Join Trees

A join tree is a binary tree with
- join operators as inner nodes
- relations as leaf nodes

Algorithms will produce different kinds of join trees
- ordered or unordered
- with cross products or without

The most common case is ordered, without cross products
Shape of Join Trees

Commonly used classes of join trees:

- left-deep tree
- right-deep tree
- zigzag tree
- bushy tree

The first three are summarized as *linear trees*. 
Join Selectivity

Input:
- cardinalities $|R_i|$
- selectivities $f_{i,j}$: if $p_{i,j}$ is the join predicate between $R_i$ and $R_j$, define

$$f_{i,j} = \frac{|R_i \times_{p_{i,j}} R_j|}{|R_i \times R_j|}$$

Calculate:
- result cardinality:

$$|R_i \times_{p_{i,j}} R_j| = f_{i,j} |R_i| |R_j|$$

Rational: The selectivity can be computed/estimated easily (ideally).
Cardinality of Join Trees

Given a join tree $T$, the result cardinality $|T|$ can be computed recursively as

$$|T| = \begin{cases} 
|R_i| & \text{if } T \text{ is a leaf } R_i \\
(\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) |T_1||T_2| & \text{if } T = T_1 \Join T_2
\end{cases}$$

- allows for easy calculation of join cardinality
- requires only base cardinalities and selectivities
- assumes independence of the predicates
Sample Statistics

As running example, we use the following statistics:

\[ |R_1| = 10 \]
\[ |R_2| = 100 \]
\[ |R_3| = 1000 \]
\[ f_{1,2} = 0.1 \]
\[ f_{2,3} = 0.2 \]

- implies query graph \( R_1 - R_2 - R_3 \)
- assume \( f_{i,j} = 1 \) for all other combinations
A Basic Cost Function

Given a join tree $T$, the cost function $C_{out}$ is defined as

$$C_{out}(T) = \begin{cases} 
0 & \text{if } T \text{ is a leaf } R_i \\
|T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \Join T_2 
\end{cases}$$

- sums up the sizes of the (intermediate) results
- rational: larger intermediate results cause more work
- we ignore the costs of single relations as they have to be read anyway
Basic Join Specific Cost Functions

For single joins:

\[ C_{nlj}(e_1 \bowtie e_2) = |e_1||e_2| \]
\[ C_{hj}(e_1 \bowtie e_2) = 1.2|e_1| \]
\[ C_{smj}(e_1 \bowtie e_2) = |e_1| \log(|e_1|) + |e_2| \log(|e_2|) \]

For sequences of join operators \( s = s_1 \bowtie \ldots \bowtie s_n \):

\[ C_{nlj}(s) = \sum_{i=2}^{n} |s_1 \bowtie \ldots \bowtie s_{i-1}| |s_i| \]
\[ C_{hj}(s) = \sum_{i=2}^{n} 1.2|s_1 \bowtie \ldots \bowtie s_{i-1}| \]
\[ C_{smj}(s) = \sum_{i=2}^{n} |s_1 \bowtie \ldots \bowtie s_{i-1}| \log(|s_1 \bowtie \ldots \bowtie s_{i-1}|) + \sum_{i=2}^{n} |s_i| \log(|s_i|) \]
Remarks on the Basic Cost Functions

- cost functions are simplistic
- algorithms are modelled very simplified (e.g. 1.2, no n-way sort etc.)
- designed for left-deep trees
- $C_{hj}$ and $C_{smj}$ do not work for cross products (fix: take output cardinality then, which is $C_{nl}$)
- in reality: other parameters than cardinality play a role
- cost functions assume the same join algorithm for the whole join tree
## Sample Cost Calculations

