Administrivia

- HW5 is due **Wed Oct 26th**.
- Mid-term on **Wed Oct 19th**
- Covers up to *External Sorting* (inclusive).
  - Closed book, one sheet of notes (double-sided)
  - Please email Christos + Andy if you need special accommodations.
- [http://cmudb.io/f16-midterm](http://cmudb.io/f16-midterm)

Office Hours

- **Christos:**
  - Tuesday Oct 18th @ 1:00pm-3:00pm
- **Andy:**
  - Monday Oct 17th @ 10:30am-11:30am
  - Monday Oct 17th @ 1:00pm-2:00pm

Last Class

- **Sorting:**
  - External Merge Sort
- **Projection:**
  - External Merge Sort
  - Two-Phase Hashing

These are for when the data is larger than the amount of memory available.
Query Processing

- Some database operations are **expensive**.
- The DBMS can greatly improve performance by being “smart”
  – e.g., can speed up 1,000,000x over naïve approach

Query Processing

- There are clever implementation techniques for operators.
- We can exploit “equivalencies” of relational operators to do less work.
- Use statistics and cost models to choose among these.

*Work smarter, not harder.*

Today’s Class

- Explain/Analyze
- Selection
- Joins
Sample Database

SAILORS

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Kayne</td>
<td>999</td>
<td>45.0</td>
</tr>
<tr>
<td>3</td>
<td>Obama</td>
<td>50</td>
<td>52.0</td>
</tr>
<tr>
<td>2</td>
<td>Tupac</td>
<td>32</td>
<td>26.0</td>
</tr>
<tr>
<td>6</td>
<td>Bieber</td>
<td>10</td>
<td>19.0</td>
</tr>
</tbody>
</table>

Each tuple is 50 bytes
80 tuples per page
500 pages total
\[N = 500, p_S = 80\]

RESERVES

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>103</td>
<td>2014-02-01</td>
<td>matlock</td>
</tr>
<tr>
<td>1</td>
<td>102</td>
<td>2014-02-02</td>
<td>maggyver</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2014-02-01</td>
<td>a-team</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>2014-02-01</td>
<td>dallas</td>
</tr>
</tbody>
</table>

Each tuple is 40 bytes
100 tuples per page
1000 pages total
\[M = 1000, p_R = 100\]

EXPLAIN

- When you precede a `SELECT` statement with the keyword `EXPLAIN`, the DBMS displays information from the optimizer about the statement execution plan.
- The system “explains” how it would process the query, including how tables are joined and in which order.

Pseudo Query Plan:

```
SELECT bid, COUNT(*) AS cnt
FROM Reserves
GROUP BY bid
ORDER BY cnt
```

EXPLAIN SELECT bid, COUNT(*) AS cnt
FROM Reserves
GROUP BY bid
ORDER BY cnt

Postgres v9.1
**EXPLAIN**

```sql
EXPLAIN SELECT bid, COUNT(*) AS cnt
  FROM Reserves
GROUP BY bid
ORDER BY cnt
```

MySQL v5.5

**EXPLAIN ANALYZE**

- **ANALYZE** option causes the statement to be actually executed.
- The actual runtime statistics are displayed.
- This is useful for seeing whether the planner's estimates are close to reality.
- Note that **ANALYZE** is a Postgres idiom.

**EXPLAIN ANALYZE**

- Works on any type of query.
- Since **ANALYZE** actually executes a query, if you use it with a query that modifies the table, that modification will be made.

Postgres v9.1
Single-Table Selection

```
SELECT *
FROM Reserves AS R
WHERE R.rname < 'C%
```

σ\text{rname<'C%'} (Reserves)

- **What’s the best way to execute this query?**
  - A: It depends on…
    - What **indexes** and **access paths** are available.
    - What is the **expected size** of the result (in terms of number of tuples and/or number of pages)

Access Paths

- How the DBMS retrieves tuples from a table for a query plan.
  - **File Scan** (aka Sequential Scan)
  - **Index Scan** (B+Tree, Hash, BitMaps, …)
- Selectivity of an access path:
  - % of pages we retrieve
  - e.g., Selectivity of a hash index, on range query: 100% (no reduction!)
Simple Selections

• Size of result approximated as:
  – \((\text{size of } R) \cdot (\text{selectivity})\)
• Selectivity is also called **Reduction Factor**.
• The estimate of reduction factors is based on statistics – we will discuss shortly.

