6.5830 Lecture 5

Database Internals Continued

September 20, 2023

What is GoDB?

- A basic database system
- SQL Front-end (Provided for later labs)
  - Heap files (Lab 1)
  - Buffer Pool (Labs 1)
  - Basic Operators (Labs 1 & 2)
    - Scan, Filter, JOIN, Aggregate
  - Transactions (Lab 3)
  - Recovery (Lab 3)
  - Query optimizer
  - B-Tree Indexes

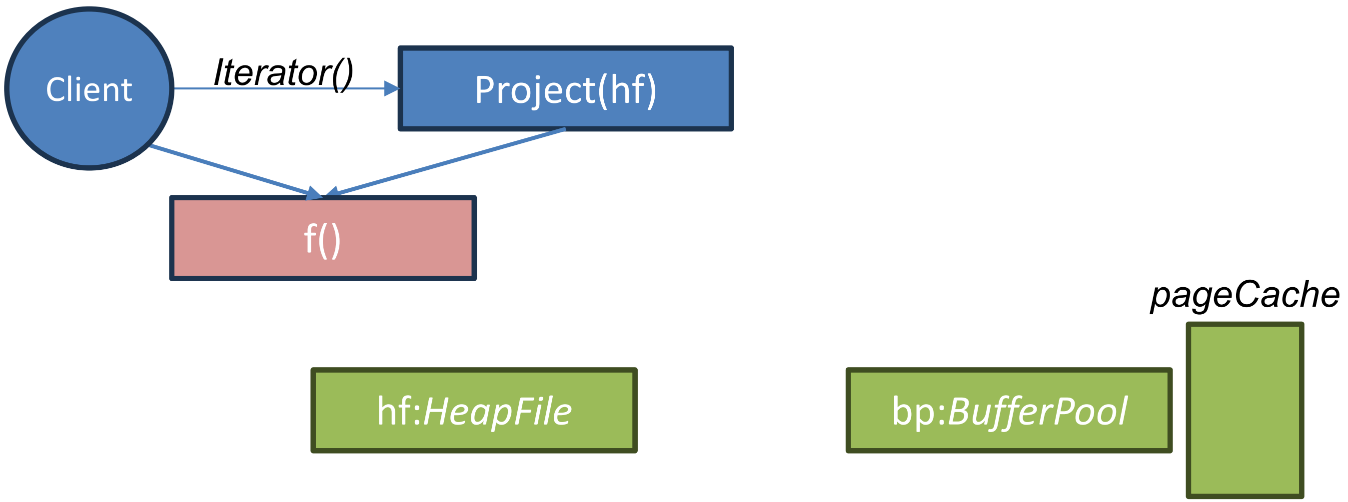
Start Early: It looks trivial until you get into it

Before Starting Lab 1

Finishing Lab 1

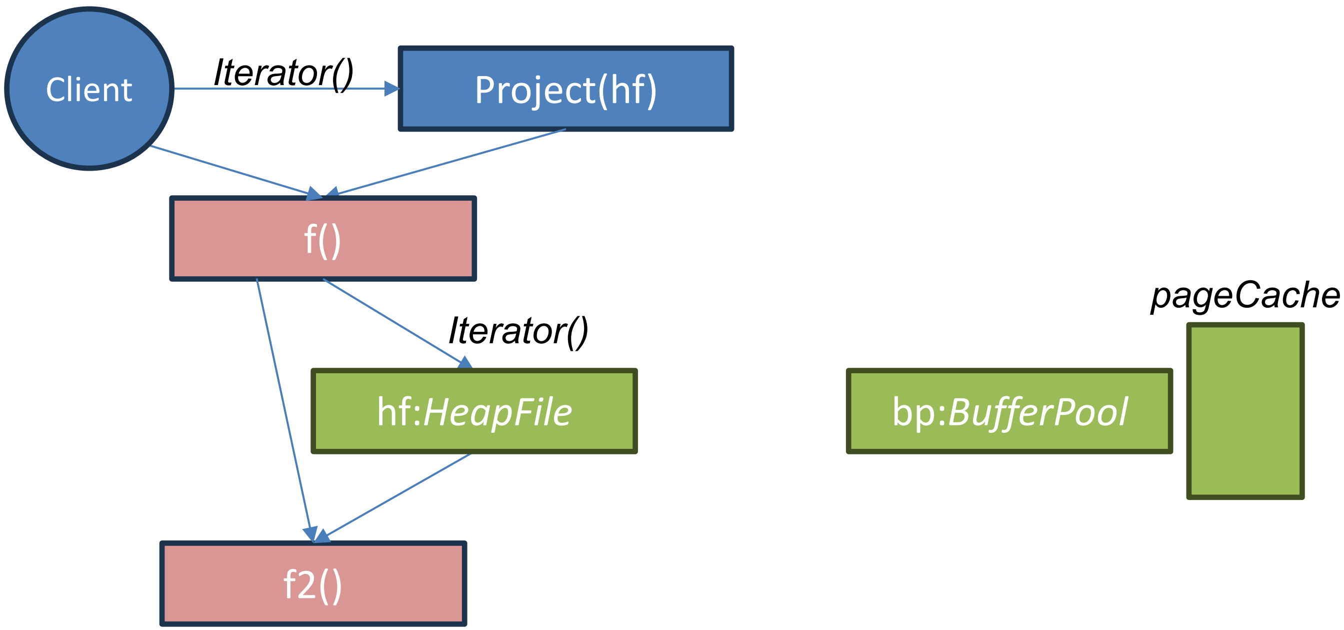

Client
pageCache
hf:HeapFile
bp:BufferPool

Example



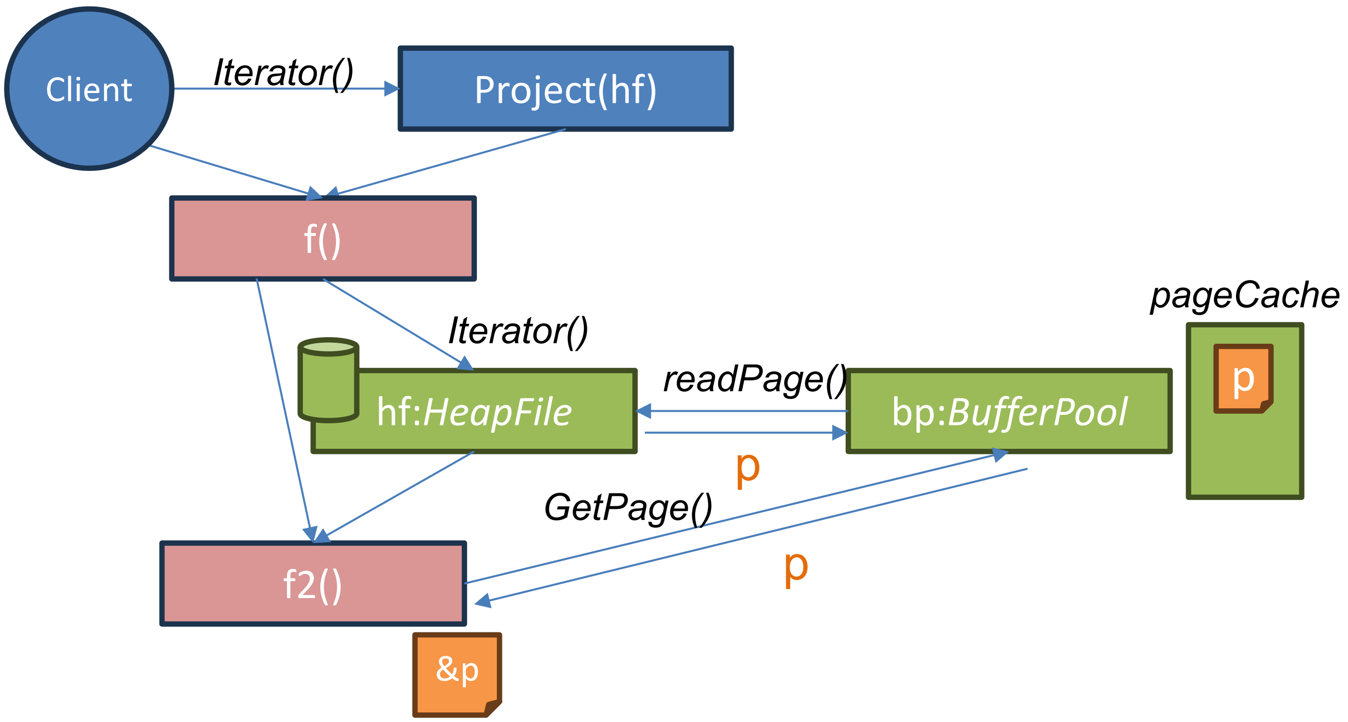
Client
Iterator()
Project(hf)
f()
pageCache
hf:HeapFile
bp:BufferPool

