6.5830 Lecture 6

Buffer Pools, Indexing, and Access Methods

Florent Willems, “The Accountant”

September 23, 2024

Administrivia

- **We will have a change to the syllabus!**
- Quiz 1: *No longer on Oct 7.* Now on Oct 9.
- Lec 10: *No longer on Oct 9.* Now on Oct 7.
- Project proposals: Now due on Oct 11
- PS2: Now due on Oct 7

If these changes pose a profound problem for you, please come talk to us

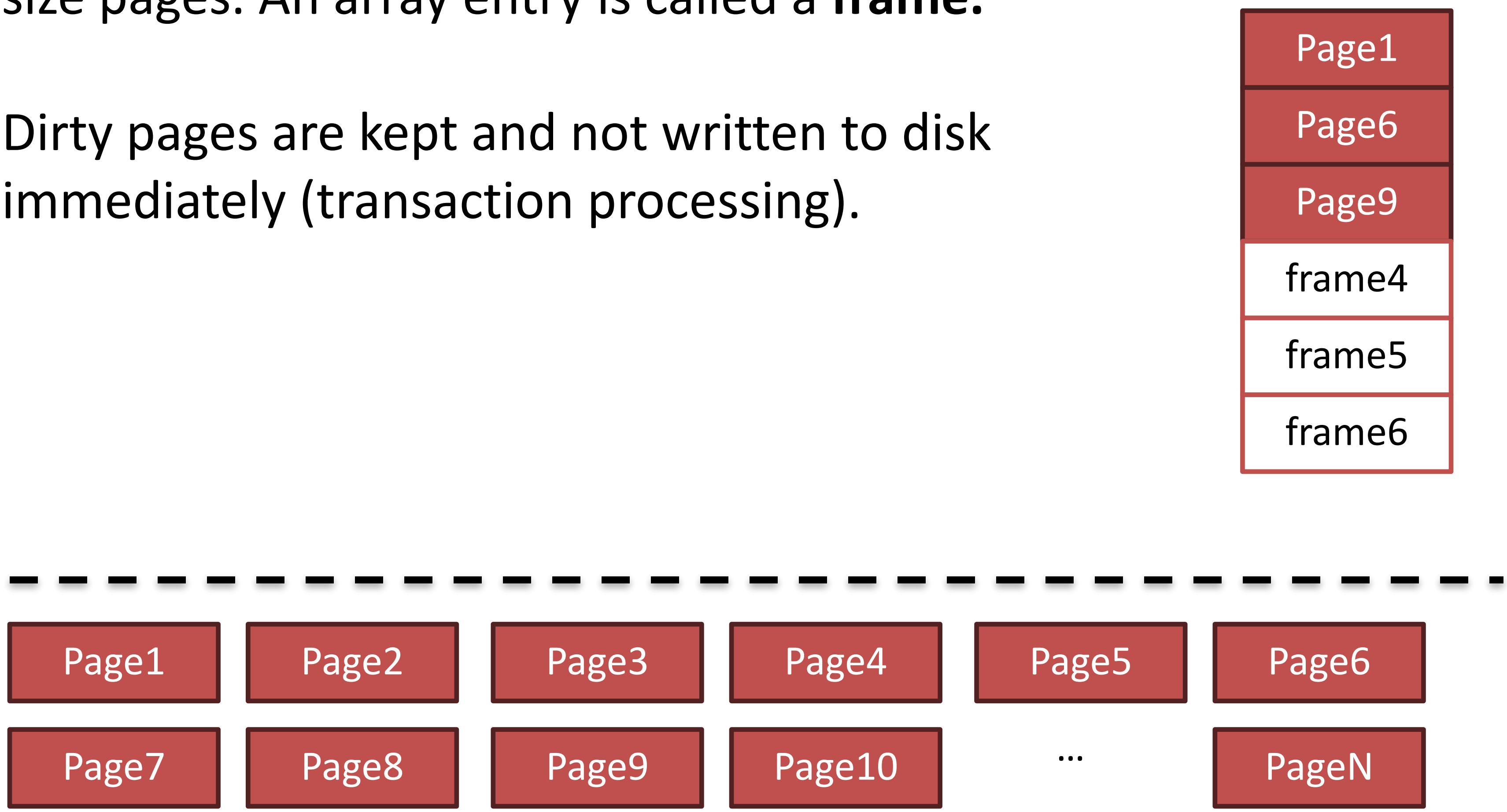
Recap: Buffer Pools

- **Buffer pool** is a cache for memory access. Caches pages of files / indices.
- When page is in buffer pool, don't need to read from disk
- Updates can also be cached
  - Discuss more w/ transactions

Buffer Pool

Memory region organized as an array of fixed size pages. An array entry is called a **frame.**

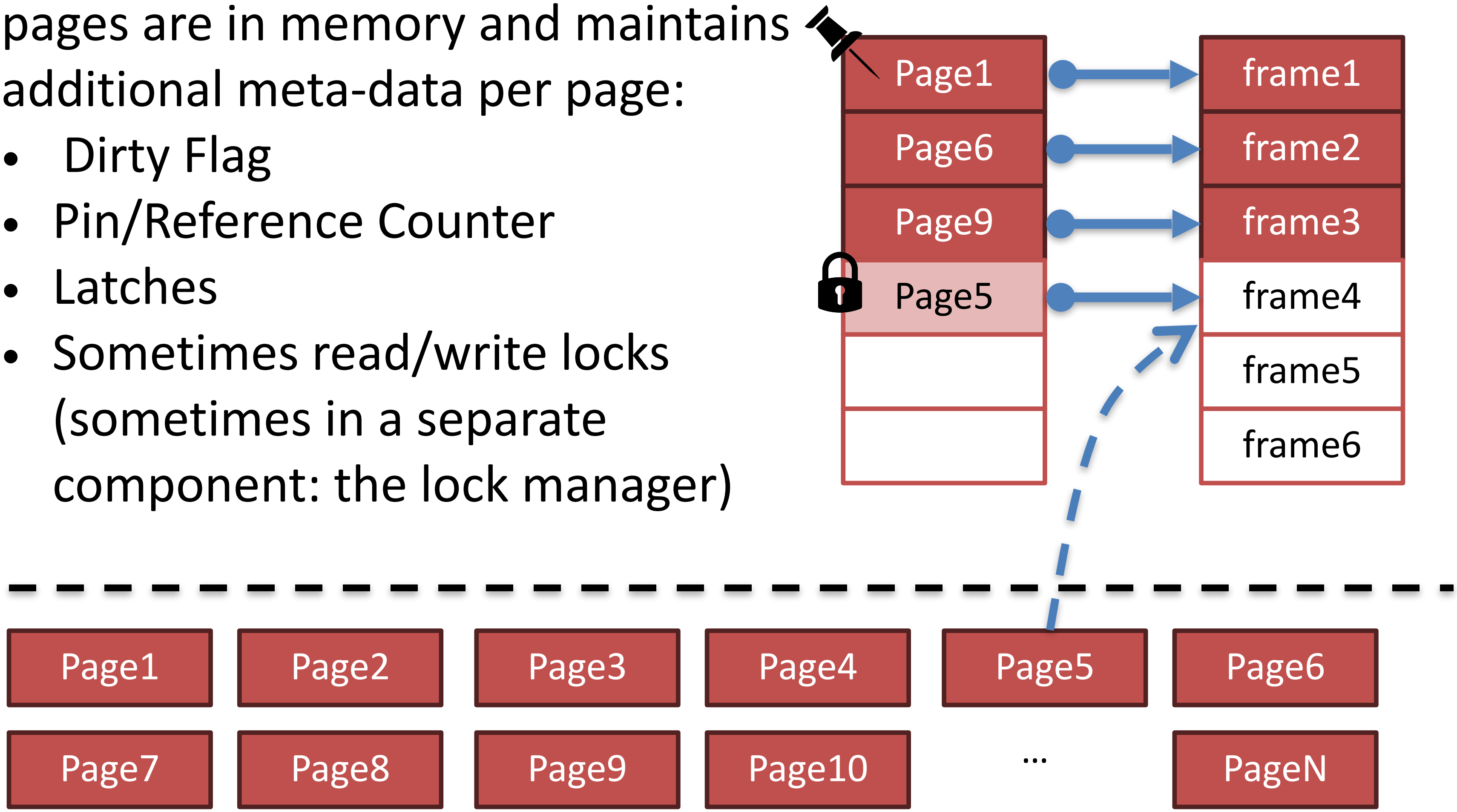
Dirty pages are kept and not written to disk immediately (transaction processing).



Buffer Pool

The **page table** keeps track of what pages are in memory and maintains additional meta-data per page:

- Dirty Flag
- Pin/Reference Counter
- Latches
- Sometimes read/write locks (sometimes in a separate component: the lock manager)



Eviction Policy

- Least Recently Used (LRU)
  - Evict oldest page accessed
  - Intuitively, makes sense because recently accessed data is likely to be accessed again

- Is LRU always optimal?

Is LRU Always Optimal?

- No! What if some relation doesn't fit into memory?

Consider: 2 pages RAM, 3 pages of a relation R -- a, b c, accessed sequentially in a loop

	Access			
RAM Page	1	2	3	4
1	a	a	c	c
2		b	b	a

LRU Always misses!

Databases do not comply with some traditional OS assumptions

Consider MRU

Consider: 2 pages RAM, 3 pages of a relation R -- a, b c, accessed sequentially in a loop

	Access							
RAM Page	1 (a)	2 (b)	3 (c)	4 (a)	5 (b)	6 (c)	7 (a)	8 (b)
1	a	a	a	**A - hit**	b	b	b	**B - hit**
2		b	c	c	c	**C – hit**	a	a

MRU hits on 1 out of 2!

Better Policies

What other policies can you think of?

Better Policies

- LRU-K: Keep the last k accesses. Estimate when the next one will happen
- Query-local-policies: Queries often know better what the access pattern is. Leverage it (e.g., Postgres maintains a small ring buffer that is private to the query)
- Priority hints: For example, set a priority hint for the top index pages rather data pages

Buffer Pool Optimization

What other optimizations can you think of?

Buffer Pool Optimizations

- Multiple Buffer Pools
- Pre-Fetching
- Scan Sharing
- Buffer Pool Bypass

Scan Sharing

- How does Scan Sharing work?
- PostgreSQL:
  `synchronize_seqscans (Boolean)`
- This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload.
- …. *This can result in unpredictable changes in the row ordering returned by queries that have no ORDER BY clause. Why?*

Let's Test Some Postgres Query Plans

```
create table dept (
	dno int primary key,
	bldg int);
```

```
insert into dept (dno, bldg)
select x.id, (random() * 10)::int
FROM generate_series(0,100000) AS x(id);
```

```
create table emp (
	eno int primary key,
	dno int references dept(dno),
	sal int,
	ename varchar);
```

```
insert into emp (eno, dno, sal, ename)
select x.id,
	(random() * 100000)::int,
	(random() * 55000)::int,
	'emp' || x.id
	from generate_series(0,10000000) AS x(id);
```

```
create table kids (
	kno int primary key,
	eno int references emp(eno),
	kname varchar);
```

```
insert into kids (kno,eno,kname)
select x.id,
	(random() * 1000000)::int,
	'kid' || x.id
	from generate_series(0,3000000) AS x(id);
```

Postgres Costs

```
explain select * from emp;
                    QUERY PLAN
--------------------------------------------------------------
 Seq Scan on emp  (cost=0.00..163696.15 rows=10000115 width=22)
(1 row)

test=# select relpages from pg_class where relname = 'emp';
 relpages
----------
   63695
(1 row)

test=# show cpu_tuple_cost;
 cpu_tuple_cost
----------------
 0.01
(1 row)
```

