Maximum Value Precedence Algorithm

- greedy heuristics can produce poor results
- IKKBZ only support acyclic queries and ASI cost functions
- Maximum Value Precedence (MVP) [4] algorithm is a polynomial time heuristic with good results
- considers join ordering a graph theoretic problem
Directed Join Graph

Given a conjunctive query with predicates $P$.

- for all join predicates $p \in P$, we denote by $\mathcal{R}(p)$ the relations whose attributes are mentioned in $p$.
- the \textit{directed join graph} of the query is a triple $G = (V, E_p, E_v)$, where $V$ is the set of predicates and $E_p$ and $E_v$ are sets of directed edges defined as follows
  - for any nodes $u, v \in V$, if $\mathcal{R}(u) \cap \mathcal{R}(v) \neq \emptyset$ then $(u, v) \in E_p$ and $(v, u) \in E_p$
  - if $\mathcal{R}(u) \cap \mathcal{R}(v) = \emptyset$ then $(u, v) \in E_v$ and $(v, u) \in E_v$
  - edges in $E_p$ are called \textit{physical edges}, those in $E_v$ \textit{virtual edges}

Note: all nodes $u, v$ there is an edge $(u, v)$ that is either physical or virtual. Hence, $G$ is a clique.
Examples: Spanning Tree and Join Tree

- every spanning tree in the directed join graph leads to a join tree

query graph

\[ R_1 - R_2 - R_3 - R_4 \]

spanning tree I

\[ p_{1,2} \]

\[ p_{2,3} \]

\[ p_{3,4} \]

\[ \]

directed join graph

\[ p_{1,2} \]

\[ p_{2,3} \]

\[ p_{3,4} \]

\[ R_4 \]

\[ R_3 \]

\[ R_2 \]

\[ R_1 \]

join tree I
Examples: Spanning Tree and Join Tree (2)
Examples: Spanning Tree and Join Tree (3)

- spanning tree does not correspond to a (effective) join tree!
Effective Spanning Trees

It can be shown that a spanning tree $T = (V, E)$ is *effective*, it is satisfies the following conditions:

1. $T$ is a binary tree
2. for all inner nodes $v$ and nodes $u$ with $(u, v) \in E$: $\mathcal{R}(T(u)) \cap \mathcal{R}(v) \neq \emptyset$
3. for all nodes $v, u_1, u_2$ with $u_1 \neq u_2, (u_1, v) \in E$ and $(u_2, v) \in E$ one of the following conditions holds:
   3.1 $((\mathcal{R}(T(u_1)) \cap \mathcal{R}(v)) \cap (\mathcal{R}(T(u_2)) \cap \mathcal{R}(v))) = \emptyset$ or
   3.2 $(\mathcal{R}(T(u_1)) = \mathcal{R}(v)) \lor (\mathcal{R}(T(u_2)) = \mathcal{R}(v))$

We denote by $T(v)$ the partial tree rooted at $v$. 

Adding Weights to the Edges

For two nodes $v, u \in V$ we define $u \sqcap v = \mathcal{R}(u) \cap \mathcal{R}(v)$

- for simplicity, we assume that every predicate involves exactly two relations
- then for all $u, v \in V$, $u \sqcap v$ contains a single relation (or none)

Let $v \in V$ be a node with $\mathcal{R}(v) = \{ R_i, R_j \}$

- we abbreviate $R_i \bowtie_v R_j$ by $\bowtie_v$

Using these notations, we can attach weights to the edges to define the *weighted directed join graph*. 
Adding Weights to the Edges (2)

Let $G = (V, E_p, E_v)$ be a directed join graph for a conjunctive query with join predicates $P$. The *weighted directed join graph* is derived from $G$ by attaching a weight to each edge as follows:

- Let $(u, v) \in E_p$ be a physical edge. The weight $w_{u,v}$ of $(u, v)$ is defined as
  \[
  w_{u,v} = \frac{|\Delta_u|}{|u \cap v|}
  \]

- For virtual edges $(u, v) \in E_v$, we define
  \[
  w_{u,v} = 1
  \]

Note that $w_{u,v}$ is not symmetric.
Remark on Edge Weights

The weights of physical edges are equal to the $s_i$ used in the IKKBZ-Algorithm.