Selection Options

• No Index, Unsorted Data
• No Index, Sorted Data
• B+Tree Index
• Hash Index, Equality Selection

Selection: No Index, Unsorted Data

\[
\text{SELECT } * \\
\text{FROM Reserves AS R} \\
\text{WHERE R.rname < 'C%'}
\]

• Must scan the whole relation.
  – **Cost: \(M\)**
• For “Reserves” = 1000 I/Os.

Selection: No Index, Sorted Data

\[
\text{SELECT } * \\
\text{FROM Reserves AS R} \\
\text{WHERE R.rname < 'C%'}
\]

• Cost of binary search + number of pages containing results.
  – **Cost: \(\log_2 M + \lceil\text{selectivity} \cdot \#\text{pages}\rceil\)**
Selection: B+Tree Index

- With an index on selection attribute:
  - Use index to find qualifying data entries, then retrieve corresponding data records.

 SELECT *
 FROM Reserves AS R
 WHERE R.rname < ‘C%’

• Cost depends on #qualifying tuples, and clustering.
  - Finding qualifying data entries (typically small)
  - Plus cost of retrieving records (could be large w/o clustering).

In example “Reserves” relation, if 10% of tuples qualify (100 pages, 10,000 tuples):
  - With a *clustered* index, cost will be less than 100 I/Os;
  - If *unclustered*, could be up to 10,000 I/Os! unless…
Selection: B+Tree Index

- Refinement for unclustered indexes:
  - Find qualifying data records by their \textit{rid}.
  - Sort \textit{rid}'s of the data records to be retrieved.
  - Fetch \textit{rids} in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Partial Indexes

- Create an index on a \textit{subset} of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo
    ON foo (a, b)
    WHERE c = 'WuTang'

SELECT b FROM foo
    WHERE a = 123 AND c = 'WuTang'
```

Covering Indexes

- If all of the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

```
CREATE INDEX idx_foo
    ON foo (a, b)

SELECT b FROM foo
    WHERE a = 123
```

Index Include Columns

- Embed additional columns in indexes to support index-only queries.
- Not part of the search key.

```
CREATE INDEX idx_foo
    ON foo (a, b)
    INCLUDE (c)

