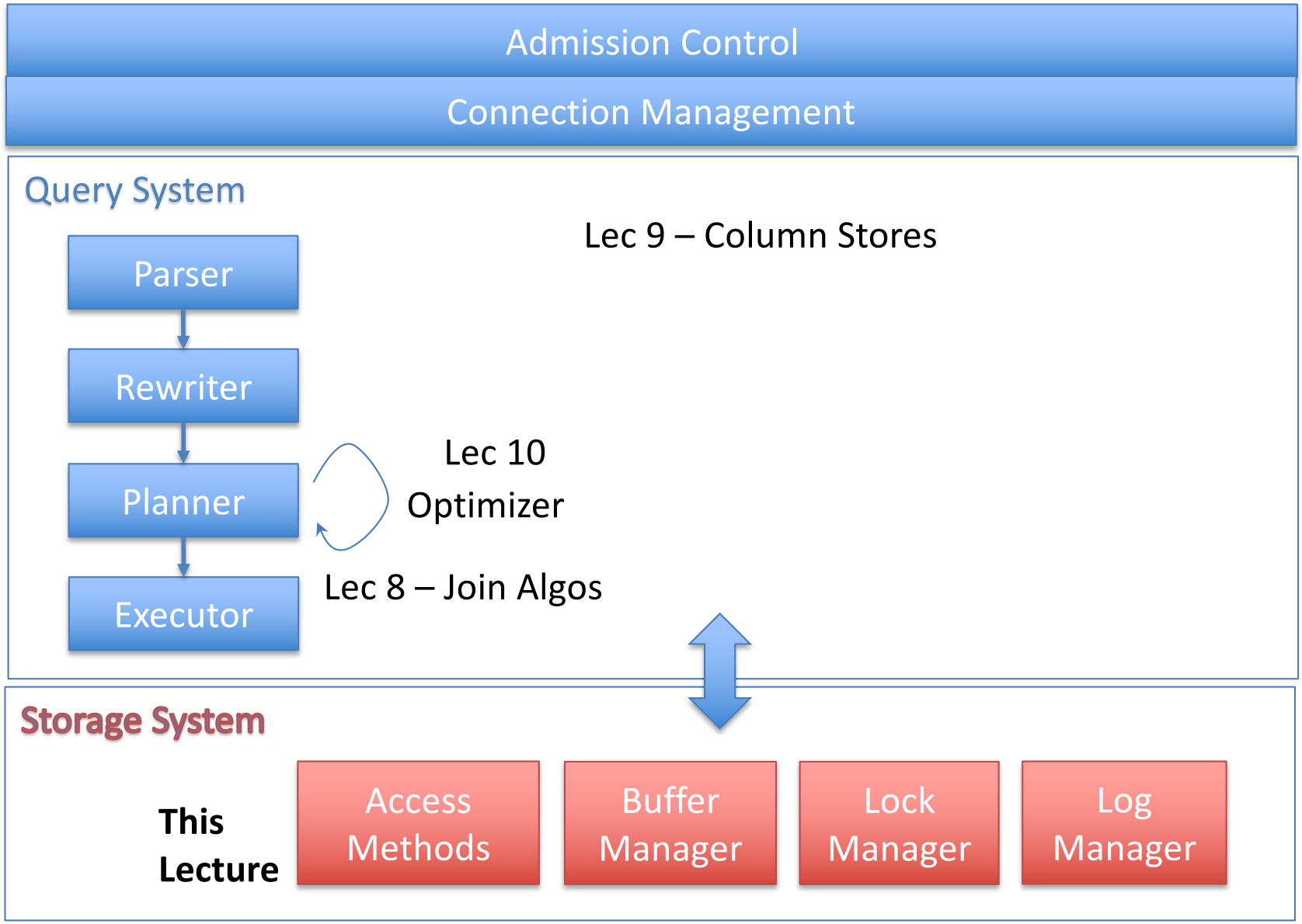
6.5830 Lecture 7

Florent Willems, “The Accountant”

September 28, 2022

Cost Estimation and Indexing

Plan for Next Few Lectures



Recap: Bandwidth vs Latency

- 1st access latency often high relative to the rate device can stream data sequentially (bandwidth)

- RAM: 50 ns per 16 B cache line (100x difference)

  → random access bandwidth of 16 * $1/5x10^{-8}$ = 320 MB / sec

  If streaming sequentially, bandwidth 20-40 GB/sec

- Flash disk: 250 us per 4K page (125x difference)

  → Random access bandwidth of 4K * $1/2.5x10^{-4}$= 16 MB / sec

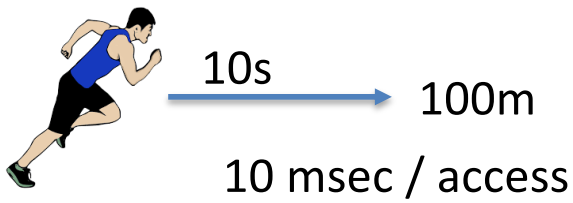
  If streaming sequentially, bandwidth 2+ GB/sec

Important Numbers

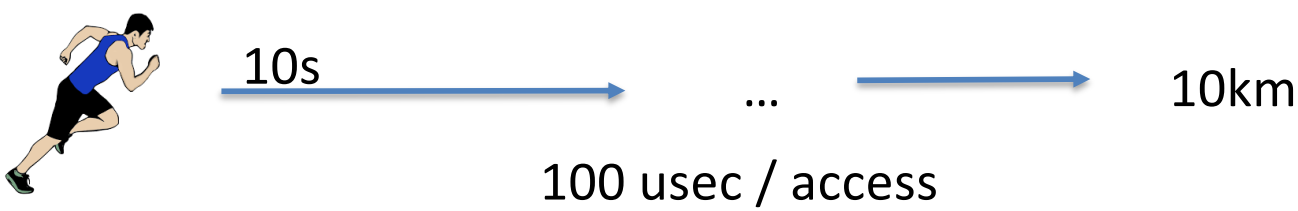
CPU Cycles / Sec	2+ Billion (.5 nsec latency)
L1 latency	2 nsec (4 cycles)
L2 latency	6 nsec (12 cycles)
L3 latency	18 nsec (36 cycles)
Main latency	50 – 100 ns (150-300 cycles)
Sequential Mem Bandwidth	20-40+ GB/sec
SSD Latency	250+ usec
SSD Seq Bandwidth	2-4 + GB/sec
HD (spinning disk) latency	10 msec
HD Seq Bandwidth	100+ MB
Local Net Latency	10 – 100 usec
Local Net Bandwidth	1 – 40 Gbit /sec
Wide Area Net Latency	10 – 100 msec
Wide Area Net Bandwidth	100 – 1 Gbit / sec

Speed Analogy

Disk



Flash



Main Memory

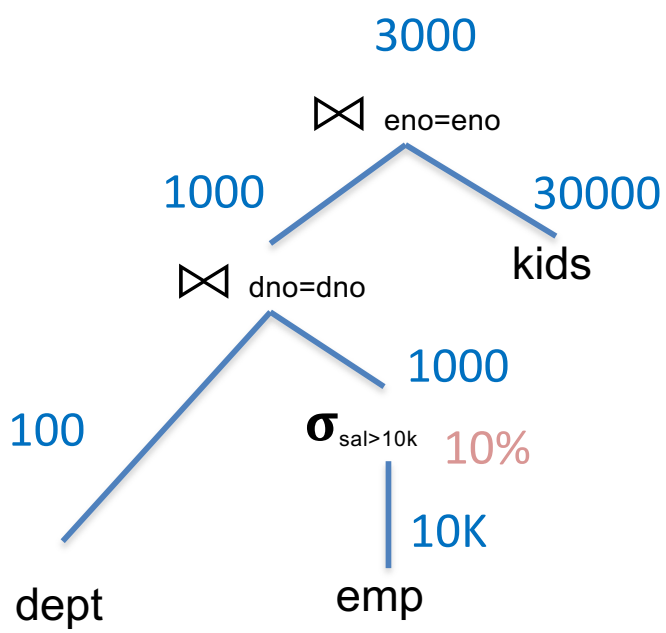
Database Cost Models

- Typically try to account for both CPU and I/O
  - I/O = ”input / output”, i.e., data access costs from disk
- Database algorithms try to optimize for sequential access (to avoid massive random access penalties)
- Simplified cost model for 6.5830:

  # seeks (random I/Os) x random I/O time +
  sequential bytes read x sequential B/W

Example

SELECT * FROM emp, dept, kids
WHERE sal > 10k
AND emp.dno = dept.dno
AND emp.eid = kids.eid



100 tuples/page
10 pages RAM
10 KB/page

|dept| = 100 records = 1 page = 10 KB
|emp| = 10K = 100 pages = 1 MB
|kids| = 30K = 300 pages = 3 MB

Spinning Disk:
10 ms / random access page
100 MB/sec sequential B/W

Assume nested loops joins, no indexes

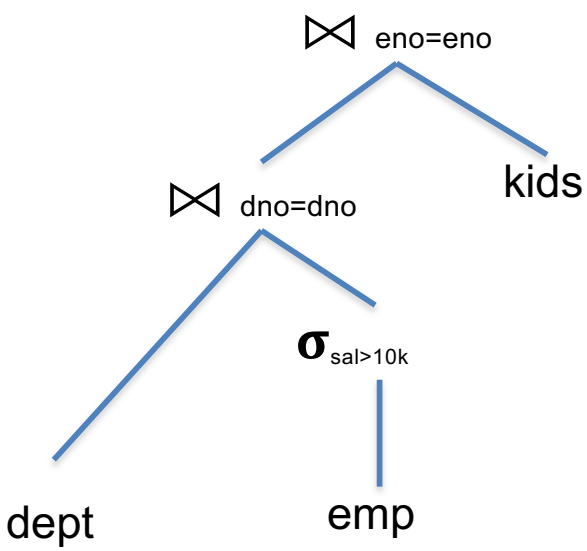
Example w/ Simple Cost Model

seeks (random disk I/Os) x random I/O time + sequential bytes read / sequential disk B/W

100 tuples/page
10 pages RAM
10 KB/page

|deptl = 100 records = 1 page = 10 KB
|empl = 10K = 100 pages = 1 MB
|kidsl = 30K = 300 pages = 3 MB

Spinning Disk:
10 ms / random access page
100 MB/sec sequential B/W



<u>Dept is outer in NL Join of dept/emp join:</u>

1 scan of dept
100 scans of emp (cannot cache)

<u>Dept is inner in NL Join:</u>

1 scan of emp
10k scans of dept?
No, because it can be cached

Study Break

- Assuming disk can do 100 MB/sec I/O, and 10ms / seek
- And the following schema:

```
grades (cid int, g_sid int, grade char(2))
students (s_int, name char(100))
```

1. Estimate time to sequentially scan grades, assuming it contains 1M records (Consider: field sizes, headers)

2. Estimate time to join these two tables, using nested loops, assuming students fits in memory but grades does not, and students contains 10K records.

