Carnegie Mellon Univ.  
Dept. of Computer Science  
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo  
Lecture#14: Implementation of  
Relational Operations

Administrivia

• HW4 is due this Thursday.
• Reminder: Mid-term on Tues March 4th  
  – Will cover everything up to and including this  
    week’s lectures.  
  – Closed book, one sheet of notes (double-sided)  
  – We will release last year’s exam.

Extended Office Hours

• Christos:  
  – Wednesday Feb 26th 12:00pm-1:00pm  
  – Friday Feb 28th 3:00pm-5:00pm

• Andy:  
  – Friday Feb 28th 10:00am-12:00pm  
  – Monday Mar 3rd 9:00am-10:00am  
  – Tuesday Mar 4th 10:00am-12:00pm
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.

---

Cost-based Query Sub-System

```
Select *
From Blah B
Where B.blah = blah
```

Queries → Query Parser → Query Optimizer → Plan Generator → Plan Cost Estimator → Catalog Manager → Schema Statistics → Query Plan Evaluator

---

Cost-based Query Sub-System

```
Select *
From Blah B
Where B.blah = blah
```

Queries → Query Parser → Query Optimizer → Plan Generator → Plan Cost Estimator → Catalog Manager → Schema Statistics → Query Plan Evaluator

---
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

• Introduction
• Selection
• Joins
Lame Query Execution

- For each `SELECT - FROM - WHERE` query block
  - Do cartesian products first
  - Then do selections + extras:
    - `GROUP BY, HAVING`
    - Projections
    - `ORDER BY`
- Incredibly inefficient
  - Huge intermediate results!
  - Makes small children cry.

Query Optimizer

- "Optimizer" is a bit of a misnomer…
- Goal is to pick a "good" (i.e., low expected cost) plan.
  - Involves choosing access methods, physical operators, operator orders, …
  - Notion of cost is based on an abstract "cost model"

Sample Database

```
SAILORS
<table>
<thead>
<tr>
<th>sid</th>
<th>sid</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
<td>999</td>
<td>45.0</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>50</td>
<td>26.0</td>
</tr>
<tr>
<td>3</td>
<td>102</td>
<td>32</td>
<td>26.0</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>10</td>
<td>19.0</td>
</tr>
</tbody>
</table>

Reserves
<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>2014-02-01</td>
<td>macgyver</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>2014-02-02</td>
<td>a-team</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>2014-02-02</td>
<td>dallas</td>
</tr>
</tbody>
</table>
```

Sailors (sid: int, sname: varchar, rating: int, age: real)

Reserves (sid: int, bid: int, day: date, rname: varchar)
Sample Database

<table>
<thead>
<tr>
<th>SAILORS</th>
<th>RESERVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Christina</td>
</tr>
<tr>
<td>2</td>
<td>Obama</td>
</tr>
<tr>
<td>3</td>
<td>Topac</td>
</tr>
<tr>
<td>4</td>
<td>Bieber</td>
</tr>
</tbody>
</table>

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

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

Single-Table Selection

\[
\sigma_{\text{name}<"C"}(\text{Reserves})
\]

\[
\sigma_{\text{name}<"C"}(\text{Reserves})
\]

\[
\sigma_{\text{name}<"C"}(\text{Reserves})
\]

\[
\sigma_{\text{name}<"C"}(\text{Reserves})
\]

- 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** (Tree, Hash, List, …)

- 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 Options

<table>
<thead>
<tr>
<th></th>
<th>Scan</th>
<th>Eq</th>
<th>Range</th>
<th>Ins</th>
<th>Del</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>B</td>
<td>B/2</td>
<td>B</td>
<td>2</td>
<td>Search+1</td>
</tr>
<tr>
<td>sorted</td>
<td>B</td>
<td>log₂B</td>
<td>c − m</td>
<td>Search+B</td>
<td>Search+B</td>
</tr>
<tr>
<td>Clust.</td>
<td>1.5B</td>
<td>h</td>
<td>c − m</td>
<td>Search+1</td>
<td>Search+1</td>
</tr>
<tr>
<td>u-tree</td>
<td>B</td>
<td>1+h</td>
<td>c − m</td>
<td>Search+2</td>
<td>Search+2</td>
</tr>
<tr>
<td>u-hash</td>
<td>B</td>
<td>B−2</td>
<td>B</td>
<td>Search+2</td>
<td>Search+2</td>
</tr>
</tbody>
</table>