Assume $\mathcal{R}(u) = \{R_1, R_2\}$, $\mathcal{R}(v) = \{R_2, R_3\}$. Then

$$w_{u,v} = \frac{|\bigtriangleup_u|}{|u \cap v|} = \frac{|R_1 \bigtriangleup R_2|}{|R_2|} = \frac{f_{1,2}|R_1||R_2|}{|R_2|} = f_{1,2}|R_1|$$

Hence, if the join $R_1 \bigtriangleup_u R_2$ is executed before the join $R_2 \bigtriangleup_v R_3$, the input size to the latter join changes by a factor of $w_{u,v}$.
Adding Weights to the Nodes

- the weight of a node reflects the change in cardinality to be expected when certain other joins have been executed before
- it depends on a (partial) spanning tree $S$

Given $S$, we denote by $\Join^S_{p_{i,j}}$ the result of the join $\Join_{p_{i,j}}$ if all joins preceding $p_{i,j}$ in $S$ have been executed. Then the weight attached to node $p_{i,j}$ is defined as

$$w(p_{i,j}, S) = \frac{|\Join^S_{p_{i,j}}|}{|R_i \Join_{p_{i,j}} R_j|}$$

For empty sequences we define $w(p_{i,j}, \epsilon) = |R_i \Join_{p_{i,j}} R_j|$. Similarly, we define the cost of a node $p_{i,j}$ depending on other joins preceding it in some given spanning tree $S$. We denote this by $C(p_{i,j}, S)$.

- the actual cost function can be chosen arbitrarily
- if we have several join implementations: take the minimum
Algorithm Overview

The algorithm builds an effective spanning tree in two phases:

1. it takes those edges with a weight $< 1$
2. it adds the remaining edges

keeping track of effectiveness during the process.

- rational: weight $< 1$ is good
- decreases the work for later operators
- should be done early
- increasing intermediate results as late as possible
MVP Algorithm

MVP($G$)
Input: a weighted directed join graph $G = (V, E_p, E_v)$
Output: an effective spanning tree
$Q_1$ = a priority queue for nodes, smallest $w$ first
$Q_2$ = a priority queue for nodes, largest $w$ first
insert all nodes in $V$ to $Q_1$
$G' = (V', E')$ with $V' = V$ and $E' = E_p$  // working graph
$S = (V_S, E_s)$ with $V_S = V$ and $E_s = \emptyset$  // result
MVP-Phase1($G, G', S, Q_1, Q_2$)
MVP-Phase2($G, G', S, Q_1, Q_2$)
return $S$
MVP Algorithm (2)

MVP-Phase1($G, G', S, Q_1, Q_2$)

**Input:** state from MVP

**Output:** modifies the state

while $|Q_1| > 0 \land |E_s| < |V| - 1$

\[ v = \text{head of } Q_1 \]

\[ U = \{ u | (u, v) \in E' \land w_{u,v} < 1 \land (V, E_S \cup \{(u, v)\}) \text{ is acyclic and effective} \} \]

if $U = \emptyset$

\[ Q_1 = Q_1 \setminus \{ v \} \]

\[ Q_2 = Q_2 \cup \{ v \} \]

else

\[ u = \arg \max_{u \in U} C(\nabla v, S) - C(\nabla v, (V, E_S \cup \{(u, v)\})) \]

MVPUpdate($G, G', S, (u, v)$)

recompute $w$ for $v$ and its ancestors

}
MVP Algorithm (3)

MVP-Phase2($G, G', S, Q_1, Q_2$)