Cost =

cpu_tuple_cost * rows + pages =

.01 * 10000115 + 63695 = 163696.15

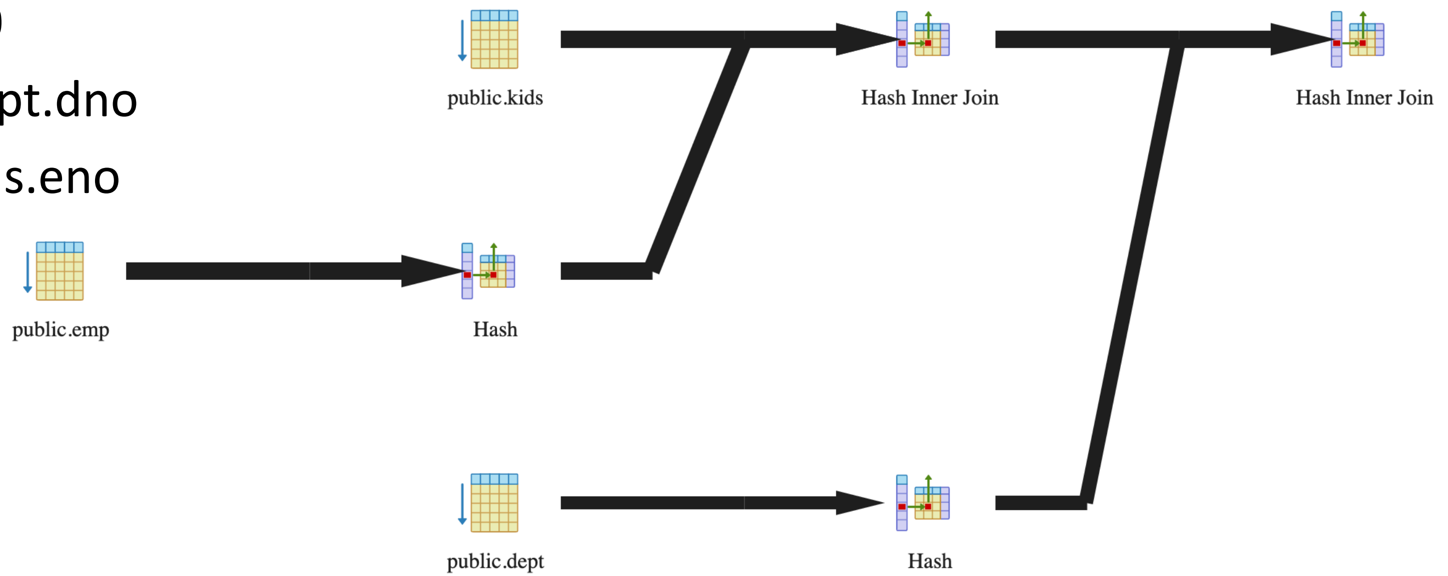
Postgres Plans

SELECT * FROM emp, dept, kids

WHERE sal > 10000

AND emp.dno = dept.dno

AND emp.eno = kids.eno




```
QUERY PLAN
-----------------------------------------------------------------------------
Hash Join  (cost=342160.30..527523.82 rows=2457233 width=48)
  Hash Cond: (emp.dno = dept.dno)
  ->  Hash Join  (cost=339076.28..479202.29 rows=2457233 width=40)
        Hash Cond: (kids.eno = emp.eno)
        ->  Seq Scan on kids  (cost=0.00..49099.01 rows=3000001 width=18)
        ->  Hash  (cost=188696.44..188696.44 rows=8190867 width=22)
              ->  Seq Scan on emp  (cost=0.00..188696.44 rows=8190867 width=22)
                    Filter: (sal > 10000)
  ->  Hash  (cost=1443.01..1443.01 rows=100001 width=8)
        ->  Seq Scan on dept  (cost=0.00..1443.01 rows=100001 width=8)
(10 rows)
```

https://clicker.mit.edu/6.5830/

- 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.

https://clicker.mit.edu/6.5830/

- 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 (4B))

