CS 277: Database System Implementation

Notes 7: Query Optimization

Arthur Keller
Query Optimization

--> Generating and comparing plans

Query Optimization

Generate 
Pruning 
Estimate Cost 
Select 

Generate 
Prune x 
Prune x 
Select 

Pick Min

Cost

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:

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

\[ HT(i) = \# \text{ levels in index } i \]
\[ LB(i) = \# \text{ of leaf blocks in index } i \]
Clustering index

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

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

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

- Clustering index
Example \( R_1 \bowtie R_2 \) over common attribute \( C \)

\[
\begin{align*}
T(R1) &= 10,000 \\
T(R2) &= 5,000 \\
S(R1) &= S(R2) = 1/10 \text{ block} \\
\text{Memory available} &= 101 \text{ blocks}
\end{align*}
\]

\( \rightarrow \) 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
- Joint algorithms:
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join
• **Iteration join** (conceptually)
  
  for each \( r \in R1 \) do

  for each \( s \in R2 \) do

  if \( r.C = s.C \) then output \( r,s \) pair
• Merge join (conceptually)
  (1) if R1 and R2 not sorted, sort them
  (2) \( i \leftarrow 1; j \leftarrow 1; \)

  While \( (i \leq T(R1)) \land (j \leq T(R2)) \) do
    if \( R1\{ i \}.C = R2\{ j \}.C \) then OutputTuples
    else if \( R1\{ i \}.C > R2\{ j \}.C \) then \( j \leftarrow j+1 \)
    else if \( R1\{ i \}.C < R2\{ j \}.C \) then \( i \leftarrow i+1 \)
Procedure OutputTuples

While (R1{ i }.C = R2{ j }.C) \land (i \leq T(R1)) do
    [jj ← j;
    while (R1{ i }.C = R2{ jj }.C) \land (jj \leq 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

\[
[ \ X \leftarrow \text{index}(R2, C, r.C) \\
\quad \text{for each } s \in X \text{ do} \\
\quad \text{output } r,s \text{ pair} ]
\]

Assume R2.C index

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

then \( X = \text{set of rel tuples with attr = value} \)
• Hash join (conceptual)
  – Hash function \( h \), range \( 0 \rightarrow k \)
  – Buckets for \( R_1 \): \( G_0, G_1, \ldots, G_k \)
  – Buckets for \( R_2 \): \( H_0, H_1, \ldots, H_k \)

**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>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Even</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Odd:</strong></td>
</tr>
<tr>
<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 R1 \( \bowtie \) R2

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

Cost: for each R1 tuple:

\[
[\text{Read tuple} + \text{Read R2}] \\
\text{Total} = 10,000 [1 + 5000] = 50,010,000 \text{ IOs}
\]
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

Total = \frac{10,000 \times 6000}{1,000} = 60,000 IOs
• Can we do better?

* Reverse join order:  R2 $\bowtie$ R1

Total = \( \frac{5000}{1000} \times (1000 + 10,000) = 5 \times 11,000 = 55,000 \) IOs
Example 1(b) Iteration Join $R2 \bowtie R1$

- Relations contiguous

Cost
For each R2 chunk:
  Read chunk: 100 IOs
  Read R1: $1000$ IOs
  \[ \frac{1000}{1} = 1,100 \]

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

- Both R1, R2 ordered by C; relations contiguous

Total cost: Read R1 cost + read R2 cost
= 1000 + 500 = 1,500 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 blk chunk of R:
- Read chunk
- Sort in memory
- Write to disk
(ii) Read all chunks + merge + write out

Sorted file

Memory

Sorted
Chunks
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  \( R_1 = 10,000 \) blocks  contiguous
\( R_2 = 5,000 \) blocks  not ordered

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

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

\[ \text{100 chunks} \Rightarrow \text{to merge, need 100 blocks!} \]
In general:

Say k blocks in memory
x blocks for relation sort

# chunks = (x/k)  size of chunk = k

# chunks ≤ buffers available for merge

so...  (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?

R1

R2

sorted runs

Join?
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
  \[
  E = \frac{(0) \cdot 99}{200} + \frac{(1) \cdot 101}{200} \approx 0.5
  \]
Total cost (including probes)

\[
500 + 5000 \quad [\text{Probe + get records}]
\]

\[
= 500 + 5000 \times [0.5 + 2] \quad \text{uniform assumption}
\]

\[
= 500 + 12,500 = 13,000 \quad \text{(case b)}
\]

For case (c):

\[
= 500 + 5000 \times [0.5 \times 1 + (1/100) \times 1]
\]

\[
= 500 + 2500 + 50 = 3050 \text{ IOs}
\]
So far

not contiguous

\[
\begin{align*}
\text{Iterate } R2 & \bowtie R1 & 55,000 \text{ (best)} \\
\text{Merge Join} & & \\
\text{Sort+ Merge Join} & & \\
\text{R1.C Index} & & \\
\text{R2.C Index} & & \\
\end{align*}
\]

contiguous

\[
\begin{align*}
\text{Iterate } R2 & \bowtie R1 & 5500 \\
\text{Merge join} & & 1500 \\
\text{Sort+Merge Join} & & 7500 \rightarrow 4500 \\
\text{R1.C Index} & & 5500 \rightarrow 3050 \rightarrow 550 \\
\text{R2.C Index} & & \\
\end{align*}
\]
Example 1(f) Hash Join

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

\[ \text{R1} \rightarrow \text{10 blocks} \]

100
-> 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 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 = \( \frac{x}{k} \)

- \( k \) = number of memory buffers
- \( x \) = number of R1 blocks

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

\[ k > \sqrt{x} \quad \text{need: } k+1 \text{ total memory buffers} \]
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 = 500/33 = 16 blocks
– Two of the R2 buckets joined immediately with G0,G1
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?

* See textbook for answer...
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

\[
\text{Total cost} = \begin{array}{c}
\text{Read R2:} \\
\text{Read R1:} \\
\text{Get tuples:}
\end{array} \begin{array}{c}
500 \\
1000 \\
100 \\
\hline
1600
\end{array}
\]
So far:

- Iterate 5500
- Merge join 1500
- Sort+merge joint 7500
- R1.C index 5500 \rightarrow 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
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 16 summary

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