Example

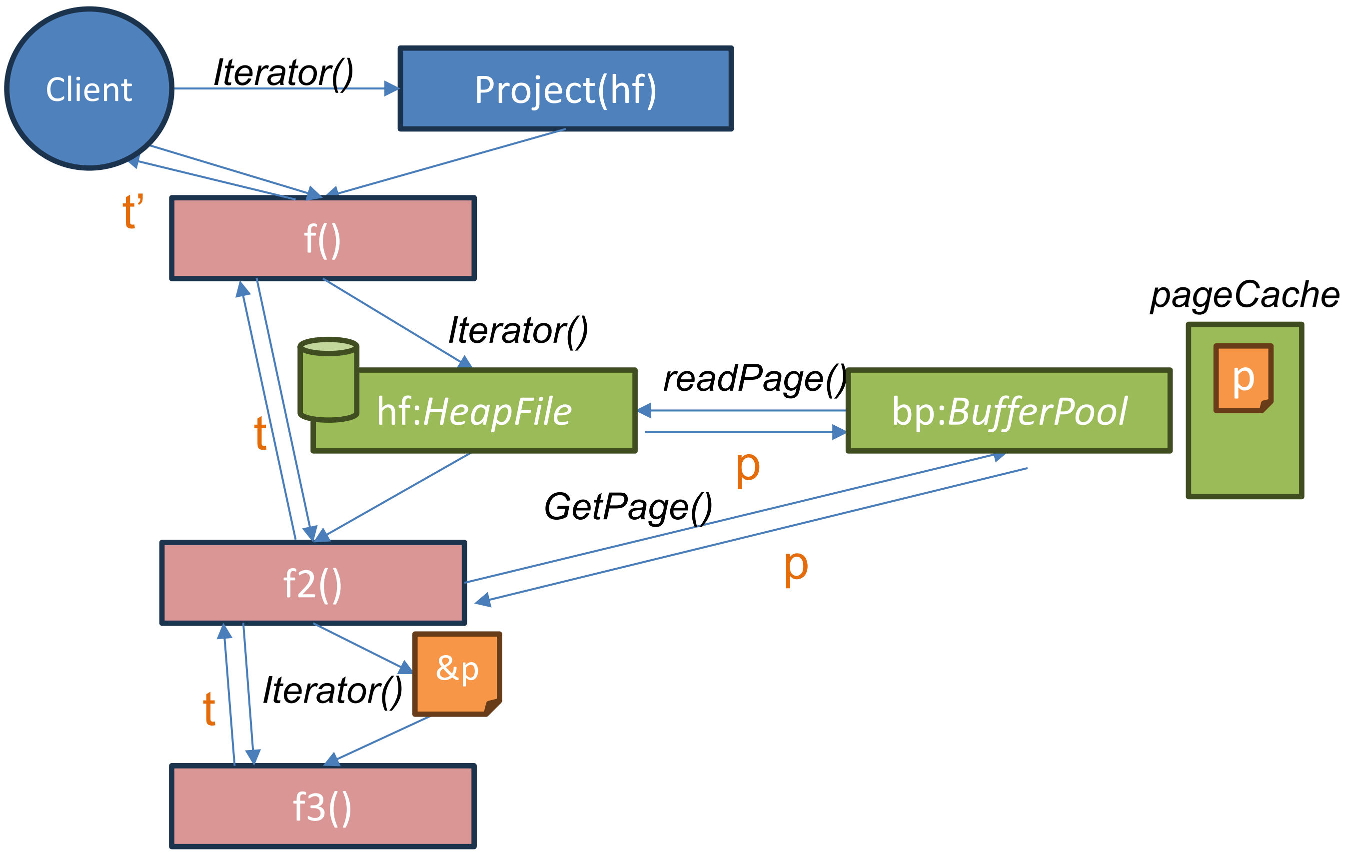


Client
Iterator()
Project(hf)
f()
Iterator()
hf:*HeapFile*
f2()
pageCache
bp:*BufferPool*

Example


Client
Iterator()
Project(hf)
f()
Iterator()
pageCache
p
readPage()
hf:HeapFile
bp:BufferPool
p
GetPage()
f2()
p
&p

Example



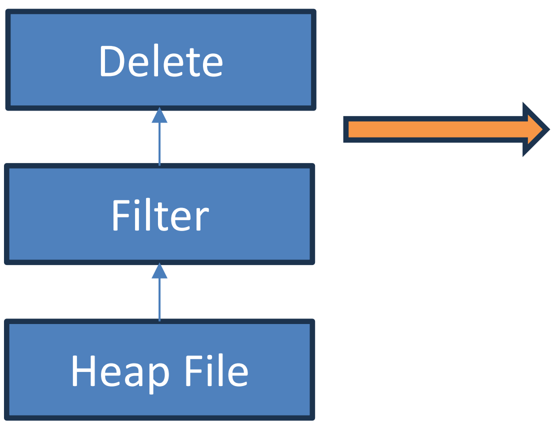
Client
Iterator()
Project(hf)
t'
f()
Iterator()
pageCache
hf:HeapFile
readPage()
bp:BufferPool
p
t
p
GetPage()
p
f2()
&p
Iterator()
t
f3()

Deleting Records and Rids

- Consider a query like:

  DELETE FROM x WHERE f > 10

  This is translated into a plan like



Q: How does the delete operator know which records to delete?

A: Each record from the HeapFile is annotated with a *record id* that is used to identify the position of the record in the heap file to be deleted

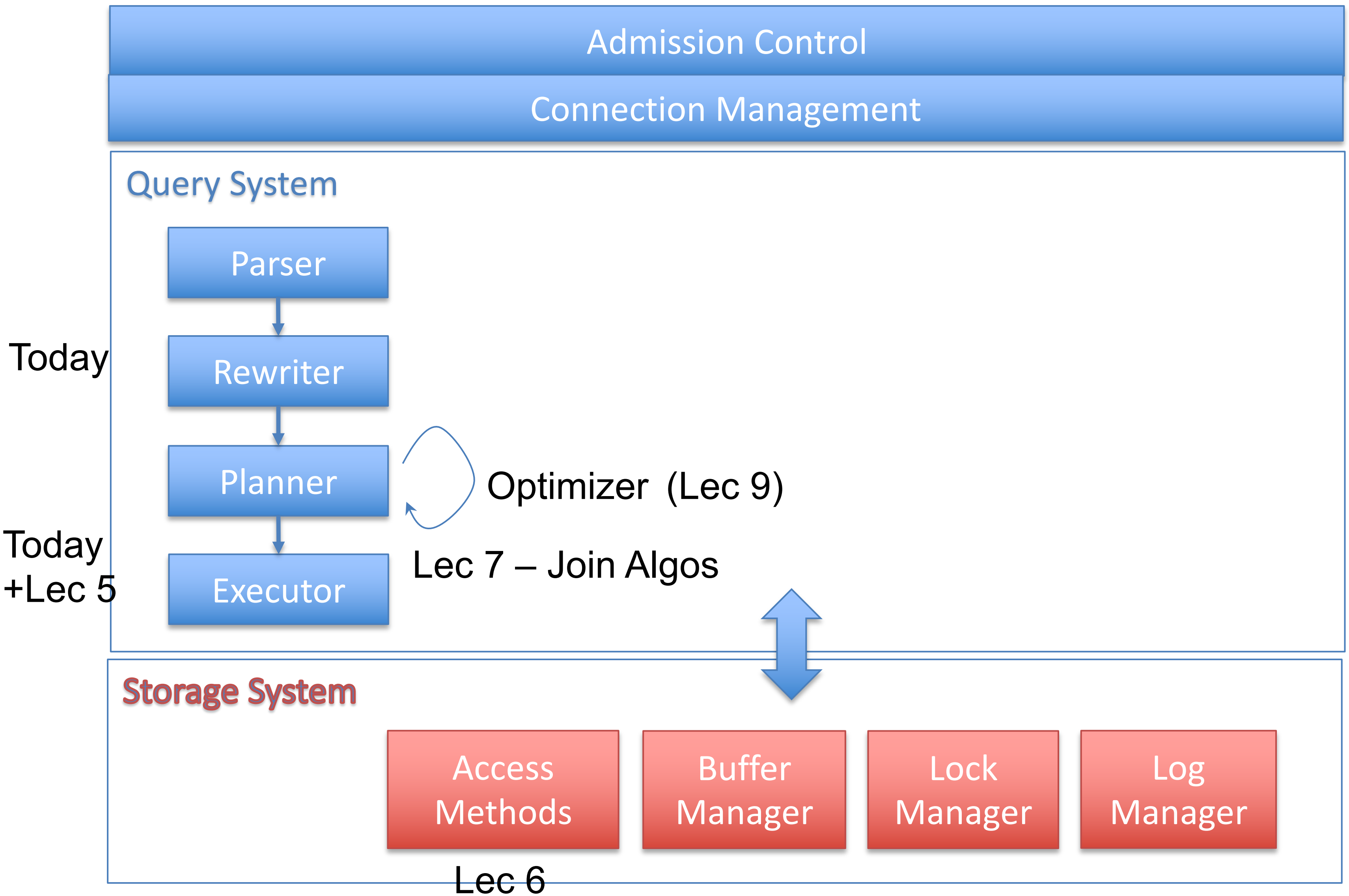
Deleting Records and Rids

```go
// Remove the provided tuple from the HeapFile. This method should use the
// [Tuple.Rid] field of t to determine which tuple to remove.
// This method is only called with tuples that are read from storage via the
// [Iterator] method, so you can so you can supply the value of the Rid
// for tuples as they are read via [Iterator]. Note that Rid is an empty interface,
// so you can supply any object you wish. You will likely want to identify the
// heap page and slot within the page that the tuple came from.
func (f *HeapFile) deleteTuple(t *Tuple, tid TransactionID) error {
```

- deleteTuple will be called by the delete operator
- Using the t.Rid object, you can clear out the position in the heap file containing the record
- Your heapfile implementation supplies the Rid in the iterator, and so you can identify this position however you like
- A standard Rid implementation is a page number and a slot within the page
  - Recall that all pages have the same number of slots

```go
func computeFieldSum(fileName string, td TupleDesc, sumField string
) (int, error) {

    //Create buffer pool
    bp := NewBufferPool(10)

    hf, err := NewHeapFile("myfile.dat", &td, bp)
    …
    err = hf.LoadFromCSV(CSVfile, true, ",", false)

    //find the column
    fieldNo, err := findFieldInTd(FieldType{sumField, "", IntType}, &td)

    //Start a transaction -> we will do the implementation in another lab
    tid := NewTID()
    bp.BeginTransaction(tid)
    iter, err := hf.Iterator(tid)

    //Iterate through the tuples and sum them up.
    sum := 0
    for {
        tup, err := iter()
        f := tup.Fields[fieldNo].(IntField)
        sum += int(f.Value)
    }

    bp.CommitTransaction() //commit transaction
    return sum, nil //return the value
}
```