Seq Scan Grades

`grades (cid int, g_sid int, grade char(2))`

- `8 bytes (cid) + 8 bytes (g_sid) + 2 bytes (grade) + 4 bytes (header) = 22 bytes`
- 22 x 1M = 22 MB / 100 MB/sec = .22 sec + 10ms seek

➔ .23 sec

NL Join Grades and Students

```
grades (cid int, g_sid int, grade char(2))
students (s_int, name char(100))
```

10 K students x (100 + 8 + 4 bytes) = 1.1 MB

<u>Students Inner (Preferred)</u>

- Cache students in buffer pool in memory: 1.1/100 s = .011 s
- One pass over students (cached) for each grade (no additional cost beside caching)
- Time to scan grades (previous slide) = .23 s

➔ .244 s

<u>Grades Inner</u>

- One pass over grades for each student, at .22 sec / pass, plus one seek at 10 ms (.01 sec) ➔ .23 sec / pass

➔ 2300 seconds overall

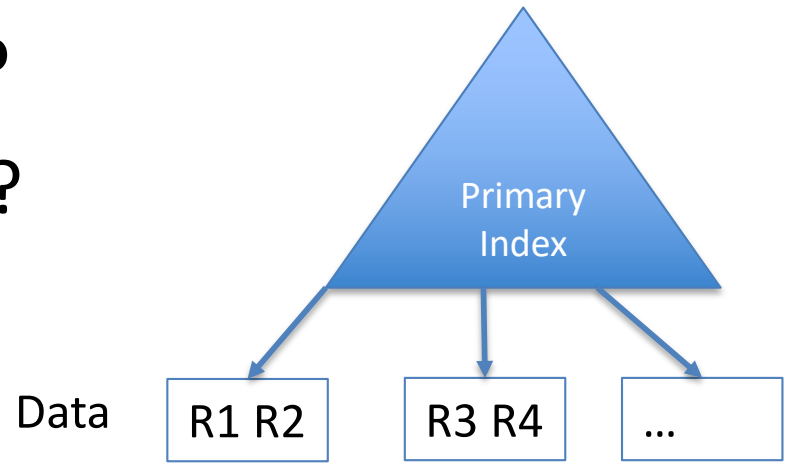
- (Time to scan students is .011 s, so negligible)

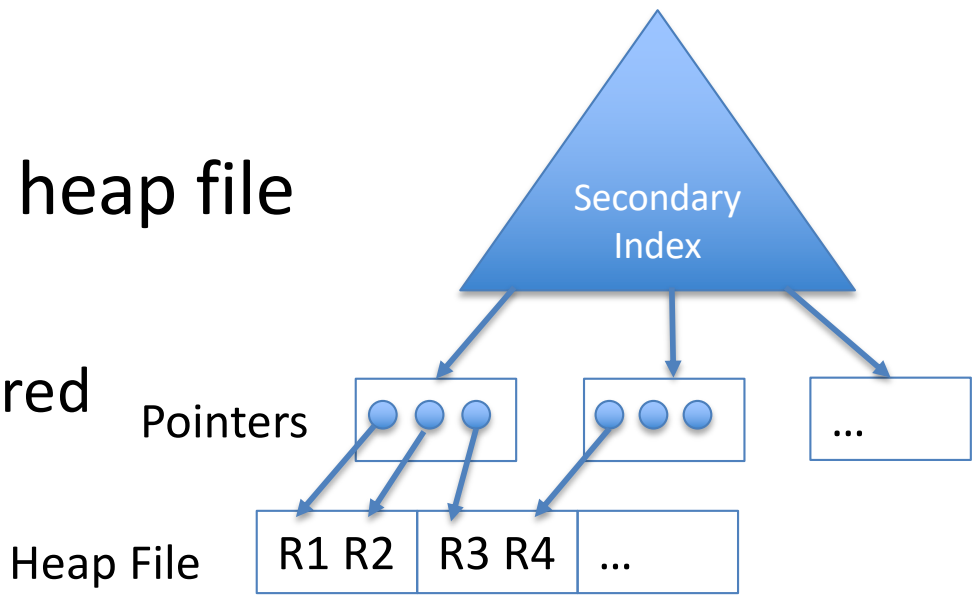
Today: Access Methods

- Access method: way to access the records of the database
- 3 main types:
  - Heap file / heap scan
  - Hash index / index lookup
  - B+Tree index / index lookup / scan ← next time
- Many alternatives: e.g., R-trees ← next time
- Each has different performance tradeoffs

Design Considerations for Indexes

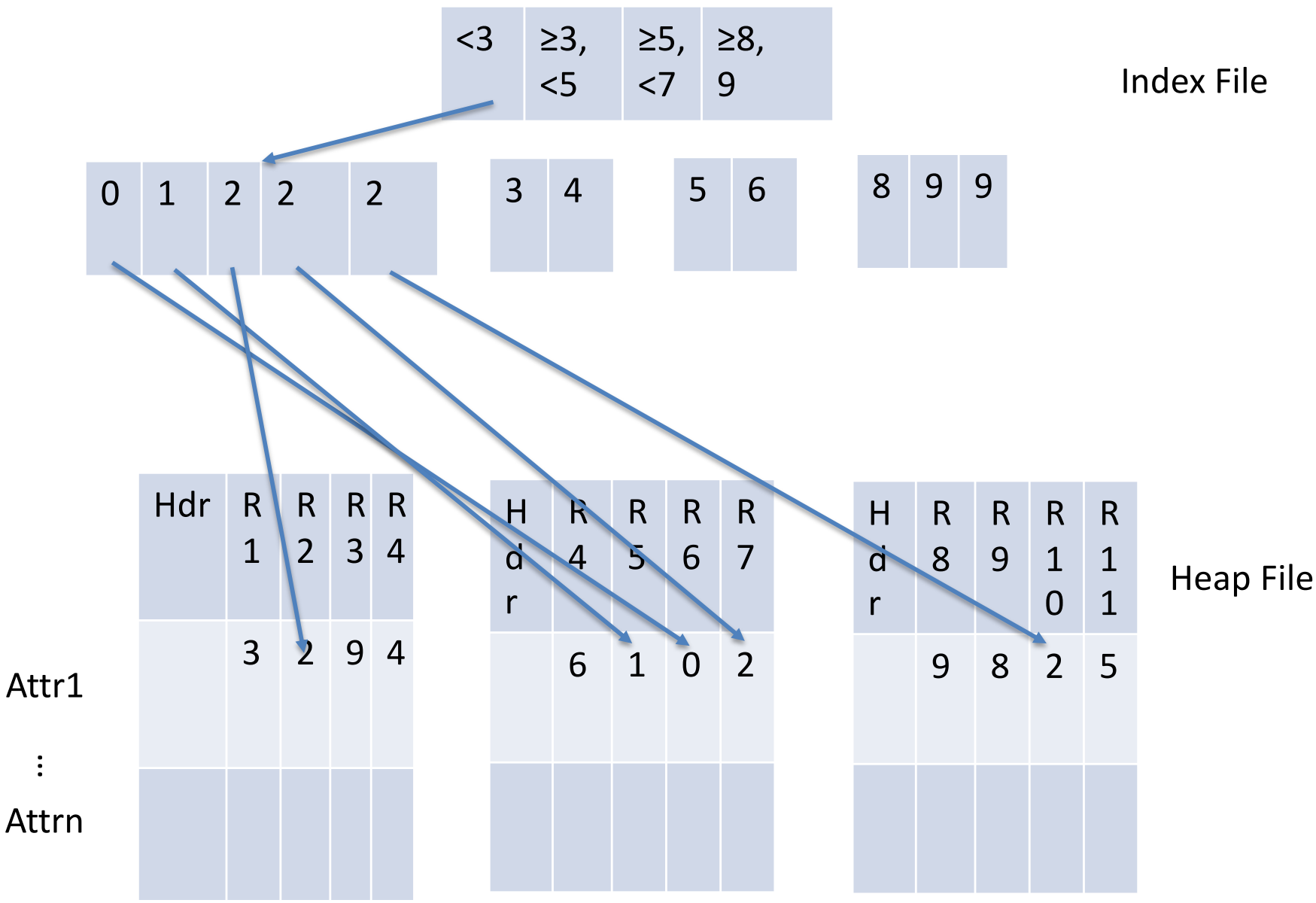
- What attributes to index?
  - Why not index everything?
- Index structure:
  - Leaves as data
    - Only one index?
    - "Primary Index"
  - Leaves as pointers to heap file
    - "Secondary Index"
    - Clustered vs unclustered





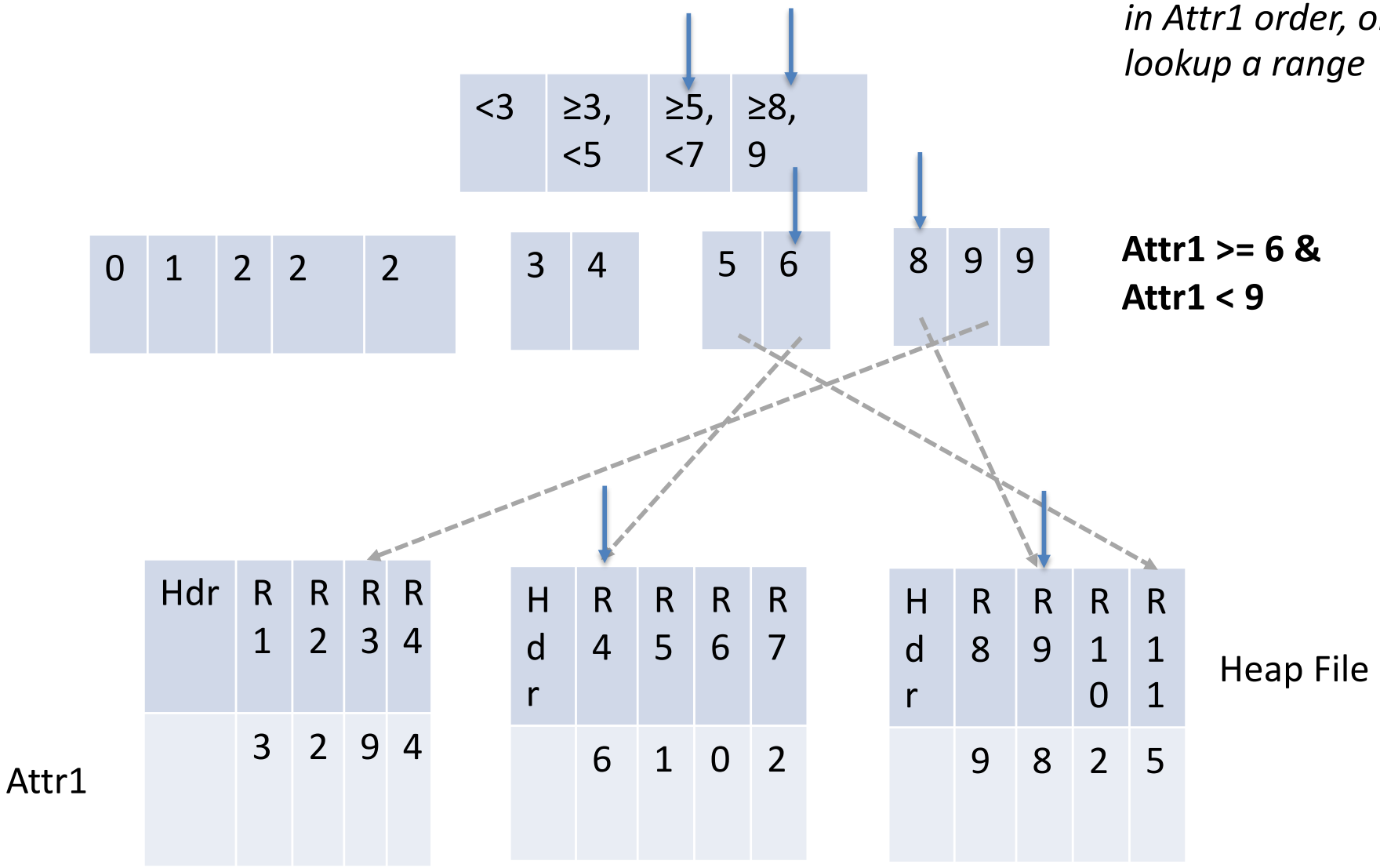
In 6.5830 we will use secondary indexes, and distinguish between clustered and unclustered

Tree Index


Index File
<3
≥3, <5
≥5, <7
≥8, 9
0 1 2 2 2
3 4
5 6
8 9 9
Heap File
Hdr R1 R2 R3 R4
3 2 9 4
Hdr R4 R5 R6 R7
6 1 0 2
Hdr R8 R9 R10 R11
9 8 2 5
Attr1
⋮
Attrn

