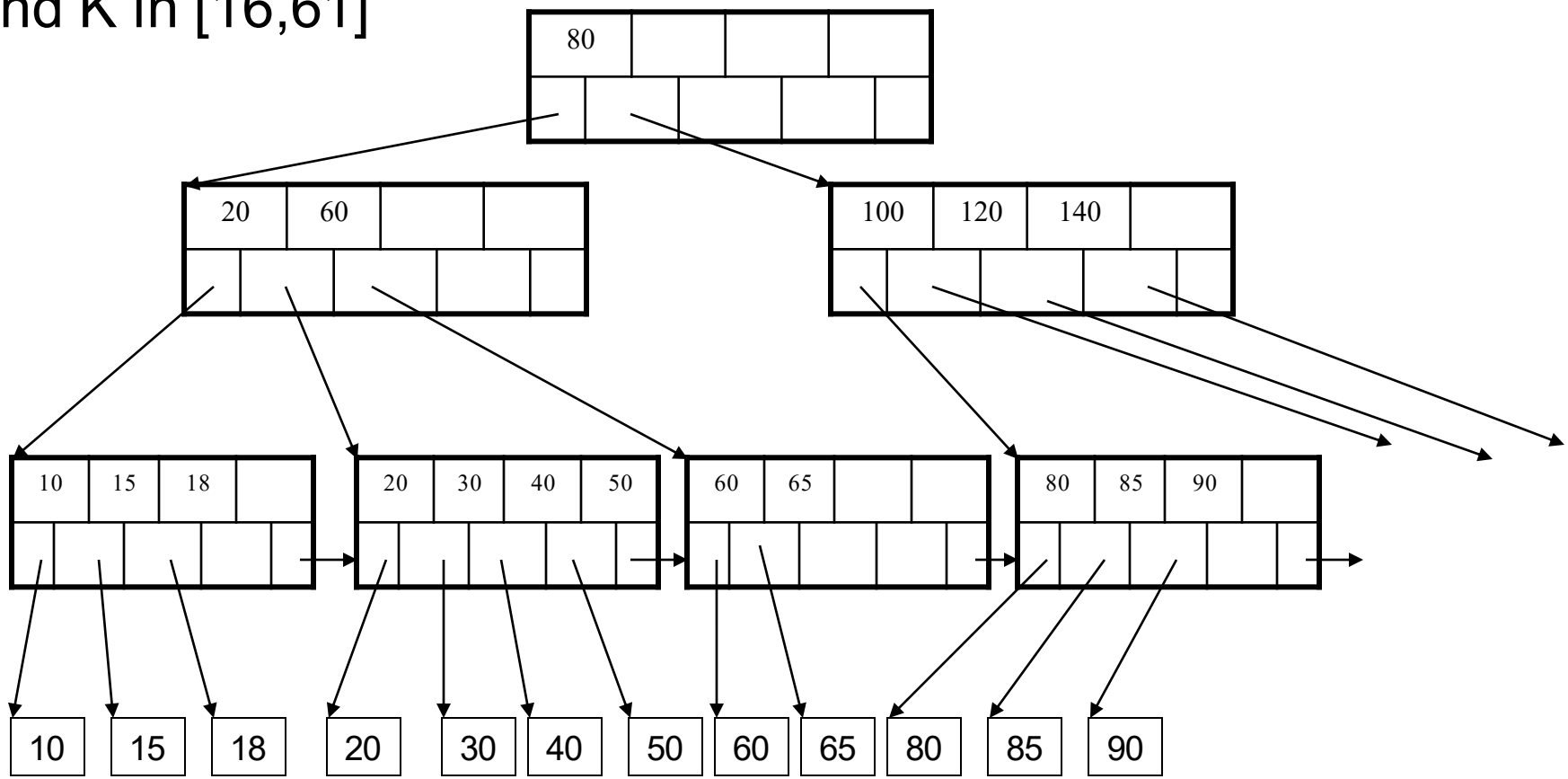


Multi-attribute Index

- A pair ordered lexicographically:
  - (Smith,Alice) < (Smith,Bob)<(Yu,Alice)
- Only 1st attribute known: range seach
  - Find (Smith, *)
- Only 2nd attribute known: impossible
  - Find (*, Alice)

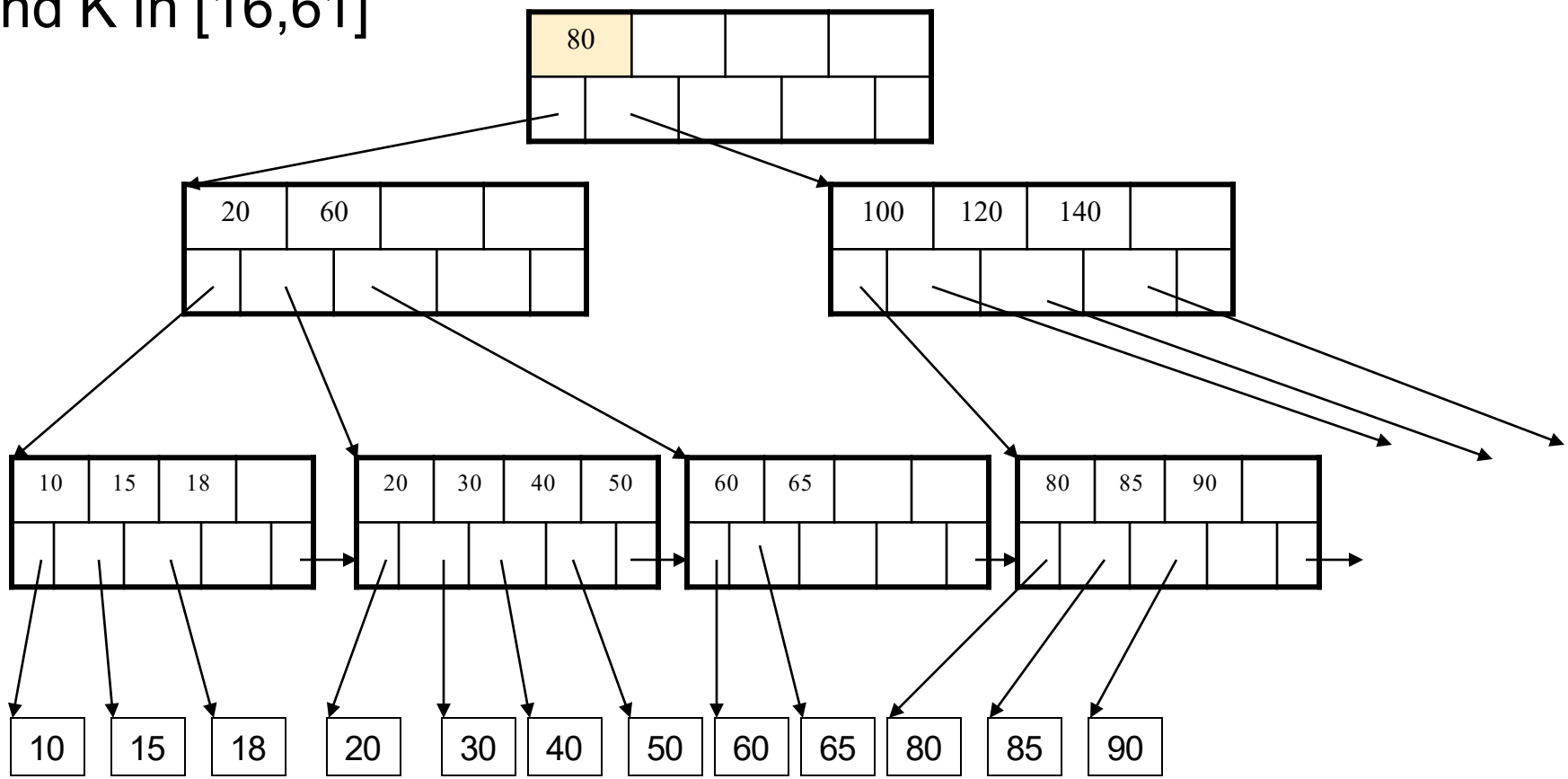
Range Search in B+ Tree

Find K in [16,61]



