CSE544
Data Management

Lectures 4: Storage + Indexes

Announcements

- HW1 was due on Sunday
  - At most 3 late days allowed, i.e. today
- HW2 is due next Sunday
  - How is it going?
- Project proposals due following Sunday

Project

- February 9: Proposal
- March 2: Milestone
- March 11, 12: one-on-one meetings
  - No class on March 12
- March 14: Presentations (2 sessions)
- March 19: Final report due

Where We Are

Discussed so far:

- The relational data model
- Database Design
- SQL

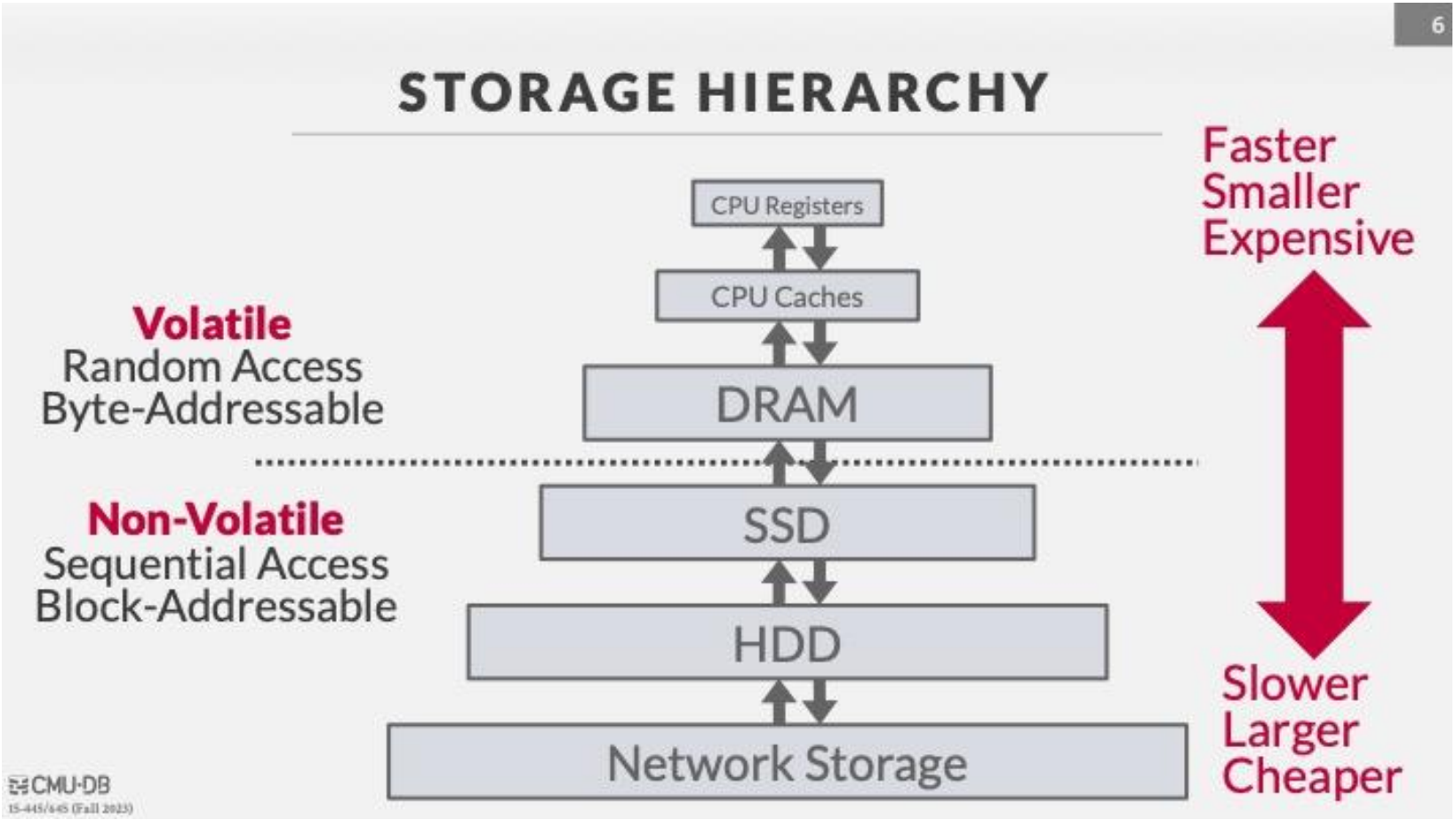
Next:

- Internals of the database engine

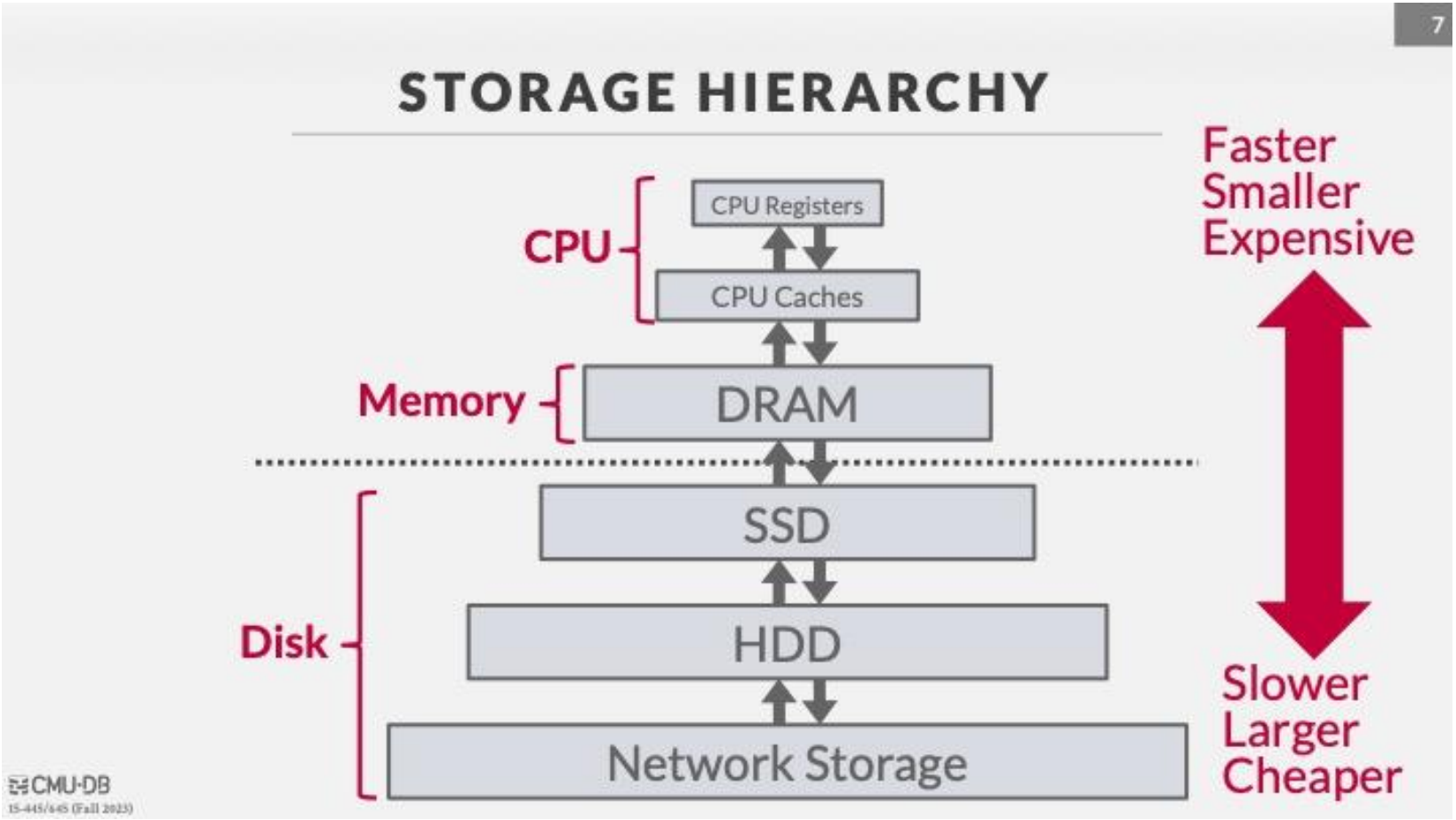
Database Management System

- Stores data persistently (on disk)
- Inserts/updates/deletes data
- Optimizes, executes queries
- Runs transactions

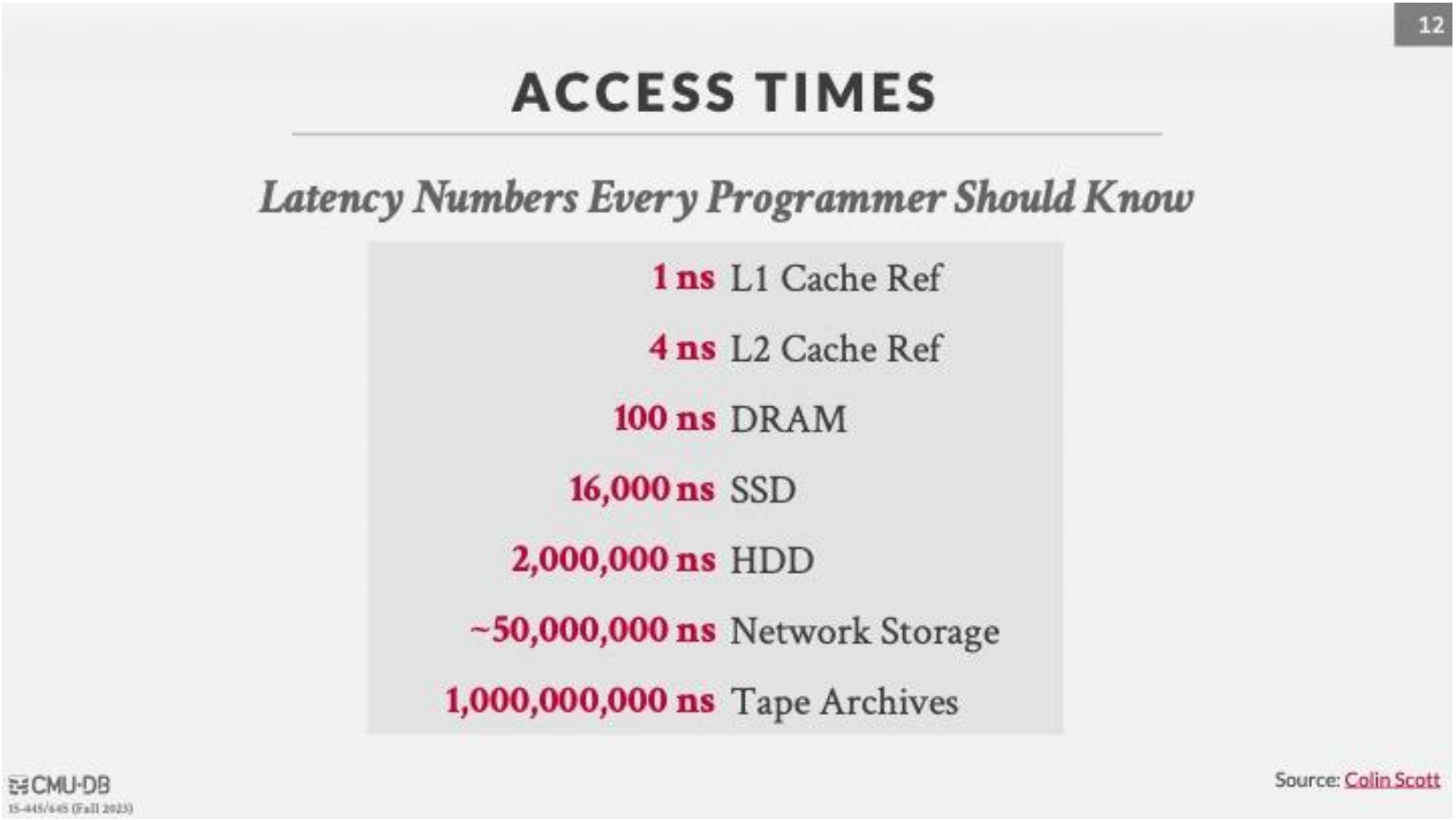
Persistent Storage



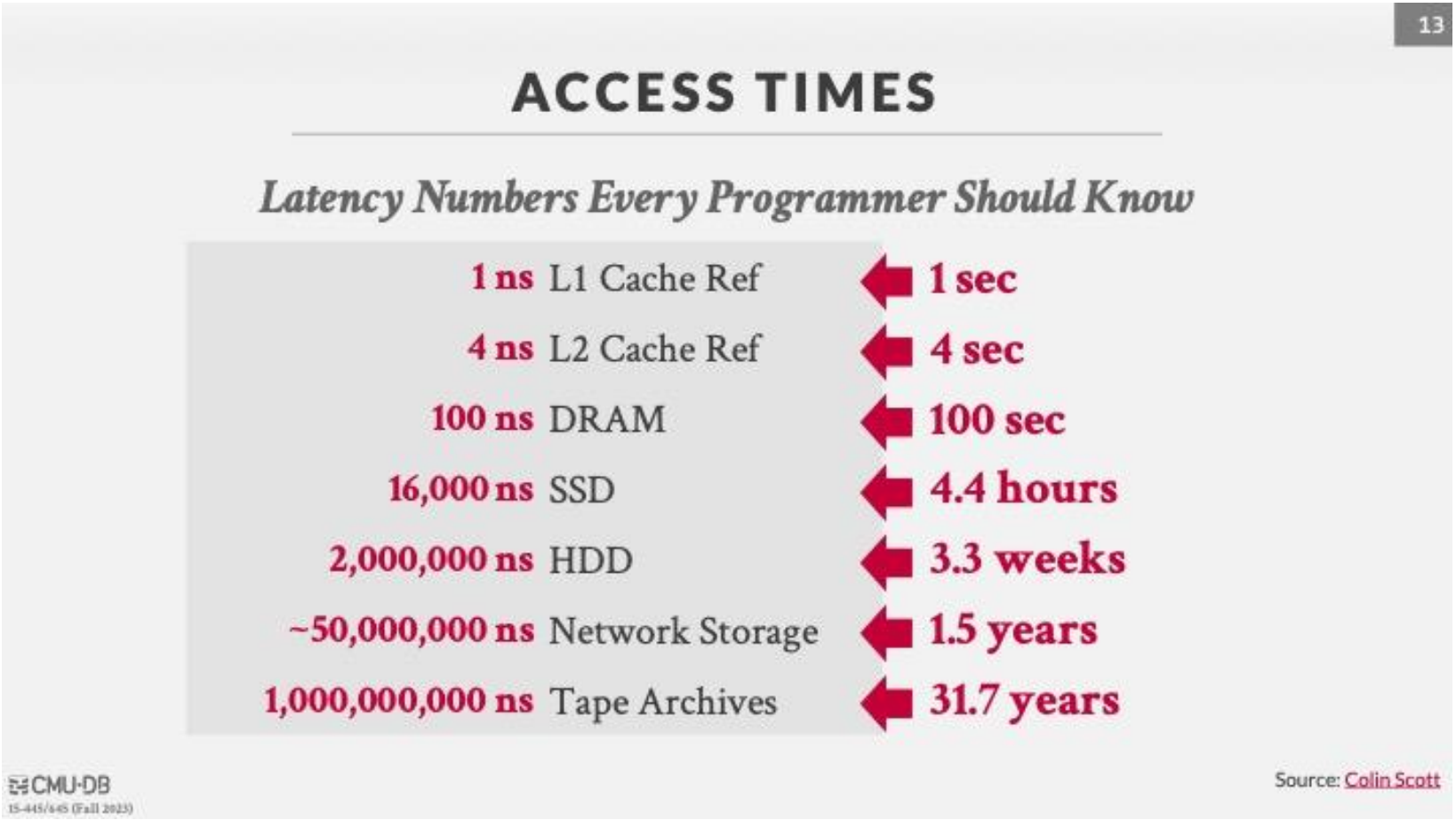
Credit: https://15445.courses.cs.cmu.edu/fall2023/



Credit: https://15445.courses.cs.cmu.edu/fall2023/



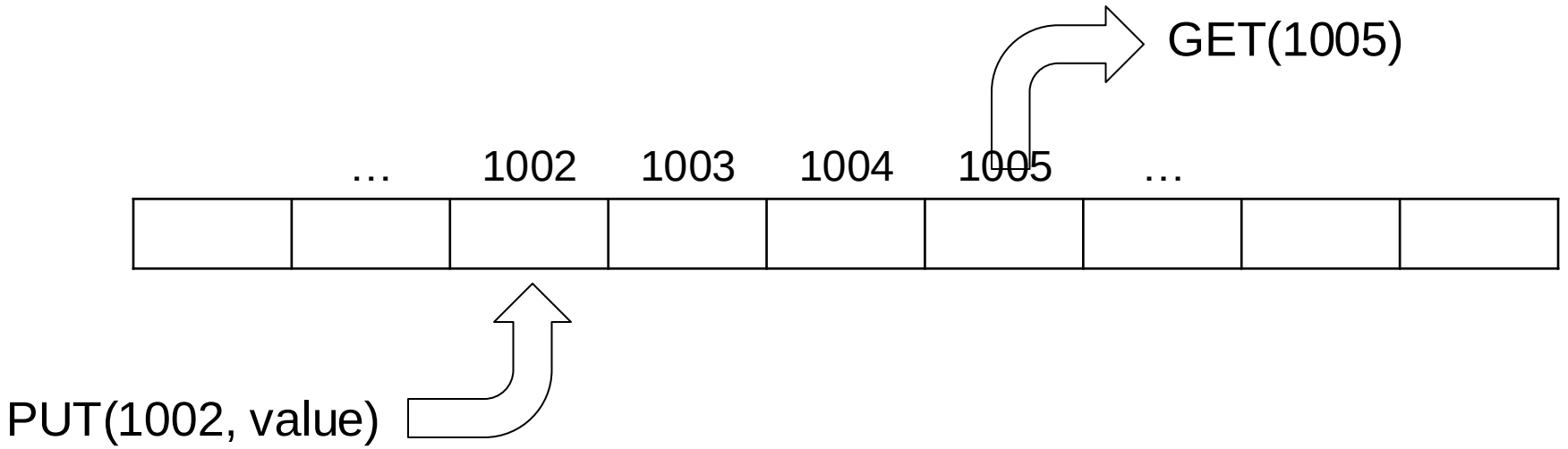
Credit: https://15445.courses.cs.cmu.edu/fall2023/



Credit: https://15445.courses.cs.cmu.edu/fall2023/

Basics of Storage Methods

- Multiple storage layers
- Each layer: an array of locations
- Location: word or page/block



Persistent Storage

- Different technologies:
  - SDD: your laptop, or cache for HDD
  - HDD: most today's data is stored here
  - Tapes: long-term archives

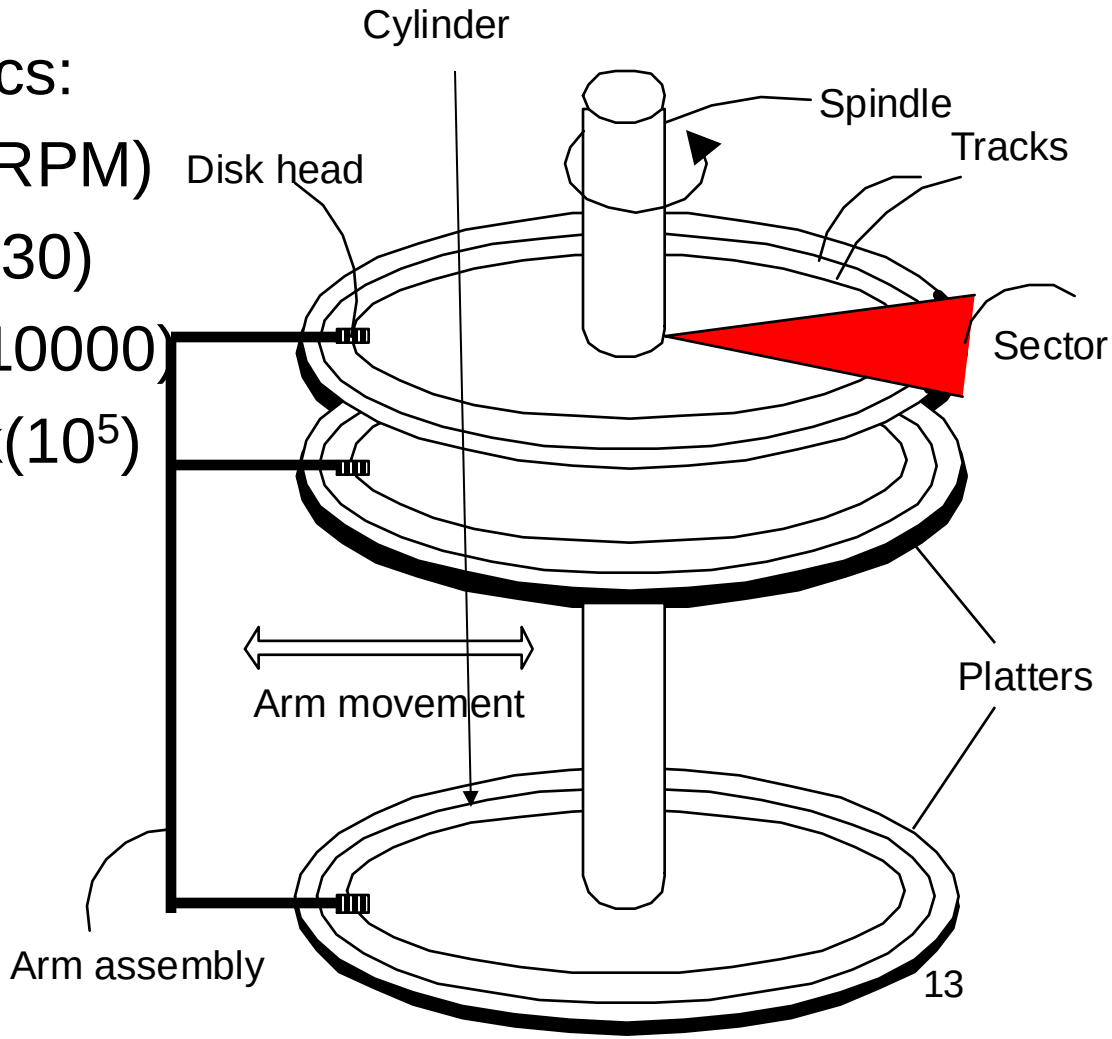
- Unit of Read/Write operation:

  1 block = 0.5k .. 32k

Hard Drive Disk (HDD)

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)

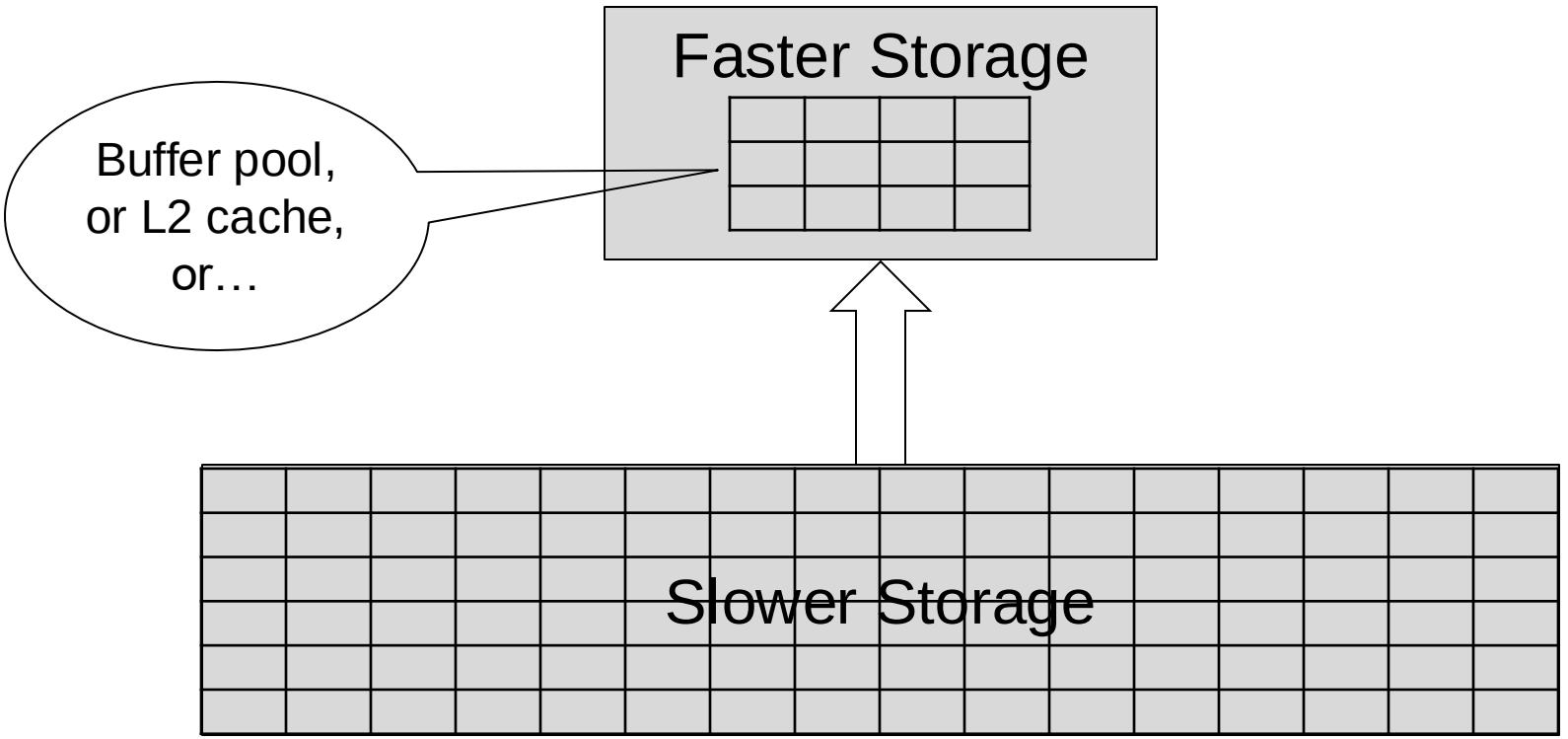


Disk Access Characteristics

- Latency = seek time + rotational latency
- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for sector to rotate
  - Rotation time = 10ms
  - Average latency = 10ms / 2
- Throughput = typically 40-80MB/s

The Buffer Pool

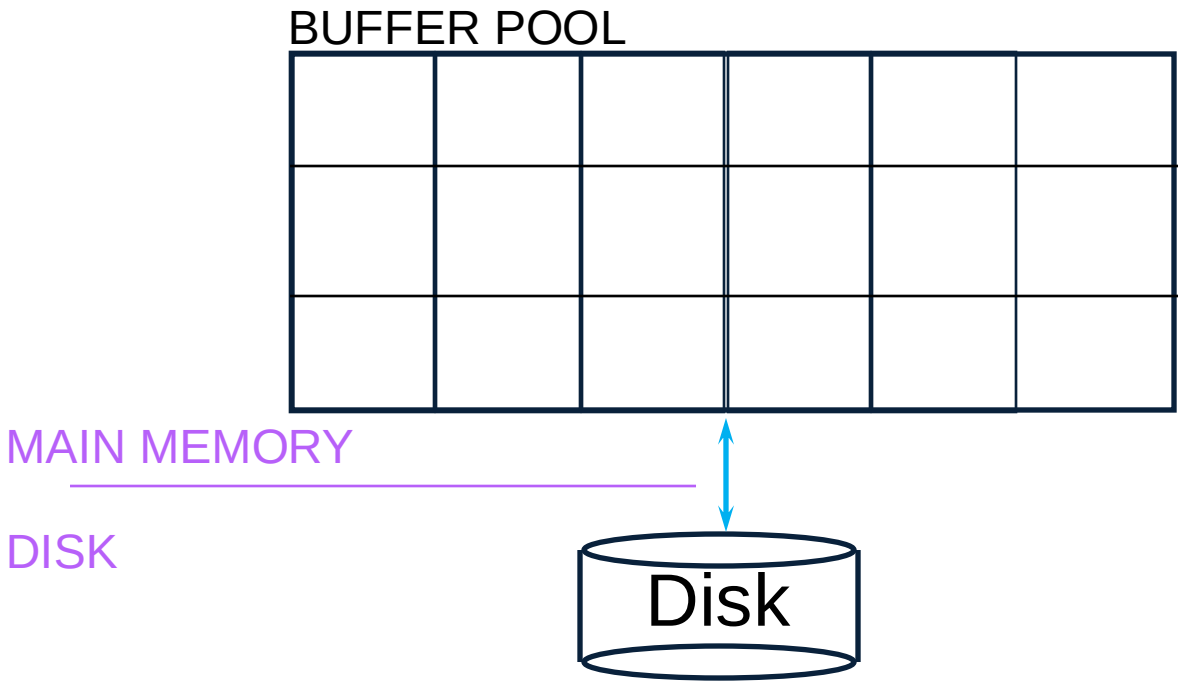
Hiding Latency



The Buffer Pool

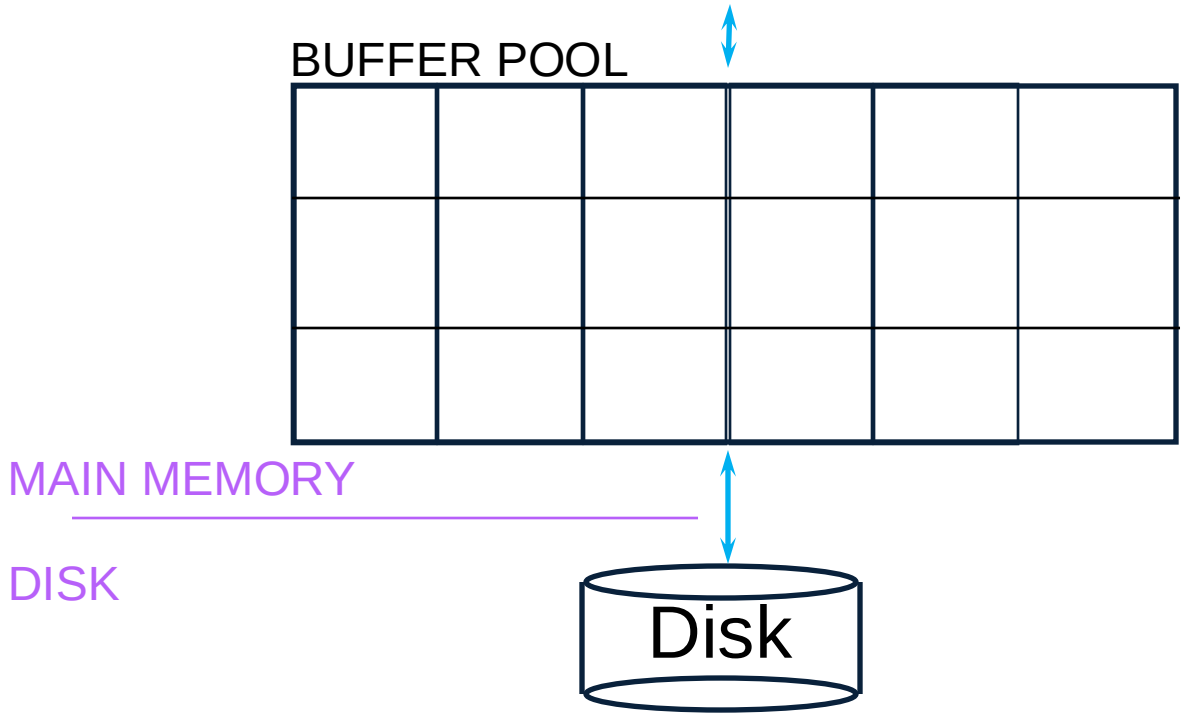
- Fixed chunk of main memory
- After disk read, store it in buffer pool
- If requested later, read from buffer pool
- If buffer pool is full, evict one page

Buffer Manager



Buffer Manager

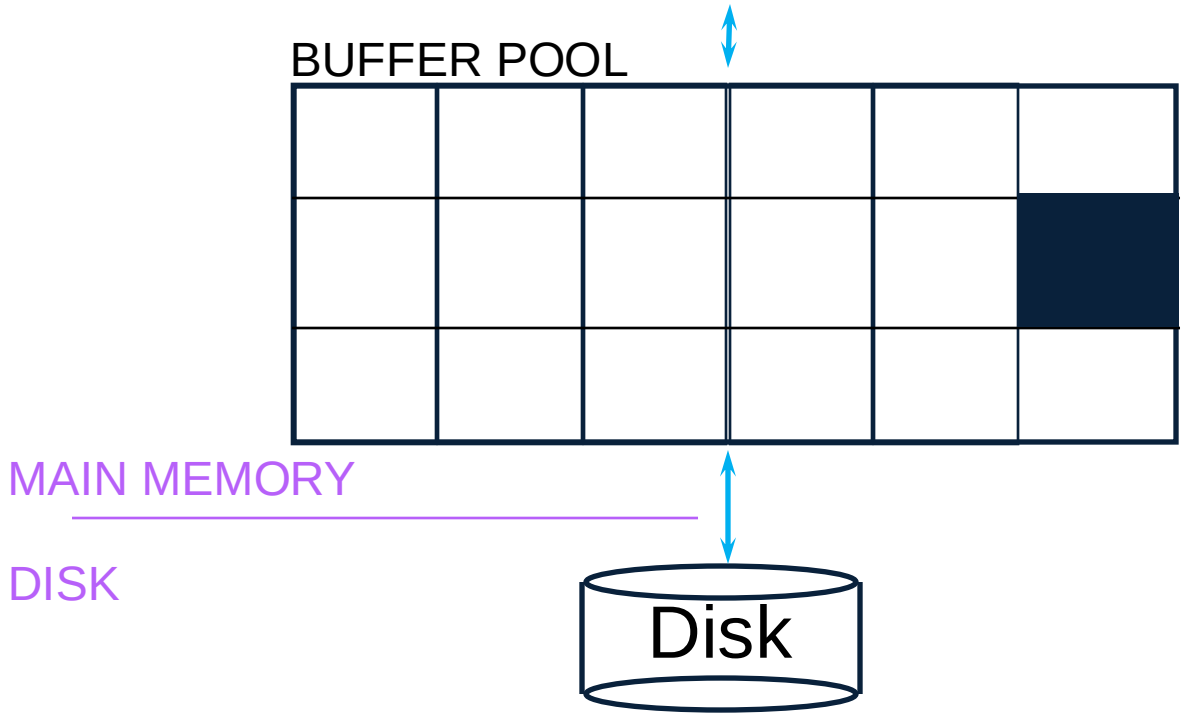
Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager

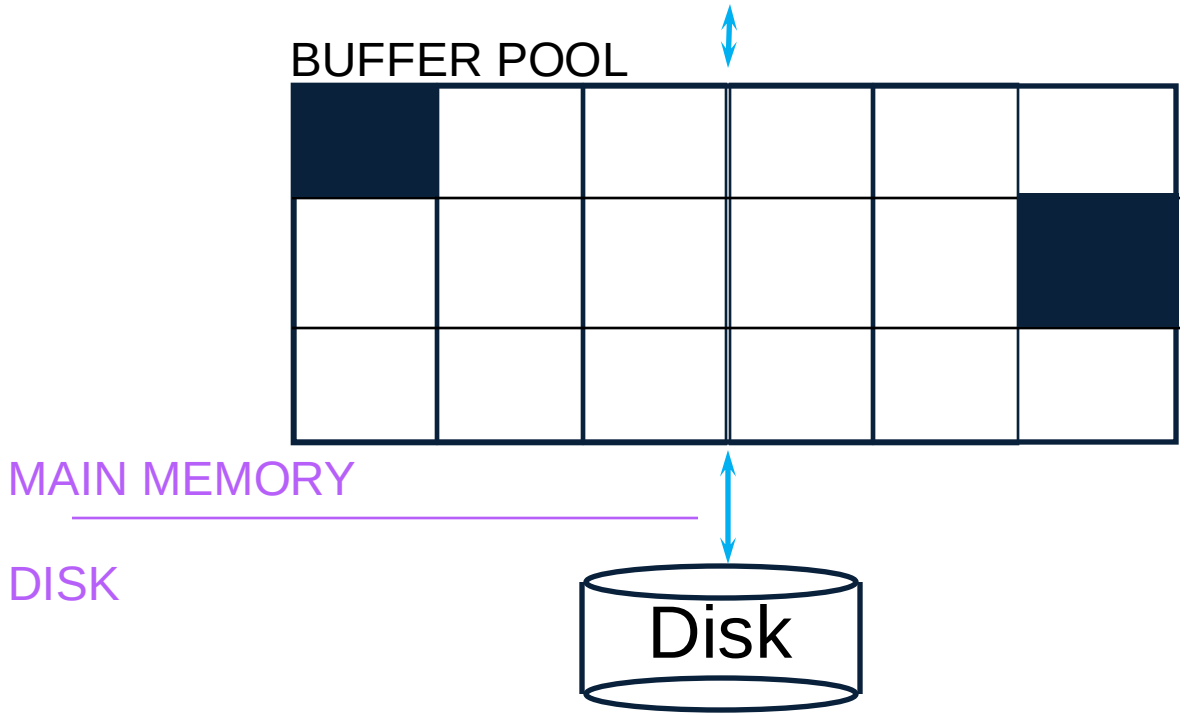
Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager

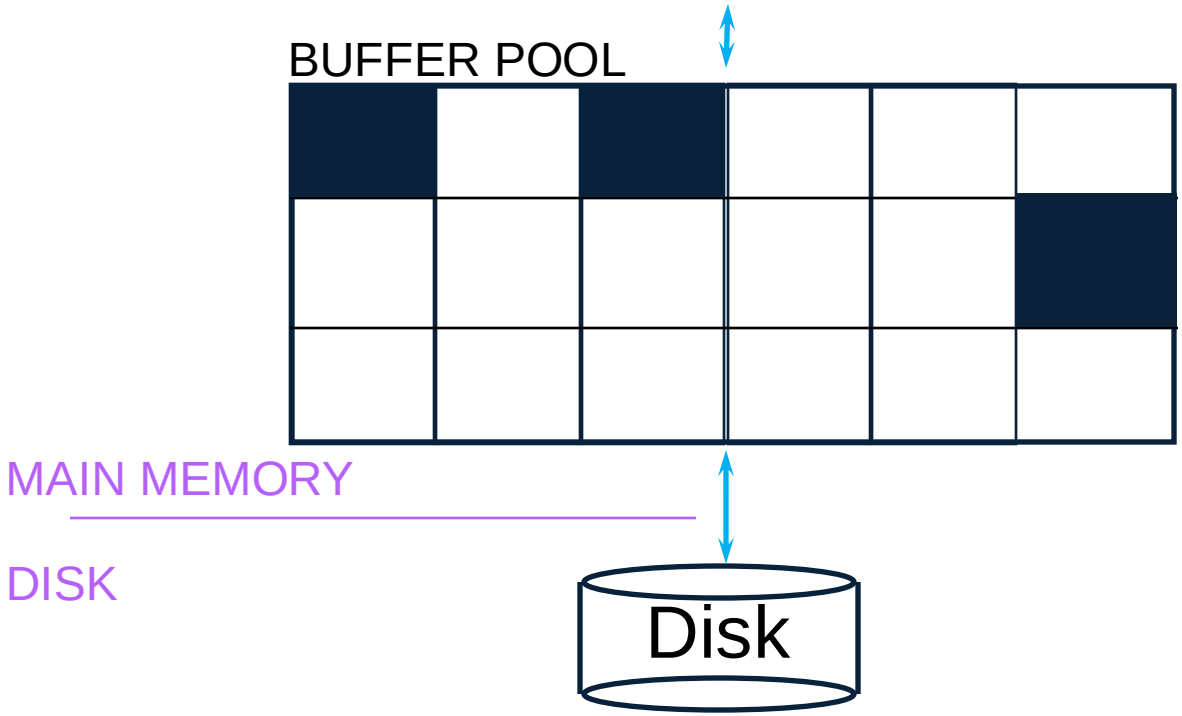
Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager

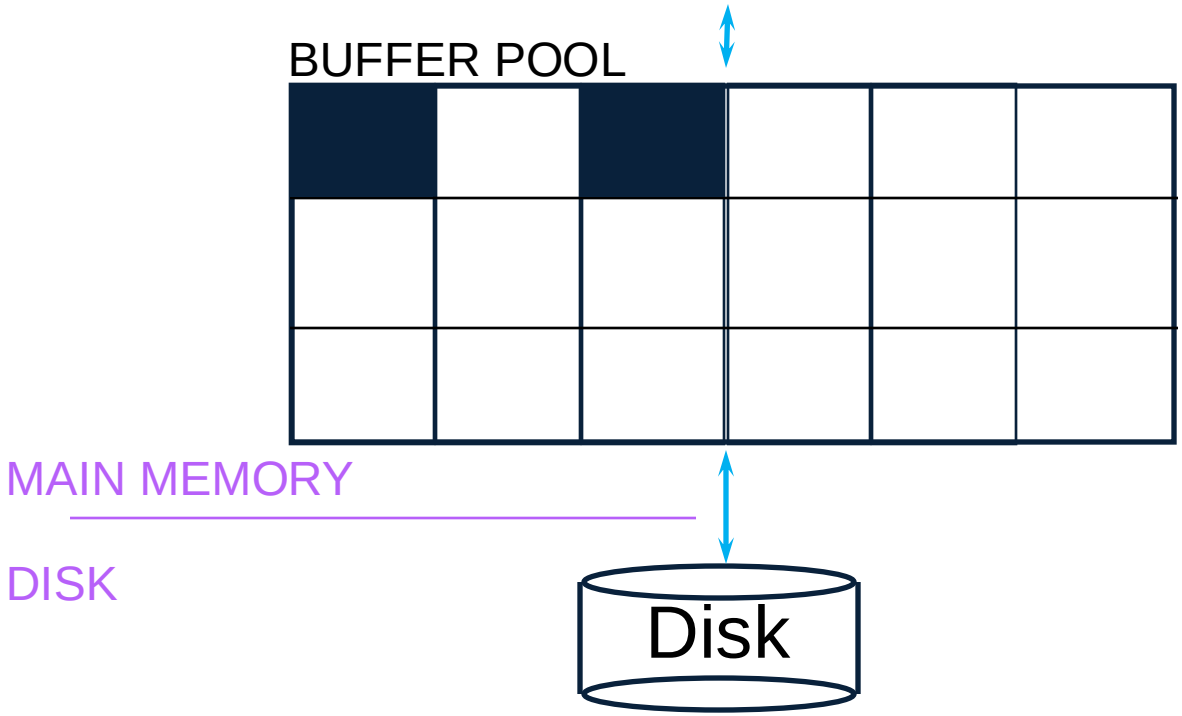
Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager

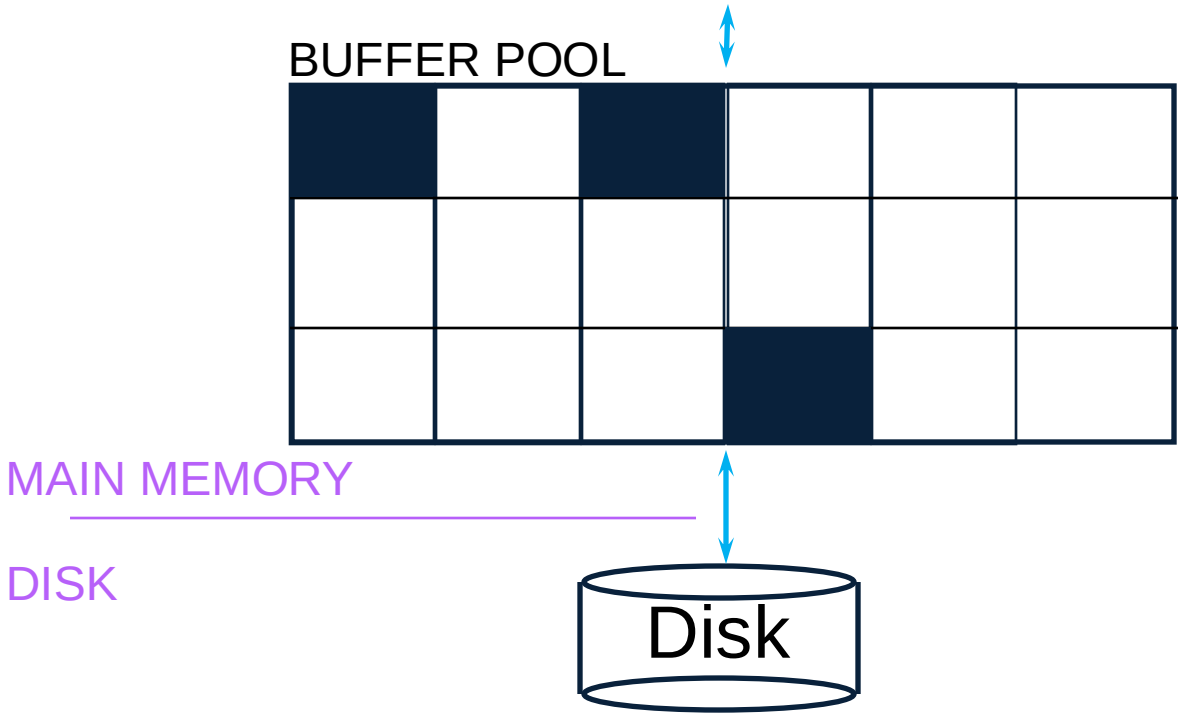
Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

