Overview of Query Evaluation: JOINS

Chapter 14
Cost metric: # of I/Os.

We will ignore output costs.
Schema for our Running Example

Sailors \((sid: \text{integer}, sname: \text{string}, rating: \text{integer}, age: \text{real})\)

Reserves \((sid: \text{integer}, bid: \text{integer}, day: \text{dates}, rname: \text{string})\)

- Reserves R:
  - Each tuple is 40 bytes long
  - Number of pages \(M\) (1000 pages)
  - \(p_R\) tuples per page (100 tuples)

- Sailors S:
  - Each tuple is 50 bytes long
  - Number of pages \(N\) (500 pages)
  - \(p_S\) tuples per page (80 tuples per page)
Equality Joins With One Join Column

SELECT * 
FROM Reserves R1, Sailors S1
WHERE R1.sid=S1.sid

❖ In algebra: R ∞ S.
❖ Common!
❖ Must be carefully optimized.
Typical Choices for Joins

- **Nested Loops Join**
  - Simple Nested Loops Join: Tuple-oriented
  - Simple Nested Loops Join: Page-oriented
  - Block Nested Loops Join
  - Index Nested Loops Join

- **Sort Merge Join**

- **Hash Join**
**Simple Nested Loops Join**

![Diagram of R and S relations]

- **Algorithm:**
  For each tuple in outer relation R, we scan inner relation S.

- **Cost:**
  - Scan of outer relation $R$ + for each tuple of outer, scan of inner relation $S$.
  - Cost = $M + (p_R \times M) \times N$
  - Cost = 1000 + $(100 \times 1000) \times 500$ IOs.
Simple Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_j then add <r, s> to result

- **Tuple-oriented:**
  *For each tuple in outer relation R, we scan inner relation S.*
  - Cost: $M + p_R \cdot M \cdot N = 1000 + 100 \cdot 1000 \cdot 500$ I/Os.

- **Page-oriented:**
  *For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r in R-page and S is in S-page.*

- **Cost:**
  - Scan of outer pages + for each page of outer, scan of inner relation.
  - Cost = $M + M \cdot N$
  - Cost = $1000 + 1000 \cdot 500$ IOs.
  - smaller relation (S) is outer, cost = $500 + 500 \cdot 1000$ IOs.
Join

❖ What if I had more buffer space available?
Block Nested Loops Join

What if B buffer pages available?

- One page as input buffer for scanning inner S
- One page as output buffer,
- **Remaining pages to hold ``block’’ of outer R.**
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result.
  - Then read next R-block, scan S again. Etc.
  - To find matching tuple? → Could use in-memory hashing!

![Diagram of Block Nested Loops Join](image_url)
Cost of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  
  - #outer blocks = \left\lceil \frac{\text{# of pages of outer}}{\text{blocksize}} \right\rceil
Examples of Block Nested Loops

- **Cost**: Scan of outer + \#outer blocks * scan of inner

- **With Reserves (R) as outer & 100 pages of R as block**:
  - Cost of scanning R is 1000 IOs; a total of 10 blocks.
  - Per block of R, we scan Sailors (S); 10*500 IOs.

- **With 100-page block of Sailors as outer**:
  - Cost of scanning S is 500 IOs; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 IOs.
Examples of Block Nested Loops

❖ Optimizations?
  ▪ With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.
  ▪ Double buffering would also be suitable.
Typical Choices for Joins

- **Nested Loops Join**
  - Simple Nested Loops Join: Tuple-oriented
  - Simple Nested Loops Join: Page-oriented
  - Block Nested Loops Join: Block-oriented

- Index Nested Loops Join
- Sort Merge Join
- Hash Join
Index Nested Loops Join

- An index on join column of one relation (say S), use S as inner and exploit the index.

