Query Optimization

--> Generating and comparing plans

Query

- Generate
- Pruning
- Estimate Cost
- Select

Plans

Cost

Pick Min

Chris Clifton - CS54100
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:

- 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:

\[ B(R) = \# \text{ of blocks containing } R \text{ tuples} \]
\[ f(R) = \max \# \text{ of tuples of } R \text{ per block} \]
\[ M = \# \text{ memory blocks available} \]

\[ HT(i) = \# \text{ levels in index } i \]
\[ LB(i) = \# \text{ of leaf blocks in index } i \]
Clustering index
Index that allows tuples to be read in an order that corresponds to physical order

Notions of clustering

• Clustered file organization

• Cluster on

• Cluster on
Example

$R_1 \bowtie R_2$ over common attribute $C$

- $T(R_1) = 10,000$
- $T(R_2) = 5,000$
- $S(R_1) = S(R_2) = 1/10$ block
- Memory available = 101 blocks

→ Metric: # of IOs
   (ignoring writing of result)
Caution!
This may not be the best way to compare
- ignoring CPU costs
- ignoring timing
- ignoring double buffering requirements

Options
- Transformations: R1 \( \bowtie \) R2, R2 \( \bowtie \) R1
- Join algorithms:
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join
• Nested Loop 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 \( R_1 \) and \( R_2 \) not sorted, sort them
      \((n \log n - \text{but I/O cost?})\)
  (2) \( i \leftarrow 1; j \leftarrow 1; \)
      While \((i \leq T(R_1)) \land (j \leq T(R_2))\) do
        if \( R_1\{ i \}.C = R_2\{ j \}.C \) then outputTuples
        else if \( R_1\{ i \}.C > R_2\{ j \}.C \) then \( j \leftarrow j+1 \)
        else if \( R_1\{ i \}.C < R_2\{ j \}.C \) then \( i \leftarrow i+1 \)
Procedure Output-Tuples

While (R1{ i }.C = R2{ j }.C) ∧ (i ≤ T(R1)) do

[jj ← j; 
while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do 

[output pair R1{ i }, R2{ jj }; 
jj ← jj+1 ]

i ← 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>
• Join with index (Conceptually)

For each \( r \in R1 \) do

\[ X \leftarrow \text{index (} R2, C, r.C) \]

for each \( s \in X \) do

output \( r,s \) pair

Assume \( R2.C \) index

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

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

• Hash join (conceptual)

– Hash function \( h \), range \( 0 \rightarrow k \)
– Buckets for \( R1: G0, G1, \ldots Gk \)
– Buckets for \( R2: H0, H1, \ldots Hk \)

Algorithm

(1) Hash \( R1 \) tuples into \( G \) buckets
(2) Hash \( R2 \) tuples into \( H \) buckets
(3) For \( i = 0 \) to \( k \) do

match tuples in \( Gi, Hi \) buckets
Simple example  hash: even/odd

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

- Relations **not** contiguous
- Recall
  \[ T(R1) = 10,000 \quad T(R2) = 5,000 \]
  \[ S(R1) = S(R2) = \frac{1}{10} \text{ block} \]
  \[ \text{MEM} = 101 \text{ blocks} \]

**Cost:** for each R1 tuple:

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

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

---

Can we do better?

**Use our memory**

1. Read 100 blocks of R1
2. Read all of R2 (using 1 block) + join
3. Repeat until done
Cost: for each R1 chunk:

Read chunk: 1000 IOs

Read R2 5000 IOs

6000

Total = \[ \frac{10,000 \times 6000}{1,000} = 60,000 \text{ IOs} \]

• Can we do better?