<table>
<thead>
<tr>
<th></th>
<th>$C_{out}$</th>
<th>$C_{nl}$</th>
<th>$C_{hj}$</th>
<th>$C_{smj}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \times R_2$</td>
<td>100</td>
<td>1000</td>
<td>12</td>
<td>697.61</td>
</tr>
<tr>
<td>$R_2 \times R_3$</td>
<td>20000</td>
<td>100000</td>
<td>120</td>
<td>10630.26</td>
</tr>
<tr>
<td>$R_1 \times R_3$</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000.00</td>
</tr>
<tr>
<td>$(R_1 \times R_2) \times R_3$</td>
<td>20100</td>
<td>101000</td>
<td>132</td>
<td>11327.86</td>
</tr>
<tr>
<td>$(R_2 \times R_3) \times R_1$</td>
<td>40000</td>
<td>300000</td>
<td>24120</td>
<td>32595.00</td>
</tr>
<tr>
<td>$(R_1 \times R_3) \times R_2$</td>
<td>30000</td>
<td>1010000</td>
<td>22000</td>
<td>143542.00</td>
</tr>
</tbody>
</table>

- costs differ vastly between join trees
- different cost functions result in different costs
- the cheapest plan is always the same here, but relative order varies
- join trees with cross products are expensive
- join order is essential under all cost functions
More Examples

For the query $|R_1| = 1000, |R_2| = 2, |R_3| = 2, f_{1,2} = 0.1, f_{1,3} = 0.1$ we have costs:

<table>
<thead>
<tr>
<th></th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \bowtie R_2$</td>
<td>200</td>
</tr>
<tr>
<td>$R_2 \times R_3$</td>
<td>4</td>
</tr>
<tr>
<td>$R_1 \bowtie R_3$</td>
<td>200</td>
</tr>
<tr>
<td>$(R_1 \bowtie R_2) \bowtie R_3$</td>
<td>240</td>
</tr>
<tr>
<td>$(R_2 \times R_3) \bowtie R_1$</td>
<td>44</td>
</tr>
<tr>
<td>$(R_1 \bowtie R_3) \bowtie R_2$</td>
<td>240</td>
</tr>
</tbody>
</table>

- here cross product is best
- but relies on the small sizes of $|R_2|$ and $|R_3|$  
- attractive if the cardinality of one relation is small
More Examples (2)

For the query $|R_1| = 10, |R_2| = 20, |R_3| = 20, |R_4| = 10$, $f_{1,2} = 0.01$, $f_{2,3} = 0.5$, $f_{3,4} = 0.01$

we have costs:

<table>
<thead>
<tr>
<th>Join Tree</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \Join R_2$</td>
<td>2</td>
</tr>
<tr>
<td>$R_2 \Join R_3$</td>
<td>200</td>
</tr>
<tr>
<td>$R_3 \Join R_4$</td>
<td>2</td>
</tr>
<tr>
<td>$((R_1 \Join R_2) \Join R_3) \Join R_4$</td>
<td>24</td>
</tr>
<tr>
<td>$((R_2 \Join R_3) \Join R_1) \Join R_4$</td>
<td>222</td>
</tr>
<tr>
<td>$(R_1 \Join R_2) \Join (R_3 \Join R_4)$</td>
<td>6</td>
</tr>
</tbody>
</table>

- covers all join trees due to the symmetry of the query
- the bushy tree is better than all join trees
Symmetry and ASI

- cost function $C_{impl}$ is called symmetric if $C_{impl}(e_1 \boxtimes_{impl} e_2) = C_{impl}(e_2 \boxtimes_{impl} e_1)$
- for symmetric cost functions commutativity can be ignored
- ASI: adjacent sequence interchange (see IKKBZ algorithm for a definition)

Our basic cost functions can be classified as:

<table>
<thead>
<tr>
<th></th>
<th>ASI</th>
<th>¬ASI</th>
</tr>
</thead>
<tbody>
<tr>
<td>symmetric</td>
<td>$C_{out}$</td>
<td>$C_{smj}$</td>
</tr>
<tr>
<td>¬symmetric</td>
<td>$C_{hj}$</td>
<td>-</td>
</tr>
</tbody>
</table>

- more complex cost functions are usually ¬ASI, often also ¬symmetric
- symmetry and especially ASI can be exploited during optimization