- Cost:
  - Scan the outer relation R
  - **For each R tuple**, sum the costs of finding matching S tuples
  - Cost = \( M + (M \times p_R) \times \text{cost of finding matching S tuples} \)
  - with \( M = \# \text{pages of R} \) and \( p_R = \# \text{R tuples per page} \)
Using Clustered vs. Un-Clustered Index for matching tuples

- Data records are sorted on index key
- If index returns N tuples, how many I/Os?
  - \( \frac{N}{\text{number of tuples per block}} \)

\[ \Rightarrow \text{Number of tuples per block} = \frac{T(R)}{B(R)} \]

- Data records are randomly stored
- If index returns N tuples, how many I/Os?
  - \( N \)
Index-Based Join

For each \( r \in R \) do
\[ X \leftarrow \text{index-on-S.Y-lookup}(r.Y) \]

For each \( s \in X \) do
Output \((r,s)\) pair

Read R block at a time
\[ \Rightarrow B(R) \text{ if } R \text{ is clustered} \]
\[ \Rightarrow T(R) \text{ if } R \text{ is not clustered} \]

What is the expected size of \( X \)?
\[ \Rightarrow T(S) / V(S,Y) \]
(we assume uniform dist.)

What is the index I/O cost? (Index height = \( H \))
\[ \Rightarrow 0 \text{ if the index in memory} \]
\[ \Rightarrow H \text{ if entirely not in memory} \]
\[ \Rightarrow (H-z) \text{ if the } 1^{st} z \text{-levels of index are in memory} \]

How many lookups we do?
\[ \Rightarrow T(R) \]
Example: Index Nested Loops Join

- For each R tuple, cost of probing S index is:
  - about 1.2 for hash index,
  - 2-4 for B+ tree.

- Cost of retrieving S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering and on # of tuples retrieved:
  - Clustered: 1 I/O (typical),
  - Unclustered: up to 1 I/O per matching S tuple.
Examples of Index Nested Loops

- Hash-index (Alt. 2) on sid of Sailors (as inner):
  - Scan Reserves:
    - 1000 page I/Os,
    - 100*1000 tuples.
  - For each Reserves tuple:
    - 1.2 IOs to get data entry in index,
    - plus 1 IO to get (the exactly one) matching Sailors tuple.
    - We have $100,000 \times (1.2 + 1) = 220,000$ IOs.
  - In total, we have:
    - 1000 IOs plus
    - 220,000 IOs.
    - Equals 221,000 IOs
Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves:
    - 1000 page I/Os,
    - 100*1000 tuples.
  - For each Reserves tuple:
    - 1.2 IOs to get data entry in index,
    - plus 1 IO to get (the exactly one) matching Sailors tuple.
    - We have 100,000 * (1.2 + 1) = 220,000 IOs.
  - In total, we have:
    - 1000 IOs plus
    - 220,000 IOs.
    - Total is 221,000 IOs
Example of Index Nested Loop Join

- Hash-index (Alt. 2) on sid of Reserves (as inner):
  - Scan Sailors:
    - 500 page I/Os,
    - $80 \times 500$ tuples = 40,000 tuples.
  - For each Sailors tuple:
    - 1.2 IOs to find index page with data entries,
    - Plus, cost of retrieving matching Reserves tuples.
    - Assuming uniform distribution:
      - 2.5 reservations per sailor ($100,000 / 40,000$).
    - Cost of retrieving them is 1 or 2.5 IOs (if index unclustered)

Total: $500 + 40,000 \times (1.2 + 2.5 \times 1)$. 
Simple vs. Index Nested Loops Join

- Assume: M Pages in R, $p_R$ tuples per page, N Pages in S, $p_S$ tuples per page, B Buffer Pages.

- Nested Loops Join
  - Simple Nested Loops Join
    - Tuple-oriented: $M + p_R * M * N$
    - Page-oriented: $M + M * N$
    - Smaller as outer helps.
  - Block Nested Loops Join
    - $M + \lceil M/(B-2) \rceil * N$
    - Dividing buffer evenly between R and S helps.
  - Index Nested Loops Join
    - $M + ( (M*p_R) * \text{cost of finding matching S tuples})$
    - $\text{cost of finding matching S tuples} = (\text{cost of Probe} + \text{cost of retrieval})$

- With unclustered index, if number of matching inner tuples for each outer tuple is small, cost of INLJ is smaller than SNLJ.
Join

Use Sorting ?
Join: Sort-Merge \((R \bowtie S)_{i=j}\)

1. Sort R and S on the join column.
2. Scan R and S to do a "merge" on join column.
3. Output result tuples.
### Example of Sort-Merge 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>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</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>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>
Join: Sort-Merge \((R \bowtie S)_{i=j}\)

- **Note:**
  - R is scanned once; each S group is scanned once per matching R tuple.
  - Multiple scans of an S group are likely to find needed pages in buffer.
Join: \textbf{Sort-Merge} \((R \bowtie S)\)

\[
\begin{array}{c}
\text{(1). Sort R and S on the join column.} \\
\text{(2). Scan R and S to do a \textquote{merge} on join col.} \\
\text{(3). Output result tuples.}
\end{array}
\]

- **Merge on Join Column:**
  - Advance scan of R until current R-tuple \(\geq\) current S tuple,
  - then advance scan of S until current S-tuple \(\geq\) current R tuple;
  - do this until current R tuple = current S tuple.

