Chapters 15 and 16b: Query Optimization

(Slides by Hector Garcia-Molina, http://www-db.stanford.edu/~hector/cs245/notes.htm)

---> Generating and comparing plans

To generate plans consider:
- Transforming relational algebra expression (e.g. order of joins)
- Use of existing indexes
- Building indexes or sorting on the fly
• Implementation details:
  e.g. - Join algorithm
  - Memory management
  - Parallel processing

Estimating IOs:
• Count # of disk blocks that must be read (or written) to execute query plan

To estimate costs, we may have additional parameters:
  B(R) = # of blocks containing R tuples
  f(R) = max # of tuples of R per block
  M   = # memory blocks available
  HT(i) = # levels in index i
  LB(i) = # of leaf blocks in index i
Clustering index

Index that allows tuples to be read in an order that corresponds to physical order

A

\[
\begin{array}{c}
10 \\
15 \\
17 \\
19 \\
35 \\
37 \\
\end{array}
\]

Notions of clustering

- Clustered file organization
- Clustered relation
- Clustering index

Example  \( R_1 \Join R_2 \) over common attribute \( C \)

\[
\begin{align*}
T(R_1) & = 10,000 \\
T(R_2) & = 5,000 \\
S(R_1) & = S(R_2) = 1/10 \text{ block} \\
\text{Memory available} & = 101 \text{ blocks} \\
\text{Metric: # of IOs} & \text{ (ignoring writing of result)}
\end{align*}
\]
Caution!

This may not be the best way to compare
- ignoring CPU costs
- ignoring timing
- ignoring double buffering requirements

Options

- Transformations: $R_1 \bowtie R_2$, $R_2 \bowtie R_1$
- Joint algorithms:
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join

• Iteration join (conceptually)
  for each $r \in R_1$ do
    for each $s \in R_2$ do
      if $r.C = s.C$ then output $r,s$ pair
• Merge join (conceptually)
  (1) if R1 and R2 not sorted, sort them
  (2) \(i \leftarrow 1; j \leftarrow 1;\)
     \[\text{While } (i \leq T(R1)) \land (j \leq T(R2)) \text{ do}\]
     \[\text{if } R1\{i\}.C = R2\{j\}.C \text{ then outputTuples}\]
     \[\text{else if } R1\{i\}.C > R2\{j\}.C \text{ then } j \leftarrow j+1\]
     \[\text{else if } R1\{i\}.C < R2\{j\}.C \text{ then } i \leftarrow i+1\]

Procedure Output-Tuples
\[\text{While } (R1\{i\}.C = R2\{j\}.C) \land (i \leq T(R1)) \text{ do}\]
\[jj \leftarrow j;\]
\[\text{while } (R1\{i\}.C = R2\{jj\}.C) \land (jj \leq T(R2)) \text{ do}\]
\[\text{[output pair } R1\{i\}, R2\{jj\};\]
\[jj \leftarrow jj+1 \]
\[i \leftarrow i+1 \]

Example
\[
\begin{array}{|c|c|c|c|}
\hline
i & R1\{i\}.C & R2\{j\}.C & j \\
\hline
1 & 10 & 5 & 1 \\
2 & 20 & 20 & 2 \\
3 & 20 & 20 & 3 \\
4 & 30 & 30 & 4 \\
5 & 40 & 30 & 5 \\
& & 50 & 6 \\
& & 52 & 7 \\
\hline
\end{array}
\]
• Join with index (Conceptually)

For each \( r \in R_1 \)

\[ X \leftarrow \text{index}\left( R_2, C_r, r.C \right) \]

for each \( s \in X \)

output \( r,s \) pair

Note: \( X \leftarrow \text{index}(\text{rel}, \text{attr}, \text{value}) \)

then \( X = \) set of rel tuples with attr = value

• Hash join (conceptual)

- Hash function \( h \), range 0 \( \rightarrow \) \( k \)
- Buckets for \( R_1 \): \( G_0, G_1, ... G_k \)
- Buckets for \( R_2 \): \( H_0, H_1, ... H_k \)

Algorithm

(1) Hash \( R_1 \) tuples into \( G \) buckets
(2) Hash \( R_2 \) tuples into \( H \) buckets
(3) For \( i = 0 \) to \( k \) do

match tuples in \( G_i, H_i \) buckets

Simple example  hash: even/odd

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>Even Buckets</th>
<th>Odd Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>2 4 8</td>
<td>3 5 9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4 8 14</td>
<td>5 3 13 11</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>2 4 8</td>
<td>4 12 8 14</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4 8 14</td>
<td>5 3 13 11</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Factors that affect performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Example 1(a) Iteration Join \( R_1 \times R_2 \)

