CPS216: Advanced Database Systems

Notes 07: Query Execution
(Sort and Join operators)

Shivnath Babu
Query Processing - In class order

1. SQL query
   - Parse
     - Parse tree
   - Query rewriting
     - Logical query plan
     - Statistics
       - Physical plan generation
         - Physical query plan
           - Execute
             - Result

1; 13, 15
2; 16.1
3; 16.2, 16.3
4; 16.4–16.7
Roadmap

• A simple operator: Nested Loop Join
• Preliminaries
  – Cost model
  – Clustering
  – Operator classes
• Operator implementation (with examples from joins)
  – Scan-based
  – Sort-based
  – Using existing indexes
  – Hash-based
• Buffer Management
• Parallel Processing
Nested Loop Join (NLJ)

- NLJ (conceptually)
  for each $r \in R1$ do
    for each $s \in R2$ do
      if $r.C = s.C$ then output $r,s$ pair
Nested Loop Join (contd.)

- Tuple-based
- Block-based
- Asymmetric
Implementing Operators

- Basic algorithm
  - Scan-based (e.g., NLJ)
  - Sort-based
  - Using existing indexes
  - Hash-based (building an index on the fly)

- Memory management
  - Tradeoff between memory and #IOs

- Parallel processing
Roadmap

- A simple operator: Nested Loop Join
- Preliminaries
  - Cost model
  - Clustering
  - Operator classes
- Operator implementation (with examples from joins)
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Buffer Management
- Parallel Processing
Operator Cost Model

• **Simplest:** Count # of disk blocks read and written during operator execution
• Extends to query plans
  – Cost of query plan = Sum of operator costs
• Caution: Ignoring CPU costs
Assumptions

• Single-processor-single-disk machine
  – Will consider parallelism later
• Ignore cost of writing out result
  – Output size is independent of operator implementation
• Ignore # accesses to index blocks
Parameters used in Cost Model

\[ B(R) = \# \text{ blocks storing } R \text{ tuples} \]
\[ T(R) = \# \text{ tuples in } R \]
\[ V(R,A) = \# \text{ distinct values of attr } A \text{ in } R \]
\[ M = \# \text{ memory blocks available} \]
Roadmap

• A simple operator: Nested Loop Join
• Preliminaries
  – Cost model
  – Clustering
  – Operator classes
• Operator implementation (with examples from joins)
  – Scan-based
  – Sort-based
  – Using existing indexes
  – Hash-based
• Buffer Management
• Parallel Processing
Notions of clustering

• Clustered file organization
  
  \[
  \begin{array}{cccc}
  R1 & R2 & S1 & S2 \\
  R3 & R4 & S3 & S4 \\
  \end{array}
  \quad \ldots
  \]

• Clustered relation
  
  \[
  \begin{array}{cccc}
  R1 & R2 & R3 & R4 \\
  R5 & R5 & R7 & R8 \\
  \end{array}
  \quad \ldots
  \]

• Clustering index
Clustering Index

Tuples with a given value of the search key packed in as few blocks as possible
Examples

T(R) = 10,000
B(R) = 200
If R is clustered, then \# R tuples per block = 10,000/200 = 50
Let V(R,A) = 40

⇒ If I is a clustering index on R.A, then \# IOs to access \( \sigma_{R.A} = "a"(R) = 250/50 = 5 \)

⇒ If I is a non-clustering index on R.A, then \# IOs to access \( \sigma_{R.A} = "a"(R) = 250 \ (> B(R)) \)
## Operator Classes

<table>
<thead>
<tr>
<th></th>
<th>Tuple-at-a-time</th>
<th>Full-relation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary</strong></td>
<td>Select</td>
<td>Sort</td>
</tr>
<tr>
<td><strong>Binary</strong></td>
<td></td>
<td>Difference</td>
</tr>
</tbody>
</table>
Roadmap