Selection: No Index, Unsorted Data

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

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

Selection: No Index, Sorted Data

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

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

With an index on selection attribute:
- Use index to find qualifying data entries, then retrieve corresponding data records.
- Note: Hash indexes are only useful for equality selections.

```
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).

B+Tree Indexes

CLUSTERED
Index entries direct search for data entries

UNCLUSTERED
Index entries

Data Entries
Data Records

B+Tree Indexes

Index entries direct search for data entries

Data entries (Index File) (Data file)

Data Records

CLUSTERED

<key, rid>

UNCLUSTERED

rid→data

Selection: B+Tree Index

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

• In example "Reserves" relation, if 10% of tuples qualify (100 pages, 10,000 tuples):
  – With a clustered index, cost is little more 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 rid.
  – Sort rid's of the data records to be retrieved.
  – Fetch 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).
Selection Conditions

• Q: What would you do?
• A: Try to find a selective (clustering) index.

• Convert to conjunctive normal form (CNF):

\[
\begin{align*}
\text{SELECT} & \ast \\
\text{FROM Reserves AS R} \\
\text{WHERE} & (R.\text{day} < '2014-02-01' \text{ AND} \\
& R.\text{rname} = 'Christos') \\
& \text{OR} R.\text{bid} = 5 \\
& \text{OR} R.\text{sid} = 3 \\
\end{align*}
\]

AND

\[
\begin{align*}
(R.\text{day}< '2014-02-01' \text{ OR} R.\text{bid}=5 \text{ OR} R.\text{sid}=3) \\
(R.\text{rname}= 'Christos' \text{ OR} R.\text{bid}=5 \text{ OR} R.\text{sid}=3)
\end{align*}
\]

• A B-tree index matches (a conjunction of) terms that involve only attributes in a prefix of the search key.
  – Index on \(<a, b, c>\) matches \((a=5 \text{ AND } b=3)\), but not \(b=3\).
  – For Hash index, we must have all attributes in search key.
Two Approaches to 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

\[(\text{day}<'2014-02-01' \text{ AND bid}=5 \text{ AND sid}=3)\]

- A B+ tree index on **day** can be used;
  - then, **bid=5** and **sid=3** must be checked for each retrieved tuple.
- Similarly, a hash index on **<bid,sid>** could be used;
  - Then, **day<‘2014-02-01’** must be checked.
Approach #1 – Example

• How about a B+tree on $<$rname,day$>$?
• How about a B+tree on $<$day,rname$>$?
• How about a Hash index on $<$day,rname$>$?

What if we have multiple indexes?

Approach #2

• Get $\textit{rids}$ from first index; $\textit{rids}$ from second index; intersect and fetch.
• 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

– With a B+ tree index on day and an index on sid,
  – We can retrieve $\textit{rids}$ of records satisfying $\text{day}< '2014\cdot 02\cdot 01'$ using the first,
  – $\textit{rids}$ of recs satisfying $\text{sid}=3$ using the second,
  – intersect,
  – retrieve records and check $\text{bid}=5$. 
Approach #2 – Example

(day<'2014-02-01' AND bid=5 AND sid=3)

Set intersection can be done with bitmaps, hash tables, or bloom filters.

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

- Introduction
- 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.

SQL JOINS

- SELECT * FROM Table A
- LEFT JOIN Table B ON A.Key = B.Key
- SELECT * FROM Table A
- INNER JOIN Table B ON A.Key = B.Key
- SELECT * FROM Table A
- FULL OUTER JOIN Table B ON A.Key = B.Key
- WHERE A.Key = B.Key
Joins

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