(A) 0.21 seconds
(B) 0.23 seconds
(C) 0.25 seconds
(D) I don't know

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

https://clicker.mit.edu/6.5830/

- 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))
```

2. Estimate the time to join these two tables, using nested loops, assuming students fits in memory but grades does not, and students contains 10K records (grades contains 1M records).

   (A) 0.251 s
   (B) 2300.0 s
   (C) 4000.0 s
   (D) I don't know.

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 + 10ms seek= .011 s + 0.01s
- One pass over students (cached) for each grade (no additional cost beside caching)
- Time to scan grades (previous slide) = .23 s

➔ 0.23 + 0.011 + 0.01 == .251 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

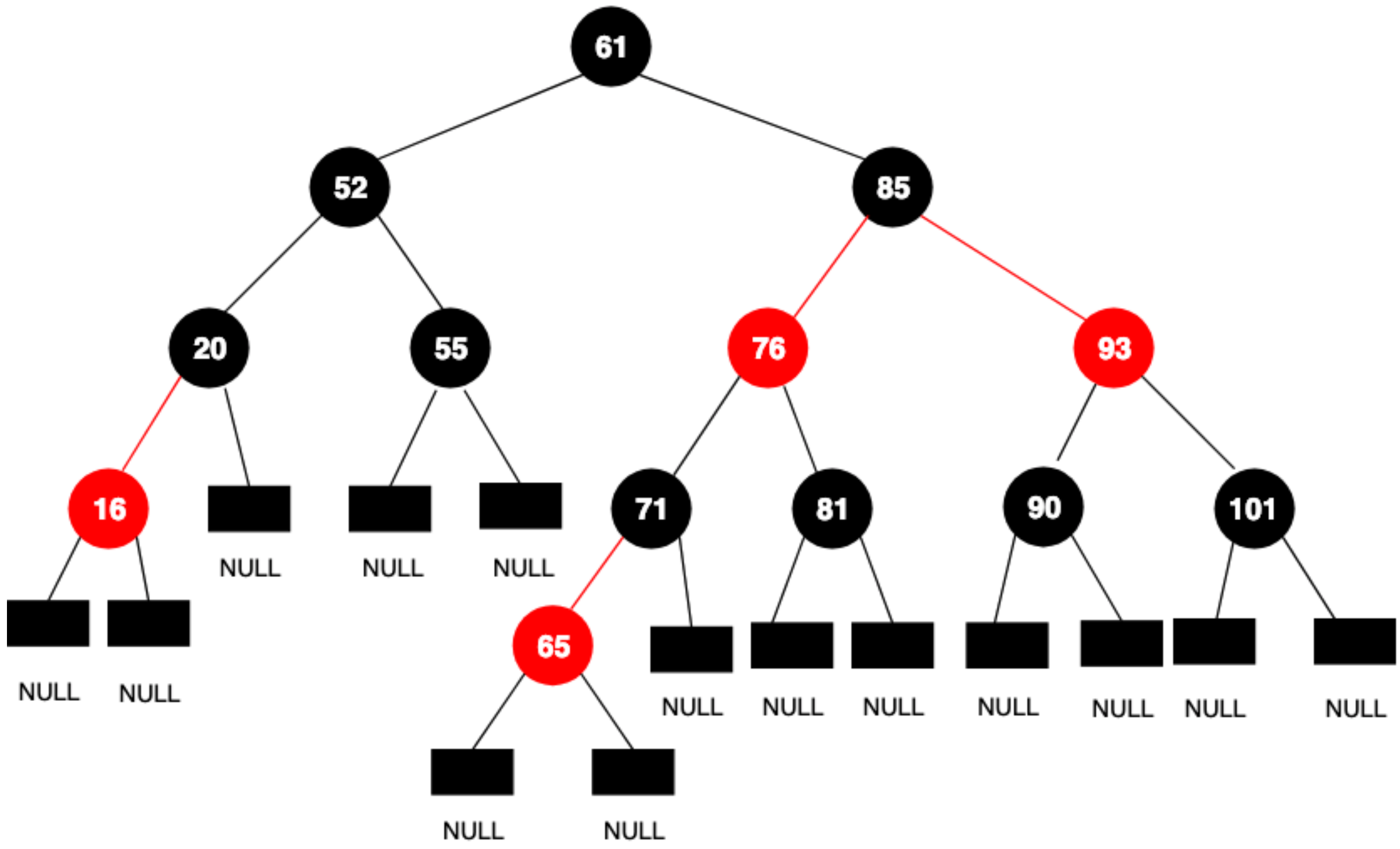
➔ 2300 seconds overall

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

Today: Access Methods

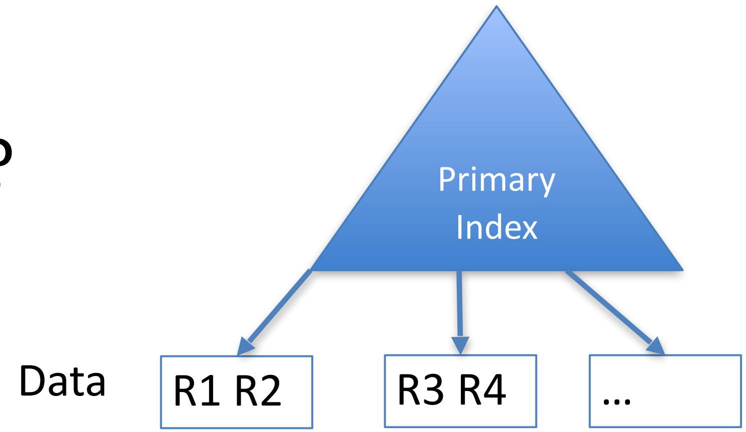
- Access method: way to access the records of the database
- 3 main types:
  - Heap file / heap scan
  - Hash index / index lookup
  - B+Tree index / index lookup / scan ← next time
- Many alternatives: e.g., R-trees ← next time
- Each has different performance tradeoffs

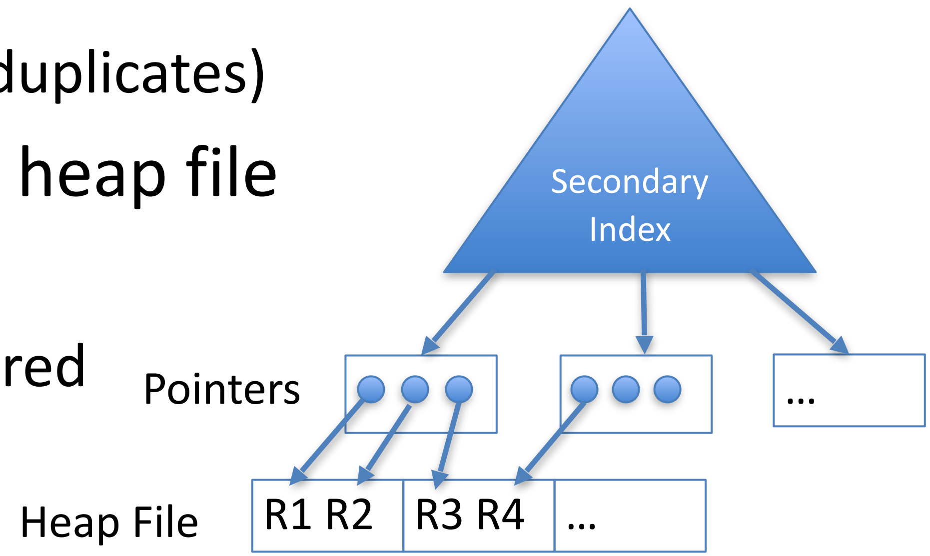
Design Considerations for Indexes



Design Considerations for Indexes

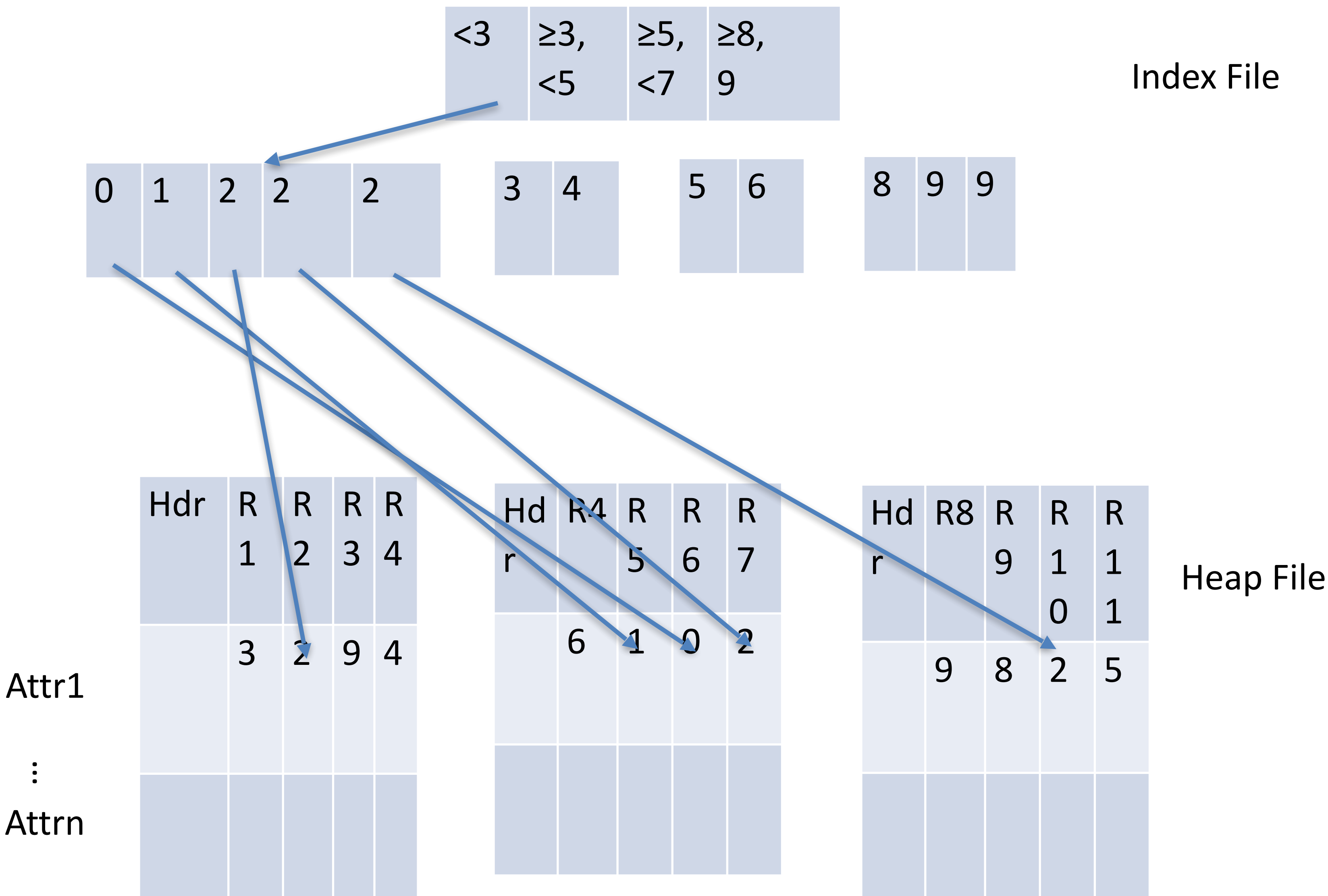
- What attributes to index?
  - Why not index everything?
- Index structure:
  - Leaves as data
    - Only one index?
    - "Primary Index" (no duplicates)
  - Leaves as pointers to heap file
    - "Secondary Index"
    - Clustered vs unclustered





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

Tree Index

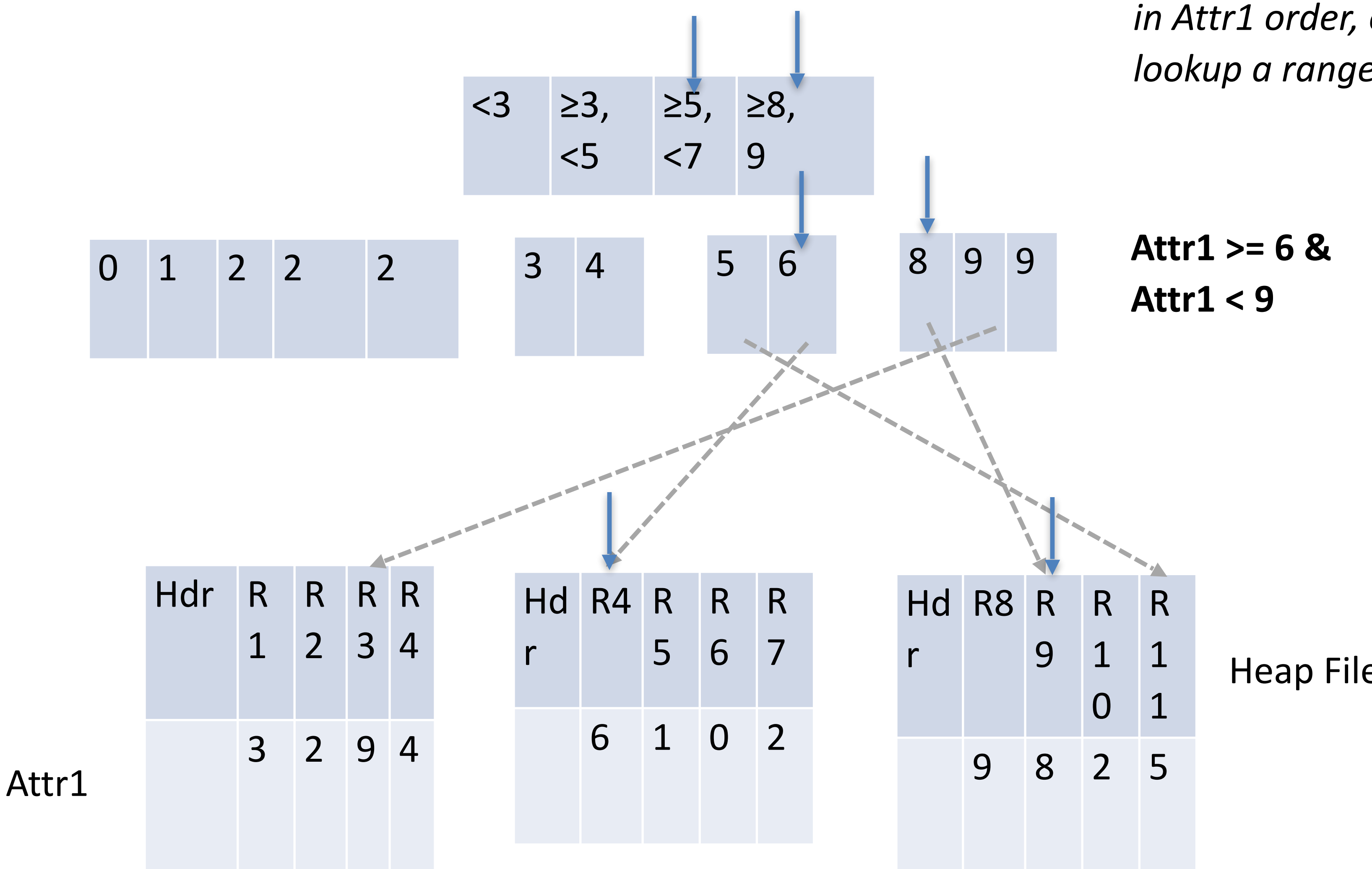


<3
≥3, <5
≥5, <7
≥8, 9
Index File
0 1 2 2 2
3 4
5 6
8 9 9
Hdr R1 R2 R3 R4
3 2 9 4
