Polynomial algorithm for join ordering (original [2], improved [3])

- produces optimal left-deep trees without cross products
- requires acyclic join graphs
- cost function must have ASI property
- join method must be fixed

Can be used as heuristic if the requirements are violated
Overview

- the algorithms considers each relation as first relation to be joined
- it tries to order the other relations by ”benefit” (rank)
- if the ordering violates the query constraints, it constructs compounds
- the compounds guarantee the constraints (locally) and are again ordered by benefit
- related to a known job-ordering algorithm
Cost Function

The IKKBZ algorithm considers only cost functions of the form

\[ C(T_i \bowtie R_j) = |T_i| \ast h_j(|R_j|) \]

- each relation \( R_j \) can have its own \( h_j \)
- we denote the set of \( h_j \) by \( H \), writing \( C_H \) for the parametrized cost function
- examples: \( h_j \equiv 1.2 \) for \( C_{h_j} \), \( h_j \equiv id \) for \( C_{id} \)

We will often use cardinalities, thus we define \( n_i \):

- \( n_i \) is the cardinality of \( R_i \) (\( n = R_i \))
- \( h_i(n_i) \) is are the costs per input tuple of a join with \( R_i \)
Precedence Graph

Given a query graph $G = (V, E)$ and a starting relation $R_k$, we construct the directed precedence graph $G_k^P = (V_k^P, E_k^P)$ rooted in $R_k$ as follows:

1. choose $R_k$ as the root node of $G_k^P$, $V_k^P = \{ R_k \}$
2. while $|V_k^P| < |V|$, choose a $R_i \in V \setminus V_k^P$ such that $\exists R_j \in V_k^P : (R_j, R_i) \in E$. Add $R_i$ to $V_k^P$ and $R_j \rightarrow R_i$ to $E_k^P$.

The precedence graph describes the (partial) ordering of joins implied by the query graph.
Sample Precedence Graph

query graph

precedence graph rooted in $R_1$
Conformance to a Precedence Graph

A sequence $S = v_1, \ldots, v_k$ of nodes conforms to a precedence graph $G = (V, E)$ if the following conditions are satisfied:

1. $\forall i \in [2, k] \exists j \in [1, i]: (v_j, v_i) \in E$
2. $\forall i \in [1, k], j \in [i, k]: (v_j, v_i) \in E$

Note: IKKBZ constructs left-deep trees, therefore it is sufficient to consider sequences.
**Notations**

For non-empty sequences $S_1$ and $S_2$ and a precedence graph $G = (V, E)$, we write $S_1 \rightarrow S_2$ if $S_1$ must occur before $S_2$. More precisely $S_1 \rightarrow S_2$ iff:

1. $S_1$ and $S_2$ conform to $G$
2. $S_1 \cap S_2 = \emptyset$
3. $\exists v_i, v_j \in V : v_i \in S_1 \land v_j \in S_2 \land (v_i, v_j) \in E$
4. $\forall v_i, v_j \in V : v_i \in S_1 \land v_j \in V \setminus S_1 \setminus S_2 \land (v_i, v_j) \in E$

Further, we write

$$R_{1,2,...,k} = R_1 \Join R_2 \Join \ldots \Join R_k$$
$$n_{1,2,...,k} = |R_{1,2,...,k}|$$
Selectivities

For a given precedence graph, let \( R_i \) be a relation and \( \mathcal{R}_i \) be the set of relations from which there exists a path to \( R_i \):

- in any conforming join tree which includes \( R_i \), all relations from \( \mathcal{R}_i \) must be joined first
- all other relations \( R_j \) that might be joined before \( R_i \) will have no connection to \( R_i \), thus \( f_{i,j} = 1 \)

Hence, we can define the selectivity of the join with \( R_i \) as

\[
s_i = \begin{cases} 
1 & \text{if } |\mathcal{R}_i| = 0 \\
\prod_{R_j \in \mathcal{R}_i} f_{i,j} & \text{if } |\mathcal{R}_i| > 0 
\end{cases}
\]