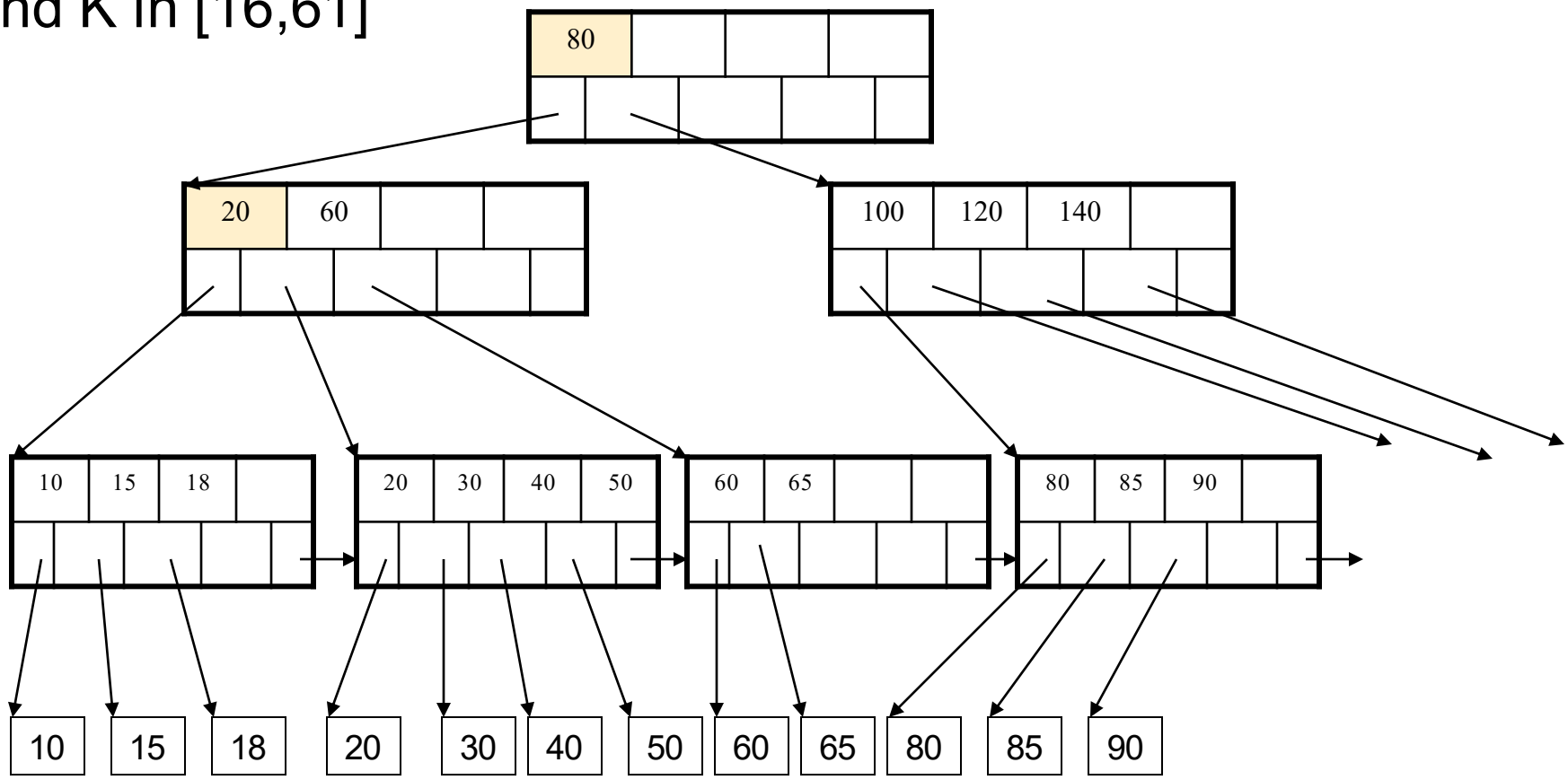
Range Search in B+ Tree

Find K in [16,61]



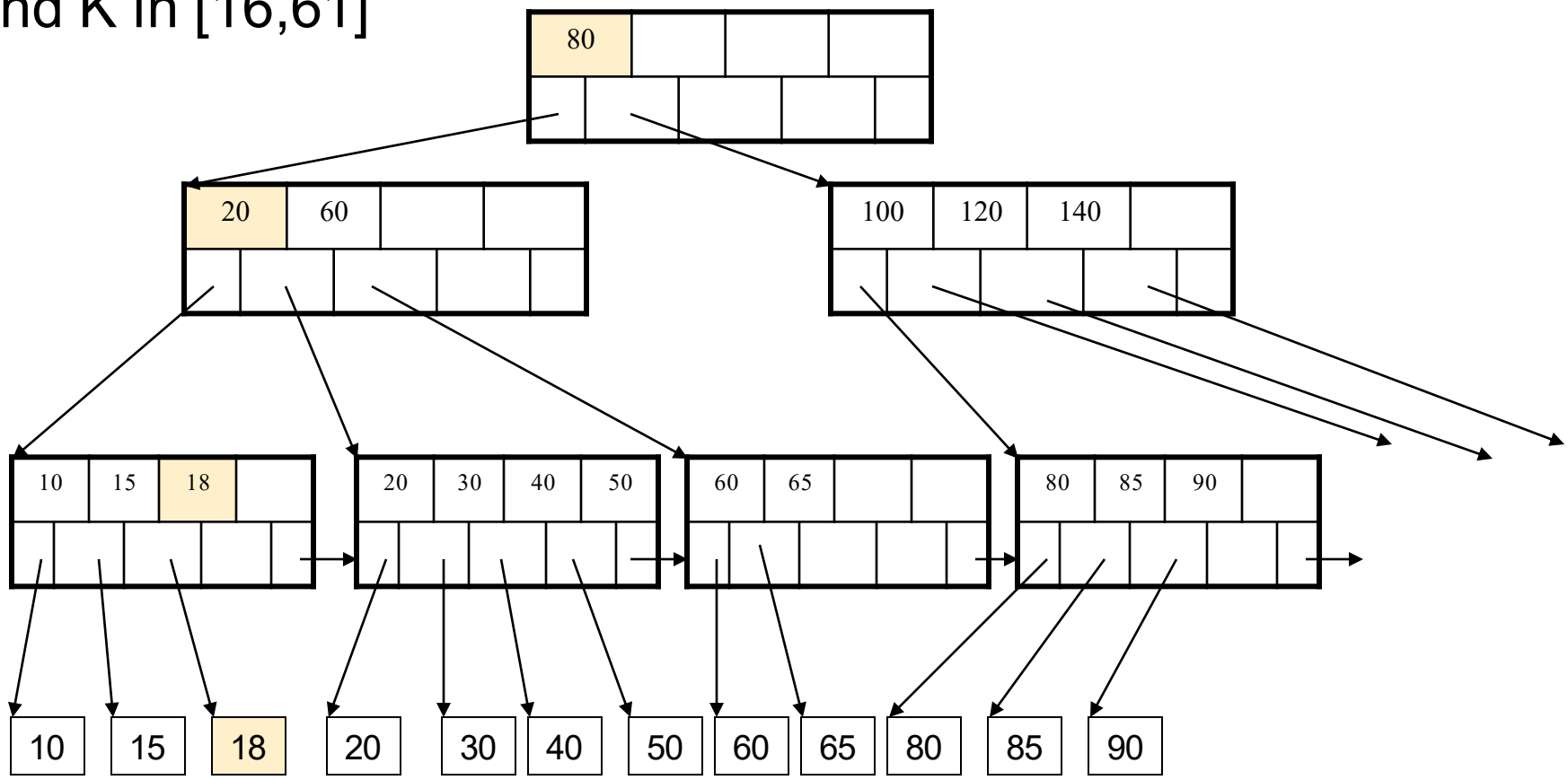
Range Search in B+ Tree

Find K in [16,61]



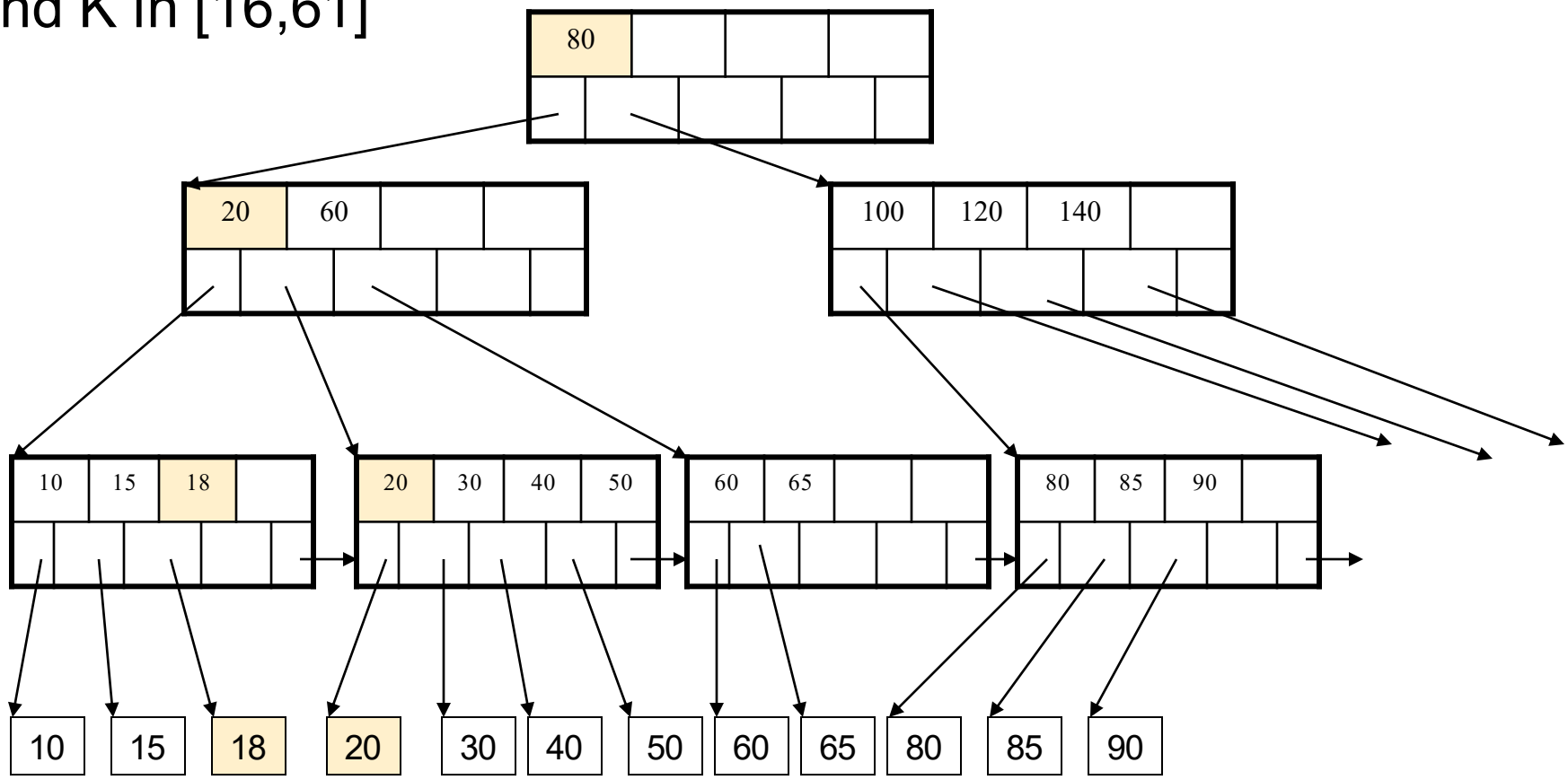
Range Search in B+ Tree

Find K in [16,61]