- Relations not contiguous
- Recall
  \[
  T(R_1) = 10,000 \quad T(R_2) = 5,000
  \]
  \[
  S(R_1) = S(R_2) = \frac{1}{10} \text{ block}
  \]
  \[
  \text{MEM} = 101 \text{ blocks}
  \]

Cost: for each \( R_1 \) tuple:

\[
[\text{Read tuple} + \text{Read } R_2]
\]

Total \( = 10,000 \times [1 + 5000] = 50,010,000 \) IOs

Can we do better?

Use our memory

(1) Read 100 blocks of \( R_1 \)
(2) Read all of \( R_2 \) (using 1 block) + join
(3) Repeat until done
Cost: for each R1 chunk:
- Read chunk: 1000 IOs
- Read R2: 5000 IOs
  6000

Total = \( \frac{10,000 \times 6000}{1,000} = 60,000 \) IOs

Can we do better?

- Reverse join order: \( R2 \bowtie R1 \)

Total = \( \frac{5000 \times (1000 + 10,000)}{1000} \)
  \( 5 \times 11,000 = 55,000 \) IOs

Example 1(b) Iteration Join \( R2 \bowtie R1 \)
- Relations contiguous

Cost
- For each R2 chunk:
  - Read chunk: 100 IOs
  - Read R1: 1000 IOs
  1,100

Total = 5 chunks \( \times 1,100 = 5,500 \) IOs
Example 1(c)  Merge Join

- Both R1, R2 ordered by C; relations contiguous

\[
\begin{array}{c}
\text{Memory} \\
\hline
\text{R1} & \text{...} & \text{R1} \\
\text{R2} & \text{...} & \text{R2}
\end{array}
\]

Total cost: Read R1 cost + read R2 cost
\[= 1000 + 500 = 1,500 \text{ I/Os}\]

Example 1(d)  Merge Join

- R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first.... HOW?

One way to sort:  Merge Sort

(i) For each 100 blk chunk of R:
- Read chunk
- Sort in memory
- Write to disk

\[
\begin{array}{c}
\text{R1} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} \\
\text{R2} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} \\
\text{Memory} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} \\
\text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} \\
\text{sorted chunks} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots}
\end{array}
\]
(ii) Read all chunks + merge + write out

Cost: Sort
   Each tuple is read, written,
   read, written
so...
Sort cost R1: 4 x 1,000 = 4,000
Sort cost R2: 4 x 500 = 2,000

Example 1(d) Merge Join (continued)
R1, R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500 = 7,500 IOs

But: Iteration cost = 5,500
so merge joint does not pay off!
But say \( R_1 = 10,000 \text{ blocks} \) contiguous
\( R_2 = 5,000 \text{ blocks} \) not ordered

Iterate:
\[
5000 \times \frac{(100+10,000)}{100} = 50 \times 10,100 = 505,000 \text{ IOs}
\]

Merge join:
\[
5(10,000+5,000) = 75,000 \text{ IOs}
\]

Merge Join (with sort) WINS!

How much memory do we need for merge sort?
E.g.: Say I have 10 memory blocks
\[
\begin{array}{c}
10 \\
R_1 \\
\end{array}
\]
100 chunks \(\Rightarrow\) to merge, need 100 blocks!

In general:
Say \( k \text{ blocks in memory} \)
\( x \text{ blocks for relation sort} \)
\ # chunks = \( (x/k) \) size of chunk = \( k \)
\ # chunks \(\leq\) buffers available for merge
so... \( (x/k) \leq k \)
or \( k^2 \geq x \) or \( k \geq \sqrt{x} \)
In our example

R1 is 1000 blocks, \( k \geq 31.62 \)
R2 is 500 blocks, \( k \geq 22.36 \)

Need at least 32 buffers

Can we improve on merge join?

Hint: do we really need the fully sorted files?