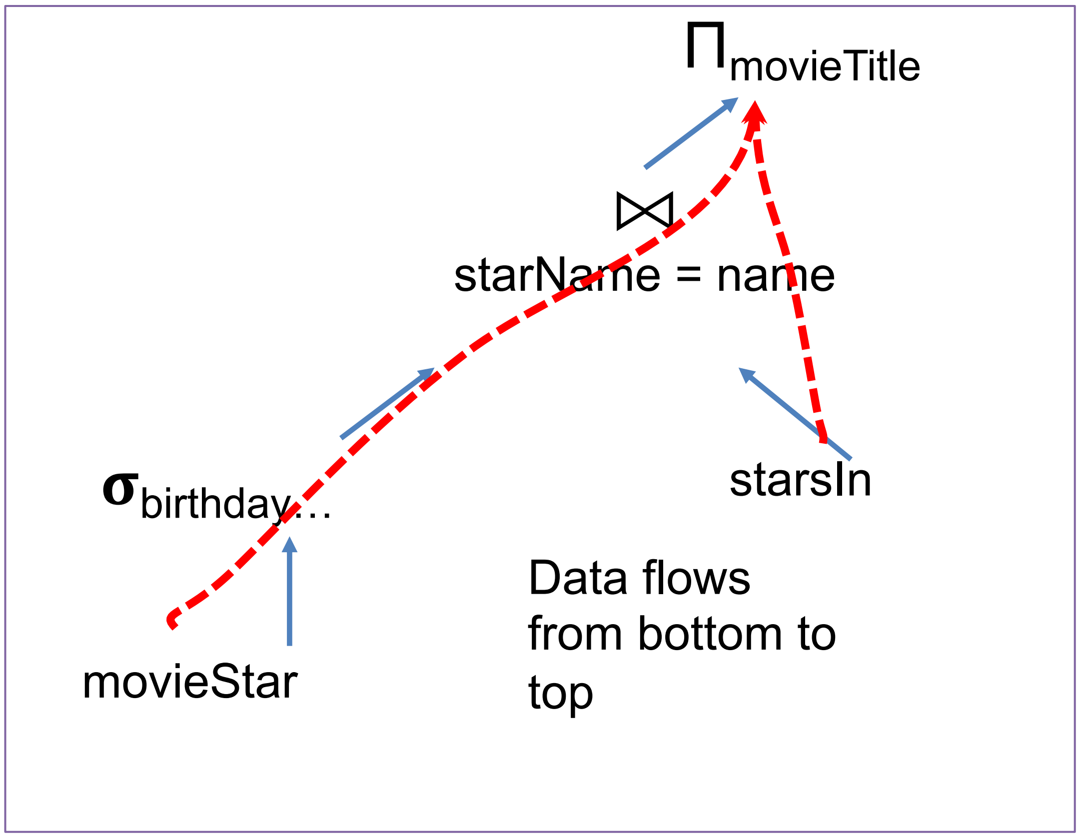
Plan for Next Few Lectures



Query Processing Steps

- Admission Control
- Query Rewriting
- Plan Formulation (SQL → Tree)
- Optimization

Connecting Operators: Iterator Model



Each operator implements a simple iterator interface:

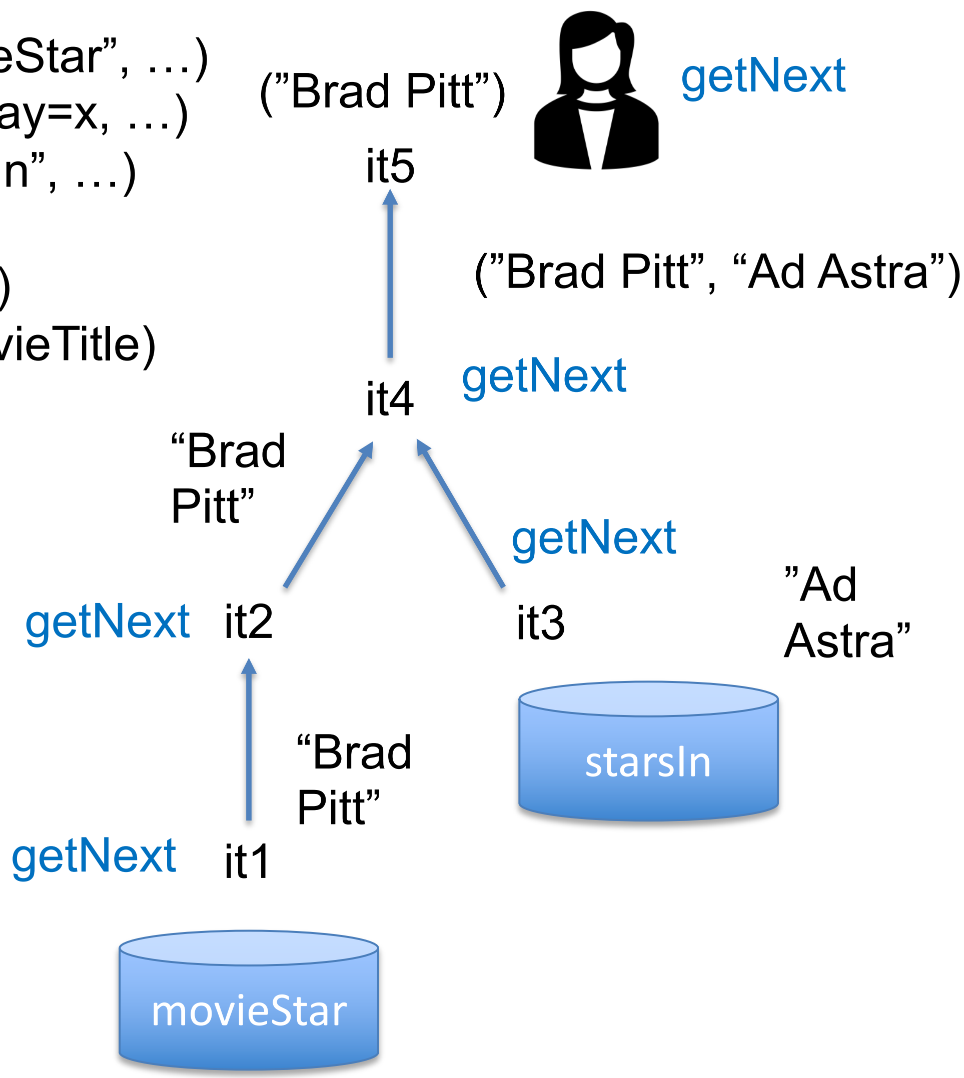
open(params)
getNext() → record
close() → cleanup

Any iterator can compose with any other iterator

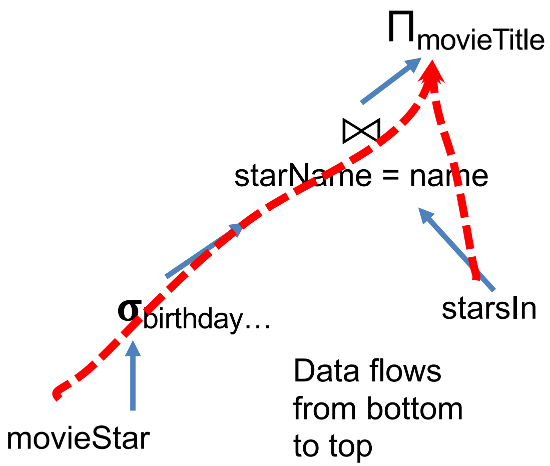
it1 = Scan.open("movieStar", …)
it2 = Filter.open(it1, bday=x, …)
it3 = Scan.open("starsIn", …)
it4 = Join.open(it2, it3,
starName=name)
it5 = Proj.open(it4, movieTitle)

Iterator Model

it1 = Scan.open("movieStar", …)
it2 = Filter.open(it1, bday=x, …)
it3 = Scan.open("starsIn", …)
it4 = Join.open(it2, it3,
        starName=name)
it5 = Proj.open(it4, movieTitle)



GoDB Iterator

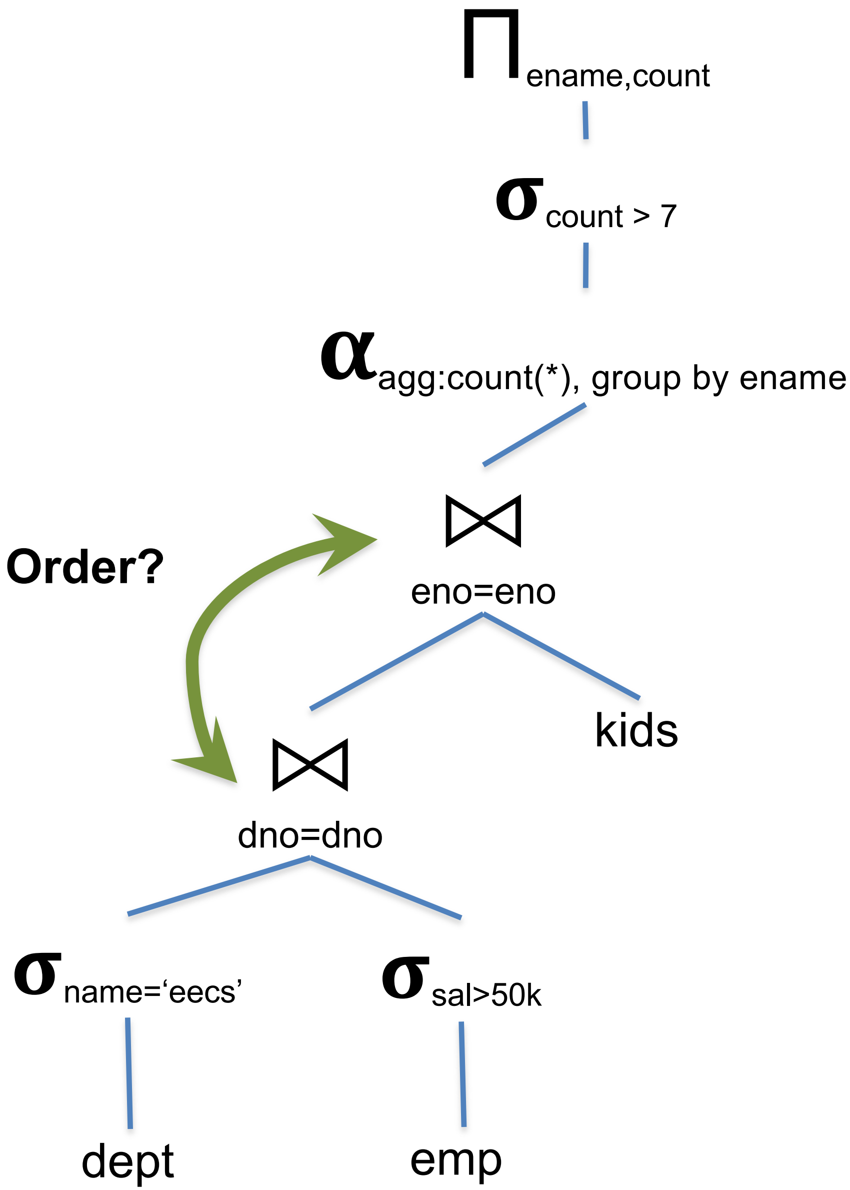



```
hf1, _ := NewHeapFile(MovieStarsFile,…)
filt, _ := NewIntFilter(&ConstExpr{IntField{..}, IntType}, OpGt, &fieldExp, hf1)
hf2, _ := NewHeapFile(StarsInFile, …)
join, _ := NewStringEqJoin (filt, &leftField, hf2, &rightField, 100)
proj, _ := NewProjectOp([]Expr{&fieldExpr}, outNames, false, join)
iter, _ := proj.Iterator(tid)
for {
	tup, err := iter()
	if err != nil { t.Errorf(err.Error())}
	if tup == nil {
		break
	}
	///do something with tup
}
```