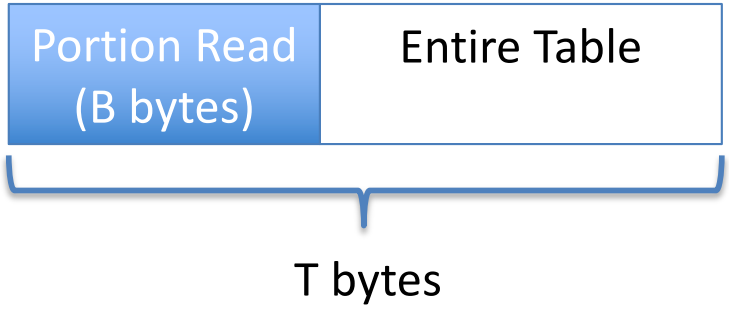
Index Scan

Traverse the records in Attr1 order, or lookup a range



Note random access! – this is an "unclustered" index

Costs of Random Access



- Consider an SSD with 100 usec latency, 1 GB/sec BW
- Query accesses B bytes, R bytes per record, whole table is T bytes
- Seq scan time S = T / 1GB/sec
- Rand access via index time = 100 usec * B/R + B / 1GB/sec
- Suppose R is 100 bytes, T is 10 GB

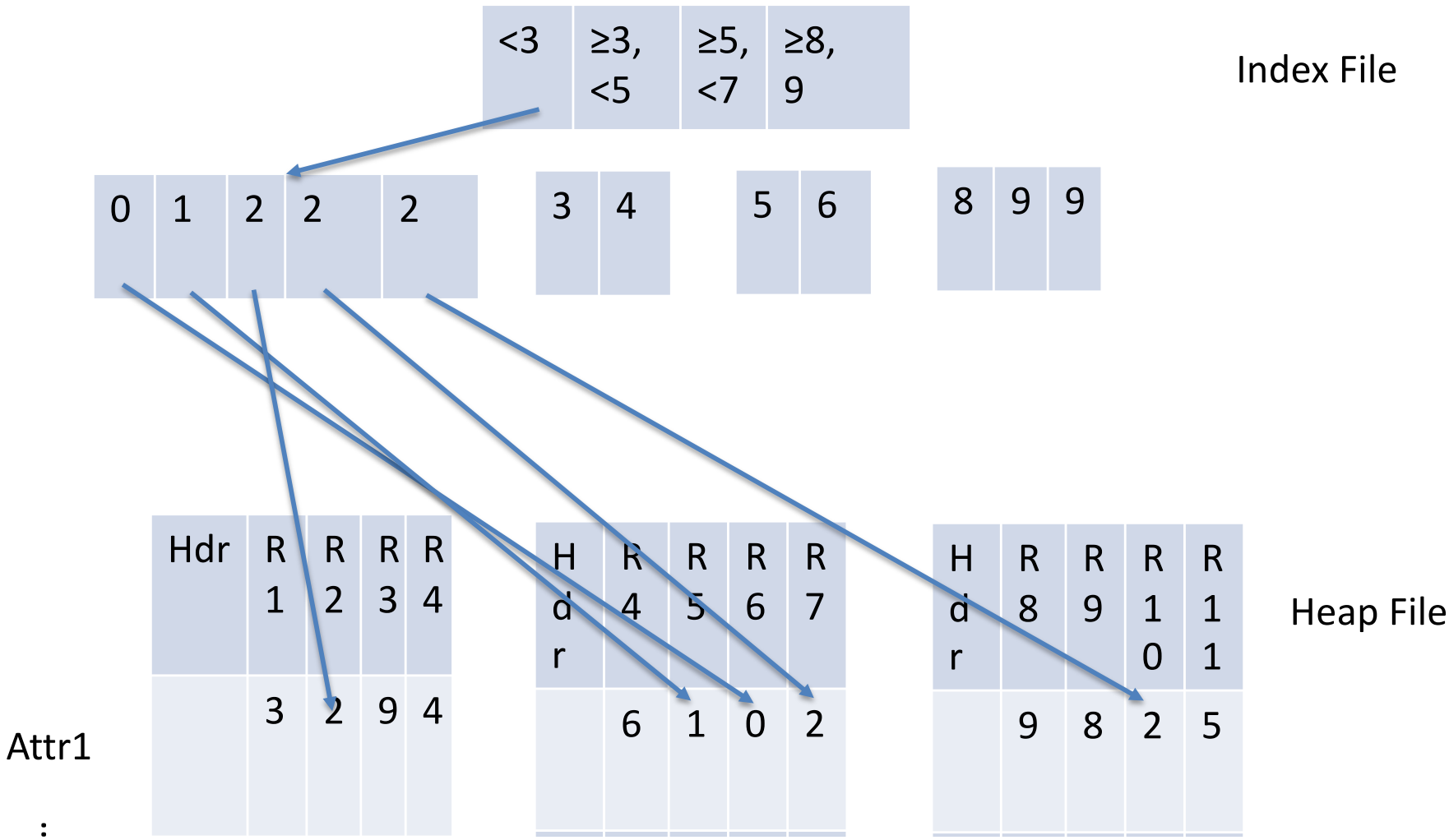
- When is it cheaper to scan than do random lookups via index?

$100 \times 10^{-6} * B / 100 + B/1 \times 10^{9} > 10 \times 10^{9} / 1 \times 10^{9}$
$1 \times 10^{-6} B + 1 \times 10^{-9} B > 10$
$B > 9.99 \times 10^{6}$

For scans of larger than 10 MB, cheaper to scan entire 10 GB table than to use index

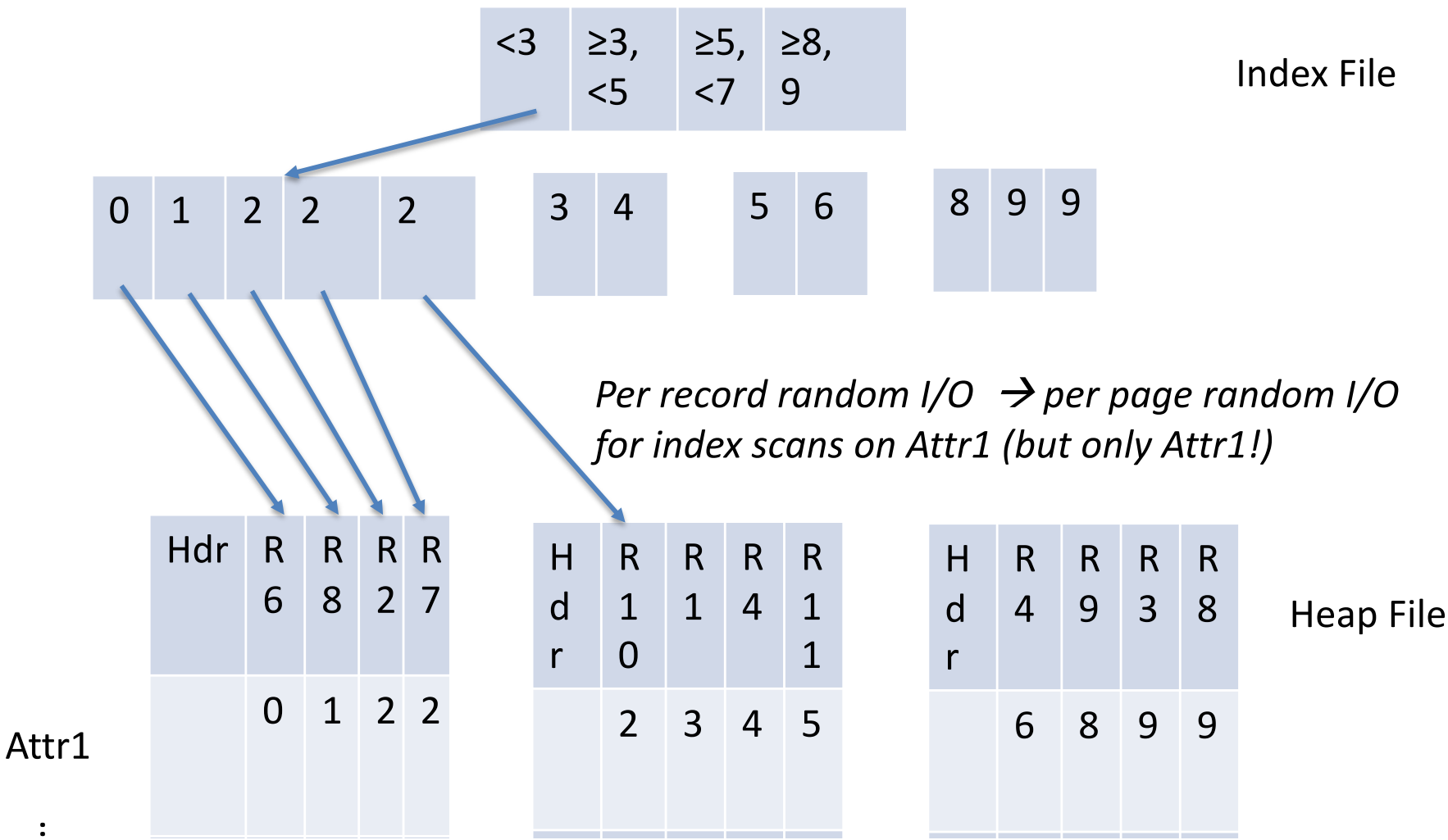
Clustered Index

- Order pages on disk in index order



Clustered Index

- Order pages on disk in index order



Benefit of Clustering

- Consider an SSD with 100 usec latency, 1 GB/sec BW
- Query accesses B bytes, R bytes per record, whole table is T bytes
- **Pages are P bytes**
- Seq scan time S = T / 1GB/sec
- Clustered index access time = 100 usec * B/P~~R~~ + B / 1GB/sec
- Suppose R is 100 bytes, T is 10 GB, **P is 1 MB**

- When is it cheaper to scan than do random lookups via clustered index?

$100\text{x}10^{-6} * B / \mathbf{1\text{x}10^{6}} + B/1\text{x}10^{9} > 10\text{x}10^{9} / 1\text{x}10^{9}$

$1\text{x}10^{-12}B + 1\text{x}10^{-9}B > 10$

$B > 9.99\text{x}10^{9}$

For scans of larger than 9.9 GB, cheaper to scan entire 10 GB table than to use **clustered** index

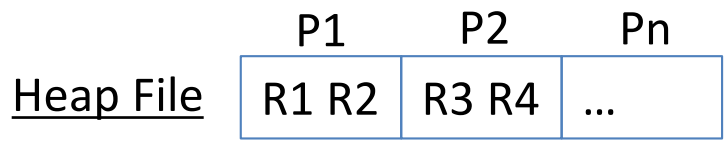
Rest of Lecture

- Details of access methods
- Heap files (already seen)
- Hash indexes (this lecture)
- Trees (B+/R) (next lecture)

Access Method Costs

	Heap File	Hash File	B+Tree
Insert	O(1)		
Delete	O(P)		
Scan	O(P) *sequential*		
Lookup	O(P)		

n : number of tuples
P : number of pages in file
B : branching factor of B-Tree
R : number of pages in scanned range