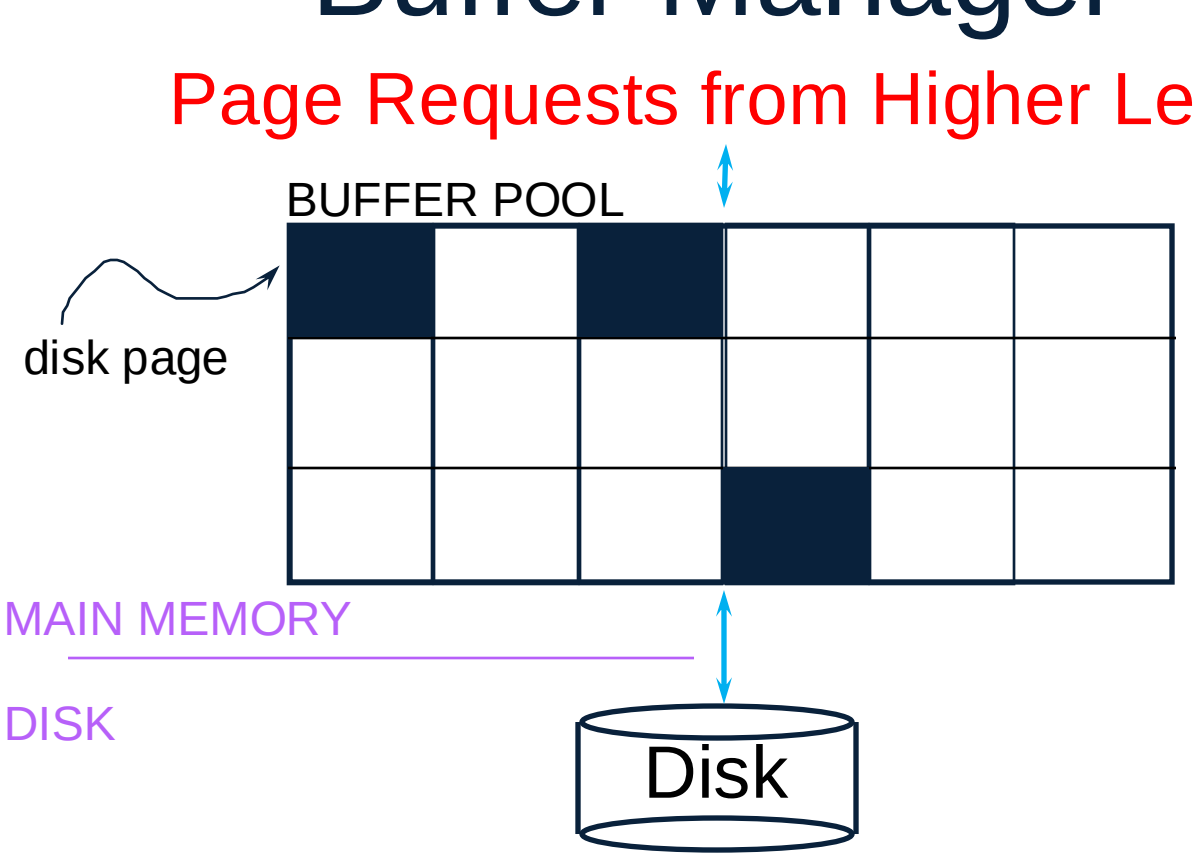
Buffer Manager

Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

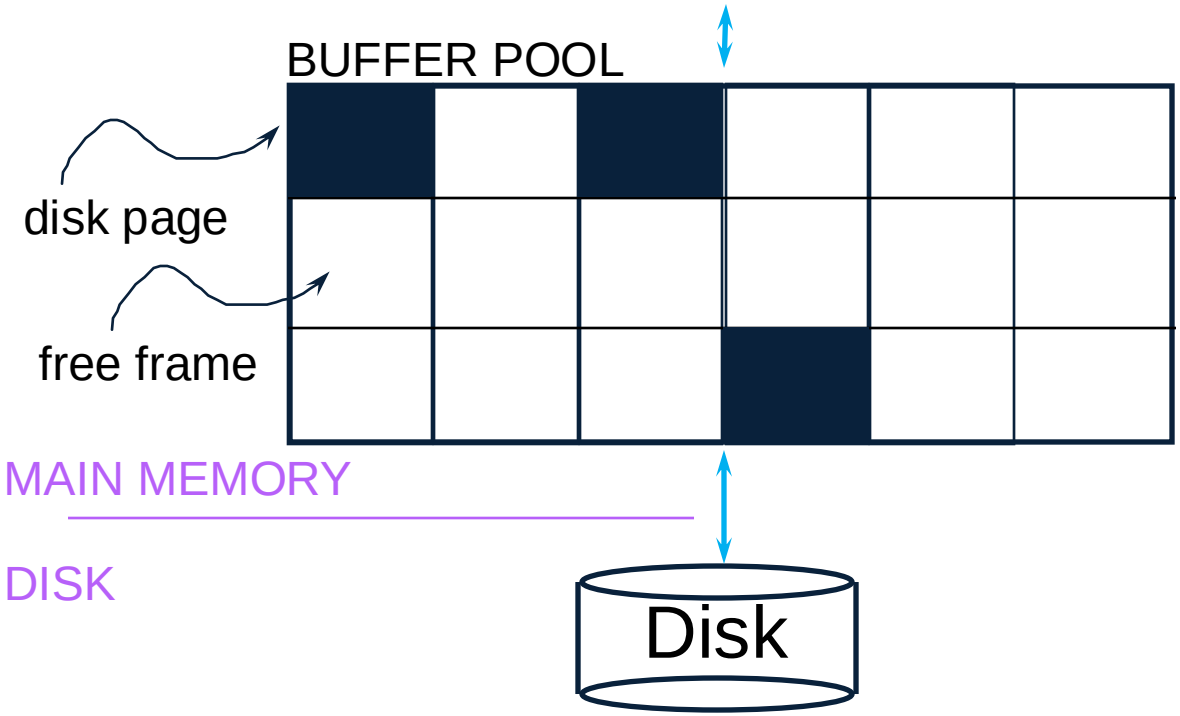
Buffer Manager



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

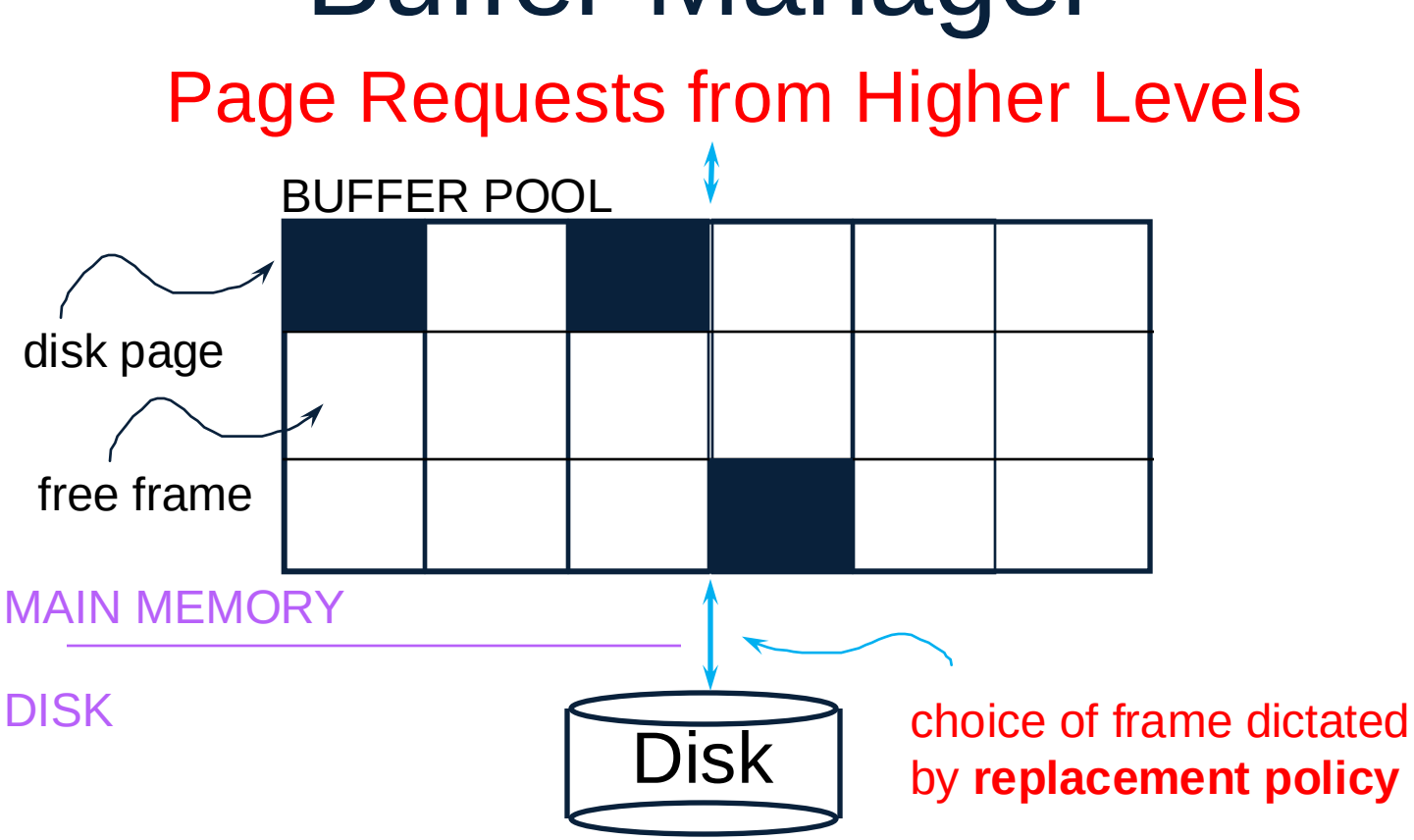
Buffer Manager

Page Requests from Higher Levels



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs

Buffer Manager

When buffer is full, need to evict a page

Page replacement policy:

- LRU: Least Recently Used (in class)
- Clock algorithm (on your own)

Both work well in OS, need tweaks in DBs

Discussion

- Database resides on disk (next)
- Initially: cold start, cold cache
- Eventually: warm cache
- Always run experiments with warm cache

Storage Manager

Storage Manager

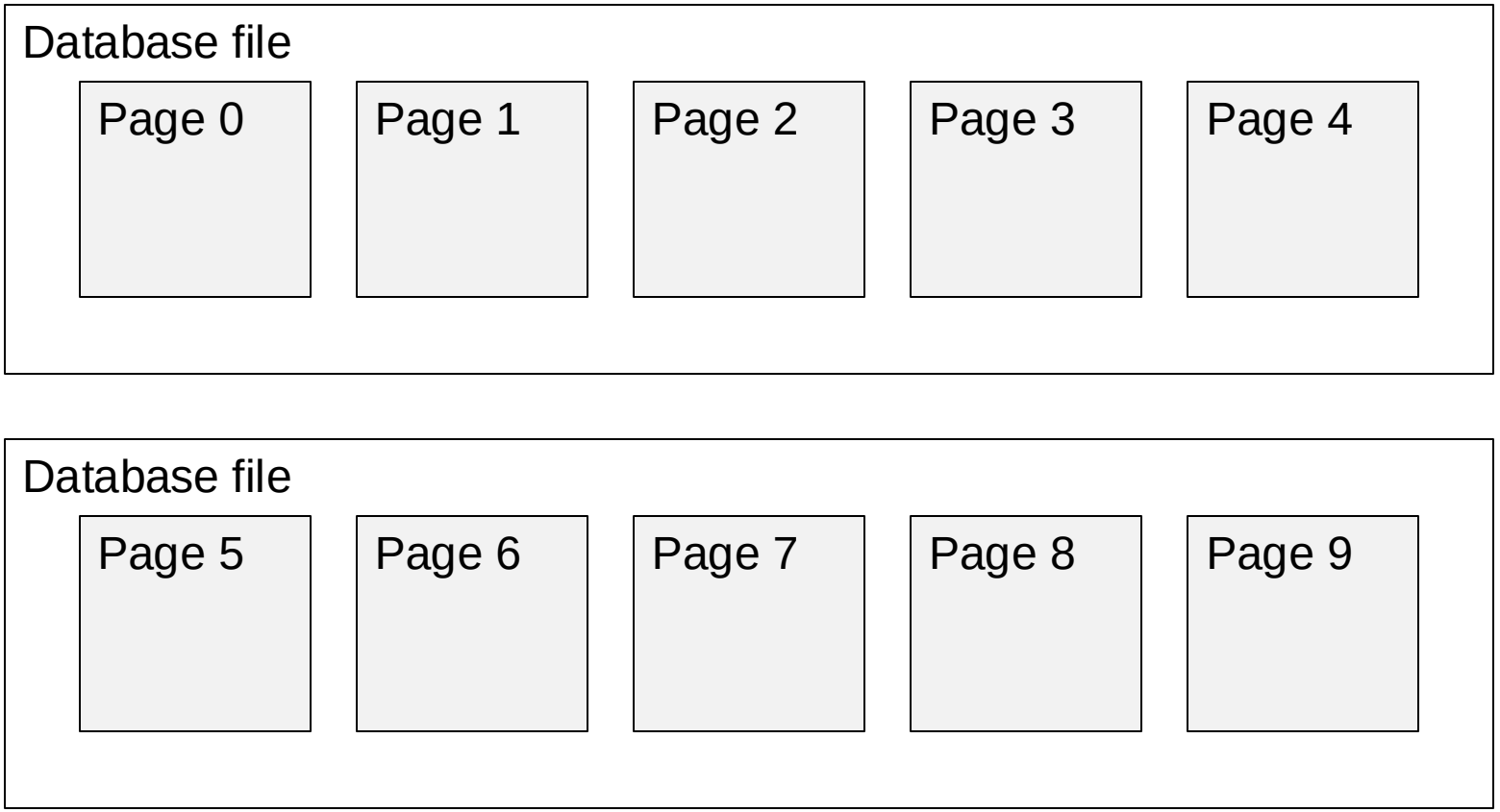
Past: DBMS managed the raw disk

Today: DBMS use OS to create files

- 1 file = multiple (continuous?) blocks
- 1 block = multiple records, free space

Storage manager keeps track of the files, their content, free space

Storing Pages on Disk



Catalog of metadata: files, pages, records, free space

Page Format

- 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

- 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: How does this differ?

Page Format

- 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

- 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: How does it differ?

Page Format

- 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
  - Each page contains values from only one attribute

Page Format

- 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
  - Each page contains values from only one attribute

Row-Oriented Storage

Logical schema

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*

Row-Oriented Storage

Sequential file: unordered set of records

- Advantage:
  - Can insert a new record at the end of the file, or in any page that has free space
- Disadvantage
  - Sequential search for a record (→indexes)
  - Overwrite entire block on update (→LSM)

Page Format

- One page contains several records
- Each record has unique record ID: RID
  - Needed for indexes, recovery log, etc
- 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

Page Format Approach 1

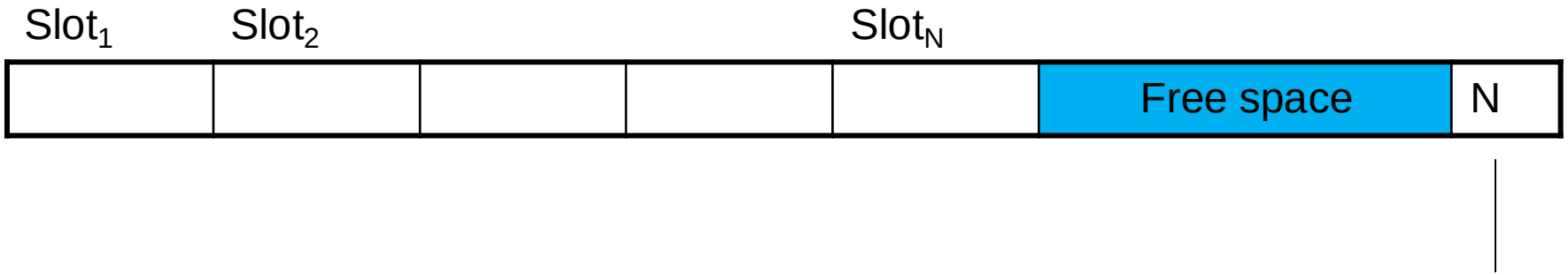
Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

Page Format Approach 1

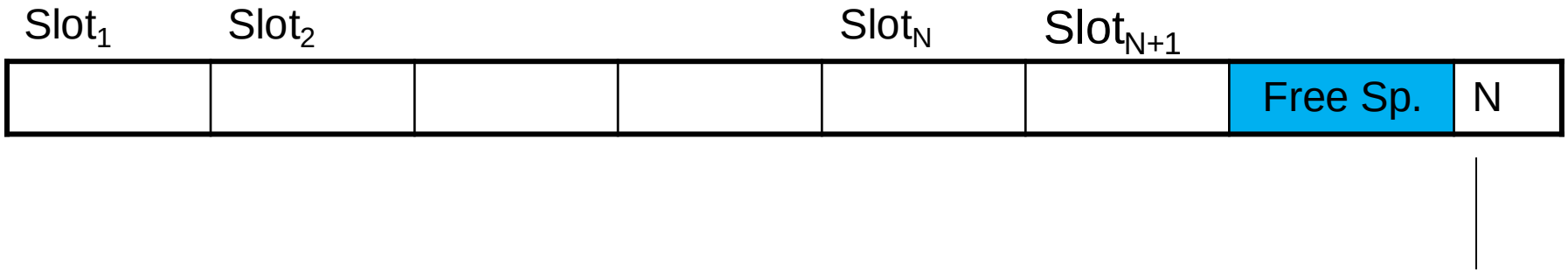
Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)



How do we insert a new record?

Page Format Approach 1

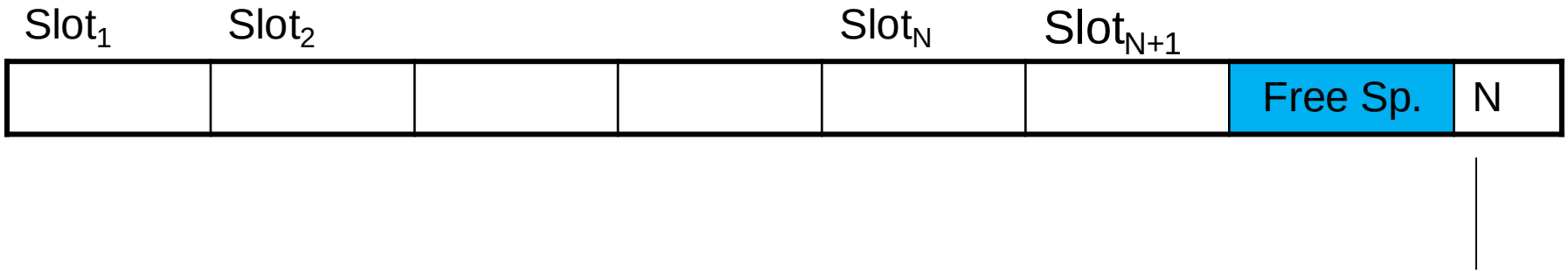
Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)



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)

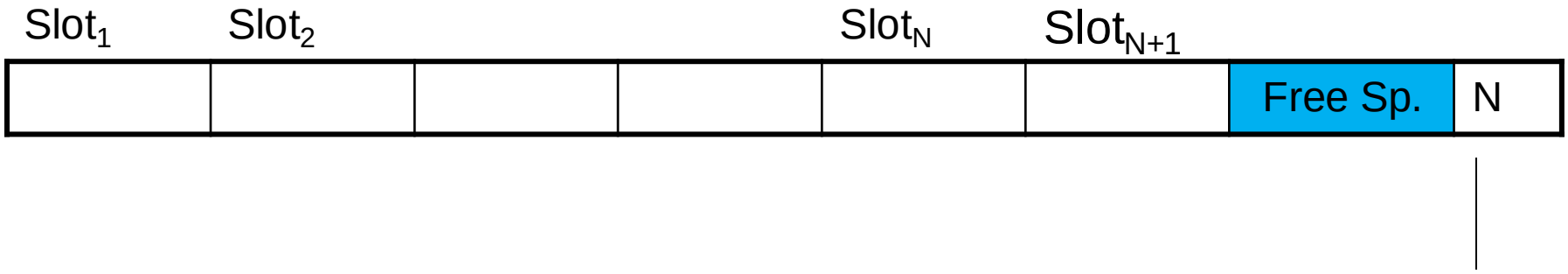


How do we insert a new record?

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)

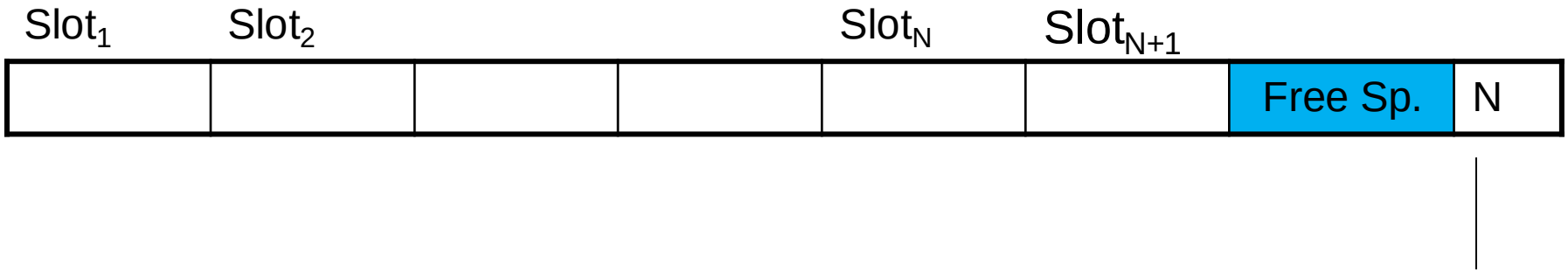


How do we insert a new record?

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

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)



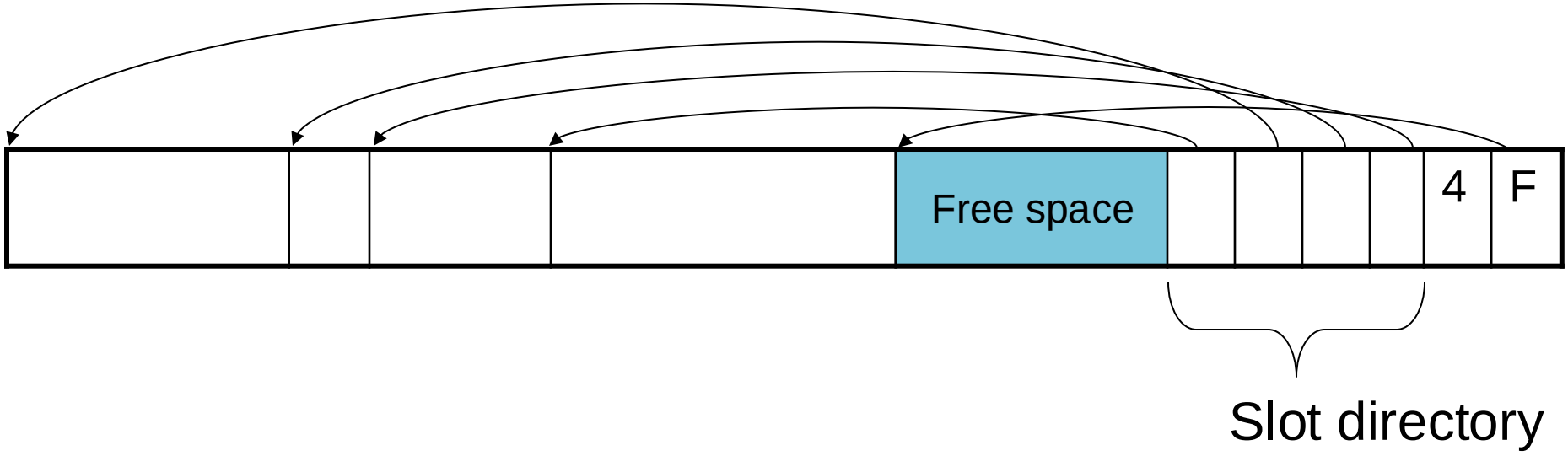
How do we insert a new record?

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

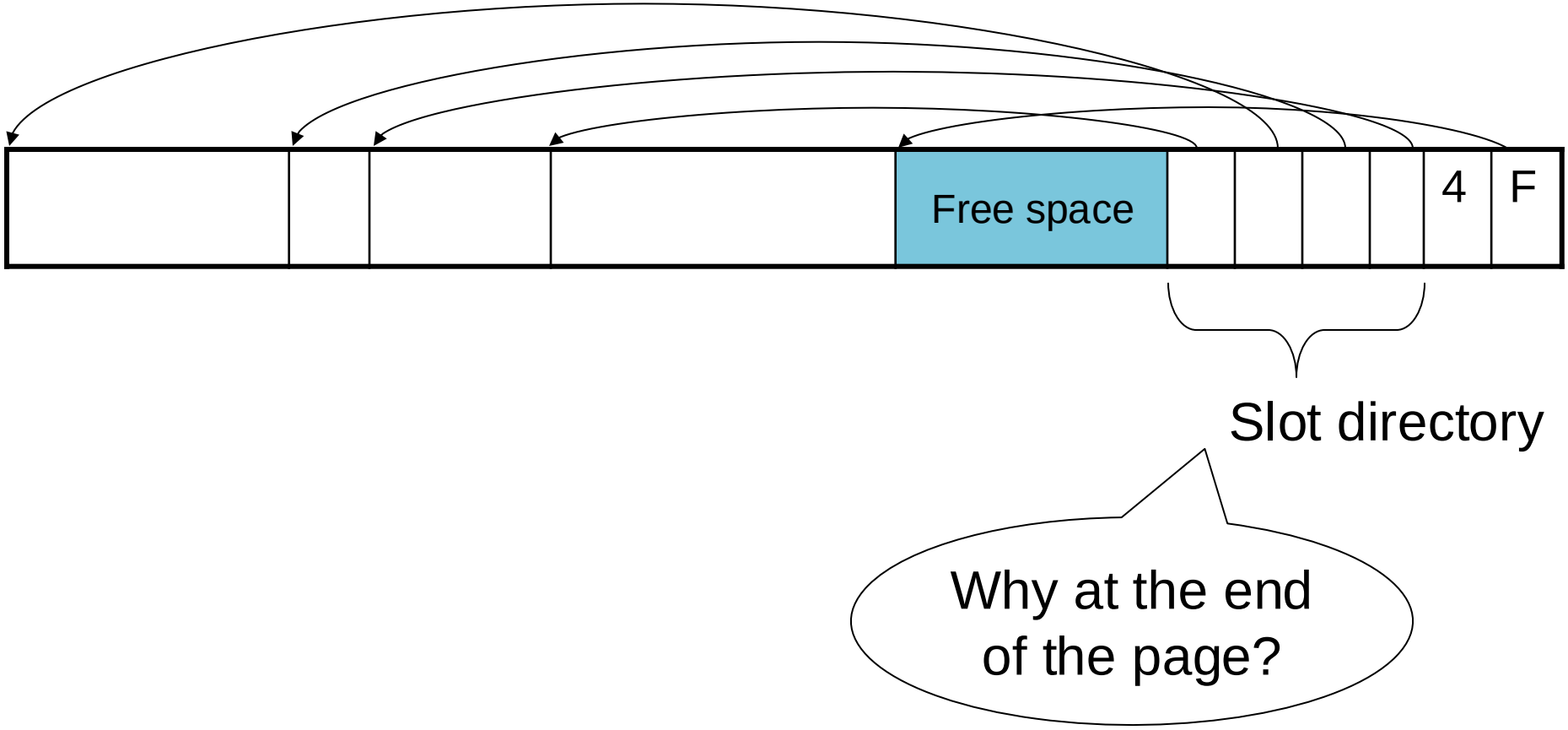
How do we handle variable-length records?

Page Format Approach 2

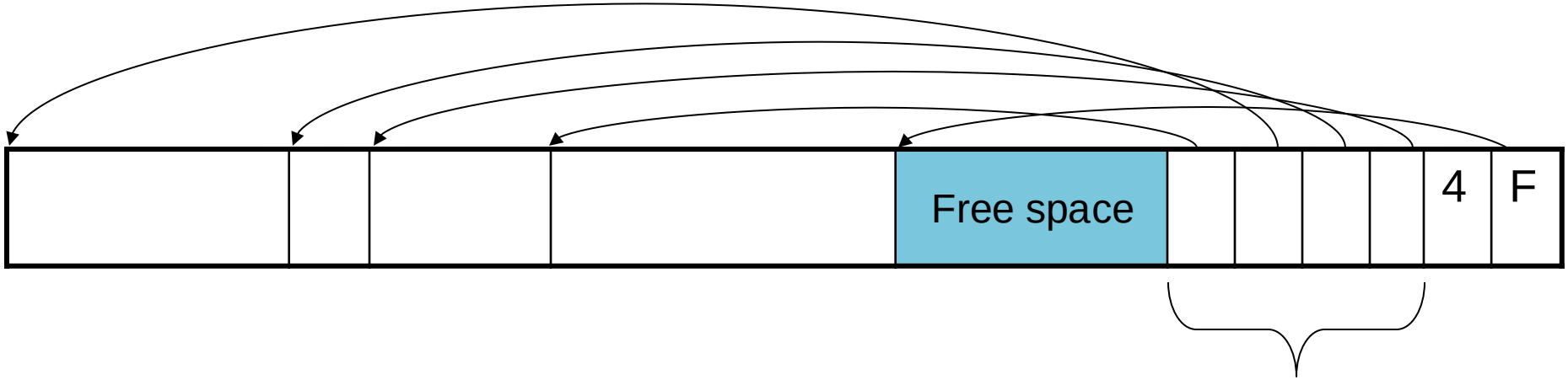
Page Format Approach 2



Page Format Approach 2



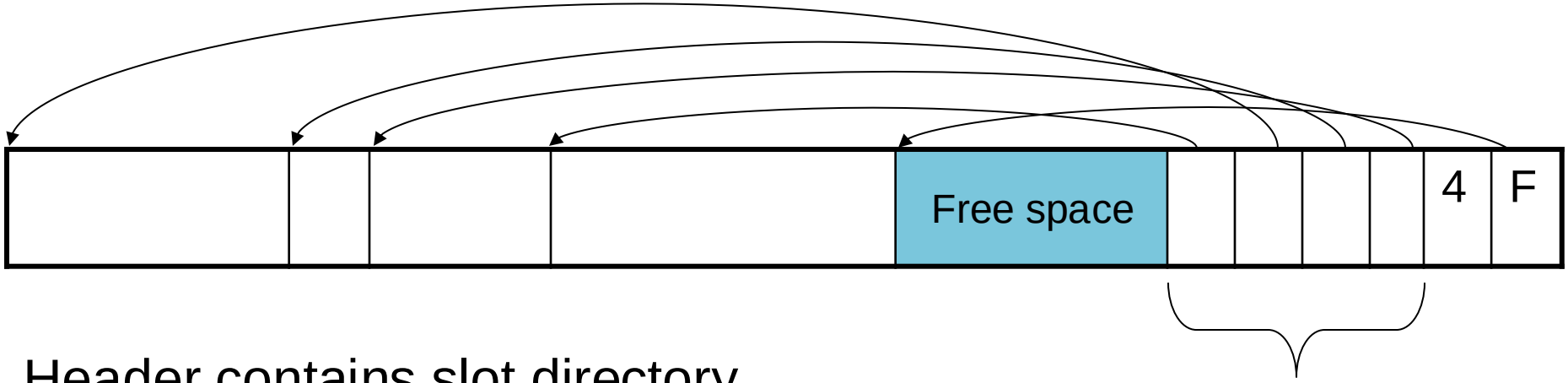
Page Format Approach 2



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

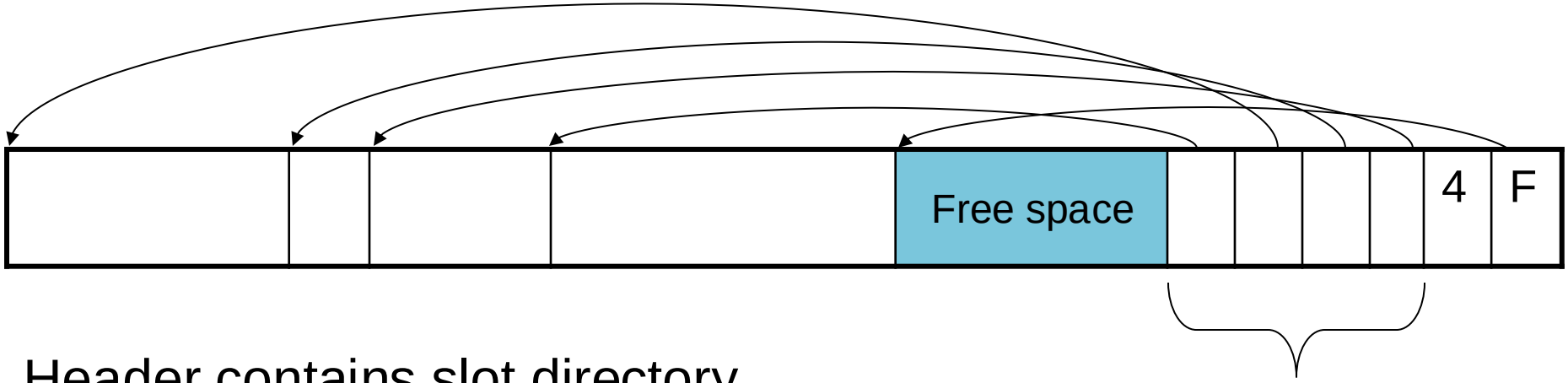
Why at the end of the page?

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)
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?

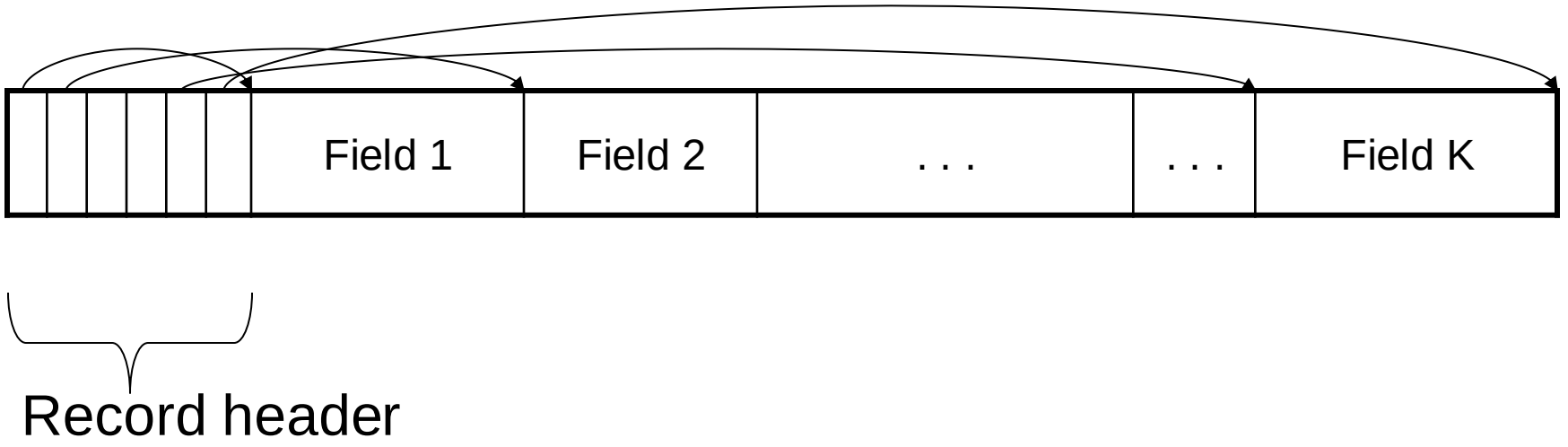
Record Formats

Fixed-length records: when each field has a fixed length

Information about field lengths and types is in the catalog

Record Formats

Variable length records



Remark: NULLS require no space at all

Row-Oriented: Summary

- Sequential file: records in arbitrary order
- One page: a set of records
  - Record cannot cross block boundary
- One record: sequence of attributes

PAX

PAX

- One page: set of records (as before)
- Records: column-oriented WHY?

PAX

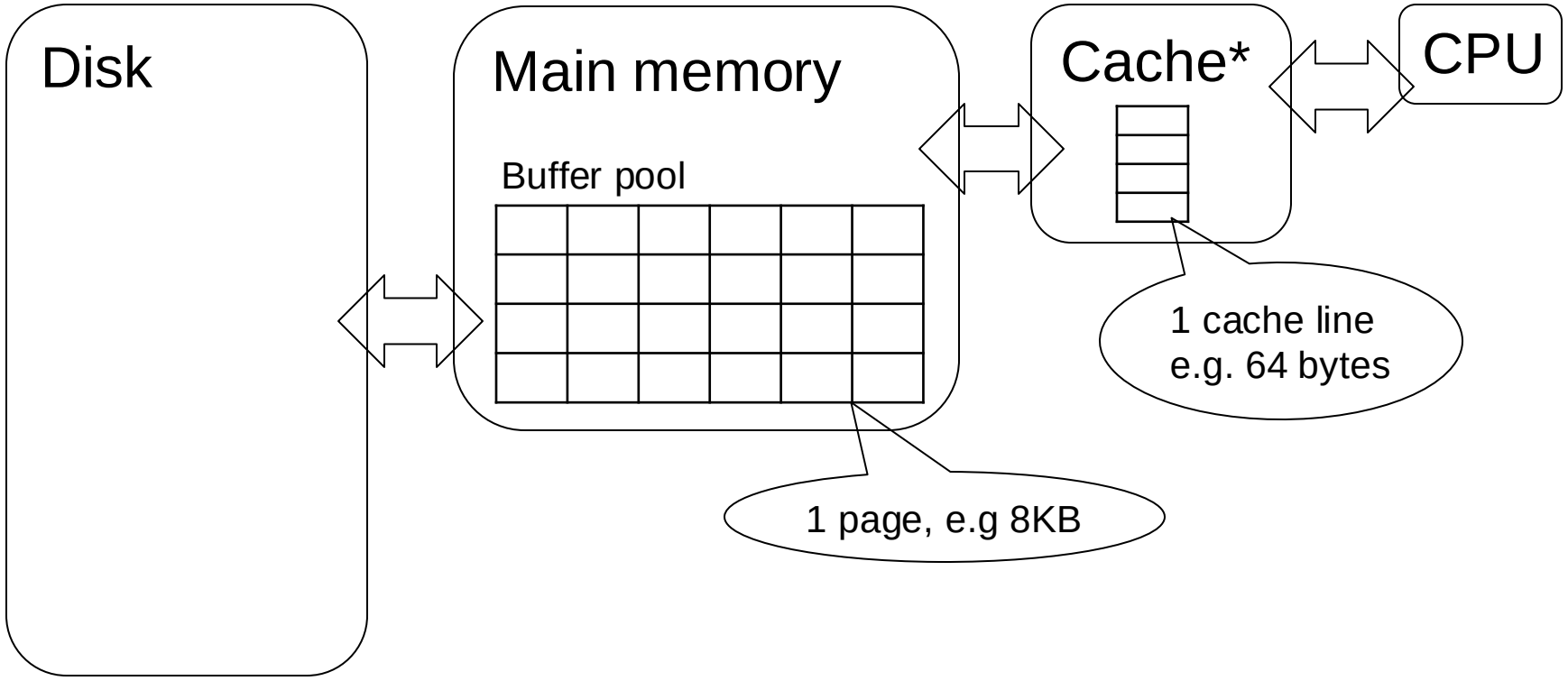
- One page: set of records (as before)
- Records: column-oriented
- This improves the L2-cache locality, as we will see next

PAX

- One page: set of records (as before)
- Records: column-oriented
- This improves the L2-cache locality, as we will see next
- I'm using (w/ permission) the slides from the original presentation at VLDB 2001

Recap

Memory hierarchies:



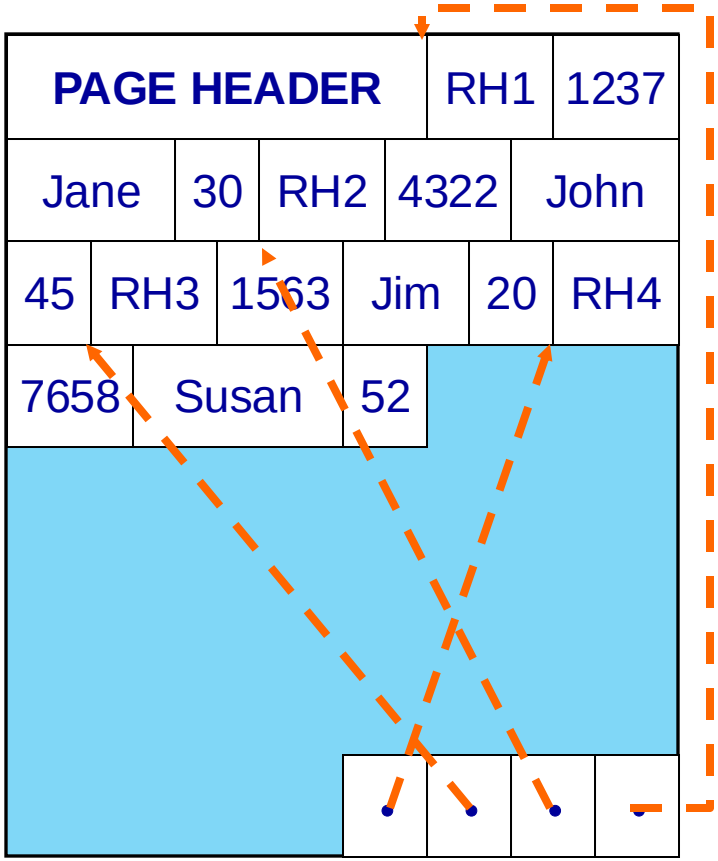
*aka CPU cache; several! L3, L2, L1 cache

Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

R

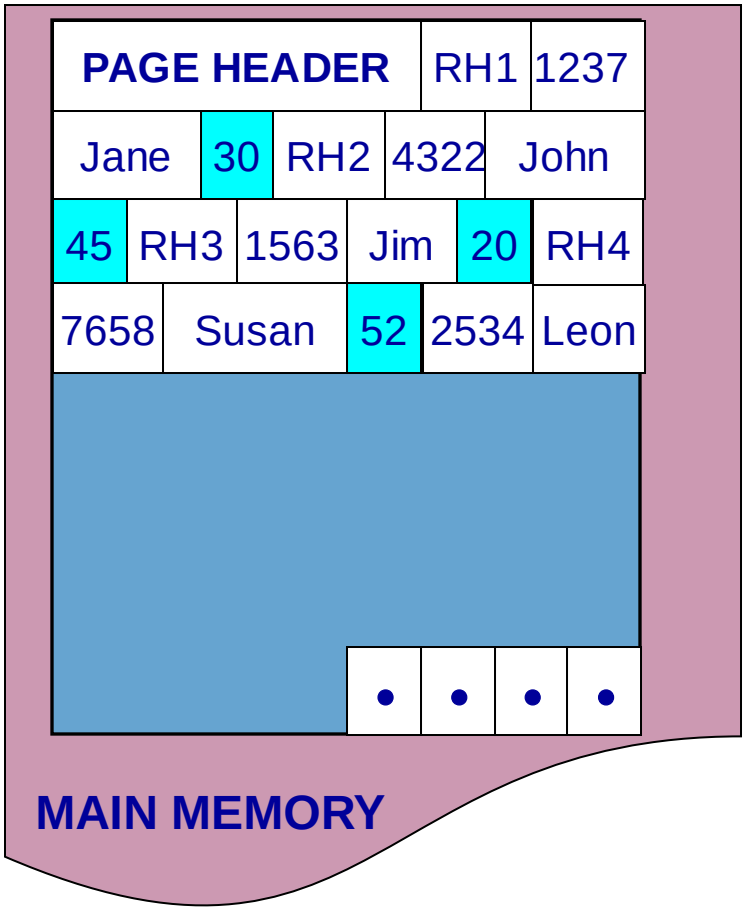
RID	SSN	Name	Age
1	1237	Jane	30
2	4322	John	45
3	1563	Jim	20
4	7658	Susan	52
5	2534	Leon	43
6	8791	Dan	37

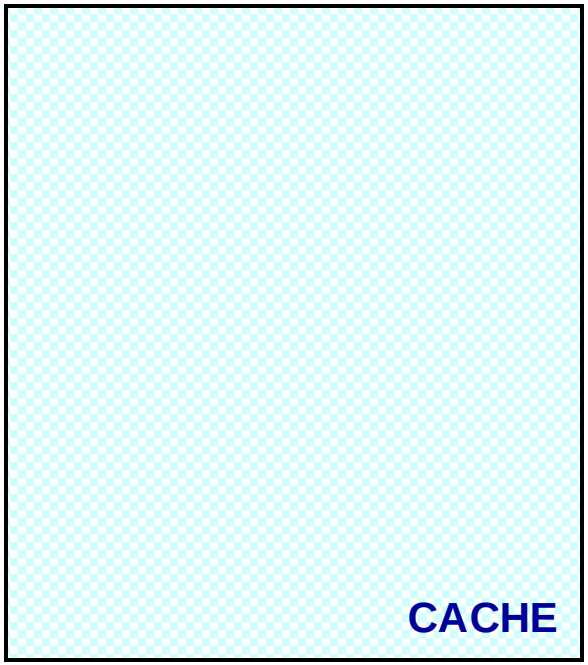


- Records are stored sequentially
- Offsets to start of each record at end of page

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

Predicate Evaluation using NSM



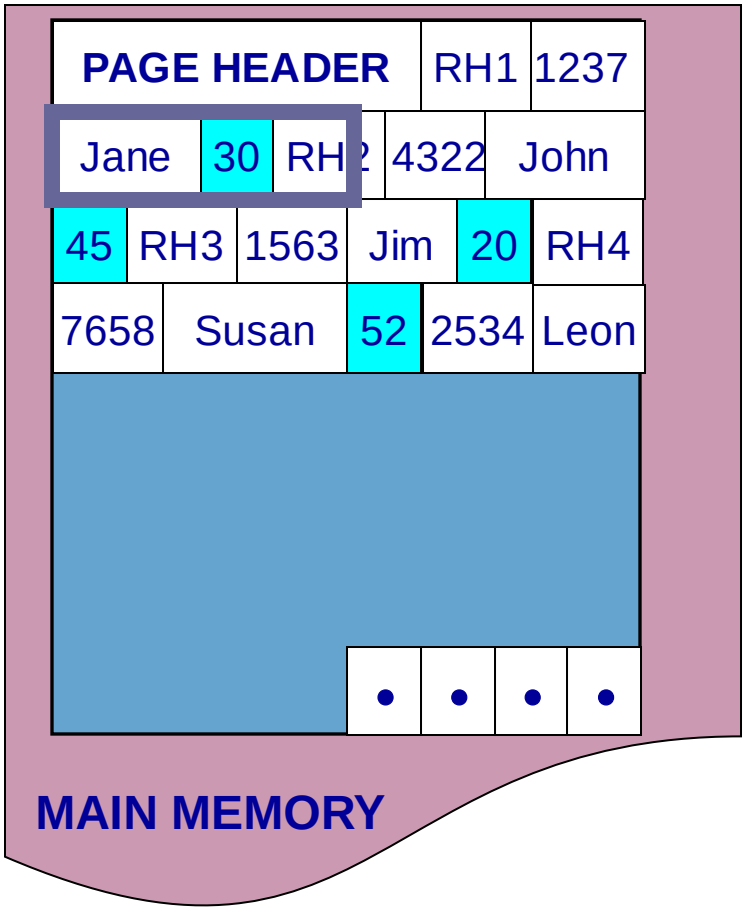


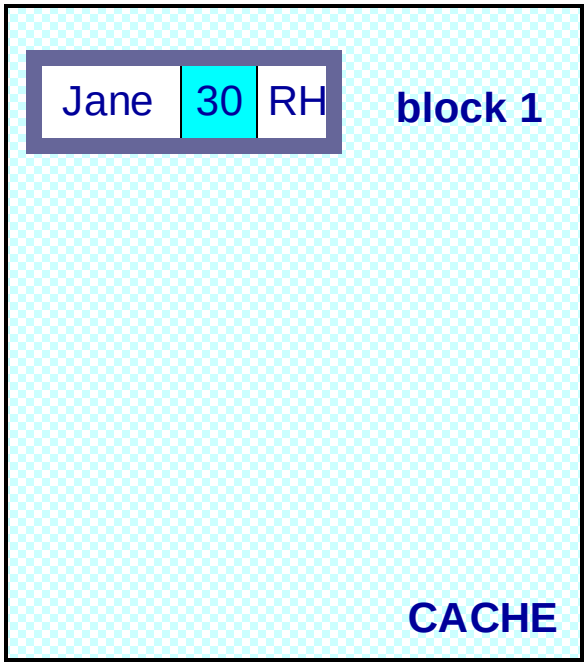

```
select name
from R
where age > 50
```

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



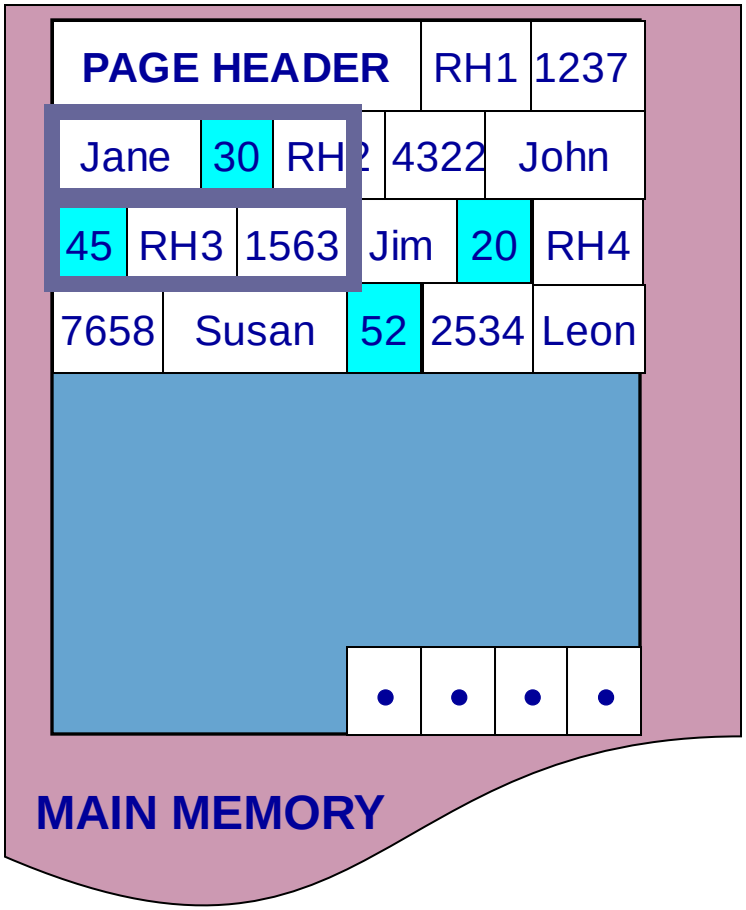


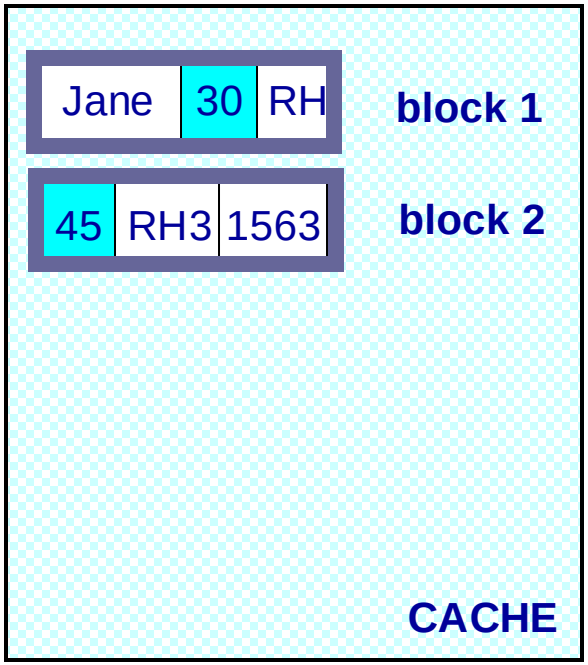

```
select name
from R
where age > 50
```

