ICS 421 Spring 2010
Query Evaluation (ii)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa
What do these queries have in common?

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.rating > 5
ORDER BY S.age

SELECT DISTINCT S.sname
FROM    Sailors S

SELECT S.age, AVG(S.rating)
FROM    Sailors S
GROUP BY S.age
```
The Sort Operator

• Sorting is a classic problem in computer science!
• Data requested in sorted order
  – e.g., find students in increasing gpa order
• Sorting is first step in bulk loading B+ tree index.
• Sorting useful for eliminating duplicate copies in a collection of records (Why?)
• Sort-merge join algorithm involves sorting.
• Problem: sort 100Gb of data with 1Gb of RAM.
  – why not virtual memory?
Two-Way External Merge Sort

- **Pass 0:**
  - Read a page, sort it in memory, write it to disk
  - Only one buffer page is needed

- **Pass 1, 2, 3, 4 ...:**
  - Read two (sorted) pages, merge them to fill output page, flush output page when full.
  - 2 input pages and 1 output page are needed
Two-Way Merge Sort: Example

Input file
PASS 0
1-page runs
PASS 1
2-page runs
PASS 2
4-page runs
PASS 3
8-page runs

3,4  6,2  9,4  8,7  5,6  3,1  2
3,4  2,6  4,9  7,8  5,6  1,3  2
2,3  4,6  4,7  8,9  1,3  5,6  2
2,3  4,4  6,7  8,9
2,3  3,4  4,5  6,6  7,8  9
Two-Way Merge Sort: Analysis

- Input file has $N$ pages
- Each pass reads $N$ pages and writes $N$ pages.
- The number of passes $= \lceil \log_2 N \rceil + 1$
- So total cost is $= 2N(\lceil \log_2 N \rceil + 1)$
- **Idea:** **Divide and conquer:** sort subfiles and merge
K-Way External Merge Sort

• What if we have more memory?
• Sort a file with $N$ pages using $B$ buffer pages:
  - **Pass 0:**
    • read in $B$ pages, sort all $B$ pages in memory, write to disk as 1 run, repeat until all $N$ pages are sorted -- outputs $\left\lceil \frac{N}{B} \right\rceil$ sorted runs
  - **Pass 1,2,...:**
    • Use $B-1$ buffer pages as input and perform $(B-1)$-way merge to fill 1 output buffer page.
K-Way Merge Sort: Analysis

• B=5 buffer pages, N=108 pages
  – Pass 0: $\left\lfloor \frac{108}{5} \right\rfloor = 22$ sorted runs of 5 pages each
  – Pass 1: $\left\lfloor \frac{22}{4} \right\rfloor = 6$ sorted runs of 20 pages each
  – Pass 2: $\left\lfloor \frac{6}{4} \right\rfloor = 2$ sorted runs of 80 & 28 pages
  – Pass 3: 1 sorted file of 108 pages

• Number of passes = $\left\lceil \log_{B-1} \left\lfloor \frac{N}{B} \right\rfloor \right\rceil + 1$

• Each pass still reads N pages and writes N pages

• Total number of I/O’s = $2N \times (\left\lceil \log_{B-1} \left\lfloor \frac{N}{B} \right\rfloor \right\rceil + 1)$
Selection Operator

• Index vs Table Scan

• Multiple Indexes
  – Eg. Use index(age) & index(rating) for “age>20 AND rating>9”
  – Intersect RID sets using bloom filters
  – Eg. Use index(age) & index(rating) for “age>20 OR rating>9”
  – Union RID sets
Projection Operator

• Two steps:
  1. Remove unwanted columns
  2. Eliminate duplicates

• How do we do step 2?
  – External merge sort
  – Scan sorted data to remove duplicates

• Optimization: combine the 2 steps into merge sort:
  – Remove unwanted columns in Pass 0.
  – Subsequent passes can remove duplicates whenever they are encountered.
Join Algorithms

• Cost model
  – Single DBMS server: I/Os in number of pages
  – Distributed DBMS: network I/Os + local disk I/Os
  – $t_d$: time to read/write one page to local disk
  – $t_s$: time to ship one page over the network to another node

• Single server:
  – Nested Loop Join
  – Index Nested Loop Join
  – Sort Merge Join
  – Hash Join

• Distributed:
  – Semi-Join
  – Bloom Join
Nested Loop Join

For each data page $P_{S1}$ of $S1$
For each tuple $s$ in $P_{S1}$
  For each data page $P_{R1}$ of $R1$
  For each tuple $r$ in $P_{R1}$
    if ($s$.sid == $r$.sid)
      then output $s,r$

• Worst case number of local disk reads
  $= N\text{pages}(S1) + |S1| \times N\text{pages}(R1)$
Index Nested Loop Join

For each data page $P_{S_1}$ of $S_1$
For each tuple $s$ in $P_{S_1}$
  if ($s.\text{sid} \in \text{Index}(R1.\text{sid})$)
  then fetch $r$ & output $<s,r>$

