Buffer pool & cost estimation ctd. and indexing
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?
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
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)
• 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.
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) 21 seconds
(B) 23 seconds
(C) 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
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) 0.244 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

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 ← 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

Index File

Heap File

Attr1

Attrn
Index Scan

Traverse the records in Attr1 order, or lookup a range

Attr1 \( \geq 6 \) \& 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 $S = \frac{T}{1\text{GB/sec}}$
- Rand access via index time = 100 usec * $\frac{B}{R} + \frac{B}{1\text{GB/sec}}$
- Suppose R is 100 bytes, T is 10 GB

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

(a) Scans larger than $\approx 1\text{MB}$ (0.01%)
(b) Scans larger than $\approx 10\text{MB}$ (0.1%)
(c) Scans larger than $\approx 100\text{MB}$ (1%)
(d) Scans larger than $\approx 1\text{GB}$ (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 $S = \frac{T}{1\,\text{GB/sec}}$
- 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
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!)

Attr1:

<table>
<thead>
<tr>
<th>Index File</th>
<th>Heap File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hdr</td>
<td>Hdr</td>
</tr>
<tr>
<td>R 6</td>
<td>R 4</td>
</tr>
<tr>
<td>R 8</td>
<td>R 9</td>
</tr>
<tr>
<td>R 2</td>
<td>R 3</td>
</tr>
<tr>
<td>R 7</td>
<td>R 8</td>
</tr>
<tr>
<td>0 1 2 2</td>
<td>6 8 9 9</td>
</tr>
<tr>
<td>3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>8 9 9</td>
<td></td>
</tr>
</tbody>
</table>
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 = $100\ \text{usec} \times \frac{B}{P} + \frac{B}{1\text{GB/sec}}$
- 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
Rest of Lecture

• Details of access methods
• Heap files (already seen)
• Hash indexes
• Trees (B+/R)
## Access Method Costs

<table>
<thead>
<tr>
<th></th>
<th>Heap File</th>
<th>Hash File</th>
<th>B+Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>O(P)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>O(P)</td>
<td>O(P)</td>
<td></td>
</tr>
<tr>
<td>Lookup</td>
<td>O(P)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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

On Disk Hash Table

n buckets, on n disk pages

Disk page 1 ...

Disk Page n

Issues

How big to make table?
If we get it wrong, either
waste space, or
end up with long overflow chains, or
have to rehash

\[ H(f1) \]

H(x) = x mod n

H(‘sam’, 10k, …)
H(‘mike’, 20k, …)
Extensible Hashing

- Create a family of hash tables parameterized by $k$
  \[ H_k(x) = H(x) \mod 2^k \]
- 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**

\( k=1 \)

<table>
<thead>
<tr>
<th>( H_k(x) )</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Hash Table**

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Directory

<table>
<thead>
<tr>
<th>$H_k(x)$</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Directory

\( k=1 \)

\[
\begin{array}{|c|c|}
\hline
H_k(x) & Page \\
\hline
0 & 0 \\
1 & 1 \\
\hline
\end{array}
\]

Hash Table

\[
\begin{array}{ccc}
\text{Page Number} & \text{Page Contents} \\
0 & 0 & 0 \\
1 & \emptyset & \emptyset \\
\end{array}
\]

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

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

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

Example

Directory

<table>
<thead>
<tr>
<th>$H_k(x)$</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

2 mod 2 = 0
Example

Directory

\[
\begin{array}{|c|c|}
\hline
H_k(x) & \text{Page} \\
\hline
0 & 0 \\
1 & 1 \\
\hline
\end{array}
\]

Hash Table

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Page Number} & \text{Page Contents} \\
\hline
0 & 0 & 0 & 2 \\
1 & 3 \\
\hline
\end{array}
\]

\[H_k(x) = x \mod 2^k\]

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

3 mod 2 = 1
Example

Directory

$H_k(x)$ | Page
---|---
0 | 0
1 | 1

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

2 mod 2 = 0
- FULL!
Example

Directory
\[ k = 1 \quad 2 \]

<table>
<thead>
<tr>
<th>( H_k(x) )</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>1</td>
<td>1 3</td>
</tr>
</tbody>
</table>

\[ H_k(x) = x \mod 2^k \]

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

Directory

\[ k = \pm 2 \]

<table>
<thead>
<tr>
<th>( H_k(x) )</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ H_k(x) = x \mod 2^k \]
H_k(x) = x mod 2^k

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

Example

Directory
k=1  2

<table>
<thead>
<tr>
<th>H_k(x)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash Table

Page Number | Page Contents |
------------|--------------|
0           | 0 0 0 2      |
1           | 3            |
2           |              |

Page Number | Page Contents |
------------|--------------|
0           | 0 0 0 2      |
1           | 3            |
2           |              |

Rehash

Only allocate 1 new page!
Example

Directory

\( k = 4 \)

<table>
<thead>
<tr>
<th>( H_k(x) )</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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

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

Directory

\[ k = 4 \]

\[ \begin{array}{|c|c|} 
\hline
H_k(x) & Page \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 1 \\
\hline
\end{array} \]

Hash Table

\[ \begin{array}{|c|c|c|} 
\hline
Page Number & Page Contents \\
\hline
0 & 0 & 0 \\
1 & 3 &  \text{ } \\
2 & 2 & 2 \\
\hline
\end{array} \]

\[ H_k(x) = x \mod 2^k \]

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

2 mod 4 = 2
## Access Method Costs

<table>
<thead>
<tr>
<th></th>
<th>Heap File</th>
<th>Hash File</th>
<th>B+Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>$O(P)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td><strong>R-Scan</strong></td>
<td>$O(P)$</td>
<td>- / $O(P)$</td>
<td></td>
</tr>
<tr>
<td>(sequential)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Lookup</strong></td>
<td>$O(P)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
</tbody>
</table>

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

Root node

Index on Attr A

<val11

ptr val21 ptr val22 ptr val23 ...

Inner nodes

>val21, <val22

ptr valn1 ptr valn2 ptr valn3 ...

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

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

Root node

Inner nodes

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

<table>
<thead>
<tr>
<th></th>
<th>Heap File</th>
<th>B+Tree</th>
<th>Hash File</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>O(1)</td>
<td>O( ( \log_B n ) )</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>O(P)</td>
<td>O( ( \log_B n ) )</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>R-Scan</strong></td>
<td>O(P)</td>
<td>O( ( \log_B n + R ) )</td>
<td>-- / O(P)</td>
</tr>
<tr>
<td><strong>Lookup</strong></td>
<td>O(P)</td>
<td>O( ( \log_B n ) )</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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

```sql
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
```
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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

- 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: $1 \leq X \leq 5$, $4 < Y < 5$

B+Tree on X,Y

Query for Y in some range
R-Trees / Spatial Indexes
R-Trees / Spatial Indexes
R-Trees / Spatial Indexes
Allows lookups on any sized region of X or Y
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

“Extract, Transform, Load”