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



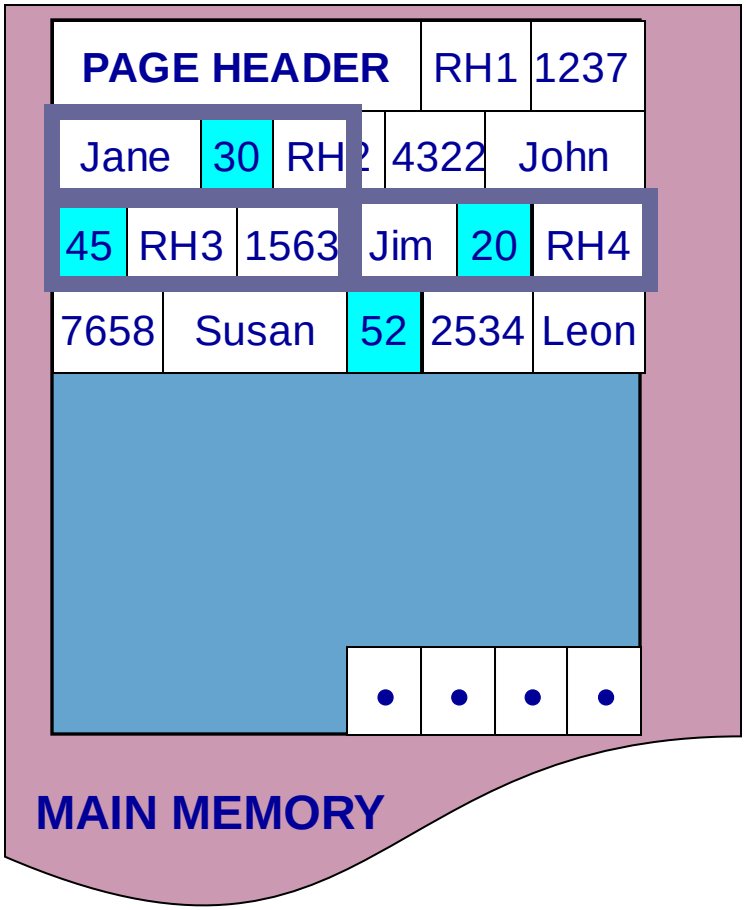


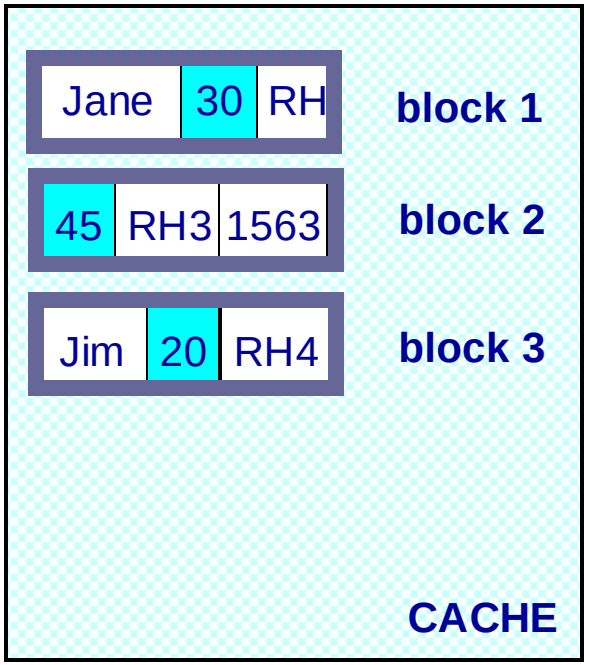

```
select name
from R
where age > 50
```

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



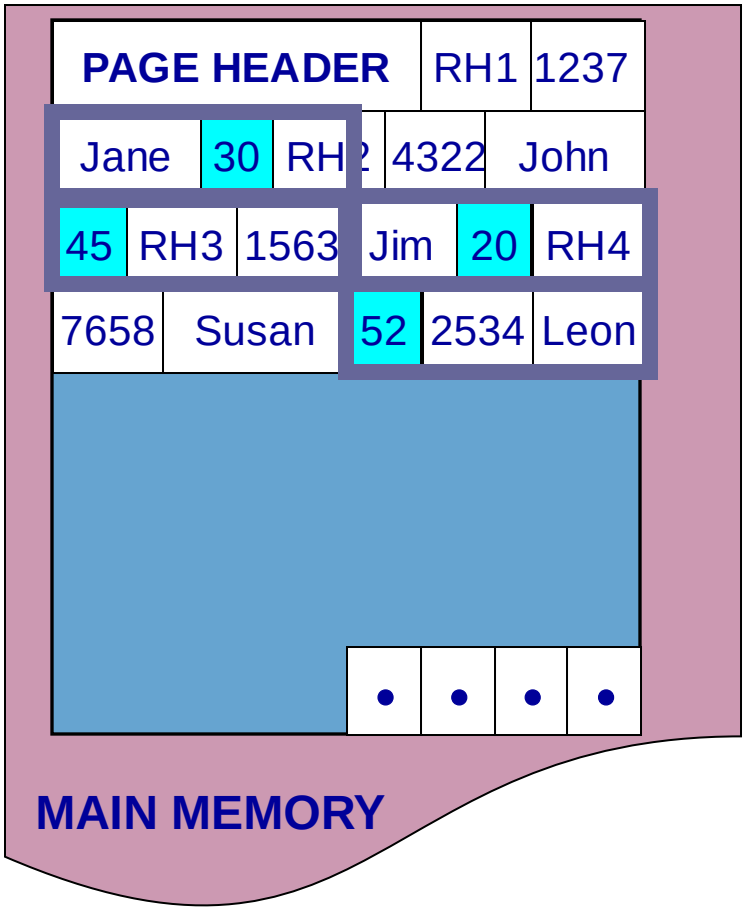


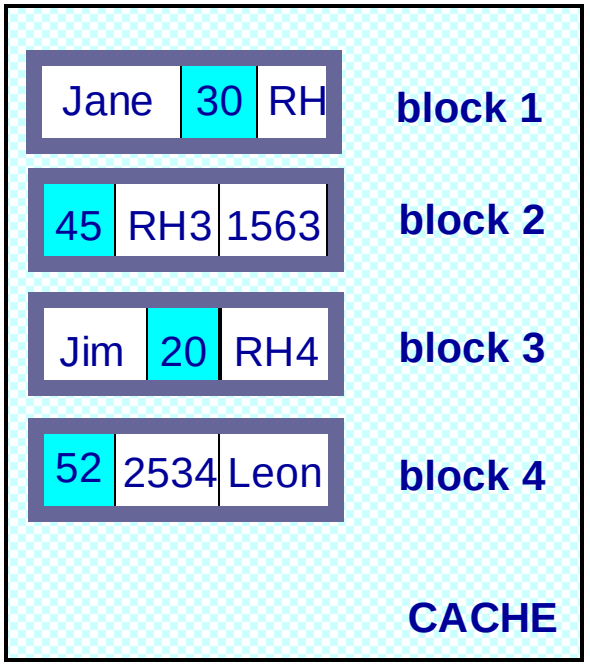

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

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

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

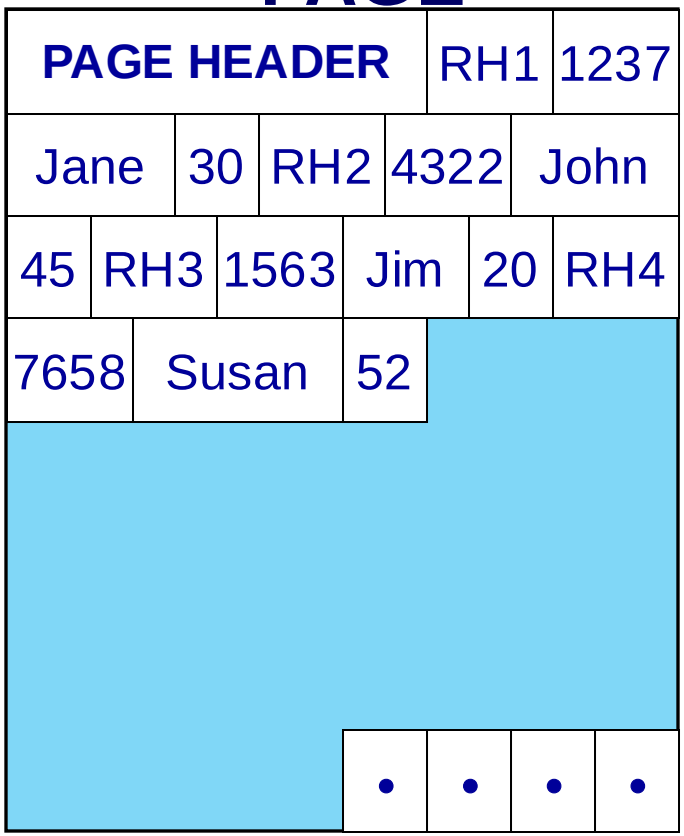
and, most importantly, is…

low-implementation-cost, high-impact

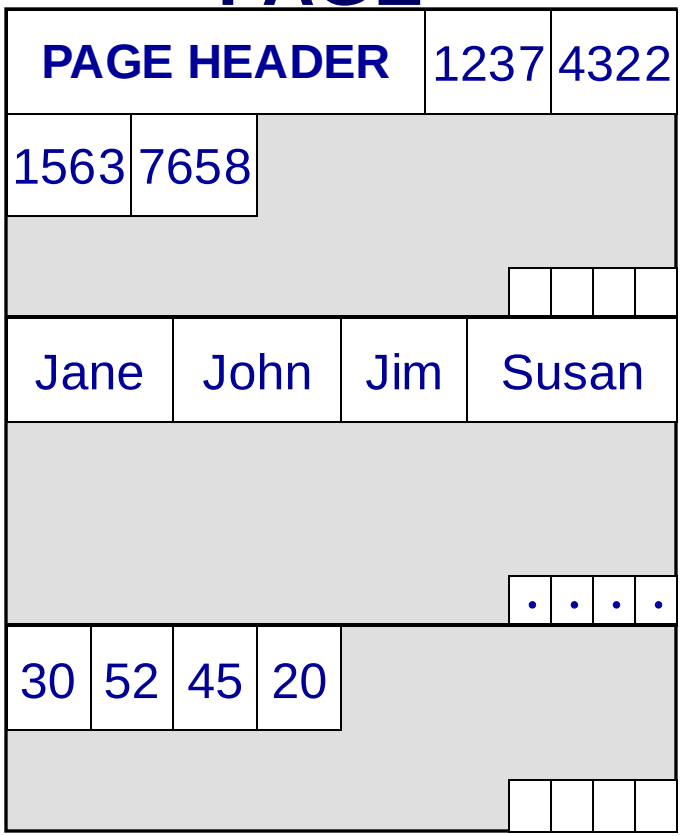
Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt

Partition Attributes Across (PAX)

NSM PAGE



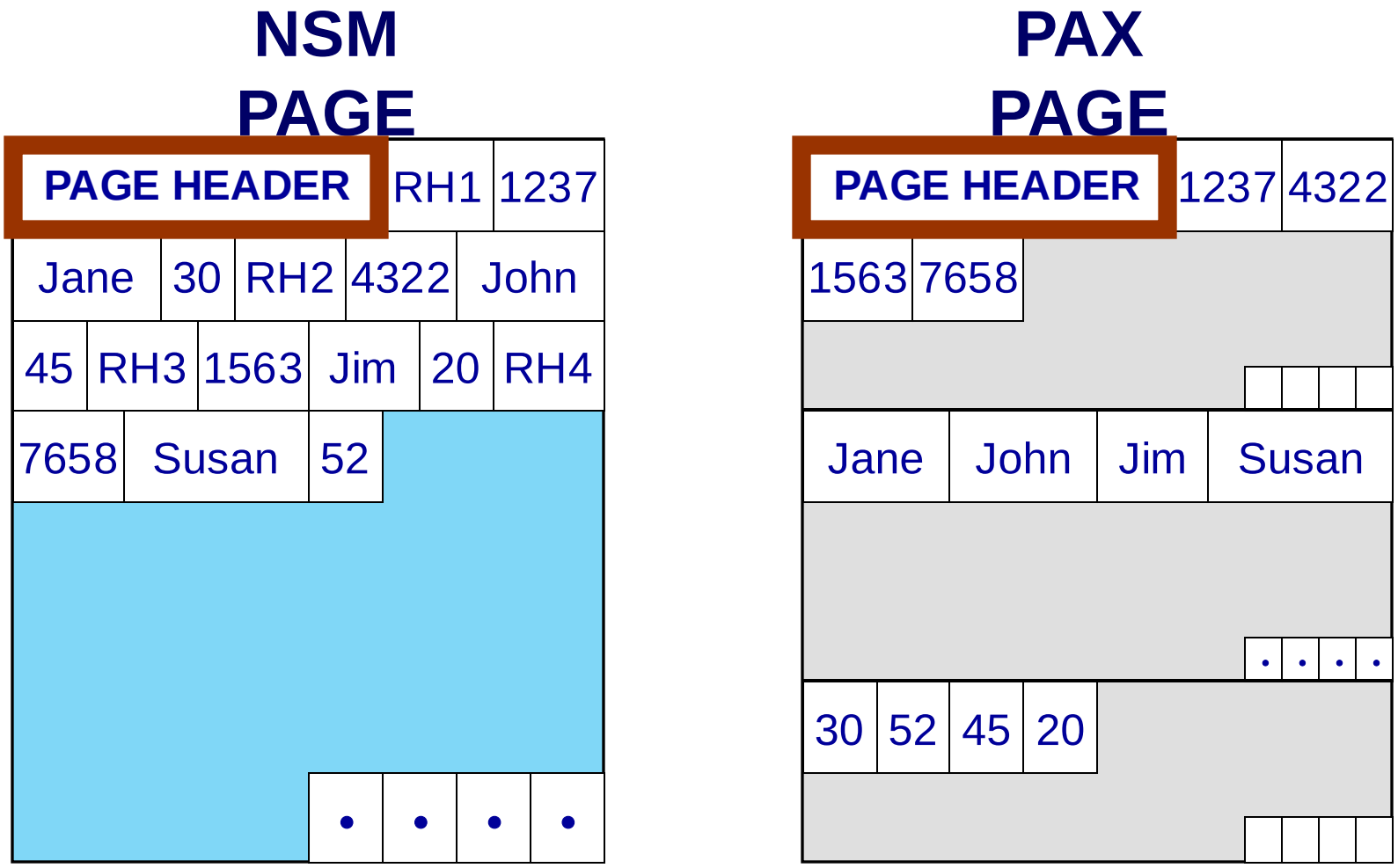
PAX PAGE



Partition data *within* the page for spatial locality

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

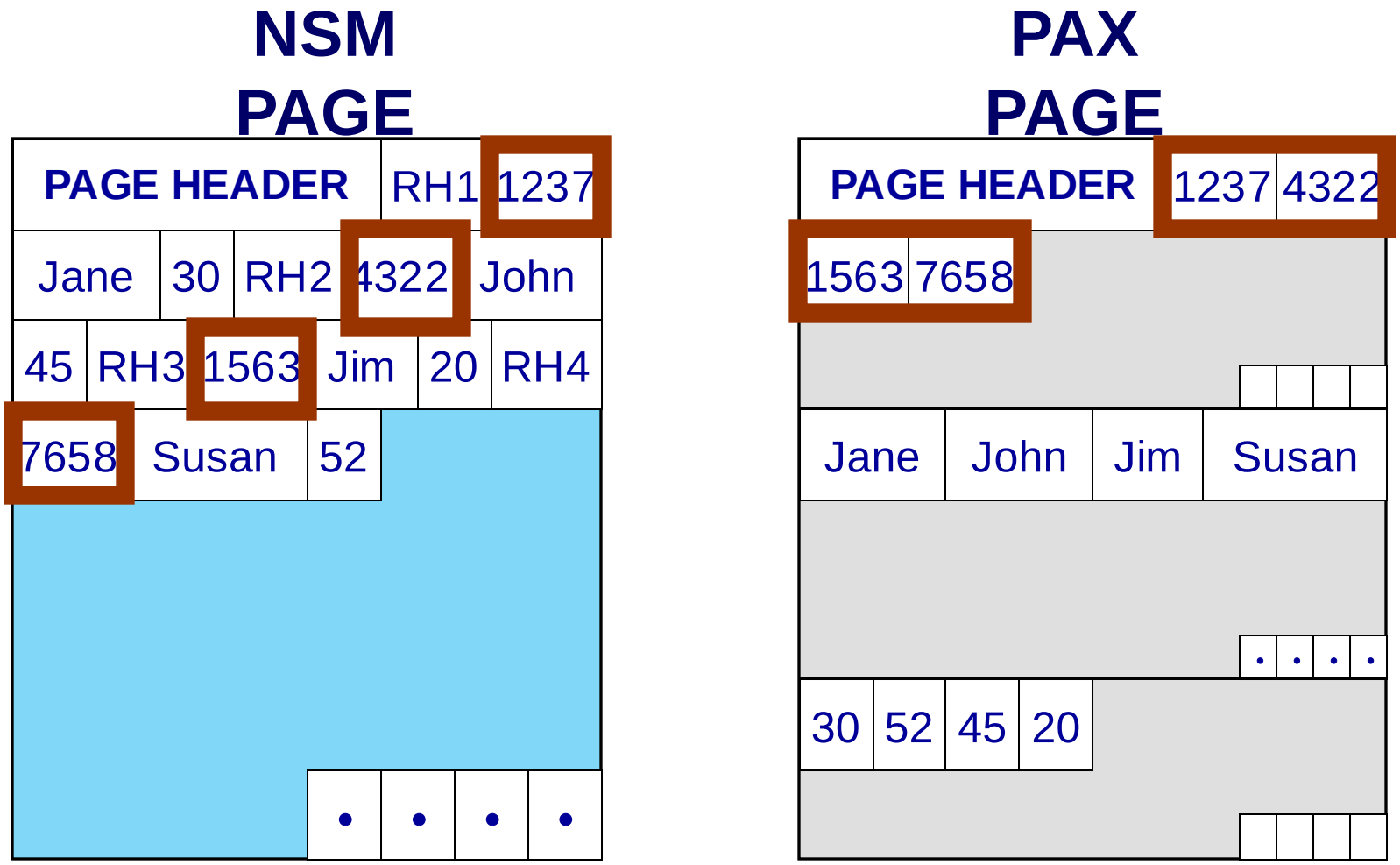
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

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

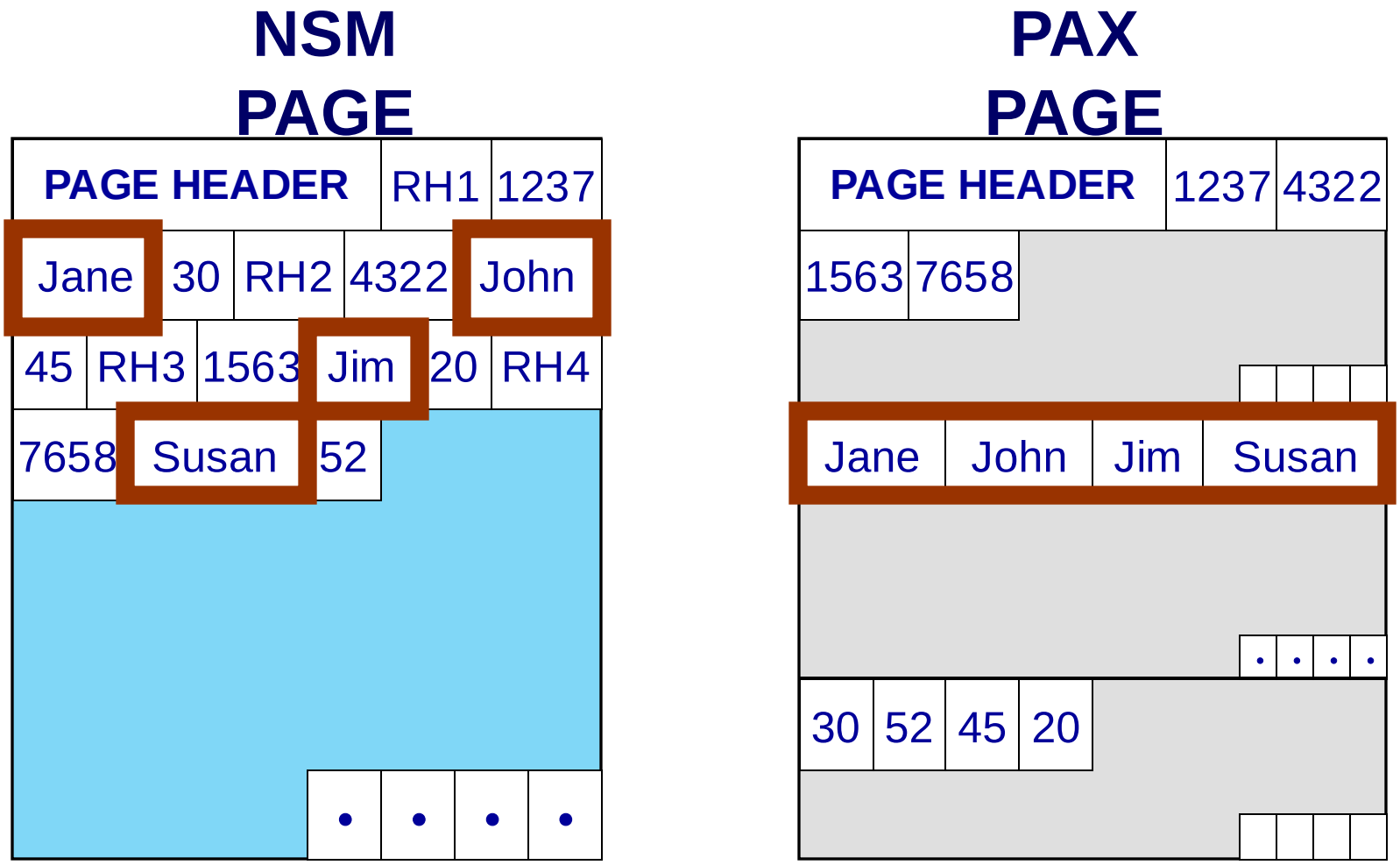
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

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

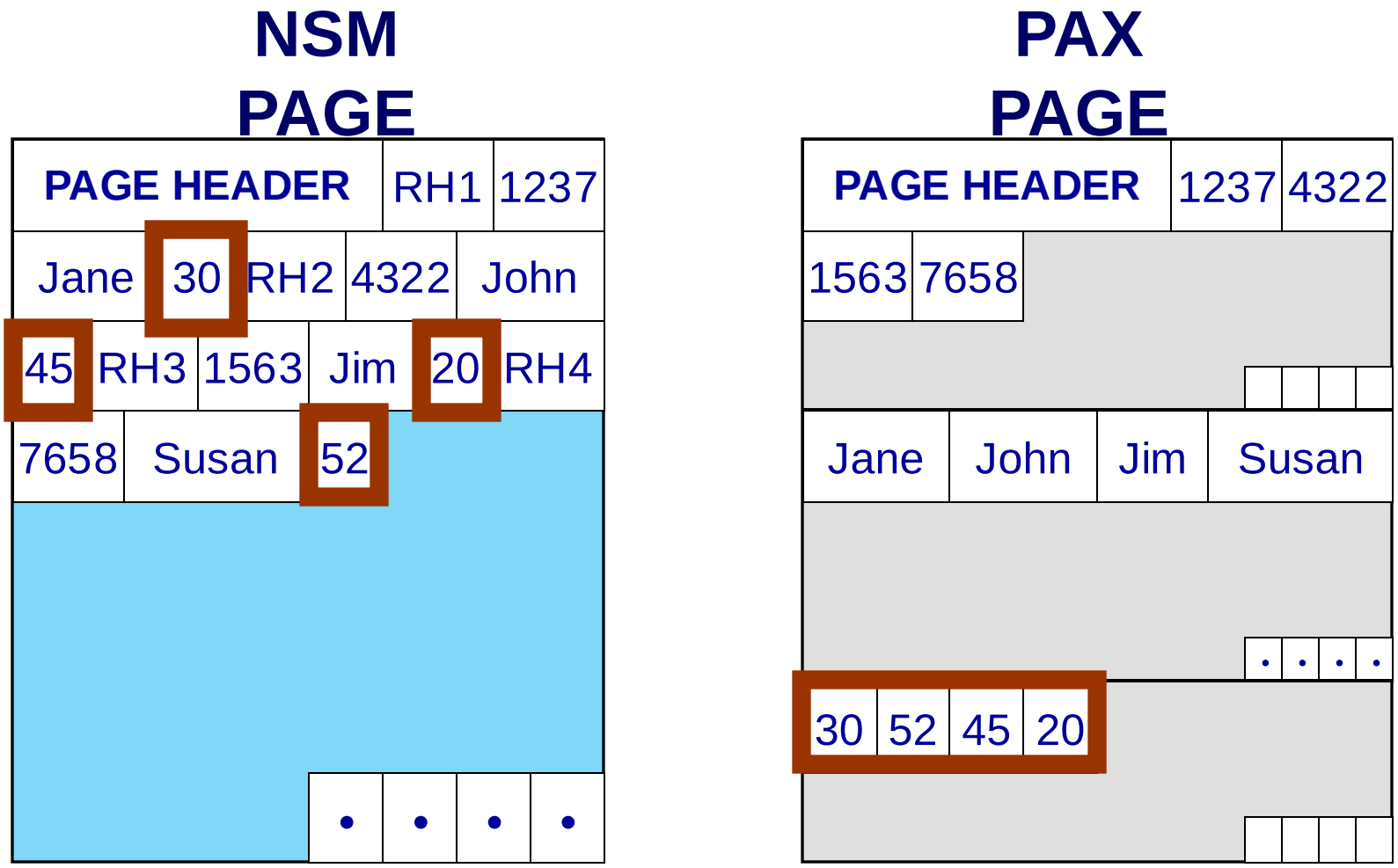
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

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

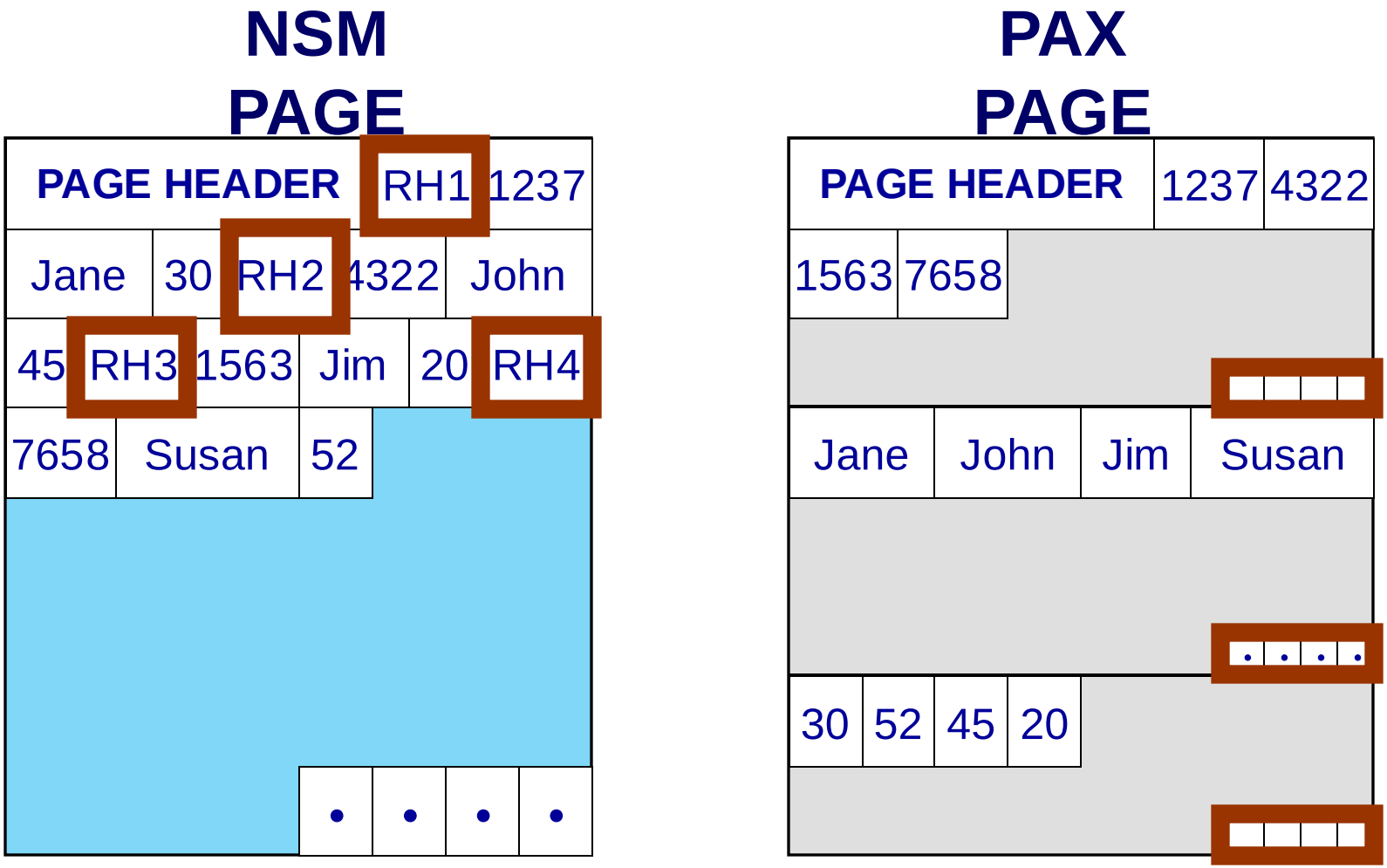
Partition Attributes Across (PAX)



Partition data *within* the page for spatial locality

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

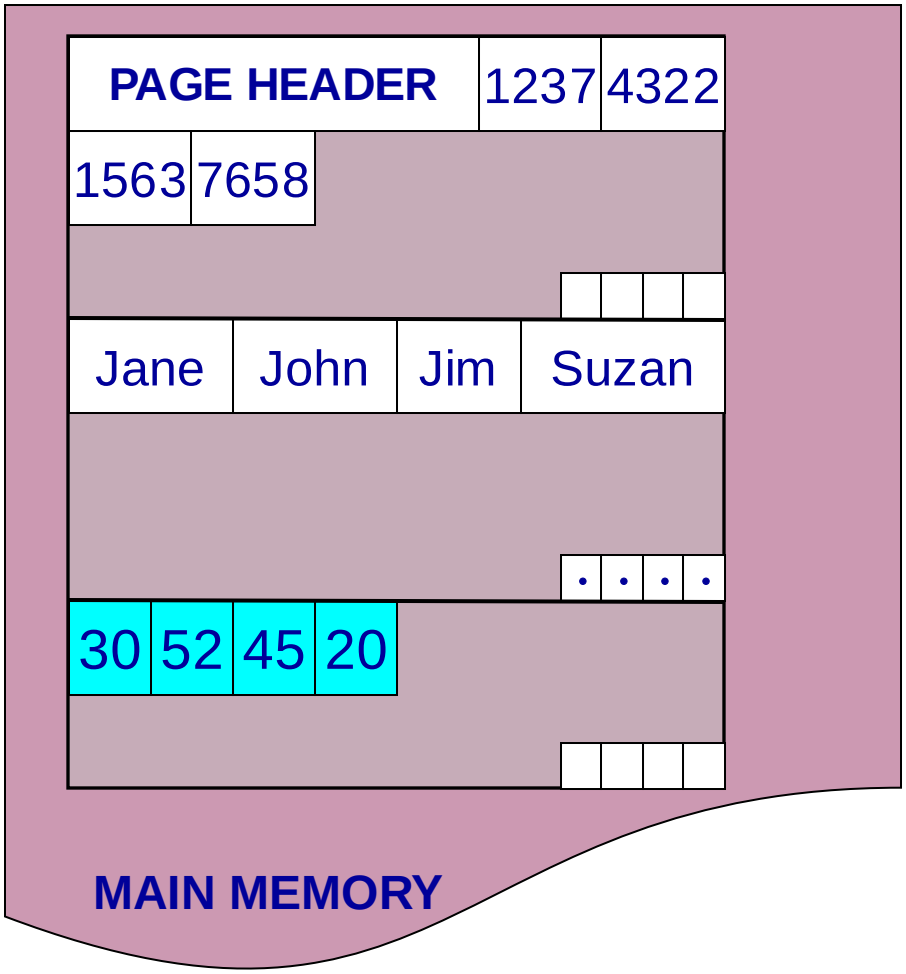
Partition Attributes Across (PAX)

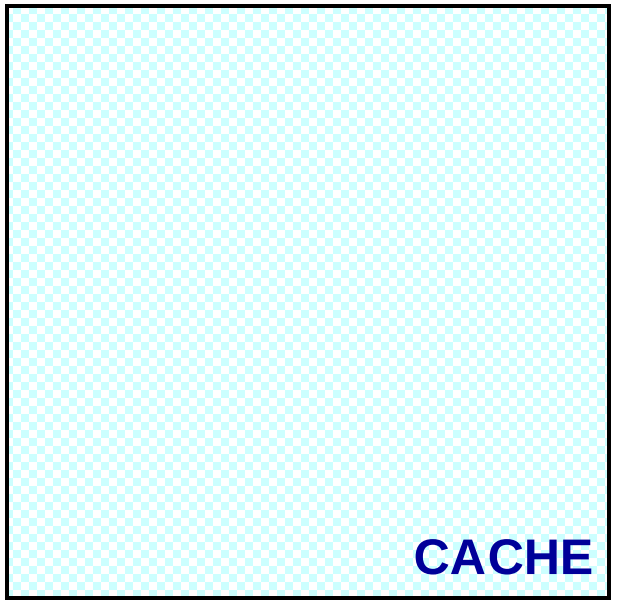


Partition data *within* the page for spatial locality

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

Predicate Evaluation using PAX



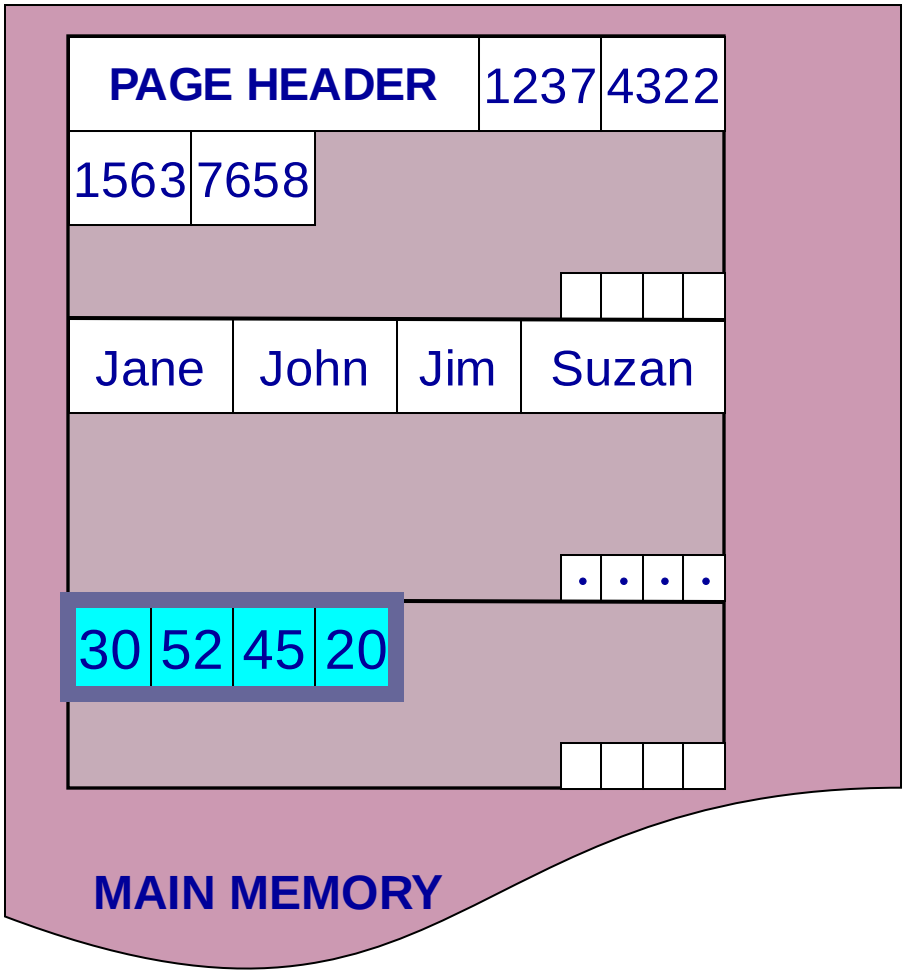


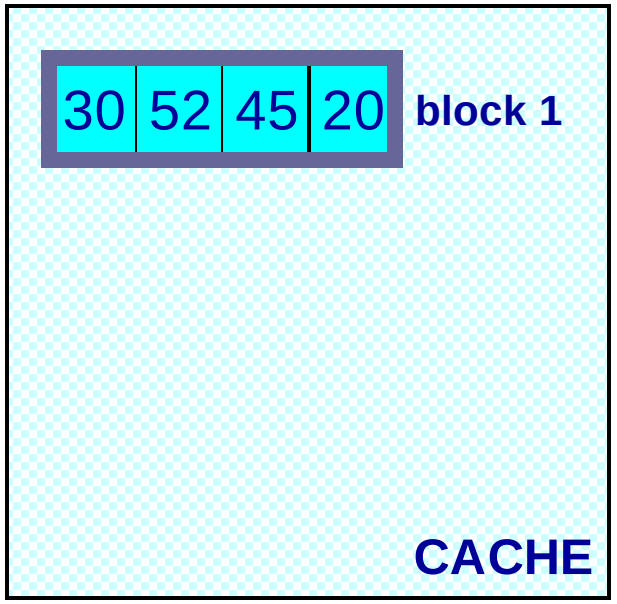

```
select name
from R
where age > 50
```

Fewer cache misses, low reconstruction cost

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

Predicate Evaluation using PAX

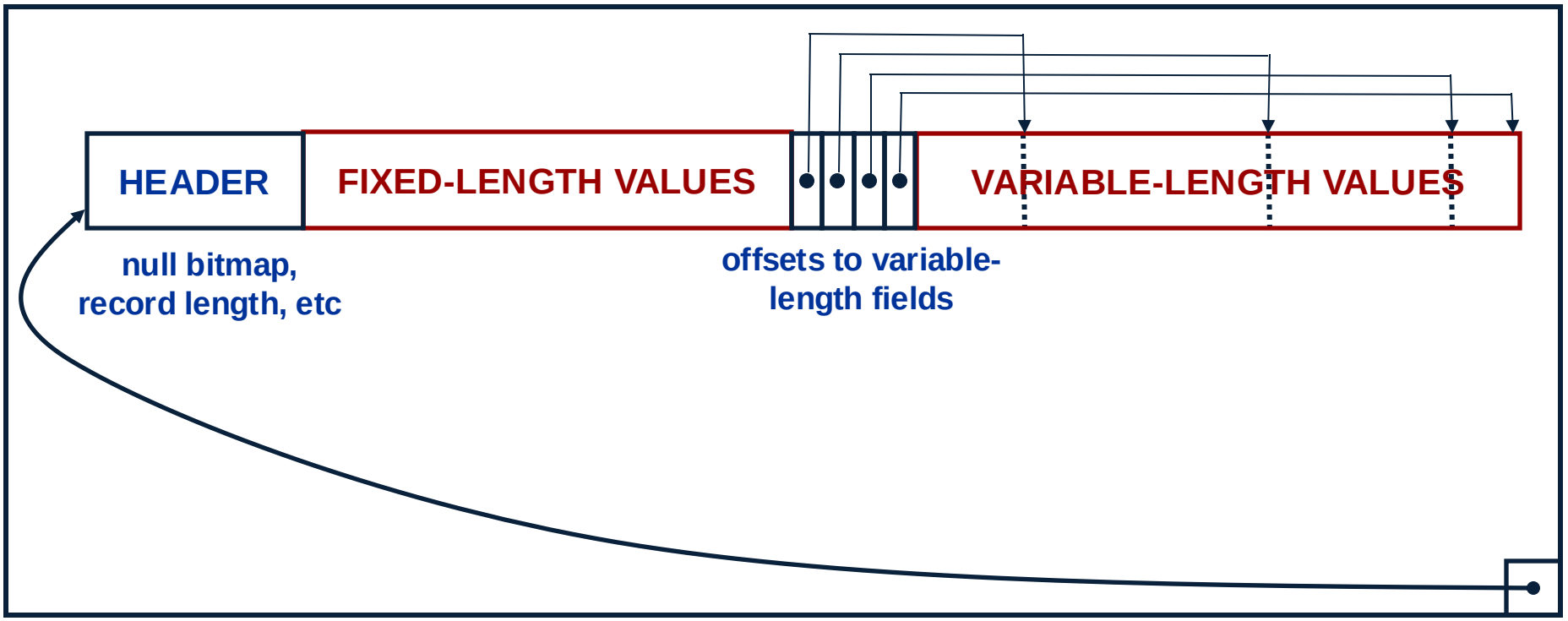





```
select name
from R
where age > 50
```

Fewer cache misses, low reconstruction cost

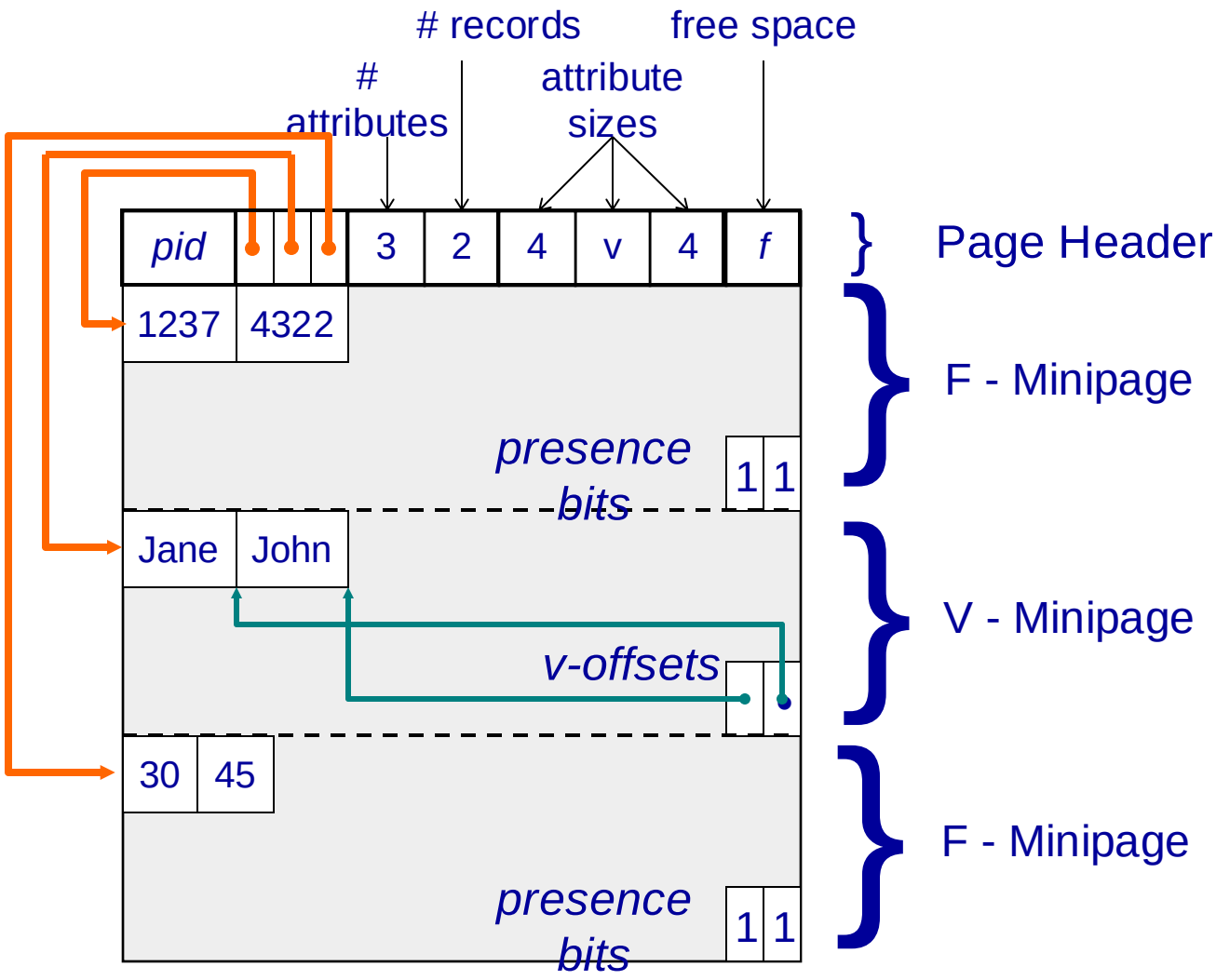
A Real NSM Record



NSM: All fields of record stored together + slots

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

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 CPU-memory latency
- Does not affect memory-disk latency
- No need to change the buffer manager
- Most DB engines use PAX today
- Snowflake uses PAX for cloud DB

Column-oriented DB

Column-oriented Storage

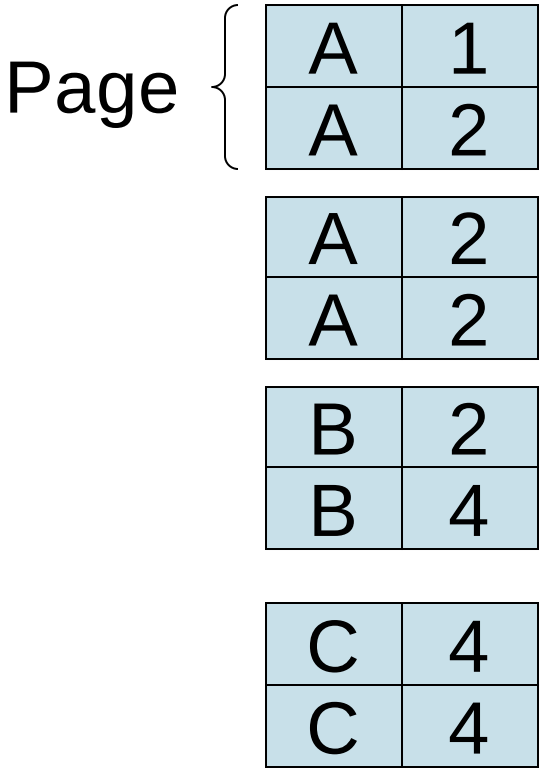
- Store each attribute in a different file

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

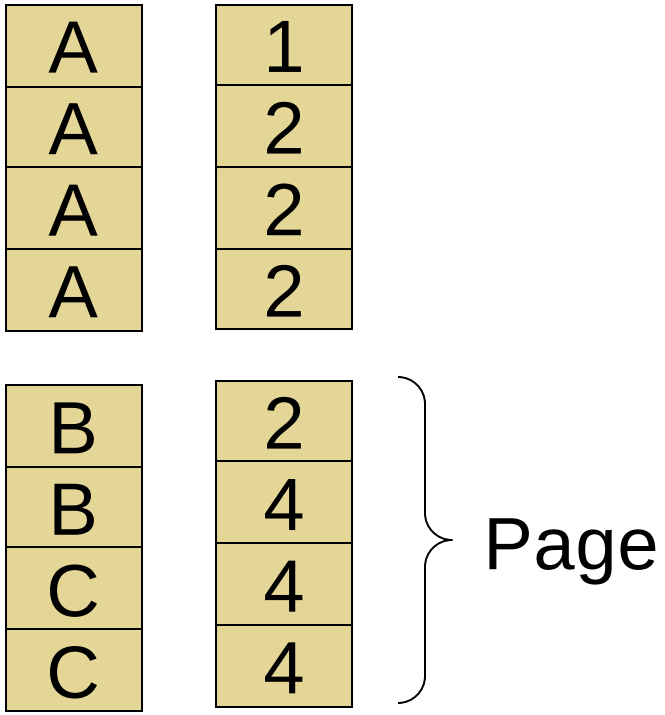
The Design and Implementation of Modern Column-Oriented Database Systems, Abadi et al., 2013

Column-oriented Storage

Row-based
(4 pages)



Column-based
(4 pages)



C-Store also avoids large tuple headers

From Row to Column Storage (Modern Designs)

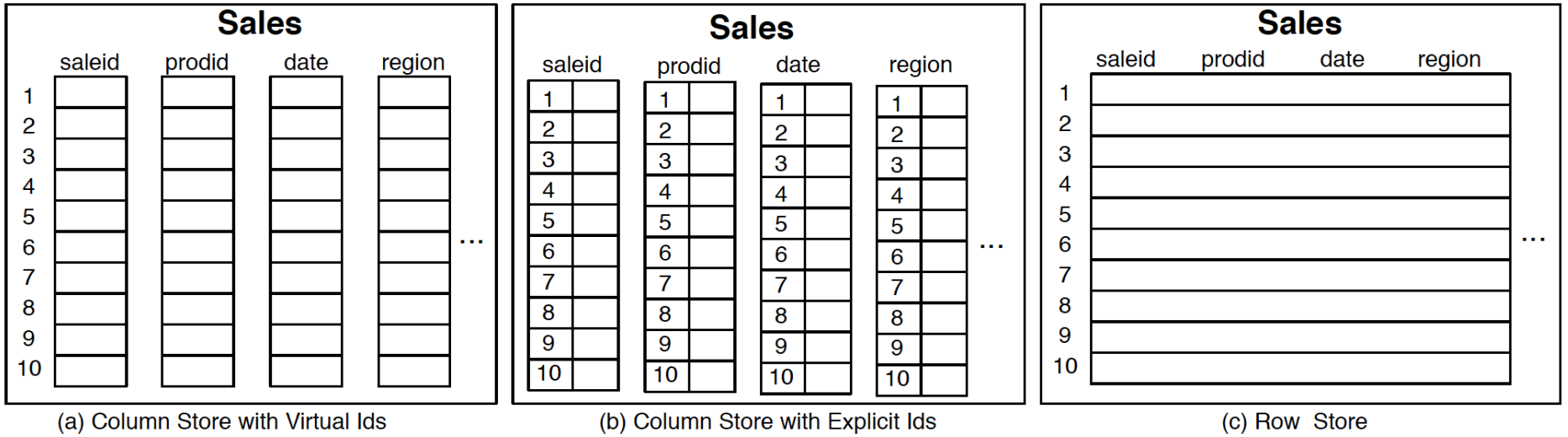


Figure 1.1: Physical layout of column-oriented vs row-oriented databases.