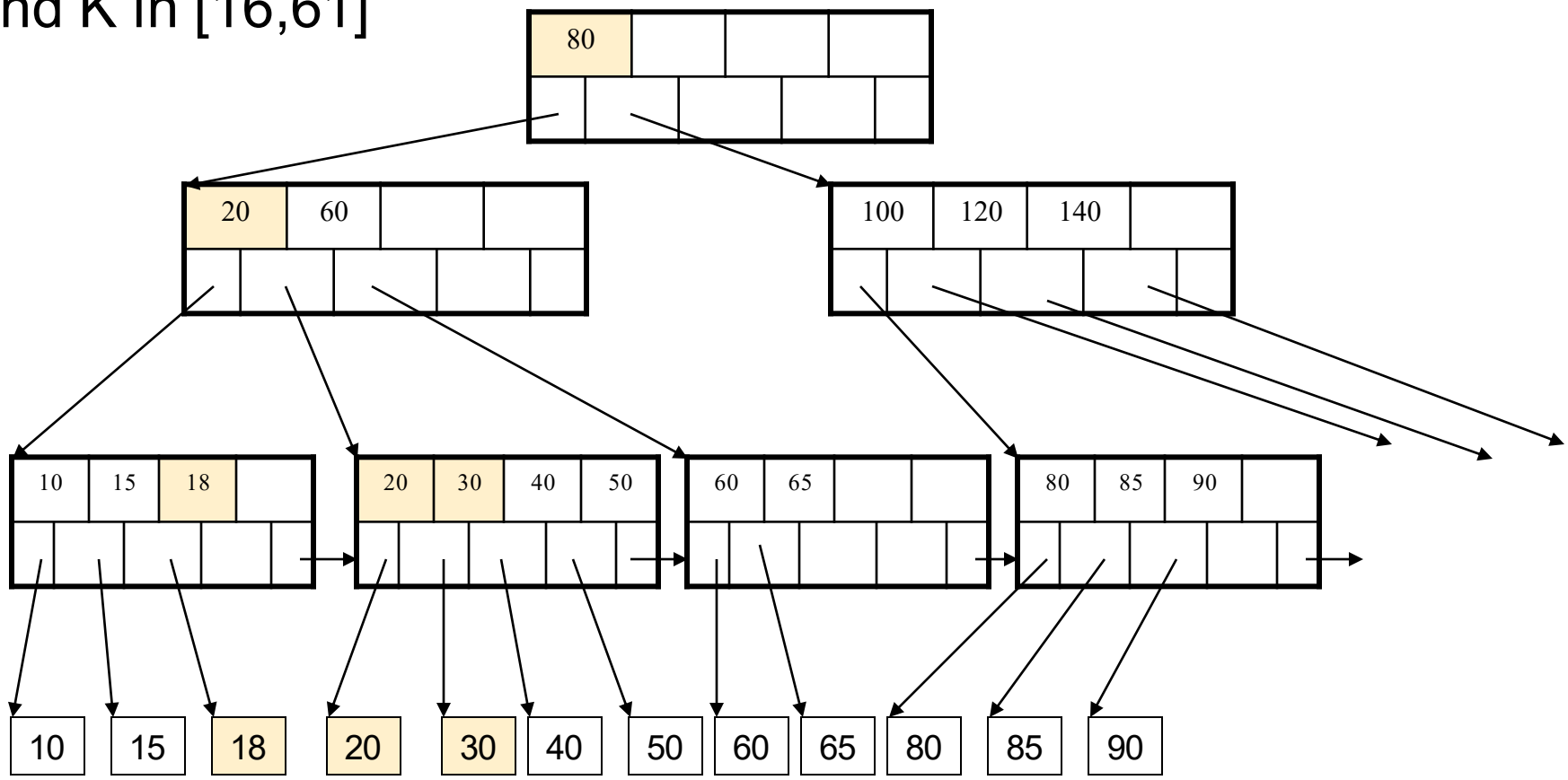
Range Search in B+ Tree

Find K in [16,61]



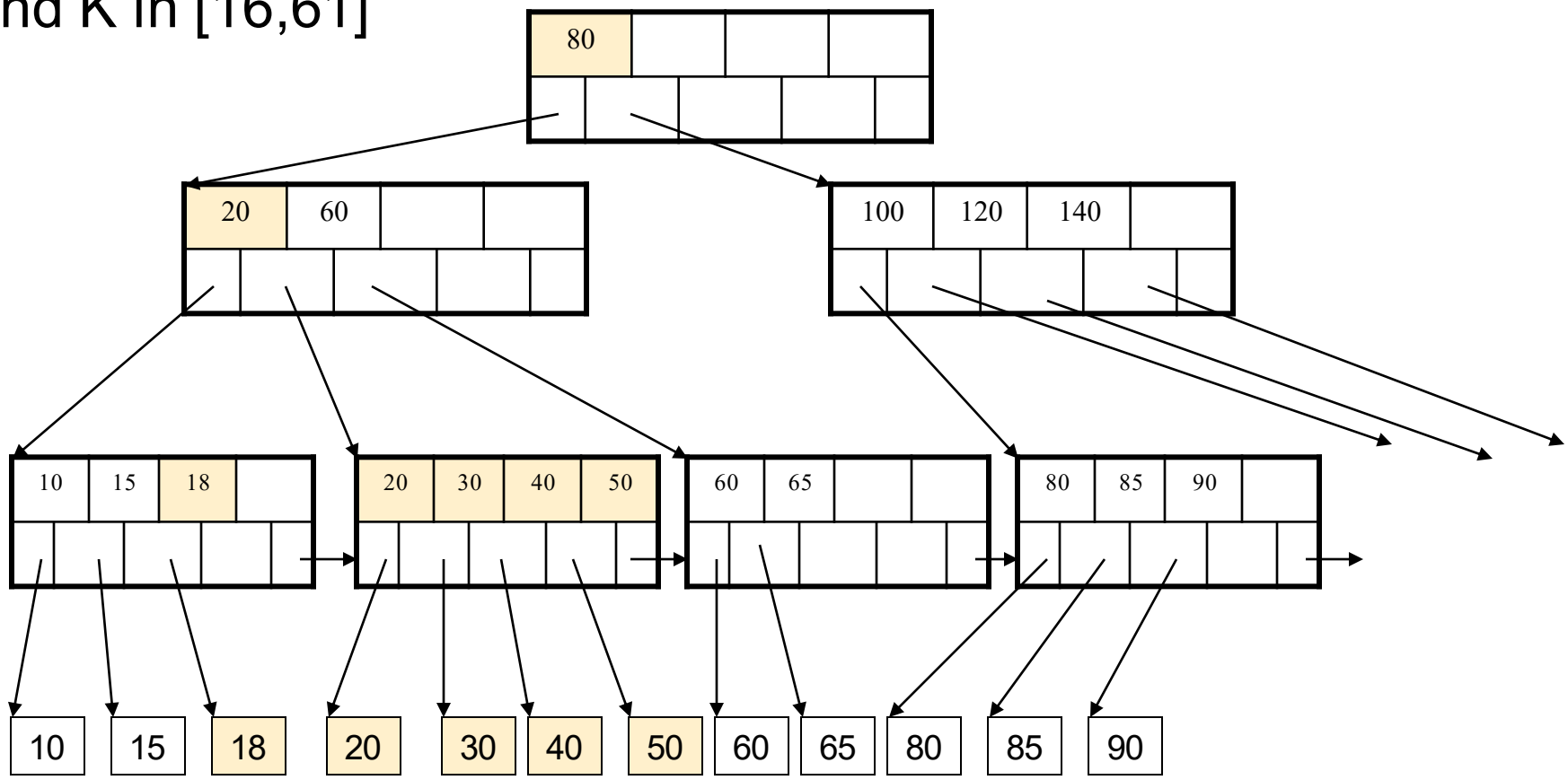
Range Search in B+ Tree

Find K in [16,61]



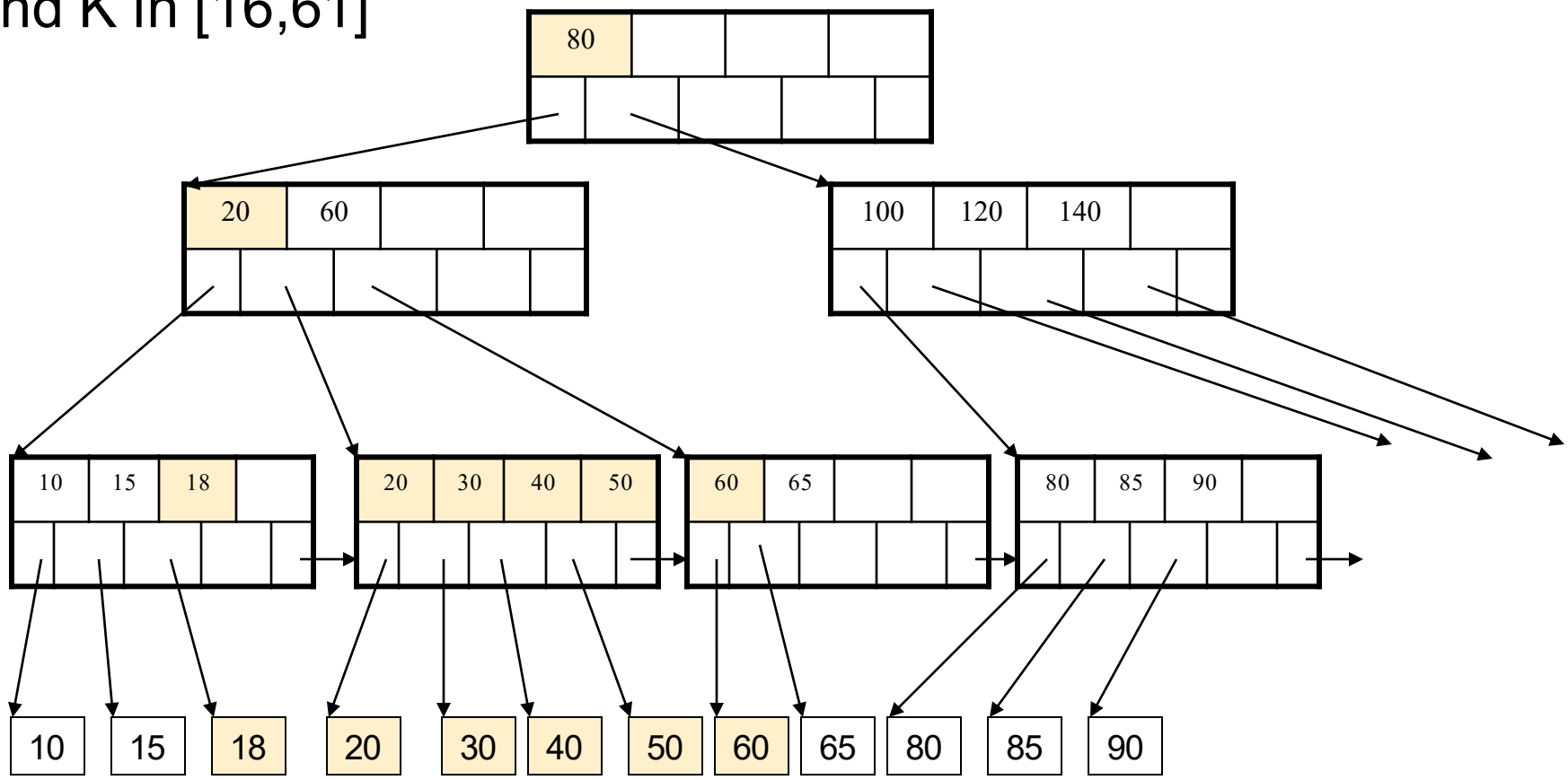
Range Search in B+ Tree

Find K in [16,61]