```plaintext
foreach tuple r of R
  foreach tuple s of S
    output, if they match
```

![Diagram](image)

M pages, \( m \) tuples

N pages, \( n \) tuples

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

- **Cost:** \( M + (pR \cdot M) \cdot N \)
Simple Nested Loop Join

- Actual number:
  - $M + (pR \cdot M) \cdot N = 1000 + 100 \cdot 1000 \cdot 500$
  - $= 50,001,000$ I/Os
  - At 10ms/IO, Total time $\approx 5.7$ days

- What if smaller relation (S) was outer?
  - Slightly better...

- What assumptions are being made here?
  - 1 buffer for each table (and 1 for output)

SSD $\approx 1.3$ hours at 0.1ms/IO
Simple Nested Loop Join

- Actual number:
  - \( M + (pR \cdot M) \cdot N = 1000 + 1 \)
  \( = 50,001,000 \) I/Os
  - At 10ms/IO, Total time \( \approx 5.7 \text{ days} \)
- What if smaller relation (S) was outer?
  - Slightly better...
- What assumptions are being made here?
  - 1 buffer for each table (and 1 for output)

Block Nested Loop Join

- Algorithm #1: Block Nested Loop Join

\[
\begin{array}{c}
\text{read block from } R \\
\text{read block from } S \\
\text{output, if tuples match}
\end{array}
\]

- Things are better.
- How many disk accesses (‘M’ and ‘N’ are the number of blocks for ‘R’ and ‘S’)?
  - Cost: \( M + (M-N) \)
Block Nested Loop Join

- **Algorithm #1: Optimizations**
- Which one should be the outer relation?
  - *The smallest (in terms of # of pages)*

\[ R(A, ...) \]
\[ S(A, ..., ...) \]

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

- What if we use the smaller one as the outer relation?

\[ \text{SSD} \approx 50 \text{ seconds at 0.1ms/IO} \]
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?

Algorithm #1: Using multiple buffers.

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

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

\( S(A, .......) \)

\( R(A, ...) \)

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

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

\(- \) Cost: \( M + \left\lceil \frac{M}{B - 2} \right\rceil \cdot N \)
Block Nested Loop Join

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

### Joins

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

### 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.
Index Nested Loop Join

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

\[
\begin{align*}
\text{foreach tuple } r \text{ of } R \\
\text{foreach tuple } s \text{ of } S, \text{ where } r_i &= s_j
\end{align*}
\]

Output

- Index Probe

\[
\begin{align*}
M \text{ pages, } m \text{ tuples} \\
R(A, \ldots) \\
N \text{ pages, } n \text{ tuples} \\
S(A, \ldots)
\end{align*}
\]

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

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

Index Nested Loop

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

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

  - **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
Joins

- 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.

Sort-Merge Join

- 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.
Sort-Merge Join

- Algorithm #3: Sort-Merge Join
- How many disk accesses (\(M\) and \(N\) are the number of blocks for \(R\) and \(S\))? 
  \[\text{Cost: } (2M \cdot \log M / \log B) + (2N \cdot \log N / \log B) + M + N\]

\[\begin{array}{c}
\text{\(M\) pages, } m \text{ tuples} \\
\text{\(R(A_{...})\)} \\
\downarrow \\
\text{\(S(A_{......})\)} \\
\uparrow \\
\text{\(N\) pages, } n \text{ tuples} \\
\end{array}\]

Sort-Merge Join Example

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Obama</td>
<td>50</td>
<td>52.0</td>
</tr>
<tr>
<td>2</td>
<td>Tupac</td>
<td>32</td>
<td>36.0</td>
</tr>
<tr>
<td>6</td>
<td>Bieber</td>
<td>10</td>
<td>19.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1001</td>
<td>2014-02-01</td>
<td>marlnel</td>
</tr>
<tr>
<td>2</td>
<td>1002</td>
<td>2014-02-02</td>
<td>magpyry</td>
</tr>
<tr>
<td>3</td>
<td>1001</td>
<td>2014-02-02</td>
<td>a-team</td>
</tr>
<tr>
<td>5</td>
<td>1001</td>
<td>2014-02-01</td>
<td>dallas</td>
</tr>
</tbody>
</table>

\[\text{Sort! Sort! Sort!}\]
Sort-Merge Join Example

\begin{center}
\textbf{SELECT * FROM Reserves R, Sailors S WHERE R.sid = S.sid}
\end{center}

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{sid} & \textbf{sname} & \textbf{rating} & \textbf{age} & \textbf{rname} \\
\hline
1 & Christos & 999 & 45.0 & \\
2 & Tupac & 32 & 26.0 & \\
3 & Obama & 50 & 52.0 & \\
6 & Bieber & 10 & 19.0 & \\
\hline
\end{tabular}

\begin{center}
\textbf{Merge!} \hspace{1cm} \textbf{Merge!}
\end{center}

• With 100 buffer pages, both Reserves and Sailors can be sorted in 2 passes:
  – Cost: 7,500 I/Os
  – At 10ms/Io, Total time \( \approx 75 \text{ seconds} \)

• Block Nested Loop:
  – Cost: 2,500 to 15,000 I/Os

SSD \( \approx 0.75 \text{ seconds} \) 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.

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
  foreach tuple s of S
  output, if h(s) ∈ 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:
  – Build another hash table for $H_{S(i)}$, and probe it
    with each tuple of $H_{R(i)}$

  \[
  \begin{array}{c}
  \text{R(A, ...)} \\
  \hline
  h_1 \\
  \vdots \\
  h_m \\
  \end{array}
  \hspace{2cm}
  \begin{array}{c}
  H_{R(i)} \\
  H_{S(i)} \\
  \vdots \\
  h_1 \\
  \end{array}
  \hspace{2cm}
  \begin{array}{c}
  \text{S(A, ...)} \\
  \end{array}
  \]

Grace Hash Join

• Choose the (page-wise) smallest - if it fits in memory, do a nested loop join
  – Build a hash table (with $h_2() \neq h()$)
  – And then probe it for each tuple of the other

Grace Hash Join

• What if $H_{S(i)}$ is too large to fit in memory?
  – Recursive Partitioning!
  – More details (overflows, hybrid hash joins) 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

- Actual number:
  - 3(M + N) = 3 · (1000 + 500) = 4,500 I/Os
  - At 10ms/IO, Total time ≈ 45 seconds

Sort-Merge Join vs. Hash Join

- 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?
Sort-Merge Join vs. Hash Join

• **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.
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
• Brief Midterm Review