Basic tradeoffs:

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

Column-oriented Storage

- Improves the disk-memory latency
- Requires full rewriting of the DBMS

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

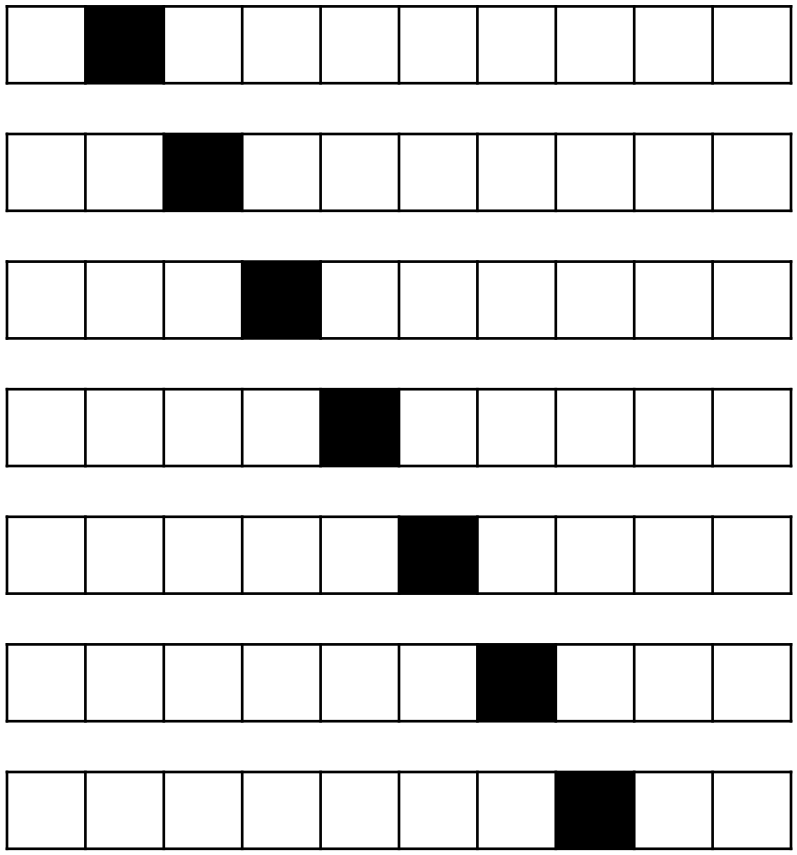
Access Path

Access Path

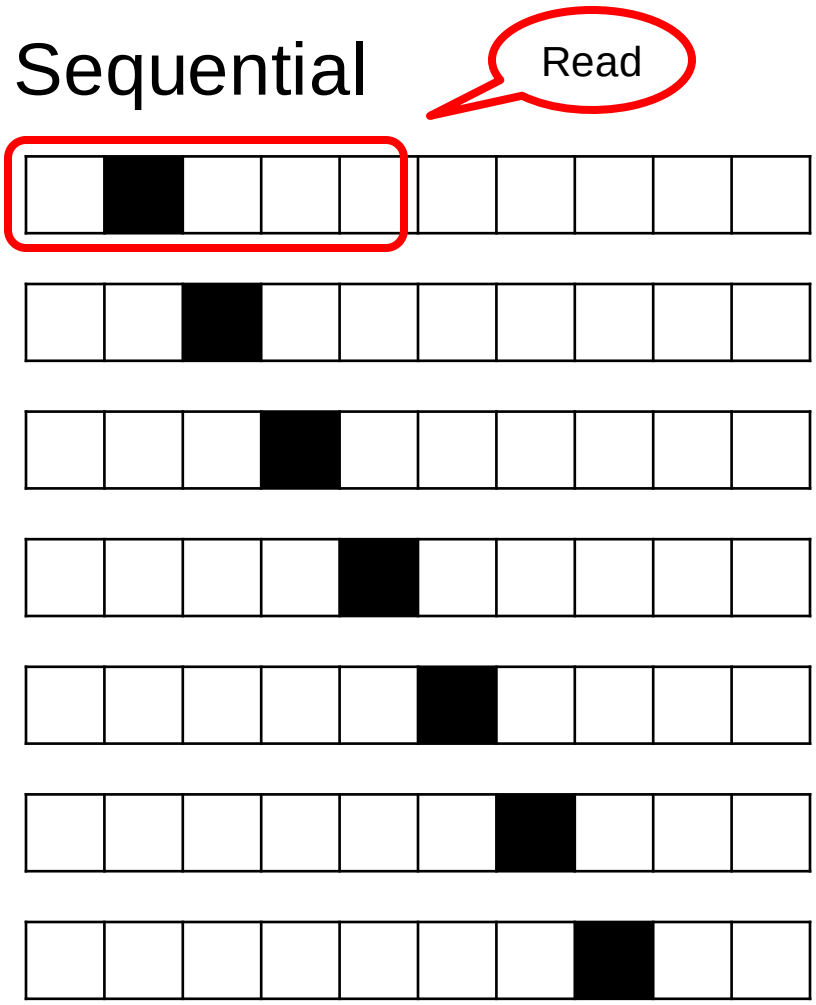
- Sequential scan:
  - Useful when we read most records in R
  - SELECT * FROM R;
- Random access:
  - Useful when we have a filter predicate
  - SELECT * FROM R WHERE R.A=55;

Sequential/Random Access

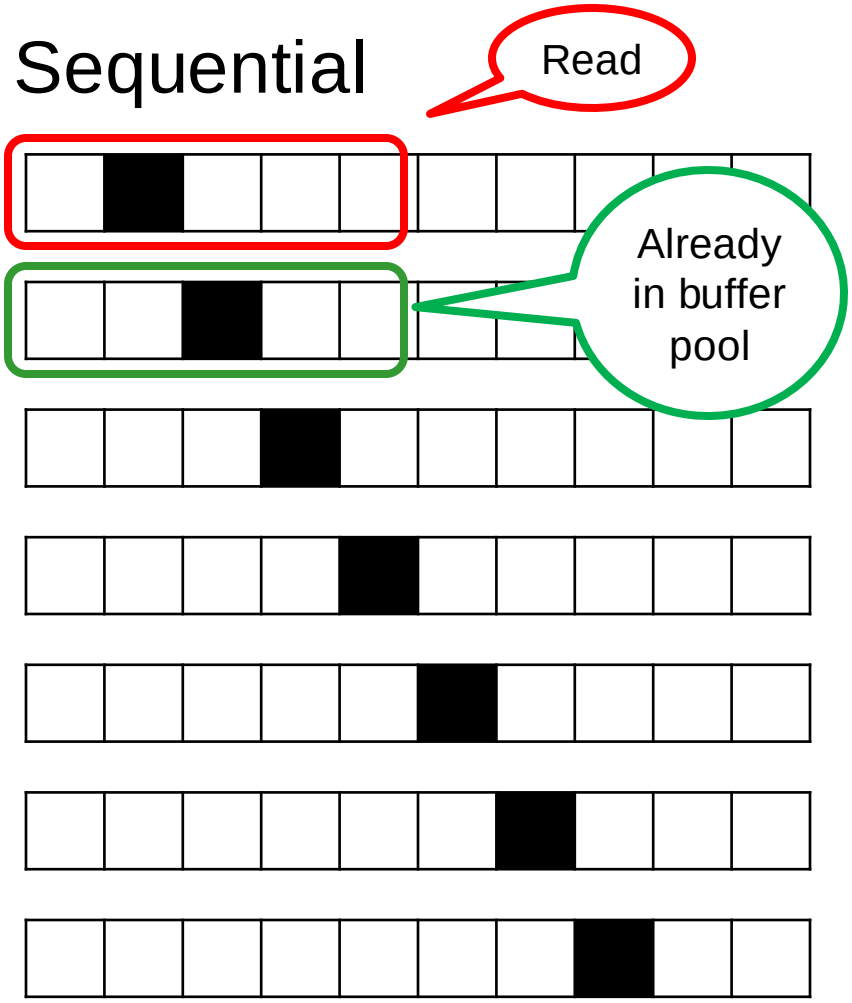
Sequential



Sequential/Random Access

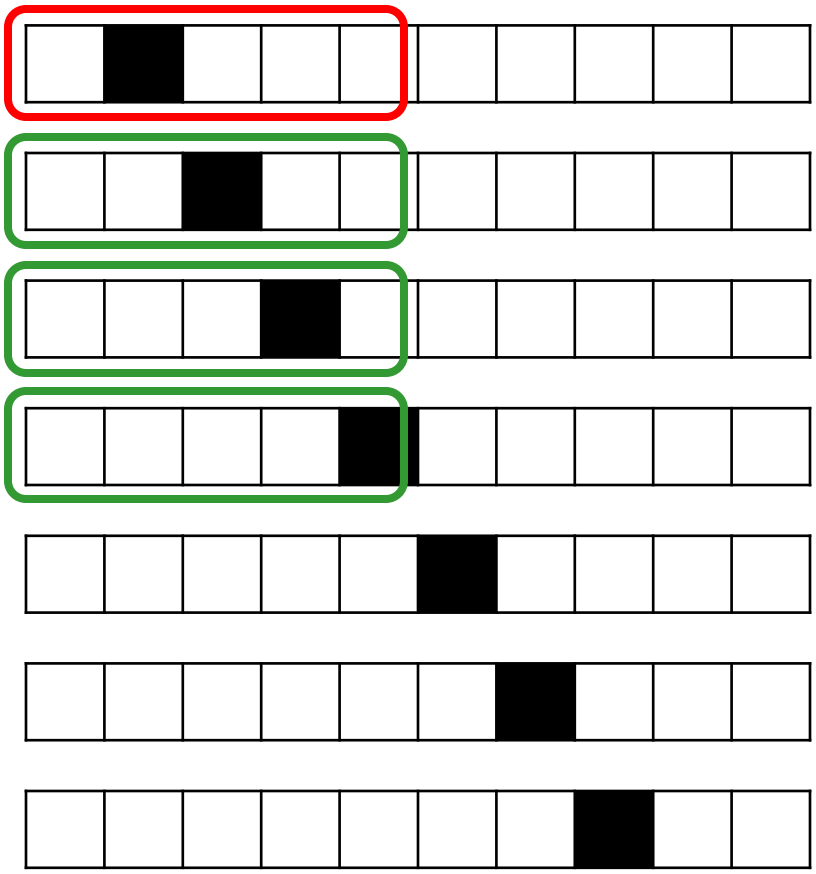


Sequential/Random Access



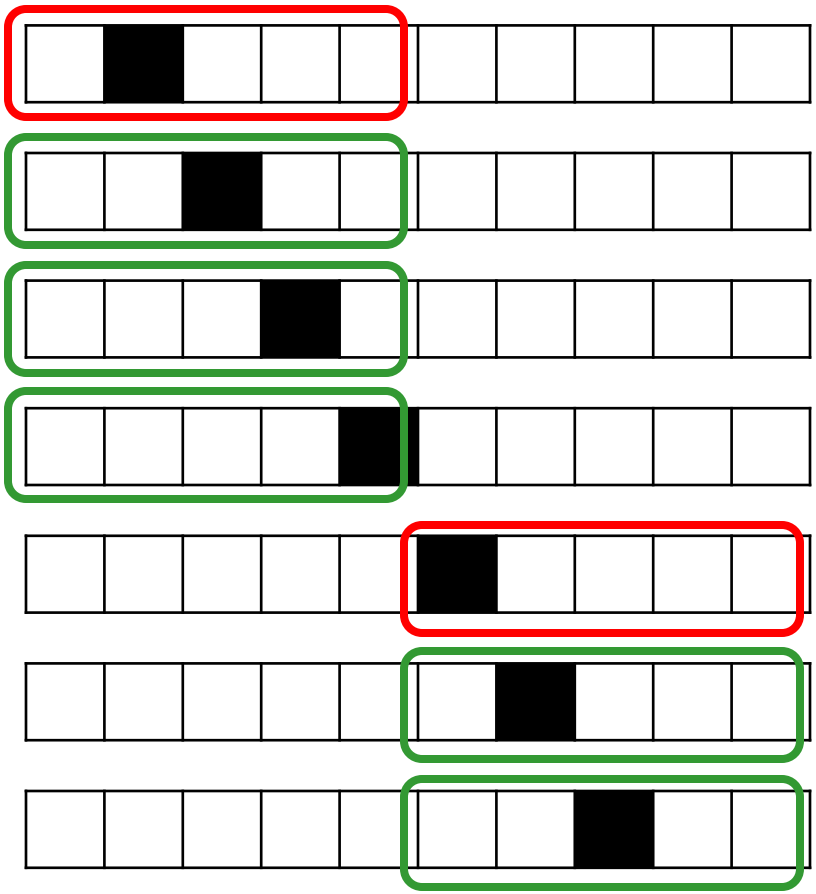
Sequential/Random Access

Sequential



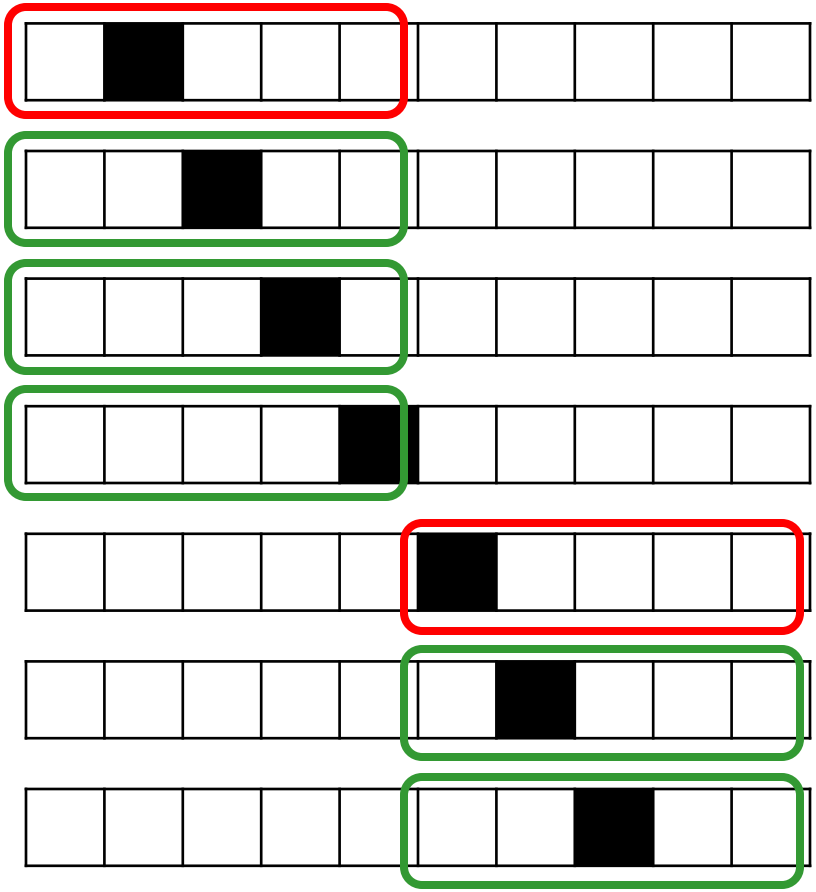
Sequential/Random Access

Sequential: 2 reads

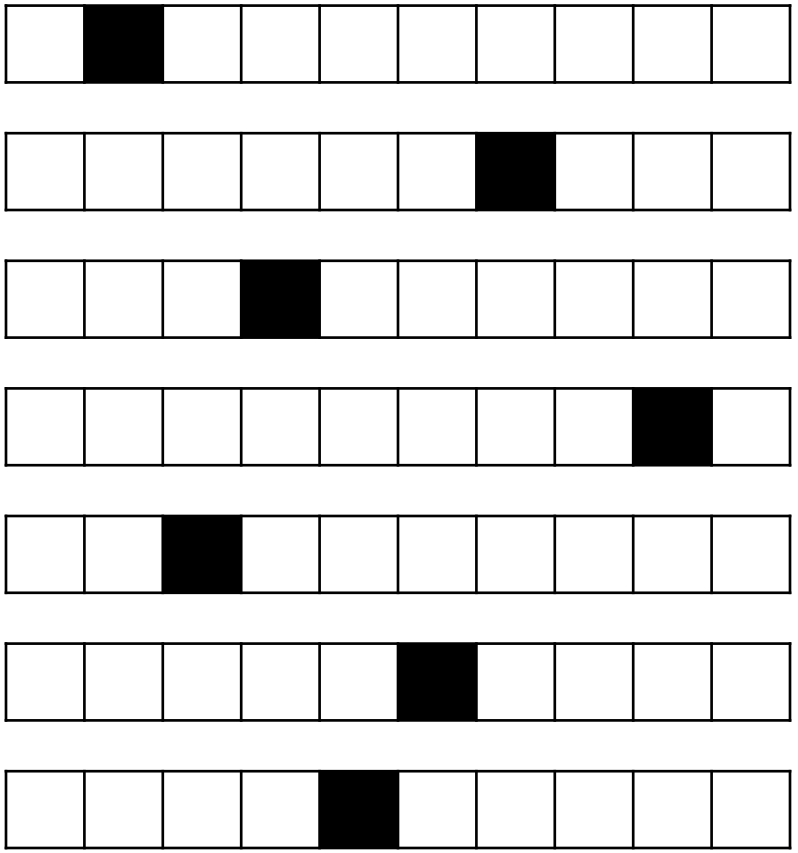


Sequential/Random Access

Sequential: 2 reads

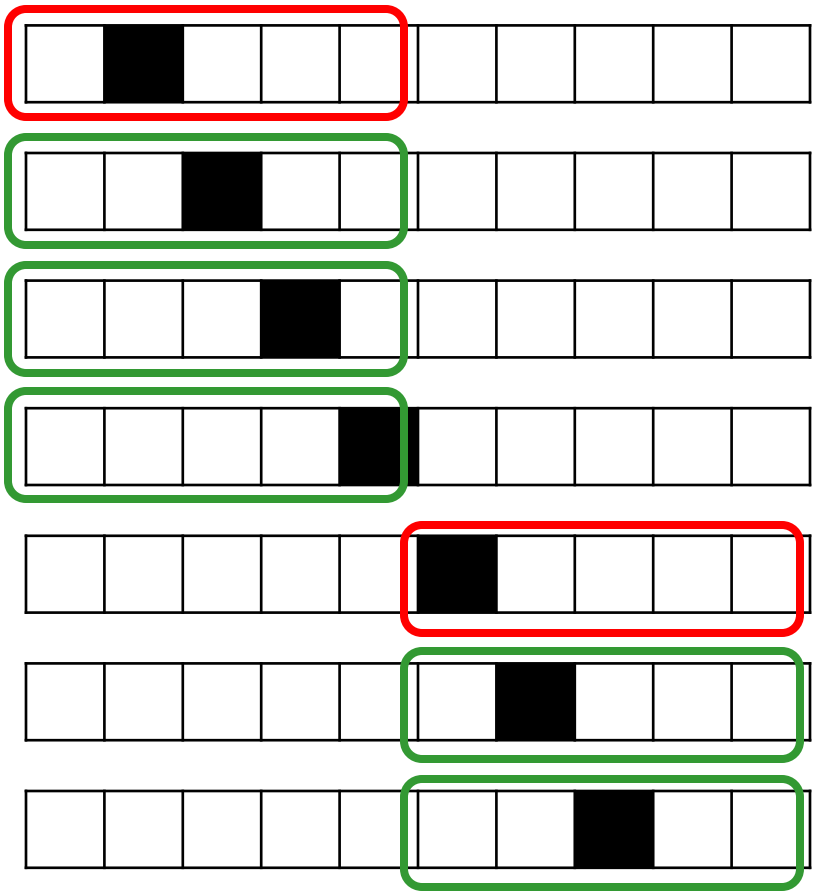


Random

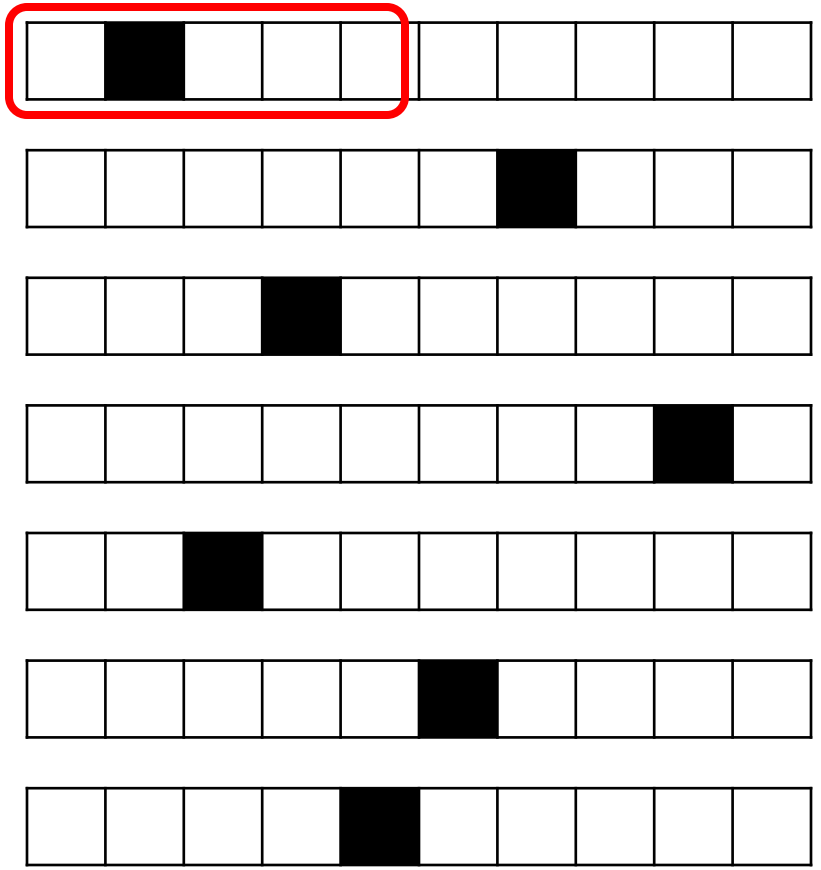


Sequential/Random Access

Sequential: 2 reads

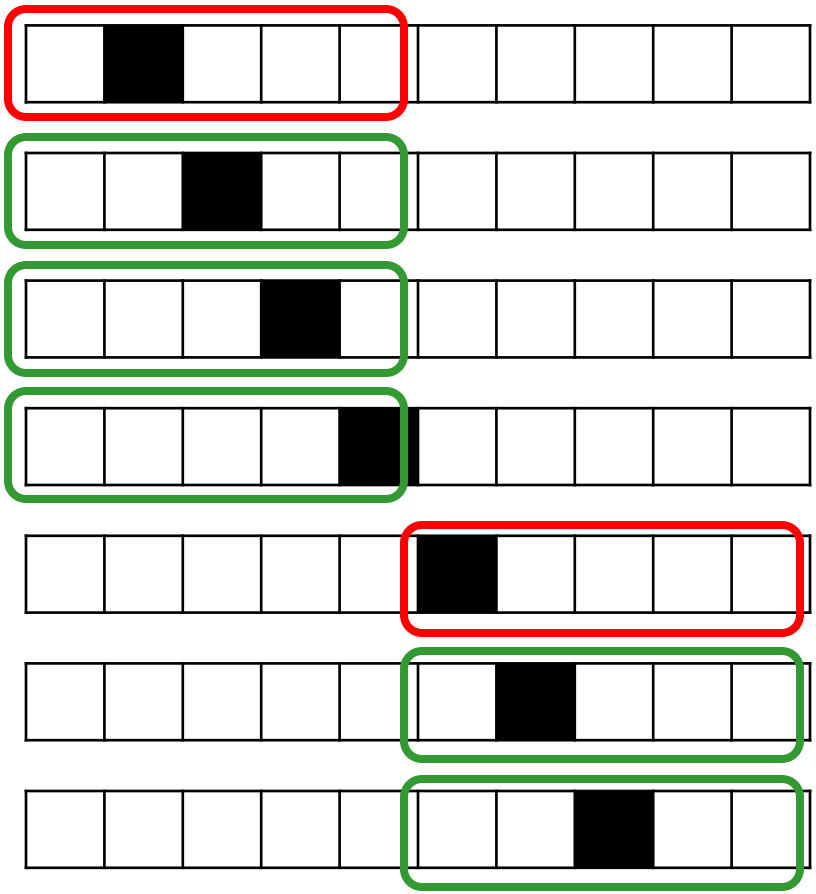


Random

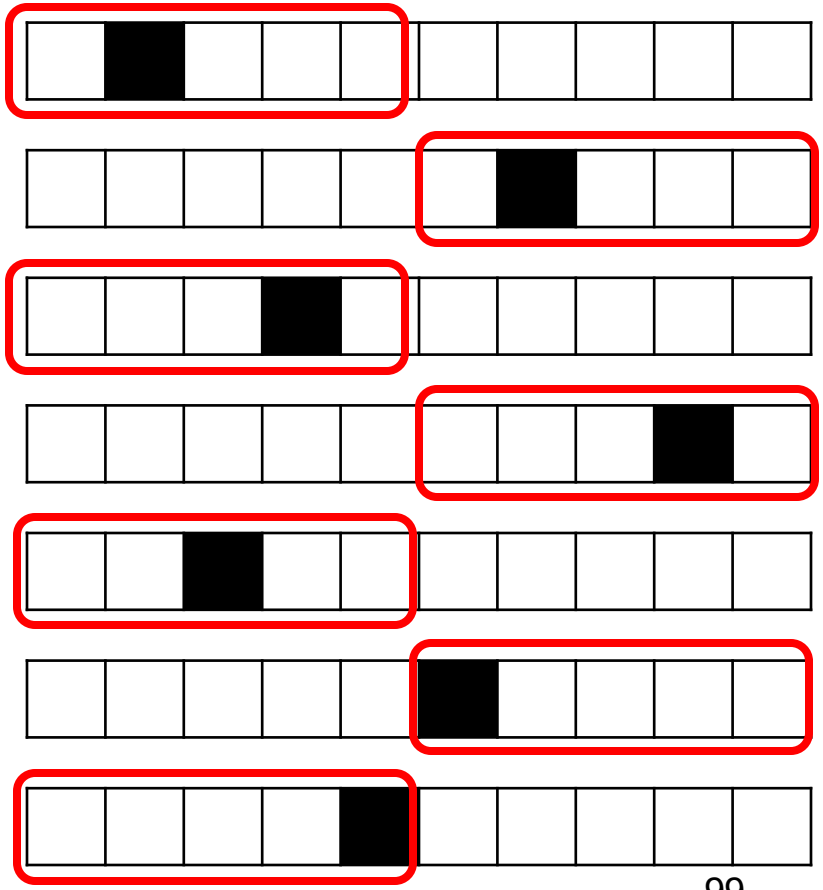


Sequential/Random Access

Sequential: 2 reads



Random: many reads



Discussion

- Sequential scan: best for reading large amounts of data
- Random access: best for reading a small amount of data
  - Need to know which block contains it
  - Indexes help us find that block

B+ Trees

Binary Search Tree

- Sorted arrays: best for main-memory search, but they cannot handle updates

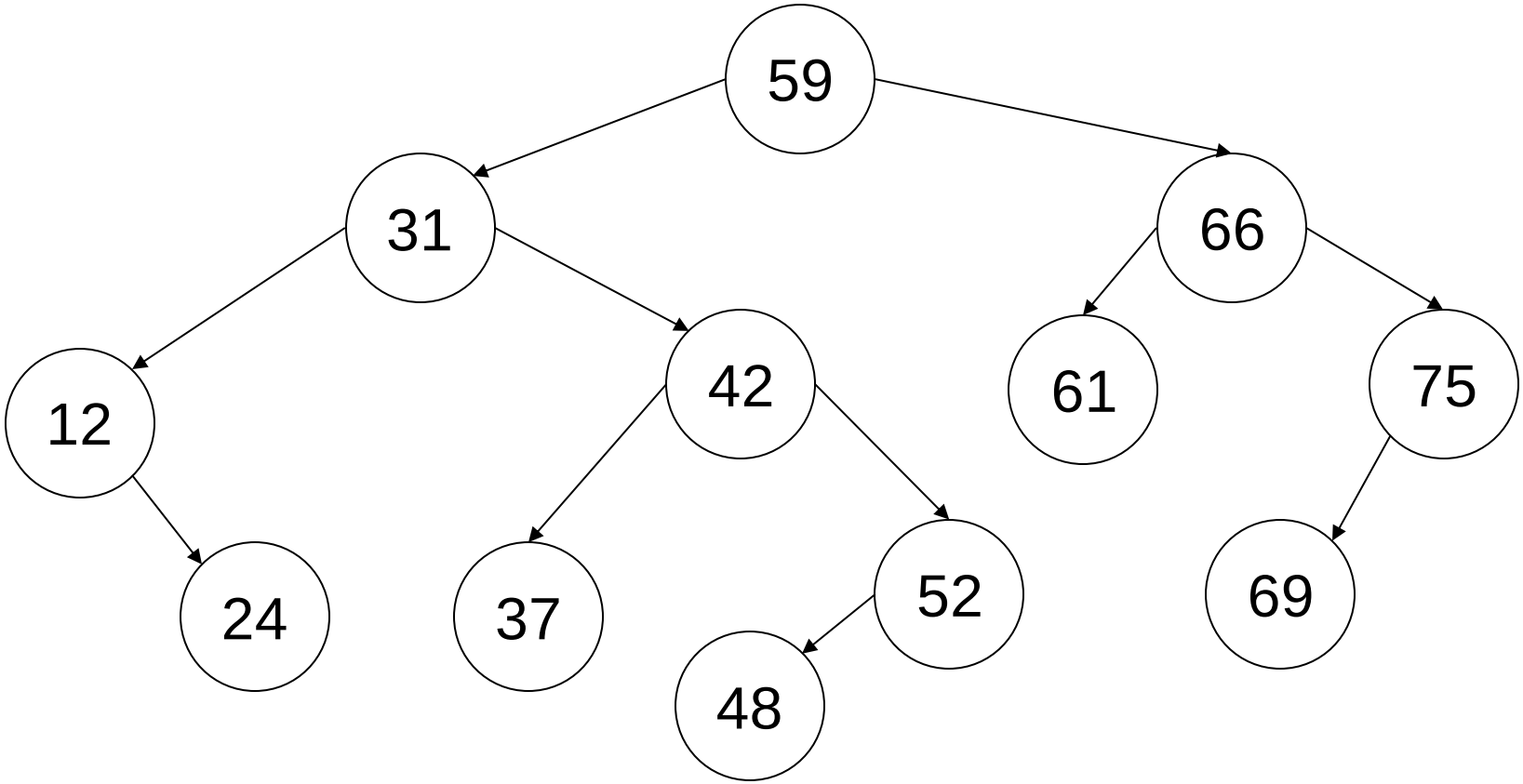
- Binary search trees:
  - Keep the binary search idea
  - Allow for updates

Binary Search Tree

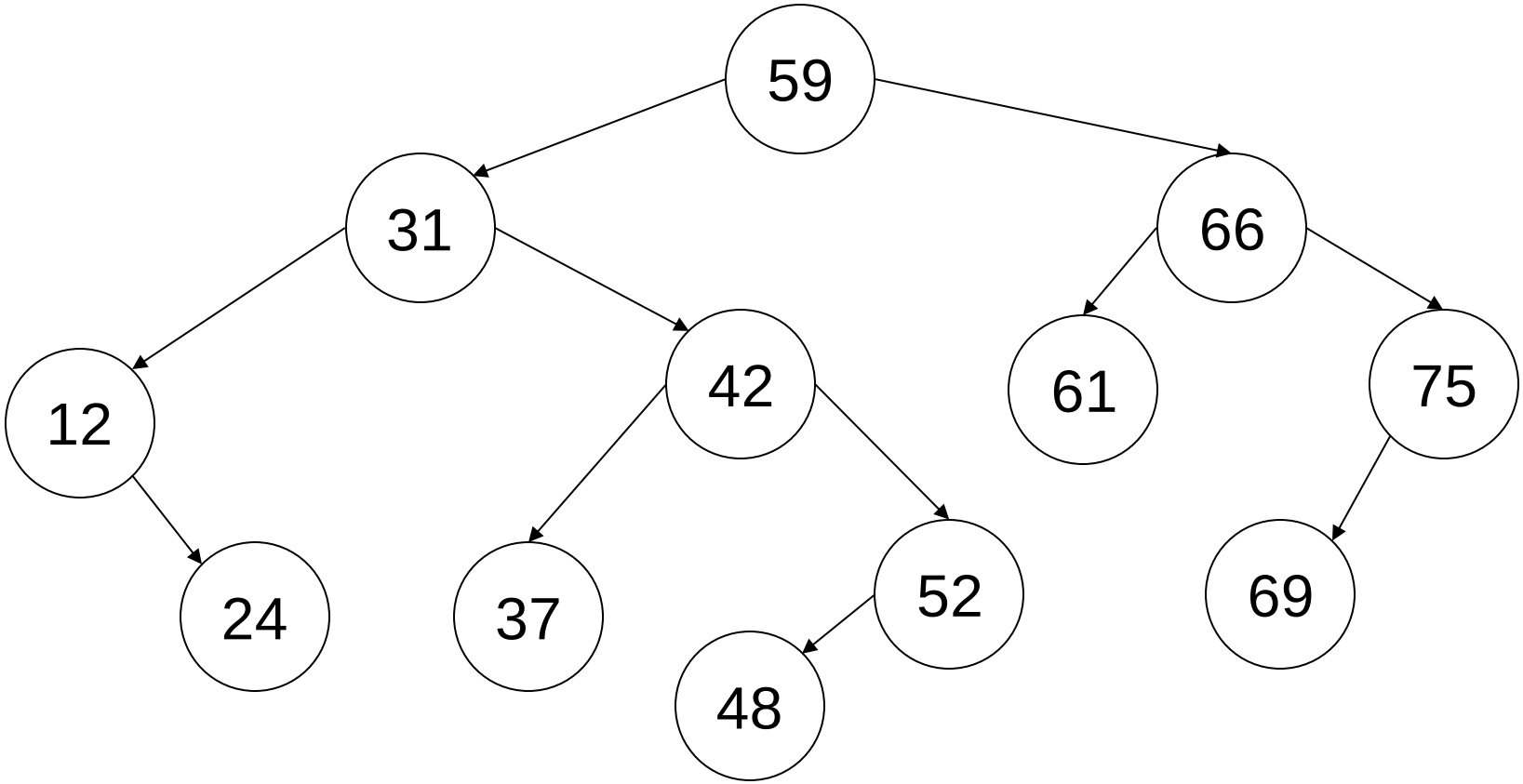
12	24	31	37	42	48	52	59	61	66	69	75

Binary Search Tree

12	24	31	37	42	48	52	59	61	66	69	75

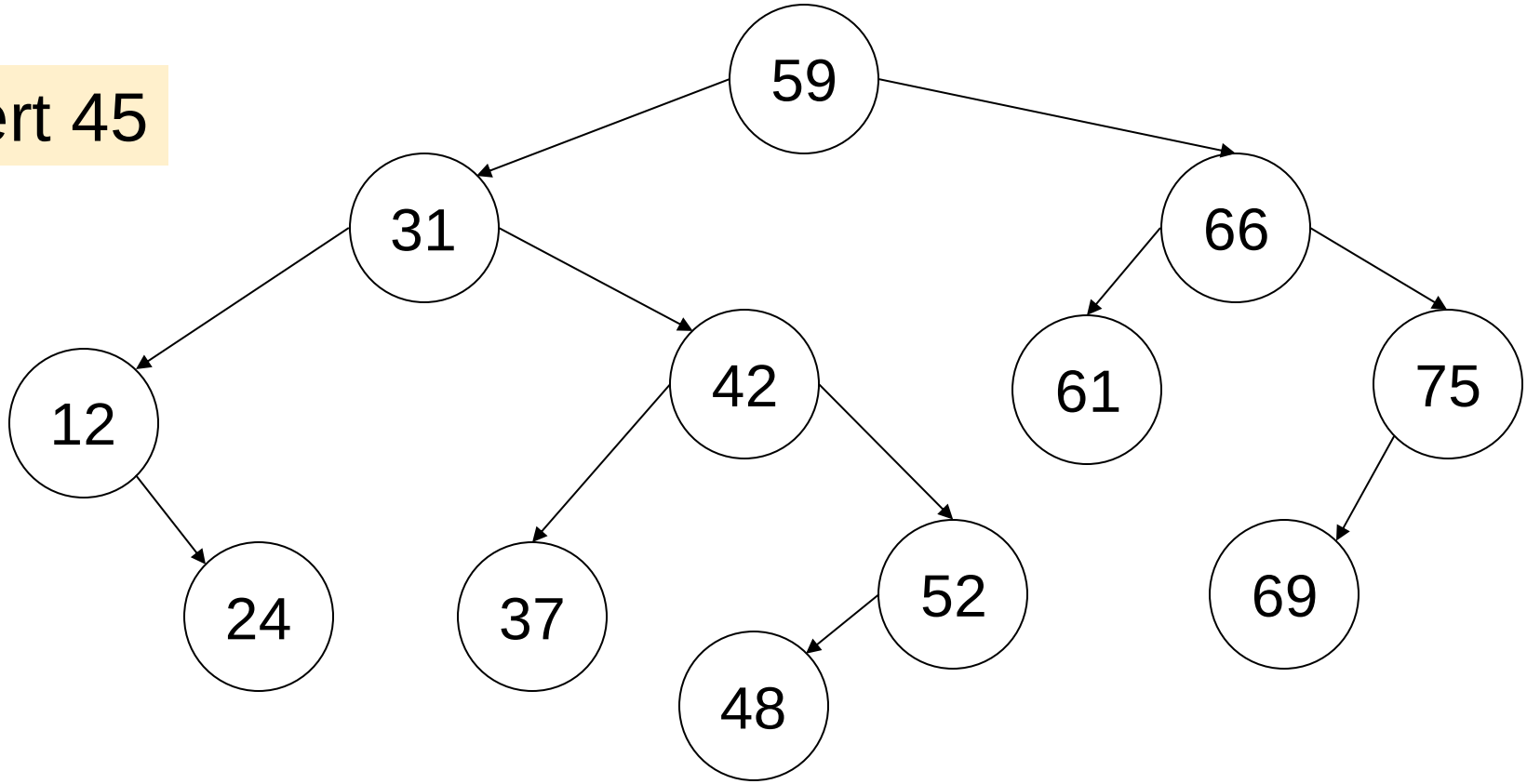


Binary Search Tree



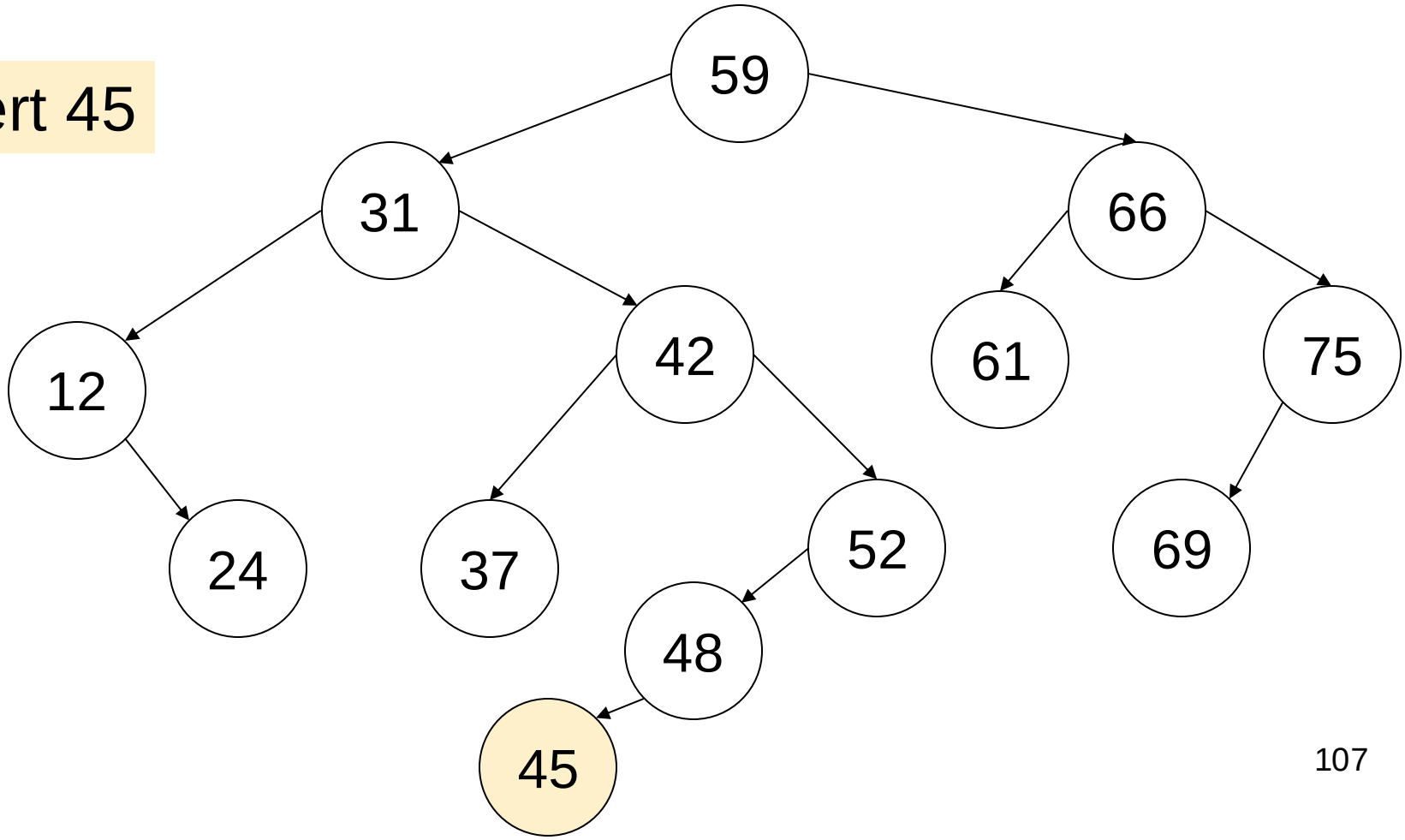
Binary Search Tree

Insert 45



Binary Search Tree

Insert 45

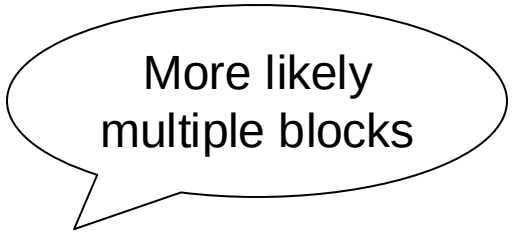


Binary Search Tree

- Need to be balanced: depth=O(log N)
- Various methods to rebalance:
  - Red/black trees, splay trees, …
- Time for search/insert/delete = O(log N)

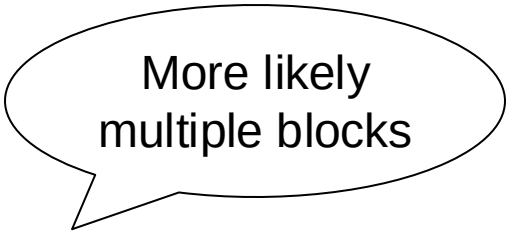
But not good for disk -- why??