Range Search in B+ Tree

Find K in [16,61]



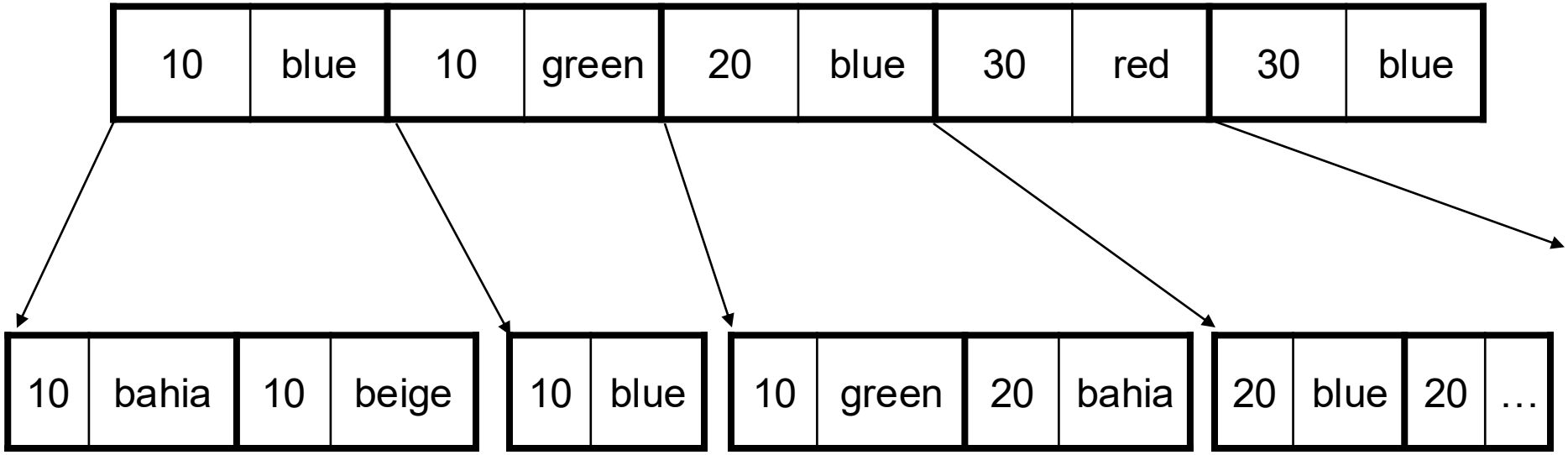
Multi-Attribute Index

- If we only know a prefix of a multi-attribute index, then we can use a range-search

- If we know only a subset that is not a prefix, then we cannot use the index

```
create index idxSC on Part(psize, pcolor);
```