Note: we call the \( s_i \) a selectivities, although they depend on the precedence graph
Cardinalities

If the query graph is a chain (totally ordered), the following conditions holds:

\[
n_{1,2,...,k} = s_k \cdot |R_k| \cdot |R_{1,2,...,k-1}|
\]

As a closed form, we can write

\[
n_{1,2,...,k} = \prod_{i=1}^{k} s_i n_i
\]

as \( s_1 = 1 \)
Costs

The costs for a totally ordered precedence graph $G$ can be computed as follows:

$$C_H(G) = \sum_{i=2}^{n} [n_{1,2,...,i-1} h_i(n_i)]$$

$$= \sum_{i=2}^{n} [(\prod_{j=1}^{i} s_j n_j) h_i(n_i)]$$

- if we choose $h_i(n_i) = s_i n_i$ then $C_H \equiv C_{out}$
- the factor $s_i n_i$ determines how much the input relation to be joined with $R_i$ changes its cardinality after the join has been performed
- if $s_i n_i$ is less than one, we call the join *decreasing*, if it is larger than one, we call the join *increasing*
Join Ordering

Costs (2)

For the algorithm, we prefer a (equivalent) recursive definition of the cost function:

\[
\begin{align*}
C_H(\epsilon) &= 0 \\
C_H(R_i) &= 0 \text{ if } R_i \text{ is the root} \\
C_H(R_i) &= h_i(n_i) \text{ else} \\
C_H(S_1S_2) &= C_H(S_1) + T(S_1) \times C_H(S_2)
\end{align*}
\]

where

\[
\begin{align*}
T(\epsilon) &= 1 \\
T(S) &= \prod_{R_i \in S} s_in_i
\end{align*}
\]
ASl Property

Let $A$ and $B$ be two sequences and $V$ and $U$ two non-empty sequences. We say a cost function $C$ has the adjacent sequence interchange property (ASI property), if and only if there exists a function $T$ and a rank function defined as

$$\text{rank}(S) = \frac{T(S) - 1}{C(S)}$$

such that the following holds

$$C(AUVB) \leq C(AVUB) \iff \text{rank}(U) \leq \text{rank}(V)$$

if $AUVB$ and $AVUB$ satisfy the precedence constraints imposed by a given precedence graph.
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 \textit{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 $A \to B$ but $\text{rank}(B) \leq \text{rank}(A)$, we speak of contradictionary 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 nodes 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 *normalization*
- the opposite step, replacing a compound relation by the sequence of relations it was derived from is called *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

$S = \emptyset$

for $\forall R_i \in R$

\[
G_i = \text{the precedence graph derived from } G \text{ rooted at } R_i
\]

$S_i = \text{IKKBZ-Sub}(G_i, C_H)$

$S = S \cup \{S_i\}$

\[\text{return } \arg \min_{S_i \in S} C_H(S_i)\]

- 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 {
  $r = \text{a subtree of } G_i \text{ whose subtrees are chains}$
  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

Step 1: precedence graph for $R_1$

the precedence graph includes the ranks
Sample Algorithm Execution (2)

Step 1: precedence graph for $R_1$

$rank(R_6) > rank(R_7)$, 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

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

Step 4: normalization

\[ \text{rank}(R_{4,6,7}) < \text{rank}(R_5) < \text{rank}(R_3) < \text{rank}(R_2) \]
Sample Algorithm Execution (5)

\[
\begin{align*}
R_1 \\
\downarrow \\
R_{4,6,7} & \quad \frac{199}{320} \\
\downarrow \\
R_5 & \quad \frac{5}{6} \\
\downarrow \\
R_3 & \quad \frac{24}{25} \\
\downarrow \\
R_2 & \quad \frac{49}{50}
\end{align*}
\]

Step 5: merging subchains    Step 6: denormalization

Algorithm has to continue for all other root relations.