• A simple operator: Nested Loop Join
• Preliminaries
  – Cost model
  – Clustering
  – Operator classes
• Operator implementation (with examples from joins)
  – Scan-based
  – Sort-based
  – Using existing indexes
  – Hash-based
• Buffer Management
• Parallel Processing
Implementing Tuple-at-a-time Operators

• One pass algorithm:
  – Scan
  – Process tuples one by one
  – Write output

• Cost = B(R)
  – Remember: Cost = # IOs, and we ignore the cost to write output
Implementing a Full-Relation Operator, Ex: Sort

• Suppose $T(R) \times \text{tupleSize}(R) \leq M \times |B(R)|$
• Read $R$ completely into memory
• Sort
• Write output
• Cost = $B(R)$
Implementing a Full-Relation Operator, Ex: Sort

• Suppose R won’t fit within M blocks
• Consider a two-pass algorithm for Sort; generalizes to a multi-pass algorithm
• Read R into memory in M-sized chunks
• Sort each chunk in memory and write out to disk as a sorted sublist
• Merge all sorted sublists
• Write output
Two-phase Sort: Phase 1

Suppose $B(R) = 1000$, $R$ is clustered, and $M = 100$
Two-phase Sort: Phase 2

Sorted Sublists

Memory

Sorted R
Analysis of Two-Phase Sort

• Cost = 3xB(R) if R is clustered, 
  = B(R) + 2B(R’) otherwise

• Memory requirement M >= B(R)\(^{1/2}\)
Duplicate Elimination

• Suppose $B(R) \leq M$ and $R$ is clustered
• Use an in-memory index structure
• Cost = $B(R)$
• Can we do with less memory?
  – $B(\delta(R)) \leq M$
  – Aggregation is similar to duplicate elimination
Duplicate Elimination Based on Sorting

• Sort, then eliminate duplicates
• Cost = Cost of sorting + B(R)
• Can we reduce cost?
  – Eliminate duplicates during the merge phase
Back to Nested Loop Join (NLJ)

- NLJ (conceptually)
  for each $r \in R$ do
    for each $s \in S$ do
      if $r.C = s.C$ then output $r,s$ pair
Analysis of Tuple-based NLJ

- Cost with R as outer = $T(R) + T(R) \times T(S)$
- Cost with S as outer = $T(S) + T(R) \times T(S)$
- $M \geq 2$
Block-based NLJ

- Suppose R is outer
  - Loop: Get the next M-1 R blocks into memory
  - Join these with each block of S

- \( B(R) + \left( \frac{B(R)}{M-1} \right) \times B(S) \)

- What if S is outer?
  - \( B(S) + \left( \frac{B(S)}{M-1} \right) \times B(R) \)
Let us work out an NLJ Example

• Relations are not clustered
• $T(R1) = 10,000 \quad T(R2) = 5,000$
  10 tuples/block for $R1$; and for $R2$
  $M = 101$ blocks

**Tuple-based NLJ Cost:** for each $R1$ tuple:

$$[\text{Read tuple} + \text{Read R2}]$$

Total $= 10,000 \times [1+5000] = 50,010,000$ IOs
Can we do better when R,S are not clustered?

Use our memory

(1) Read 100 blocks worth of R1 tuples
(2) Read all of R2 (1 block at a time) + join
(3) Repeat until done
Cost: for each R1 chunk:
  Read chunk: 1000 IOs
  Read R2:  5000 IOs
  Total/chunk =  6000

Total = \frac{10,000}{1,000} \times 6000 = 60,000 \text{ IOs}  
[Vs. 50,010,000!]
• Can we do better?