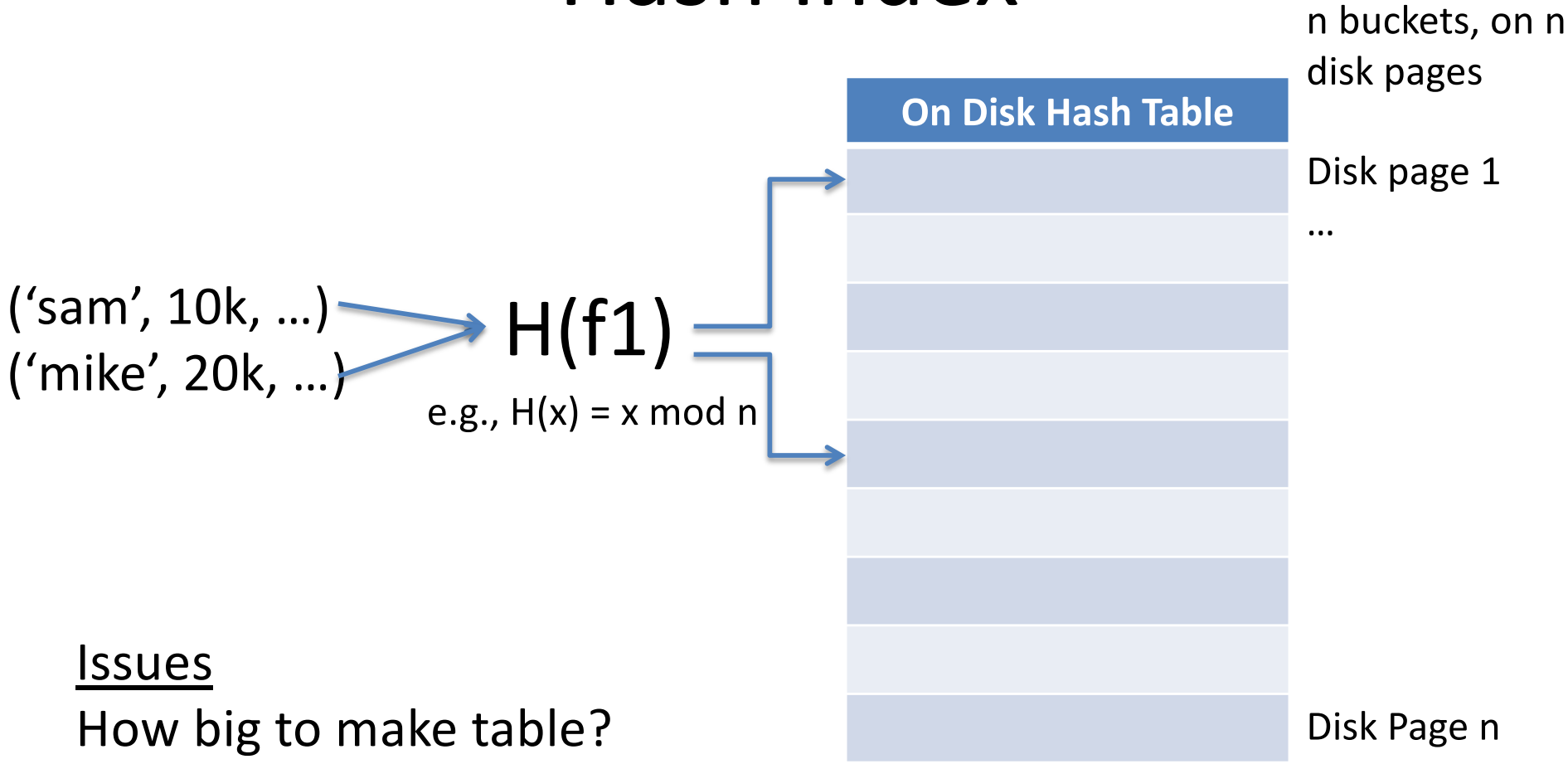


Sequentially stored pages, no seeks between records or pages

Hash Indexing Idea

- Store a hash table with pointers to records in heap file
- Hash table keyed on a particular attribute
  - Composite keys also possible
- Supports O(1) equality lookup of records
  - E.g., employees named “sam”

Hash Index



<u>Issues</u>

How big to make table?

If we get it wrong, **either**

waste space, **or**

end up with long overflow chains, **or**

have to rehash

Extensible Hashing

- Create a family of hash tables parameterized by k

$$H_k(x) = H(x) \bmod 2^k$$

- Start with k = 1 (2 hash buckets)
- Use a directory structure to keep track of which bucket (page) each hash value maps to
- When a bucket overflows, increment k (if needed), create a new bucket, rehash keys in overflowing bucket, and update directory

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

Page Number	Page Contents		
0			
1			

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

Page Number	Page Contents		
0	0		
1			

0 mod 2 = 0

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

0 mod 2 = 0

Page Number	Page Contents		
0	0	0	
1			

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

2 mod 2 = 0

Page Number	Page Contents		
0	0	0	2
1			

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

3 mod 2 = 1

Page Number	Page Contents		
0	0	0	2
1	3		

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory
k=1

$H_k(x)$	Page
0	0
1	1

Hash Table

2 mod 2 = 0

Page Number	Page Contents			
0	0	0	2	- FULL!
1	3			

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory

k=~~1~~ 2

$H_k(x)$	Page
0	0
1	1
2	
3	

Hash Table

Page Number	Page Contents		
0	0	0	2
1	3		

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory

k=~~1~~ 2

$H_k(x)$	Page
0	0
1	1
2	2
3	

Allocate new page!

Hash Table

Page Number	Page Contents		
0	0	0	2
1	3		
2			

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory

k=~~1~~ 2

$H_k(x)$	Page
0	0
1	1
2	2
3	1

Only allocate 1 new page!

Hash Table

Page Number	Page Contents			
0	0	0	2	**Rehash**
1	3			
2				

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory

k=~~1~~ 2

$H_k(x)$	Page
0	0
1	1
2	2
3	1

Hash Table

2 mod 4 = 2

Page Number	Page Contents		
0	0	0	
1	3		
2	2		

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

Example

Directory

k=~~1~~ 2

$H_k(x)$	Page
0	0
1	1
2	2
3	1

Hash Table

2 mod 4 = 2

Page Number	Page Contents		
0	0	0	
1	3		
2	2	2	

$H_k(x)$ = x mod 2^k

Insert records with keys 0, 0, 2, 3, 2

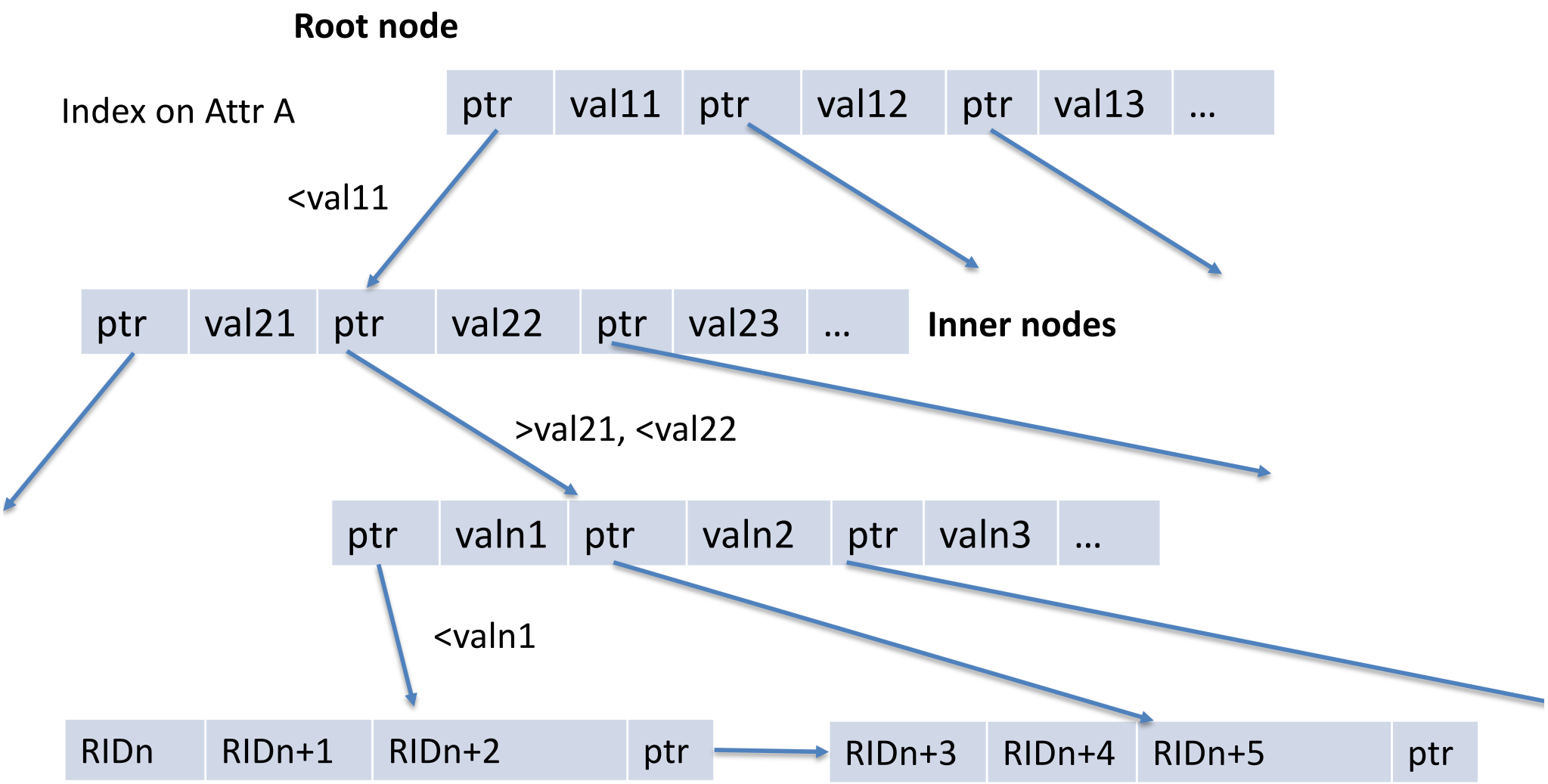
Extra bookkeeping needed to keep track of fact that pages 0/2 have split and page 1 hasn't

Access Method Costs

	Heap File	Hash File	B+Tree
Insert	O(1)	O(1)	
Delete	O(P)	O(1)	
Scan	O(P) *sequential*	-	
Lookup	O(P)	O(1)	

n : number of tuples
P : number of pages in file
B : branching factor of B-Tree
R : number of pages in range