Hdr R4 R5 R6 R7
6 1 0 2
Hdr R8 R9 R10 R11
9 8 2 5
Heap File
Attr1
⋮
Attrn

Index Scan

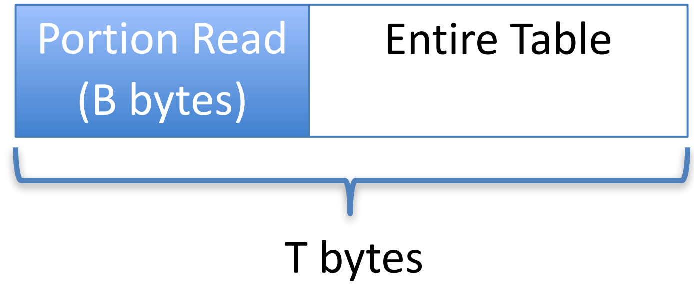
Traverse the records in Attr1 order, or lookup a range

Attr1 >= 6 & Attr1 < 9



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

Costs of Random Access
https://clicker.mit.edu/6.5830/

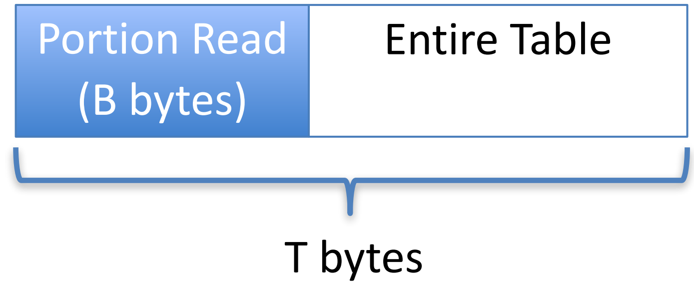


- Consider an SSD with 100 usec latency, 1 GB/sec BW
- Query accesses B bytes, R bytes per record, whole table is T bytes
- Seq scan time S = T / 1GB/sec
- Rand access via index time = 100 usec * B/R + B / 1GB/sec
- Suppose R is 100 bytes, T is 10 GB

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

(a) Scans larger than ≈1MB (0.01%)
(b) Scans larger than ≈10MB (0.1%)
(c) Scans larger than ≈100MB (1%)
(d) Scans larger than ≈1GB (10%)

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?

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

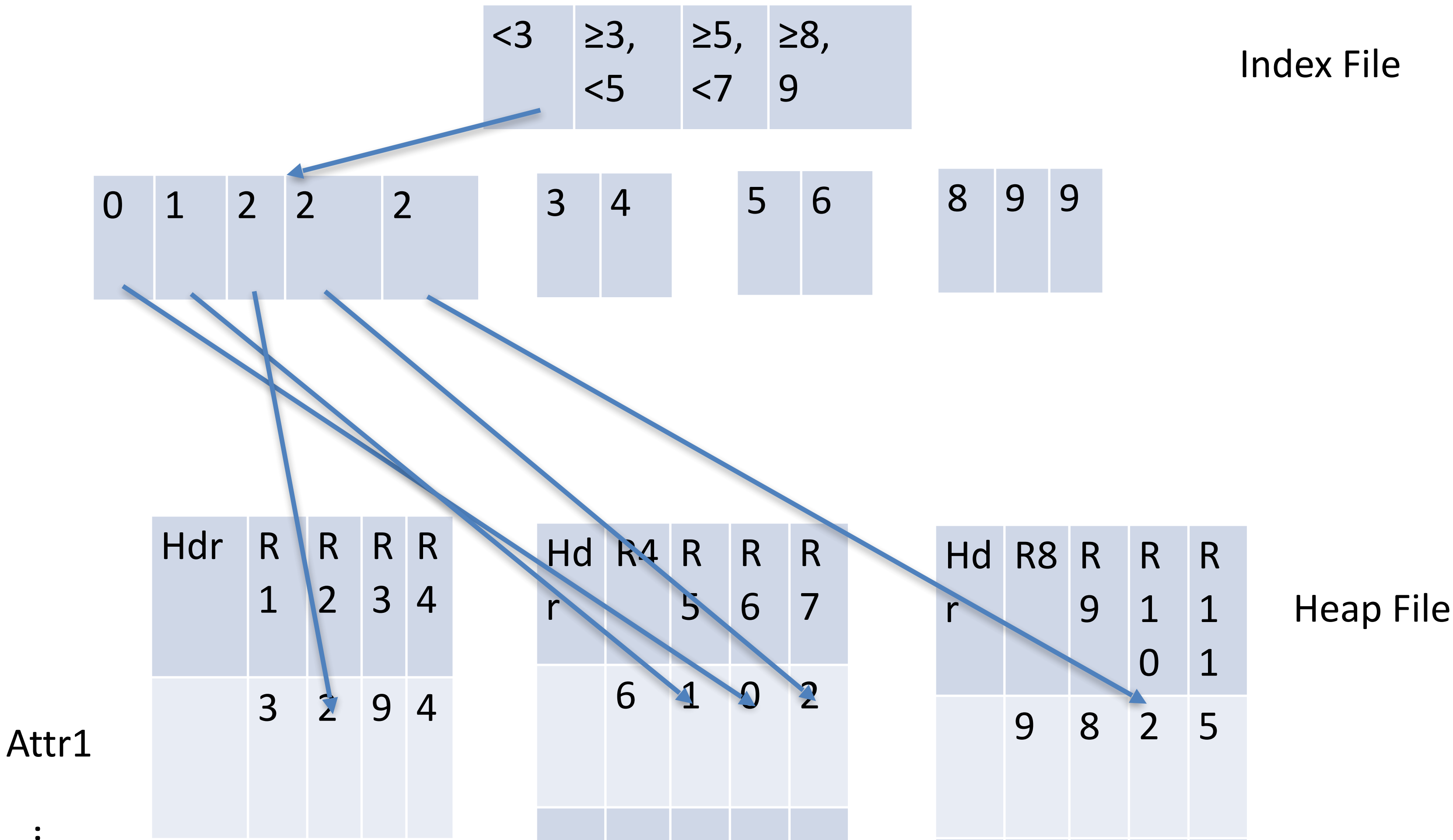
$1\text{x}10^{-6}B + 1\text{x}10^{-9}B > 10$

$B > 9.99\text{x}10^{6}$

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

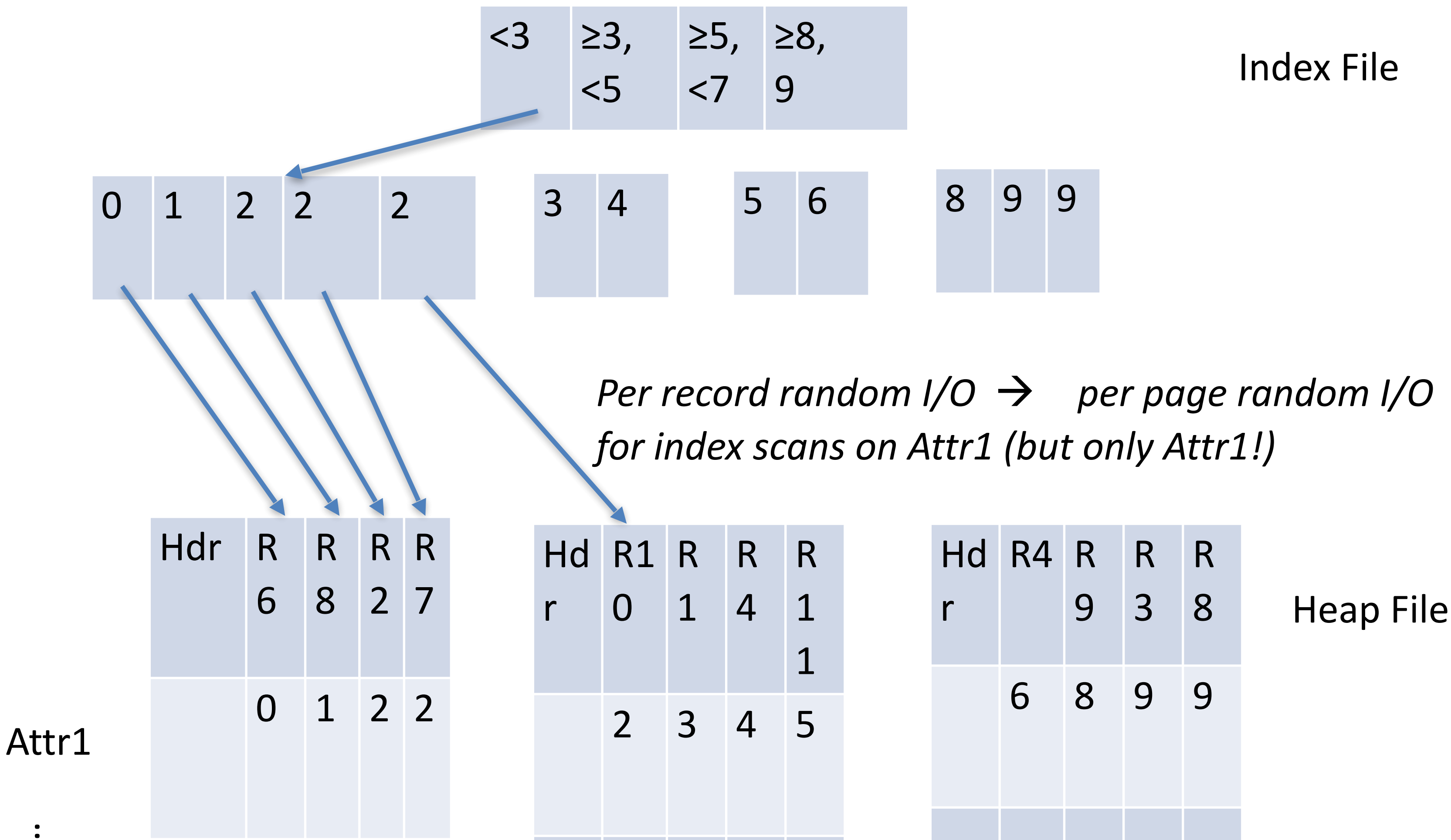
Clustered Index

- 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?

$100 \times 10^{-6} * B / \mathbf{1x10^6} + B/1 \times 10^{9} > 10 \times 10^{9} / 1 \times 10^{9}$

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

$B > 9.99 \times 10^{9}$

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
- Trees (B+/R)

Access Method Costs

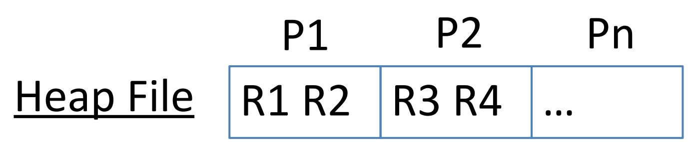
	Heap File	Hash File	B+Tree
Insert	O(1)		
Delete	O(P)		
Scan	O(P) *sequential*		
Lookup	O(P)		

n : number of tuples

P : number of pages in file

B : branching factor of B-Tree

R : number of pages in scanned range

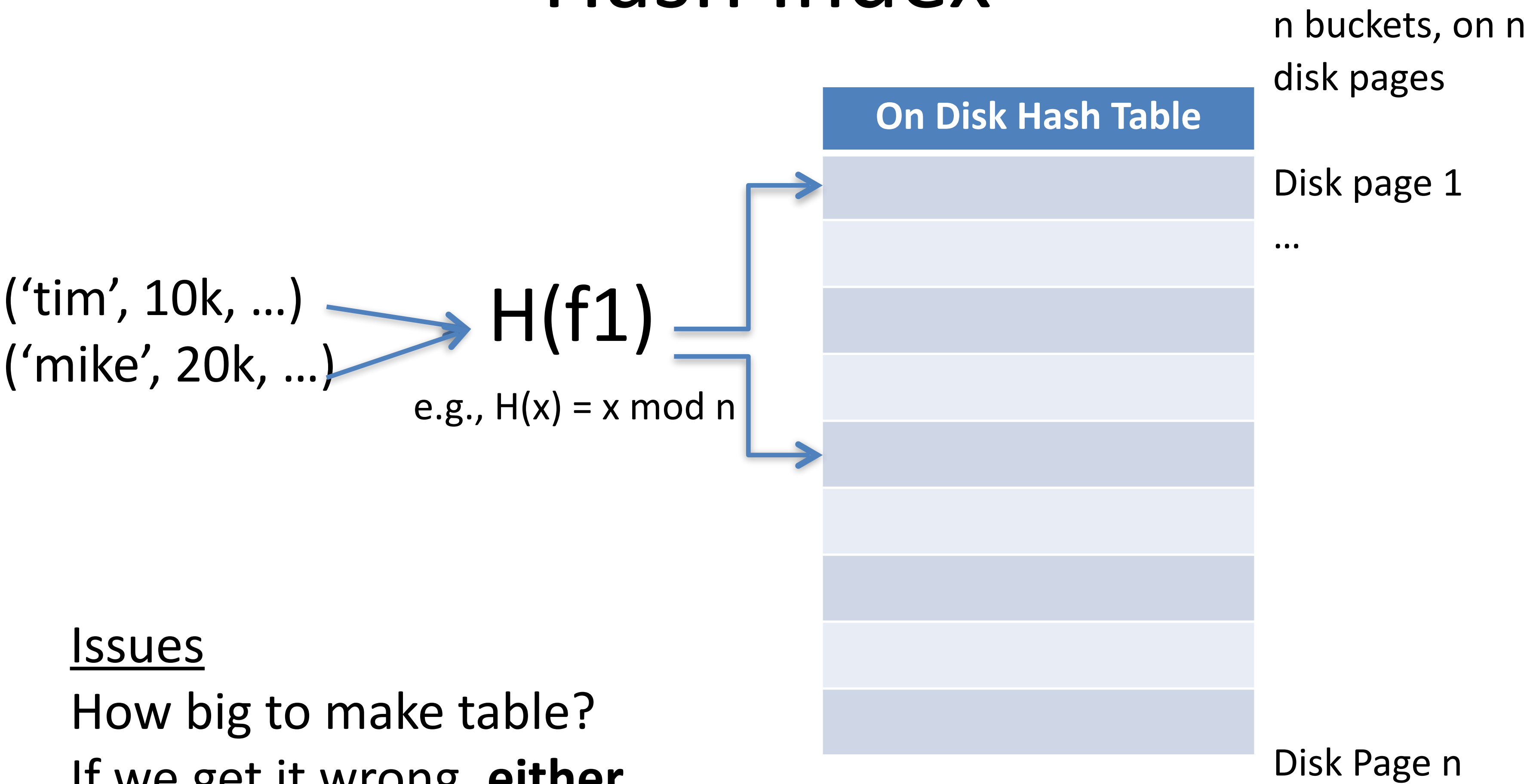


Sequentially stored pages, no seeks between records or pages