This Lecture

- What makes a good query plan?
  - Cost Estimation
- Buffer Management
- Postgres Examples

Cost Estimation

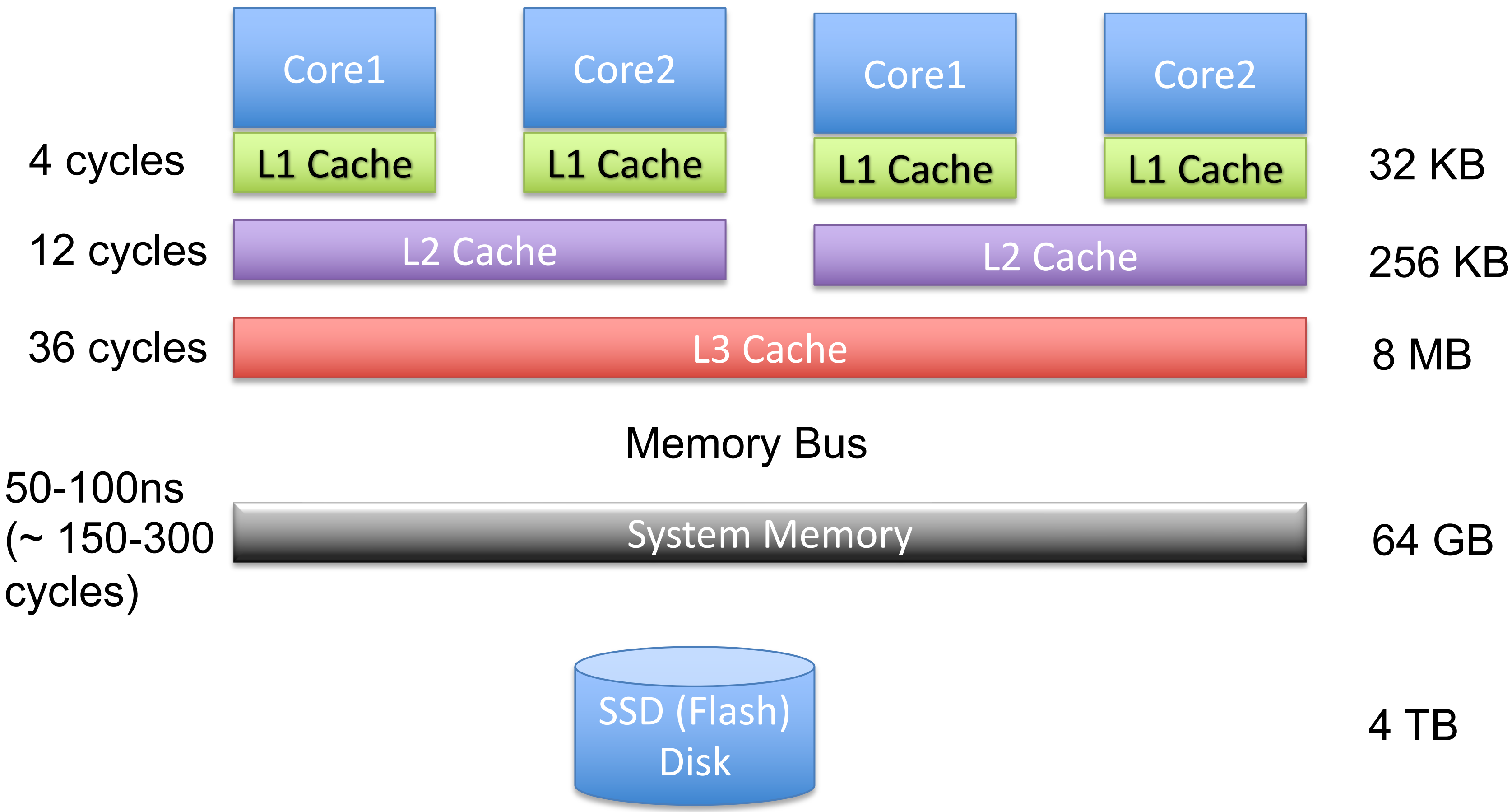


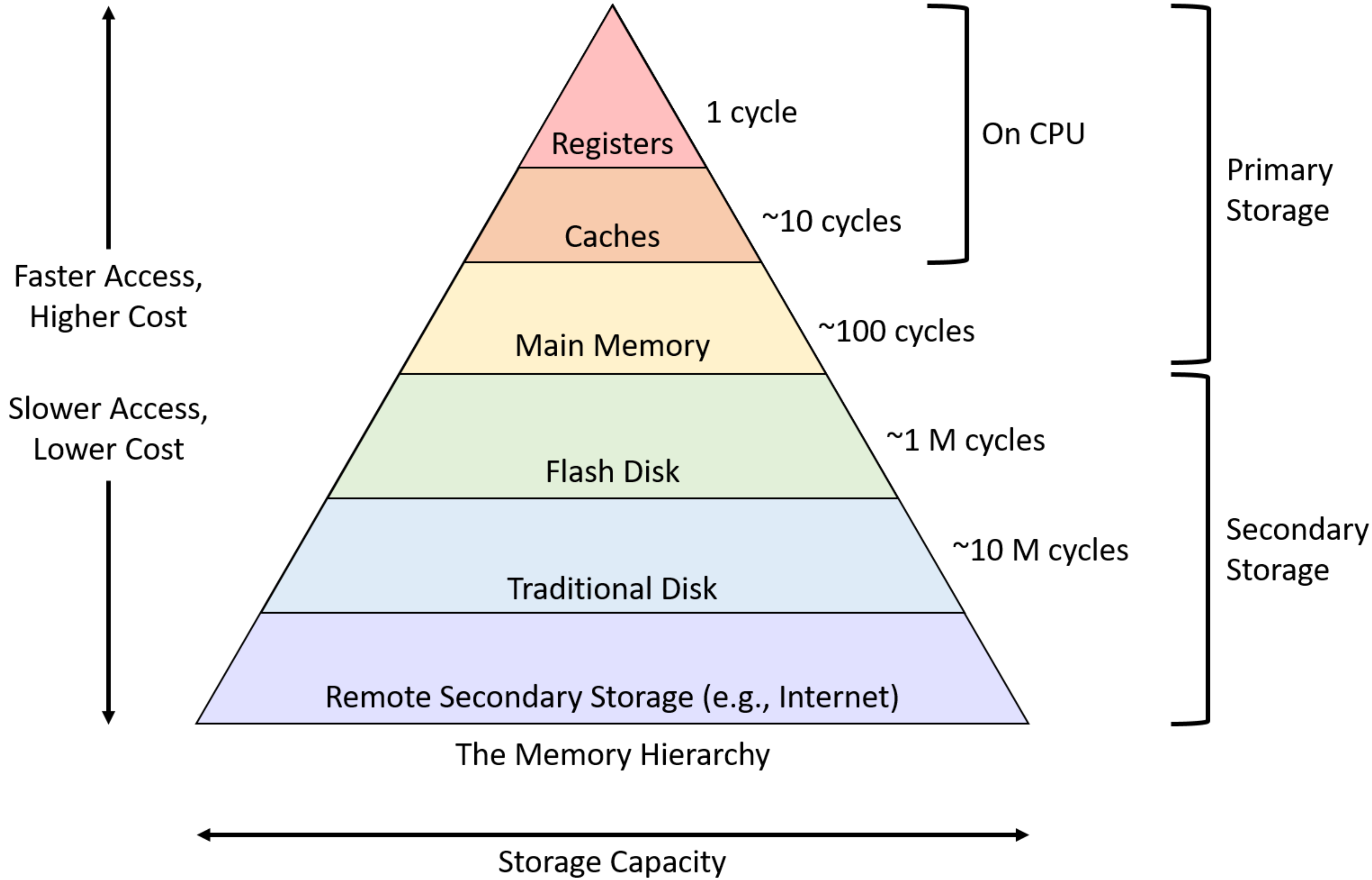
Query optimization goal: find plan that has lowest cost?

What is cost?

Disk I/O (Pages Read)
Memory Accesses
CPU Cycles
Comparisons
Records Processed

Memory Hierarchy





The Memory Hierarchy

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 * $1/5\times10^{-8}$ = 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 * $1/2.5\times10^{-4}$= 16 MB / sec

  If streaming sequentially, bandwidth 2+ GB/sec

Bandwidth v Latency (cont.)