Reverse join order: $R_2 \bowtie R_1$

Total = $\frac{5000 \times (1000 + 10,000)}{1000}$

$5 \times 11,000 = 55,000$ IOs

[Vs. 60,000]
Example contd. NLJ R2 \(\bowtie\) R1

- Now suppose relations are clustered

Cost
For each R2 chunk:
- Read chunk: 100 IOs
- Read R1: 1000 IOs
  \[\text{Total/chunk} = 1,100\]
Total = 5 chunks \(\times\) 1,100 = 5,500 IOs
  \[\text{Vs. 55,000}\]
Joins with Sorting

- **Sort-Merge Join** (conceptually)
  1. if R1 and R2 not sorted, sort them
  2. i ← 1; j ← 1;
     While (i ≤ T(R1)) ∧ (j ≤ T(R2)) do
        if R1{ i }.C = R2{ j }.C then **OutputTuples**
        else if R1{ i }.C > R2{ j }.C then j ← j+1
        else if R1{ i }.C < R2{ j }.C then i ← i+1
Procedure **Output-Tuples**

While \((R_1\{i\}.C = R_2\{j\}.C) \land (i \leq T(R_1))\) do

\[ jj \leftarrow j; \]

while \((R_1\{i\}.C = R_2\{jj\}.C) \land (jj \leq T(R_2))\) do

[output pair \(R_1\{i\}, R_2\{jj\}\);]

\[ jj \leftarrow jj+1 \]

\[ i \leftarrow i+1 \]
## Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{i}.C</th>
<th>R2{j}.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>7</td>
</tr>
</tbody>
</table>
Block-based Sort-Merge Join

• Block-based sort
• Block-based merge
Two-phase Sort: Phase 1

Suppose $B(R) = 1000$ and $M = 100$

1 2 3 4
5

1 2 3 4
5

96 97 98 99

100

Memory

1 101 201

100 200 300

Sorted Sublists

801 901

900 1000

R
Two-phase Sort: Phase 2

Memory

Sorted Sublists

100
200
300
900
1000

....
....
....
....
....

1
101
201
801
901

1
2
3
4
5
6
7
8
9
10

Sorted R

1
2
3
4
5
999
1000
Sort-Merge Join

R1 → Sorted R1

R2 → Sorted R2

sorted sublists

Apply our merge algorithm
Analysis of Sort-Merge Join

• Cost = 5 \times (B(R) + B(S))

• Memory requirement:
  \[ M \geq (\max(B(R), B(S)))^{1/2} \]
Continuing with our Example

R1, R2 clustered, but unordered

Total cost = sort cost + join cost

\[ = 6,000 + 1,500 = 7,500 \text{ IOs} \]

But: NLJ cost = 5,500

So merge join does not pay off!
However …

- NLJ cost = $B(R) + \frac{B(R)B(S)}{M-1} = O(B(R)B(S))$ [Quadratic]
- Sort-merge join cost = $5 \times (B(R) + B(S)) = O(B(R) + B(S))$ [Linear]
Can we Improve Sort-Merge Join?

Do we need to create the sorted R1, R2?
A more “Efficient” Sort-Merge Join

Apply our merge algorithm

sorted sublists
Analysis of the “Efficient” Sort-Merge Join

• Cost = 3 x (B(R) + B(S))
  [Vs. 5 x (B(R) + B(S))]  

• Memory requirement:
  M >= (B(R) + B(S))^{1/2}
  [Vs. M >= (max(B(R), B(S)))^{1/2}

Another catch with the more “Efficient” version: Higher chances of thrashing!
Cost of “Efficient” Sort-Merge join:

Cost = Read R1 + Write R1 into sublists
     + Read R2 + Write R2 into sublists
     + Read R1 and R2 sublists for Join
     = 2000 + 1000 + 1500 = 4500

[ Vs. 7500 ]
Memory requirements in our Example

\[ B(R1) = 1000 \text{ blocks, } 1000^{1/2} = 31.62 \]
\[ B(R2) = 500 \text{ blocks, } 500^{1/2} = 22.36 \]
\[ B(R1) + B(R2) = 1500, 1500^{1/2} = 38.7 \]

M > 32 buffers for simple sort-merge join
M > 39 buffers for efficient sort-merge join
Joins Using Existing Indexes