B+ Trees



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

B+ Trees

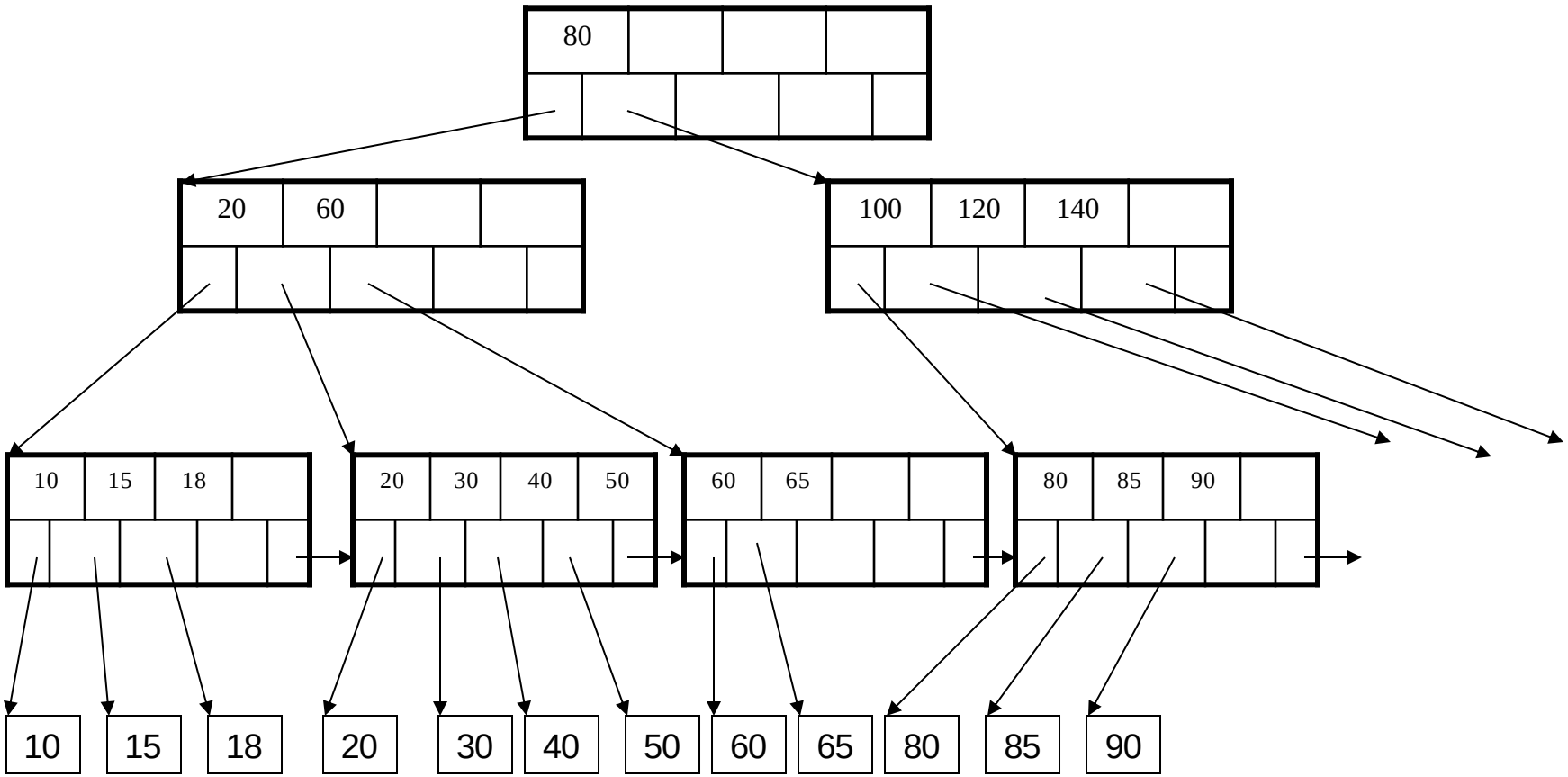


- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Store multiple keys and children
  - Read 1 page, use many keys

- 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

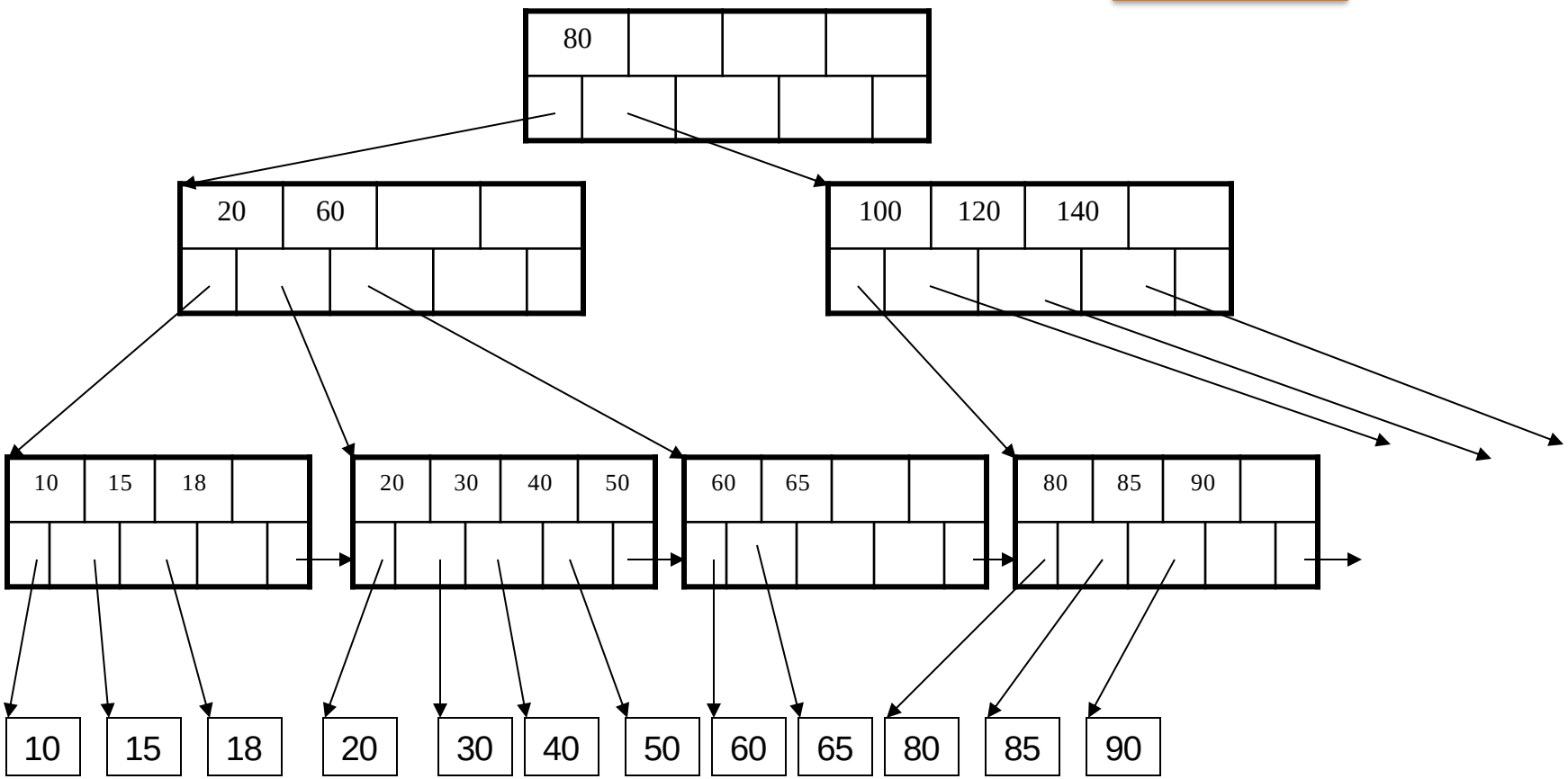
d = 2



B+ Tree Example

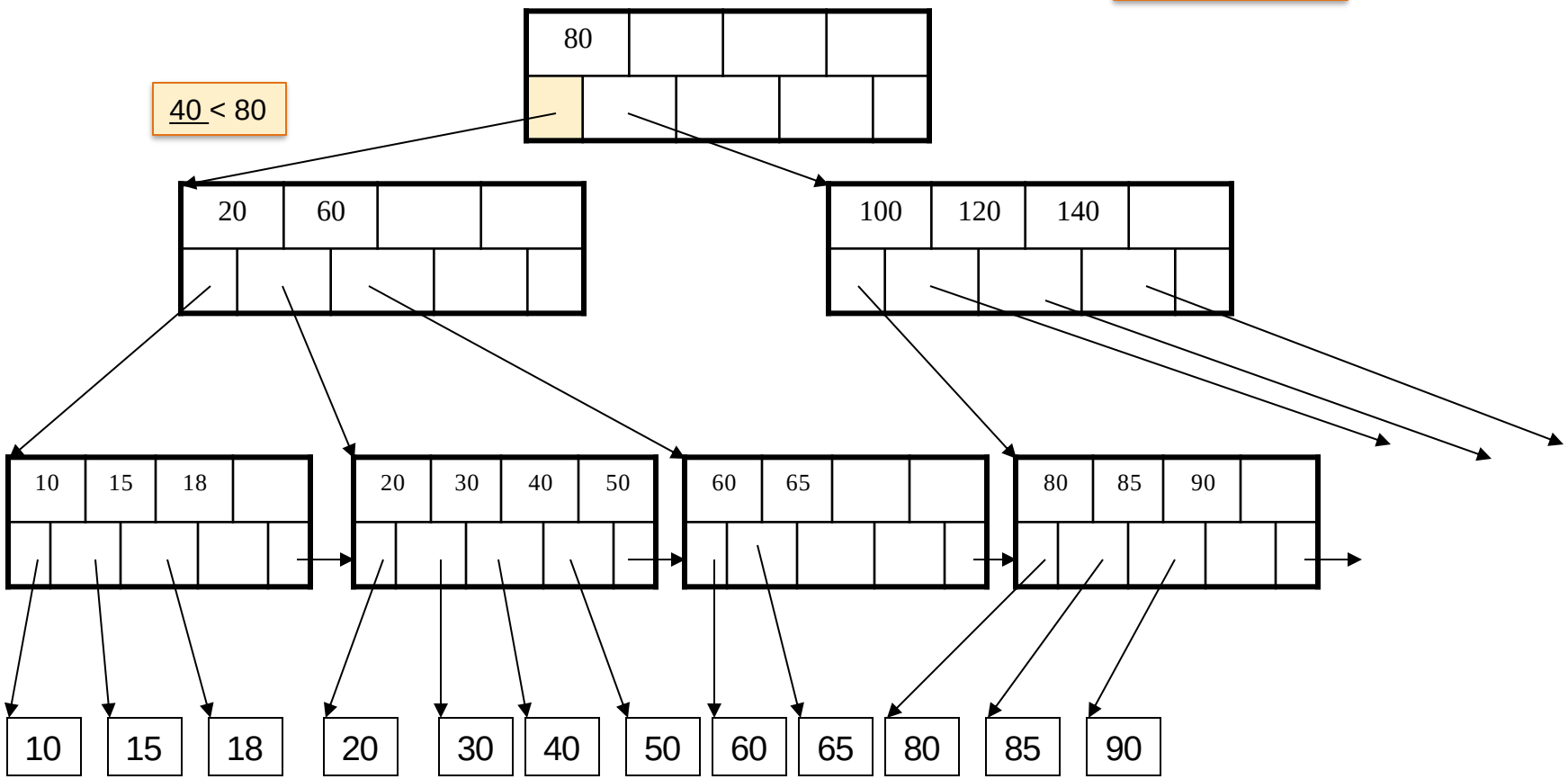
d = 2

Find the key 40



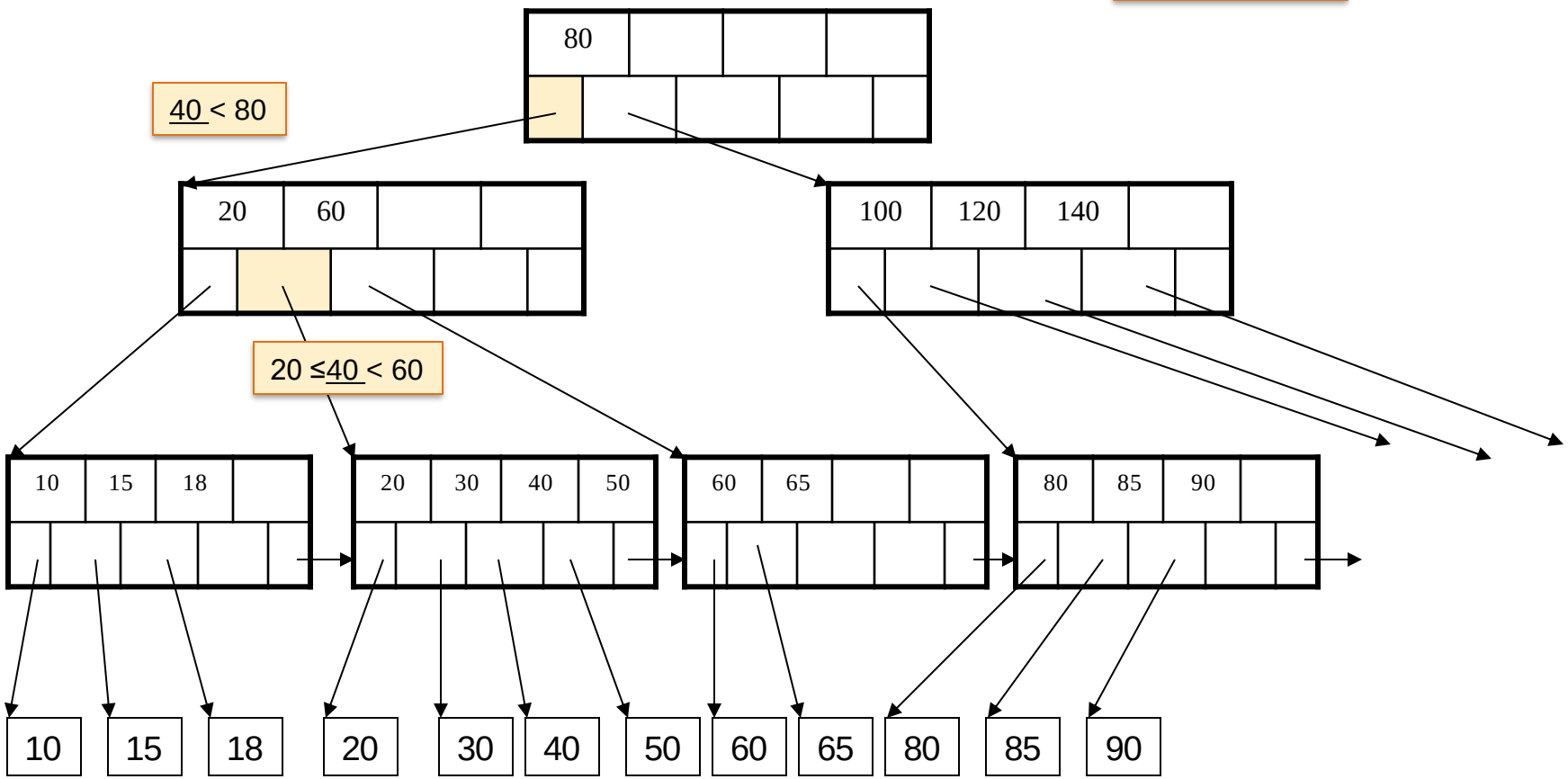
B+ Tree Example

d = 2



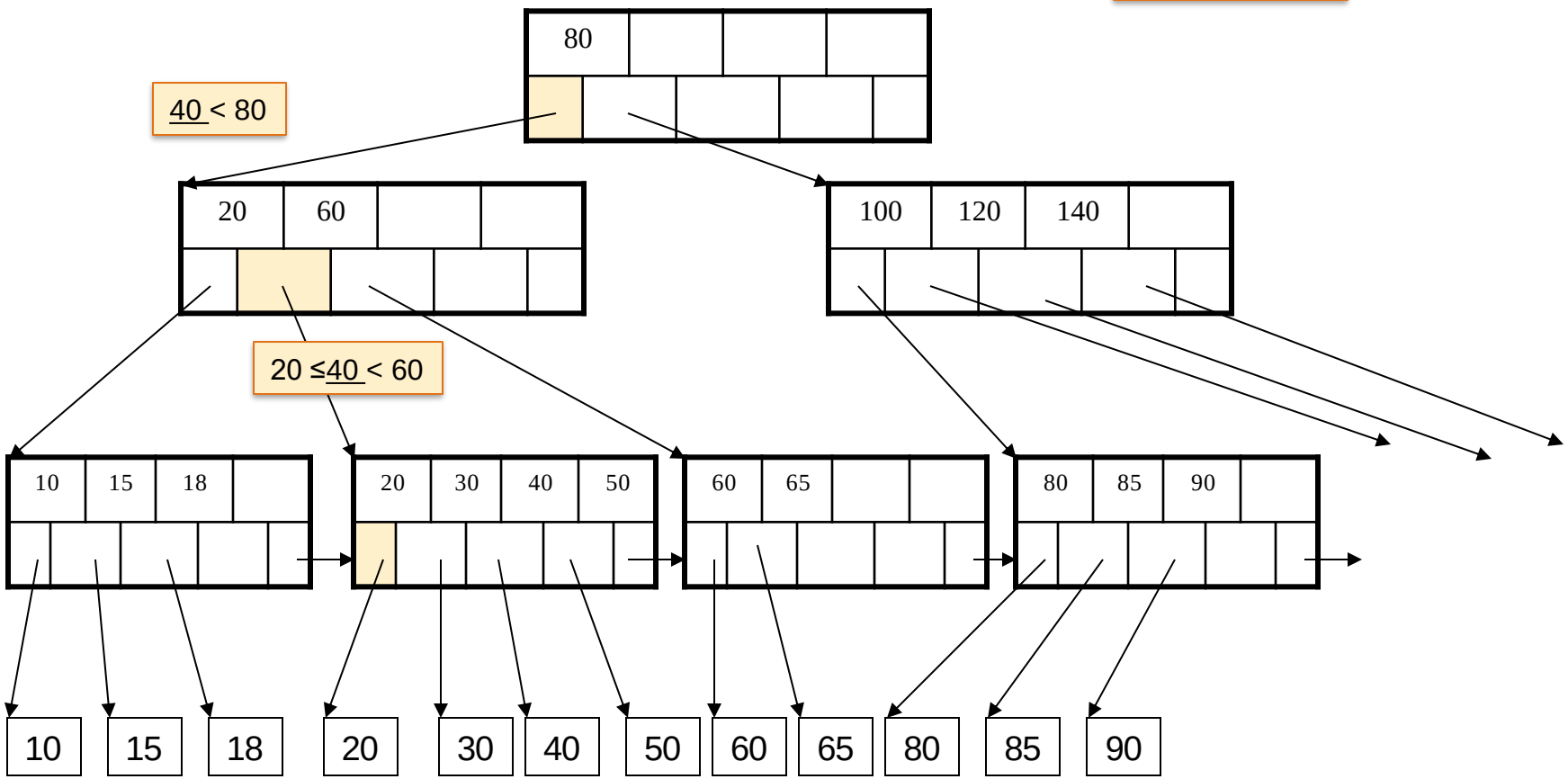
B+ Tree Example

d = 2



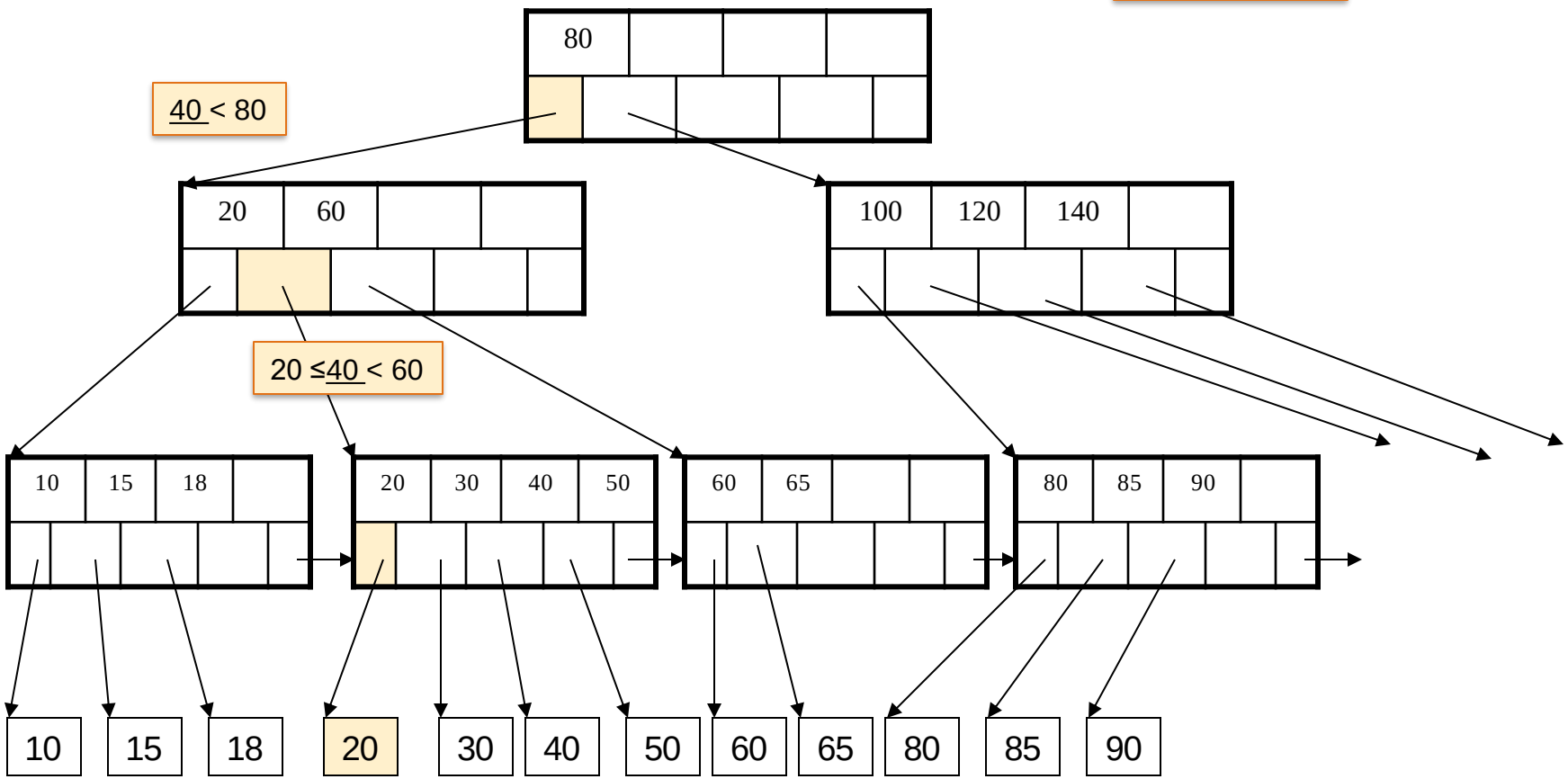
B+ Tree Example

d = 2



B+ Tree Example

d = 2



B+ Trees Properties

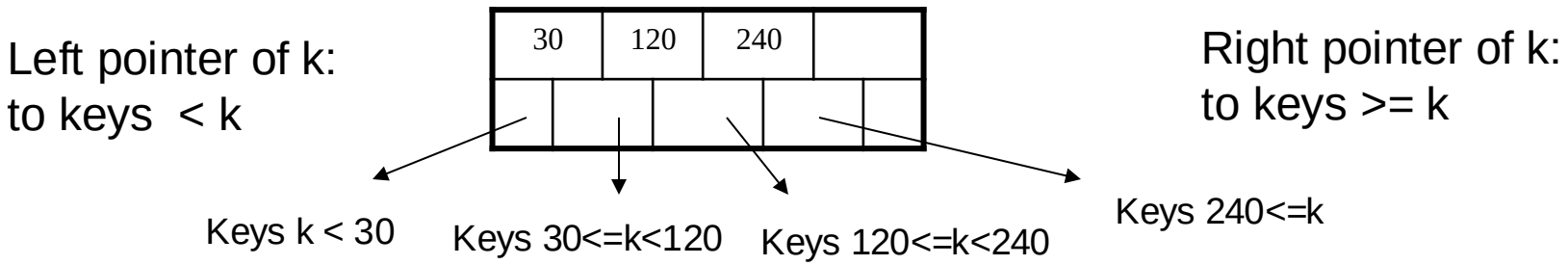
- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

B+ Trees Details

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

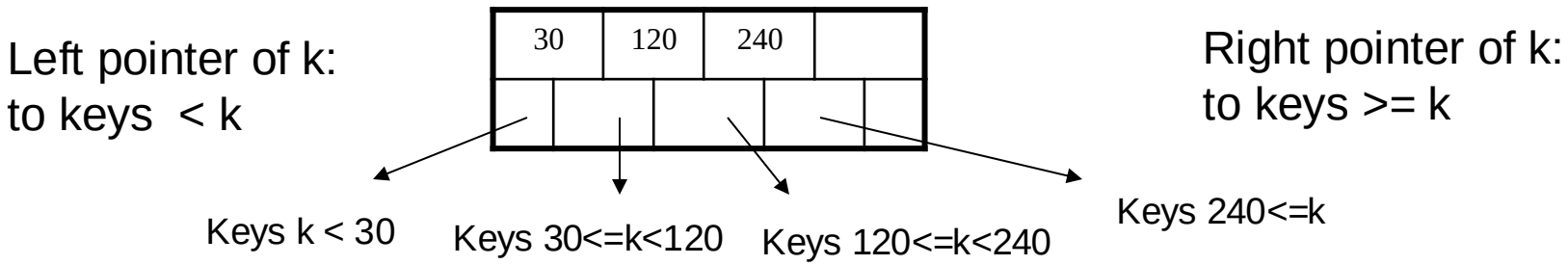
B+ Trees Details

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

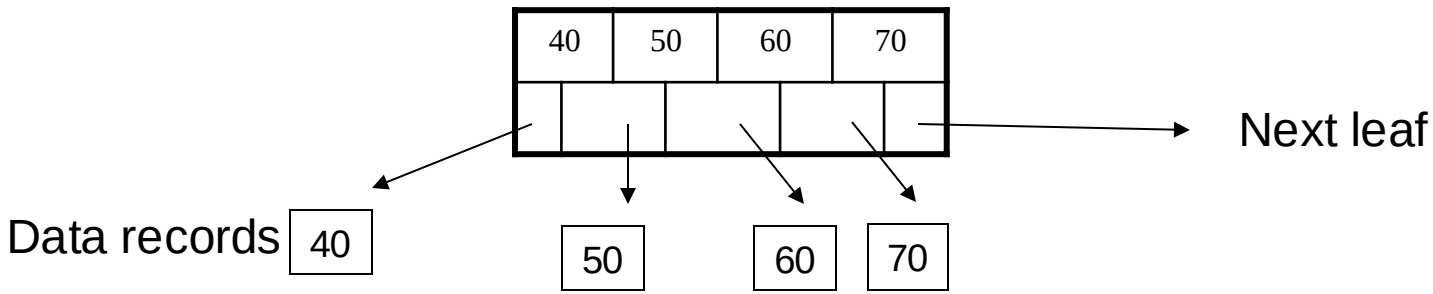


B+ Trees Details

- 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**:

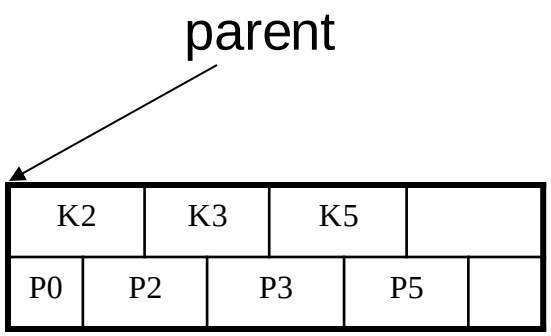


Insertion in a B+ Tree

Insert (K, P)

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

Insert k1

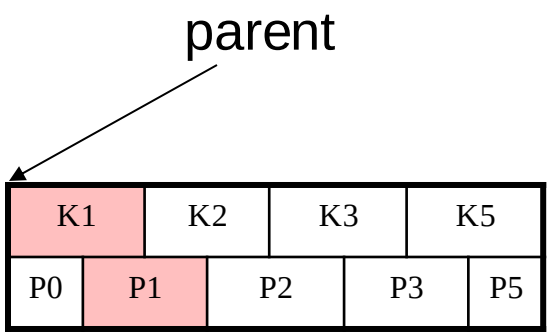


Insertion in a B+ Tree

Insert (K, P)

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

Insert k1

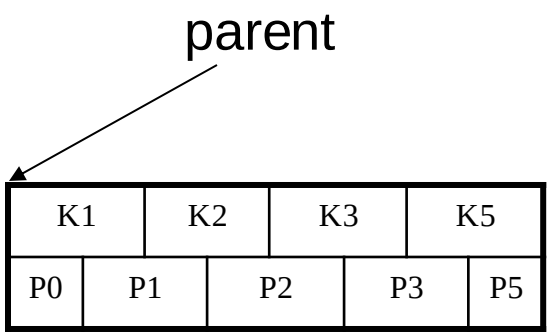


Insertion in a B+ Tree

Insert (K, P)

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

Insert k4

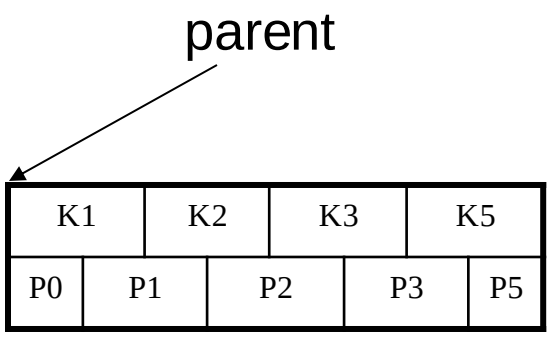


Insertion in a B+ Tree

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

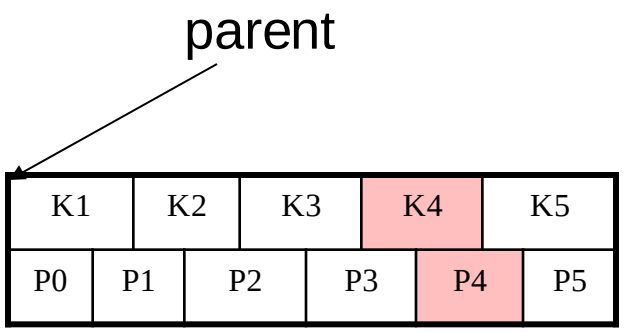


Insertion in a B+ Tree

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

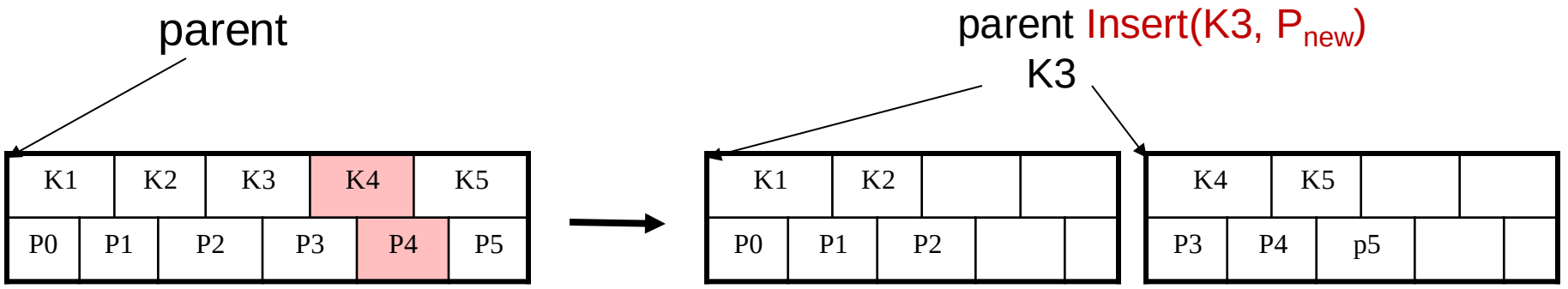


Insertion in a B+ Tree

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

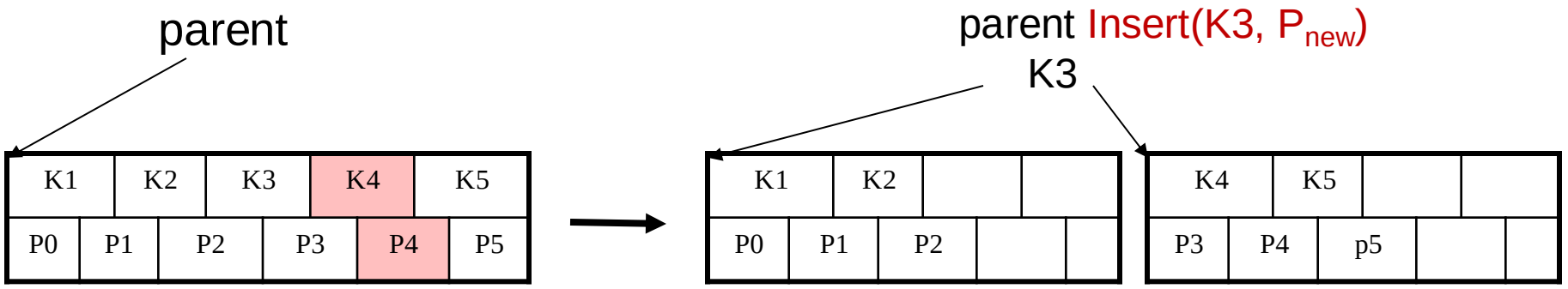


Insertion in a B+ Tree

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:

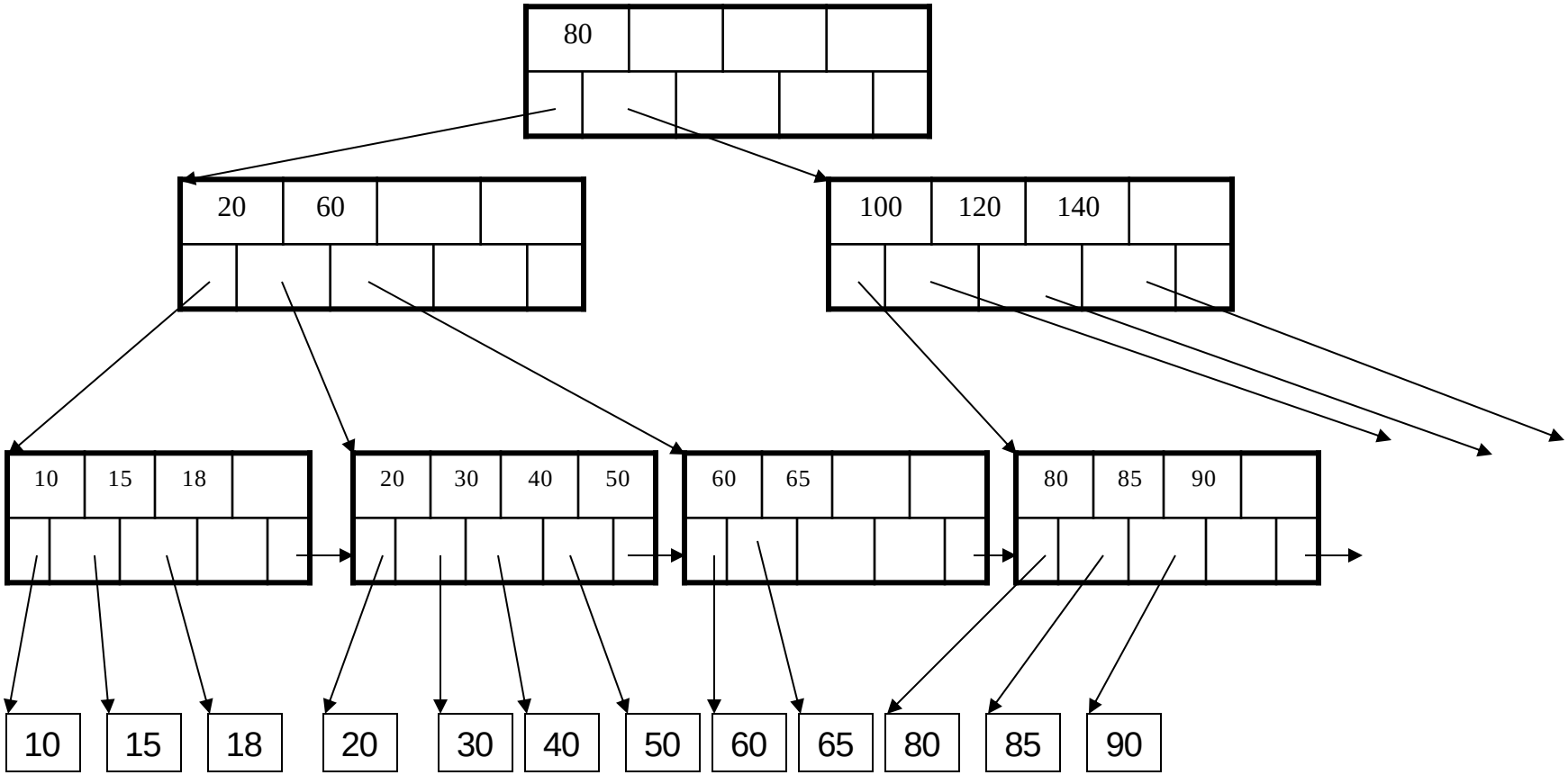
Insert k4



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

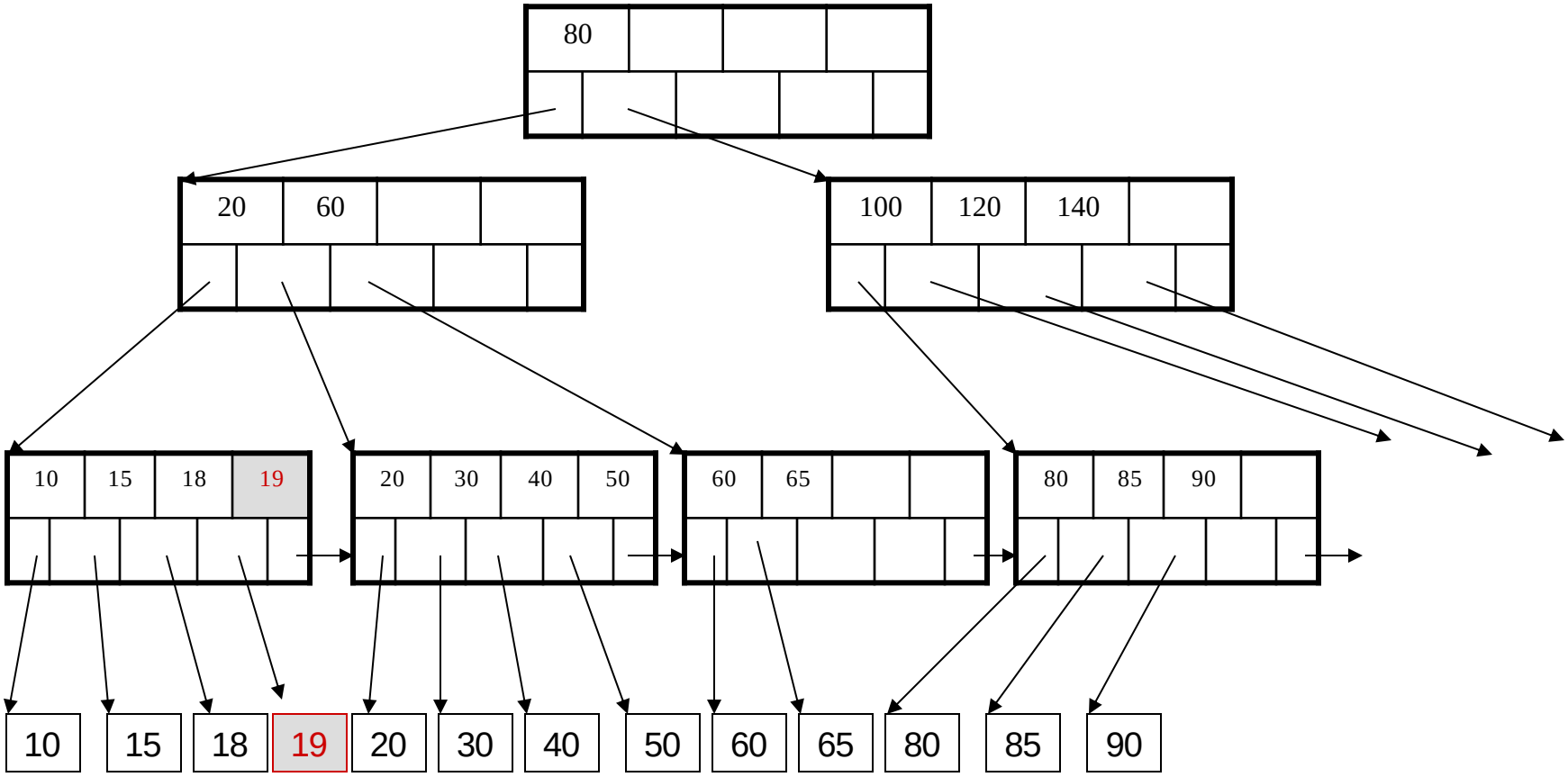
Insertion in a B+ Tree

Insert K=19



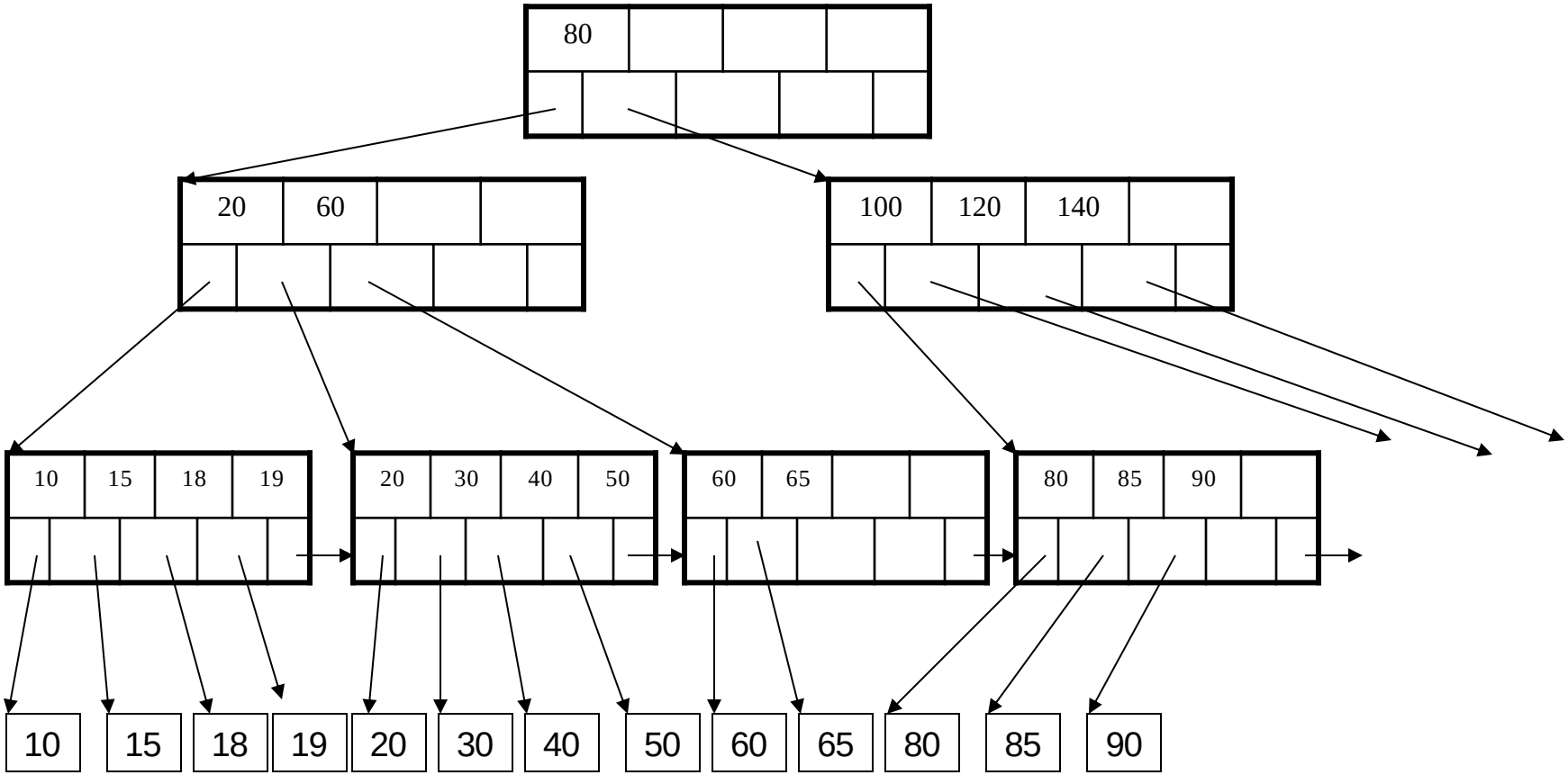
Insertion in a B+ Tree

After insertion



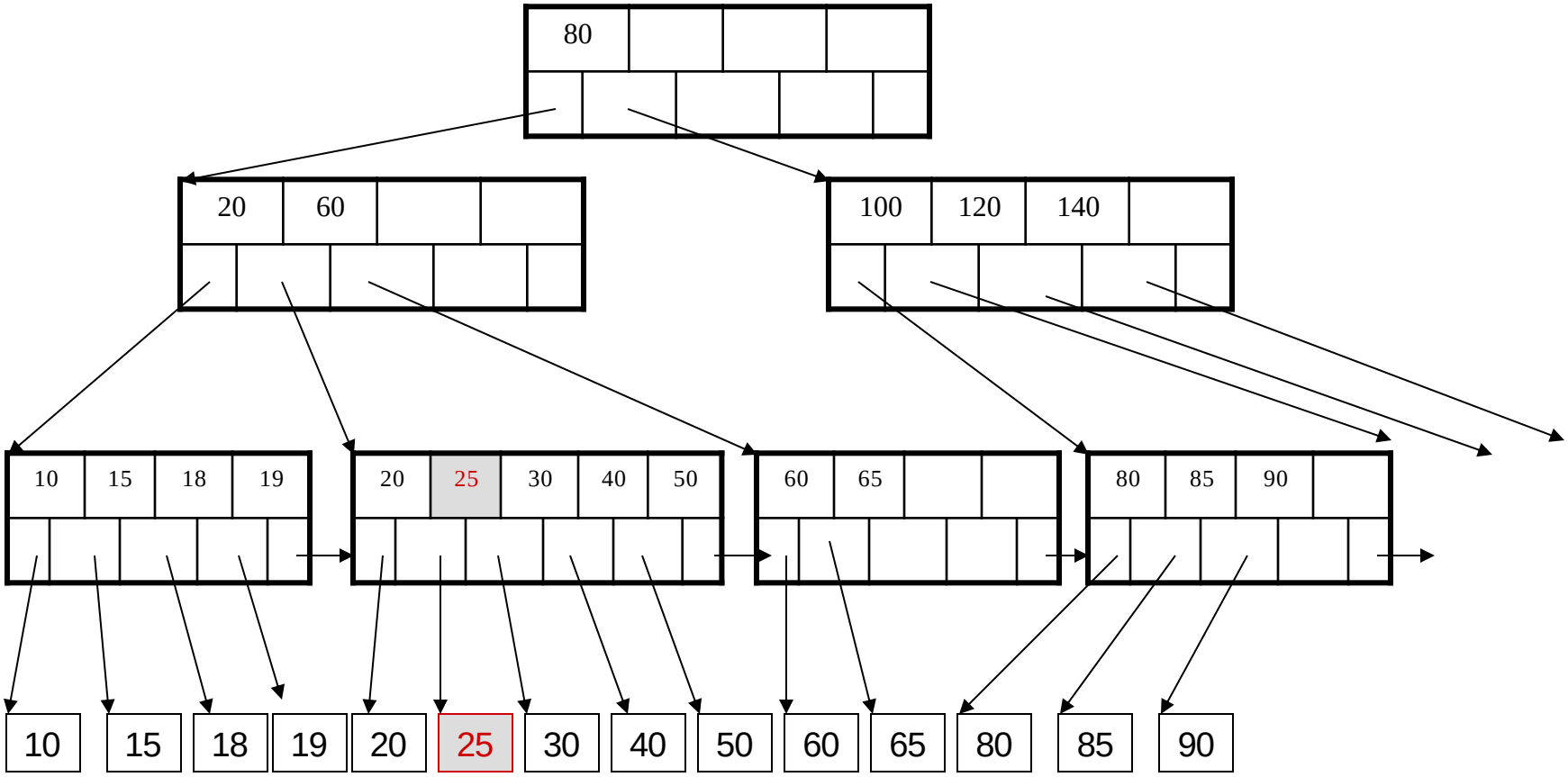
Insertion in a B+ Tree

Now insert 25



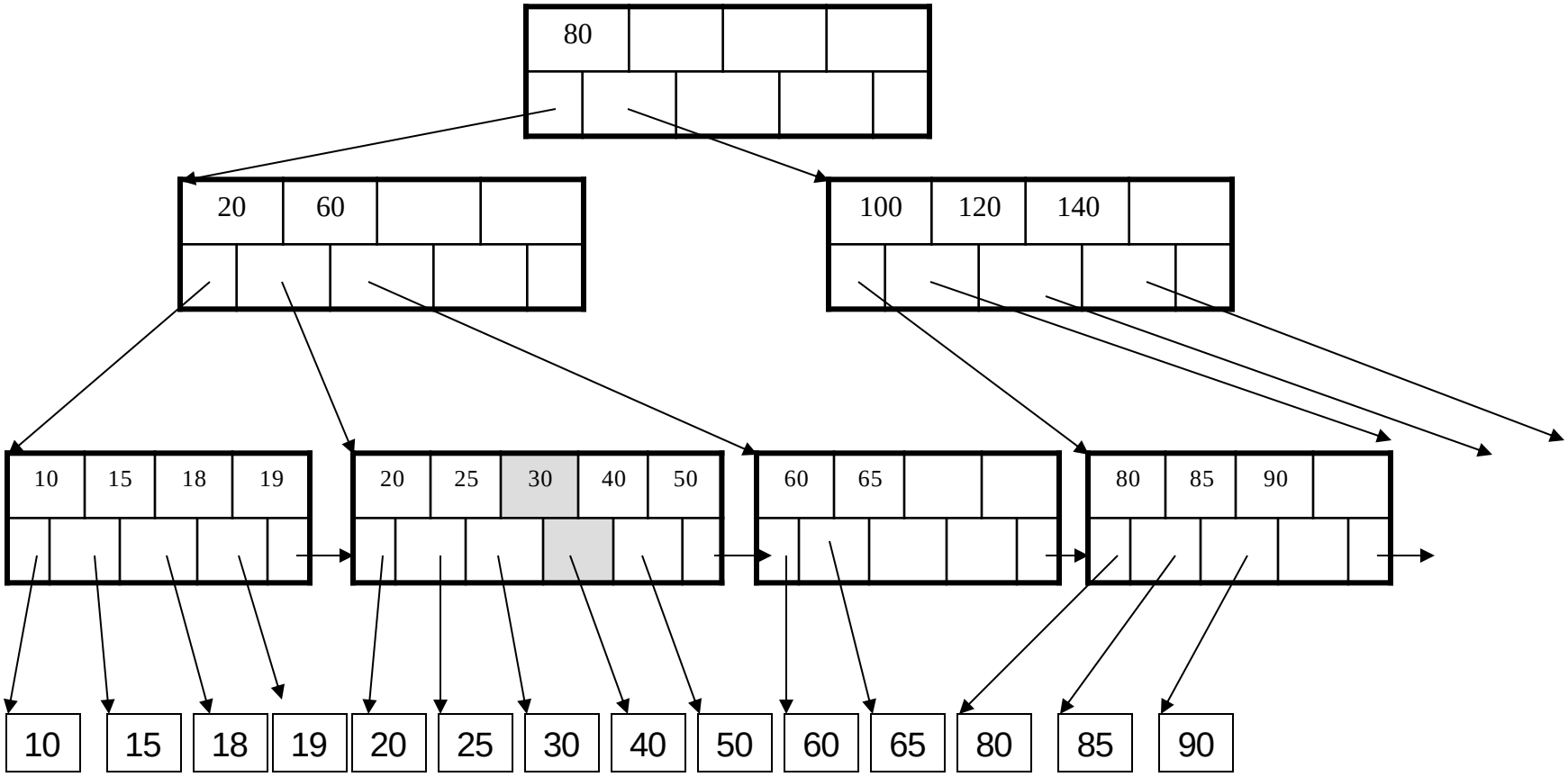
Insertion in a B+ Tree

After insertion



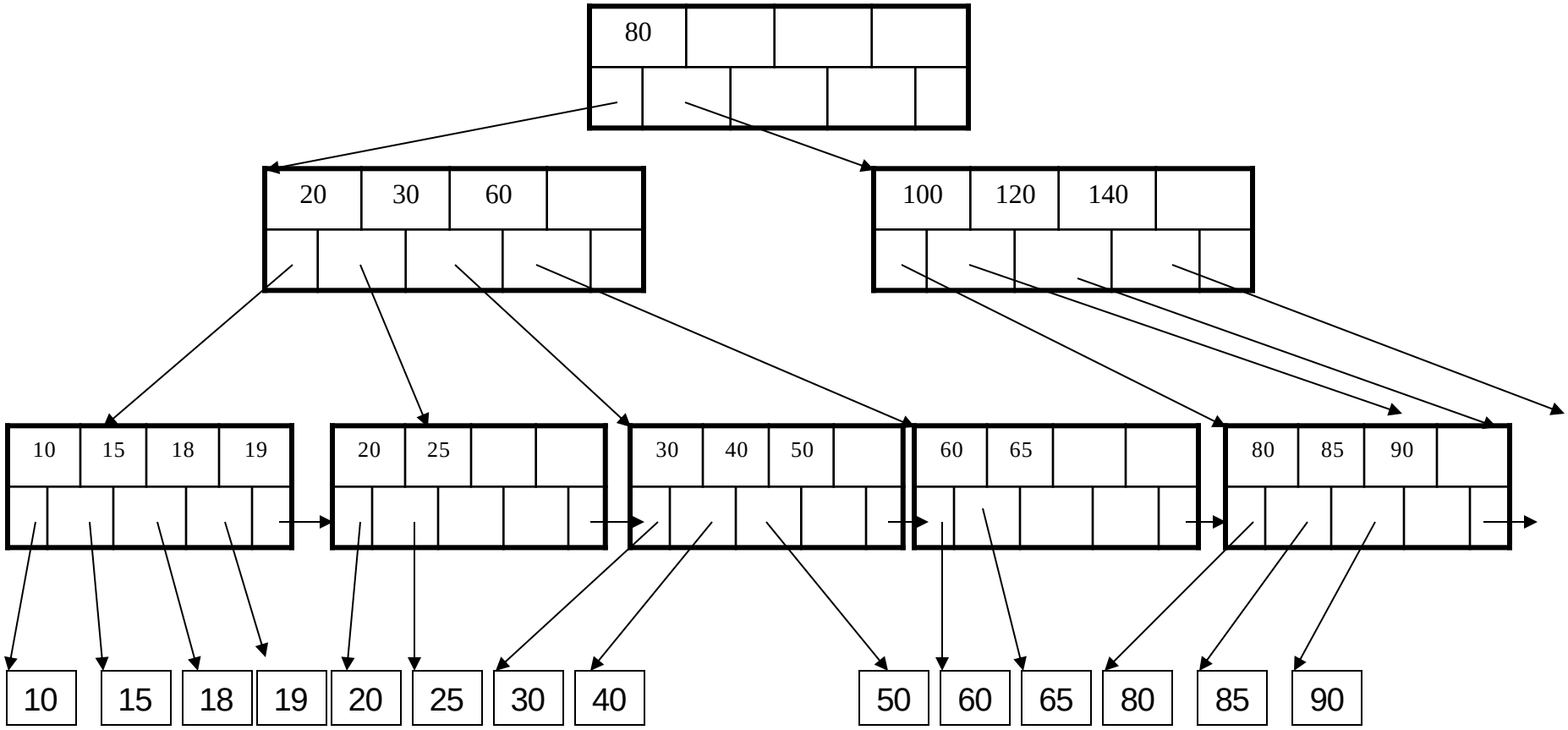
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