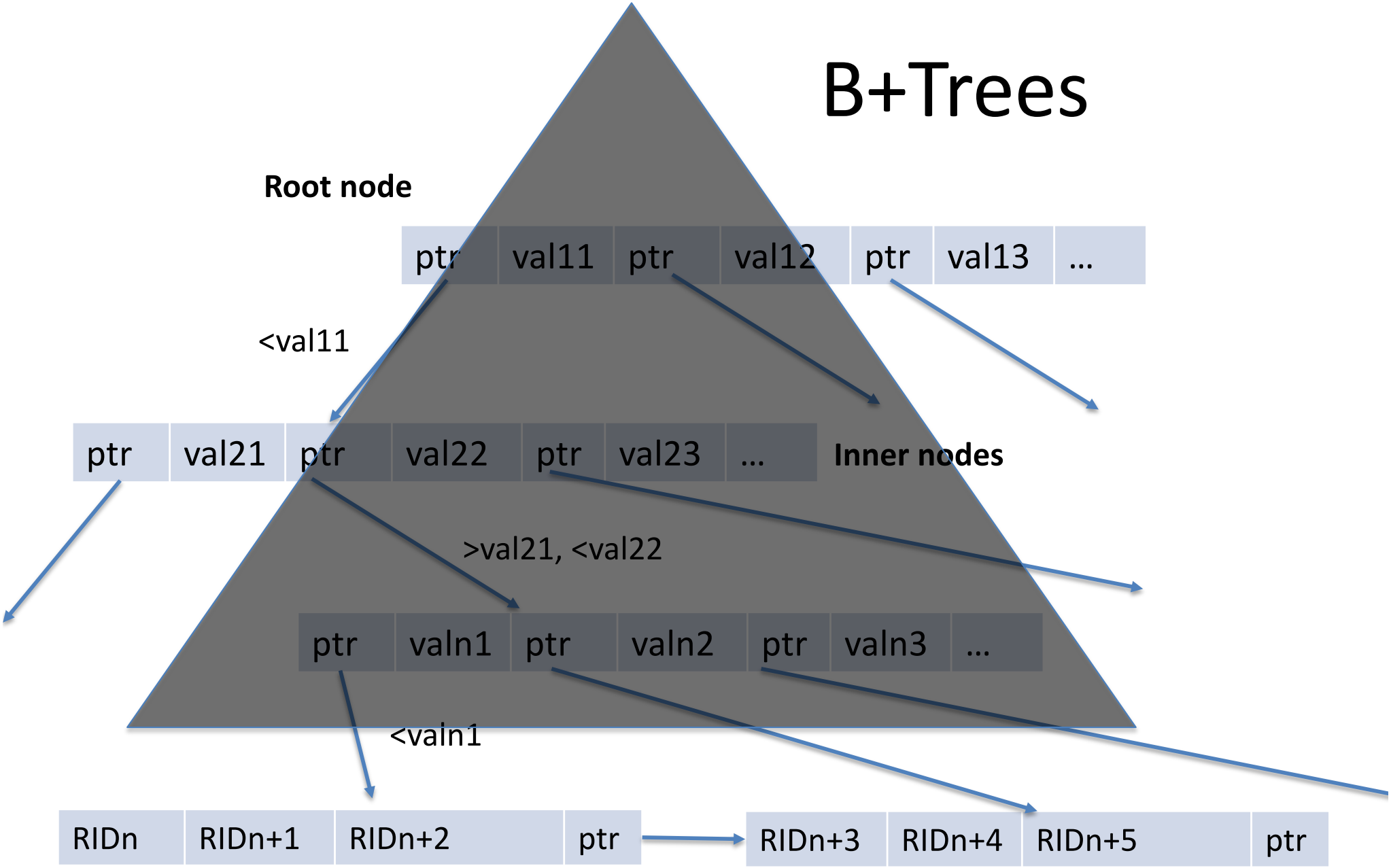
B+Trees



RID: Record ID → a reference (pointer) to a record in heap file

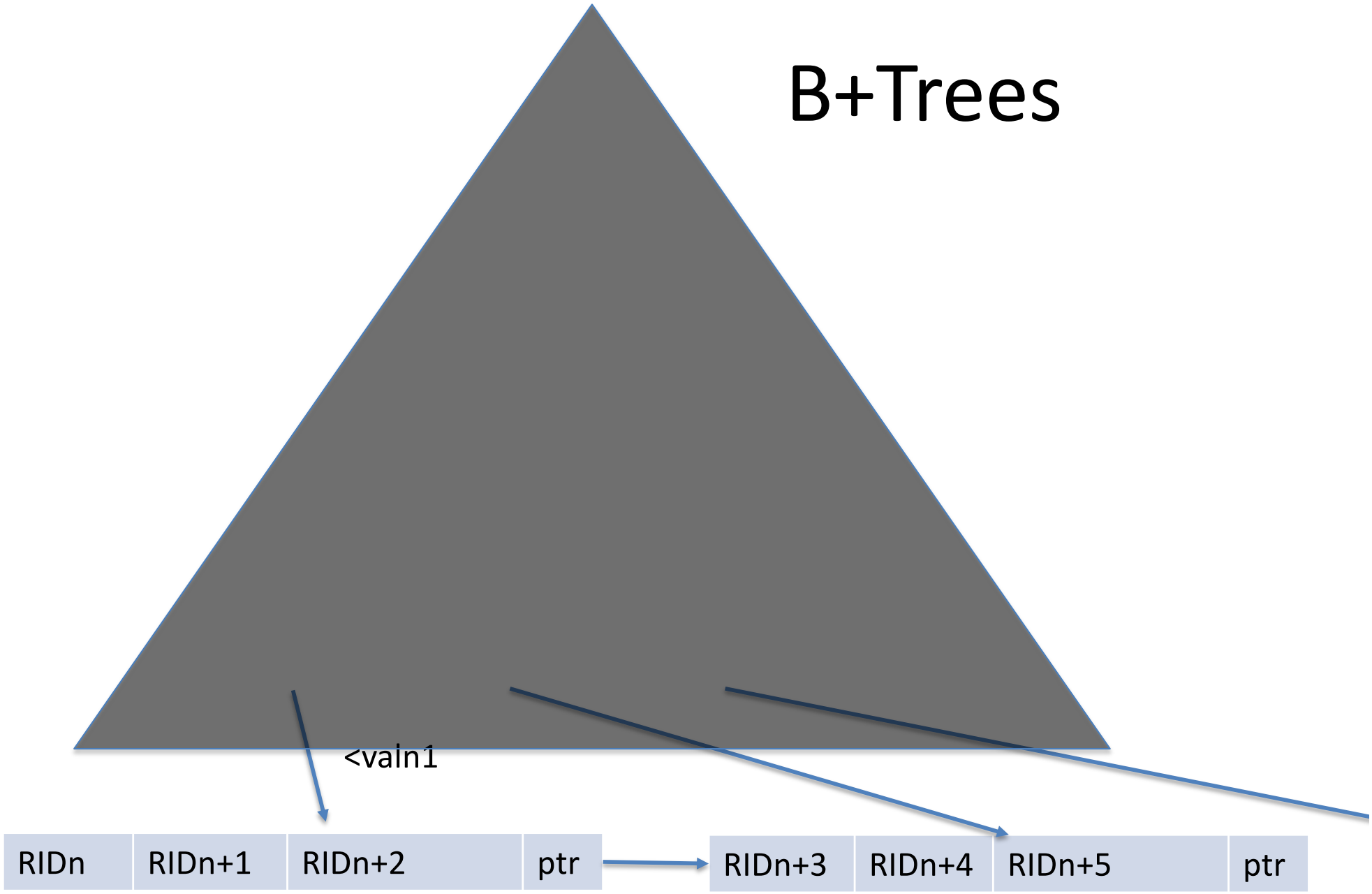
Leaf nodes; records in Attr A order, w/ link pointers

B+Trees



Leaf nodes; records in Attr A order, w/ link pointers

B+Trees



Leaf nodes; records in Attr A order, w/ link pointers

Properties of B+Trees

- Branching factor = B
- Log_B(tuples) levels
- Logarithmic insert/delete/lookup performance
- Support for range scans

- Link pointers
- No data in internal pages
- Balanced (see text “rotation”) algorithms to rebalance on insert/delete
- Fill factor: All nodes except root kept at least 50% full (merge when falls below)
- Clustered / unclustered

Indexes Recap

	Heap File	B+Tree	Hash File
Insert	O(1)	O($\log_B n$)	O(1)
Delete	O(P)	O($\log_B n$)	O(1)
Scan	O(P)	O($\log_B n + R$)	-- / O(P)
Lookup	O(P)	O($\log_B n$)	O(1)

n : number of tuples
P : number of pages in file
B : branching factor of B-Tree
R : number of pages in range

B+Trees are Inappropriate For Multi-dimensional Data

- Consider points of the form (x,y) that I want to index
- Suppose I store tuples with key (x,y) in a B+Tree
- Problem: can’t look up y’s in a particular range without also reading x’s

- Two multidimension indexes: R-Tree & QuadTree

Example Index with Key = X, Y

Index sorts data on X, then Y

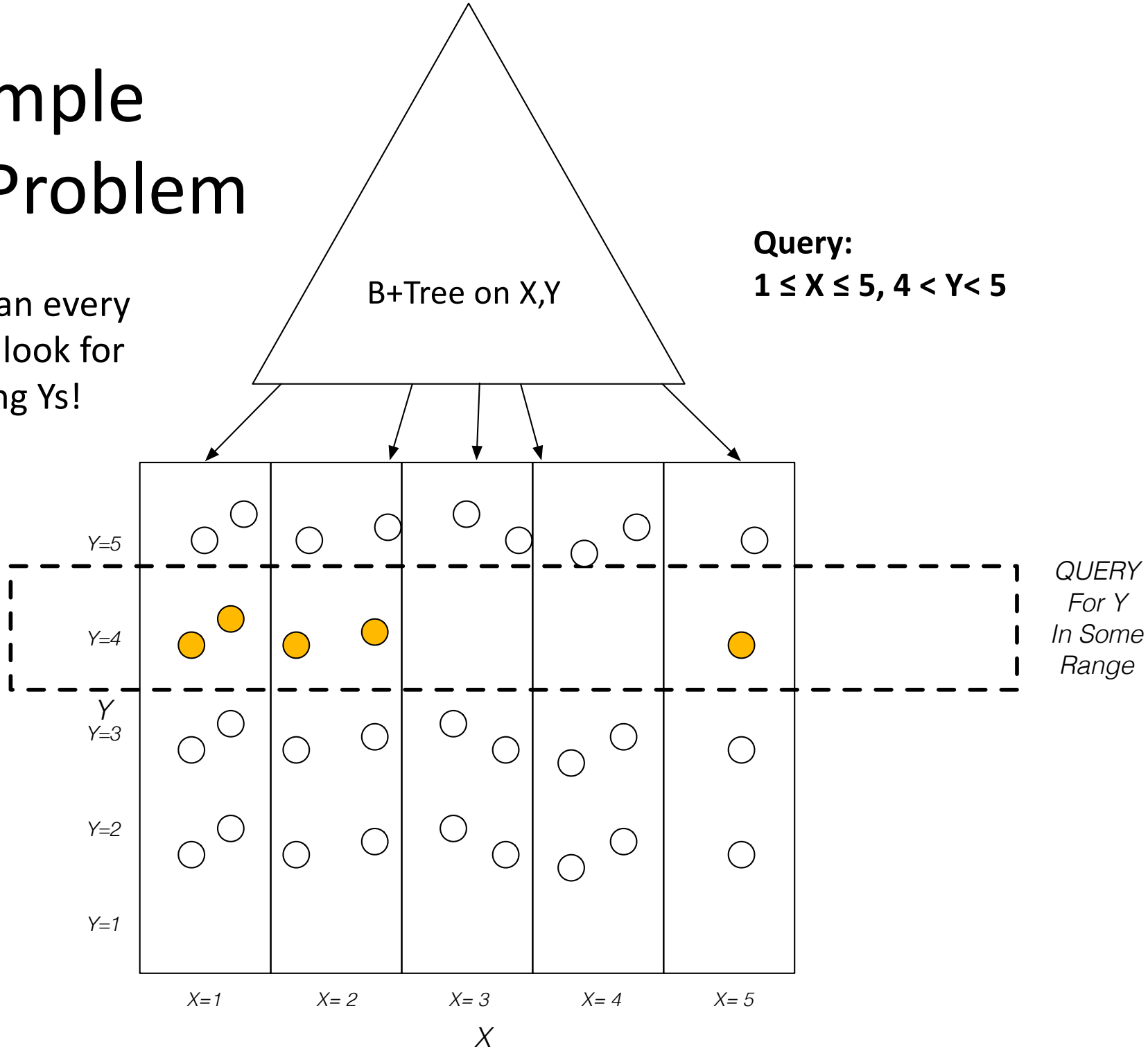
X	Y
1	2
1	3
1	5
3	12
4	3
4	9
4	11
4	15
5	1
7	1
9	4
9	6
9	7
11	2

Supports efficient range lookups on X
Allows further filtering on Y, but may be inefficient