  - At this point, all R tuples with same value in \(R_i\) \((\text{current R group})\) and all S tuples with same value in \(S_j\) \((\text{current S group})\) match;
  - So output \(<r, s>\) for all pairs of such tuples.

  - Then resume scanning R and S (as above)
### Sort-Merge Join Example:

#### Table 1:

<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>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

#### Table 2:

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

Assume sorted on same column which is also JOIN column.
Naïve Two-Pass JOIN (B(R) to denote M)

2-Pass Sort

I/O Cost = 4 M

Notice: we counted the output writing since it is intermediate

Sorted S

I/O Cost = 4 N

2-Pass Sort

I/O Cost = M + N

Total I/O Cost = 5 M + 5 N
Naïve Two-Pass JOIN

2-Pass Sort

No Constraints, B=3

>> B(R) <= B^2

Sorted S

>> B(S) <= B^2

From the sorting algorithm

Output buffer

Joined output
## Cost of Sort-Merge Join

### Cost of sort-merge:
- Sort R
- Sort S
- Merge R and S
# Example of Sort-Merge 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>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</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>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- Best case: ?
- Worst case: ?
- Average case:
Cost of Sort-Merge Join

- **Best Case Cost:** $(M+N)$
  - Already sorted.
  - The cost of scanning, $M+N$

- **Worst Case Cost:** $M \log M + N \log N + (M \times N)$
- Many pages in $R$ in same partition. (Worst, all of them). The pages for this partition in $S$ don’t fit into RAM. Re-scan $S$ is needed. Multiple scan $S$ is expensive!
Cost of Sort-Merge Join

- Average Cost:
  - In practice, roughly linear in M and N
  - So $O(M \log M + N \log N + (M+N))$

Note: Guarantee $M+N$ if key-FK join, or no duplicates.
Example of Sort-Merge Join

- Assume $B = \{35, 100, 300\}$; and $R = 1000$ pages, $S = 500$ pages

- **Sort-Merge Join**
  - both $R$ and $S$ can be sorted in 2 passes,
  - $\log M = \log N = 2$
  - total join cost: $2\times2\times1000 + 2\times2\times500 + (1000 + 500) = 7500$.

- **Block Nested Loops Join**: $2500 \sim 15000$
Refinement of Sort-Merge Join

IDEA: Combine the last merging phases when sorting R (or S) with the merging in join algorithm.

In the last round:

1. Allocate 1 page per run of each relation, and
2. ‘Merge’ while checking the join condition.
**Sort-Merge Join: Efficient Two-Pass JOIN**

**Main Idea:** Combine Pass 2 of the Sort with the Join

**Phase 1 in Sorting As Is**

- **INPUT 1**
- **INPUT 2**
- **INPUT B**

**Sorted runs of R (we have M/B)**

**Phase 2 Merge & Join**

- One buffer for each sorted run from both R & S
- One buffer for the join output

**Output buffer**

**Memory**

**Sorted runs of S (we have N/B)**

**INPUT 1**

**INPUT 2**

**INPUT B**

**M Main memory buffers**

**Disk**

**Disk**

**Disk**

**Disk**

**Disk**

**INPUT 1**

**INPUT 2**

**INPUT B**

**M Main memory buffers**

**Disk**

**Disk**

**Disk**

**Disk**

**Disk**
**Opt 2-Pass Sort-Merge-Join: Cost?**

**Main Idea:** Combine Pass 2 of the Sort with the Join

**Phase 1 in Sorting As Is**
- Sorted runs of R (we have $M/B$)
- Sorted runs of S (we have $N/B$)

**Phase 2 Merge & Join**
- One buffer for each sorted run from both R & S
- One buffer for the join output

**Total Cost**
- $3M + 3N$
Example: Refinement of Sort-Merge Join

- **Cost:**
  - (read+write R and S in Pass 0 and if needed in all but last pass)
  - + (read R and S in merging pass and join on fly)
  - + (writing of result tuples – which we typically ignore).