(250x difference)

- Spinning disk: 10 ms latency vs 100 MB seq bandwidth
  - Random access BW per 4KB page = 400 KB/sec

(1Mx difference)

- Local network: 100 us latency vs 10 GB seq bandwidth
  - Random access BW per byte = 10K / sec

(100Mx difference)

- Wide area net: 10 ms latency vs 1 GB seq bandwidth
  - Random access BW per byte = 100 B / 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 memory 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

10s → 100m

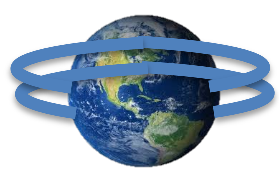
10 msec / access

Flash

10s → … → 10km

100 usec / access

Main Memory

10s →  … → 100,000 km

10 nsec/access

Database Cost Models

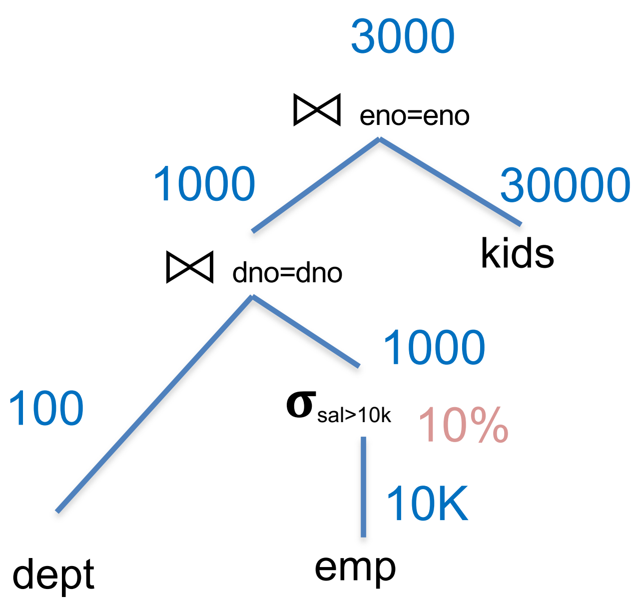
- Typically try to account for both CPU and I/O
  – I/O = ”input / output”, i.e., data access costs from disk
- Database algorithms try to optimize for sequential access (to avoid massive random access penalties)
- Simplified cost model for 6.5830:

  # 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

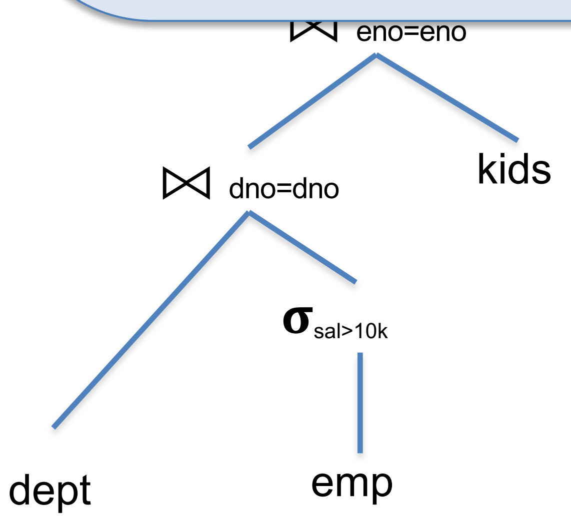
Assume nested loops joins, no indexes

WHAT IF.....

We use an index to random-seek to the 10% selection of emp?

Instead of 1 seek + 1MB/ 100MB/sec = 20ms,
it's 10 seeks for 10 pages (which is very lucky)?

10 seeks + 100k / 100MB/sec = 100ms + 1ms



1 scan of dept:
1 seek + 10KB / 100 MB/sec
10 ms + .1ms = 10.1 ms
1 scan of emp:
1 seek + 1 MB / 100 MB/sec
10 ms + 10 ms = 20 ms

100 x 20 ms + 10.1 ms = 2.1001 s

Example w/ Simple Cost Model

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

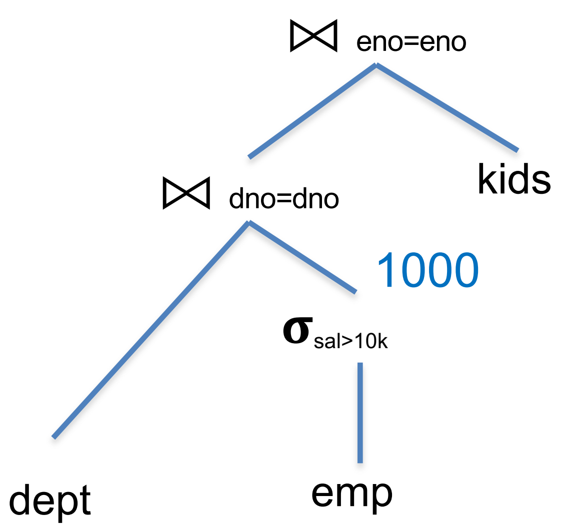
100 tuples/page
10 pages RAM
10 KB/page

|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

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

Let's take a break and try to do this individually



(Caching has huge benefit!)

mple Cost Model

x random I/O time +

quential disk B/W

0 KB

Actually…
remember we
have 10 pages
of RAM!

What's wrong
here?

Dept is inner in NL Join:

1 scan of emp

1K scans of dept (can we cache?)

Load dept (and 1k cached reads)

1 seek + 10KB / 100 MB/sec

10 ms + .1ms = 10.1 ms

1 scan of emp:

1 seek + 1 MB / 100 MB/sec

10 ms + 10 ms = 20 ms

20ms + 10.1 ms = 30.1 ms

(vs 2.1001s previously; ~70x faster!)

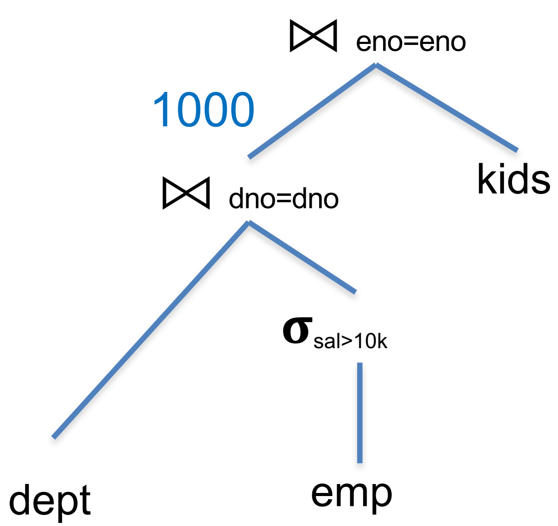
Example w/ Simple Cost Model

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

|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



2nd join – kids is inner

How much time does 2nd join take? Again, take a moment to do it out

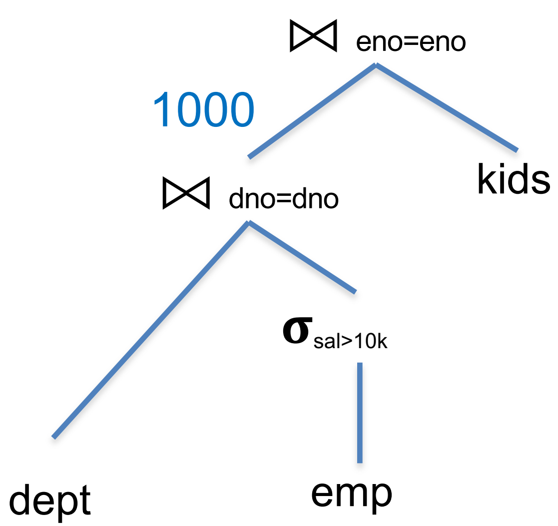
Example w/ Simple Cost Model

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

|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



2nd join – kids is inner
1000 scans x
1 seek + 3 MB / 100 MB / sec

1000 x (0.01 + 0.03) = 40 sec

Many query planners will not consider plans where "inner" (e.g., kids) is not a base relation – so called "left deep" plans

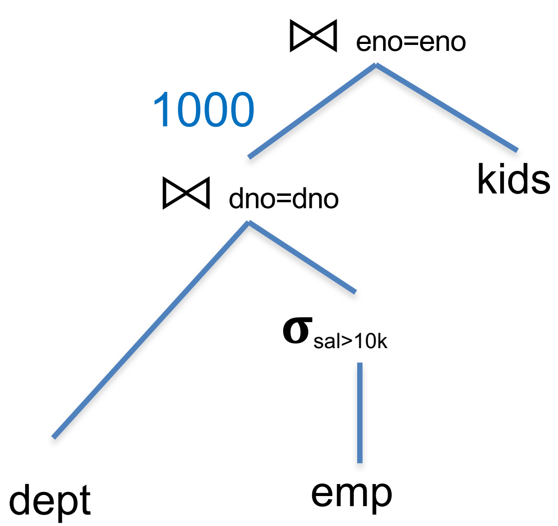
Example w/ Simple Cost Model

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

|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



What if **dept** were stored on a local network machine?

Local network: 100 us latency, 10 GB seq bandwidth
(assume data loading costs on remote machine are negligible)