SELECT b FROM foo
    WHERE a = 123 AND c = 'WuTang'
```
Selection Conditions

• A B+tree index matches terms that involve only attributes in a prefix of the search key.
  – Index on \( \langle a, b, c \rangle \) matches \((a=5 \text{ AND } b=3)\), but not \((b=3)\).
  – **Note:** Commercial systems can handle this (e.g., Oracle’s Index Skip Scan)
• For Hash index, we must have **all** attributes in search key.

B+Tree String Prefix Search

Key = ‘xy’
Key = ‘_y’

Two Approaches to Index Selection

• **Approach #1:** Find the cheapest access path, retrieve tuples using it, and apply any remaining terms that don’t match the index
• **Approach #2:** Use multiple indexes to find the intersection of matching tuples.

Approach #1

• Find the **cheapest access path**, retrieve tuples using it, and apply any remaining terms that don’t match the index:
  – Cheapest access path: An index or file scan with fewest I/Os.
  – Terms that **match** this index reduce the number of tuples retrieved; **other terms** help discard some retrieved tuples, but do not affect number of tuples/pages fetched.
Approach #1 – Example

\[(\text{day}<\textit{10/16/2016} \text{ AND } \text{bid}=5 \text{ AND } \text{sid}=3)\]

• A B+ tree index on \textit{day} can be used;
  – then, \text{bid}=5 and \text{sid}=3 must be checked for each retrieved tuple.
• Similarly, a hash index on \textit{<bid,sid>} could be used;
  – Then, \textit{day<’10/16/2016’} must be checked.

Approach #2

• If we have 2 or more matching indexes:
  – Get sets of \textit{rids} of data records using each matching index.
  – Then intersect these sets of \textit{rids}.
  – Retrieve the records and apply any remaining terms.

Approach #2 – Example

\[(\text{day}<\textit{10/16/2016} \text{ AND } \text{bid}=5 \text{ AND } \text{sid}=3)\]

• With an index on \textit{day} and an index on \textit{sid},
  – We can retrieve \textit{rids} of records satisfying \textit{day<’10/16/2016’} using the first,
  – \textit{rids} of recs satisfying \textit{sid}=3 using the second,
  – intersect,
  – retrieve records and check \text{bid}=5.
Summary

• For selections, we always want an index.
  – B+Trees are more versatile.
  – Hash indexes are faster, but only support equality predicates.
• Last resort is to just scan entire table.

Today's Class

• Explain/Analyze
  • Selection
  • Joins

Joins

• $R \bowtie S$ is very common and thus must be carefully optimized.
• $R \times S$ followed by a selection is inefficient because cross-product is large.
• There are many approaches to reduce join cost, but no one works best for all cases.
• Remember, join is associative and commutative.
Joins

• Assume:
  – \(M\) pages in \(R\), \(p_R\) tuples per page, \(m\) tuples total
  – \(N\) pages in \(S\), \(p_S\) tuples per page, \(n\) tuples total
  – In our examples, \(R\) is Reserves and \(S\) is Sailors.
• We will consider more complex join conditions later.
• Cost metric: # of I/Os

First Example

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

• Assume that we don’t know anything about the tables and we don’t have any indexes.

Simple Nested Loop Join

• Algorithm #0: Simple Nested Loop Join

\[
\begin{align*}
\text{foreach tuple } r \text{ of } R \\
\text{foreach tuple } s \text{ of } S \\
\text{output, if they match}
\end{align*}
\]

\[
\begin{array}{c}
R(A,..) \\
S(A, ......)
\end{array}
\]
Simple Nested Loop Join

- **Algorithm #0**: Why is it bad?
  - How many disk accesses (‘\(M\)’ and ‘\(N\)’ are the number of blocks for ‘\(R\)’ and ‘\(S\)’)?
    - **Cost**: \(M + (p_R \cdot M) \cdot N\)

Block Nested Loop Join

- **Algorithm #1**: Block Nested Loop Join

