CS 277: Database System Implementation

Notes 7: Query Optimization

Arthur Keller

To generate plans consider:
- Transforming relational algebra expression (e.g. order of joins)
- Use of existing indexes
- Building indexes or sorting on the fly

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
10
15
17
19
35
37
```

**Notions of clustering**

- Clustered file organization
  
- Clustered relation
  
- Clustering index

**Example**

R1 \( \bowtie \) R2 over common attribute C

T(R1) = 10,000  
T(R2) = 5,000  
S(R1) = S(R2) = 1/10 block  
Memory available = 101 blocks

\[ \rightarrow \text{Metric: } \# \text{ of IOs} \]  
(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)**

\[
\text{for each } r \in R1 \text{ do} \\
\text{for each } s \in R2 \text{ do} \\
\text{if } r.C = s.C \text{ then output } r, s \text{ pair}
\]
• 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 OutputTuples
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>50</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

• Join with index (Conceptually)
For each r ∈ R1 do
    X ← index (R2, C, r.C)
    for each s ∈ X do
        output r,s pair

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

• Hash join (conceptually)
- Hash function h, range 0 → k
- Buckets for R1: G0, G1, ..., Gk
- Buckets for R2: H0, H1, ..., 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>Even</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td>2 4 8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>4 12 8 14</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td>R1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td>R2</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td></td>
<td>3 5 9</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td></td>
<td>5 3 13 11</td>
</tr>
<tr>
<td>14</td>
<td></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 \( R_1 \bowtie R_2 \)

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

Cost: for each \( R_1 \) tuple:
   \[ \text{[Read tuple + Read R2]} \]
Total \( = 10,000 \left[ \frac{1+5000}{101} \right] = 50,010,000 \) 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 \( R_1 \) chunk:
   \( \text{Read chunk: 1000 IOs} \)
   \( \text{Read R2} \)
   \( 5000 \) IOs
   \( 6000 \)
Total \( = 10,000 \times 6000 = 60,000 \) IOs

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

Example 1(b)  Iteration Join \( R_2 \bowtie R_1 \)

- Relations contiguous

Cost
For each \( R_2 \) chunk:
   \( \text{Read chunk: 100 IOs} \)
   \( \text{Read R1: 1000 IOs} \)
   \( 1,100 \)
Total \( = 5 \text{ chunks } \times 1,100 = 5,500 \) IOs
Example 1(c) Merge Join

- Both R1, R2 ordered by C; relations contiguous

Memory

\[
\begin{array}{c}
R1 \\
R2
\end{array}
\quad
\begin{array}{c}
\ldots \quad R1 \\
\ldots \quad R2
\end{array}
\]

Total cost: Read R1 cost + read R2 cost

\[1000 + 500 = 1,500\ \text{IOs}\]

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 blk chunk of R:
- Read chunk
- Sort in memory
- Write to disk

\[
\begin{array}{c}
R1 \\
R2
\end{array}
\quad
\begin{array}{c}
\ldots
\end{array}
\]

\begin{align*}
\text{Memory} & \quad \text{sorted chunks} \\
\end{align*}

(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 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

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

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:**  
\[
5000 \times \frac{100+10,000}{100} = 50 \times 10,100
\]

= 505,000 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

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 \(\leq\) buffers available for merge

so... \(\frac{x}{k} \leq k\)  
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
(c) Say DOM(R1, C)=1,000,000
T(R1) = 10,000
with alternate assumption
Expect = 10,000/1,000,000 = 1

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

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

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

So far

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (IOs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate R1 ( \times ) R1</td>
<td>55,000</td>
</tr>
<tr>
<td>Merge Join</td>
<td>1500</td>
</tr>
<tr>
<td>Sort + Merge Join</td>
<td>7500</td>
</tr>
<tr>
<td>R1.C Index</td>
<td>5500</td>
</tr>
<tr>
<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

\[ \text{R1} \rightarrow \text{10 blocks} \rightarrow \text{100} \rightarrow \]

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

So...
\[ \frac{x}{k} < 1 \]

\[ k > \sqrt{\frac{x}{k}} \]
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:
- $G0$: 31 buffers
- $G1$: 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

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

So far:

- Iterate 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+
- 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

Join strategies for parallel processors

- 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)

Chapter 16 summary

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