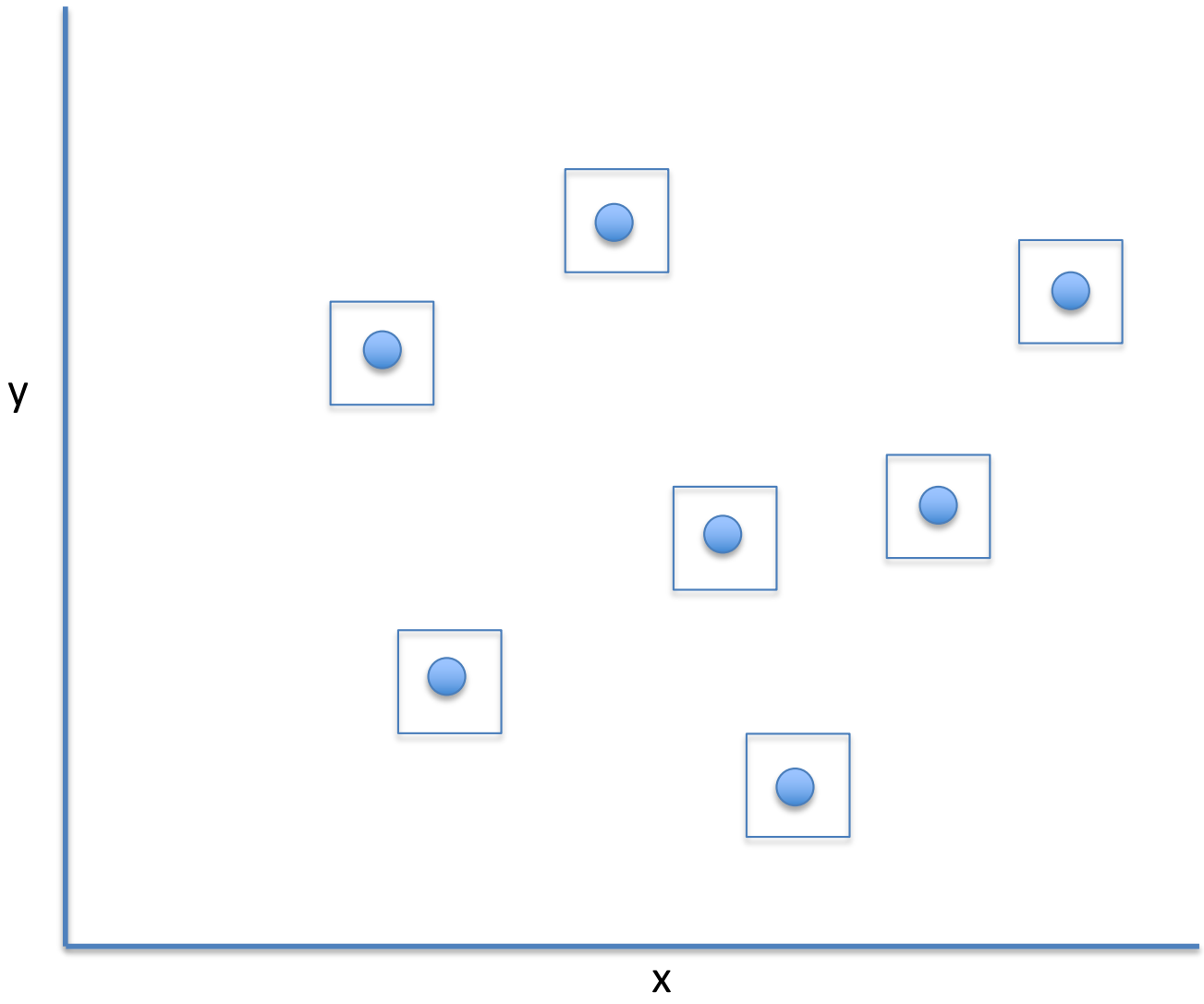
Doesn't support lookups on Y

Example of the Problem

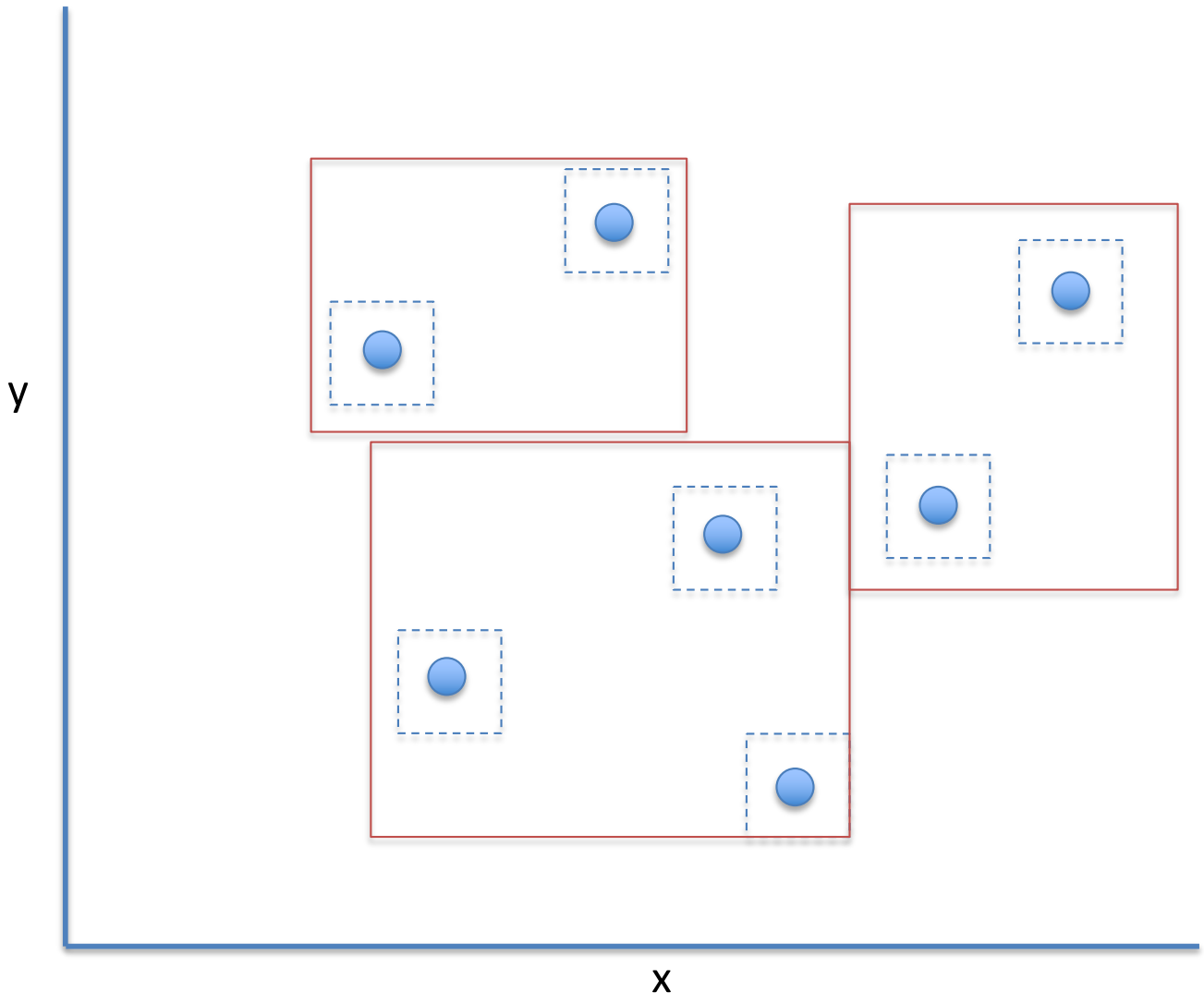
Have to scan every X value to look for matching Ys!



