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

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

Last Class

- Catalog
- Intro to Operator Evaluation
- Typical Query Optimizer
- Projection/Aggregation

Today’s Class

- More on Indexes
- Explain
- Joins
- Mid-term Review (Christos)

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!)
Selection Conditions

• 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.

B+Tree Prefix Search

Key = \(xy\)

Key = \(y\)

\(yz\)
\(xx\)
\(xy\)
\(zy\)
\(zz\)

Partial Indexes

• Create an index on a 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)
```

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

Today’s Class

- More on Indexes
  - Explain
  - Joins
  - Mid-term Review (Christos)

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.

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

**Pseudo Query Plan:**

```
SORT
COUNT
GROUP BY bid
RESERVES
```
EXPLAIN

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

EXPLAIN

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

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

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

Postgres v9.1

MySQL v5.5
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.

Today’s Class

- More on Indexes
- Explain
- Joins
- Mid-term Review (Christos)

Cost-based Query Sub-System

Sample Database

| Sailors (sid: int, sname: varchar, rating: int, age: real) |
| Reserve (sid: int, bid: int, day: date, rname: varchar) |

---

SAILORS
- sid | name  | rating | age |
- 1   | Clinton | 999    | 55  |
- 2   | Obama   | 50     | 52  |
- 3   | Tupac   | 32     | 26  |
- 6   | Bieber  | 10     | 19  |

RESERVES
- sid | bid   | day       | rname |
- 1   | 103   | 2014-02-01 | matlock |
- 1   | 102   | 2014-02-02 | macgyver |
- 2   | 101   | 2014-02-02 | a-team |
- 1   | 101   | 2014-02-01 | dallas |
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>Christos</td>
</tr>
<tr>
<td>3</td>
<td>Obama</td>
</tr>
<tr>
<td>5</td>
<td>Trump</td>
</tr>
<tr>
<td>6</td>
<td>Bieber</td>
</tr>
</tbody>
</table>

Each tuple is 50 bytes
80 tuples per page
500 pages total

Each tuple is 40 bytes
100 tuples per page
1000 pages total

$N=500, \ p_S=80$

$M=1000, \ p_R=100$

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
- RIGHT 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 IS NULL

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

We will ignore output costs

First Example

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

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

Simple Nested Loop Join

• Algorithm #0: Simple Nested Loop Join

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

Outer relation: $R(A, ..)$
Inner relation: $S(A, ......)$
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: \( M + (pR \cdot M) \cdot N \)

![Diagram](https://via.placeholder.com/150)

- 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:
  \[ M + (pR \cdot M) \cdot N = 1000 + 1 \cdot 50,001,000 \]
  - 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
  - Read block from R
  - Read block from S
  - Output, if tuples match

\[ M \text{ pages, } m \text{ tuples} \]
\[ N \text{ pages, } n \text{ tuples} \]

Block Nested Loop Join

- Algorithm #1: 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)
Block Nested Loop Join

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

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

SSD \( \approx 50 \) 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 \( \approx 1.4 \text{ hours} \)

• Algorithm #1: Using multiple buffers.

\[
\begin{align*}
\text{read } & B \cdot 2 \text{ blocks from } R \\
\text{read block from } & S \\
\text{output } & , \text{if tuples match}
\end{align*}
\]
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:** \( M + \left\lceil \frac{M}{B-2} \right\rceil \times N \)

```
M pages, m tuples  
\[ R(A_{\ldots}) \]  
S(A, ......)  
N pages, n tuples
```

- But if the outer relation fits in memory:
  - **Cost:** \( M+N = 1000 + 500 = 1500 \) I/Os
  - At 10ms/IO, Total time ≈ 15 seconds

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.

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

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

  \[
  \text{foreach} \quad \text{tuple } r \text{ of } R \quad \text{foreach} \quad \text{tuple } s \text{ of } S, \quad \text{where } r_i = s_j, \\
  \text{output}
  \]

  - Index Probe

  \[
  M \text{ pages,} \quad m \text{ tuples} \\
  R(A, \ldots) \\
  \]

  \[
  S(A, \ldots) \\
  \]

  \[
  N \text{ pages,} \quad n \text{ tuples}
  \]

  Look-up Cost

Index Nested Loop

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

  - How many disk accesses (‘\( M \)’ and ‘\( N \)’ are the number of blocks for ‘\( R \)’ and ‘\( S \)’)?
  
  \[
  \text{Cost: } M + m \cdot C
  \]

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

  \[
  M \text{ pages,} \quad m \text{ tuples} \\
  R(A, \ldots) \\
  \]

  \[
  S(A, \ldots) \\
  \]

  \[
  N \text{ pages,} \quad n \text{ tuples}
  \]

  Look-up Cost

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

- **Sort Phase**: First sort both tables on joining attribute.
- **Merge Phase**: 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 backtracking if duplicate values appear in join column.

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>Christos</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! Sort!

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

Merge! Merge!

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

✔ ✔ ✔ ✔
Sort-Merge Join

• **Algorithm #3:** Sort-Merge Join
  - How many disk accesses (‘$M$’ and ‘$N$’ are the number of blocks for ‘$R$’ and ‘$S$’)?
    - Cost: $(2M \cdot \log M / \log B) + (2N \cdot \log N / \log B) + M + N$

Sort-Merge Join Example

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

  \[
  \text{build hash table } H \text{ for } R \\
  \text{foreach tuple } s \text{ of } S \\
  \text{output, if } h(s) \in H
  \]

  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.

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

- Join each pair of matching buckets:
  - Build another hash table for $H_{S(i)}$, and probe it with each tuple of $H_{R(i)}$

- 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)

• 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

Grace Hash Join

• Actual number:
  – $3(M + N) = 3 \cdot (1000 + 500)$
  – At 10ms/IO, Total time $\approx 45$ 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.
  – Index Nested Loop when selectivity is small.
  – Sort-Merge/Hash when joining whole tables.