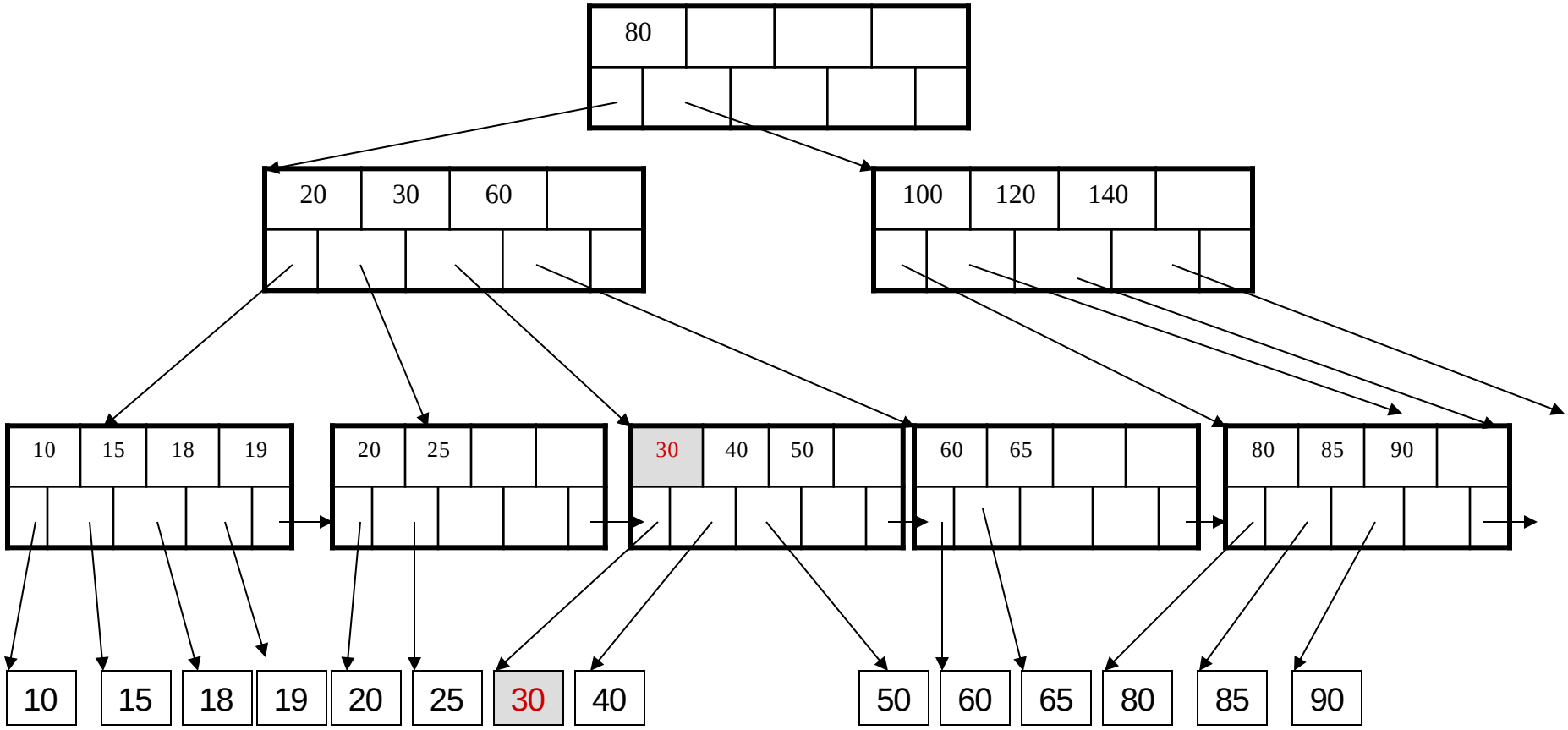
Deletion: Summary

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’)

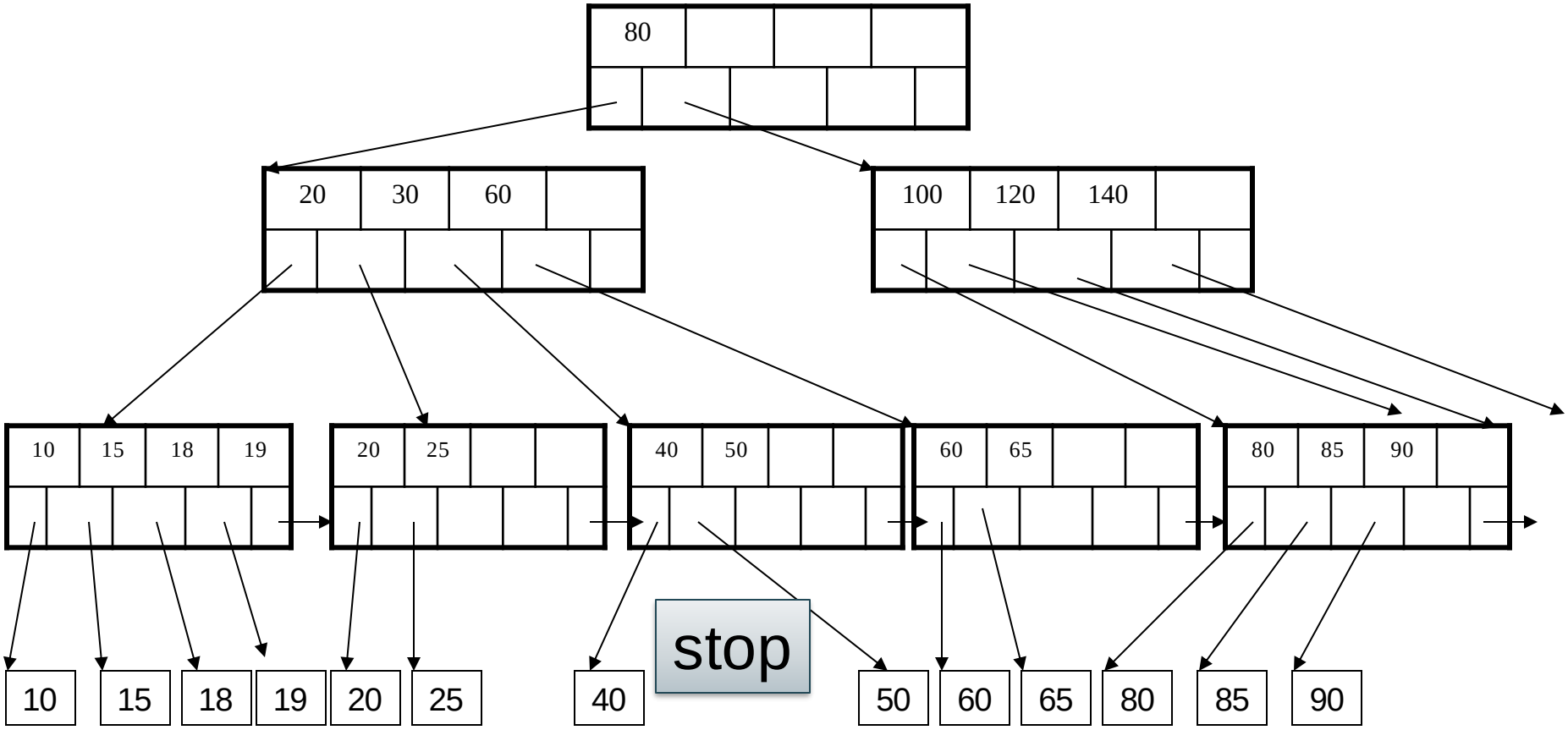
Deletion from a B+ Tree

Delete 30



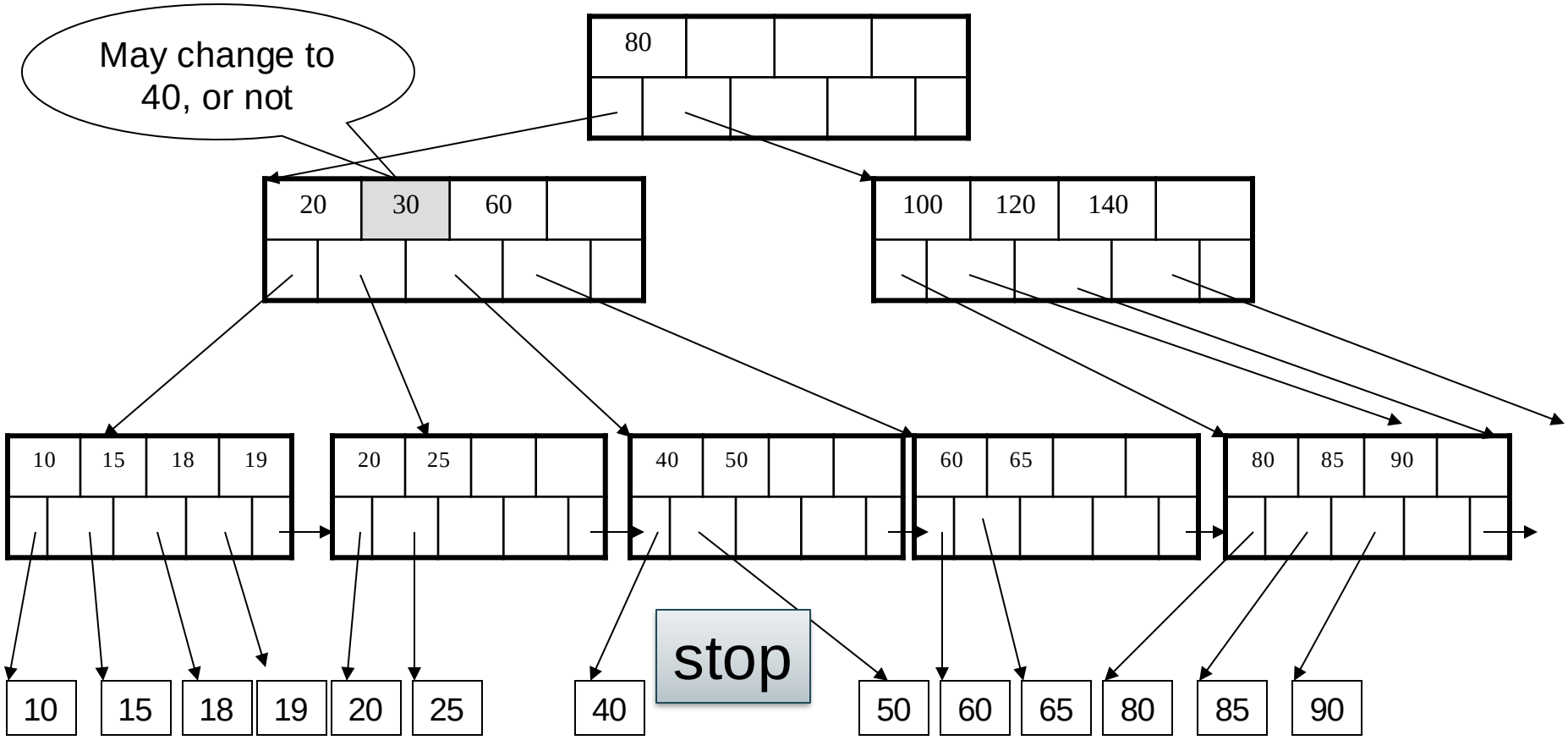
Deletion from a B+ Tree

After deleting 30



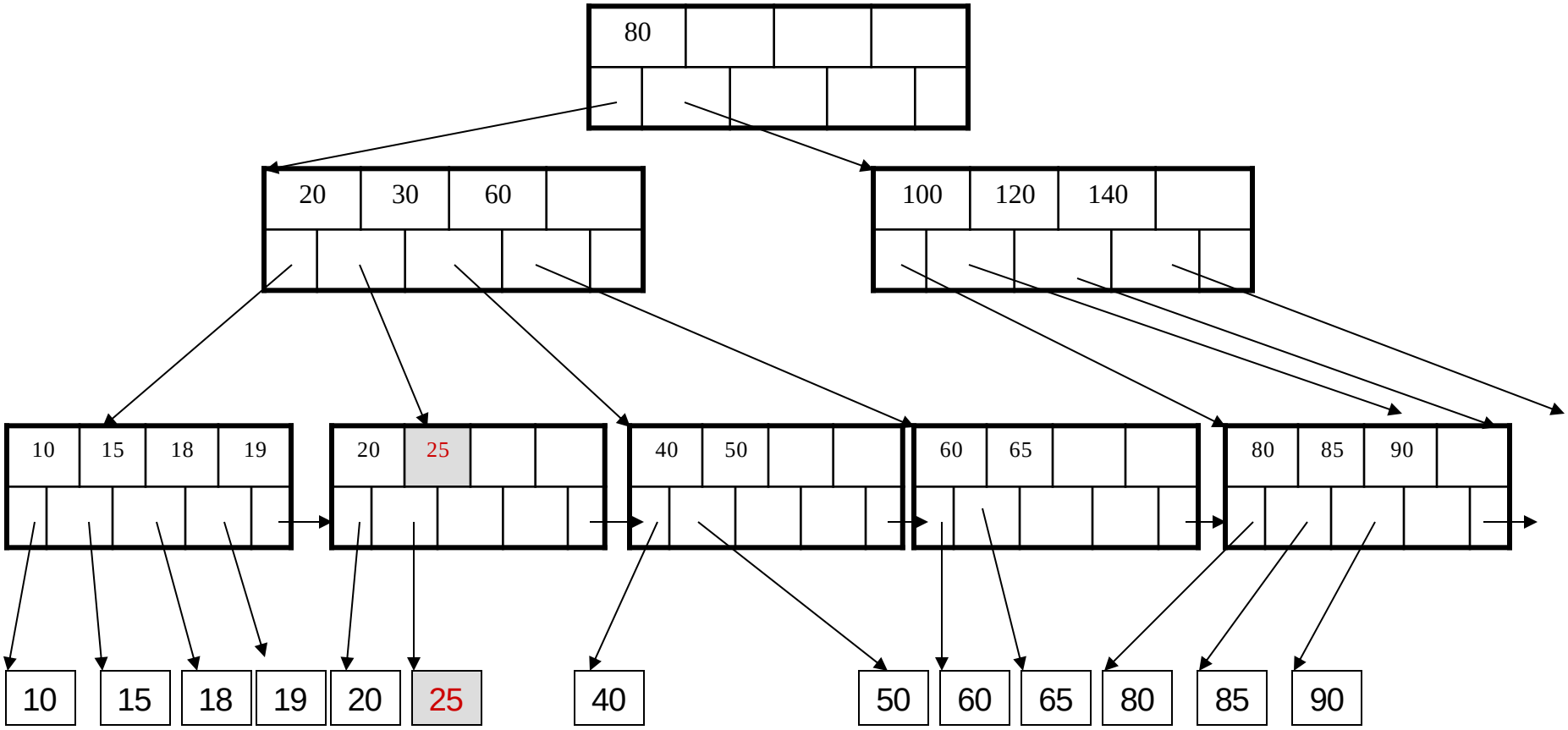
Deletion from a B+ Tree

After deleting 30



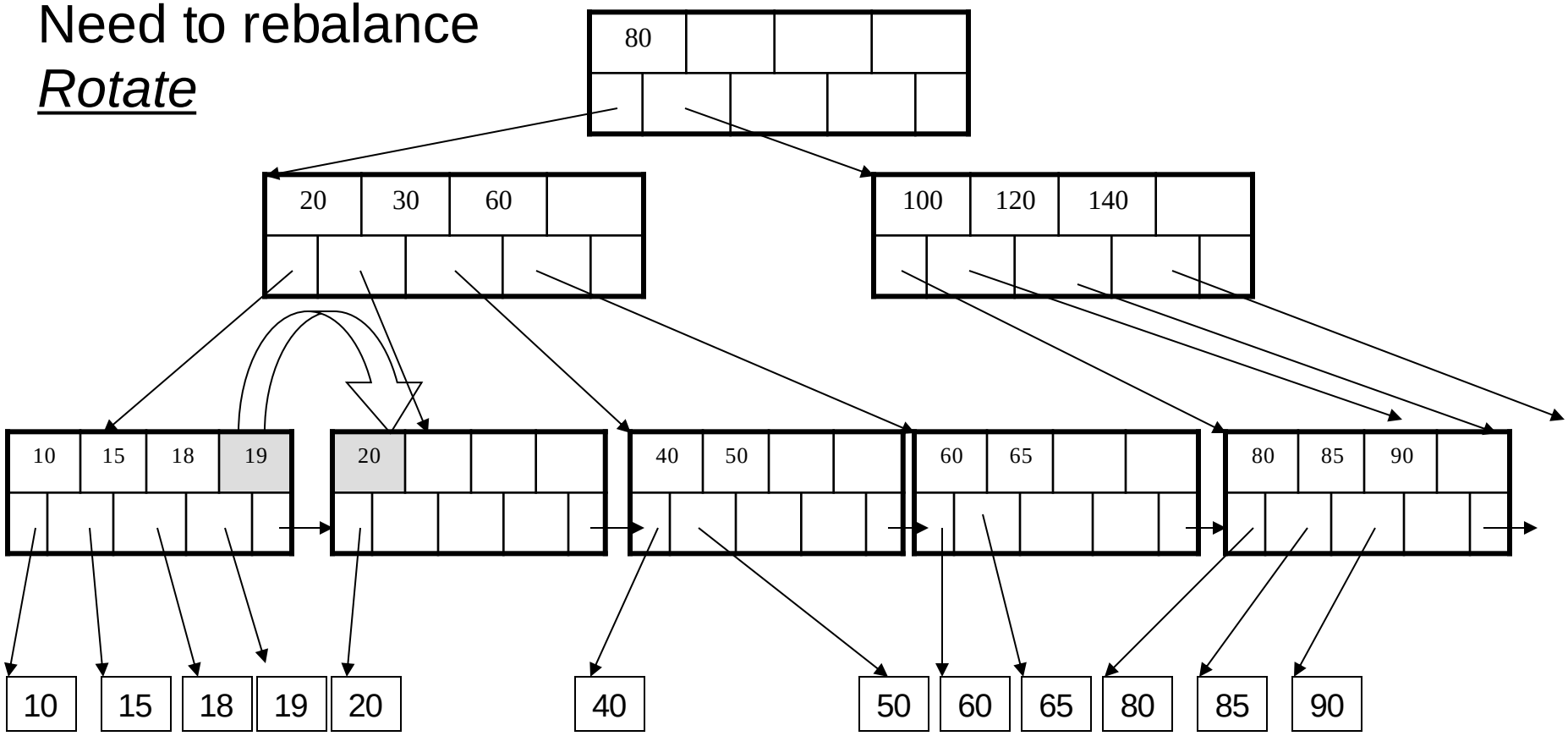
Deletion from a B+ Tree

Now delete 25

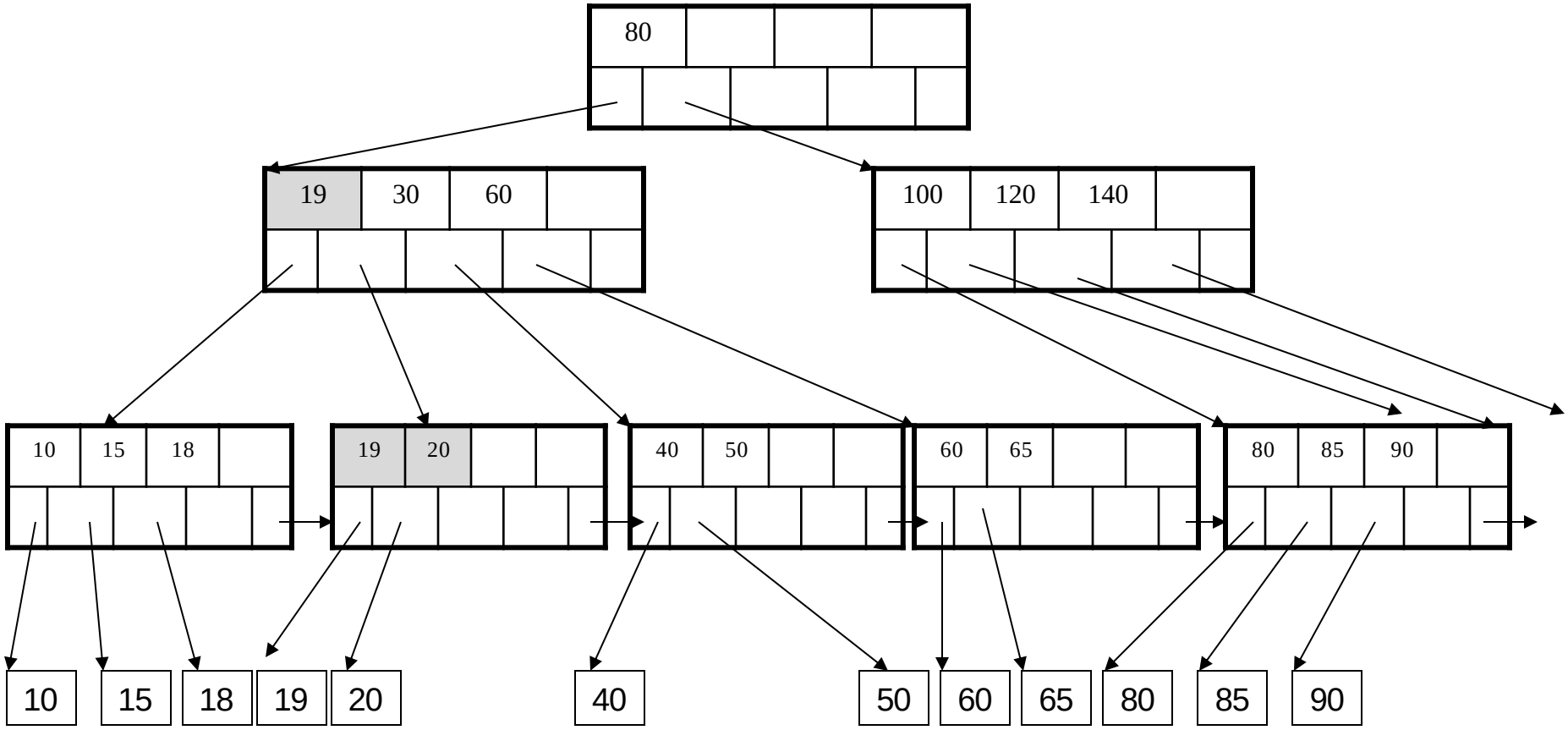


Deletion from a B+ Tree

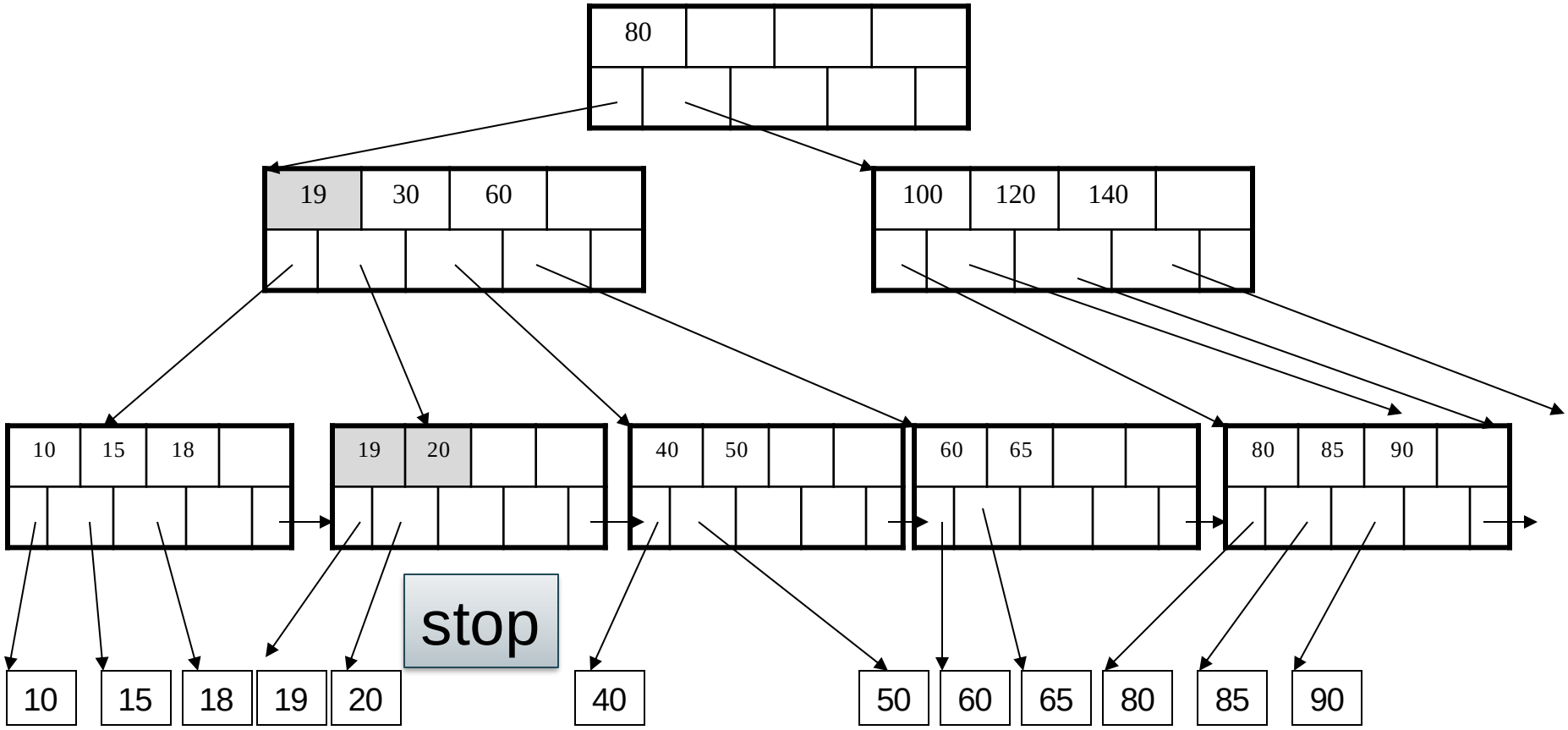
After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree

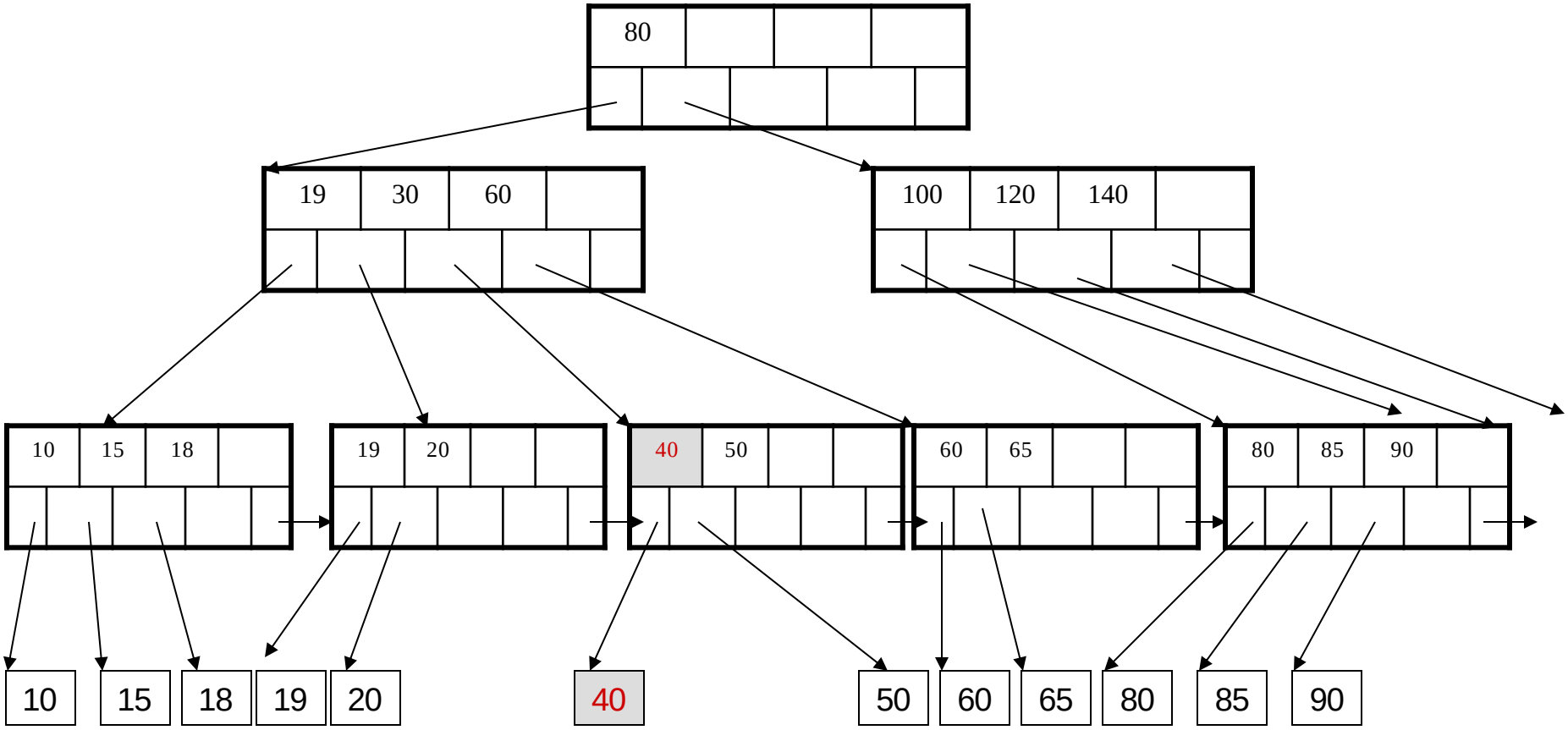


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

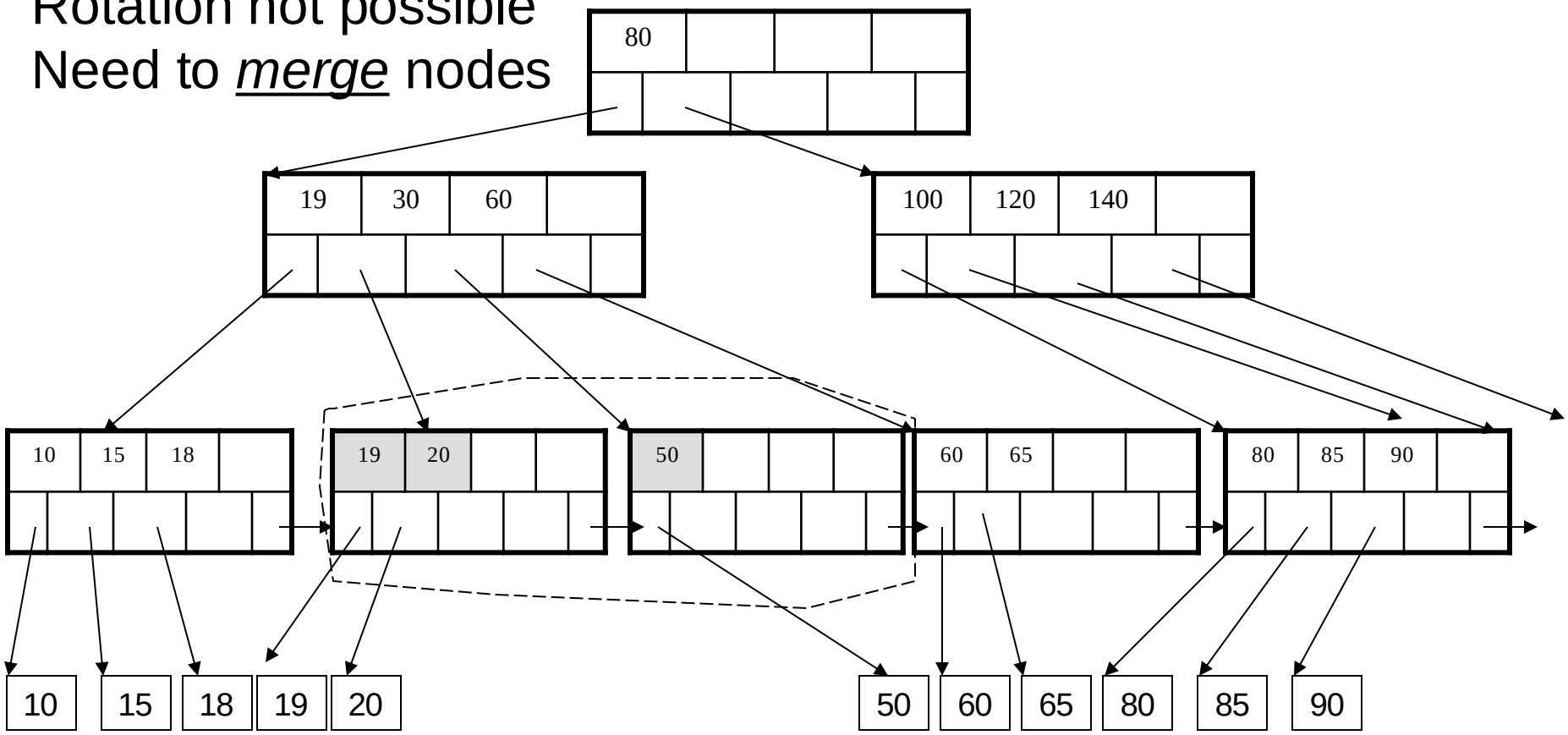


Deletion from a B+ Tree

After deleting 40

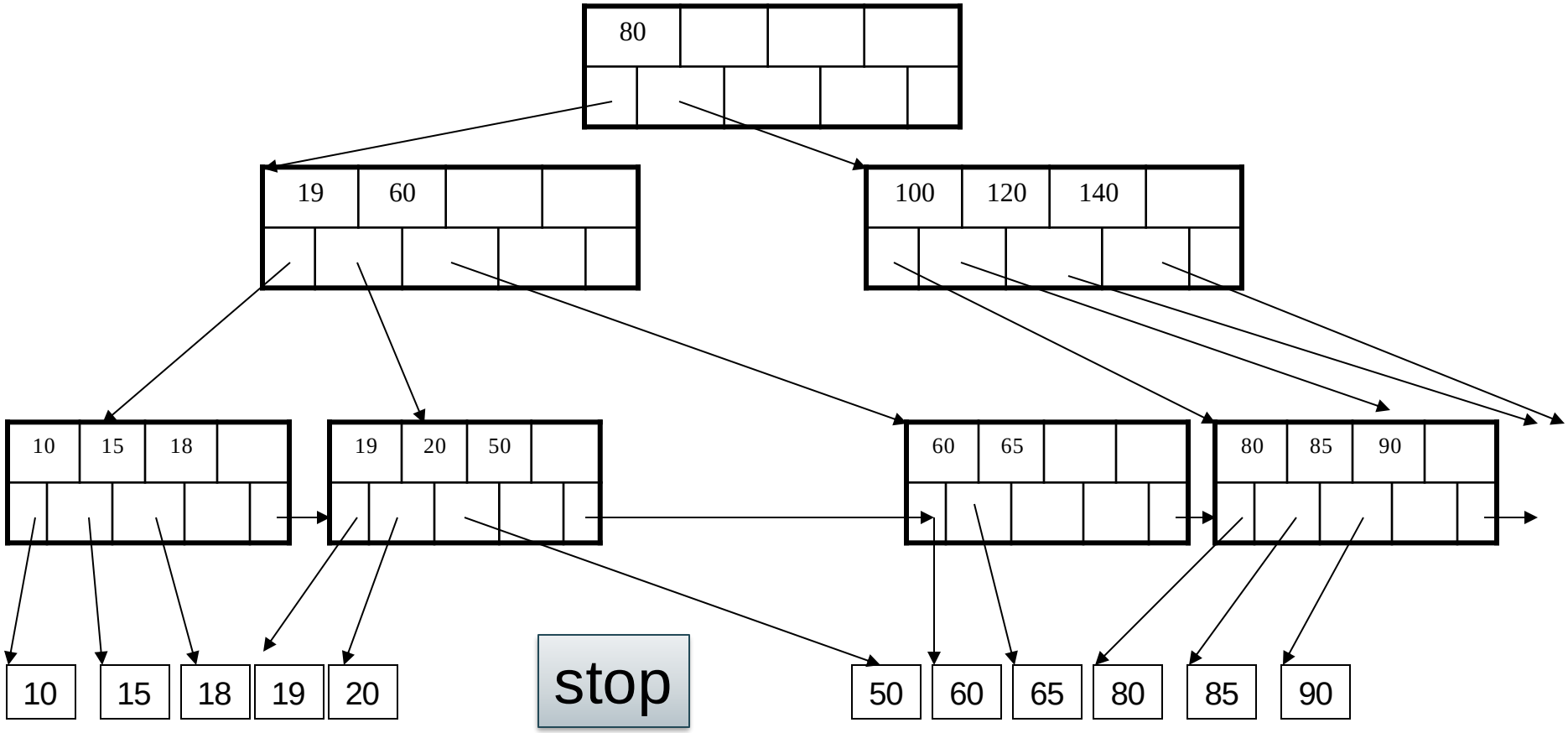
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



Discussion

- B+ trees are always perfectly balanced
- Depth is $\log_m N$, where:
  - N=number of data items
  - m=average fanout of a node
- Search, Insert, Delete = $O(\log_m N)$
- Bulk creation of index is cheaper than inserting N items

Depth in Practice

- Depths never exceeds 6

Depth in Practice

- Depths never exceeds 6
- Example: if $N=10^{10}$, $m=10^2$

Depth in Practice

- Depths never exceeds 6
- Example: if $N=10^{10}$, $m=10^2$

$$\text{depth} = \log_m N = \frac{\log N}{\log m} = \frac{\log 10^{10}}{\log 10^2} = \frac{10}{2} = 5$$

Indexes

Index

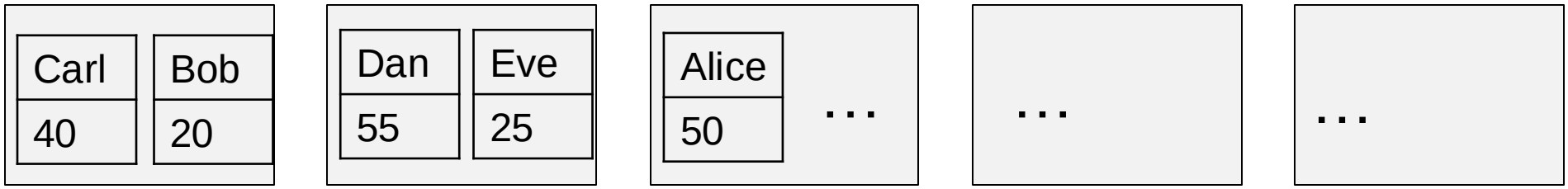
- An index is an auxiliary file that allows direct access to the main data file
- Usually a B+ tree or a hash-table
- Other specialized indices:
  - R-tree, PAT-tree, Trie, …

Index in SQL

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

Index in SQL

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

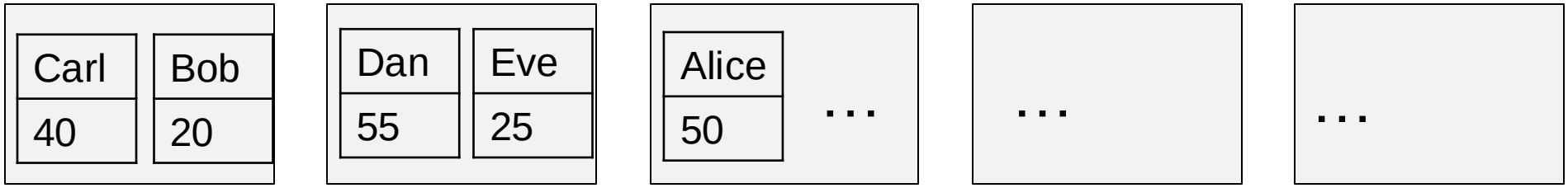


Person data file

Index in SQL

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

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

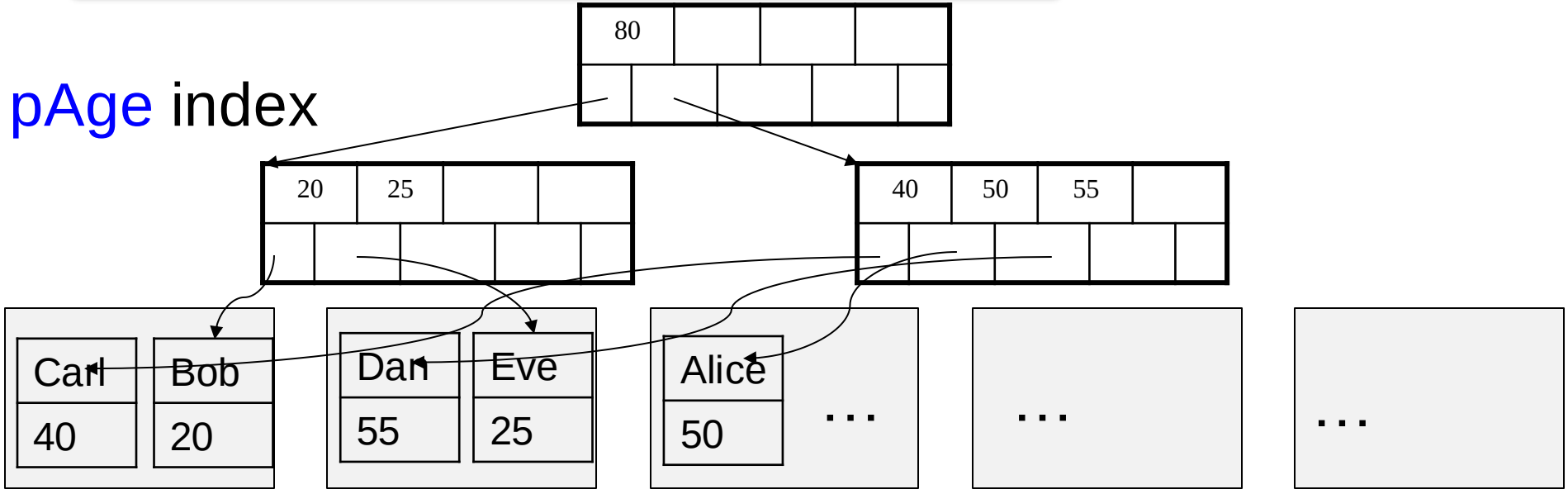


Person data file

Index in SQL

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

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

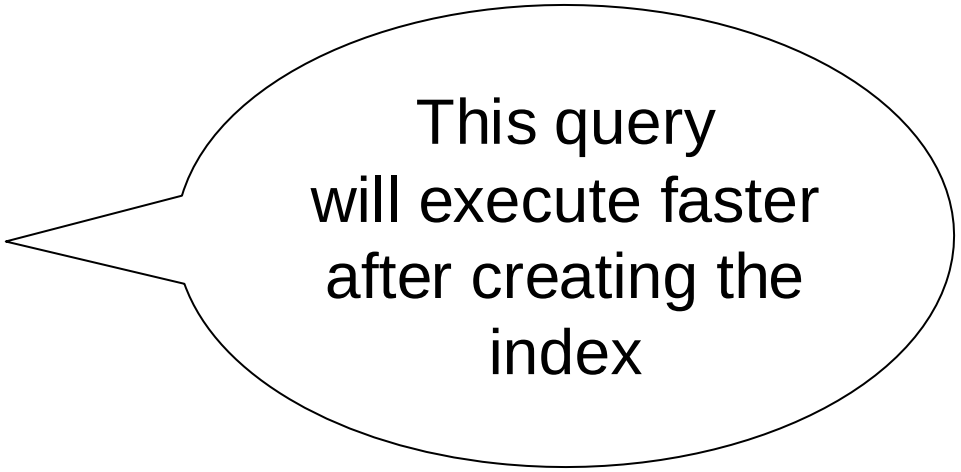


Index in SQL

```
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

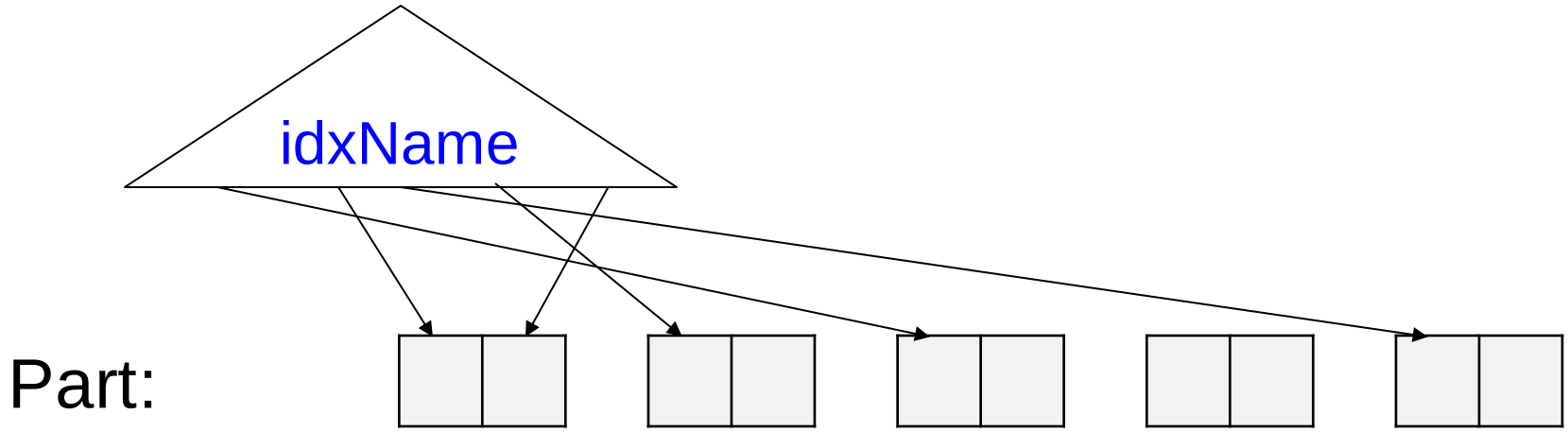
Discussion

- 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

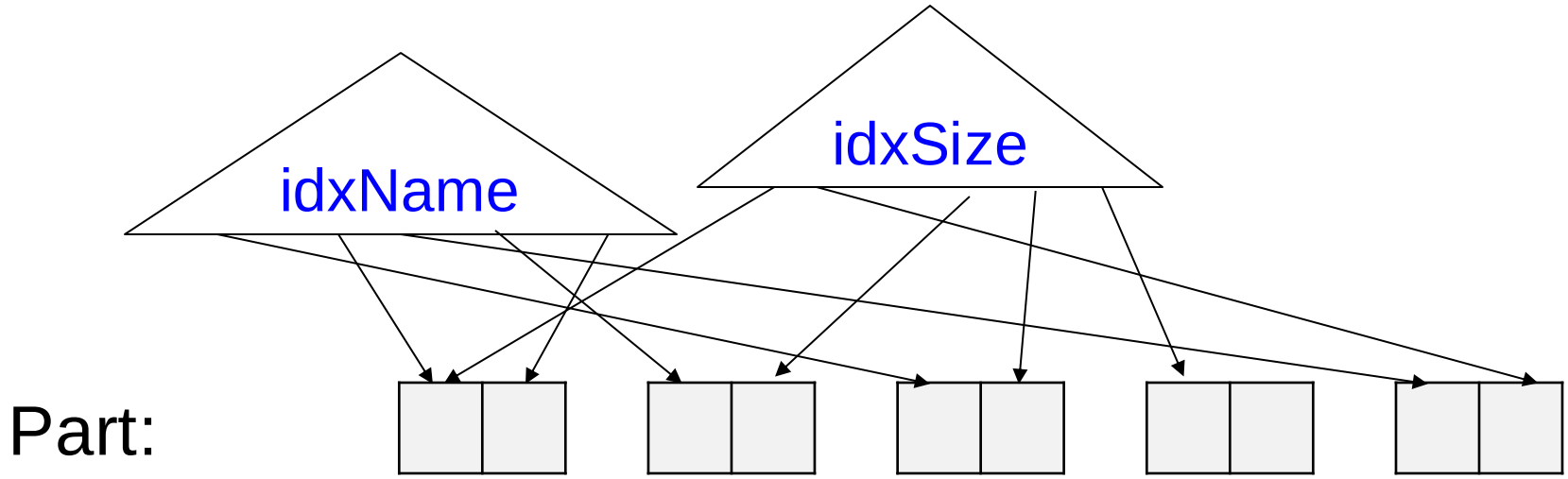
```
create index idxName on Part(pname);
```



Part(pno,pname,psize,pcolor)

Create Index

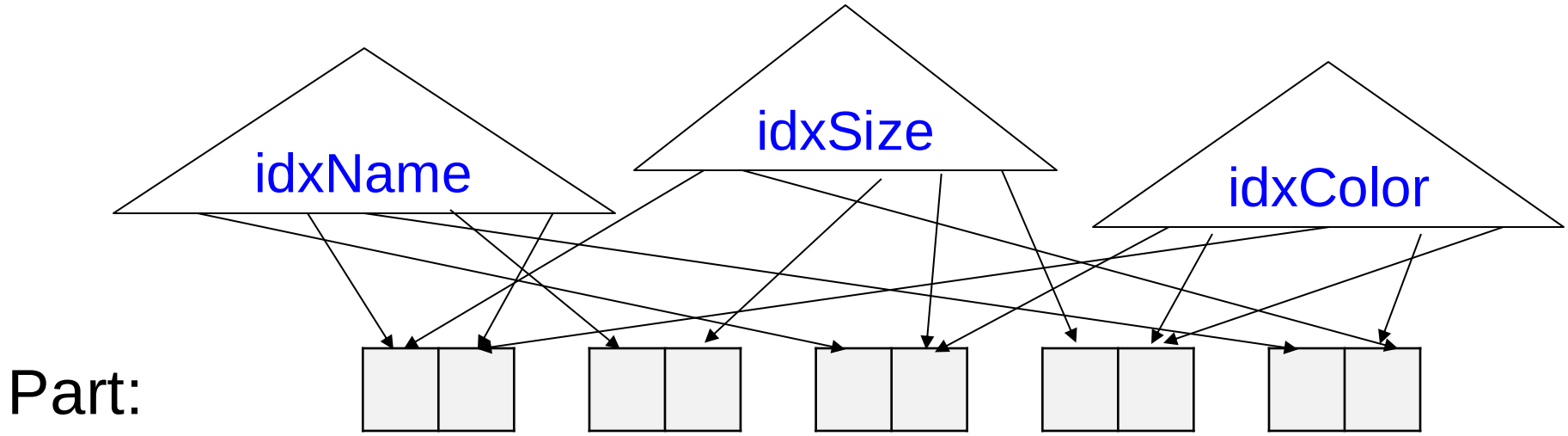
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
```