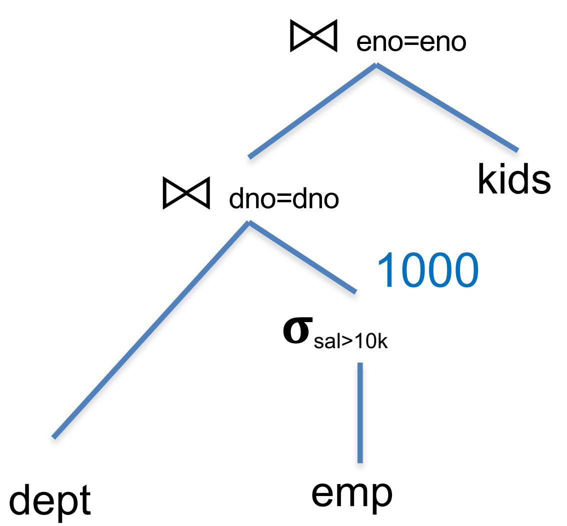
Example w/ Simple Cost Model

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

|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



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

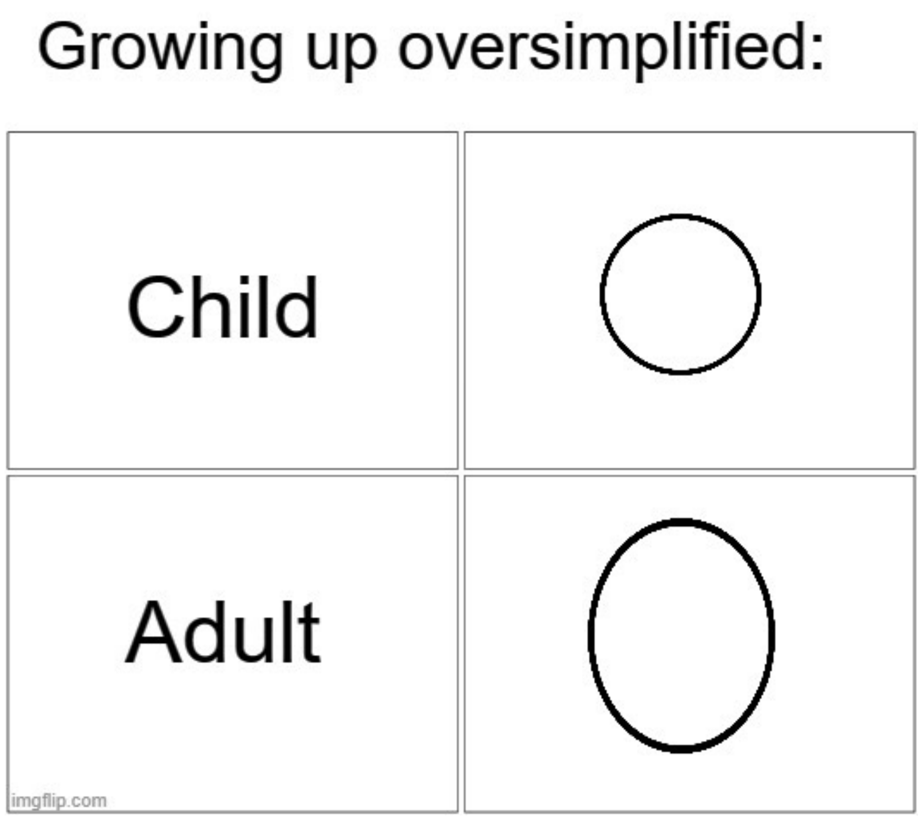
1 scan of emp
1K scans of dept (cached)

Load dept:
1 request + 10KB / 10 GB/sec
0.01 ms + .001ms = 0.011 ms

1 scan of emp:
1 seek + 1 MB / 100 MB/sec
10 ms + 10 ms = 20 ms

0.011 ms + 20 ms = 20.011 ms
(vs 30.1ms when dept is on disk)

Are we oversimplifying?



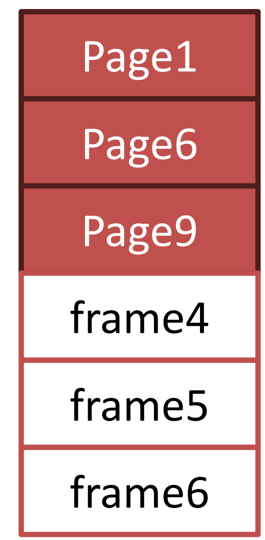
Buffer Pool

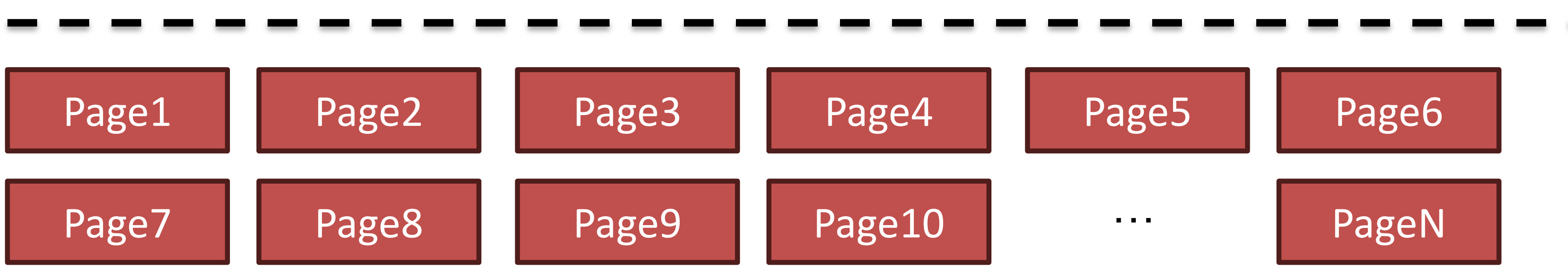
- **Buffer pool** is a cache for memory access. Caches pages of files / indices.
- When page is in buffer pool, don't need to read from disk
- Updates can also be cached
  - Discuss more w/ transactions

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).

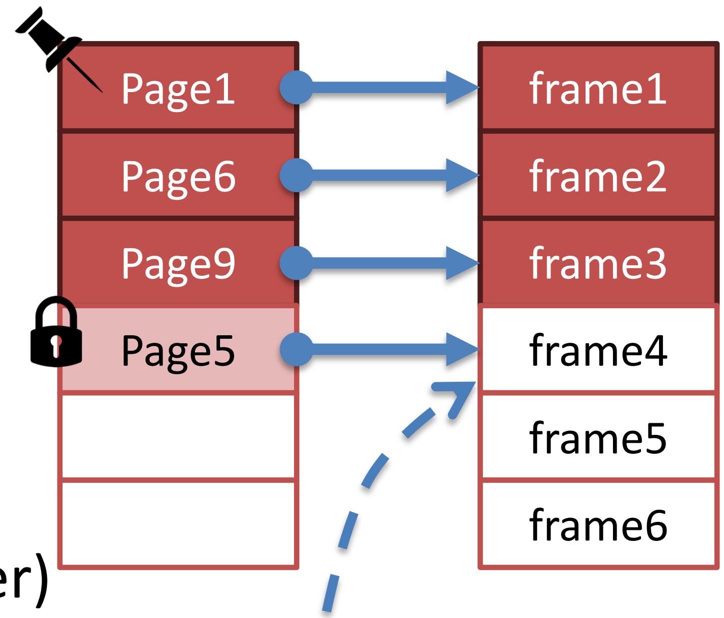


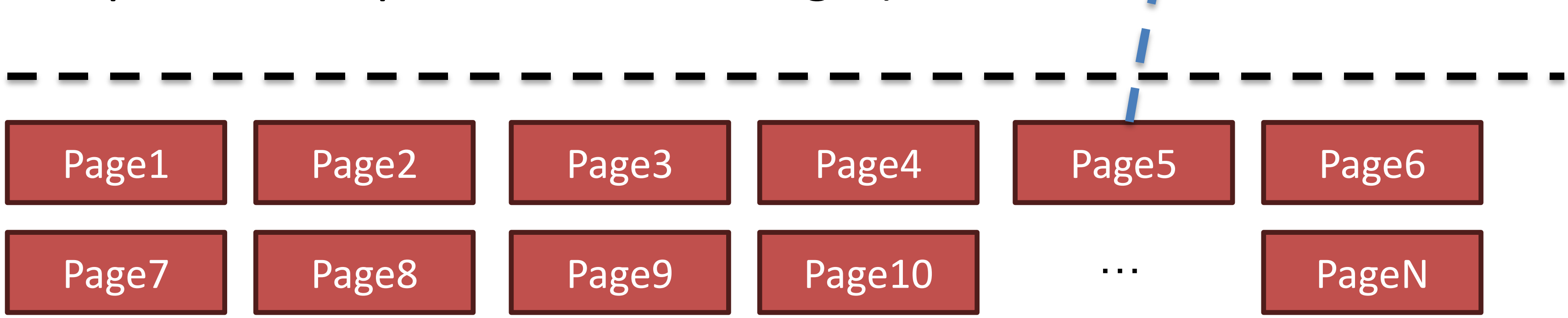


Buffer Pool

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

- Dirty Flag
- Pin/Reference Counter
- Latches
- In OpsDB also responsible for read/write locks (normally separate component lock manager)





LOCKS VS. LATCHES

- Locks:
  - Protects the database's logical contents from other transactions.
  - Held for transaction duration
  - Need to be able to rollback changes.

- Latches (Mutex)
  - Protects the critical sections of internal data structure from other threads.
  - Held for operation duration.
  - Do not need to be able to rollback changes

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.