**Input:** state from MVP

**Output:** modifies the state

while $|Q_2| > 0 \land |E_s| < |V| - 1$ {
  $$v = \text{head of } Q_2$$
  $$U = \{(x, y) | (x, y) \in E' \land (x = v \lor y = v) \land (V, E_S \cup \{(x, y)\}) \text{ is acyclic and effective}\}$$
  $$(x, y) = \arg\min_{(x, y) \in U} C(\nabla_v, (V, E_S \cup \{(x, y)\})) - C(\nabla_v, S)$$
  MVPUpdate($G, G', S, (x, y)$)
  recompute $w$ for $y$ and its ancestors
}
MVP Algorithm (4)

MVPUpdate\(G, G', S, (u, v))\)

**Input:** state from MVP, an edge to be added to \(S\)

**Output:** modifies the state

\[
E_S = E_S \cup \{(u, v)\}
\]

\[
E' = E' \setminus \{(u, v), (v, u)\}
\]

\[
E' = E' \setminus \{(u, w)|(u, w) \in E'\}
\]

\[
E' = E' \cup \{(v, w)|(u, w) \in E_p, (v, w) \in E_v\}
\]

**if** \(v\) has two incoming edges in \(S\) \{ 

\[
E' = E' \setminus \{(w, v)|(w, v) \in E'\}
\]

\}

**if** \(v\) has one outflowing edge in \(S\) \{ 

\[
E' = E' \setminus \{(v, w)|(v, w) \in E'\}
\]

\}

- checks that \(S\) is a tree (one parent, at most two children)
- detects transitive physical edges
Dynamic Programming

Basic premise:
- optimality principle
- avoid duplicate work

A very generic class of approaches:
- all cost functions (as long as optimality principle holds)
- left-deep/bushy, with/without cross products
- finds the optimal solution

Concrete algorithms can be more specialized of course.
Optimality Principle

Consider the two joins trees

\[((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \bowtie R_5\]

and

\[((R_3 \bowtie R_1) \bowtie R_2) \bowtie R_4) \bowtie R_5\]

- if we know that \(((R_1 \bowtie R_2) \bowtie R_3)\) is cheaper than \(((R_3 \bowtie R_1) \bowtie R_2)\), we know that the first join is cheaper than the second join
- hence, we could avoid generating the second alternative and still won’t miss the optimal join tree
Optimality Principle (2)

More formally, the optimality for join ordering:

Let $T$ be an optimal join tree for relations $R_1, \ldots, R_n$. Then, every subtree $S$ of $T$ must be an optimal join tree for the relations contained in it.

- optimal substructure: the optimal solution for a problem can be constructed from optimal solutions to its subproblems
- not true with physical properties (but can be fixed)
Overview Dynamic Programming Strategy

- generate optimal join trees bottom up
- start from optimal join trees of size one (relations)
- build larger join trees by (re-)using those of smaller sizes

To keep the algorithms concise, we use a subroutine \textit{CreateJoinTree} that joins two trees.
Creating Join Trees

CreateJoinTree( \( T_1, T_2 \) )

**Input:** two (optimal) join trees \( T_1, T_2 \)

for linear trees: assume that \( T_2 \) is a single relation

**Output:** an (optimal) join tree for \( T_1 \bowtie T_2 \)

\( B = \emptyset \)

for each \( \text{impl} \in \{ \text{applicable join implementations} \} \) {

if \( \neg \text{right-deep only} \) {

\( B = B \cup \{ T_1 \bowtie^{\text{impl}} T_2 \} \)

}

if \( \neg \text{left-deep only} \) {

\( B = B \cup \{ T_2 \bowtie^{\text{impl}} T_1 \} \)

}

return \( \arg \min_{T \in B} C(T) \)
Search Space with Sharing under Optimality Principle