Part(pno,pname,psize,pcolor)

Create Index

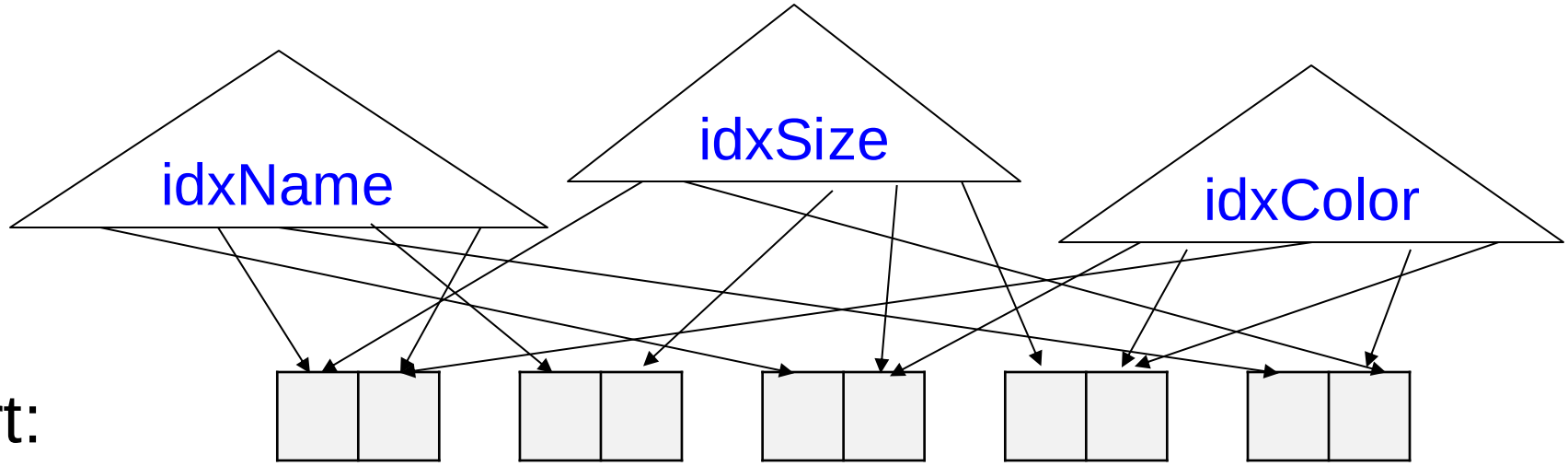
```
create index idxName on Part(pname);
create index idxSize on Part(psize);
create index idxColor on Part(pcolor);
```



Part(pno,pname,psize,pcolor)

Create Index

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

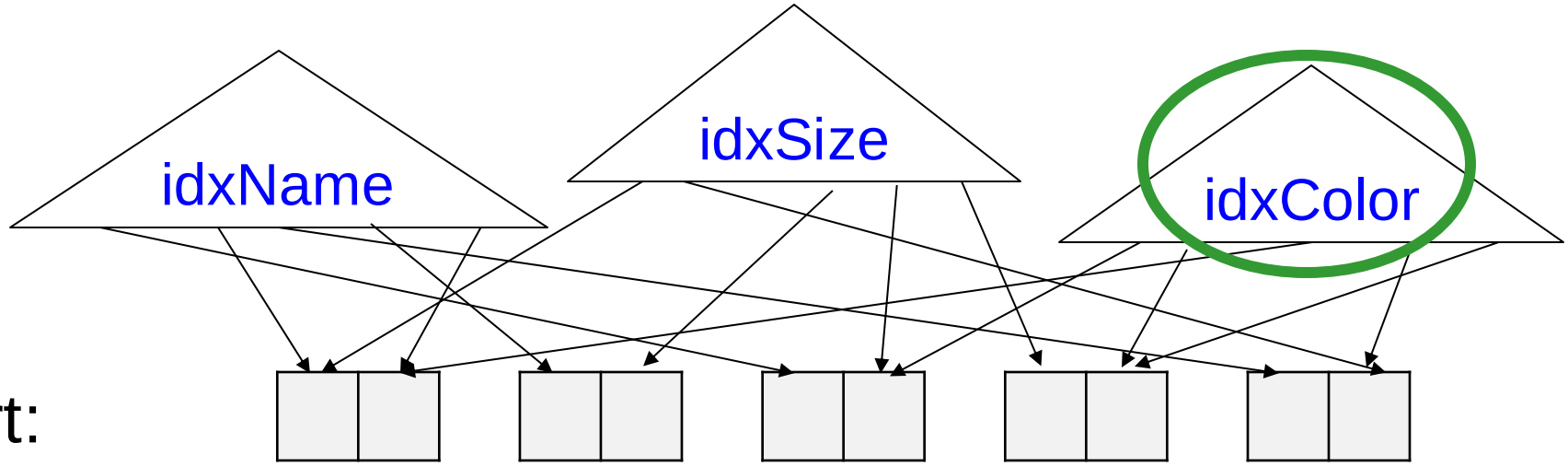


Part(pno,pname,psize,pcolor)

Create Index

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

Use idxColor

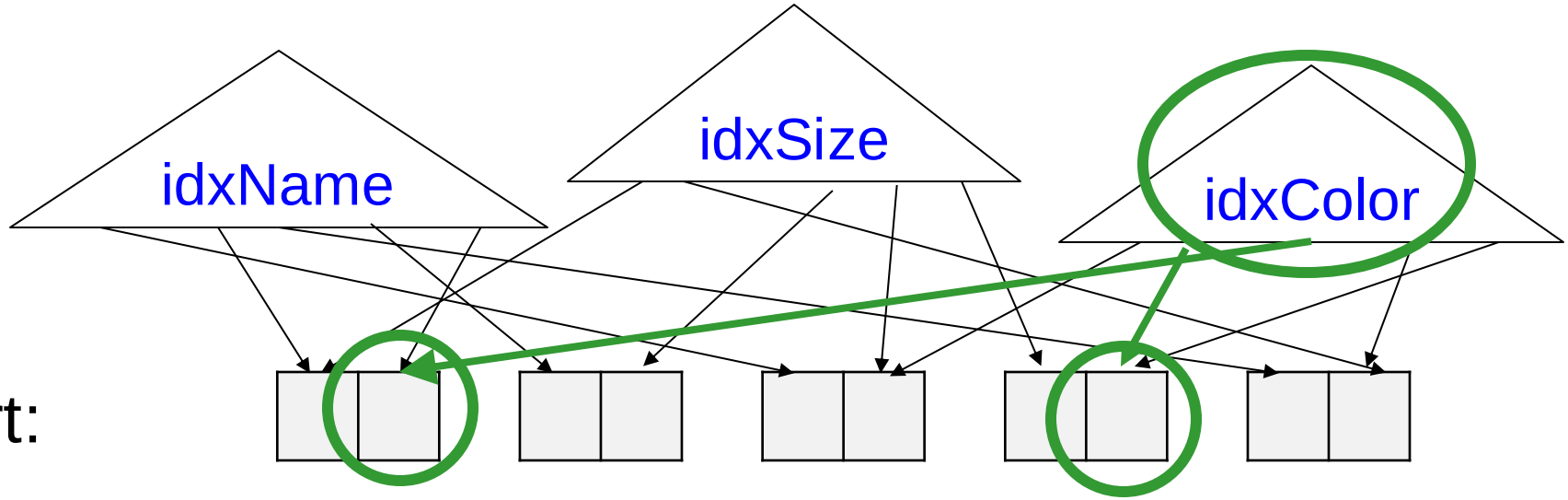


Part(pno,pname,psize,pcolor)

Create Index

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

Use idxColor

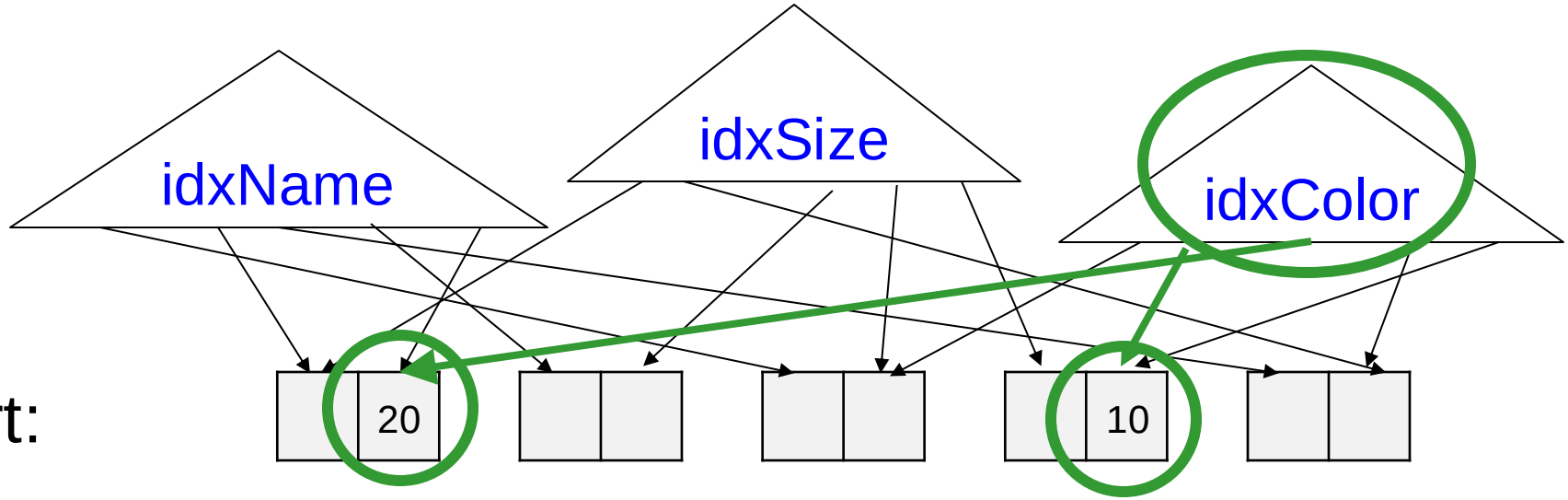


Part(pno,pname,psize,pcolor)

Create Index

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

Use idxColor



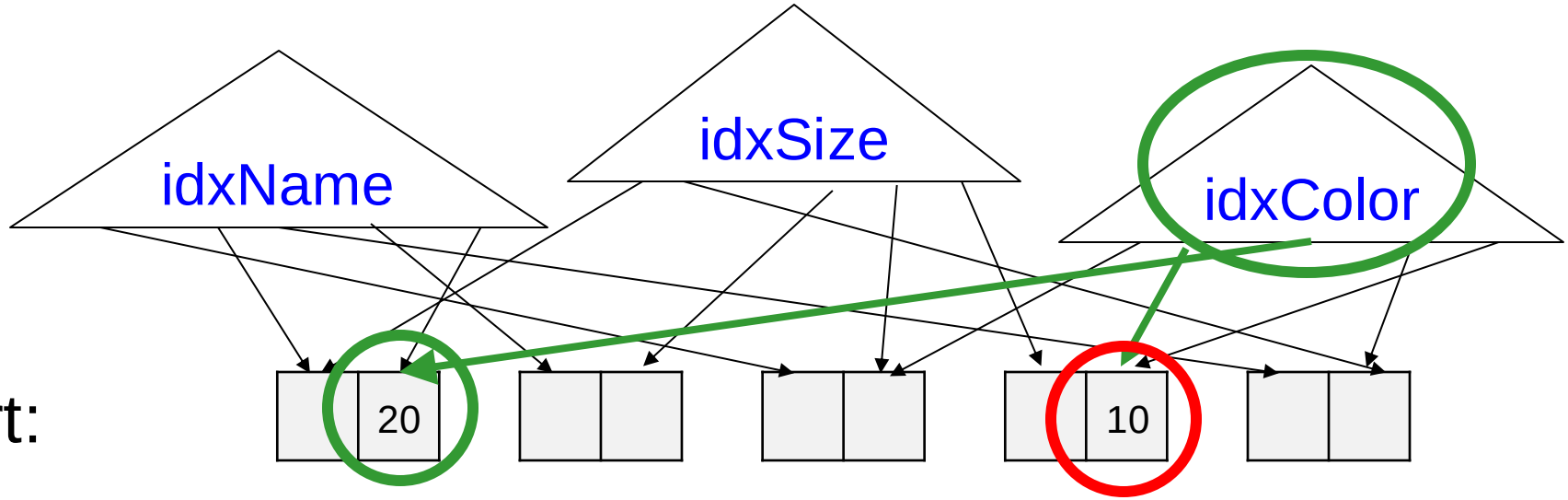
Read all green parts, return those with psize=10

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)

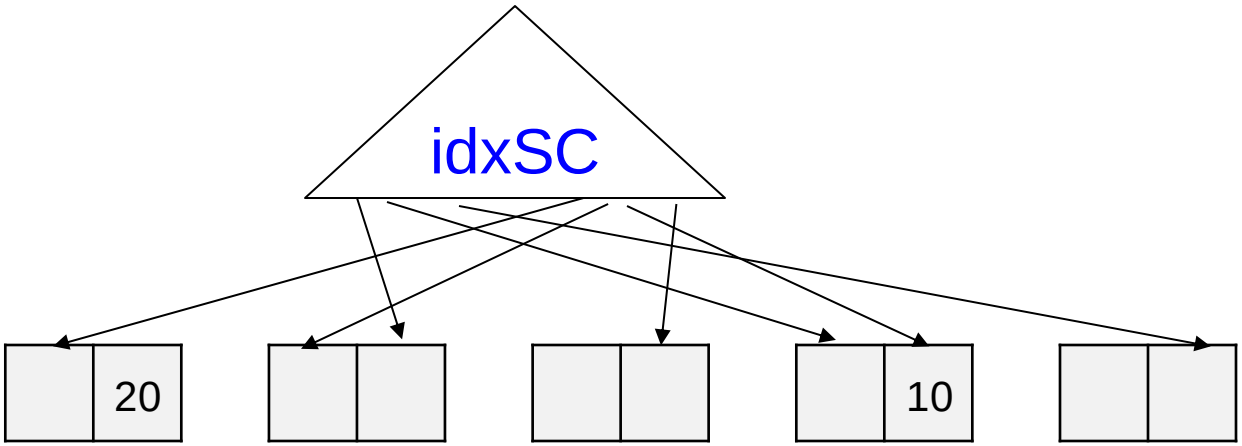
Multi-attribute Index

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

Part(pno,pname,psize,pcolor)

Multi-attribute Index

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

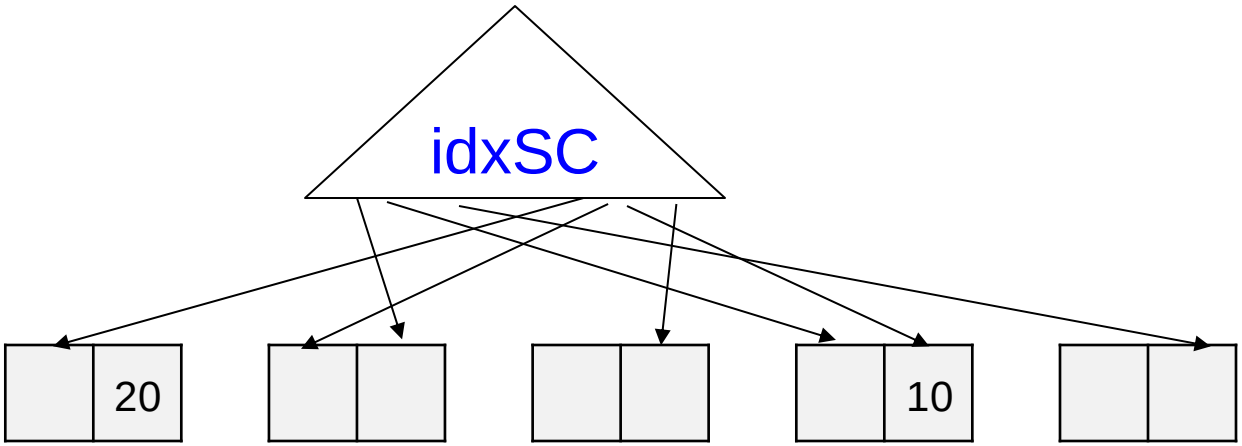


Part(pno,pname,psize,pcolor)

Multi-attribute Index

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

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

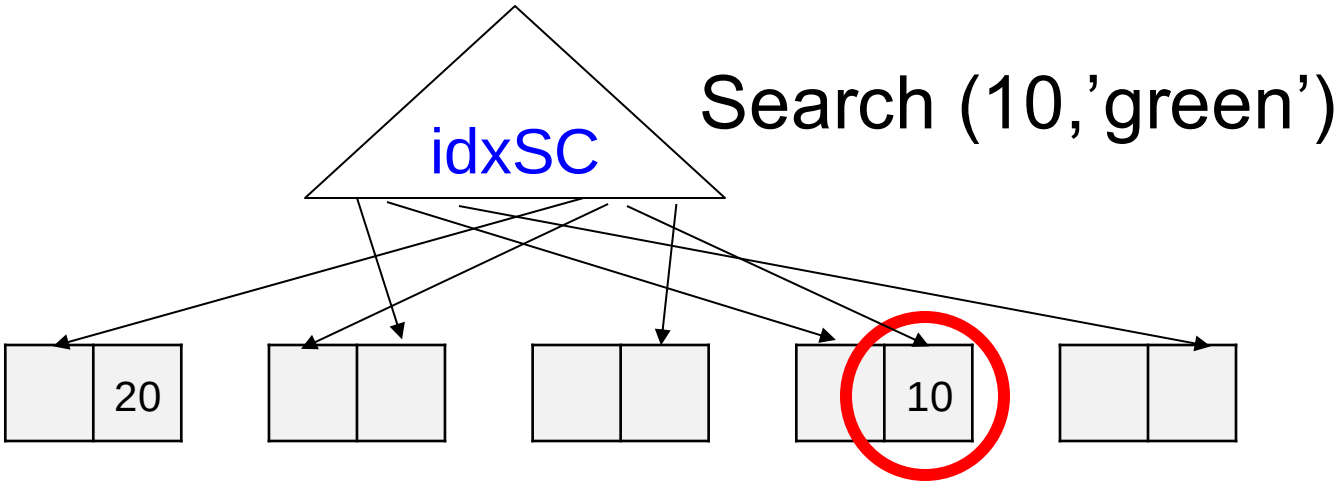


Part(pno,pname,psize,pcolor)

Multi-attribute Index

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

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

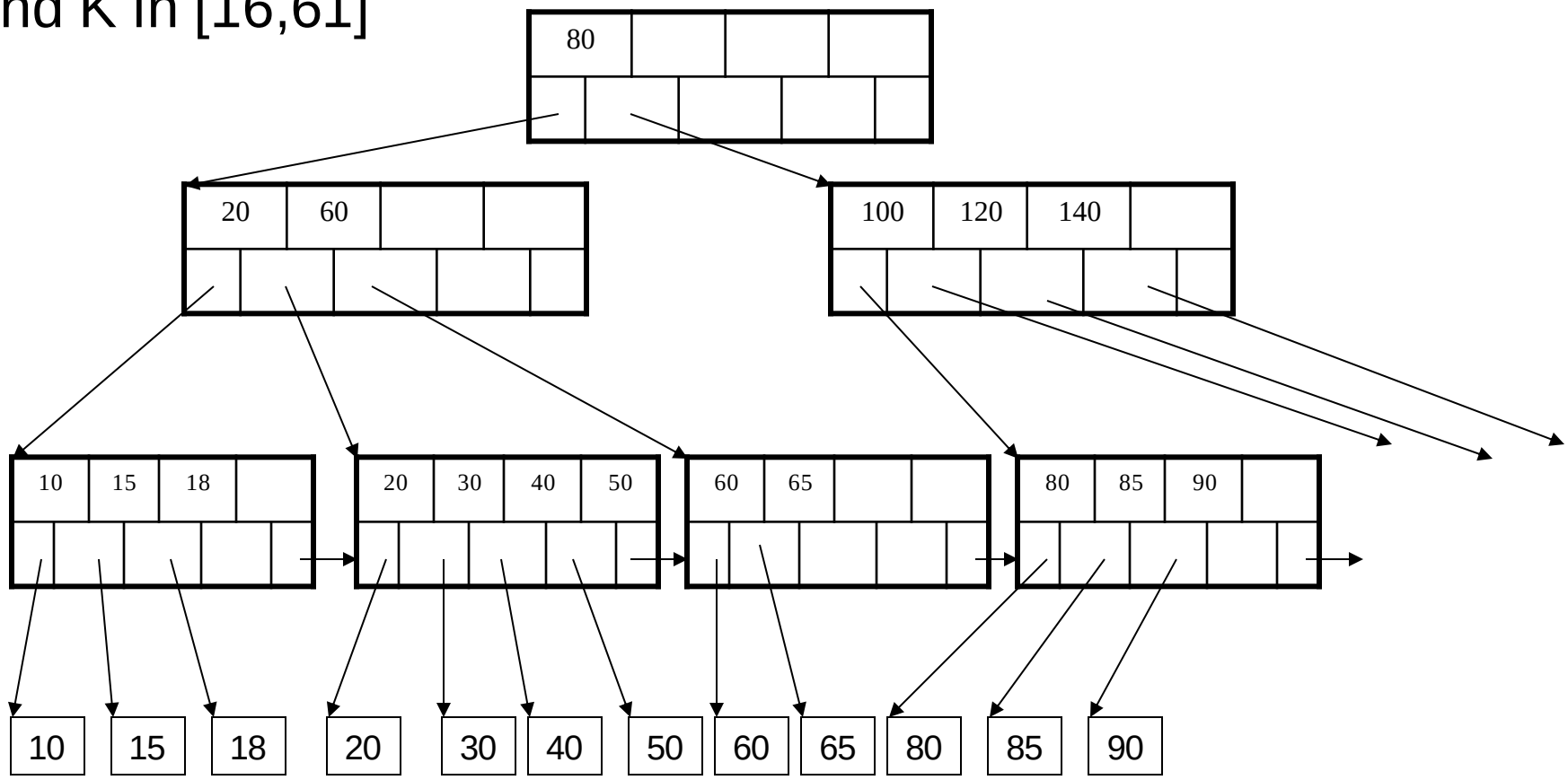


Multi-attribute Index

- 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)

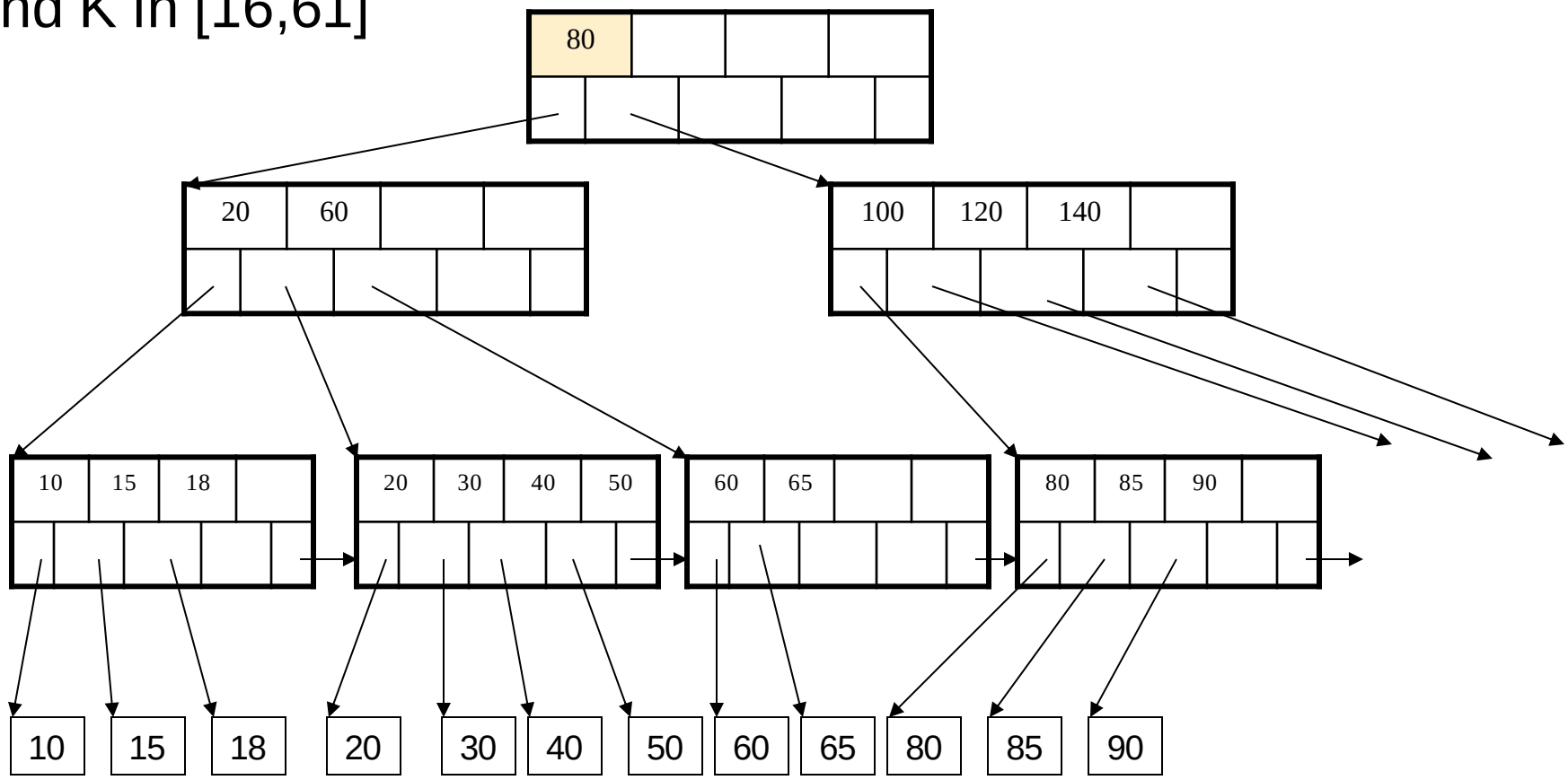
Range Search in B+ Tree

Find K in [16,61]



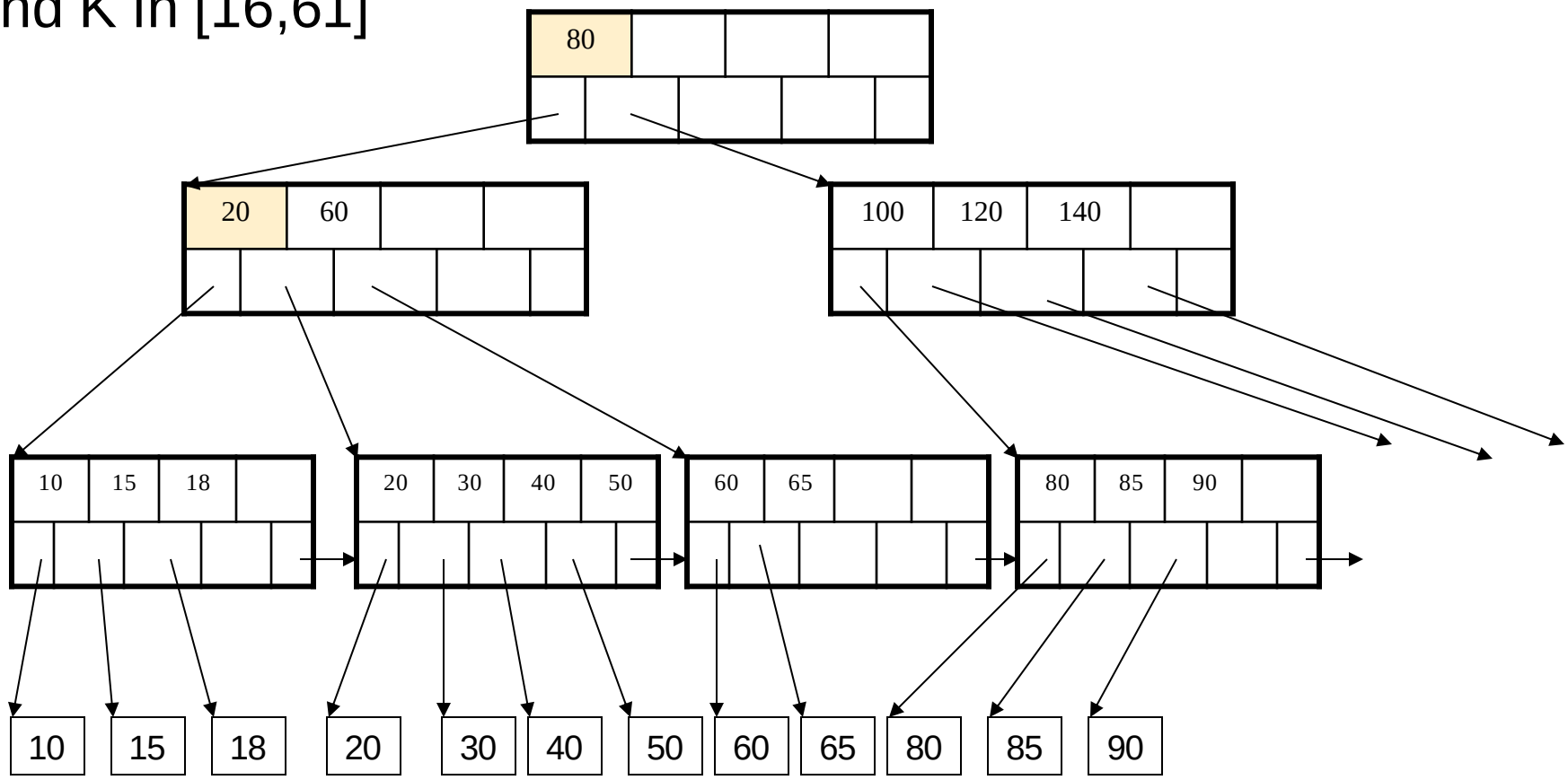
Range Search in B+ Tree

Find K in [16,61]



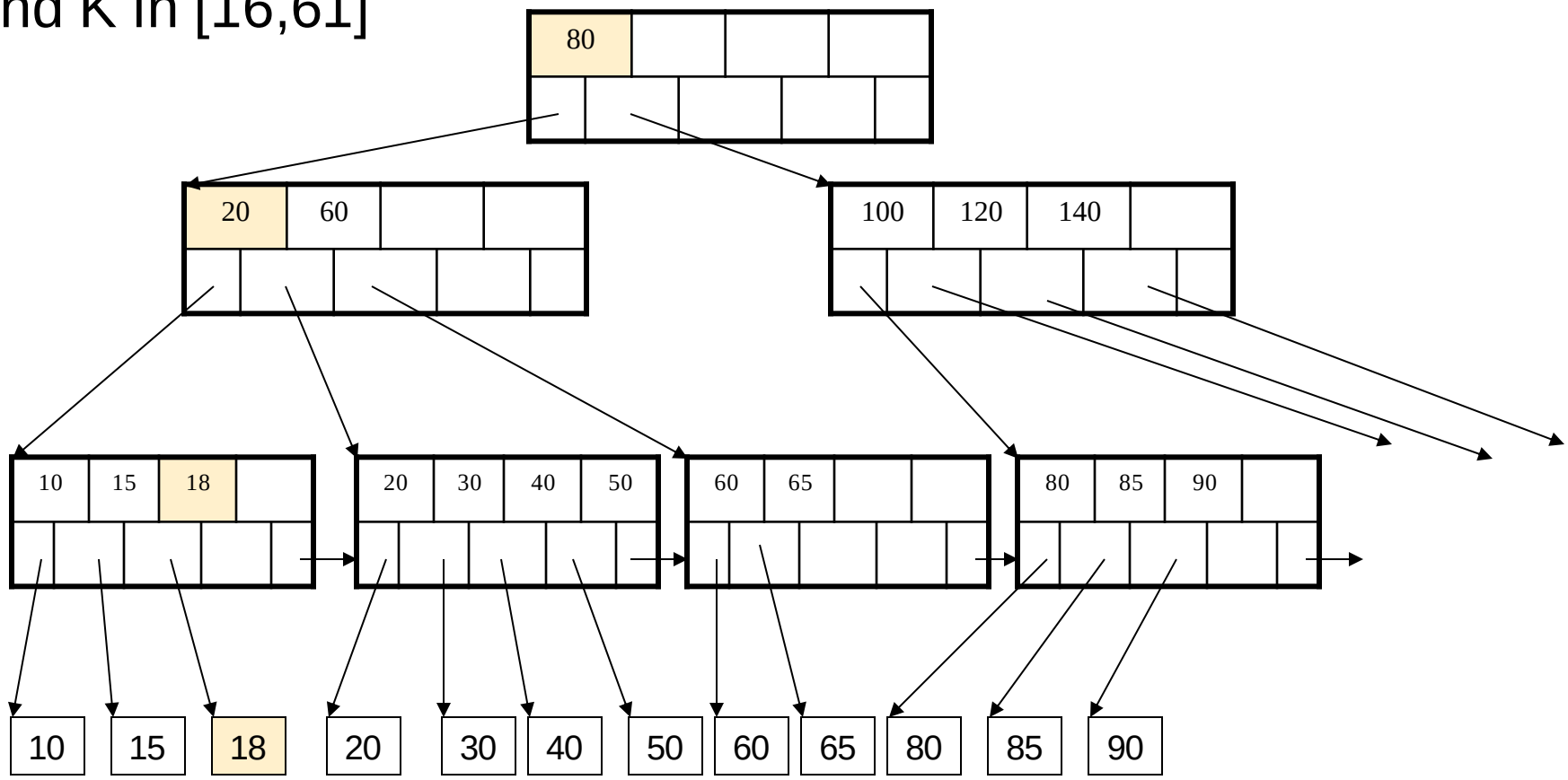
Range Search in B+ Tree

Find K in [16,61]



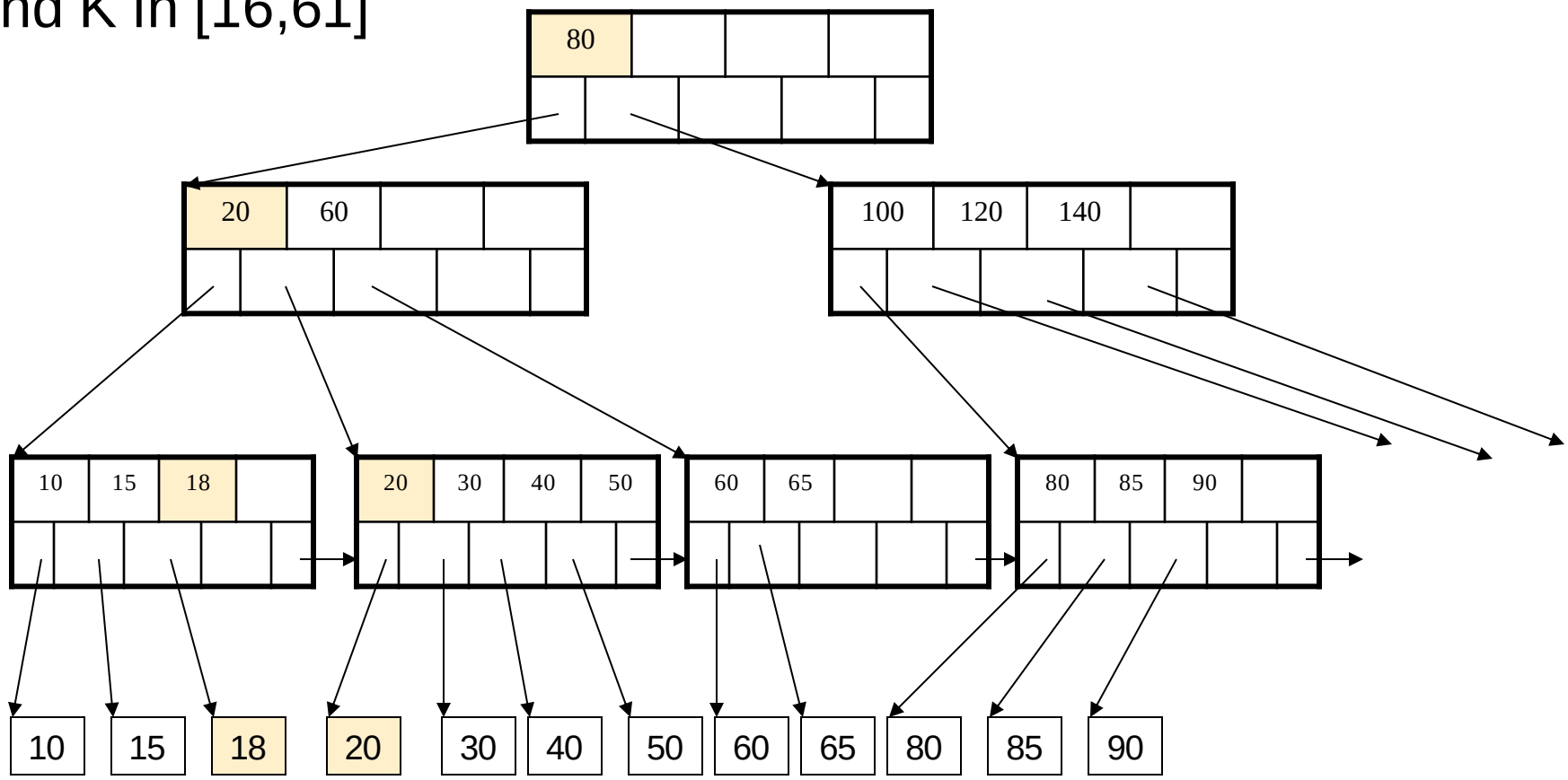
Range Search in B+ Tree

Find K in [16,61]



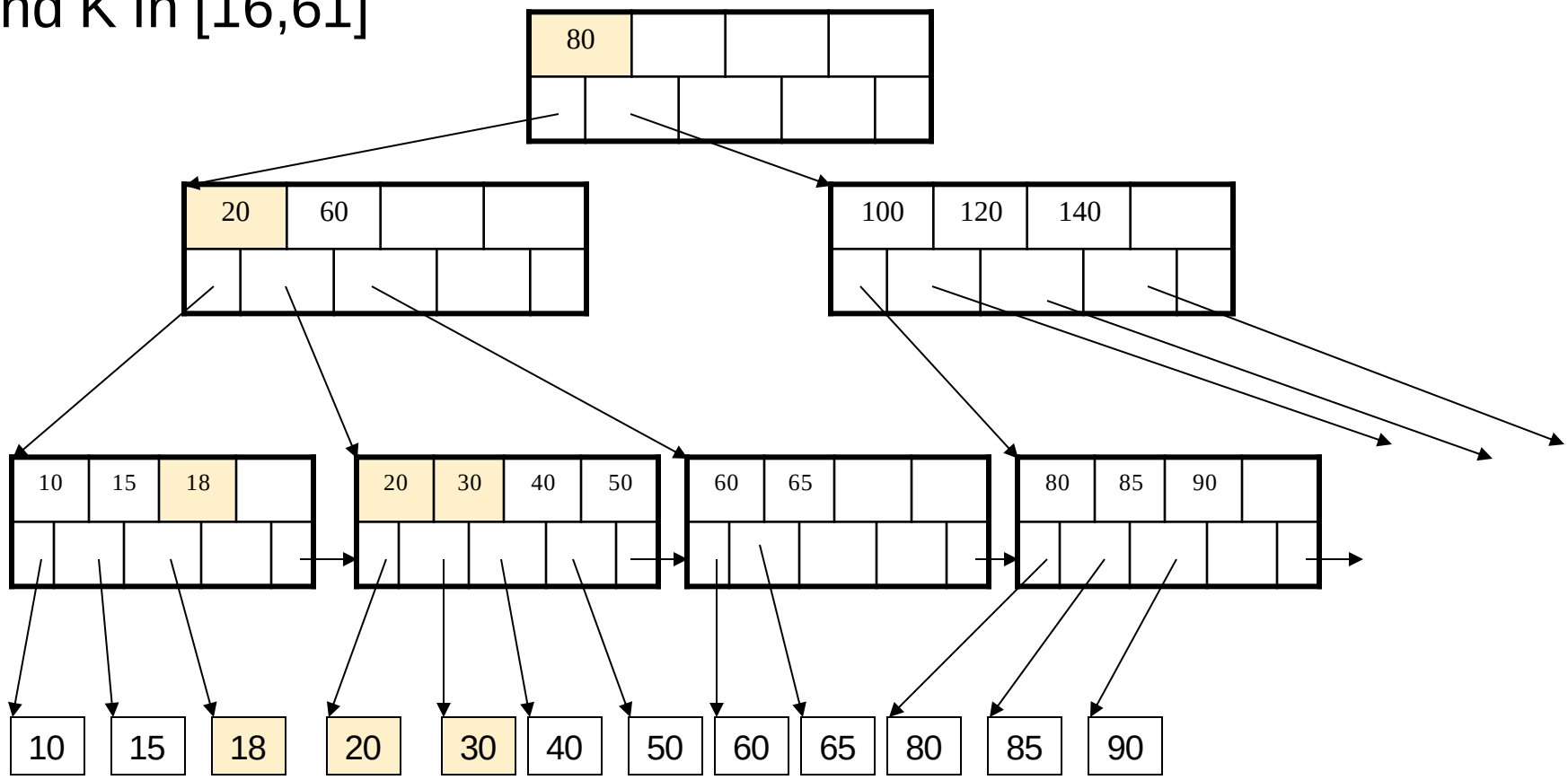
Range Search in B+ Tree

Find K in [16,61]



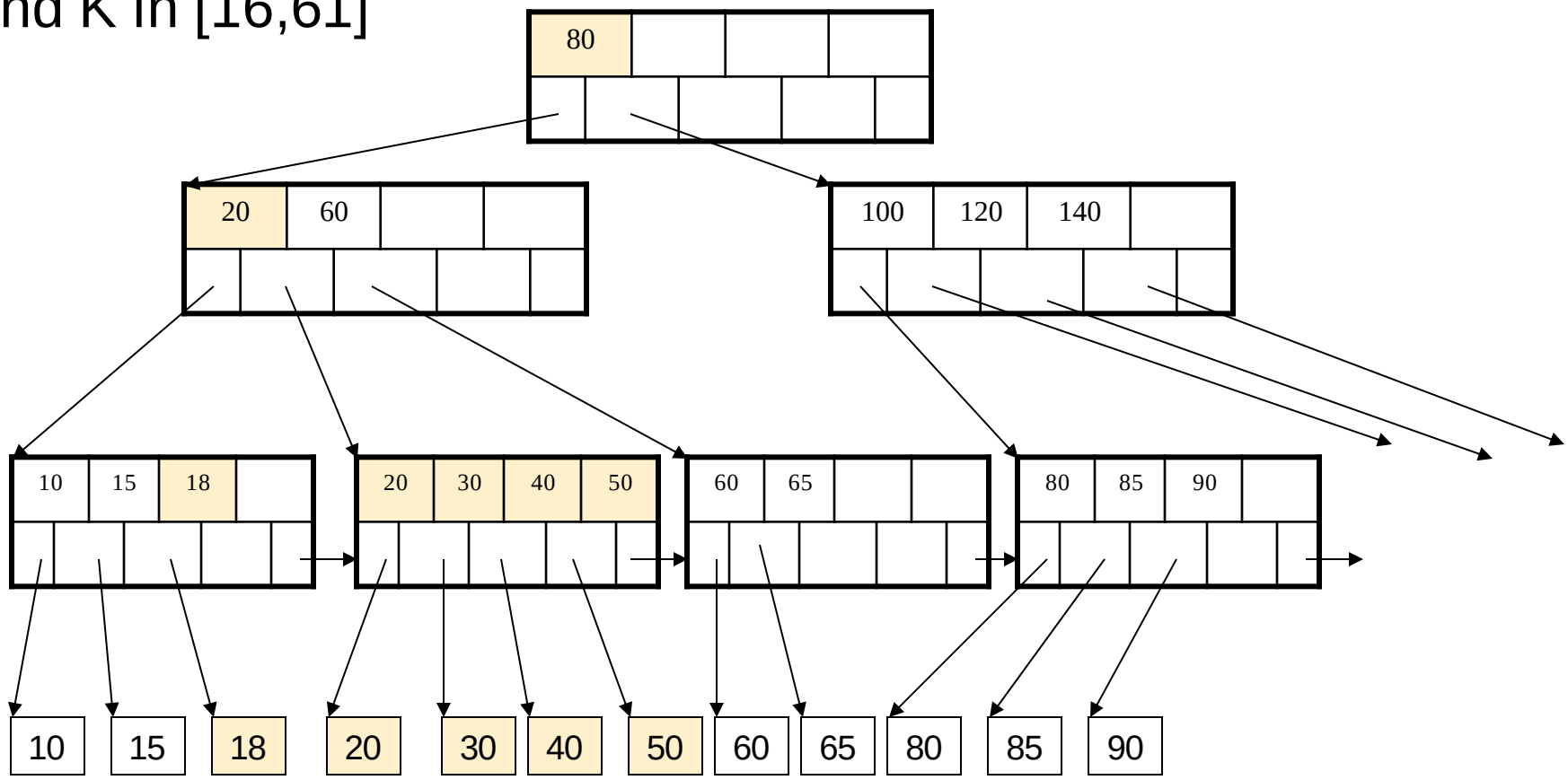
Range Search in B+ Tree

Find K in [16,61]



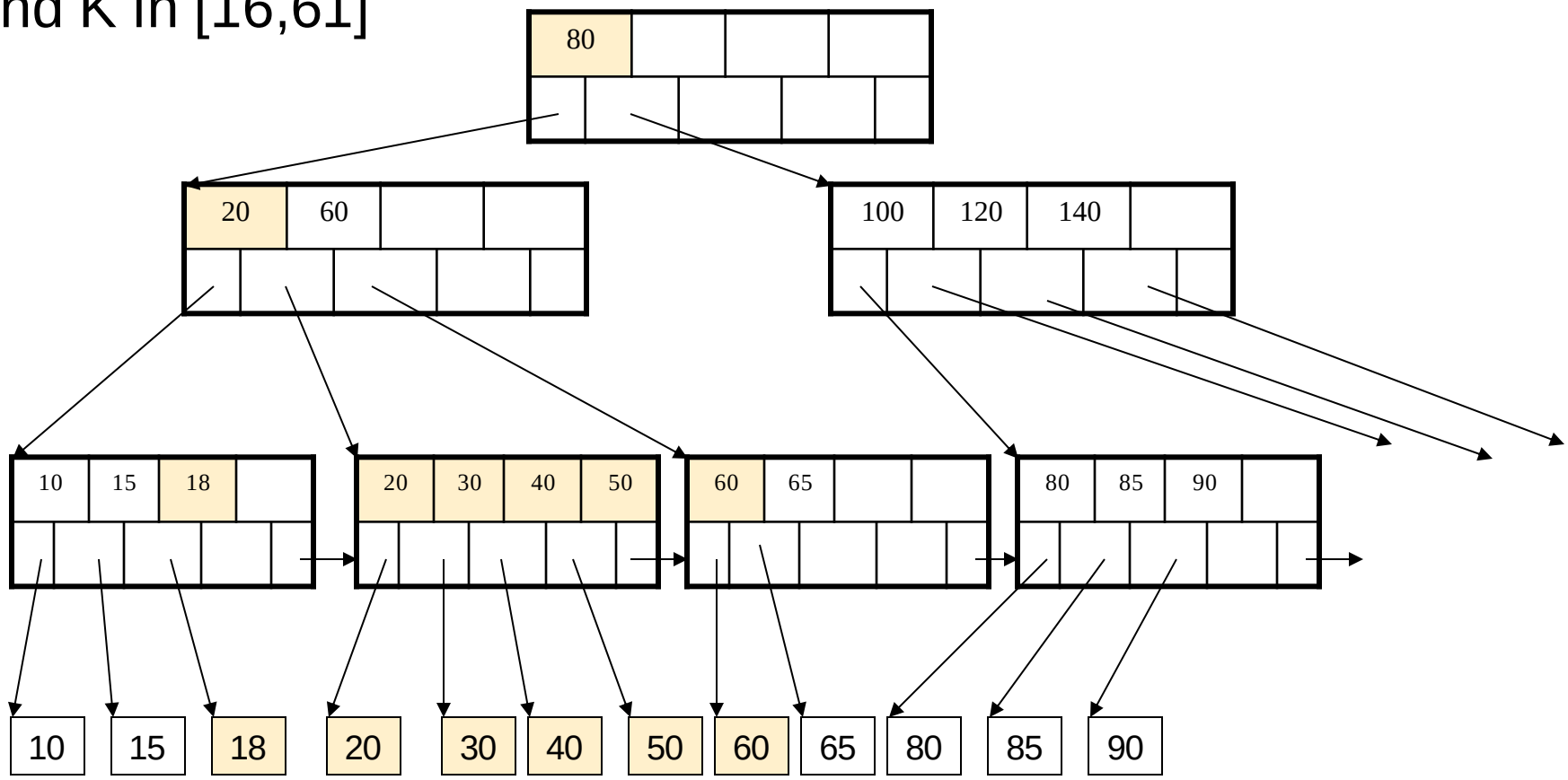
Range Search in B+ Tree

Find K in [16,61]



Range Search in B+ Tree

Find K in [16,61]