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 "tim"

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

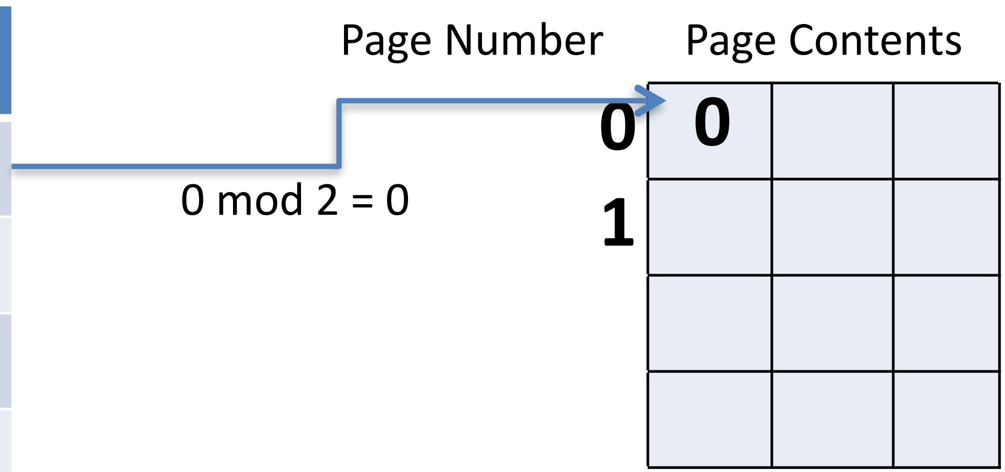
Example

Directory

k=1

$H_k(x)$	Page
0	**0**
1	**1**

Hash Table



$H_k(x)$ = 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	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

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

Page Number	Page Contents		
0	0	0	2
1	3		

- FULL!

2 mod 2 = 0

$H_k(x)$ = 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

Hash Table

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

Only allocate 1 new page!

$H_k(x)$ = 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**	

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

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

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

Access Method Costs

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

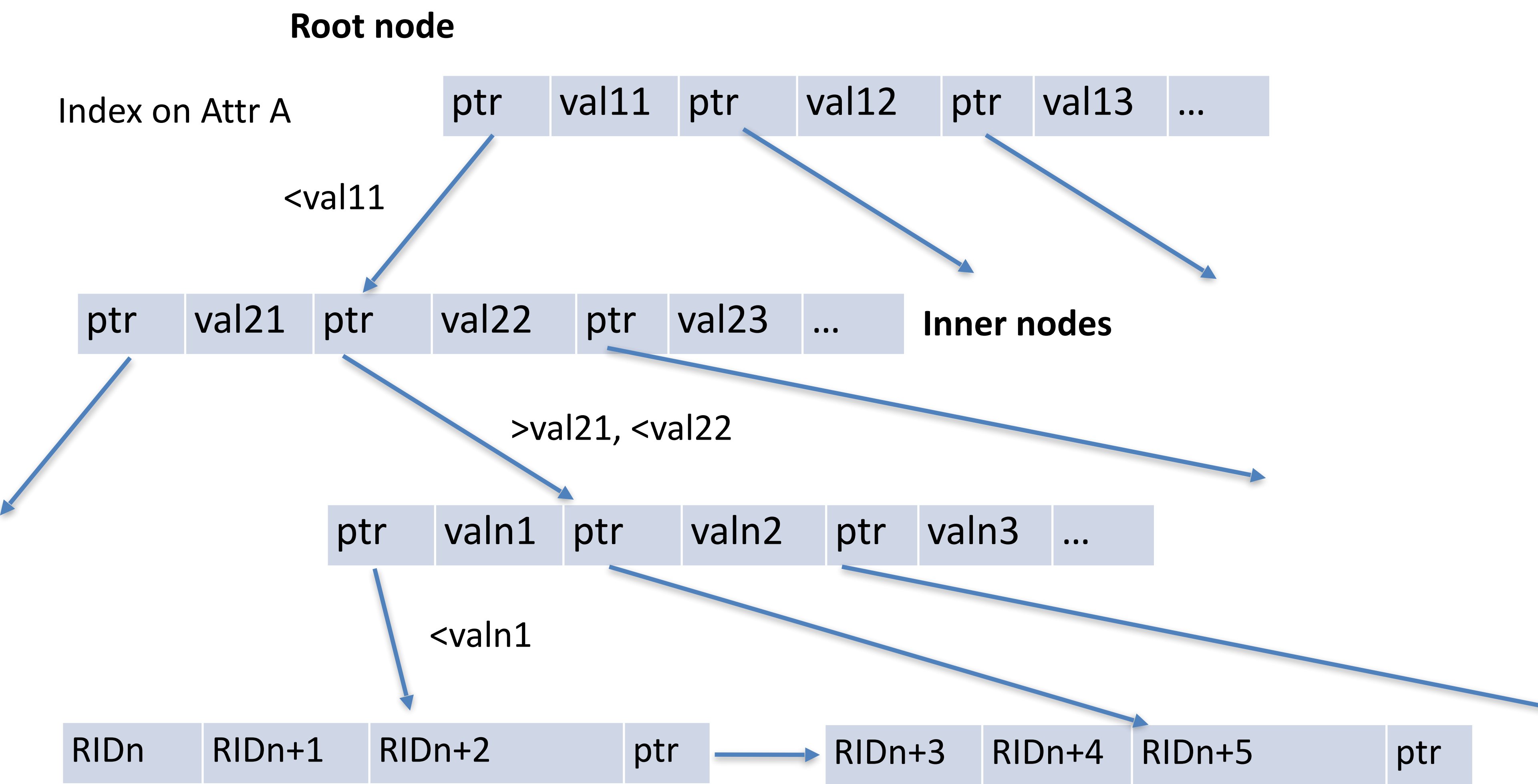
n : number of tuples

P : number of pages in file

B : branching factor of B-Tree

R : number of pages in range

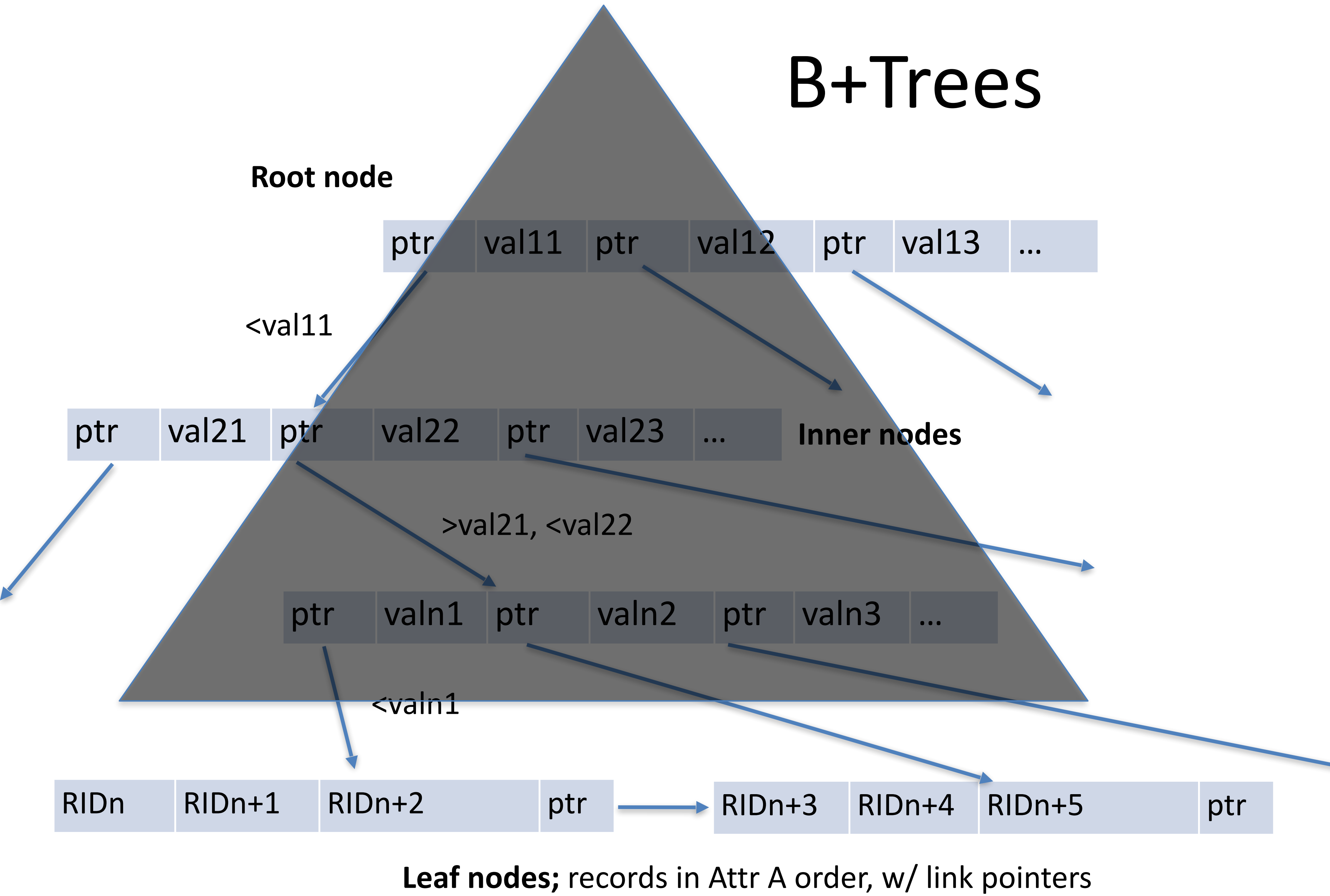
B+Trees



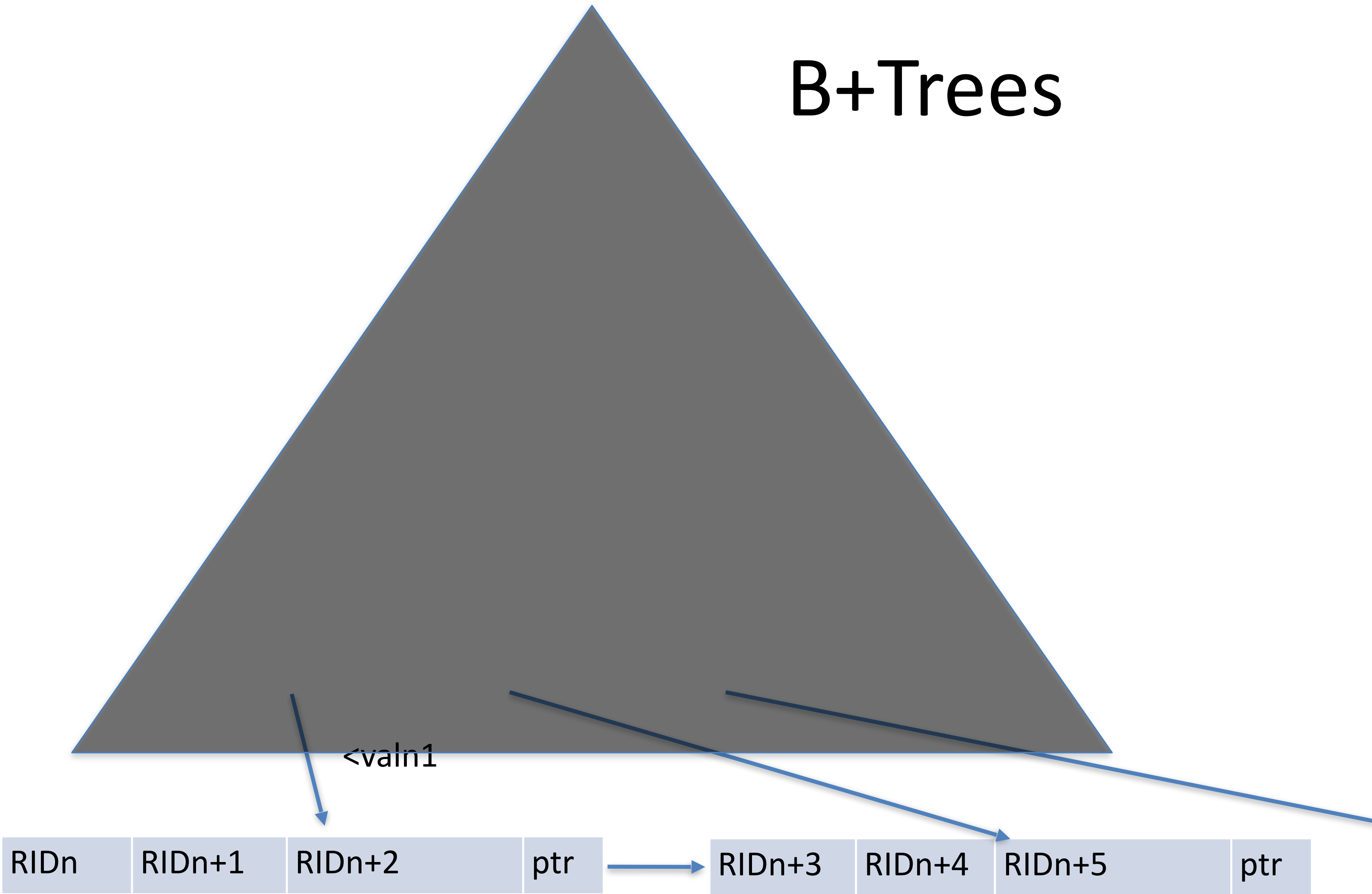
RID: Record ID → a reference (pointer) to a record in heap file

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

B+Trees


Root node
ptr
val11
ptr
val12
ptr
val13
...
<val11
ptr
val21
ptr
val22
ptr
val23
...
Inner nodes
>val21, <val22
ptr
valn1
ptr
valn2
ptr
valn3
...
<valn1
RIDn
RIDn+1
RIDn+2
ptr
RIDn+3
RIDn+4
RIDn+5
ptr
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)
R-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

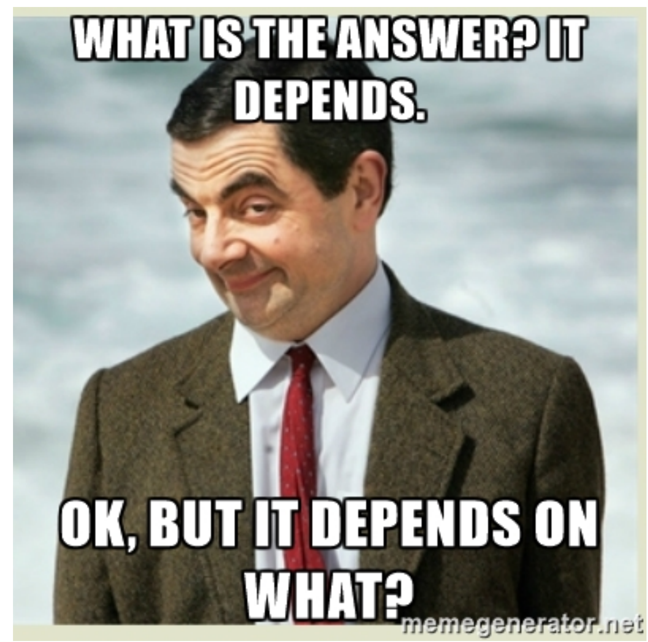
B : branching factor of B-Tree

R : number of pages in range

https://clicker.mit.edu/6.5830/
Study Break

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