- Indexed NLJ (conceptually)

  for each \( r \in R \) do
  
  for each \( s \in S \) that matches \( \text{probe}(I,r.C) \) do
  
  output \( r,s \) pair
Continuing with our Running Example

- Assume R1.C index exists; 2 levels
- Assume R2 clustered, unordered

- Assume R1.C index fits in memory
Cost: R2 Reads: 500 IOs

for each R2 tuple:
- probe index - free
- if match, read R1 tuple

# R1 Reads depends on:
- # matching tuples
- clustering index or not
What is expected # of matching tuples?

(a) say R1.C is key, R2.C is foreign key
then expected = 1 tuple

(b) say V(R1,C) = 5000,  T(R1) = 10,000
with uniform assumption
expect = 10,000/5,000 = 2
What is expected # of matching tuples?

(c) Say \( \text{DOM}(R1, C) = 1,000,000 \)
\[ T(R1) = 10,000 \]

with assumption of uniform distribution in domain

Expected = \[ \frac{10,000}{1,000,000} = \frac{1}{100} \] tuples
Total cost with Index Join with a Non-Clustering Index

(a) Total cost = 500 + 5000(1) = 5,500

(b) Total cost = 500 + 5000(2) = 10,500

(c) Total cost = 500 + 5000(1/100) = 550

Will any of these change if we have a clustering index?
What if index does not fit in memory?

Example: say R1.C index is 201 blocks

• Keep root + 99 leaf nodes in memory
• Expected cost of each index access is
  \[ E = \frac{(0)99}{200} + \frac{(1)101}{200} \approx 0.5 \]
Total cost (including Index Probes)

\[ = 500 + 5000 \text{ [Probe + Get Records]} \]
\[ = 500 + 5000 \times [0.5 + 2] \]
\[ = 500 + 12,500 = 13,000 \quad \text{(Case b)} \]

For Case (c):

\[ = 500 + 5000 \times [0.5 \times 1 + (1/100) \times 1] \]
\[ = 500 + 2500 + 50 = 3050 \text{ IOs} \]
Block-Based NLJ Vs. Indexed NLJ

• Wrt #joining records
• Wrt index clustering

Plot graphs for Block NLJ and Indexed NLJ for clustering and non-clustering indexes
Sort-Merge Join with Indexes

- Can avoid sorting
- Zig-zag join
So far

**not clustered**

- NLJ R2 \(\bowtie\) R1 \(\sim 55,000\) (best)
- Merge Join \(\sim\)
- Sort+ Merge Join \(\sim\)
- R1.C Index \(\sim\)
- R2.C Index \(\sim\)

**clustered**

- NLJ R2 \(\bowtie\) R1 \(\sim 5500\)
- Merge join \(\sim 1500\)
- Sort+Merge Join \(\sim 7500 \rightarrow 4500\)
- R1.C Index \(\sim 5500, 3050, 550\)
- R2.C Index \(\sim\)
Building Indexes on the fly for Joins

- Hash join (conceptual)
  - Hash function $h$, range $1 \rightarrow k$
  - Buckets for $R1$: $G1, G2, \ldots, Gk$
  - Buckets for $R2$: $H1, H2, \ldots, Hk$

**Algorithm**
1. Hash $R1$ tuples into $G1--Gk$
2. Hash $R2$ tuples into $H1--Hk$
3. For $i = 1$ to $k$ do
   - Match tuples in $Gi, Hi$ buckets
Example Continued: Hash Join

- R1, R2 contiguous
- Use 100 buckets
- Read R1, hash, + write buckets
-> Same for R2
-> Read one R1 bucket; build memory hash table
   [R1 is called the **build** relation of the hash join]
-> Read corresponding R2 bucket + hash probe
   [R2 is called the **probe** relation of the hash join]

R1
   :
   :
   :

Memory

Then repeat for all buckets
Cost:

“Bucketize:”  Read R1 + write
Read R2 + write

Join:  Read R1, R2

Total cost = 3 \times [1000+500] = 4500
Minimum Memory Requirements

Size of R1 bucket = \( \frac{x}{k} \)

\[ k = \text{number of buckets } (k = M-1) \]
\[ x = \text{number of R1 blocks} \]

