**First Lemma**

**Lemma:** The cost function $C_h$ has the ASI-Property.

**Proof:** The proof can be derived from the definition of $C_H$:

$$C_H(AUVB) = C_H(A) + T(A)C_H(U) + T(A)T(U)C_H(V) + T(A)T(U)T(V)C_H(B)$$

and, hence,

$$C_H(AUVB) - C_H(AVUB) = T(A)[C_H(V)(T(U) - 1) - C_H(U)(T(V) - 1)] = T(A)C_H(U)C_H(V)[\text{rank}(U) - \text{rank}(V)]$$

The lemma follows.
Module

Let $M = \{A_1, \ldots, A_n\}$ be a set of sequences of nodes in a given precedence graph. Then, $M$ is called a module, if for all sequences $B$ that do not overlap with the sequences in $M$, one of the following conditions holds:

- $B \rightarrow A_i$, $\forall A_i \in M$
- $A_i \rightarrow B$, $\forall A_i \in M$
- $B \not\rightarrow A_i$ and $A_i \not\rightarrow B$, $\forall A_i \in M$
Second Lemma

**Lemma:** Let $C$ be any cost function with the ASI property and $\{A, B\}$ a module. If $A \rightarrow B$ and additional $\text{rank}(B) \leq \text{rank}(A)$, then we find an optimal sequence among those in which $B$ directly follows $A$.

**Proof:** by contradiction. Every optimal permutation must have the form $UAVBW$ since $A \rightarrow B$.

Assumption: $V \neq \epsilon$ for all optimal solutions.

- if $\text{rank}(V) \leq \text{rank}(A)$, we can exchange $V$ and $A$ without increasing the costs.
- if $\text{rank}(A) \leq \text{rank}(V)$, $\text{rank}(B) \leq \text{rank}(V)$ due to the transitivity of $\leq$. Hence, we can exchange $B$ and $V$ without increasing the costs.

Both exchanges produces legal sequences since $\{A, B\}$ is a module.
Contradictory Sequences and Compound Relations

- If the precedence graph demands $A \rightarrow B$ but $\text{rank}(B) \leq \text{rank}(A)$, we speak of *contradictory sequences* $A$ and $B$.
- Second lemma $\Rightarrow$ no non-empty subsequence can occur between $A$ and $B$.
- We combine $A$ and $B$ into a new single node replacing $A$ and $B$.
- This node represents a *compound relation* comprising of all relations in $A$ and $B$.
- Its cardinality is computed by multiplying the cardinalities of all relations in $A$ and $B$.
- Its selectivity is the product of all selectivities $s_i$ of relations $R_i$ contained in $A$ and $B$. 
Normalization and Denormalization

- the continued process of building compound relations until no more contradictory sequences exist is called \textit{normalization}
- the opposite step, replacing a compound relation by the sequence of relations it was derived from is called \textit{denormalization}
## Algorithm

IKKBZ($G, C_H$)

**Input:** an acyclic query graph $G$ for relations $R = \{R_1, \ldots, R_n\}$, a cost function $C_H$

**Output:** the optimal left-deep tree $S = \emptyset$

1. $S = \emptyset$
2. for $\forall R_i \in R$
   - $G_i =$ the precedence graph derived from $G$ rooted at $R_i$
   - $S_i =$ IKKBZ-Sub($G_i, C_H$)
   - $S = S \cup \{S_i\}$
3. return $\arg \min_{S_i \in S} C_H(S_i)$

- considers each relation as starting relation
- constructs the precedence graph and starts the main algorithm
Algorithm (2)

IKKBZ-Sub($G_i, C_H$)

**Input:** a precedence graph $G_i$ for relations $R = \{R_1, \ldots, R_n\}$ rooted at $R_i$, a cost function $C_H$

**Output:** the optimal left-deep tree under $G_i$

**while** $G_i$ is not a chain {

$r =$ a subtree of $G_i$ whose subtrees are chains

IKKBZ-Normalize($r$)

merge the chains under $r$ according to the rank function (ascending)

}

IKKBZ-Denormalize($G_i$)

**return** $G_i$

- transforms the precedence graph into a chain
- wherever there are multiple choices, there are serialized according to the rank
- normalization required to preserve the query graph
Algorithm (3)

IKKBZ-Normalize($R$)

**Input:** a subtree $R$ of a precedence graph $G = (V, E)$

**Output:** a normalized subtree

while $\exists r, c \in T, (r, c) \in E : \text{rank}(r) > \text{rank}(c)$ {
  replace $r$ and $c$ by a compound relation $r'$ that represent $rc$
}

return $R$

- merges relations that would have been reorder if only considering the rank
- guarantees that the rank is ascending in each subchain
Algorithm (4)