- In our running example, cost goes down from 7500 to 4500 IOs.
When Possible: Efficient Sort-Merge Join?

- Must have enough space:
  - With $B > \sqrt{L}$, where $L$ is the size of the larger relation.
  - The number of runs per relation is less than $B$.
  - At end, # of runs of both relations must fit into buffer
**Opt 2-Pass Sort-Merge-Join: When?**

### Phase 1 in Sorting As Is

- **R**
- **S**

**Sorted runs of R (we have** $M/B$**)**

**Sorted runs of S (we have** $N/B$**)**

**One pass?**

### Phase 2 Merge & Join

- One buffer for each sorted run from both R & S
- One buffer for the join output

**Number of runs must fit in memory:**

$$M/B + N/B \leq B \Rightarrow M + N \leq B^2$$
Hash-Join
**Hash-Join**

**IDEA:** Partition both relations using same hash function $h_1$: R tuples in partition $R_i$ will only match S tuples in partition $S_i$. 
Hash-Join: Partitioning Phase

Original Relation

Disk

\[ \cdots \]

B main memory buffers

\text{hash function } \text{h1}

INPUT

OUTPUT

1

2

\text{B-1}

Partitions

\text{Disk}

\text{Disk}
Hash-Join: Joining Phase

Idea: Join partition $R_i$ with partition $S_i$

Process:
- Read in a partition $R_i$ of $R$
- Hash it using $h2 (<> h1!)$ in BUFFER
- Scan matching partition $S_i$ of $S$ (page by page)
- Search for matches among $R_i$ and page of $S_i$. 
Hash-Join: Joining Phase

Partitions of R & S

Hash table for partition Ri (k < B-1 pages)

Input buffer for Si

Output buffer

Join Result

Disk

B main memory buffers

Hash table for partition Ri (k < B-1 pages)

Input buffer for Si

Output buffer

Join Result

Disk
Cost of Hash-Join

- In partitioning phase, read+write both relations:
  - $2(M+N)$.

- In matching phase, read both relations:
  - $M+N$.

- Total: $3(M+N)$

- E.g., total of 4500 I/Os in our running example.
Observation on Hash-Join

- Memory Requirement: Partition fit into available memory?
  - Assuming B buffer pages. #partitions k <= B-1
  - Assuming uniformly sized partitions, and maximizing k, we get:
    - k = B-1, and M/(B-1)
      - in-memory hash table to speed up the matching of tuples, a little more memory is needed: f * M/(B-1)
        with f the fudge factor used to capture the small increase in size between the partition and a hash table for partition.
  - Probing phase, one for inputting S, one for output, B > f*M/(B-1)+2 for hash join to perform well.
Observation on Hash Join (overflow)

- If hash function does not partition uniformly, one or more R partitions may not fit in memory.

- Significantly could degrade the performance.

- IDEA: Apply hash-join technique recursively to do the join of this overflow R-partition with corresponding S-partition.
Hash-Join vs. Sort-Merge Join

- Given a certain amount of memory: \( B > \sqrt{N} \) with \( N \) the larger relation size. Then both have a cost of \( 3(M+N) \) IOs.

- If partition is not uniformly sized (data skew); Sort-Merge less sensitive; plus result is sorted.

- Hash Join superior if relation sizes differ greatly; \( B \) is between \( \sqrt{N} \) and \( \sqrt{M} \).
General Join Conditions

- **Equalities over several attributes**
  - (e.g., $R.sid=S.sid \text{ AND } R.rname=S.sname$): 
    - **INL-Join**: build index on $<sid, sname>$ (if $S$ is inner); or use existing indexes on $sid$ or $sname$.
    - **SM-Join and H-Join**: sort/partition on combination of the two join columns.

- **Inequality conditions**
  - (e.g., $R.rname < S.sname$):
    - **INL-Join**: need (clustered!) B+ tree index.
      - Range probes on inner; # matches likely much higher than for equality joins.
    - **Hash Join, Sort Merge Join** not applicable.
    - **Block NL** quite likely to be the very reasonable join method here.
Summary

- There are several alternative evaluation algorithms for each relational operator.
Conclusion

Not one method wins!

Optimizer must assess situation to select best possible candidate