- Worst case number of local disk reads with tree index
  $= N_{pages}(S1) + |S1| \times (1 + \log_F N_{pages}(\text{RIDS}(R1)))$
- Worst case number of local disk reads with hash index
  $= N_{pages}(S1) + |S1| \times 2$
Sort Merge Join

1. Sort S1 on SID
2. Sort R1 on SID
3. Compute join on SID using Merging algorithm

- If join attributes are relatively unique, the number of disk pages
  \[
  = N\text{pages}(S1) \log N\text{pages}(S1)
  + N\text{pages}(R1) \log N\text{pages}(R1)
  + N\text{pages}(S1) + N\text{pages}(R1)
  \]

- If the number of duplicates in the join attributes is large, the number of disk pages approaches that of nested loop join.

---

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>100</td>
<td>8/8/99</td>
</tr>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>99</td>
<td>10/12/95</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Distributed Joins

- Consider:
  - Reserves join Sailors

- Depends on:
  - Which node get the query
  - Whether tables are fragmented/partitioned or not

- Node 1 gets query
  - Perform join at Node 3 (or 4) ship results to Node 1 ?
  - Ship tables to Node 1 ?

- Node 3 gets query
  - Fetch sailors in loop ?
  - Cache sailors locally ?
Distributed Joins over Fragments

\[
R \text{ join } S = \sigma_{R.\text{sid} = S.\text{sid}} (R \times S) = \sigma_{R.\text{sid} = S.\text{sid}} ((R_1 \cup R_2) \times (S_1 \cup S_2)) = \sigma_{R.\text{sid} = S.\text{sid}} ((R_1 \times S_1) \cup (R_1 \times S_2) \cup (R_2 \times S_1) \cup (R_2 \times S_2)) = \sigma_{R.\text{sid} = S.\text{sid}} (R_1 \times S_1) \cup \sigma_{R.\text{sid} = S.\text{sid}} (R_1 \times S_2) \cup \sigma_{R.\text{sid} = S.\text{sid}} (R_2 \times S_1) \cup \sigma_{R.\text{sid} = S.\text{sid}} (R_2 \times S_2) = (R_1 \text{ join } S_1) \cup (R_1 \text{ join } S_2) \cup (R_2 \text{ join } S_1) \cup (R_2 \text{ join } S_2)
\]

This equivalence applies to splitting a relation into pages in a single server DBMS system too!

Equivalent to a union of joins over each pair of fragments.
Distributed Nested Loop

- Consider performing R1 join S2 on Node 1
- Page-oriented nested loop join:
  
  \[
  \text{For each page } r \text{ of } R1 \\
  \text{Fetch } r \text{ from local disk} \\
  \text{For each page } s \text{ of } S2 \\
  \text{Fetch } s \text{ if } s \notin \text{cache} \\
  \text{Output } r \text{ join } s
  \]

- Cost = \( N_{\text{pages}}(R1) \cdot t_d + N_{\text{pages}}(R1) \cdot N_{\text{pages}}(S2) \cdot (t_d + t_s) \)

- If cache can hold entire S2, cost is \( N_{\text{pages}}(R1) \cdot t_d + N_{\text{pages}}(S2) \cdot (t_d + t_s) \)
Semijoins

• Consider performing R1 join S2 on Node 1
• S2 needs to be shipped to R1
• Does every tuple in S2 join with R1?
• Semijoin:
  – Don’t ship all of S2
  – Ship only those S2 rows that will join with R1
  – Assumes that the join causes a reduction in S2!
• Cost = \( N_{pages}(R1)*t_d + N_{pages}(\pi_{sid}R1)*t_s + \text{Cost}(\cap) + N_{pages}(\sigma_{sid \in jsid}S2)*t_s + \text{Cost}(R1 \text{ join } \sigma_{sid \in jsid}S2) \)

2/9/2010
Lipyeow Lim -- University of Hawaii at Manoa
Bloomjoins

• Consider performing R1 join S2 on Node 1

• Can we do better than semijoin?

• Bloomjoin:
  – Don’t ship all of \((\pi_{sid} R1)\)
  – Node 1: Ship a “bloom filter” (like a signature) of \((\pi_{sid} R1)\)
    • Hash each sid
    • Set the bit for hash value in a bit vector
    • Send the bit vector \(v1\)
  – Node 2:
    • Hash each \((\pi_{sid} S2)\) to bit vector \(v2\)
    • Computer \((v1 \cap v2)\)
    • Send rows of S2 in the intersection

• False positives
R equijoin S on sid

1. Partition R into k partitions using hash function $h_1(R.sid)$

2. Partition S into k partitions using hash function $h_1(S.sid)$

3. Foreach partition $i$
   1. Build inmemory hash table $H(R[i])$ for $R[i]$ using $h_2(R.sid)$
   2. Foreach row in $S[i]$
      1. Probe $H(R[i])$
      2. Output join tuples $<r,s>$

• Works only on equi-joins
• Total I/Os = $2*NPages(R) + 2*NPages(S) + NPages(R) + NPages(S) = 3 * [Npages(R) + Npages(S)]$
• Can be applied in a distributed DBMS with hash partitions on the join attribute!