```
SELECT MAX(sal) FROM emp
SELECT sal FROM emp WHERE id = 1
SELECT name FROM emp
   WHERE sal > 100k
SELECT name FROM emp
   WHERE sal > 100k AND dept = 2
```



https://clicker.mit.edu/6.5830/
Study Break

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

```
SELECT MAX(sal) FROM emp
SELECT sal FROM emp WHERE id = 1
SELECT name FROM emp WHERE sal > 100k
SELECT name FROM emp WHERE sal > 100k AND dept = 2
```

(A) BTree, Btree, None, Hash
(B) BTree, Hash, BTree, none
(C) None, Hash, BTree, BTree
(D) BTree, Hash, BTree, BTree

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)
```

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

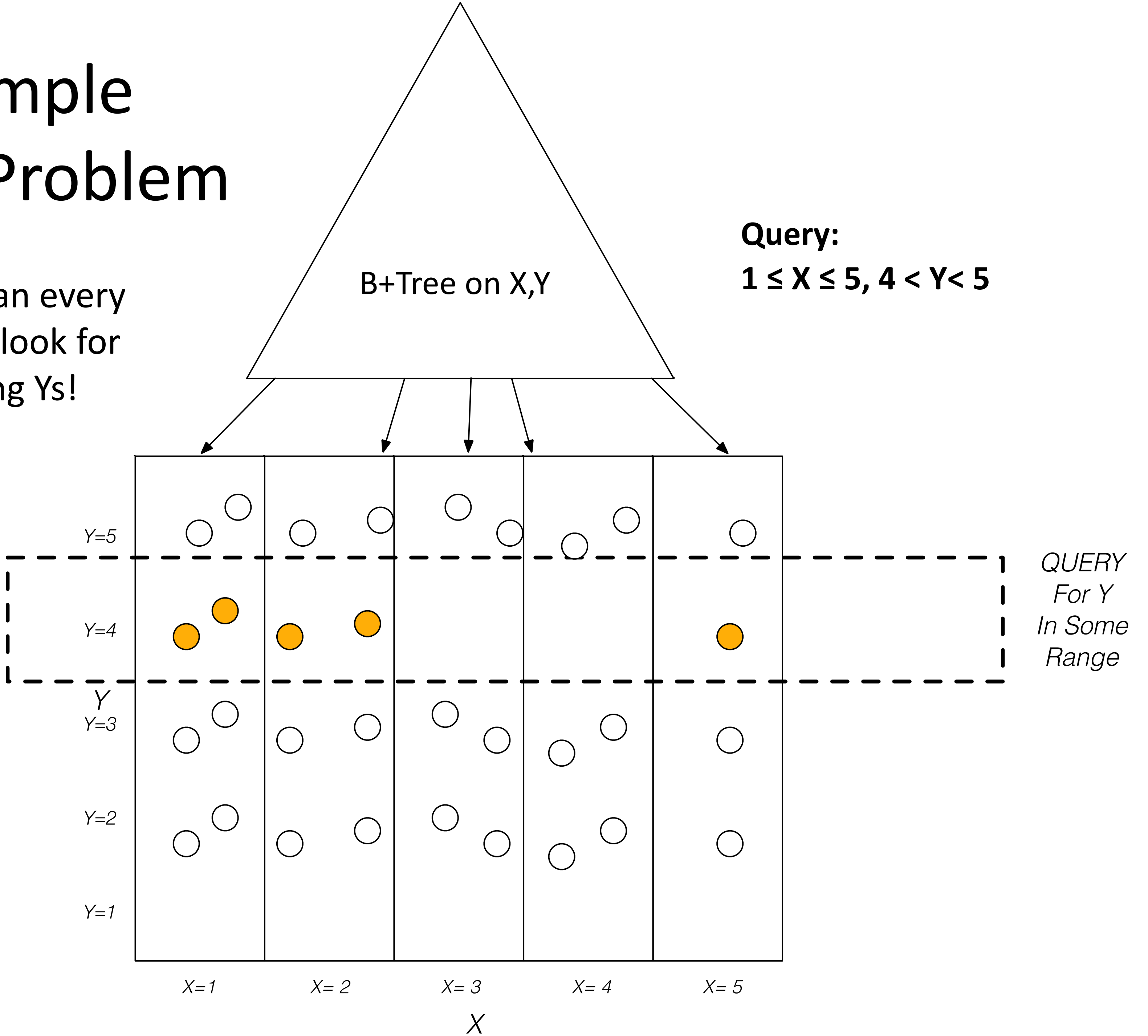
Supports efficient range lookups on X
Allows further filtering on Y, but may be inefficient