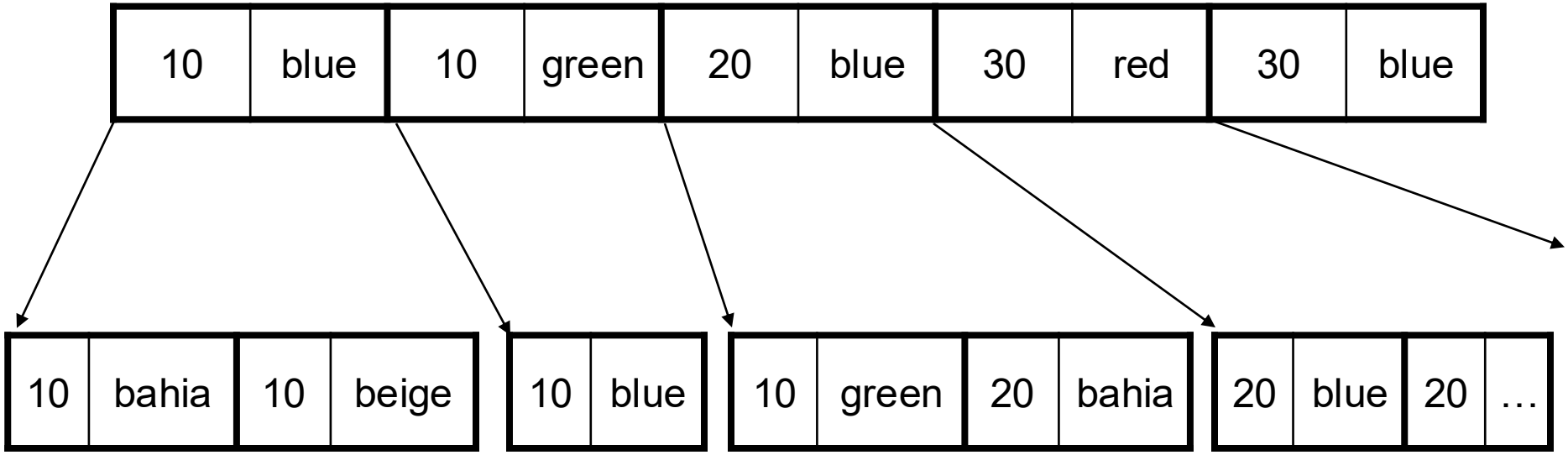
Multi-Attribute Index



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

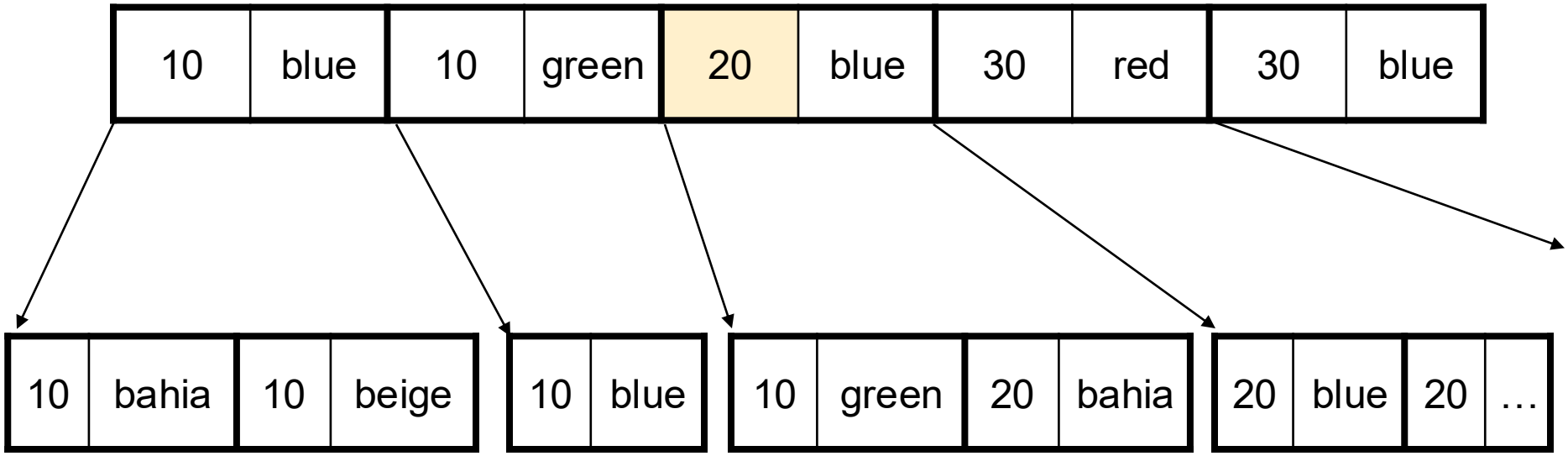
psize = 20, pcolor = *



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

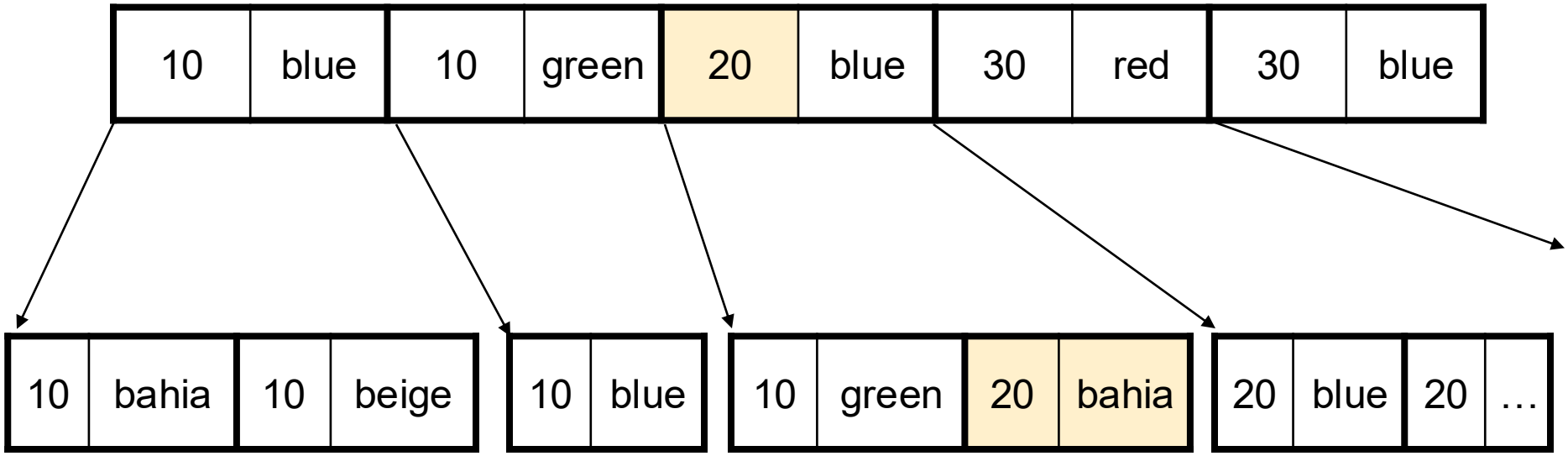
psize = 20, pcolor = *



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

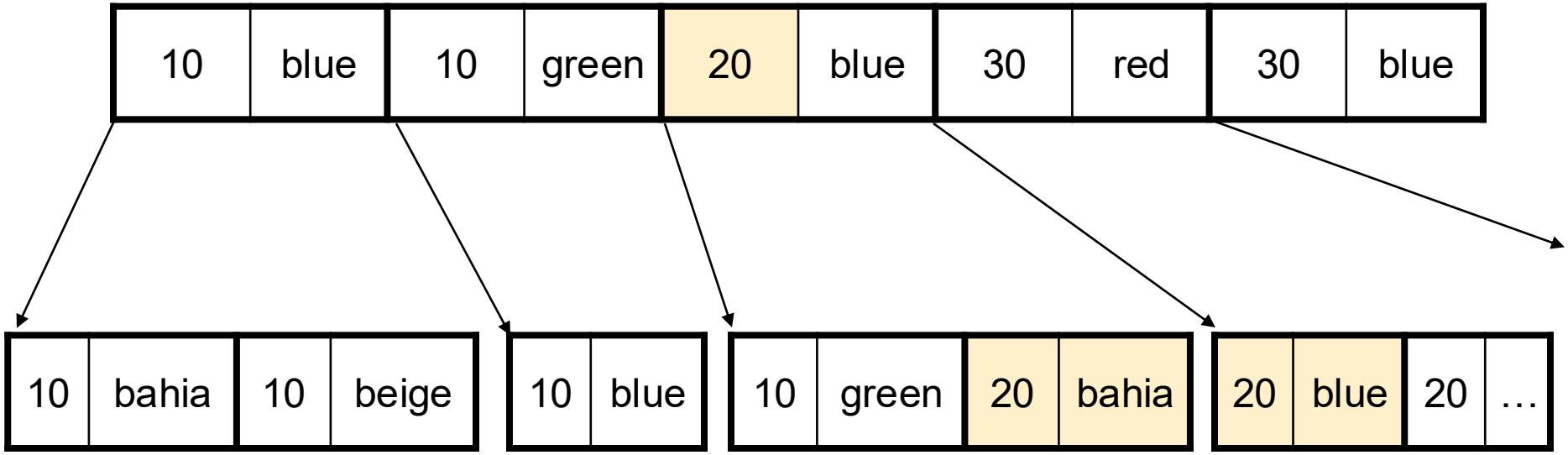
psize = 20, pcolor = *



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

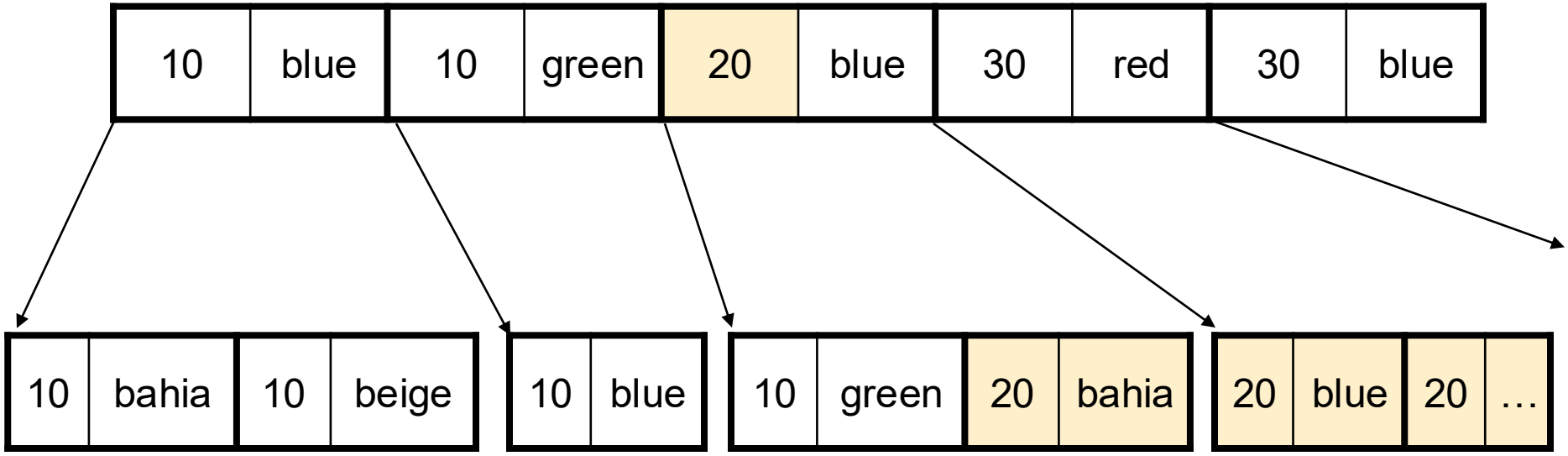
psize = 20, pcolor = *



```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

psize = 20, pcolor = *



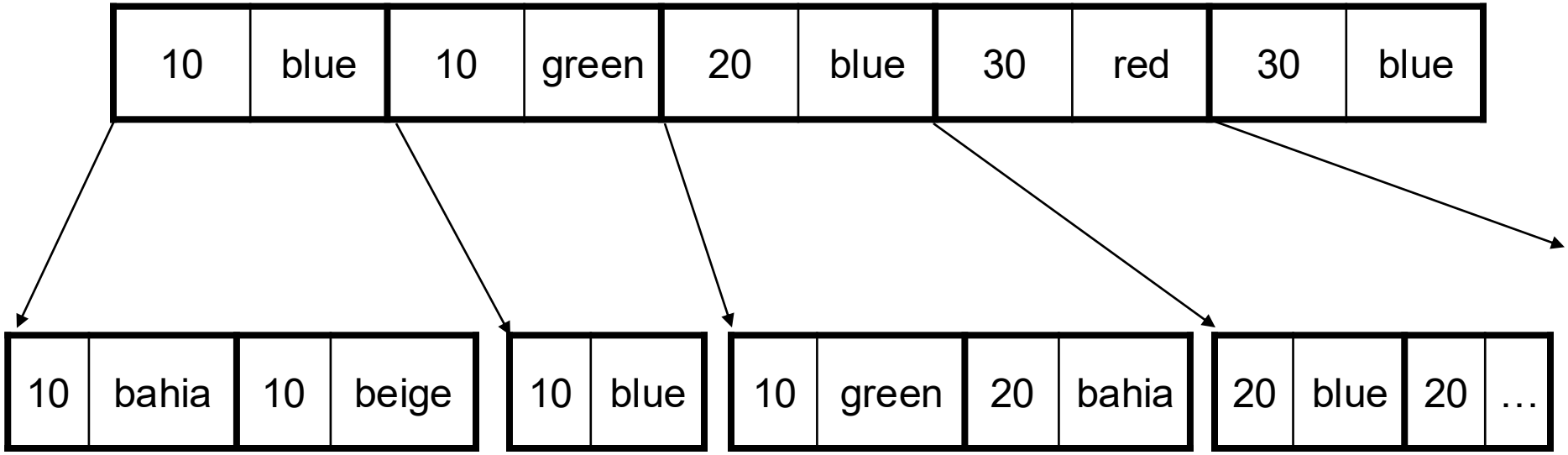
Index is useful

