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
Client

hf: HeapFile

bp: BufferPool

pageCache
Example
Example
Example

Client

Project(hf)

hf: HeapFile

bp: BufferPool

pageCache

Client

Iterator()

f()

hf: HeapFile

readPage()

bp: BufferPool

getPage()

p

&p

f2()

& Iterator()

f3()
Deleting Records and Rids

• Consider a query like:
  ```sql
  DELETE FROM x WHERE f > 10
  ```
  This is translated into a plan like

  ![Diagram](image)

  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

// 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 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
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

Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Optimizer (Lec 9)

Executor

Lec 7 – Join Algos

Storage System

Access Methods

Buffer Manager

Lock Manager

Log Manager

Today

+Lec 5

Today

Lec 6

Lec 5

Lec 6

Lec 9
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

\[
\begin{align*}
\text{it1} &= \text{Scan.open}(\text{"movieStar"}, \ldots) \\
\text{it2} &= \text{Filter.open}(\text{it1}, \text{bday}=x, \ldots) \\
\text{it3} &= \text{Scan.open}(\text{"starsIn"}, \ldots) \\
\text{it4} &= \text{Join.open}(\text{it2}, \text{it3}, \\
&\hspace{1cm}\text{starName}=\text{name}) \\
\text{it5} &= \text{Proj.open}(\text{it4}, \text{movieTitle})
\end{align*}
\]
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?

Order?

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

\[
\prod_{ename, \text{count}} \\
\sigma_{\text{count} > 7} \\
\alpha_{\text{agg:count(\*)}, \text{group by ename}} \\
\sigma_{\text{name='eecs'}} \\
\sigma_{\text{sal>50k}} \\
\]

dno=dno

dno=dno

dno=dno

ename,count

count > 7

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename

ename
Memory Hierarchy

Core1
L1 Cache
L2 Cache
Core2
L1 Cache
L3 Cache

Core1
L1 Cache
L2 Cache
Core2
L1 Cache

System Memory

Memory Bus

SSD (Flash)
Disk
32 KB
256 KB
8 MB
64 GB
4 TB

Time:
4 cycles
12 cycles
36 cycles
50-100ns (~ 150-300 cycles)
The Memory Hierarchy

- **Registers**: 1 cycle, On CPU
- **Caches**: ~10 cycles
- **Main Memory**: ~100 cycles
- **Flash Disk**: ~1 M cycles
- **Traditional Disk**: ~10 M cycles
- **Remote Secondary Storage (e.g., Internet)**

Storage Capacity

- Faster Access, Higher Cost
- Slower Access, Lower Cost
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
Bandwidth v Latency (cont.)

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

  (250x difference)

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

  (1Mx difference)

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

  (100Mx difference)
## Important Numbers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Cycles / Sec</strong></td>
<td>2+ Billion (.5 nsec latency)</td>
</tr>
<tr>
<td><strong>L1 latency</strong></td>
<td>2 nsec (4 cycles)</td>
</tr>
<tr>
<td><strong>L2 latency</strong></td>
<td>6 nsec (12 cycles)</td>
</tr>
<tr>
<td><strong>L3 latency</strong></td>
<td>18 nsec (36 cycles)</td>
</tr>
<tr>
<td><strong>Main memory latency</strong></td>
<td>50 – 100 ns (150-300 cycles)</td>
</tr>
<tr>
<td><strong>Sequential Mem Bandwidth</strong></td>
<td>20-40+ GB/sec</td>
</tr>
<tr>
<td><strong>SSD Latency</strong></td>
<td>250+ usec</td>
</tr>
<tr>
<td><strong>SSD Seq Bandwidth</strong></td>
<td>2-4 + GB/sec</td>
</tr>
<tr>
<td><strong>HD (spinning disk) latency</strong></td>
<td>10 msec</td>
</tr>
<tr>
<td><strong>HD Seq Bandwidth</strong></td>
<td>100+ MB</td>
</tr>
<tr>
<td><strong>Local Net Latency</strong></td>
<td>10 – 100 usec</td>
</tr>
<tr>
<td><strong>Local Net Bandwidth</strong></td>
<td>1 – 40 Gbit /sec</td>
</tr>
<tr>
<td><strong>Wide Area Net Latency</strong></td>
<td>10 – 100 msec</td>
</tr>
<tr>
<td><strong>Wide Area Net Bandwidth</strong></td>
<td>100 – 1 Gbit / sec</td>
</tr>
</tbody>
</table>
Speed Analogy