Doesn’t support lookups on Y

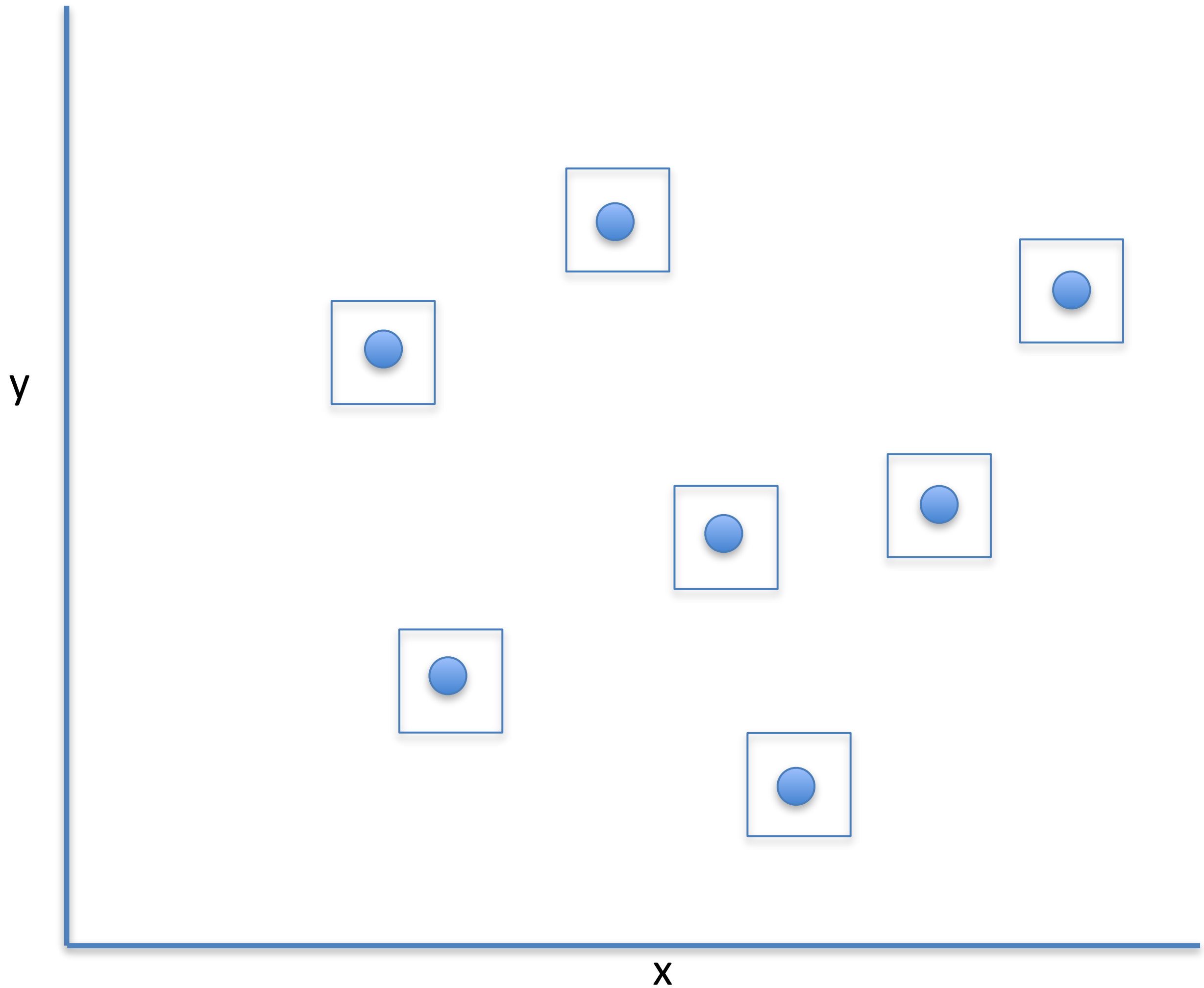
Example of the Problem

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

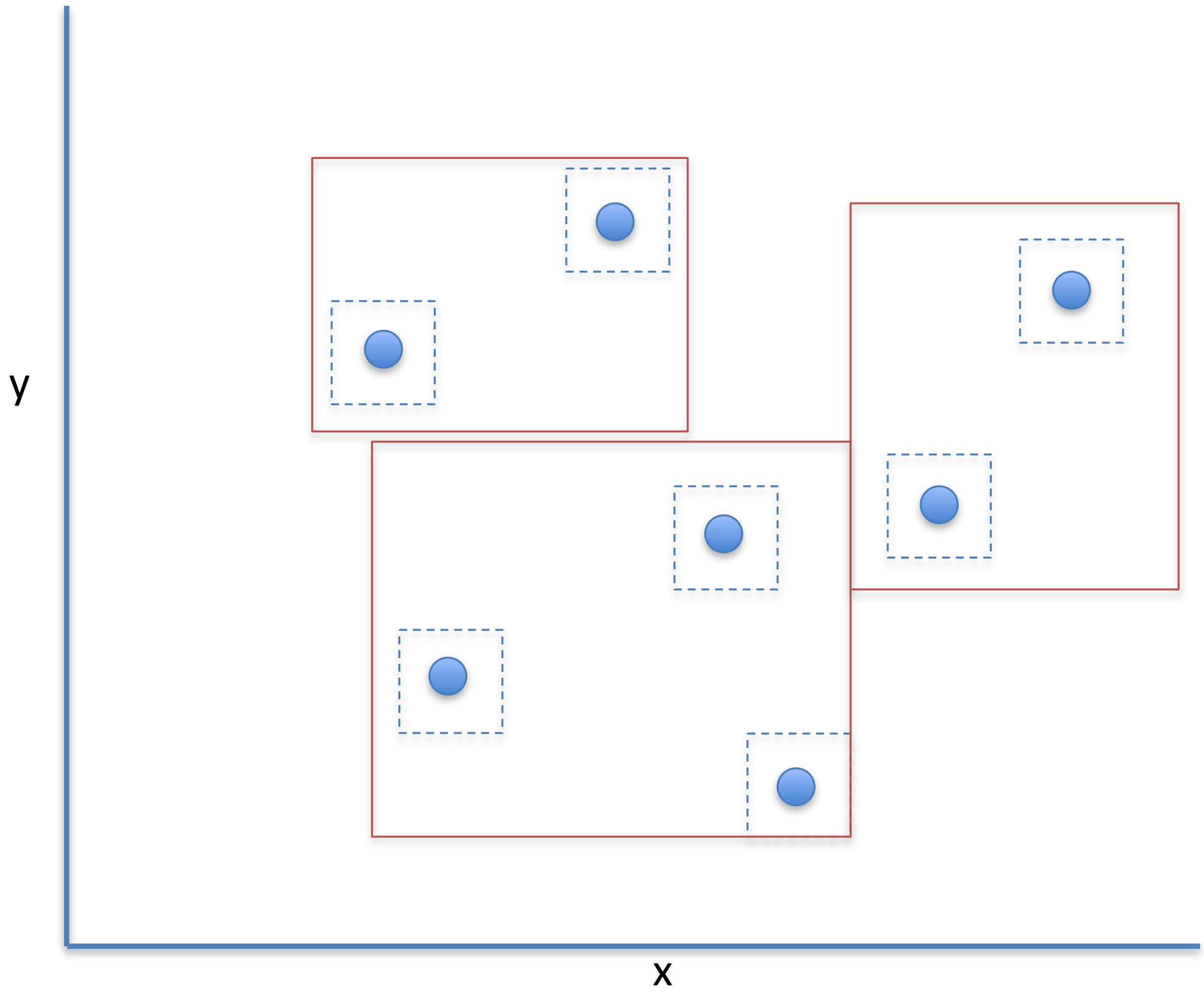
Query:
1 ≤ X ≤ 5, 4 < Y< 5



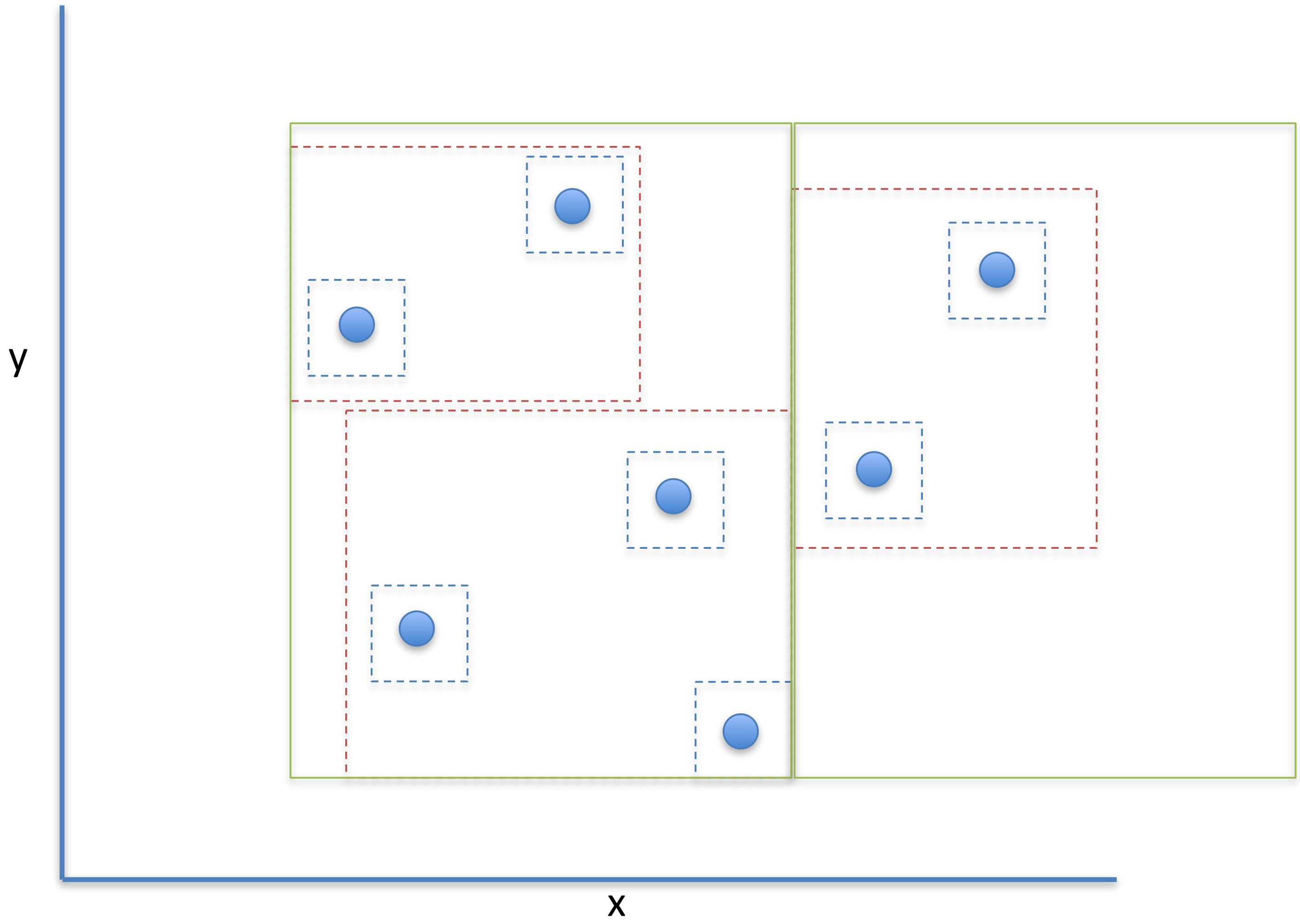
R-Trees / Spatial Indexes

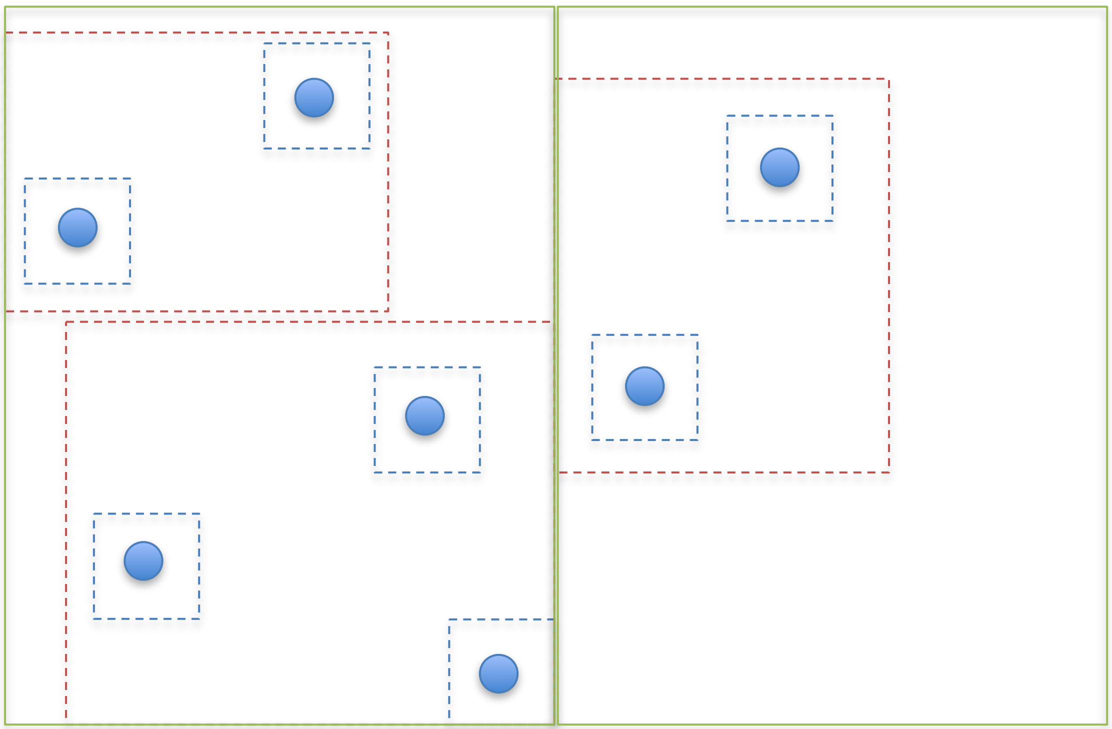


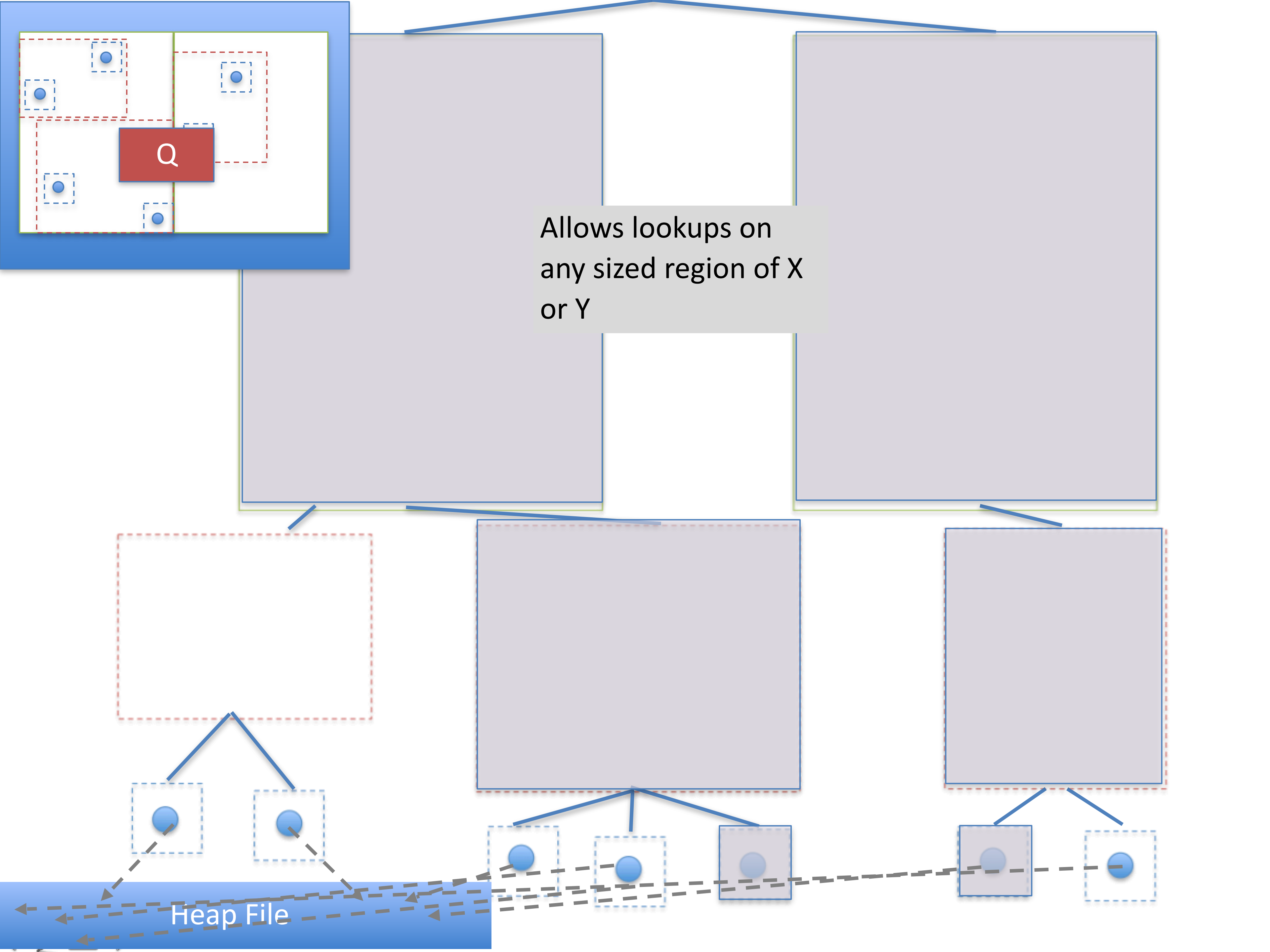
R-Trees / Spatial Indexes



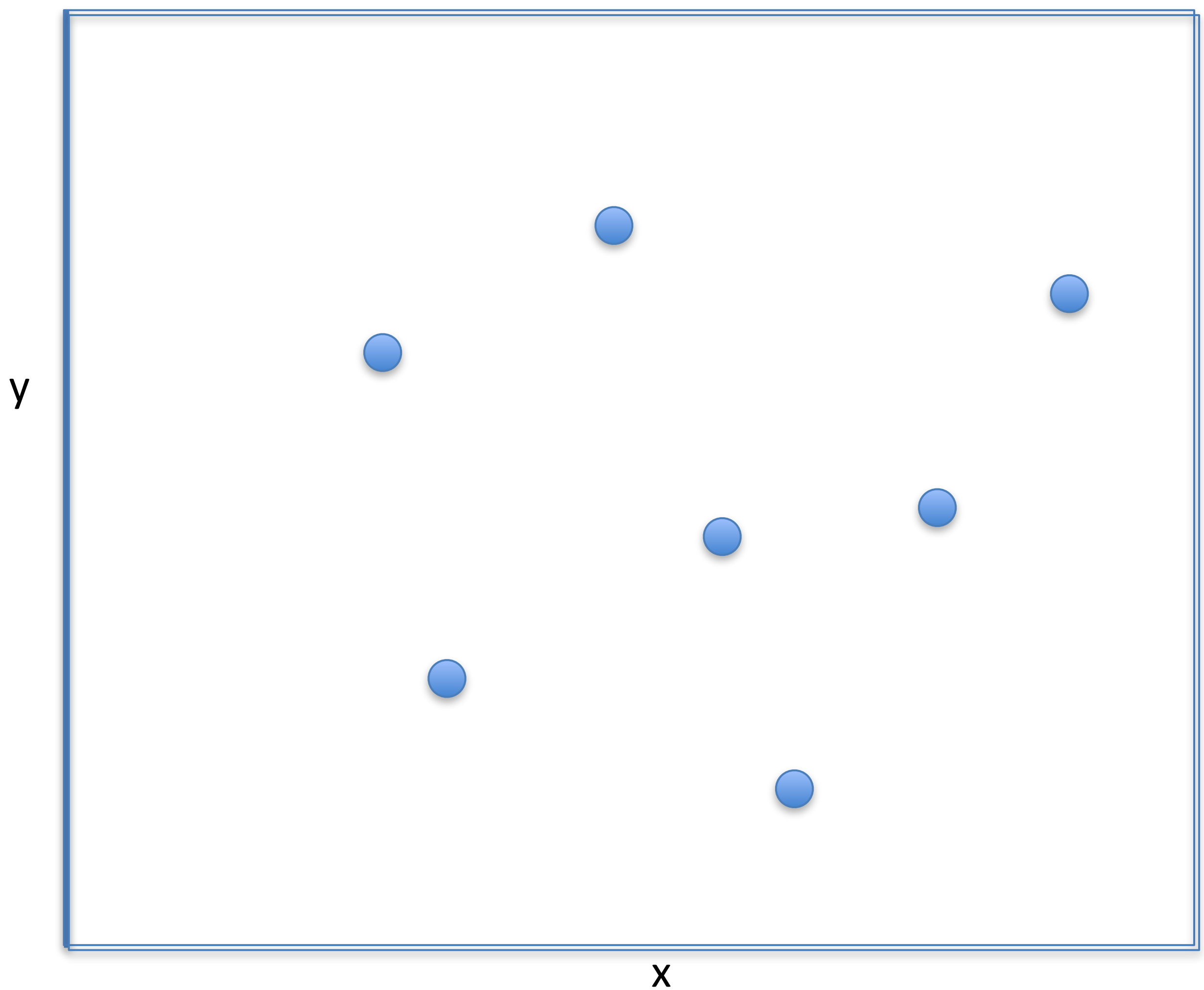