```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue

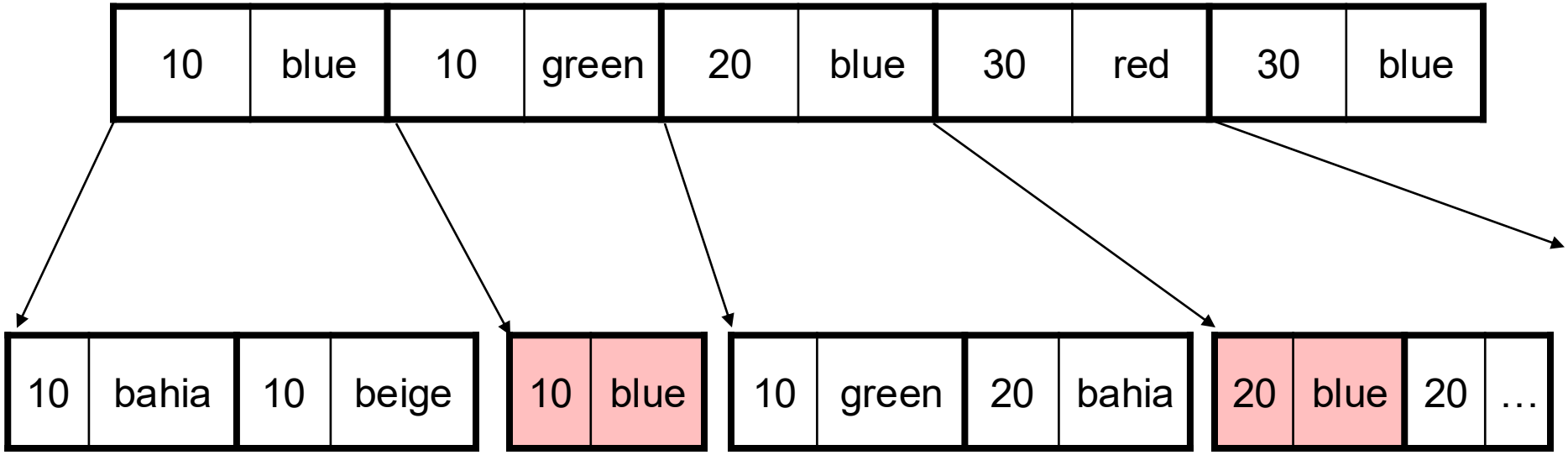


```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue

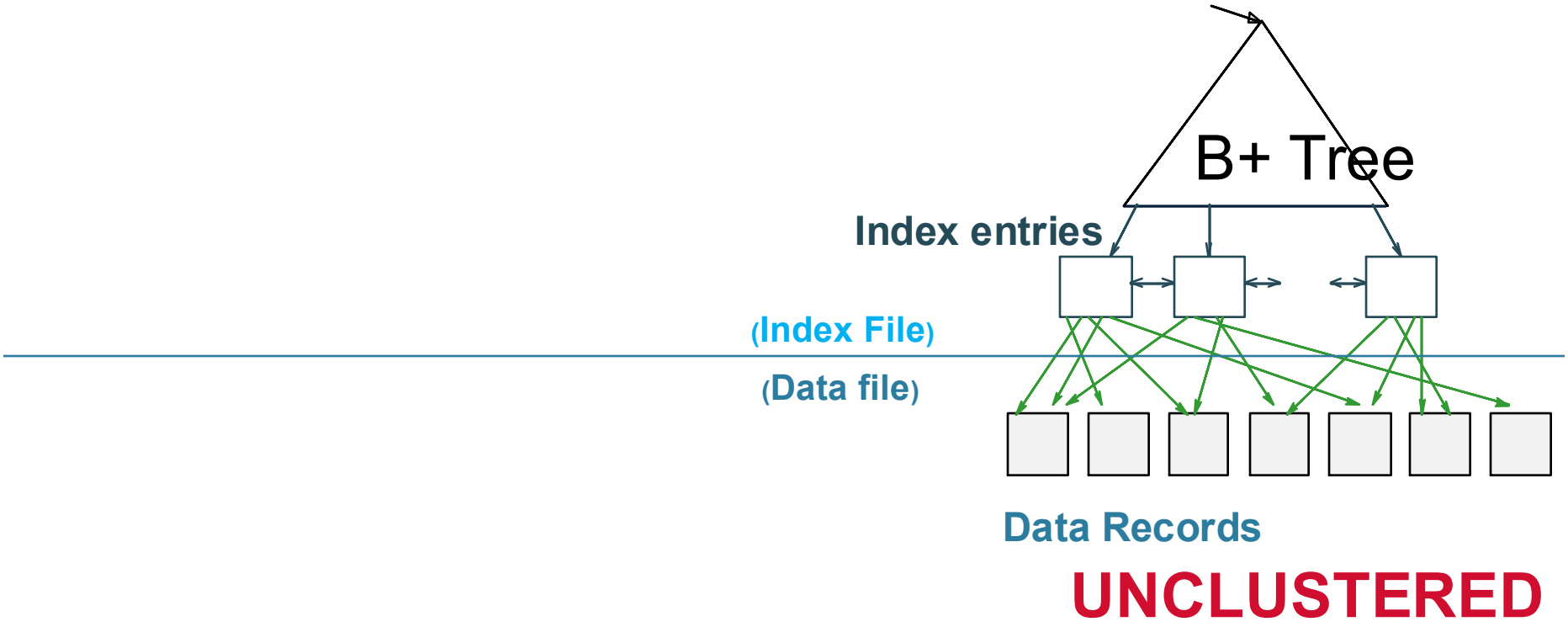


Index is not useful

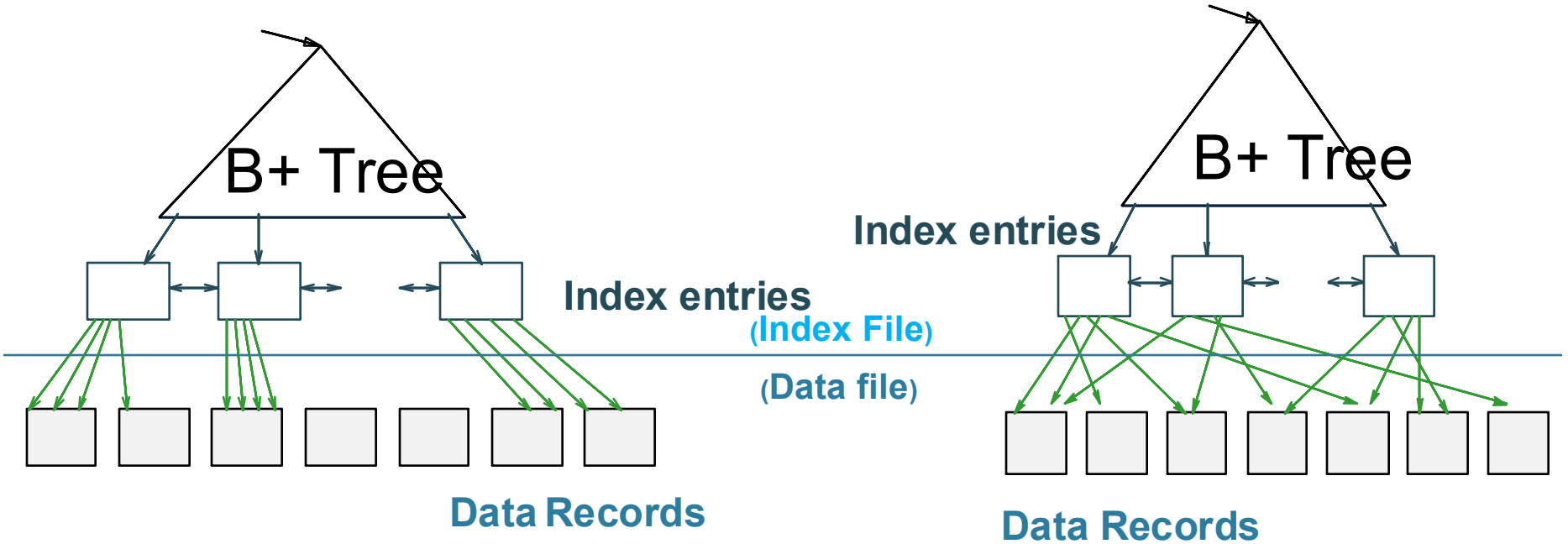
Clustered/Unclustered Index

- Unclustered index:
  - The index is separate from the data file
  - Leave nodes contain (Key,RID) pairs
  - Separate read for data record is required
- Clustered index:
  - The index is the same as the data file
  - Leave nodes contain the actual records

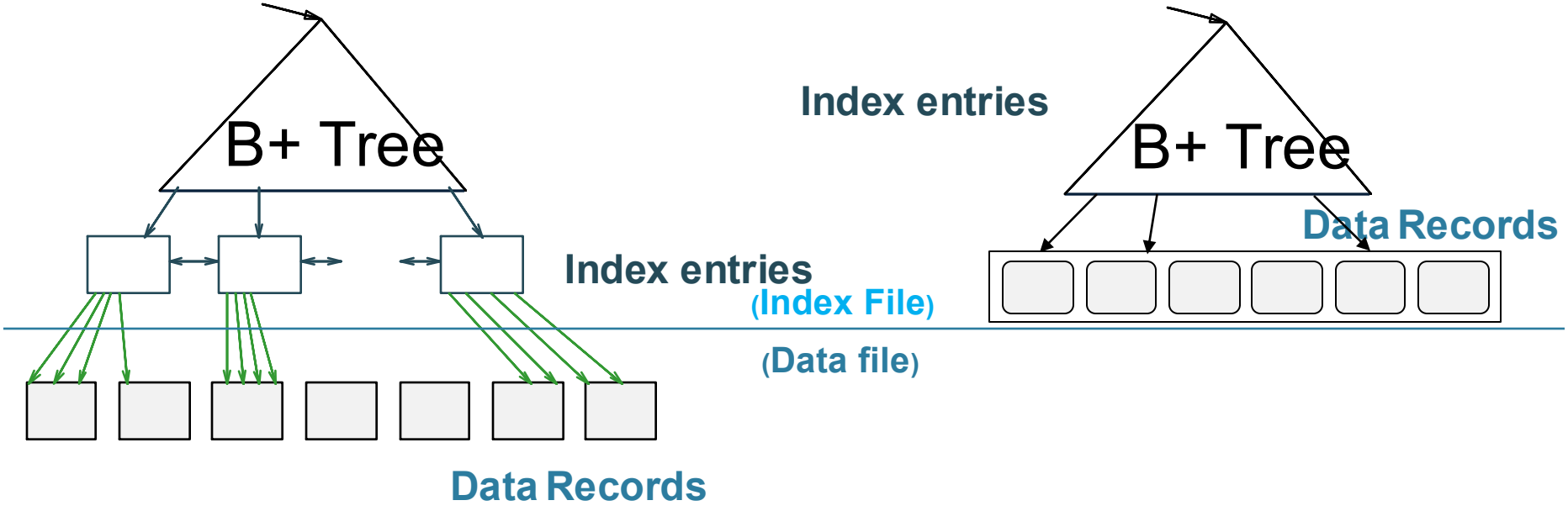