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

Total = \(\frac{5000}{1000} \times (1000 + 10,000) = 5 \times 11,000 = 55,000 \text{ IOs} \)
Example 1(b)  Iteration Join  \( R_2 \bowtie R_1 \)

- Relations contiguous

**Cost**
For each \( R_2 \) chunk:
- Read chunk: 100 IOs
- Read \( R_1 \): \( 1000 \) IOs

Total = 5 chunks \( \times 1100 = 5500 \) IOs

Example 1(c)  Merge Join

- Both \( R_1, R_2 \) ordered by \( C \); relations contiguous

**Memory**

Total cost: Read \( R_1 \) cost + read \( R_2 \) cost
\[ = 1000 + 500 = 1500 \text{ IOs} \]
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 block chunk of R:

- Read chunk
- Sort in memory
- Write to disk

R1

R2

Memory

sorted chunks

© 2012 Chris Clifton
(ii) Read all chunks + merge + write out

Cost: Sort

Each tuple is read, written, read, written

so...

Sort cost R1: $4 \times 1,000 = 4,000$
Sort cost R2: $4 \times 500 = 2,000$
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
R1 = 10,000 blocks  contiguous
R2 = 5,000 blocks  not ordered

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

Merge join: \( 5(10,000+5,000) = 75,000 \) IOs

Merge Join (with sort) WINS!
How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks

100 chunks ⇒ to merge, need 100 blocks!

In general:

Say \( k \) blocks in memory

\( x \) blocks for relation sort

\# chunks = \( \frac{x}{k} \) size of chunk = \( k \)

\# chunks ≤ buffers available for merge

so... \( \frac{x}{k} \) ≤ \( k \)

or \( k^2 ≥ x \) or \( k ≥ \sqrt{x} \)
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:

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

--> 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 $\text{DOM}(R_1, C)=1,000,000$

$T(R_1) = 10,000$

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

\[ E = \frac{(0)99}{200} + \frac{(1)101}{200} \approx 0.5 \]

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):

\[ = 500+5000 \times 0.5 \times 1 + (1/100) \times 1 \]
\[ = 500+2500+50 = 3050 \text{ IOs} \]
<table>
<thead>
<tr>
<th></th>
<th>not contiguous</th>
<th>contiguous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate R2 ⊙ R1</td>
<td>55,000 (best)</td>
<td>5500</td>
</tr>
<tr>
<td>Merge Join</td>
<td>__________</td>
<td>1500</td>
</tr>
<tr>
<td>Sort+ Merge Join</td>
<td>__________</td>
<td>4500</td>
</tr>
<tr>
<td>R1.C Index</td>
<td>__________</td>
<td>7500 → 4500</td>
</tr>
<tr>
<td>R2.C Index</td>
<td>__________</td>
<td>5500 → 3050→ 550</td>
</tr>
</tbody>
</table>
Example 1(f) Hash Join

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

R1 →

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

Then repeat for all buckets
Cost:

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

Join: Read R1, R2

Total cost = 3 \times [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 = number of memory buffers
x = number of R1 blocks

So... \left(\frac{x}{k}\right) < k

k > \sqrt{x} \quad \text{need: k+1 total memory buffers}
Trick: keep some buckets in memory

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

Memory use:
- \( G_0 \): 31 buffers  
- \( G_1 \): 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 = \( \frac{500}{33} \approx 16 \) blocks  
- Two of the R2 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+31×31=1961
• To bucketize R2, only write 31 buckets: so, cost = 500+31×16=996
• To compare join (2 buckets already done) read 31×31+31×16=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 \(<\text{val,ptr}>\) pairs/block
- expected number of result tuples is 100

- Build hash table for R2 in memory
  5000 tuples \(\rightarrow\) \(5000/100 = 50\) blocks

- Read R1 and match
- Read \(~ 100\) R2 tuples

Total cost =
<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read R2</td>
<td>500</td>
</tr>
<tr>
<td>Read R1</td>
<td>1000</td>
</tr>
<tr>
<td>Get tuples</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1600</td>
</tr>
</tbody>
</table>

So far:

<table>
<thead>
<tr>
<th>Operation</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 (\rightarrow) 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</td>
<td>4500+</td>
</tr>
<tr>
<td>Hash join, pointers</td>
<td>1600</td>
</tr>
<tr>
<td>with trick,R1 first</td>
<td>4414</td>
</tr>
<tr>
<td>with trick,R2 first</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)