\{R_1, R_2, R_3, R_4\}

\{R_1, R_2, R_4\}
\{R_1, R_2, R_3\}
\{R_1, R_2\}
\{R_1, R_3\}
\{R_1, R_4\}
\{R_2, R_3, R_4\}
\{R_1, R_3, R_4\}
\{R_1, R_2, R_3\}
\{R_1, R_2, R_4\}
\{R_1, R_2, R_3, R_4\}
Generating Linear Trees

- A (left-deep) linear tree $T$ with $|T| > 1$ has the form $T' \otimes R_i$, with $|T| = |T'| + 1$
- If $T$ is optimal, $T'$ must be optimal too
- Basic strategy: find the optimal $T$ by joining all optimal $T'$ with $T \setminus T'$

Enumeration order varies between algorithms
Generating Linear Trees (2)

DPsizeLinear(\(R\))

**Input:** a set of relations \(R = \{R_1, \ldots, R_n\}\) to be joined

**Output:** an optimal left-deep (right-deep, zig-zag) join tree

\(B = \) an empty DP table \(2^R \rightarrow \) join tree

for each \(R_i \in R\)

\(B[\{R_i\}] = R_i\)

for each \(1 < s \leq n\) ascending { 
  for each \(S \subseteq R, R_i \in R : |S| = s - 1 \land R_i \notin S\) { 
    if \(\neg\)cross products \(\land \neg S\) connected to \(R_i\) continue
    \(p_1 = B[S], p_2 = B[\{R_i\}]\)
    if \(p_1 = \epsilon\) continue
    \(P = \text{CreateJoinTree}(p_1, p_2)\);
    if \(B[S \cup \{R_i\}] = \epsilon \lor C(B[S \cup \{R_i\}]) > C(P)\)
    \(B[S \cup \{R_i\}] = P\)
  }
}

return \(B[\{R_1, \ldots, R_n\}]\)
Order in which Subtrees are generated

The ordering in which subtrees are generated does not matter as long as the following condition is not violated:

Let $S$ be a subset of $\{R_1, \ldots, R_n\}$. Then, before a join tree for $S$ can be generated, the join trees for all relevant subsets of $S$ must already be available.

- relevant means that they are valid subproblems by the algorithm
- usually this means connected (no cross products)
Generation in Integer Order

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>{R_1}</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>{R_2}</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>{R_1, R_2}</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>{R_3}</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>{R_1, R_3}</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>{R_2, R_3}</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>{R_1, R_2, R_3}</td>
<td></td>
</tr>
</tbody>
</table>

- can be done very efficiently
- set representation is just a number
Generating Linear Trees (3)

DPsubLinear($R$)

**Input:** a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined

**Output:** an optimal left-deep (right-deep, zig-zag) join tree

$B =$ an empty DP table $2^R \rightarrow \text{join tree}$

for each $R_i \in R$

\[ B[\{R_i\}] = R_i \]

for each $1 < i \leq 2^n - 1$ ascending \{\}

\[ S = \{R_j \in R | (\lfloor i/2^j \rfloor - 1) \mod 2) = 1 \} \]

for each $R_j \in S$ \{\}

if $\neg$cross products $\land \neg S \setminus \{R_j\}$ connected to $R_j$ continue

$\ p_1 = B[S \setminus \{R_j\}], \ p_2 = B[\{R_j\}]$

if $p_1 = \epsilon$ continue

$P = \text{CreateJoinTree}(p_1, p_2)$;

if $B[S] = \epsilon \lor \text{C}(B[S]) > \text{C}(P)$ $B[S] = P$

\}

\}

return $B[\{R_1, \ldots, R_n\}]$
Generating Bushy Trees

• a bushy tree $T$ with $|T| > 1$ has the form $T_1 \bowtie T_2$, with $|T| = |T_1| + |T_2|

• if $T$ is optimal, both $T_1$ and $T_2$ must be optimal too

• basic strategy: find the optimal $T$ by joining all pairs of optimal $T_1$ and $T_2$
Generating Bushy Trees (2)

DPsize($R$)