  ```
  foreach block from R
  foreach block from S
  output, if tuples match
  ```

- **Algorithm #1**: Things are better.
  - How many disk accesses (‘\(M\)’ and ‘\(N\)’ are the number of blocks for ‘\(R\)’ and ‘\(S\)’)?
    - **Cost**: \(M + (M \cdot N)\)

**Actual number:**
- \(M + ((p_R \cdot M) \cdot N) = 1000 + (100 \cdot 1000) \cdot 500\)
  - \(= 50,001,000\) I/Os
- At 10ms/IO, Total time ≈ 5.7 days
- What if smaller relation (S) was outer?
- What assumptions are being made here?

**SSD**: ≈ 1.3 hours at 0.1ms/IO
Block Nested Loop Join

• Actual number:
  – \( M + (M \cdot N) = 1000 + (1000 \cdot 500) = 501,000 \) I/Os
  – At 10ms/IO, Total time \( \approx 1.4 \) hours

• What if we use the smaller one as the outer relation?
  – The smallest in terms of # of pages.

Block Nested Loop Join

• Actual number:
  – \( N + (M \cdot N) = 500 + (1000 \cdot 500) = 500,500 \) I/Os
  – At 10ms/IO, Total time \( \approx 1.4 \) hours

• What if we have \( B \) buffers available?
  – Give \( B-2 \) buffers to outer relation, 1 to inner relation, 1 for output

Block Nested Loop Join

• Algorithm #1: Using multiple buffers.

  \[
  \text{foreach } \frac{B-2}{B} \text{ blocks from } R \quad \text{foreach block from } S \\
  \quad \text{output, if tuples match}
  \]

  \( M \) pages, \( m \) tuples

  \( N \) pages, \( n \) tuples

Block Nested Loop Join

• Algorithm #1: Using multiple buffers.
  • How many disk accesses (‘\( M \)’ and ‘\( N \)’ are the number of blocks for ‘\( R \)’ and ‘\( S \)’)?
    – Cost: \( M + \left( \left\lceil \frac{M}{B-2} \right\rceil \cdot N \right) \)

  \( M \) pages, \( m \) tuples

  \( N \) pages, \( n \) tuples
Block Nested Loop Join

- **Algorithm #1:** Using multiple buffers.
- But if the outer relation fits in memory:
  - Cost: $M+N$

\[ M \text{ pages, } m \text{ tuples} \]
\[ S(A, ...) \]
\[ N \text{ pages, } n \text{ tuples} \]

Faloutsos/Pavlo  
CMU SCS 15-415/615  
58

Block Nested Loop Join

- Actual number:
  - $M + N = 1000 + 500 = 1500$
  - At 10ms/IO, Total time $\approx 15$ seconds

SSD $\approx 0.15$ seconds at 0.1ms/IO

Faloutsos/Pavlo  
CMU SCS 15-415/615  
59

Joins

- Join techniques we will cover:
  - Nested Loop Joins
  - Index Nested Loop Joins
  - Sort-Merge Joins
  - Hash Joins

Faloutsos/Pavlo  
CMU SCS 15-415/615  
60

Index Nested Loop

- Why do basic nested loop joins suck?
  - *For each tuple in the outer table, we have to do a sequential scan to check for a match in the inner table.*
- A better approach is to use an index to find inner table matches.
  - We could use an existing index, or even build one on the fly.

Faloutsos/Pavlo  
CMU SCS 15-415/615  
61
Index Nested Loop Join

• **Algorithm #2:** Index Nested Loop Join