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

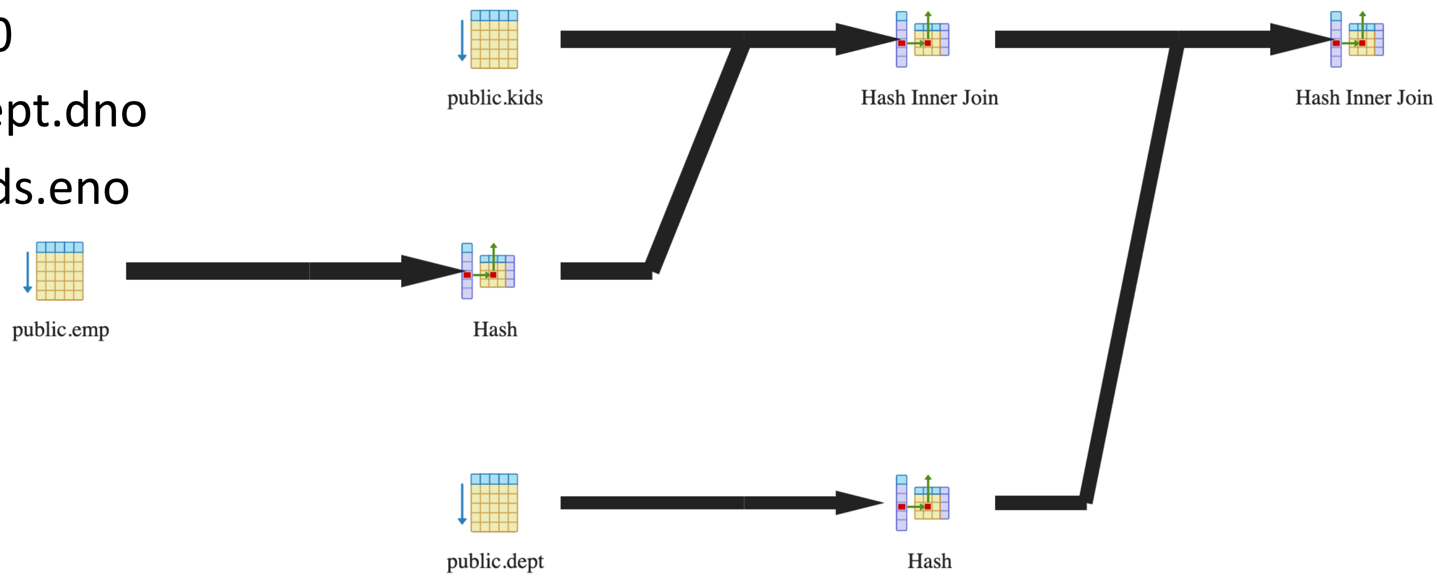
Postgres Plans

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)
```

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

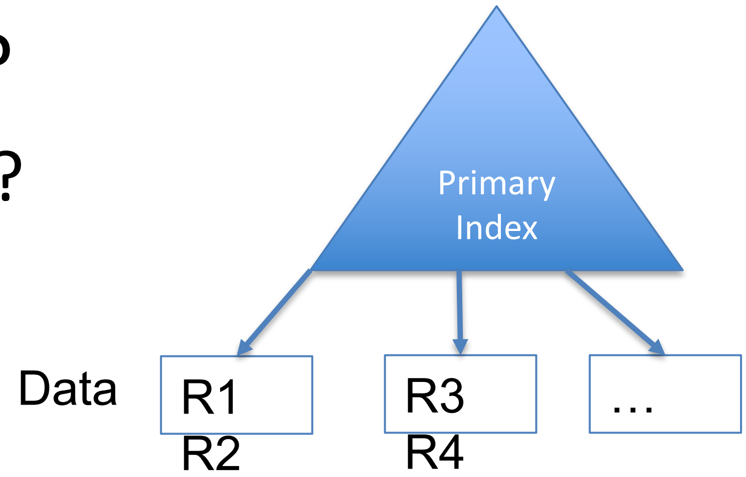
- (Time to scan students is .011 s, so negligible)

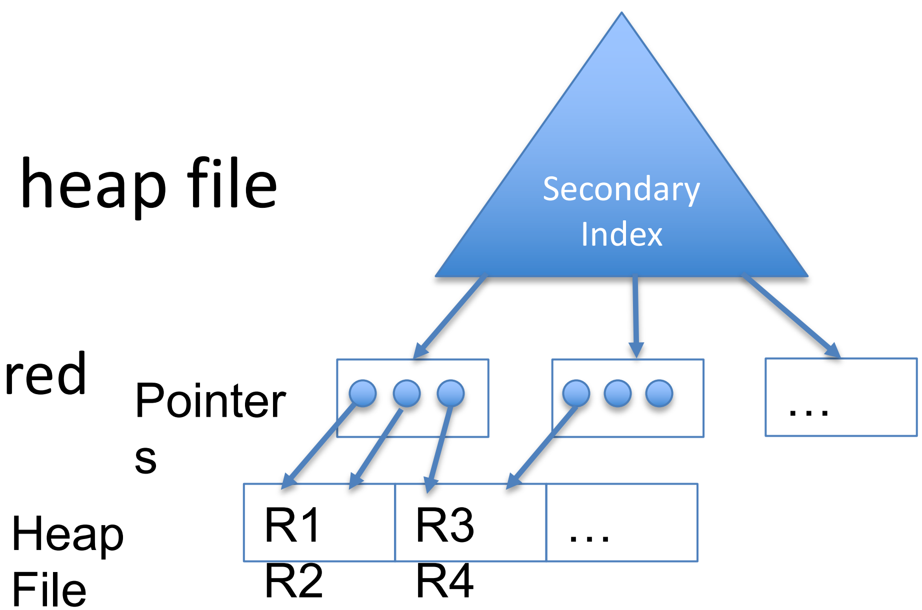
Today: Access Methods

- 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

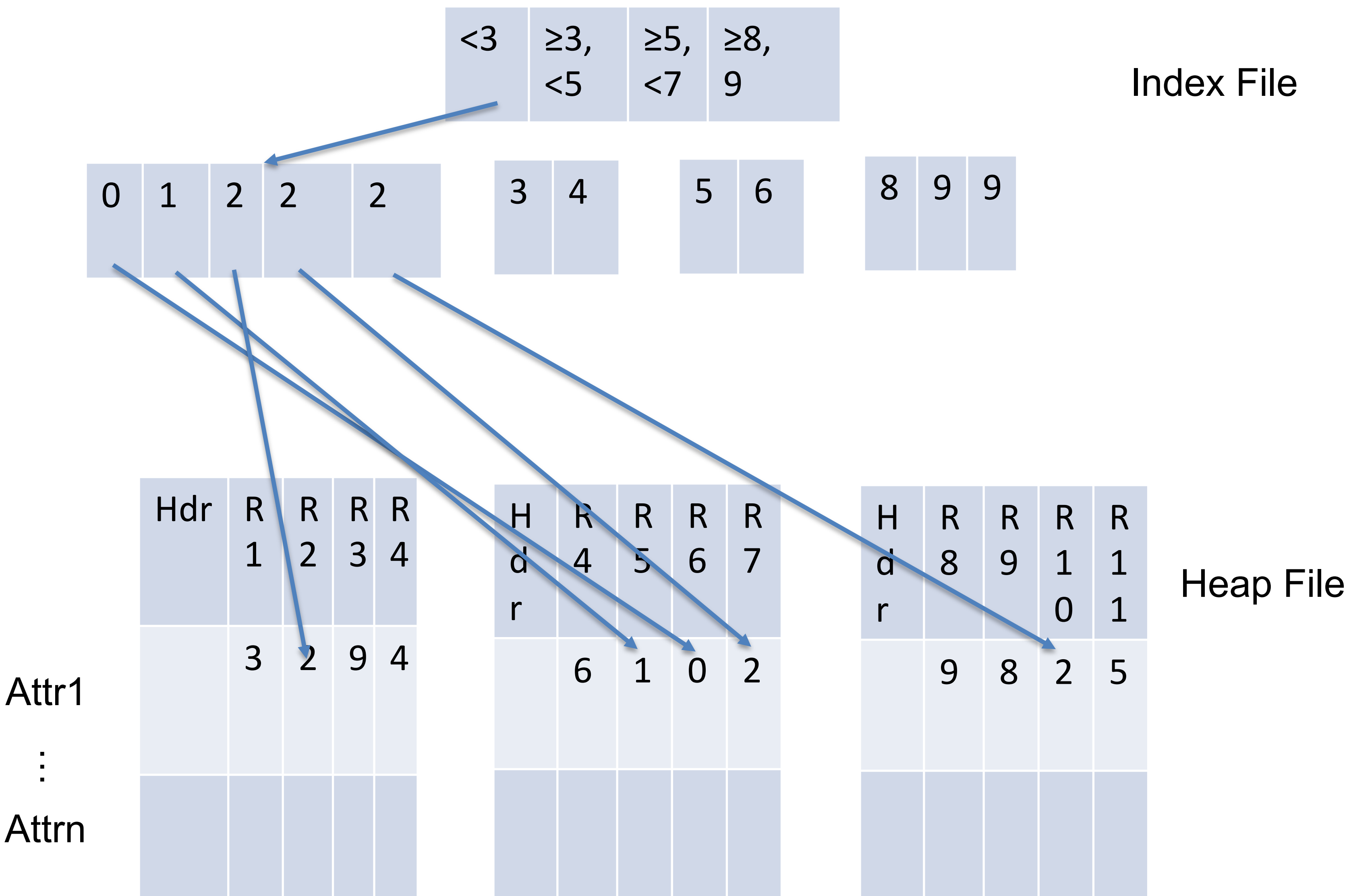
- What attributes to index?
  - Why not index everything?
- Index structure:
  - Leaves as data
    - Only one index?
    - "Primary Index"
  - Leaves as pointers to heap file
    - "Secondary Index"
    - Clustered vs unclustered





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

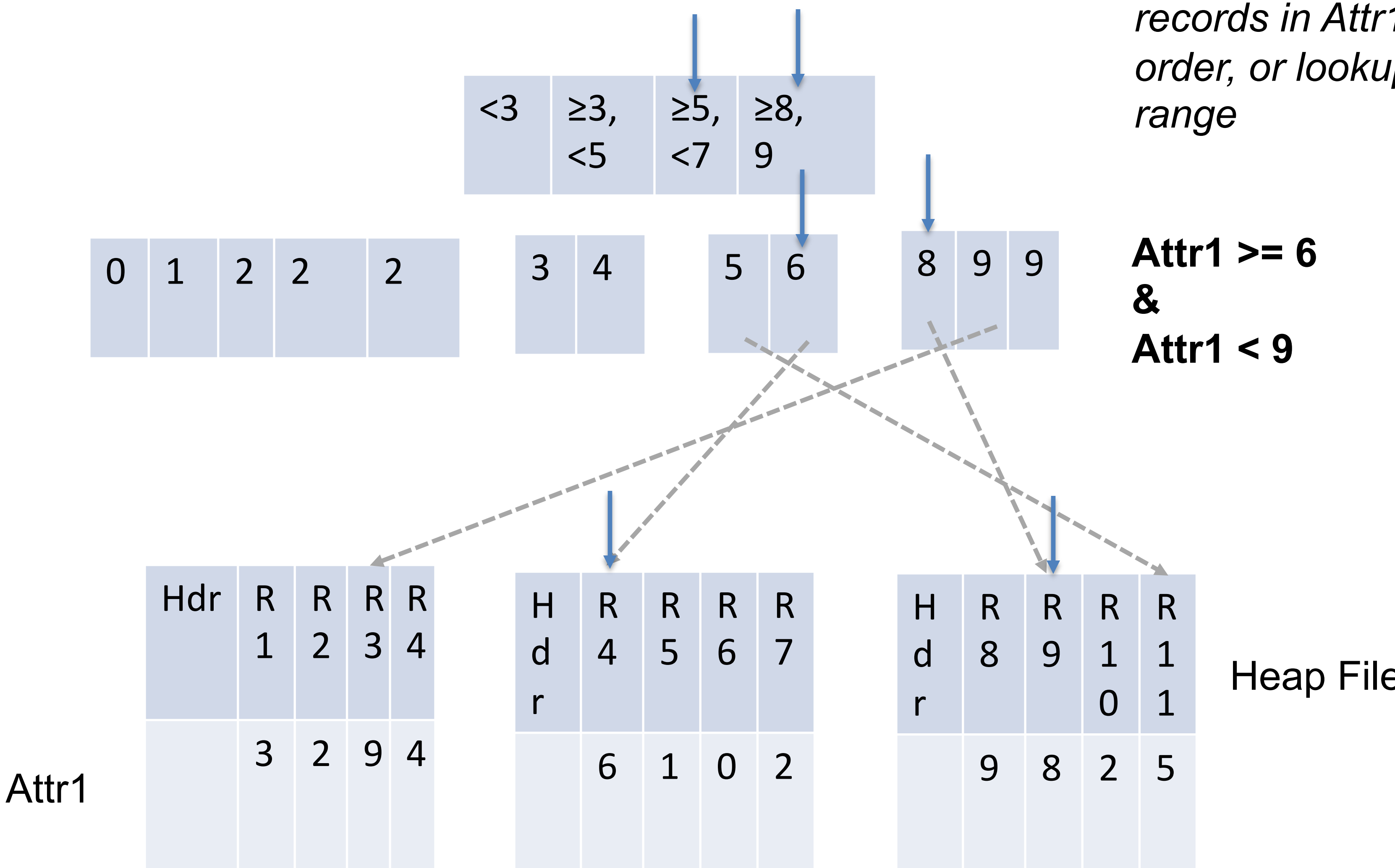
Tree Index



<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