Clustered vs Unclustered



Clustered vs Unclustered



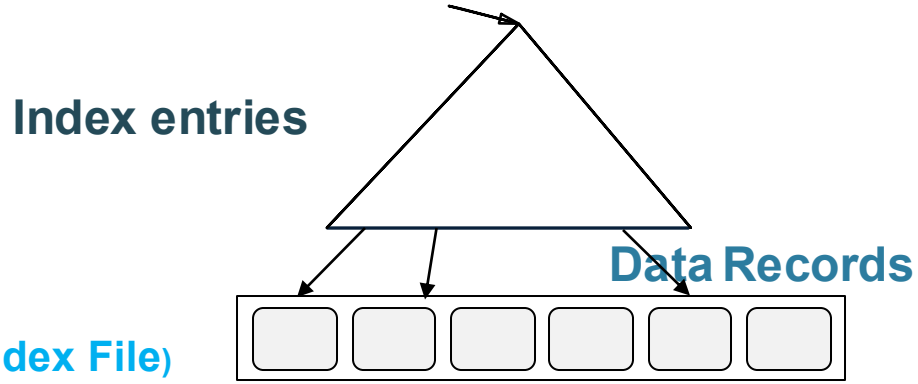
Clustered



CLUSTERED

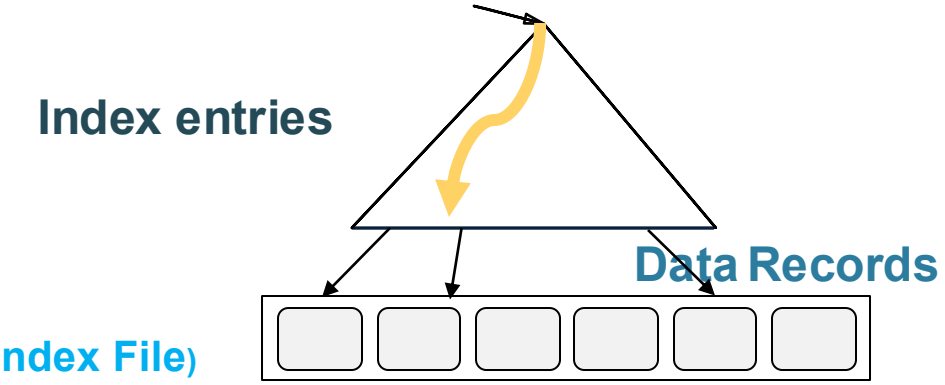
BETTER

Clustered



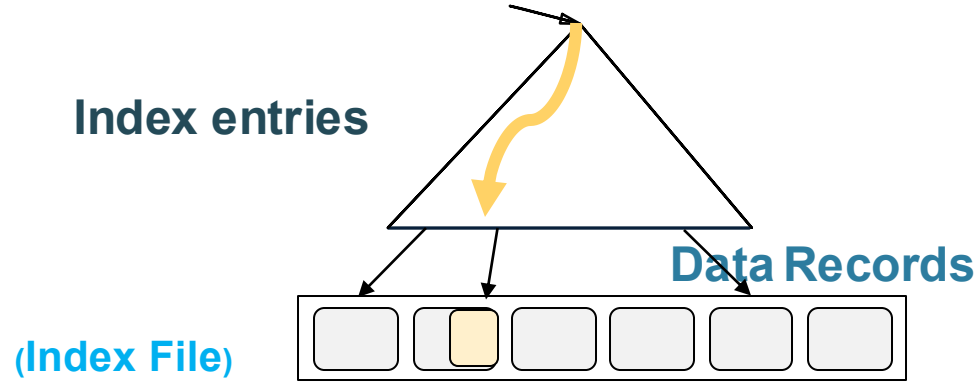
RANGE SEARCH

Clustered



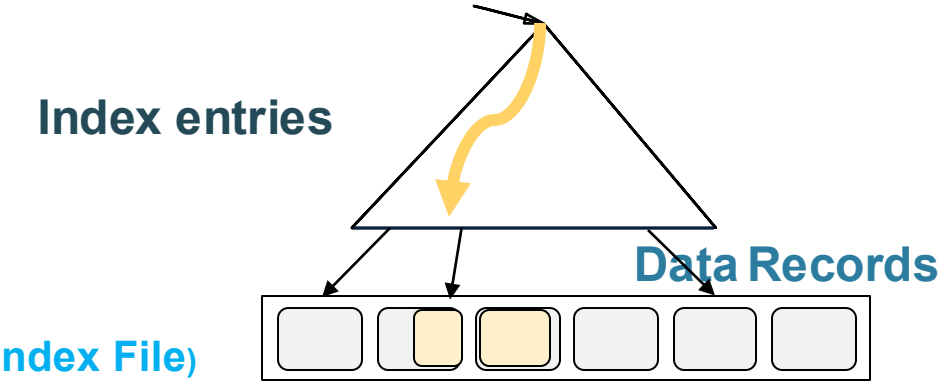
RANGE SEARCH

Clustered



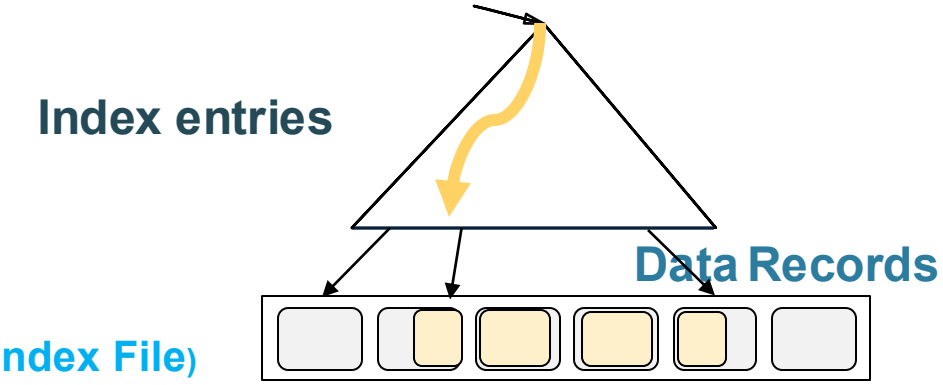
RANGE SEARCH

Clustered



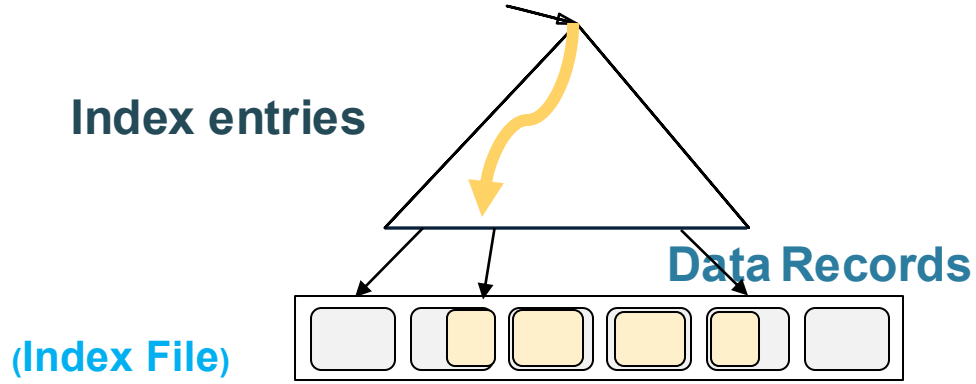
RANGE SEARCH

Clustered



RANGE SEARCH

Clustered



RANGE SEARCH

Sequential read

Part(pno,pname,psize,pcolor)

Clustered Index in Postgres