```latex
define foreach tuple r of R
  foreach tuple s of S, where r_i==s_j
  output
```

- **Index Probe**

- $M$ pages, $m$ tuples
- $S(A, ...)$

- $N$ pages, $n$ tuples
- $R(A, ...)$

- Look-up Cost

- Cost: $M + (m \cdot C)$

Nested Loop Joins Guideline

• Pick the smallest table as the outer relation
  – *i.e., the one with the fewest pages*
• Put as much of it in memory as possible
• Loop over the inner

Join techniques we will cover:

- Nested Loop Joins
- Index Nested Loop Joins
- Sort-Merge Joins
- Hash Joins
Sort-Merge Join

- First sort both tables on joining attribute.
- Then step through each one in lock-step to find matches.

This algorithm is useful if:
- One or both tables are already sorted on join attribute(s)
- Output is required to be sorted on join attributes
- The “Merge” phase can require some back tracking if duplicate values appear in join column.

Algorithm #3: Sort-Merge Join

How many disk accesses (‘M’ and ‘N’ are the number of blocks for ‘R’ and ‘S’)?

\[
\text{Cost: } (2M \cdot \log M / \log B) + (2N \cdot \log N / \log B) + M + N
\]
Sort-Merge Join Example

**SELECT** *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Kayne</td>
<td>999</td>
<td>45.0</td>
</tr>
<tr>
<td>3</td>
<td>Obama</td>
<td>50</td>
<td>52.0</td>
</tr>
<tr>
<td>2</td>
<td>Tupac</td>
<td>32</td>
<td>26.0</td>
</tr>
<tr>
<td>6</td>
<td>Bieber</td>
<td>10</td>
<td>19.0</td>
</tr>
</tbody>
</table>

**Sort!**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
<td>2014-02-02</td>
<td>macgyver</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>2014-02-01</td>
<td>dallas</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2014-02-02</td>
<td>a-team</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td>2014-02-01</td>
<td>matlock</td>
</tr>
</tbody>
</table>

**Merge!**

- With 100 buffer pages, both Reserves and Sailors can be sorted in 2 passes:
  - **Cost: 7,500 I/Os**
  - At 10ms/IO, Total time ≈ 75 seconds
- Block Nested Loop:
  - **Cost: 2,500 to 15,000 I/Os**

SSD ≈ 0.75 seconds at 0.1ms/IO
Sort-Merge Join

- Worst case for merging phase?
  - When all of the tuples in both relations contain the same value in the join attribute.
  - Cost: \( M \cdot N + \text{(sort cost)} \)

- Don’t worry kids! This is unlikely!

Sort-Merge Join Optimizations

- All the refinements from external sorting
- Plus overlapping of the merging of sorting with the merging of joining.
- Multi-threaded optimizations.

Joins

- Join techniques we will cover:
  - Nested Loop Joins
  - Index Nested Loop Joins
  - Sort-Merge Joins
  - Hash Joins

In-Memory Hash Join

- Algorithm #4: In-Memory
  - Build hash table \( H \) for \( R \)
  - For each tuple \( s \) of \( S \)
  - Output, if \( h(s_j) \in H \)

This assumes \( H \) fits in memory!
Grace Hash Join

• Hash join when tables don’t fit in memory.
  – **Partition Phase:** Hash both tables on the join attribute into partitions.
  – **Probing Phase:** Compares tuples in corresponding partitions for each table.
• Named after the GRACE database machine.

Grace Hash Join

• Hash R into (0, 1, ..., ‘max’) buckets
• Hash S into buckets (same hash function)

Grace Hash Join

• Join each pair of matching buckets:

Grace Hash Join

• Choose the (page-wise) smallest - if it fits in memory, do a **nested loop join**
• Otherwise use **recursive partitioning**:
  – Build another hash table for $H_{S(j)}$, and probe it with each tuple of $H_{R(i)}$ (with $H_2 \neq H_1$)
  – And then probe it for each tuple of the other
  – More details available in textbook (Ch 14.4.3)
Grace Hash Join

- Cost of hash join?
  - Assume that we have enough buffers.
  - **Cost:** $3(M + N)$
- **Partitioning Phase:** read+write both tables
  - $2(M+N)$ I/Os
- **Probing Phase:** read both tables
  - $M+N$ I/Os

Faloutsos/Pavlo
CMU SCS 15-415/615
82

Grace Hash Join

- **Actual number:**
  - $3(M + N) = 3 \cdot (1000 + 500)$
  - At 10ms/IO, Total time $\approx 45$ seconds

Faloutsos/Pavlo
CMU SCS 15-415/615
83

Cloudera Impala

- % of CPU Time Spent in Query Operators
- Workload: TPC-H Benchmark

Faloutsos/Pavlo
CMU SCS 15-415/615
84

Sort-Merge vs. Hash

- Given a minimum amount of memory both have a cost of $3(M+N)$ I/Os.
- When do we want to choose one over the other?

Faloutsos/Pavlo
CMU SCS 15-415/615
85
Sort-Merge vs. Hash

• **Sort-Merge:**
  – Less sensitive to data skew.
  – Result is sorted (may help upstream operators).
  – Goes faster if one or both inputs already sorted.

• **Hash:**
  – Superior if relation sizes differ greatly.
  – Shown to be highly parallelizable.

Fastest Join Algorithm Title

• 1970s – Sorting
• 1980s – Hashing
• 1990s – Both
• 2000s – Hashing
• 2010s – ???

Summary

• There are multiple ways to do selections if you have different indexes.
• Joins are difficult to optimize.
  – Index Nested Loop when selectivity is small.
  – Sort-Merge/Hash when joining whole tables.

Next Class

• Set & Aggregate Operations
• Query Optimizations
• Mid-Term Review