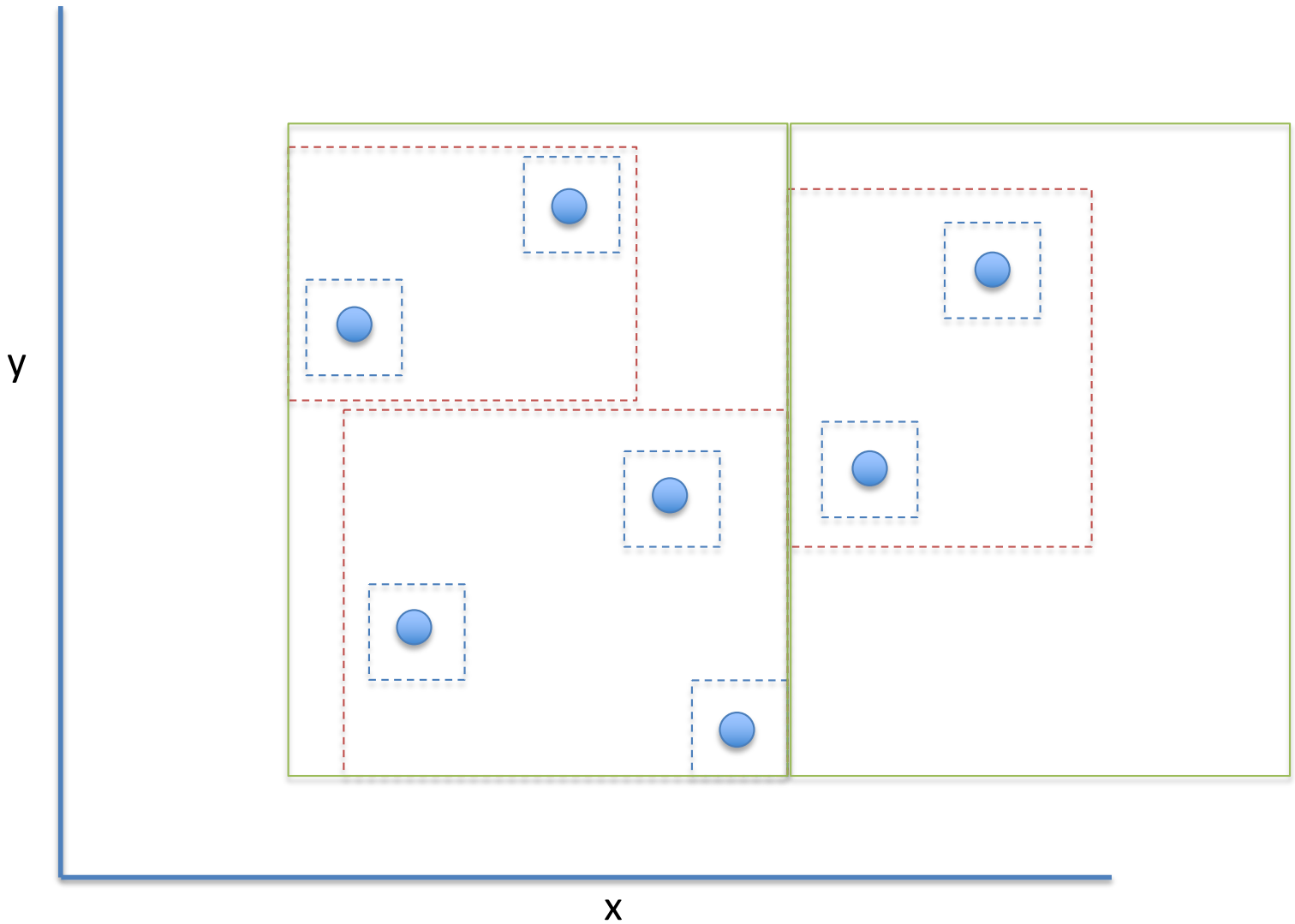
R-Trees / Spatial Indexes

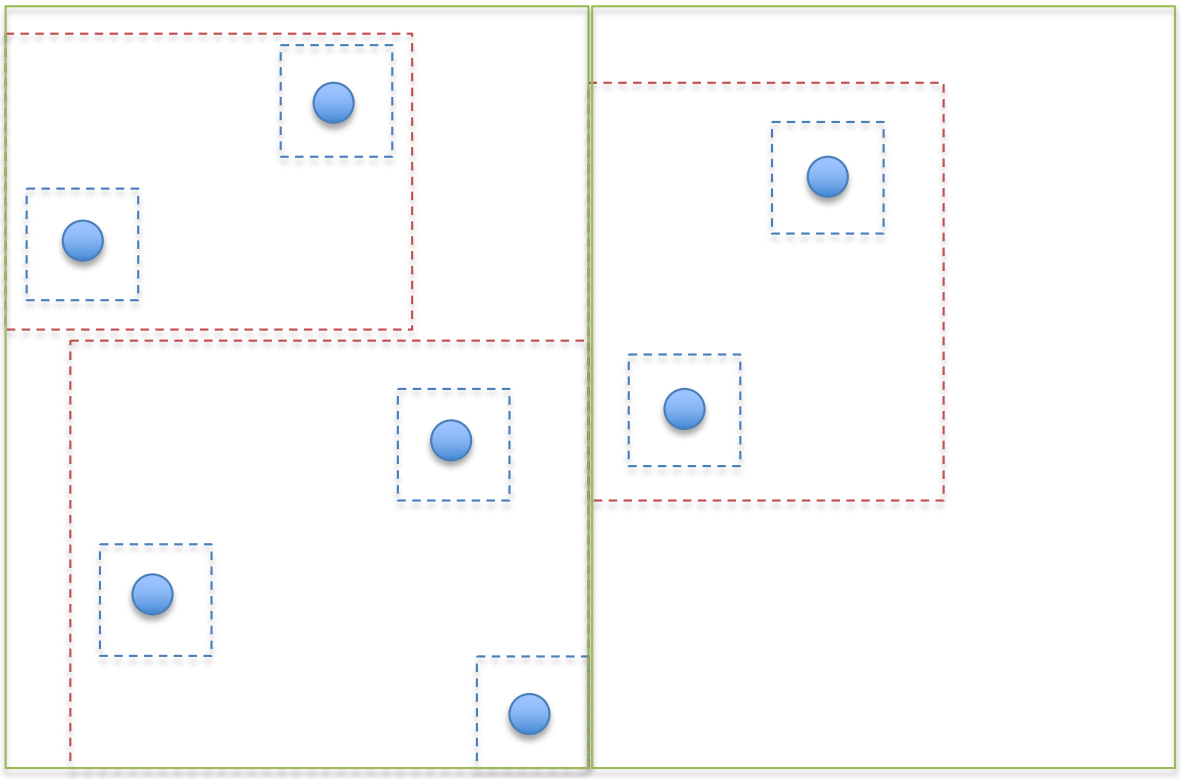


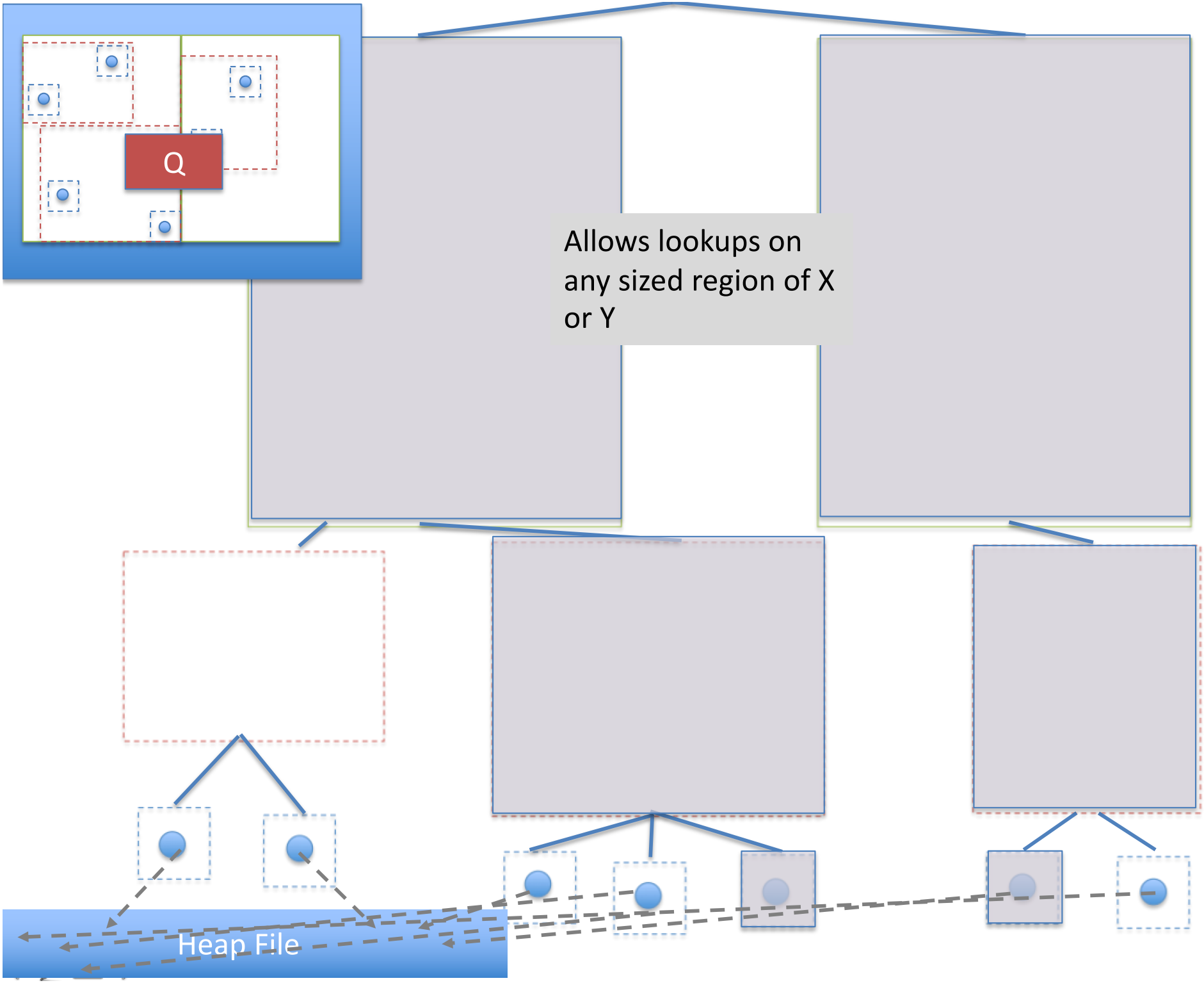
R-Trees / Spatial Indexes



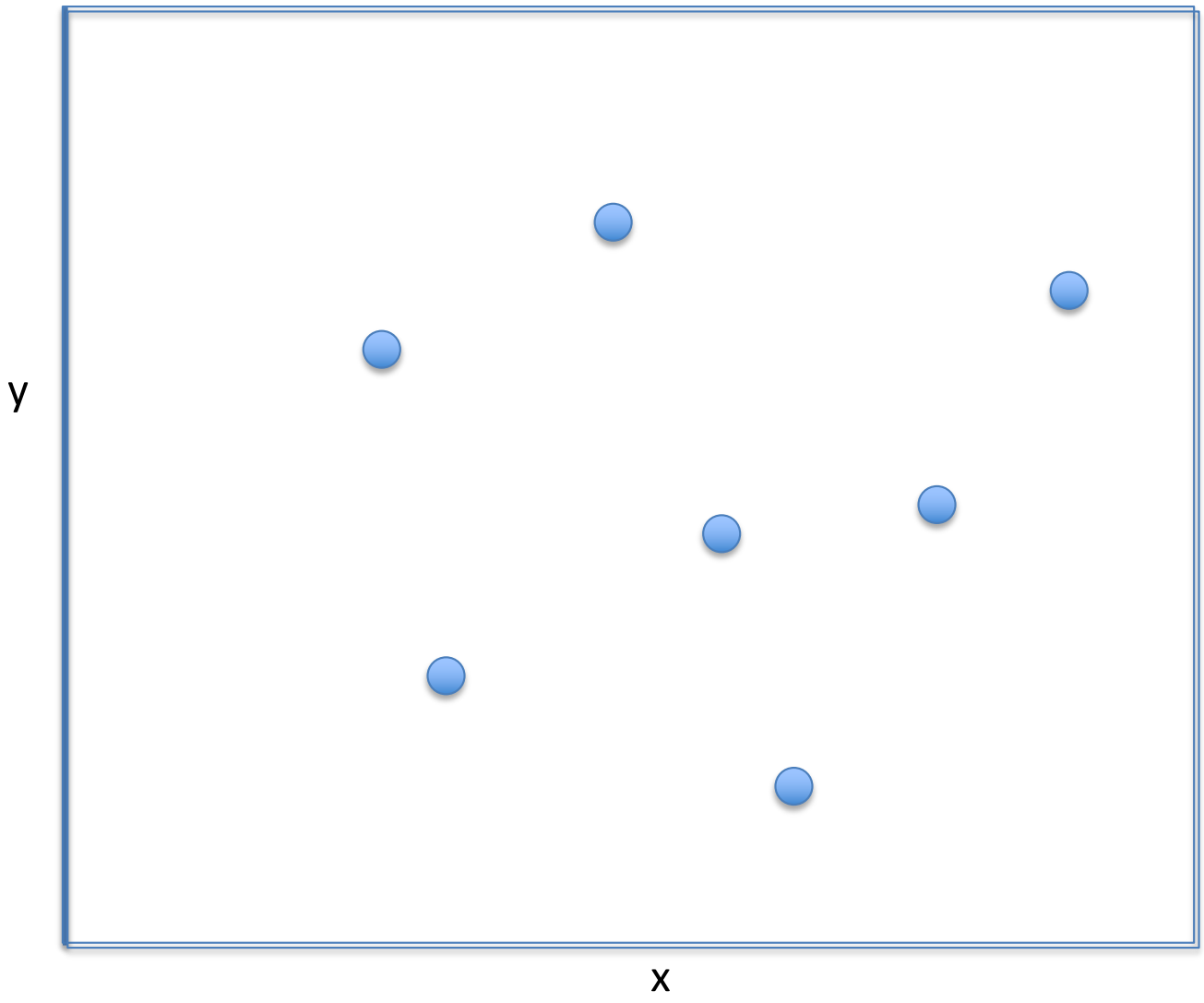
R-Trees / Spatial Indexes



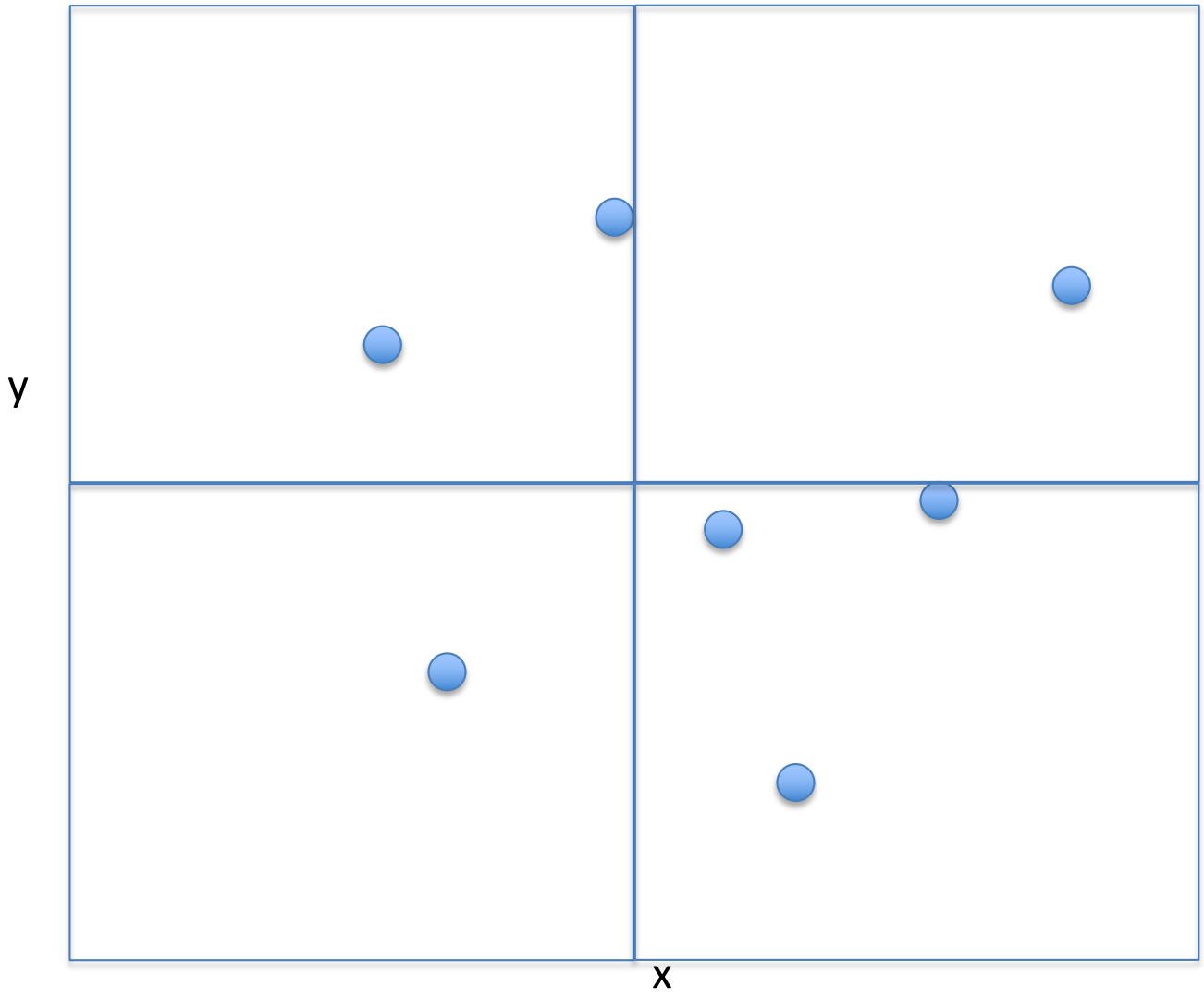



Q
Allows lookups on any sized region of X or Y
Heap File

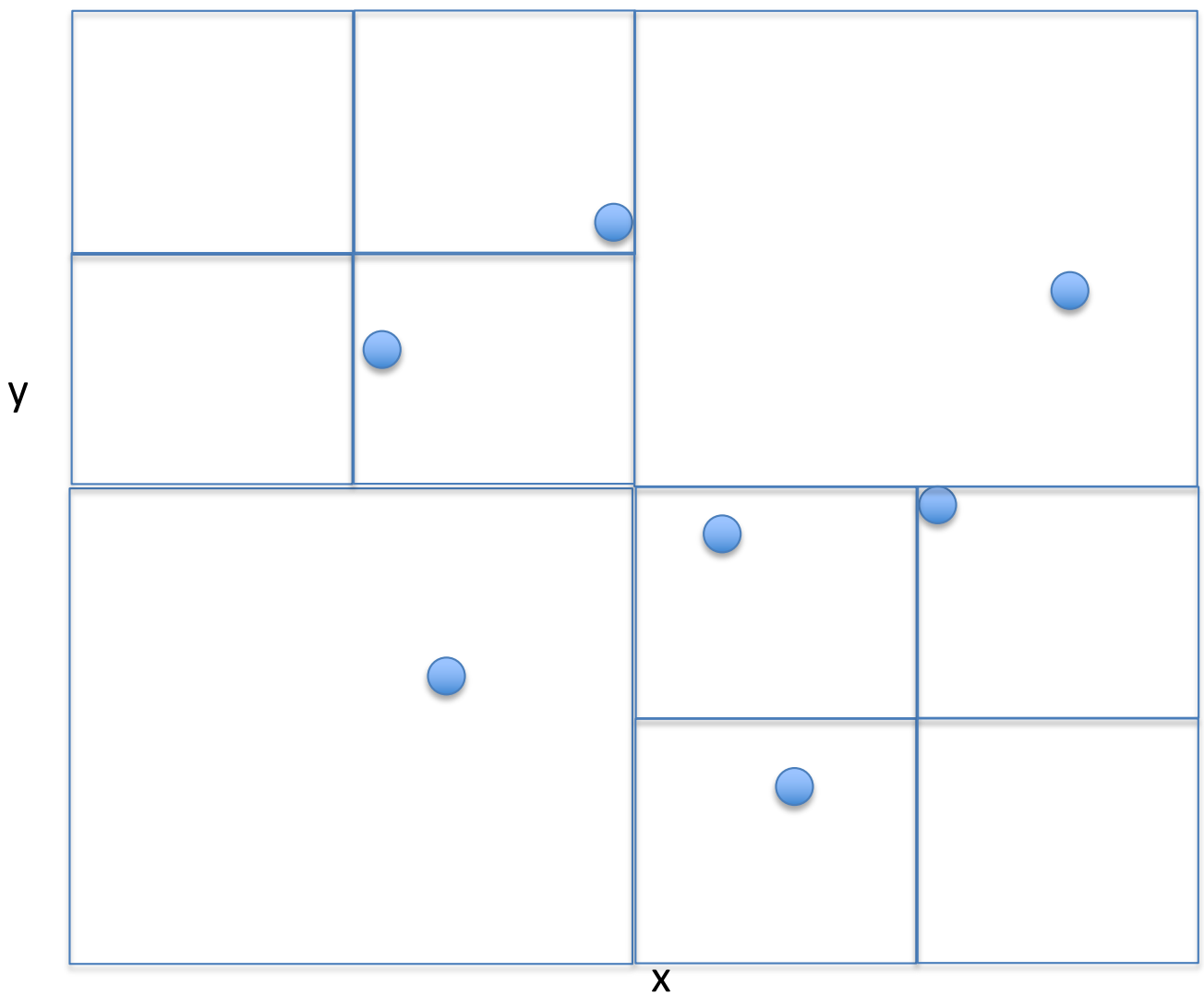
Quad-Tree



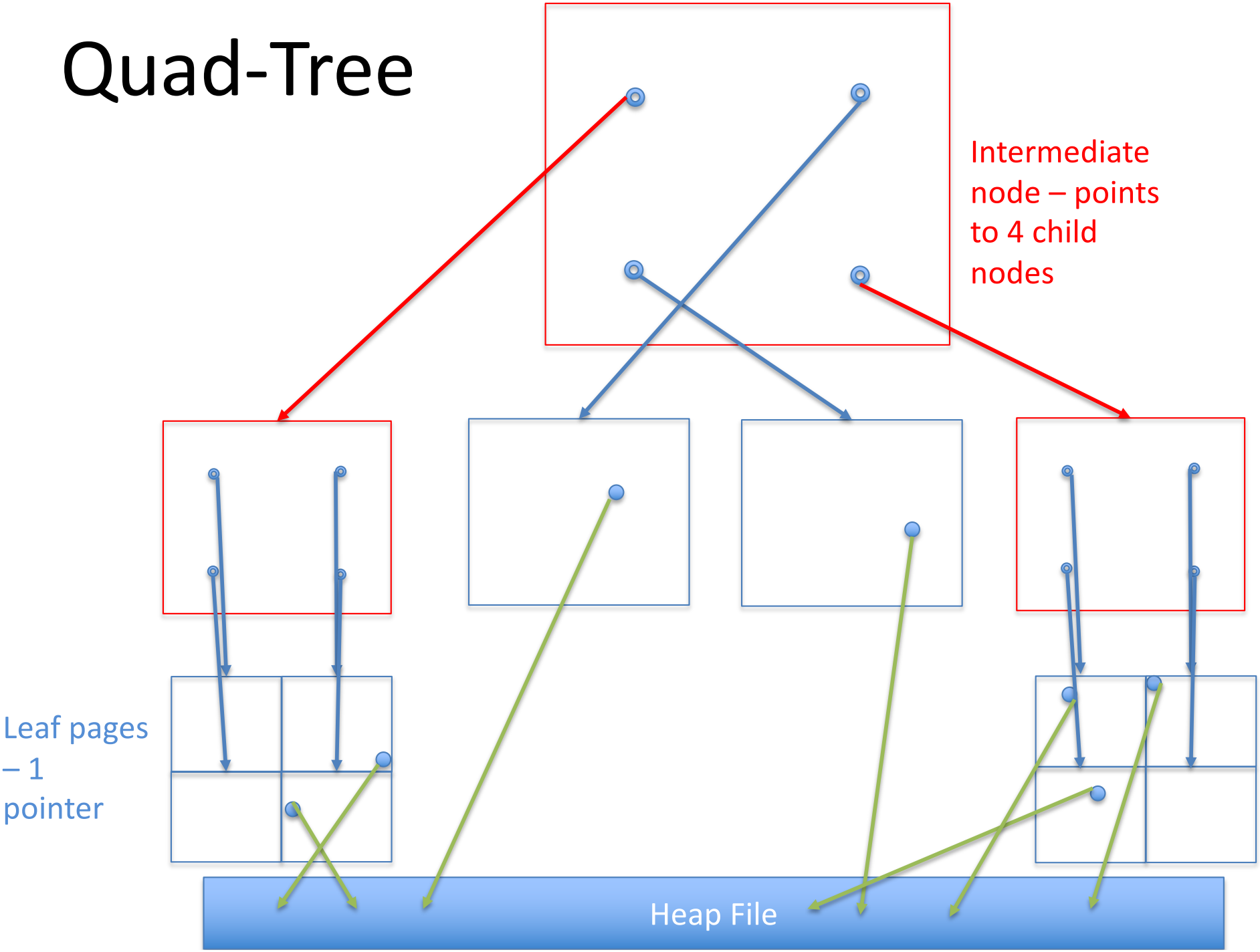
Quad-Tree



Quad-Tree



Quad-Tree

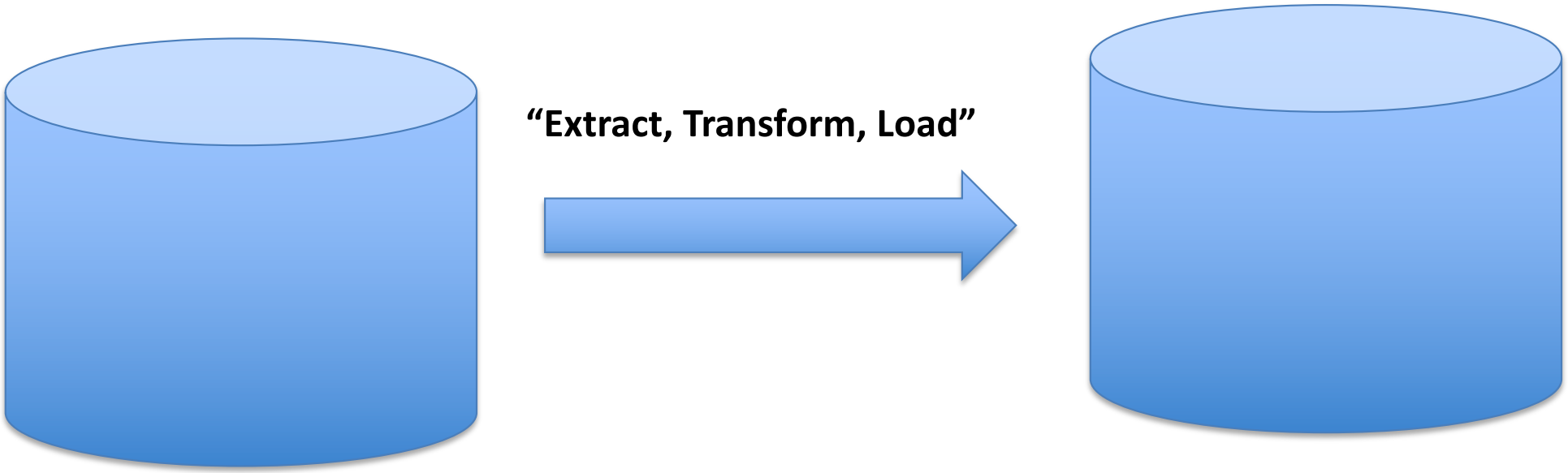

Intermediate node – points to 4 child nodes
Leaf pages – 1 pointer
Heap File

Study Break

- What indexes would you create for the following queries (assuming each query is the only query the database runs)

```
SELECT MAX(sal) FROM emp
   B+Tree on emp.sal
SELECT sal FROM emp WHERE id = 1
   Hash index on emp.id
SELECT name FROM emp WHERE sal > 100k
   B+Tree on emp.sal (maybe)
SELECT name FROM emp WHERE sal > 100k AND dept = 2
   B+tree on emp.sal (maybe), Hash on dept.dno (maybe)
```

Typical Database Setup



Transactional database
Lots of writes/updates
Reads of individual records

Analytics / Reporting Database
"Warehouse"
Lots of reads of many records
Bulk updates
Typical query touches a few columns