IKKBZ-Denormalize($R$)

**Input:** a precedence graph $R$ containing relations and compound relations

**Output:** a denormalized precedence graph, containing only relations

while $\exists r \in R : r$ is a compound relation {
    replace $r$ by the sequence of relations it represents
}

return $R$

- unpacks the compound relations
- required to get a real join tree as final result
Sample Algorithm Execution

Input: query graph

the precedence graph includes the ranks

Step 1: precedence graph for \( R_1 \)
Sample Algorithm Execution (2)

Step 1: precedence graph for $R_1$

Step 2: normalization

$\text{rank}(R_6) > \text{rank}(R_7)$, but $R_6 \rightarrow R_7$
Sample Algorithm Execution (3)

Step 2: normalization

\[\text{rank}(R_5) < \text{rank}(R_{6,7})\]
Sample Algorithm Execution (3)

Step 3: merging subchains

\[ rank(R_4) > rank(R_5), \text{ but } R_4 \rightarrow R_5 \]
Sample Algorithm Execution (4)

Step 4: normalization

\[ \text{rank}(R_{4,5}) > \text{rank}(R_{6,7}), \text{ but } R_{4,5} \rightarrow R_{6,7} \]
Sample Algorithm Execution (5)

Step 5: normalization

\[ \text{rank}(R_3) < \text{rank}(R_2) < \text{rank}(R_{4,5,6,7}) \]
Sample Algorithm Execution (6)

Step 6: merging subchains

Step 7: denormalization

Algorithm has to continue for all other root relations.
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 *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 *physical edges*, those in $E_v$ *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
Examples: Spanning Tree and Join Tree (2)

\[ R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \]

query graph

\[ p_{1,2} \rightarrow p_{2,3} \rightarrow p_{3,4} \]

spanning tree II

\[ p_{1,2} \rightarrow \ldots \rightarrow p_{3,4} \]

directed join graph

join tree II
Examples: Spanning Tree and Join Tree (3)

\[ R_1 - R_2 - R_3 - R_4 - R_5 \]

query graph

\[ p_{2,3}, p_{3,4} \]

spanning tree III

\[ p_{1,2}, p_{4,5} \]

directed join graph

\[ p_{1,2}, p_{3,4} \]

join tree III (?)

- 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 $u, v \in V$, $a \sqcap v$ contains a single relation (or none)

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

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

Using these notations, we can attach weights to the edges to define the \textit{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
  \[ w_{u,v} = \frac{|\nabla_u|}{|u \cap v|} \]

- 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

\[
\begin{align*}
    w_{u,v} &= \frac{| \bigtriangleup_u |}{|u \cap v|} \\
    &= \frac{|R_1 \bigtriangleup R_2|}{|R_2|} \\
    &= \frac{f_{1,2}|R_1||R_2|}{|R_2|} \\
    &= f_{1,2}|R_1|
\end{align*}
\]

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
Join Ordering

MVP

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 {

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

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 $|Q_2| > 0 \land |E_s| < |V| - 1$ {
    \begin{align*}
    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(\Diamond_v, (V, E_s \cup \{(x, y)\})) - C(\Box_v, S) \\
    
    \text{MVPUpdate}(G, G', S, (x, y)) \\
    \text{recompute } w \text{ for } y \text{ and its ancestors}
    \end{align*}
}
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:

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

- 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 $T_1 \Join T_2$

$B = \emptyset$

for $\forall impl \in \{ \text{applicable join implementations} \}$ {

if $\neg$right-deep only {

$B = B \cup \{ T_1 \Join^{impl} T_2 \}$

}

if $\neg$left-deep only {

$B = B \cup \{ T_2 \Join^{impl} T_1 \}$

}

}

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

\{ R_1, R_2, R_3, R_4 \}
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

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

\[
\begin{align*}
\text{for } & \forall R_i \in R \\
& B[\{R_i\}] = R_i \\
\text{for } & \forall 1 < s \leq n \text{ ascending } \{ \\
& \text{for } \forall S \subset R, R_i \in R : |S| = s - 1 \land R_i \not\in S \{ \\
& \quad \text{if } \neg\text{cross products } \land \neg S \text{ connected to } R_i \text{ continue} \\
& \quad p_1 = B[S], p_2 = B[\{R_i\}] \\
& \quad \text{if } p_1 = \epsilon \text{ continue} \\
& \quad P = \text{CreateJoinTree}(p_1, p_2); \\
& \quad \text{if } B[S \cup \{R_i\}] = \epsilon \lor C(B[S \cup \{R_i\}]) > C(P) \\
& \quad B[S \cup \{R_i\}] = P \\
& \} \\
\} \\
\text{return } B[\{R_1, \ldots, R_n\}] 
\]