**Input:** a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined

**Output:** an optimal bushy join tree

$B = \text{an empty DP table } 2^R \rightarrow \text{join tree}$

for each $R_i \in R$

$B[\{R_i\}] = R_i$

for each $1 < s \leq n$ ascending {

for each $S_1, S_2 \subset R : |S_1| + |S_2| = s$

if (¬cross products ∧¬$S_1$ connected to $S_2$) ∨ ($S_1 \cap S_2 \neq \emptyset$) continue

$p_1 = B[S_1], p_2 = B[S_2]$

if $p_1 = \epsilon \lor p_2 = \epsilon$ continue

$P = \text{CreateJoinTree}(p_1, p_2)$;

if $B[S_1 \cup S_2] = \epsilon \lor C(B[S_1 \cup S_2]) > C(P)$

$B[S_1 \cup S_2] = P$

}

return $B[\{R_1, \ldots, R_n\}]$
Generating Bushy Trees (3)

DPsub$(R)$

**Input:** a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined

**Output:** an optimal bushy join tree

$B =$ an empty DP table $2^R \rightarrow$ join tree

for each $R_i \in R$

$B[\{R_i\}] = R_i$

for each $1 < i \leq 2^n - 1$ ascending { 

$S = \{R_j \in R|([i/2^{j-1}] \mod 2) = 1\}$

for each $S_1 \subset S$, $S_2 = S \setminus S_1$

if ¬cross products ∧ ¬$S_1$ connected to $S_2$ continue

$p_1 = B[S_1], p_2 = B[S_2]$

if $p_1 = \epsilon \lor p_2 = \epsilon$ continue

$P = \text{CreateJoinTree}(p_1, p_2)$;

if $B[S] = \epsilon \lor C(B[S]) > C(P)$ $B[S] = P$

}

return $B[\{R_1, \ldots, R_n\}]$
Efficient Subset Generation

If we use integers as set representation, we can enumerate all subsets of $S$ as follows:

$$S_1 = S \& (-S)$$

**do**

$$S_2 = S - S_1$$

// Do something with $S_1$ and $S_2$

$$S_1 = S \& (S_1 - S)$$

**while** ($S_1 \neq S$)

- enumerates all subsets except $\emptyset$ and $S$ itself
- very fast
Remarks

- $\text{DPsize}/\text{DPsizeLinear}$ does not really test for $p_1 = \epsilon$
- it keeps a list of plans for a given size
- candidates can be found very fast
- ensures polynomial time in some cases (we will look at it again)
- $\text{DPsub}/\text{DPsubLinear}$ is faster if the problem is not polynomial, though
Memoization

- top-down formulation of dynamic programming
- recursive generation of join trees
- memoize already generated join trees to avoid duplicate work
- easier code
- sometimes more efficient (more knowledge, allows for pruning)
- but usually slower than dynamic programming
Memoization (2)

Memoization($R$)

**Input:** a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined

**Output:** an optimal bushy join tree

$B =$ an empty DP table $2^R \rightarrow$ join tree

for each $R_i \in R$

$B[\{R_i\}] = R_i$

MemoizationRec($B$, $R$)

return $B[\{R_1, \ldots, R_n\}]$

- initializes the DP table and triggers the recursive search
- main work done during recursion
Memoization (3)

MemoizationRec\((B, S)\)

**Input:** a DP table \(B\) and a set of relations \(S\) to be joined

**Output:** an optimal bushy join tree for the subproblem

\[
\text{if } B[S] = \epsilon \{ \\
\text{for each } S_1 \subset S, S_2 = S \setminus S \\
\quad p_1 = \text{MemoizationRec}(B, S_1), \ p_2 = \text{MemoizationRec}(B, S_2) \\
\quad P = \text{CreateJoinTree}(p_1, p_2) \\
\quad \text{if } B[S] = \epsilon \lor C(B[S]) > C(P) \ B[S] = P
\}
\]

return \(B[S]\)

- checks for connectedness omitted