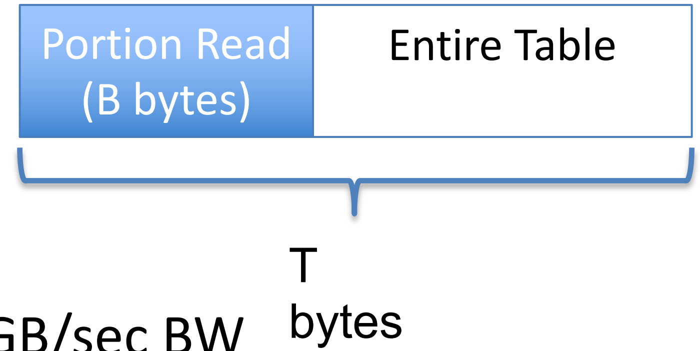
Traverse the records in Attr1 order, or lookup a range



Attr1 >= 6 & Attr1 < 9

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

Costs of Random Access



- 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?

$100\times10^{-6} * B / 100 + B/1\times10^{9} > 10\times10^{9} / 1\times10^{9}$

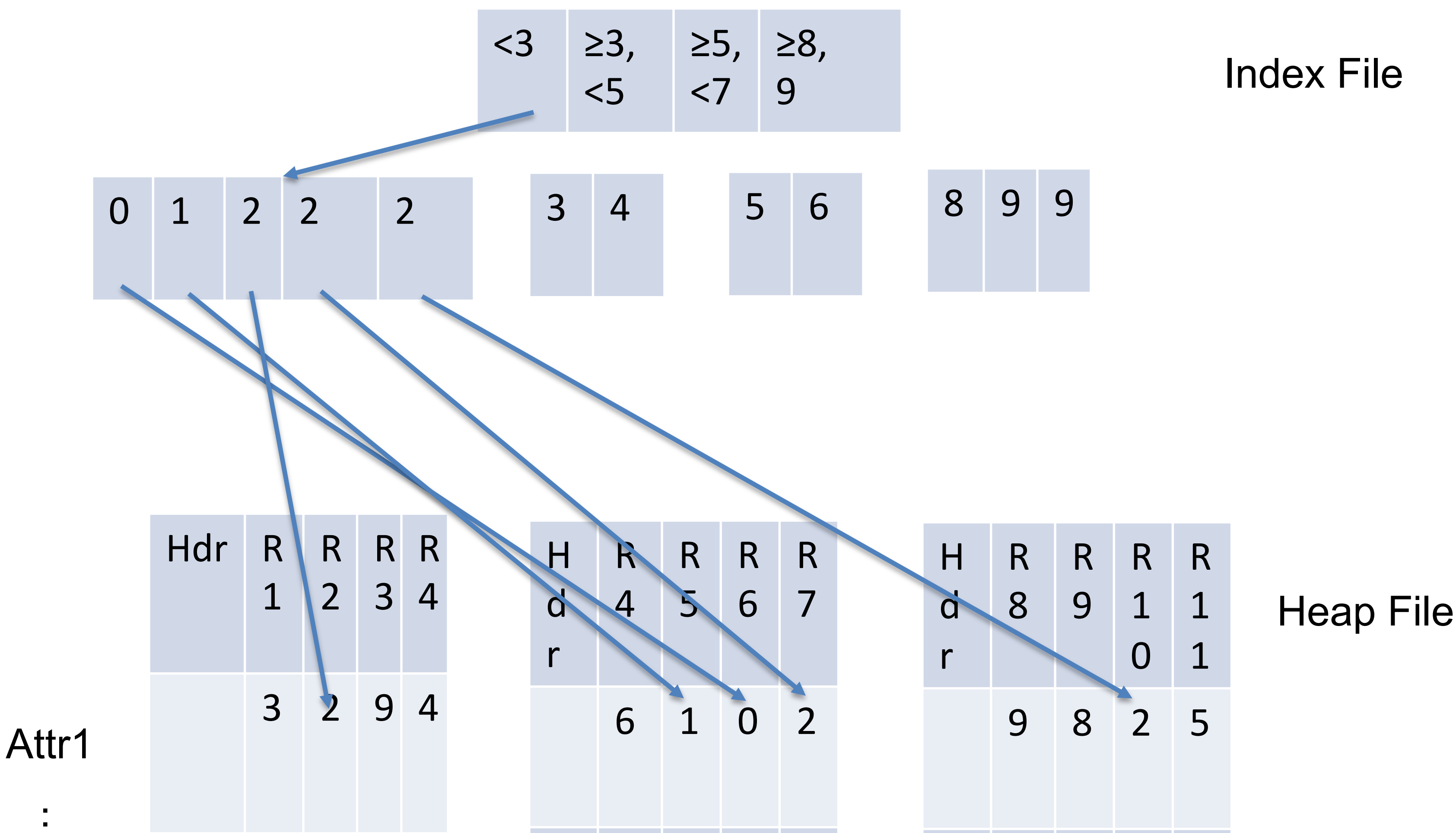
$1\times10^{-6}B + 1\times10^{-9}B > 10$

$B > 9.99\times10^{6}$

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

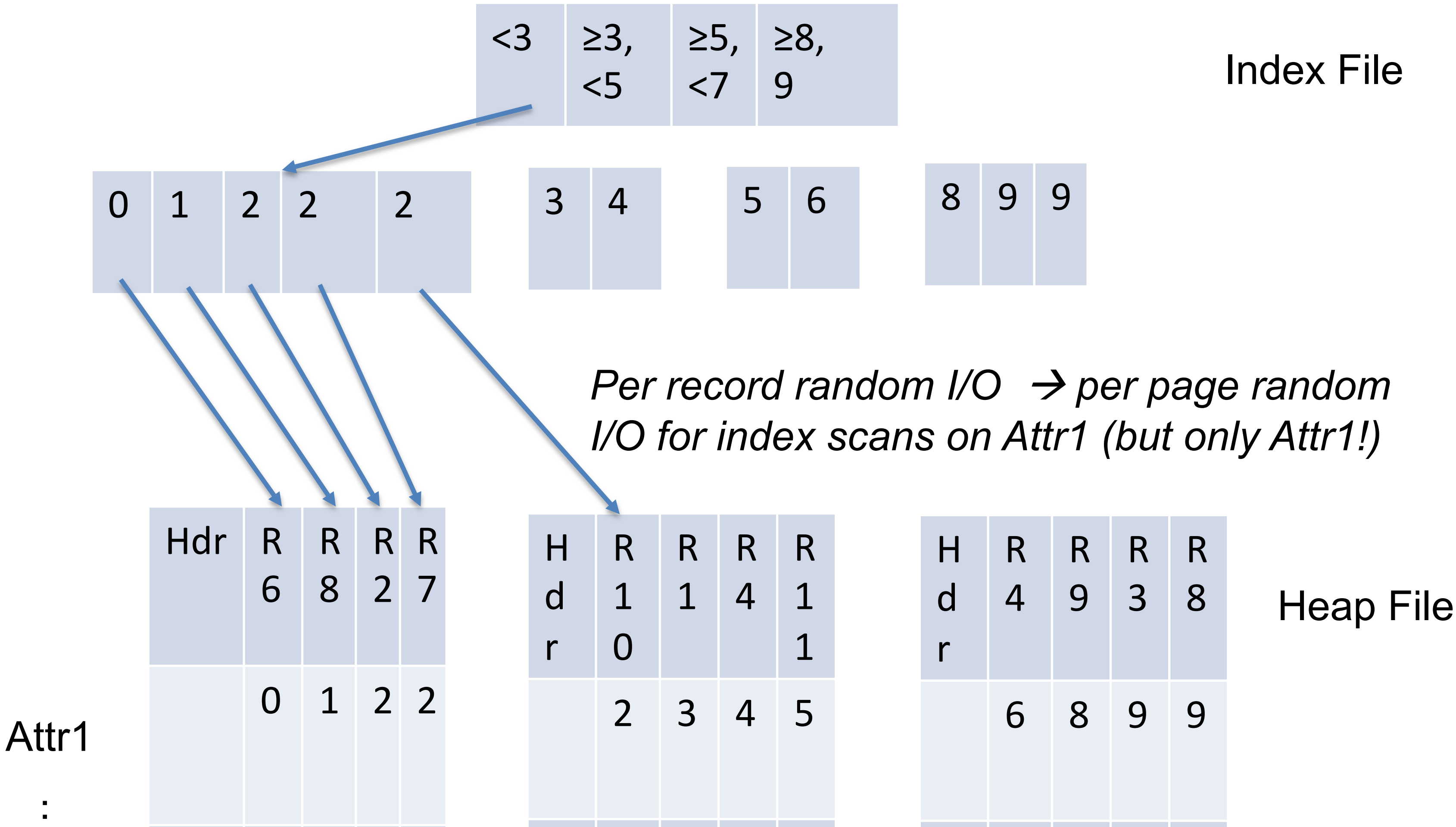
Clustered Index

- 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?

$100\times10^{-6} * B / \mathbf{1\times10^{6}} + B/1\times10^{9} > 10\times10^{9} / 1\times10^{9}$
$1\times10^{-12}B + 1\times10^{-9}B > 10$
$B > 9.99\times10^{9}$

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