R-Trees / Spatial Indexes



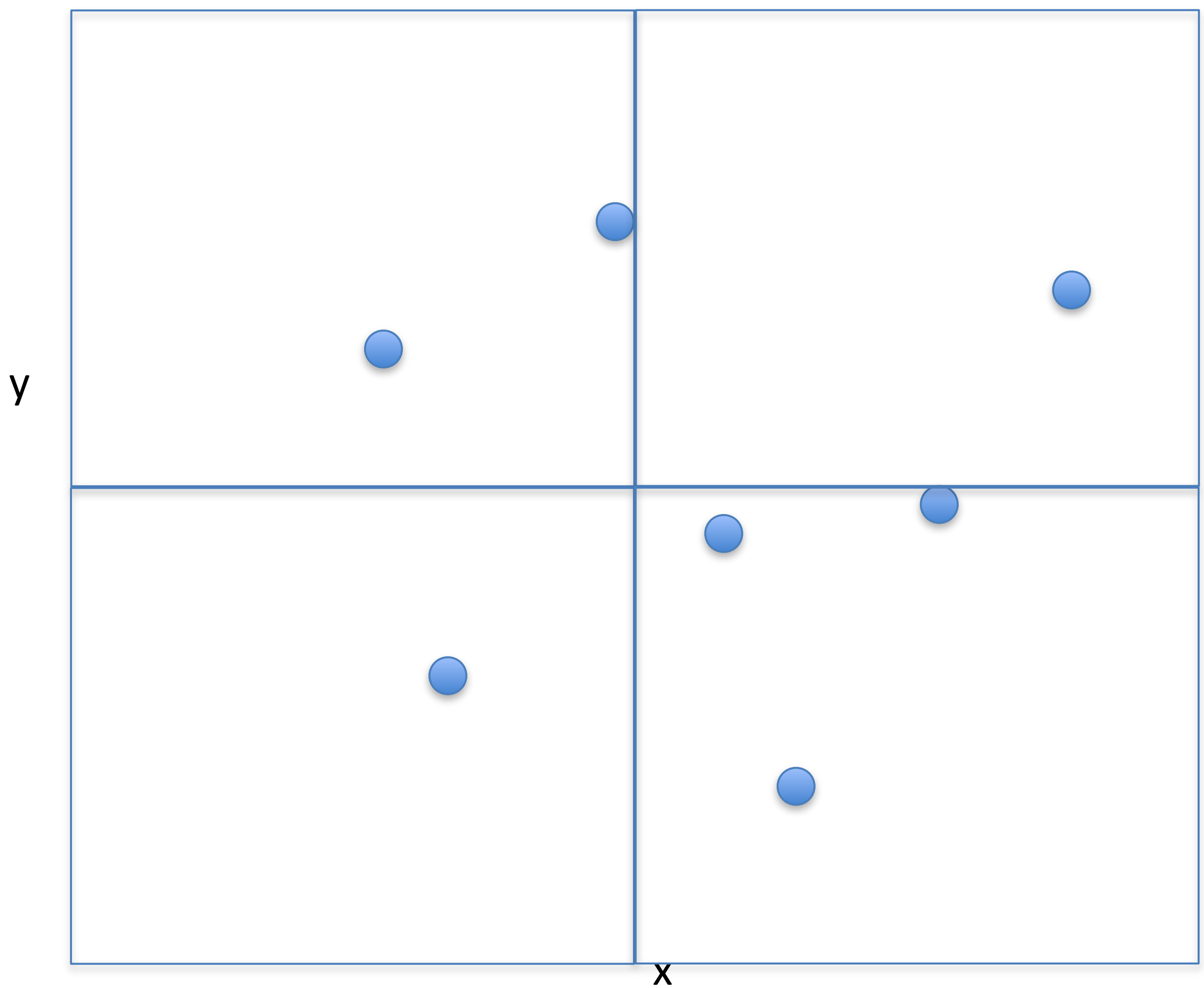



Q
Allows lookups on any sized region of X or Y
Heap File

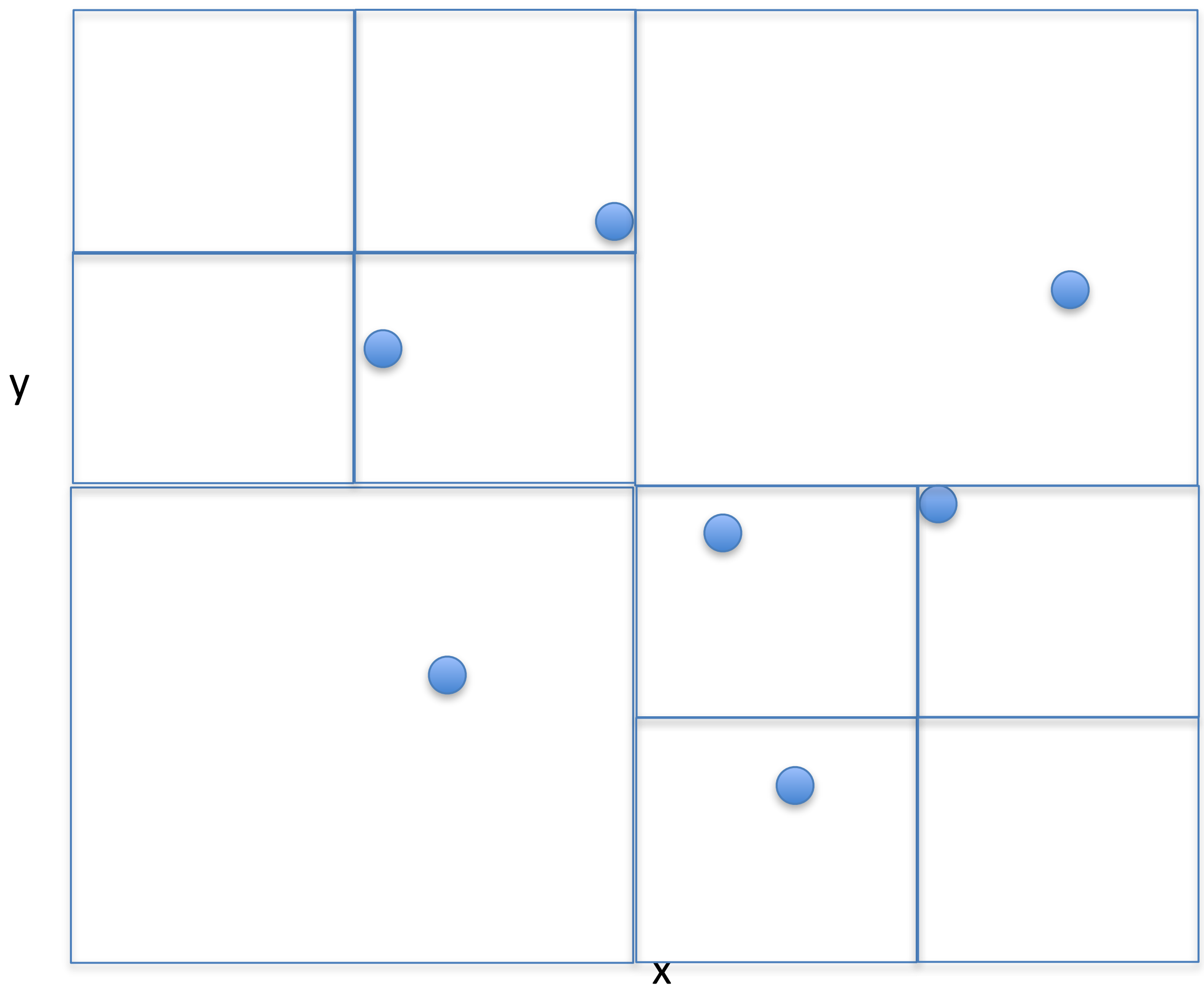
Quad-Tree



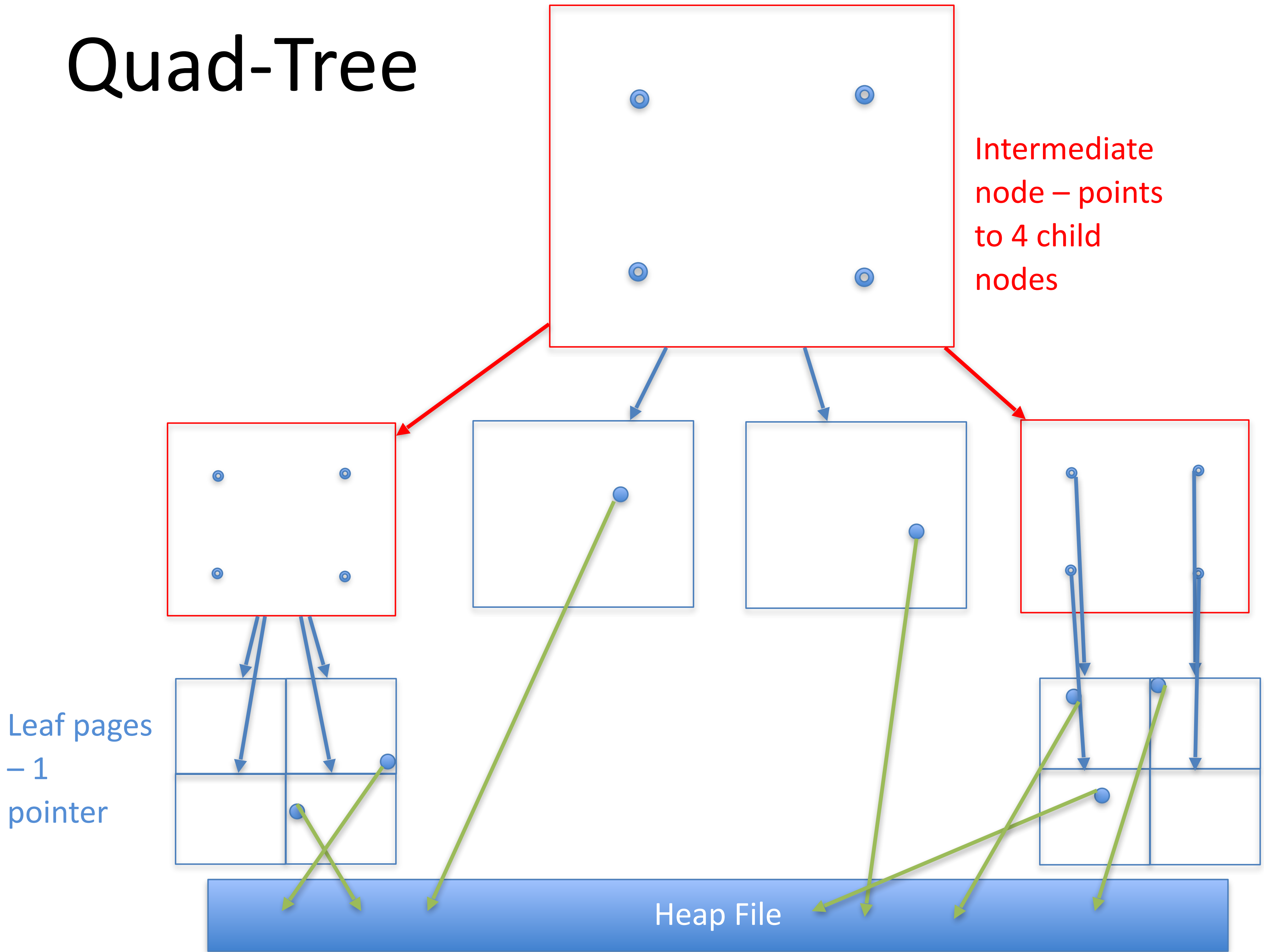
Quad-Tree



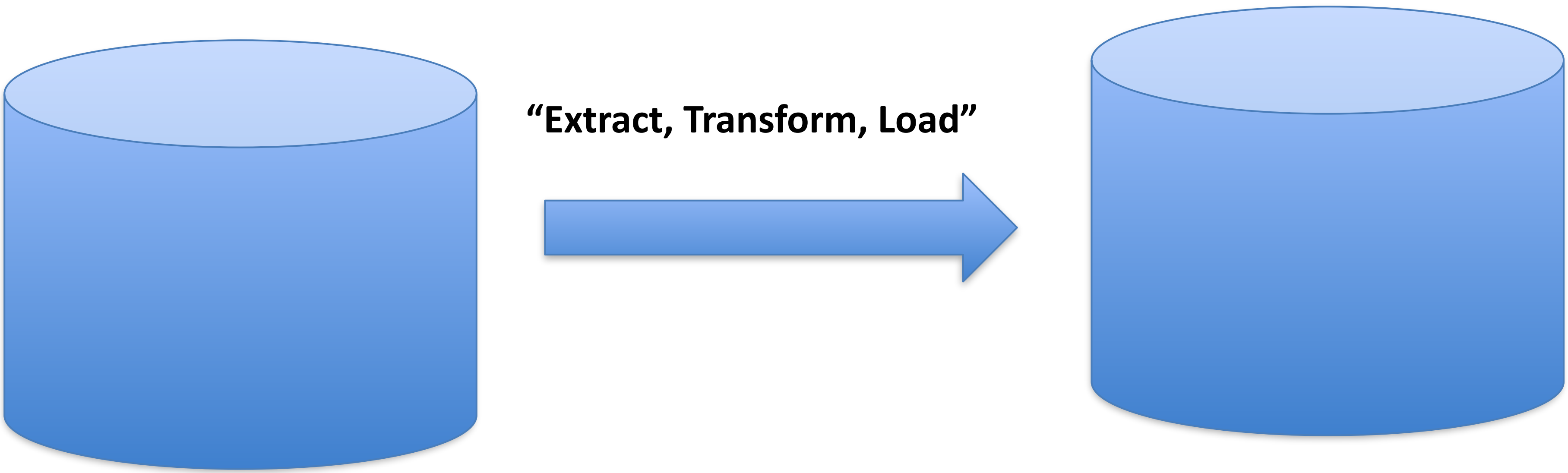
Quad-Tree



Quad-Tree


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

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