Cost of improved merge join:

\[
C = \text{Read R1} + \text{write R1 into runs} + \text{read R2} + \text{write R2 into runs} + \text{join} = 2000 + 1000 + 1500 = 4500
\]

--- Memory requirement?
Example 1(e) Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered
- Assume R1.C index fits in memory

Cost: Reads: 500 IOs
for each R2 tuple:
- probe index - free
- if match, read R1 tuple: 1 IO

What is expected # of matching tuples?
(a) say R1.C is key, R2.C is foreign key
   then expect = 1
(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 alternate assumption
\[ \text{Expect} = \frac{10,000}{1,000,000} = \frac{1}{100} \]

Total cost with index join

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

(b) Total cost = \( 500 + 5000(2)1 \) = 10,500

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

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 probe is
\[ \text{E} = \frac{(0)99 + (1)101}{200} \approx 0.5 \]
Total cost (including probes)

\[ = 500 + 5000 \text{ [Probe + get records]} \]
\[ = 500 + 5000 \times 0.5 + 2 \text{ [uniform assumption]} \]
\[ = 500 + 12,500 = 13,000 \text{ (case b)} \]

For case (c):
\[ = 500 + 5000 \times 0.5 \times 1 + (1/100) \times 1 \]
\[ = 500 + 2500 + 50 = 3050 \]

So far

<table>
<thead>
<tr>
<th>Iteration</th>
<th>R2 (\times) R1</th>
<th>55,000 (best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not contiguous</td>
<td>Merge Join</td>
<td>_______</td>
</tr>
<tr>
<td></td>
<td>Sort + Merge Join</td>
<td>_______</td>
</tr>
<tr>
<td></td>
<td>R1.C Index</td>
<td>_______</td>
</tr>
<tr>
<td></td>
<td>R2.C Index</td>
<td>_______</td>
</tr>
<tr>
<td>Contiguous</td>
<td>Iterate R2 (\times) R1</td>
<td>5500</td>
</tr>
<tr>
<td></td>
<td>Merge Join</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>Sort + Merge Join</td>
<td>7500 (\rightarrow) 4500</td>
</tr>
<tr>
<td></td>
<td>R1.C Index</td>
<td>5500 (\rightarrow) 3050 (\rightarrow) 550</td>
</tr>
<tr>
<td></td>
<td>R2.C Index</td>
<td>_______</td>
</tr>
</tbody>
</table>

Example 1(f) Hash Join

- R1, R2 contiguous (un-ordered)
  - Use 100 buckets
  - Read R1, hash, + write buckets

R1 → [Hash Join] → [Sort + Merge Join] → [Merge Join] → [R2.C Index] → [R1.C Index]
Chapters 15-16b

Same for R2
Read one R1 bucket; build memory hash table
Read corresponding R2 bucket + hash probe

R1 → memory → R2

Then repeat for all buckets

Cost:

"Bucketize:" Read R1 + write
Read R2 + write
Join: Read R1, R2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum memory requirements:

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

\[ k = \text{number of memory buffers} \]
\[ x = \text{number of R1 blocks} \]

So...

\( \frac{x}{k} < k \)

\[ k > \sqrt{x} \]

need: k+1 total memory buffers
Trick: keep some buckets in memory

E.g., $k' = 33$

- $R_1$ buckets = 31 blocks
- keep 2 in memory

<table>
<thead>
<tr>
<th>Memory use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_0$</td>
</tr>
<tr>
<td>$G_1$</td>
</tr>
<tr>
<td>Output</td>
</tr>
<tr>
<td>$R_1$ input</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

6 buffers to spare!!

called hybrid hash-join

Next: Bucketize $R_2$

- $R_2$ buckets = $500/33 = 16$ blocks
- Two of the $R_2$ buckets joined immediately with $G_0, G_1$

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?

Another hash join trick:
- Only write into buckets <val, 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

Total cost = Read R2: 500
Read R1: 1000
Get tuples: 100
1600

So far:
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate</td>
<td>5500</td>
</tr>
<tr>
<td>Merge join</td>
<td>1500</td>
</tr>
<tr>
<td>Sort+merge joint</td>
<td>7500</td>
</tr>
<tr>
<td>R1.C index</td>
<td>5500  → 550</td>
</tr>
<tr>
<td>R2.C index</td>
<td></td>
</tr>
<tr>
<td>Build R.C index</td>
<td></td>
</tr>
<tr>
<td>Build S.C index</td>
<td></td>
</tr>
<tr>
<td>Hash join, pointers</td>
<td>1600</td>
</tr>
<tr>
<td>Hash join, trick, R1 first</td>
<td>4414</td>
</tr>
<tr>
<td>Hash join, points</td>
<td></td>
</tr>
</tbody>
</table>

Summary
- Iteration ok for "small" relations (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, hash join usually best
• 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)

Join strategies for parallel processors
Later on....

Chapter 7 summary
• Relational algebra level
• Detailed query plan level
  – Estimate costs
  – Generate plans
  • Join algorithms
  – Compare costs