6.5830 Lecture 7

Florent Willems, “The Accountant”

September 28, 2022
Cost Estimation and Indexing
Plan for Next Few Lectures

Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Executor

Storage System

This Lecture

Access Methods

Buffer Manager

Lock Manager

Log Manager

Lec 8 – Join Algos

Lec 10
Optimizer

Lec 9 – Column Stores
Recap: Bandwidth vs Latency

• 1\textsuperscript{st} access latency often high relative to the rate device can stream data sequentially (bandwidth)

• RAM: 50 ns per 16 B cache line
  → random access bandwidth of $16 \times \frac{1}{5 \times 10^{-8}} = 320$ MB / sec
  If streaming sequentially, bandwidth 20-40 GB/sec

• Flash disk: 250 us per 4K page
  → Random access bandwidth of $4K \times \frac{1}{2.5 \times 10^{-4}} = 16$ MB / sec
  If streaming sequentially, bandwidth 2+ GB/sec
**Important Numbers**

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

Disk
- 10s to 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
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
|empl| = 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 outer in NL Join of dept/emp join:
  1 scan of dept
  100 scans of emp (cannot cache)

Dept is inner in NL Join:
  1 scan of emp
  10k scans of dept?
    No, because it can be cached
Study Break

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

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

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

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

grades (cid int, g_sid int, grade char(2))
• 8 bytes (cid) + 8 bytes (g_sid) + 2 bytes (grade) + 4 bytes (header) = 22 bytes
• 22 x 1M = 22 MB / 100 MB/sec = .22 sec + 10ms seek
⇒ .23 sec
NL Join Grades and Students

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

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

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

• 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

Note random access! – this is an “unclustered” index

Traverse the records in Attr1 order, or lookup a range

Attr1 >= 6 & Attr1 < 9

Heap File

Attr1

0 1 2 2 2

3 4

5 6

8 9 9

Animals
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

#### Index File

<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>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Heap File

<table>
<thead>
<tr>
<th>Attr1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hdr</td>
<td>R6</td>
<td>R8</td>
<td>R2</td>
<td>R7</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

*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 usec * \( \frac{B}{PR} \) + \( \frac{B}{1\text{GB/sec}} \)
- 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?

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

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 (this lecture)
• Trees (B+/R) (next lecture)
# 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>sequential</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

(e.g., $H(x) = x \mod n$)

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
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>

\( H_k(x) = x \text{ mod } 2^k \)

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

Directory

\[ H_k(x) \mid Page \]

\begin{array}{|c|c|}
\hline
0 & 0 \\
1 & 1 \\
\hline
\end{array}

Hash Table

\[ \text{Page Number} \]

\[ \begin{array}{c}
0 \\
1 \\
\end{array} \]

\[ \text{Page Contents} \]

\[ \begin{array}{c}
0 \\
0 \\
\end{array} \]

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

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

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

\( 0 \mod 2 = 0 \)

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

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

Directory

\[ H_k(x) \quad \text{Page} \]

\[
\begin{array}{|c|c|}
\hline
0 & 0 \\
1 & 1 \\
\hline
\end{array}
\]

\[ k=1 \]

Hash Table

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

\[ 2 \mod 2 = 0 \]

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

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

Page Number

Page Contents

\[
\begin{array}{|ccc|}
\hline
0 & 0 & 0 & 2 \\
1 & 3 &   &   \\
\hline
\end{array}
\]

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

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

3 mod 2 = 1
H_k(x) = x mod 2^k

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

<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>

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

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

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

\[ H_k(x) = x \mod 2^k \]
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>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>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Only allocate 1 new page!
Example

Directory

\( k = \log_2 4 = 2 \)

<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

<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

\[ k = 1 \]

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>

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

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

Hash Table

Page Number

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

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>Insert</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>$O(P)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>$O(P)$</td>
<td><em>sequential</em> $O(P)$</td>
<td></td>
</tr>
<tr>
<td>Lookup</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

```
ptr  val11  ptr  val12  ptr  val13  ...
```

- `<val11`
- `>val21, <val22`
- `<valn1`

**Inner nodes**

```
ptr  val21  ptr  val22  ptr  val23  ...
```

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

```
RIDn  RIDn+1  RIDn+2  ptr  RIDn+3  RIDn+4  RIDn+5  ptr
```

**RID:** Record ID $\rightarrow$ 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
• \( \log_B(\text{tuples}) \) 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>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
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!

B+Tree on X,Y

Query: $1 \leq X \leq 5$, $4 < Y < 5$
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
• 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”