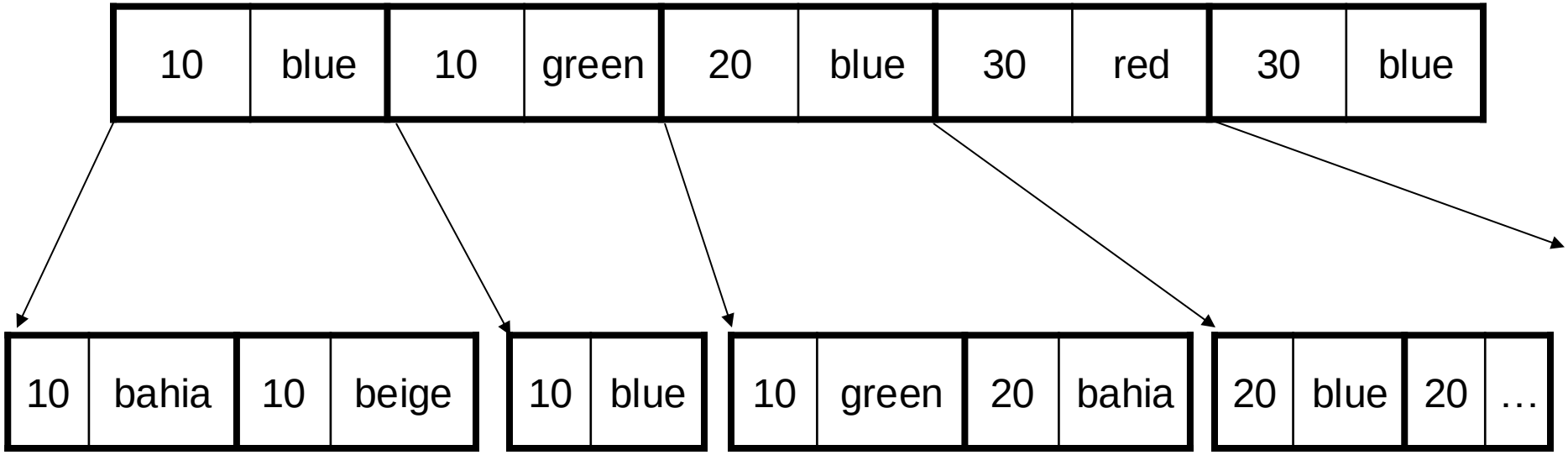


Multi-Attribute Index

- 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);
```

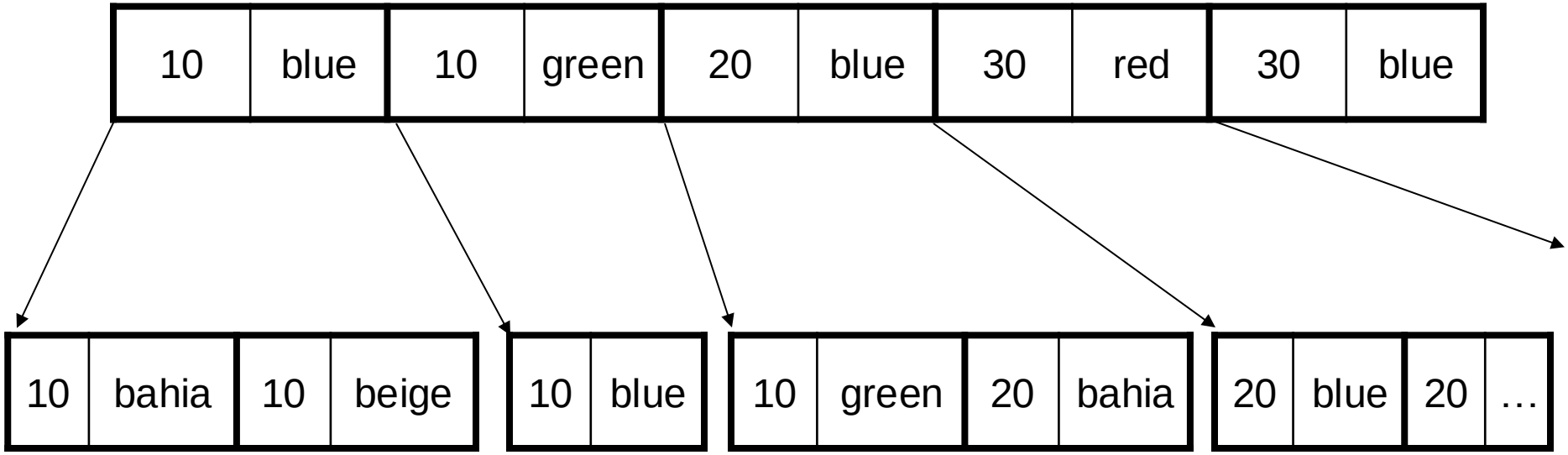
Multi-Attribute Index



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

Multi-Attribute Index

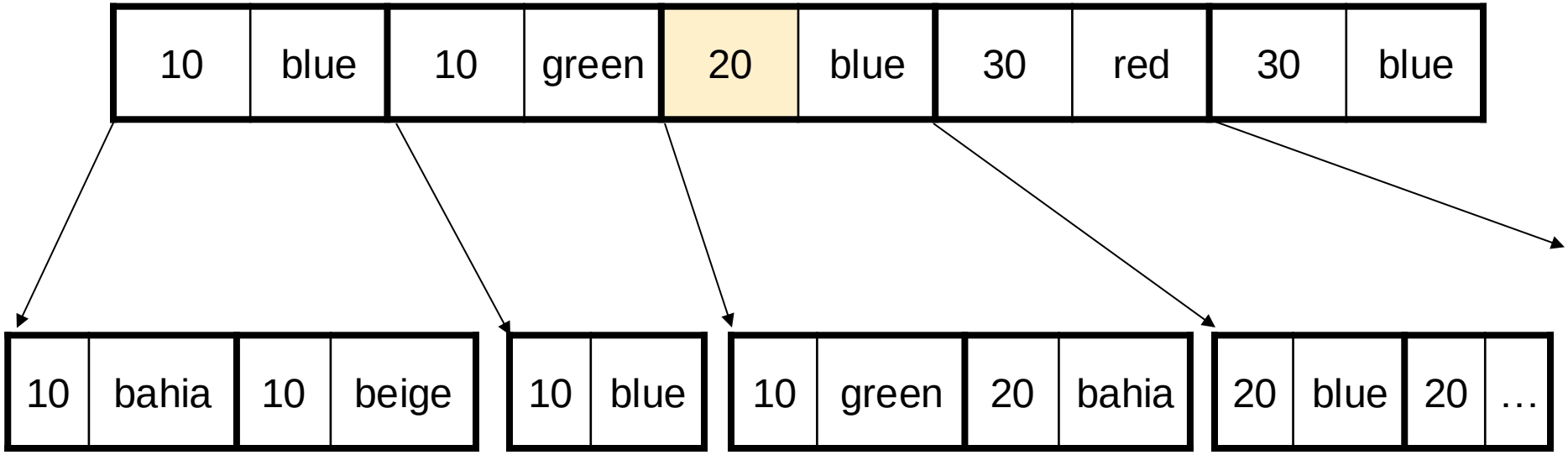
psize = 20, pcolor = *



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

Multi-Attribute Index

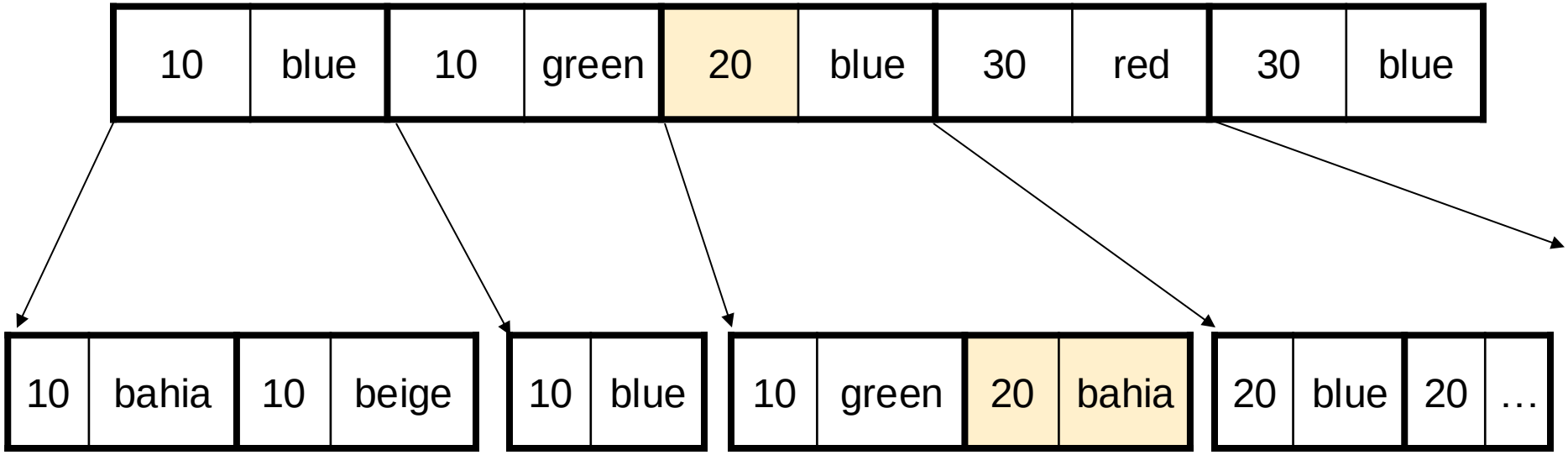
psize = 20, pcolor = *



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

Multi-Attribute Index

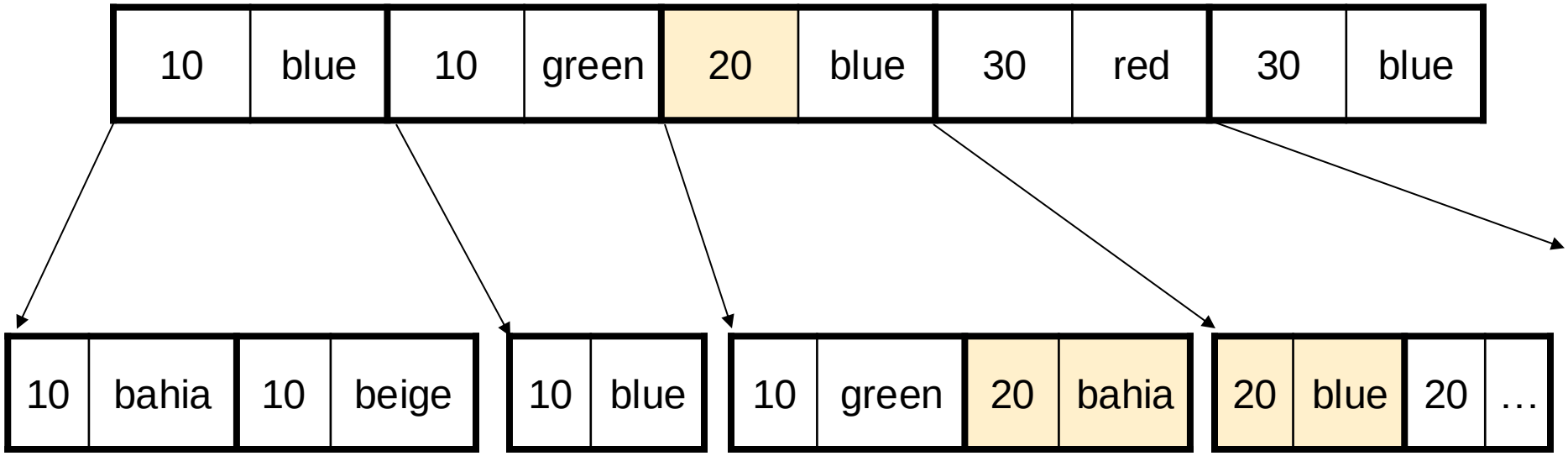
psize = 20, pcolor = *



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

Multi-Attribute Index

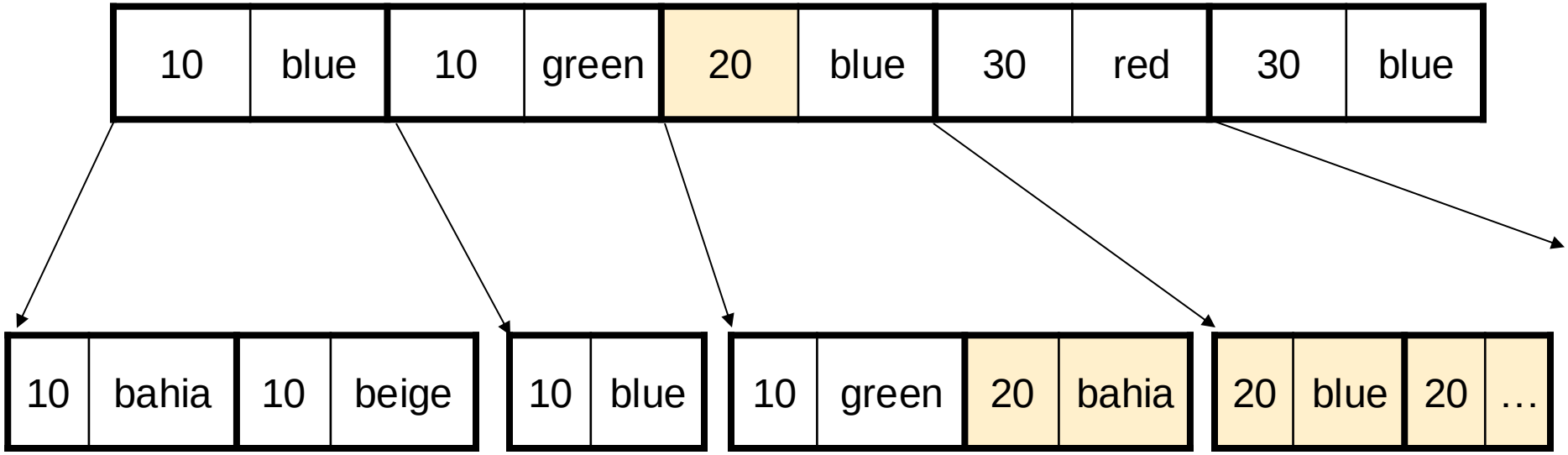
psize = 20, pcolor = *



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

Multi-Attribute Index

psize = 20, pcolor = *



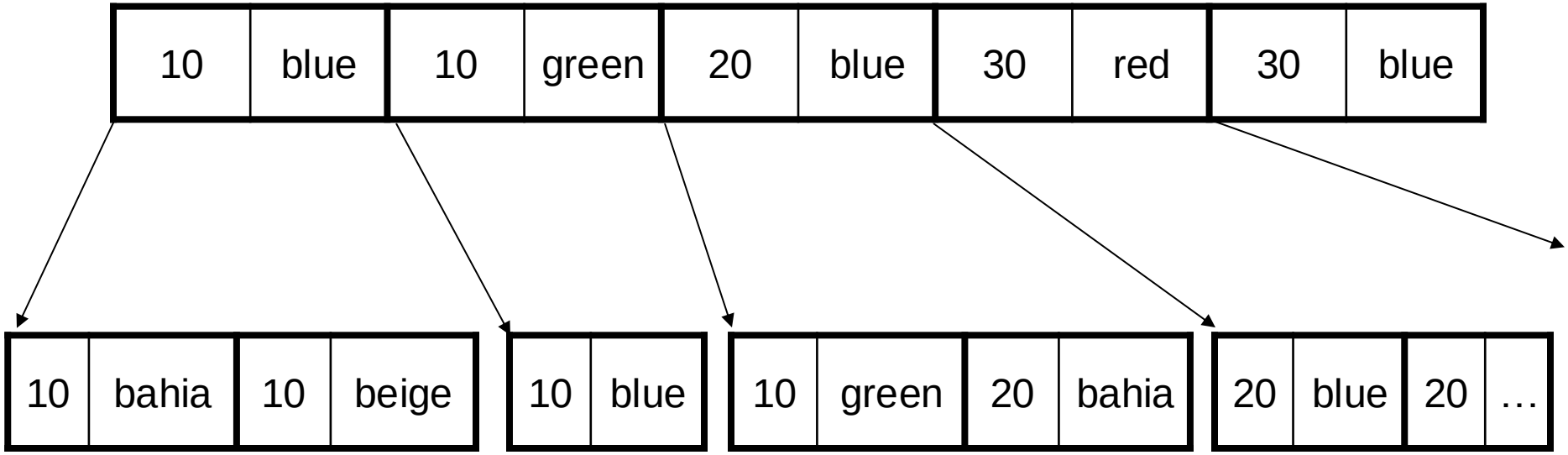
Index is useful

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

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue

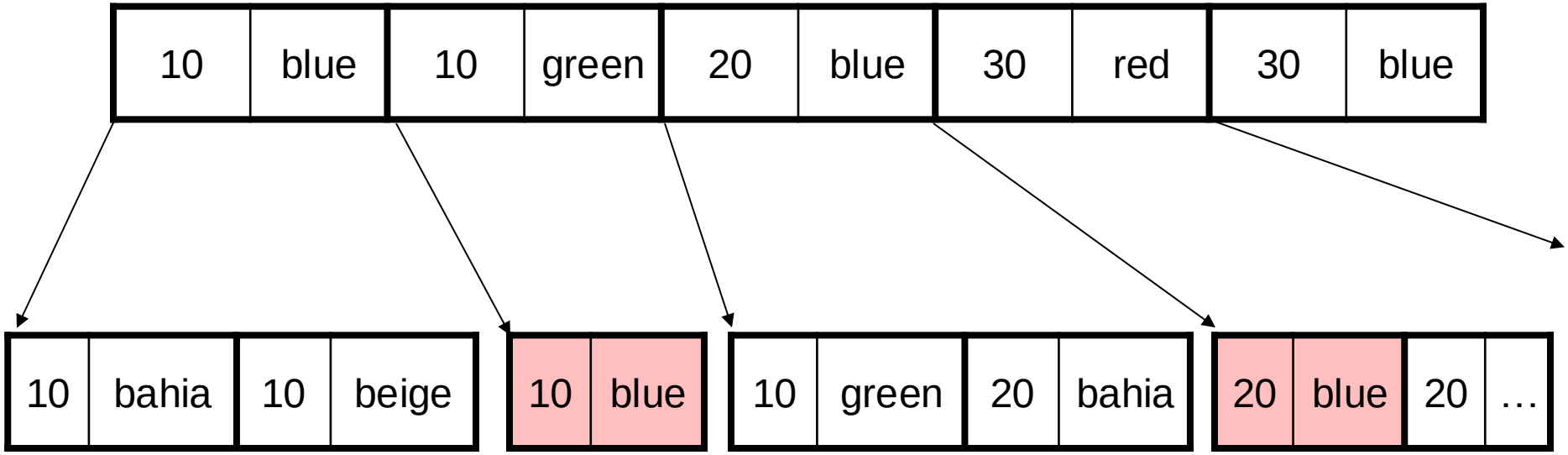


```
create index idxSC on Part(psize, pcolor);
```

Multi-Attribute Index

psize = 20, pcolor = *

psize=*, pcolor=blue



Index is not useful

Discussion

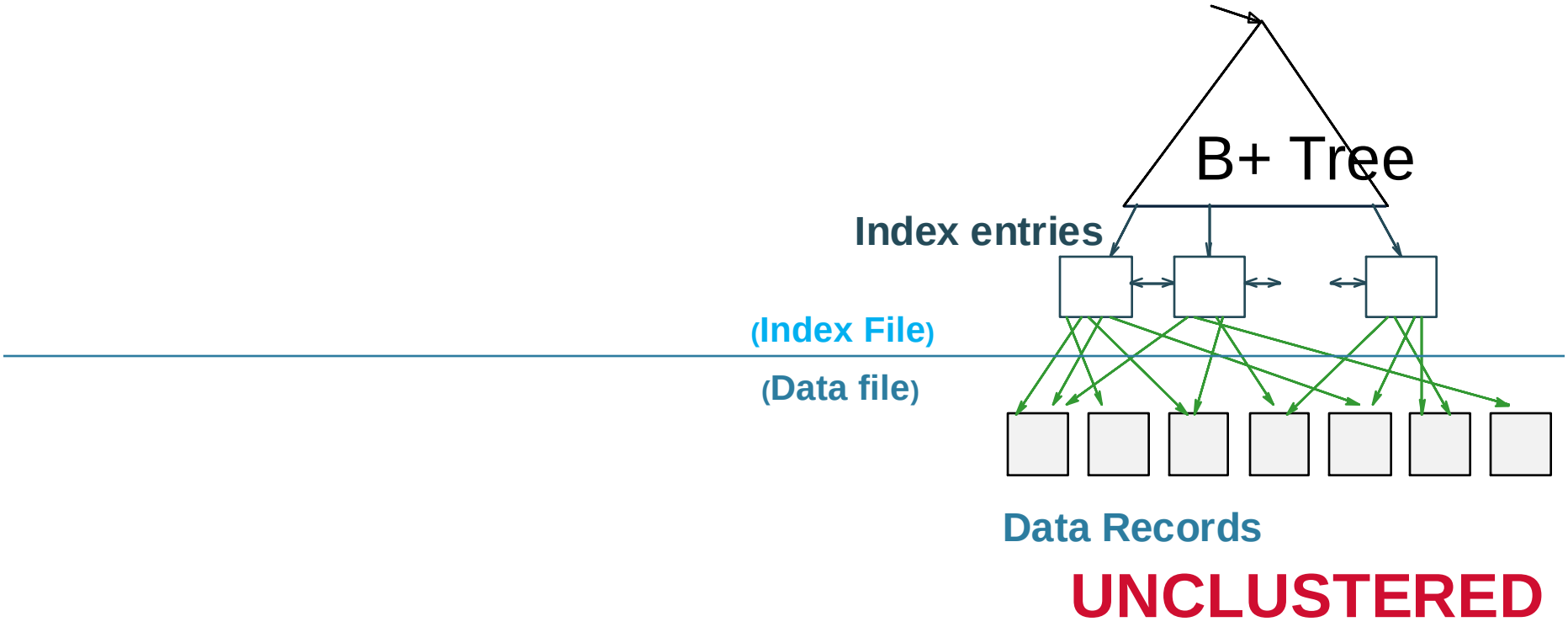
- So far, we assumed that the leaves are pointers to records in the data file
  - Pointer = (pageID, slotNumber)
  - Range search → random reads
- Solution: clustered index
  - Range search → sequential reads

Clustered Index

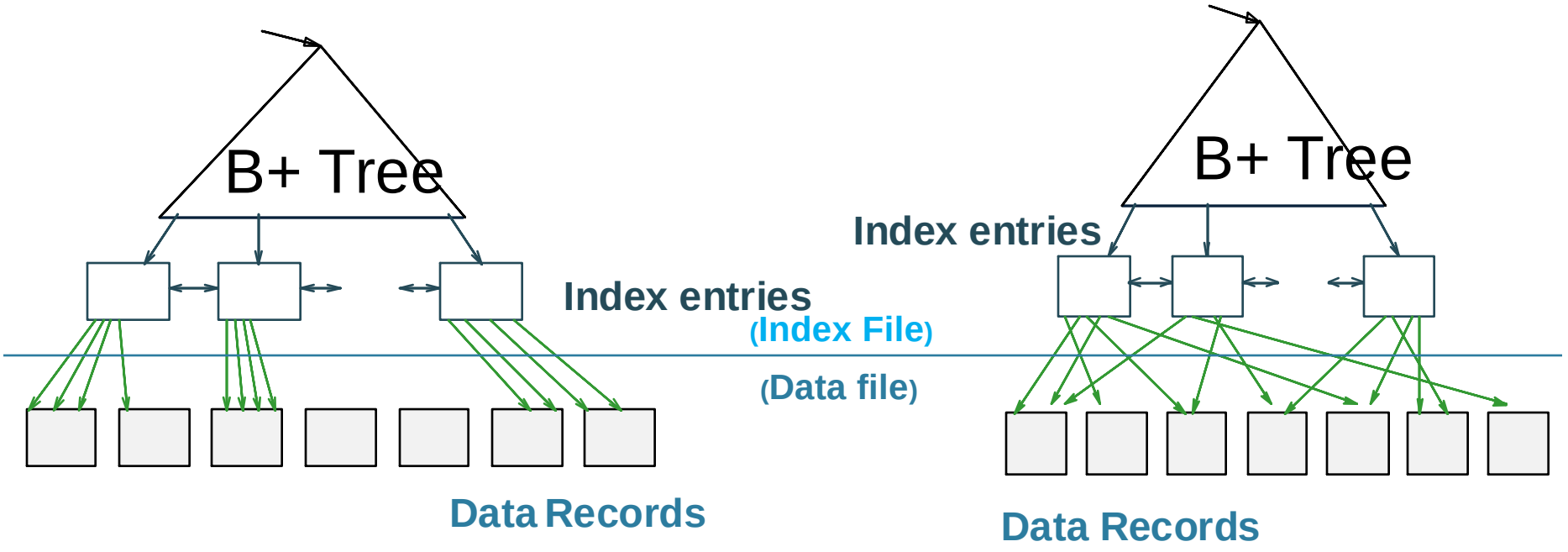
- A clustered index is sorted in the same order as the data file
  - Better: leaves of index ARE the data file
  - At most one clustered index per table

- An unclustered index: everything else
  - Many unclustered indexes per table

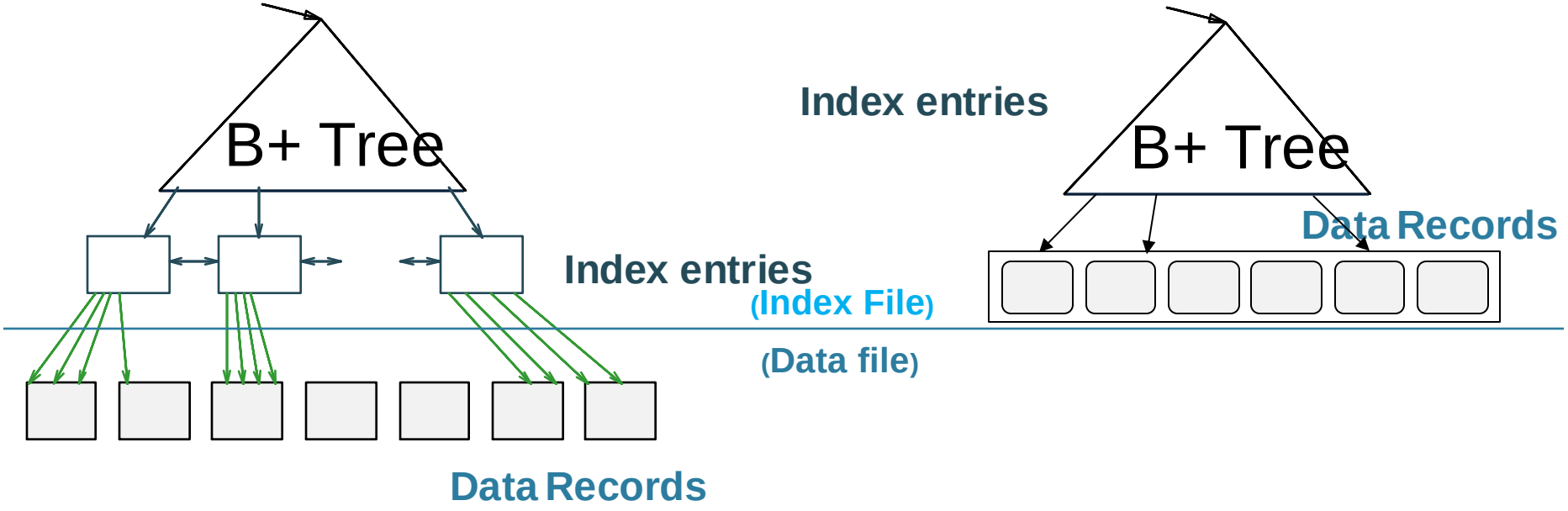
Clustered vs Unclustered



Clustered vs Unclustered



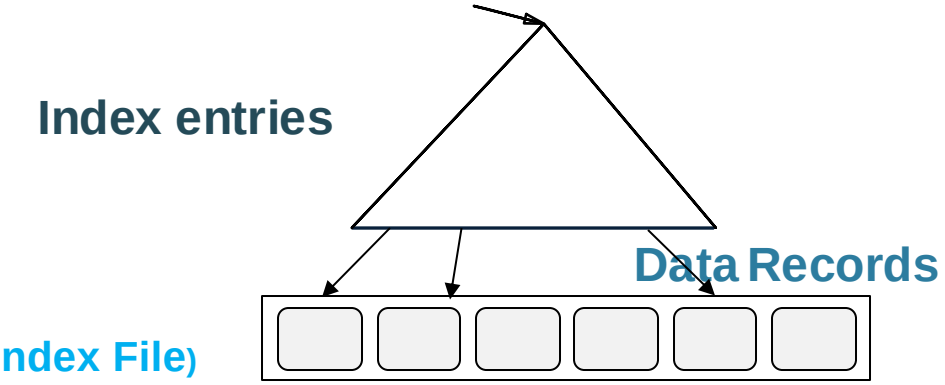
Clustered vs Unclustered



CLUSTERED

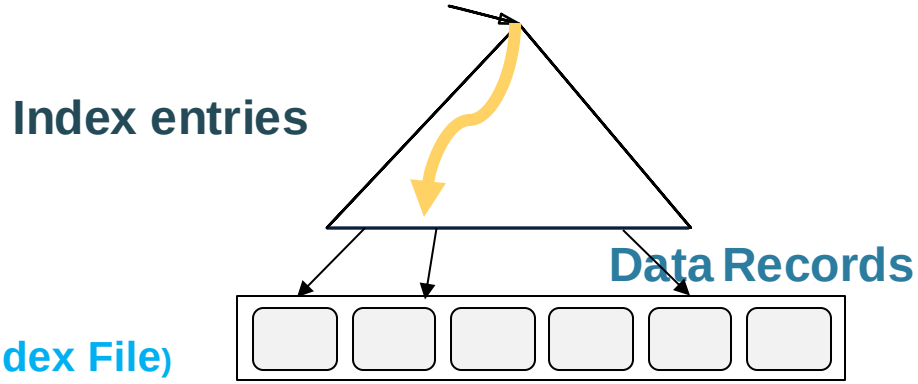
BETTER

Clustered vs Unclustered



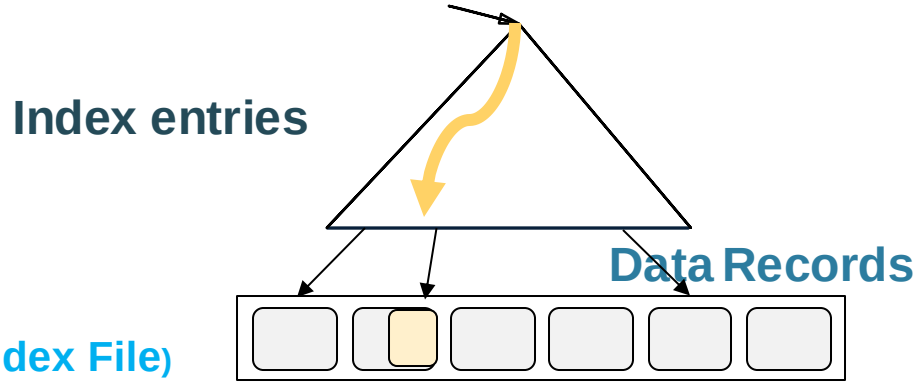
RANGE SEARCH

Clustered vs Unclustered



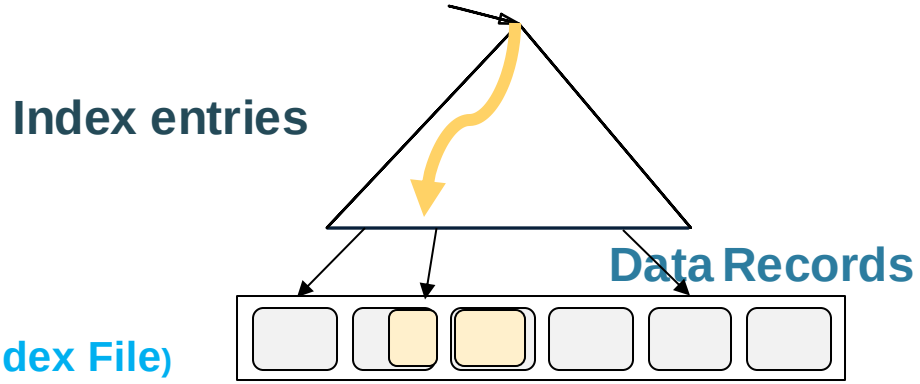
RANGE SEARCH

Clustered vs Unclustered



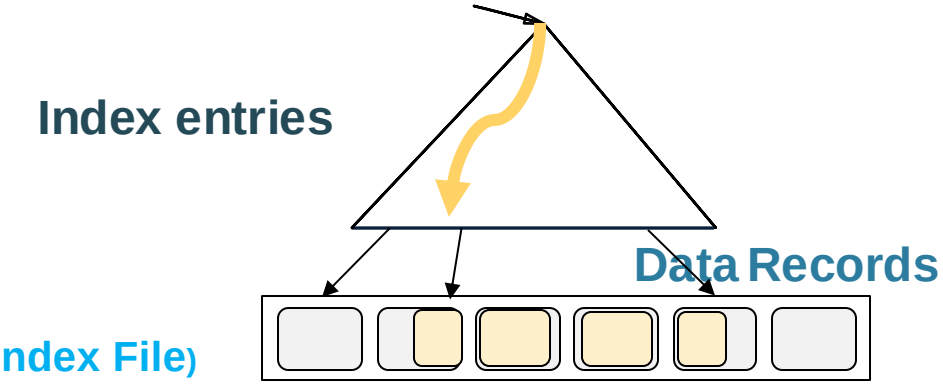
RANGE SEARCH

Clustered vs Unclustered



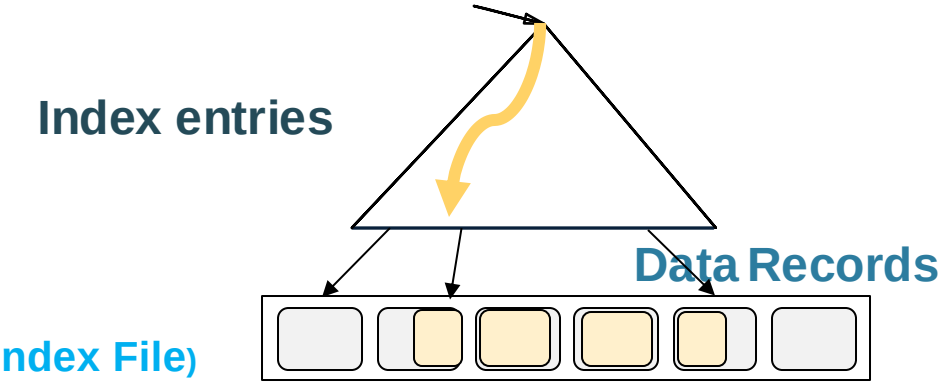
RANGE SEARCH

Clustered vs Unclustered



RANGE SEARCH

Clustered vs Unclustered



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

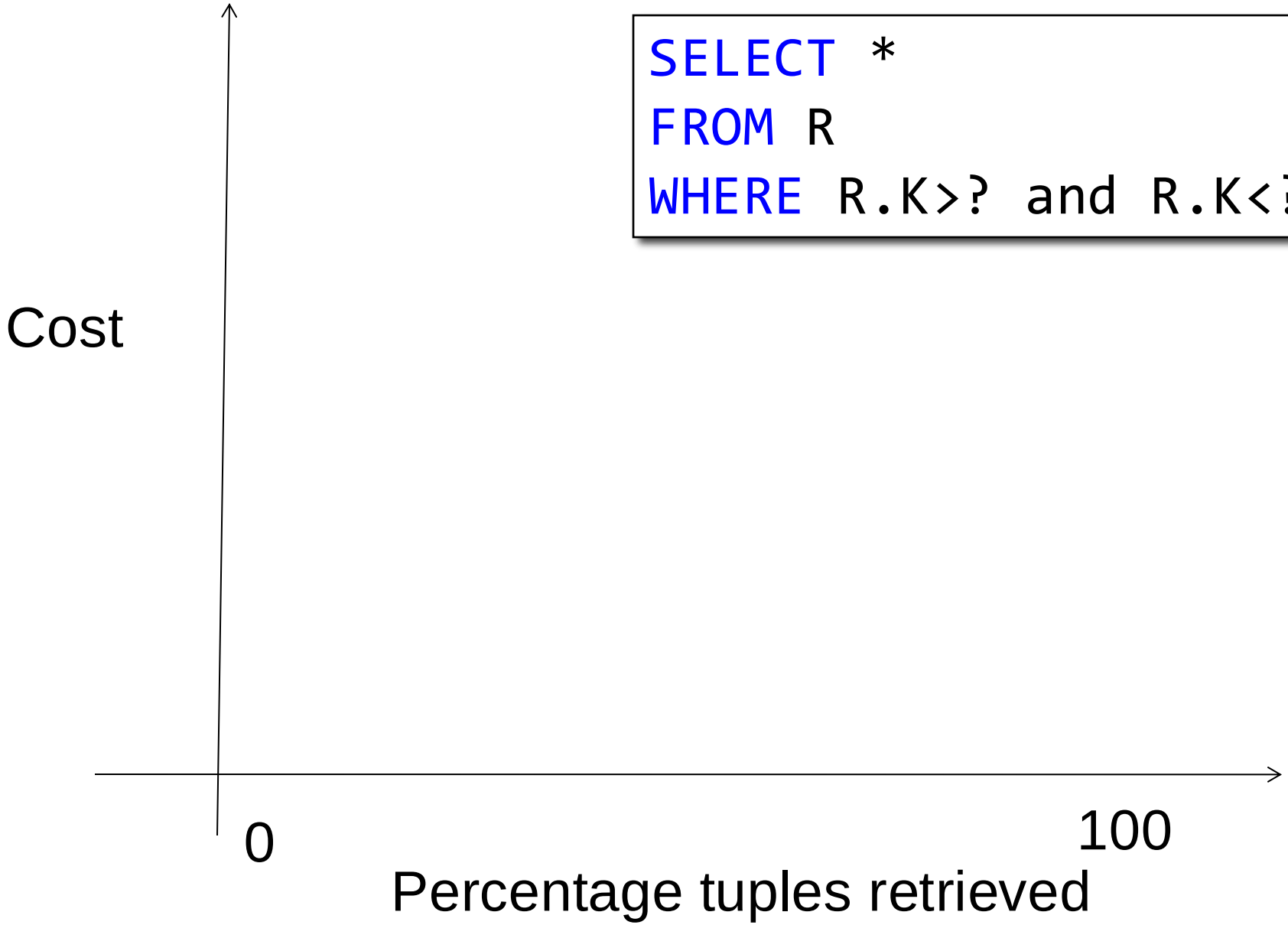
To Cluster or Not

Remember:

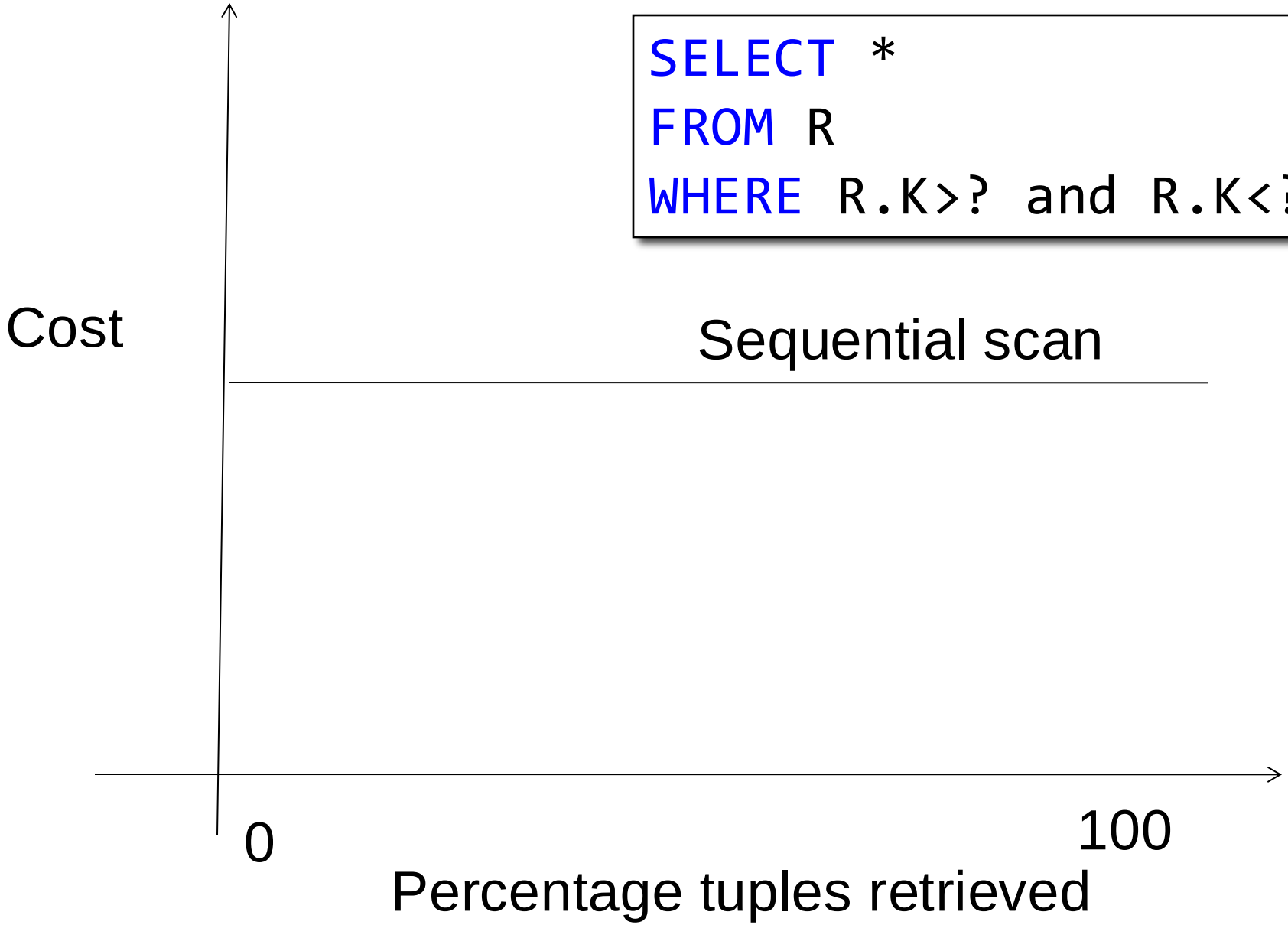
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

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

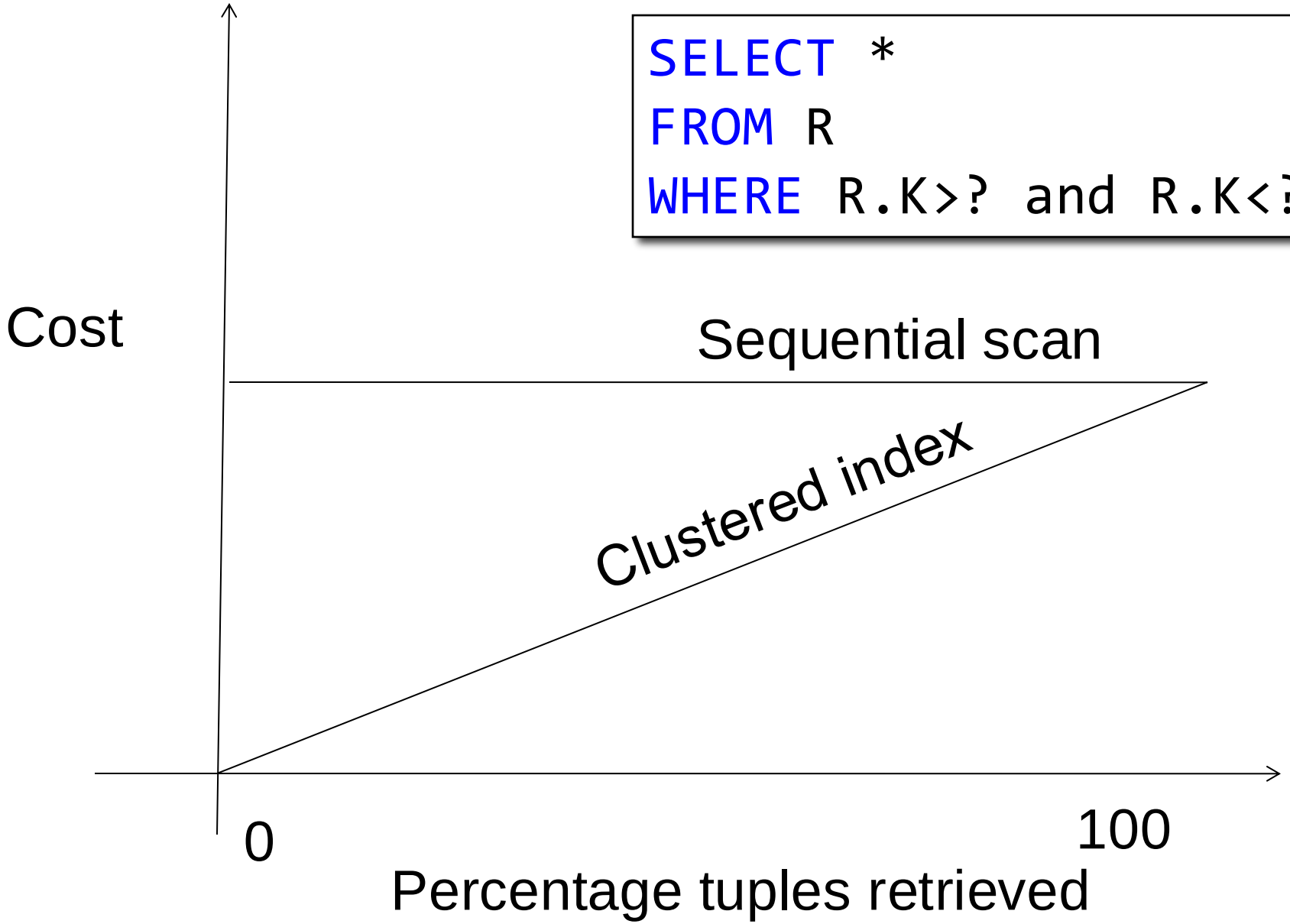
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```

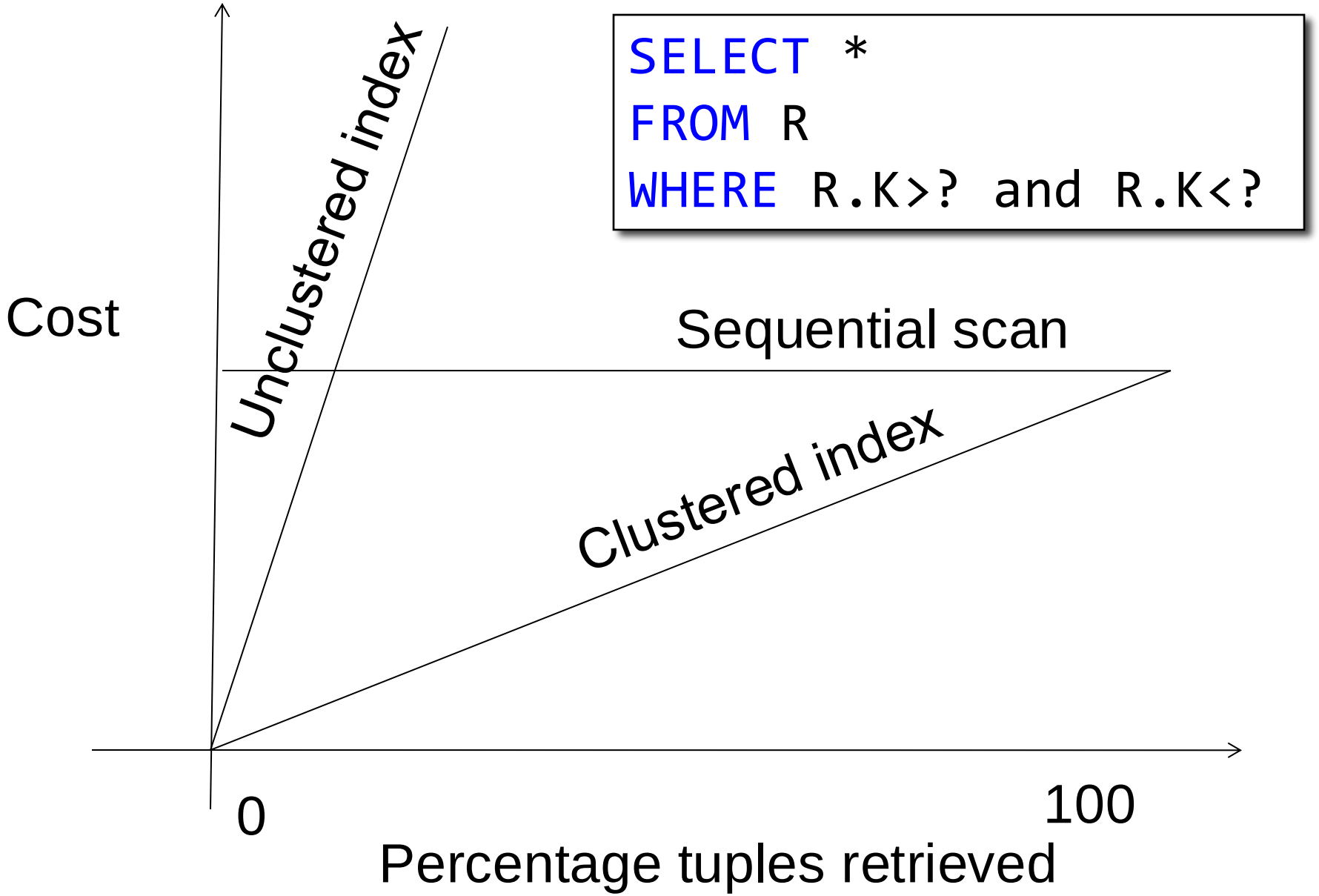


```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```
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