```
create index idxName on Part(pname);
create index idxSize on Part(psize);
create index idxColor on Part(pcolor);
```

```
cluster Part using idxName;
```

This sorts Part
(takes a while)
and converts it into
the leaves of
idxName

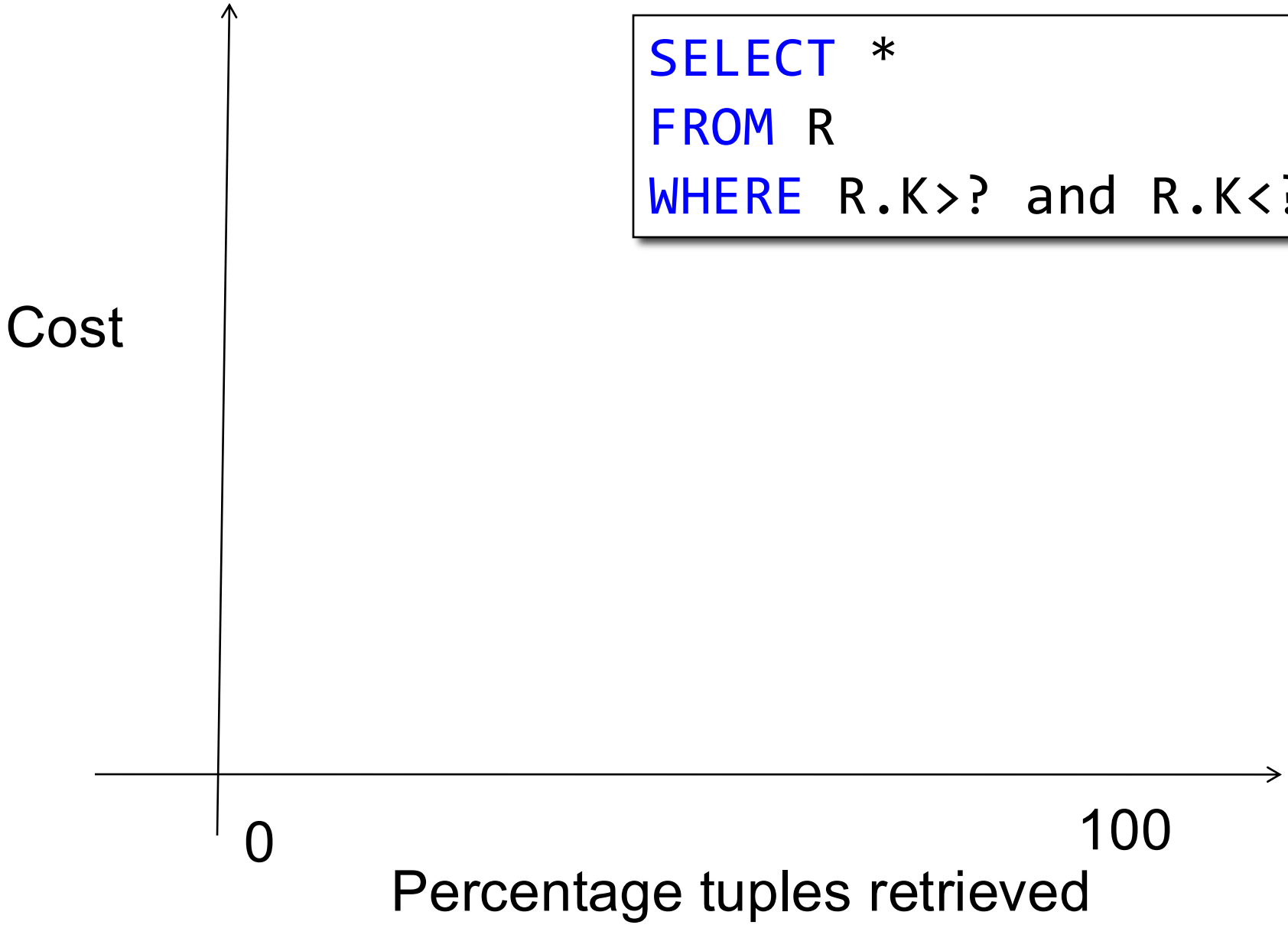
To Cluster or Not

Remember:

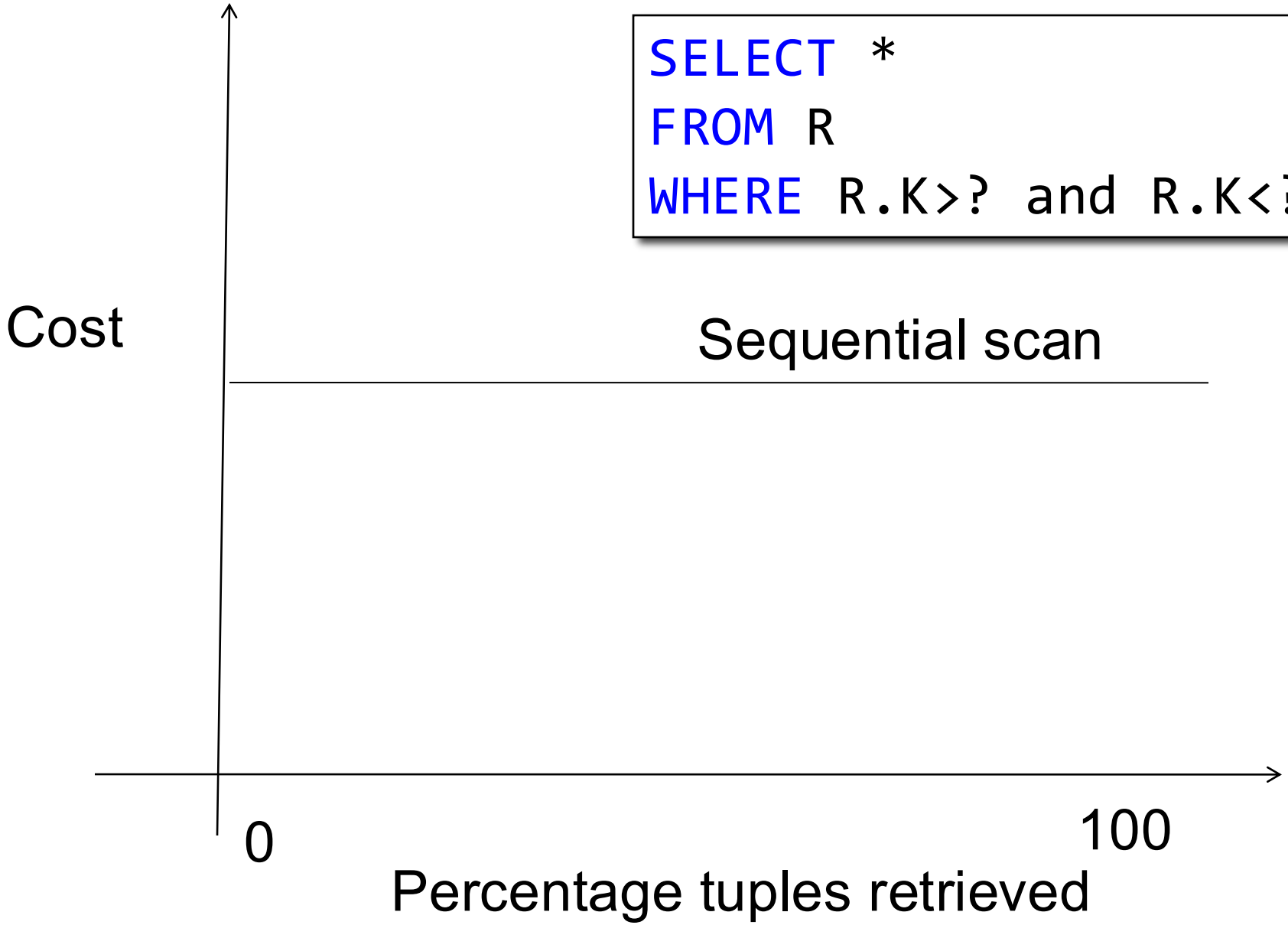
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

Range queries benefit mostly from clustering because they may read more than 1-2%

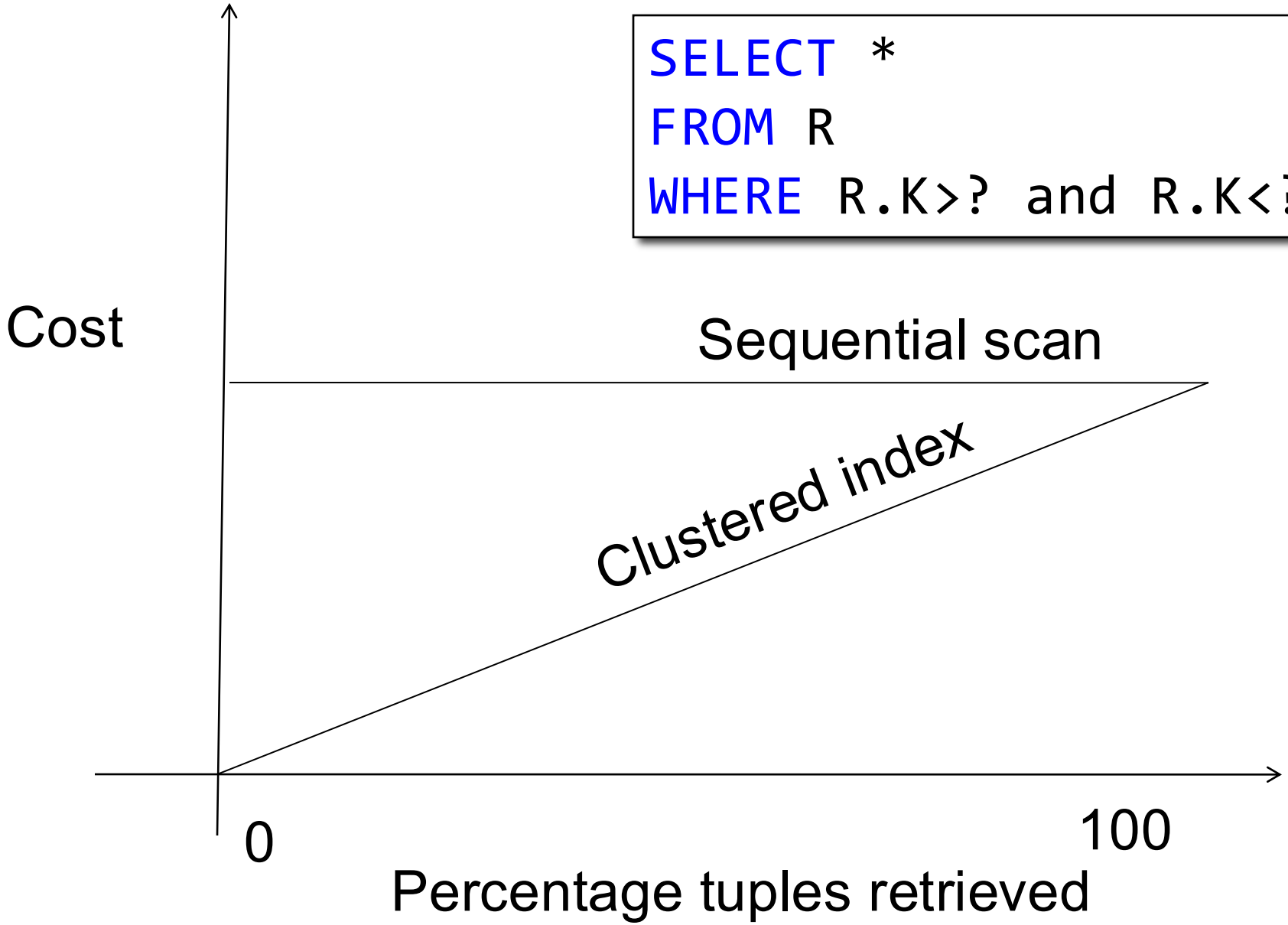
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```

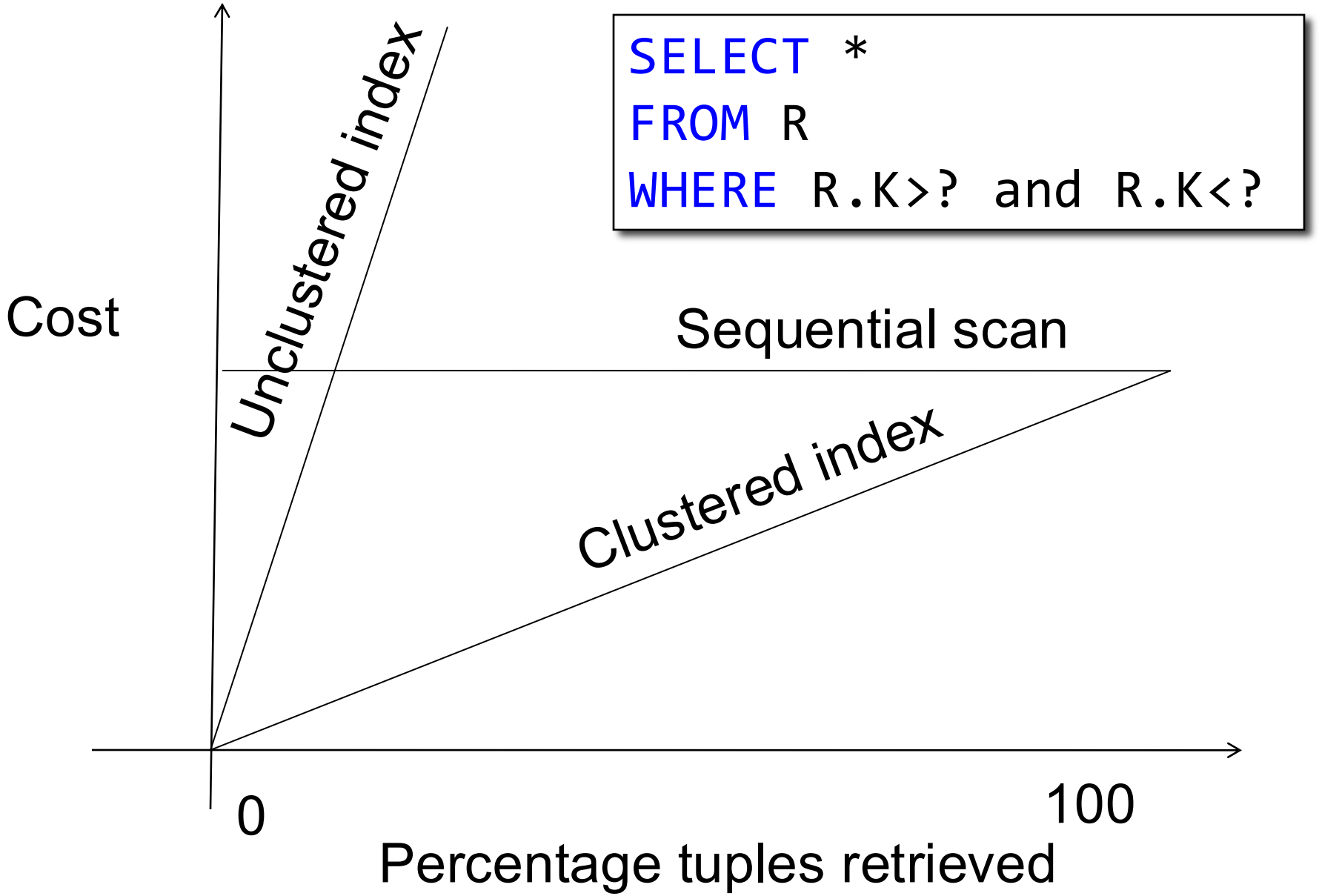


```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```




SELECT *
FROM R
WHERE R.K>? and R.K<?
Cost
Unclustered index
Sequential scan
Clustered index
0
100
Percentage tuples retrieved

Final Discussion

- Indexes:
  - Speedup predicates: A=val, A<val
  - Don't speedup joins (maybe if clustered)
  - Slow down updates
- Bulk index construction:
  - More efficient than inserting one by one
  - Lesson: create the index after data import