Disk
- 10s
- 100m
- 10 msec / access

Flash
- 10s
- ... to 10km
- 100 usec / access

Main Memory
- 10s
- ... to 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

Ideptl = 100 records = 1 page = 10 KB
Iempl = 10K = 100 pages = 1 MB
Ikidsl = 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

Dept is inner in NL Join:

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

(Caching has huge benefit!)
Example with Simple Cost Model

- 100 tuples/page
- 10 pages RAM
- 10 KB/page

<table>
<thead>
<tr>
<th>dept</th>
<th>100 records = 1 page = 10 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp</td>
<td>10K = 100 pages = 1 MB</td>
</tr>
<tr>
<td>kids</td>
<td>30K = 300 pages = 3 MB</td>
</tr>
</tbody>
</table>

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

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!)

Actually...
remember we have 10 pages of RAM!

What’s wrong here?
Example w/ Simple Cost Model

# seeks (random disk I/Os) \times \text{random I/O time} + \frac{\text{sequential bytes read}}{\text{sequential disk B/W}}

- 100 tuples/page
- 10 pages RAM
- 10 KB/page

|dept| = 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

\[ \text{2nd join} \quad \text{– kids is inner} \]

How much time does \text{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

\[ \text{2nd join – kids is inner} \]

| 1000 scans x  
| 1 seek + 3 MB / 100 MB / sec  

\[ 1000 \times (0.01 + 0.03) = 40 \text{ sec} \]

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  \( l_{\text{dept}} = 100 \) records = 1 page = 10 KB
10 pages RAM  \( l_{\text{empl}} = 10K = 100 \) pages = 1 MB
10 KB/page  \( l_{\text{kids}} = 30K = 300 \) pages = 3 MB

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

Dept is inner in NL Join:
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?

Growing up oversimplified:

- Child
- Adult
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).
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

<table>
<thead>
<tr>
<th>RAM Page</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Access</th>
<th>1 (a)</th>
<th>2 (b)</th>
<th>3 (c)</th>
<th>4 (a)</th>
<th>5 (b)</th>
<th>6 (c)</th>
<th>7 (a)</th>
<th>8 (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM Page</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>A - hit</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>B - hit</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>C - hit</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

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.
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)

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

Students Inner (Preferred)
• 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

Grades Inner
• 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 \(\leftarrow\) next time

• Many alternatives: e.g., R-trees \(\leftarrow\) 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
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 $S = \frac{T}{1\text{GB/sec}}$
- Rand access via index time $= 100 \text{usec} \times \frac{B}{R} + \frac{B}{1\text{GB/sec}}$
- Suppose R is 100 bytes, T is 10 GB

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

$100 \times 10^{-6} \times \frac{B}{100} + \frac{B}{1 \times 10^9} > \frac{10 \times 10^9}{1 \times 10^9}$

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

$B > 9.99 \times 10^6$

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

<table>
<thead>
<tr>
<th>Hdr</th>
<th>R 1</th>
<th>R 2</th>
<th>R 3</th>
<th>R 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attr1</th>
<th>&lt;3</th>
<th>≥3, &lt;5</th>
<th>≥5, &lt;7</th>
<th>≥8, 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Index File

<table>
<thead>
<tr>
<th>Hdr</th>
<th>R 8</th>
<th>R 9</th>
<th>R 10</th>
<th>R 11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Heap File
Clustered Index

• Order pages on disk in index order

Per record random I/O $\rightarrow$ per page random I/O for index scans on Attr1 (but only Attr1!)
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 $S = \frac{T}{1\text{GB/sec}}$
- Clustered index access time $= 100\text{ usec} \times \frac{B}{P}R + \frac{B}{1\text{GB/sec}}$
- 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?

$$100 \times 10^{-6} \times \frac{B}{1 \times 10^6} + \frac{B}{1 \times 10^9} > 10 \times 10^9 / 1 \times 10^9$$
$$1 \times 10^{-12}B + 1 \times 10^{-9}B > 10$$
$$B > 9.99 \times 10^9$$

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