So...

\[ \frac{x}{k} \leq k \implies k \geq \sqrt{x} \implies M > \sqrt{x} \]

Actually, \( M > \sqrt{\min(B(R),B(S))} \)

[Vs. \( M > \sqrt{B(R)+B(S)} \) for Sort-Merge Join]
Trick: keep some buckets in memory

E.g., $k' = 33$  
$R1$ buckets = 31 blocks
keep 2 in memory

Memory use:

- $G1$: 31 buffers
- $G2$: 31 buffers
- Output: 33-2 buffers
- $R1$ input: 1

Total: 94 buffers
6 buffers to spare!!

called Hybrid Hash-Join
Next: Bucketize R2

- R2 buckets = 500/33 = 16 blocks
- Two of the R2 buckets joined immediately with G1, G2
Finally: Join remaining buckets
– for each bucket pair:
  • read one of the buckets into memory
  • join with second bucket
Cost

• Bucketize R1 = 1000 + 31 \times 31 = 1961

• To bucketize R2, only write 31 buckets:
  so, cost = 500 + 31 \times 16 = 996

• To compare join (2 buckets already done)
  read 31 \times 31 + 31 \times 16 = 1457

Total cost = 1961 + 996 + 1457 = 4414
How many Buckets in Memory?

OR ...

See textbook for an interesting answer ...
Another hash join trick:

- Only write into buckets \(<\text{val},\text{ptr}\)> pairs
- When we get a match in join phase, must fetch tuples
• To illustrate cost computation, assume:
  – 100 <val,ptr> pairs/block
  – expected number of result tuples is 100
• Build hash table for R2 in memory
  5000 tuples → 5000/100 = 50 blocks
• Read R1 and match
• Read ~ 100 R2 tuples

\[
\text{Total cost} = \begin{align*}
\text{Read R2:} & \quad 500 \\
\text{Read R1:} & \quad 1000 \\
\text{Get tuples:} & \quad 100 \\
\end{align*}
\]
\[= 1600\]
So far:

- NLJ: 5500
- Merge join: 1500
- Sort+merge joint: 7500
- R1.C index: 5500 → 550
- R2.C index: ______
- Build R.C index: ______
- Build S.C index: ______
- Hash join: 4500
  - with trick, R1 first: 4414
  - with trick, R2 first: ______
- Hash join, pointers: 1600
Hash-based Vs. Sort-based Joins

• Some similarities (see textbook), some dissimilarities
• Non-equi joins
• Memory requirement
• Sort order may be useful later
Summary

• NLJ ok for “small” relations (relative to memory size)
• For equi-join, where relations not sorted and no indexes exist, Hybrid Hash Join usually best
Summary (contd.)

• Sort-Merge Join good for non-equi-join (e.g., R1.C > R2.C)

• If relations already sorted, use Merge Join

• If index exists, it could be useful
  – Depends on expected result size and index clustering

• Join techniques apply to Union, Intersection, Difference
Buffer Management

• DBMS Buffer Manager

• May control memory directly (i.e., does not allocate from virtual memory controlled by OS)
Buffer Replacement Policies

- Least Recently Used (LRU)
- Second-chance
- Most Recently Used (MRU)
- FIFO
Interaction between Operators and Buffer Management

• Memory (our M parameter) may change while an operator is running
• Some operators can take advantage of specific buffer replacement policies
  – E.g., Rocking for Block-based NLJ
Join Strategies for Parallel Processors

- May cover later if time permits
- We will see one example: Hash Join
Textbook Material

• All of Chapter 15 except 15.8
  – 15.8 covers multi-pass sort and hash
Roadmap

• A simple operator: Nested Loop Join
• Preliminaries
  – Cost model
  – Clustering
  – Operator classes
• Operator implementation (with examples from joins)
  – Scan-based
  – Sort-based
  – Using existing indexes
  – Hash-based
• Buffer Management
• Parallel Processing