Greedy Heuristics - First Algorithm

- search space of joins trees is very large
- greedy heuristics produce suitable join trees very fast
- suitable for large queries

For the first algorithm we consider:

- left-deep trees
- no cross products
- relations ordered to some weight function (e.g. cardinality)

Note: the algorithms produces a sequence of relations; it uniquely identifies the left-deep join tree.

Greedy Heuristics - First Algorithm (2)

GreedyJoinOrdering-1($R = \{R_1, \ldots, R_n\}, w : R \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \arg\min_{R_i \in R} w(R_i)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- disadvantage: fixed weight functions
- already chosen relations do not affect the weight
- e.g. does not support minimizing the intermediate result

Greedy Heuristics - Second Algorithm

GreedyJoinOrdering-2($R = \{R_1, \ldots, R_n\}$, $w : R, R^* \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \arg\min_{R_i \in R} w(R_i, S)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- can compute relative weights
- but first relation has a huge effect
- and the fewest information available

Greedy Heuristics - Third Algorithm

GreedyJoinOrdering-3($R = \{R_1, \ldots, R_n\}$, $w : R, R^* \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \emptyset$
for $\forall R_i \in R$ {
  $R' = R \setminus \{R_i\}$
  $S' =< R_i >$
  while ($|R'| > 0$) {
    $m = \arg\min_{R_j \in R'} w(R_j, S')$
    $R' = R' \setminus \{m\}$
    $S' = S' \circ < m >$
  }
  $S = S \cup \{S'\}$
}
return $\arg\min_{S' \in S} w(S'[n], S'[1 : n-1])$

- commonly used: minimize selectivities (*MinSel*)

Greedy Operator Ordering

- the previous greedy algorithms only construct left-deep trees
- Greedy Operator Ordering (GOO) [1] constructs bushy trees

Idea:

- all relations have to be joined somewhere
- but joins can also happen between whole join trees
- we therefore greedily combine join trees (which can be relations)
- combine join trees such that the intermediate result is minimal

Greedy Operator Ordering (2)

GOO($R = \{R_1, \ldots, R_n\}$)
Input: a set of relations to be joined
Output: a join tree
T = R
while $|T| > 1$ {
  $(T_i, T_j) = \arg\min_{(T_i \in T, T_j \in T), T_i \neq T_j} |T_i \bowtie T_j|$
  $T = (T \setminus \{T_i\}) \setminus \{T_j\}$
  $T = T \cup \{T_i \bowtie T_j\}$
}
return $T_0 \in T$

- constructs the result bottom up
- join trees are combined into larger join trees
- chooses the pair with the minimal intermediate result in each pass

IKKBZ

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

$$C(T_i \bowtie R_j) = |T_i| * h_j(|R_j|)$$

- 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_{hj}, $h_j \equiv id$ for C_{nl}

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

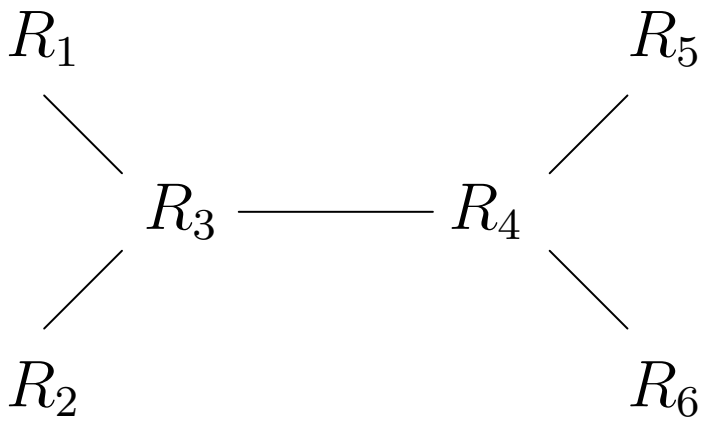
Precedence Graph

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:

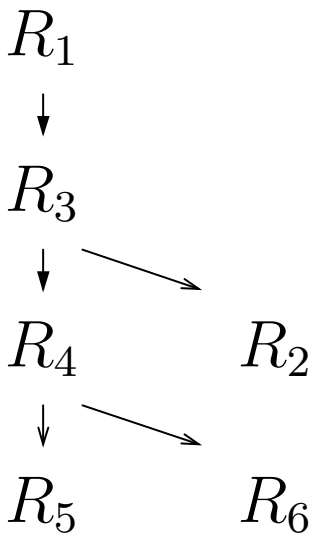
1. choose R_k as the root node of G_k^P, $V_k^P = \{R_k\}$
2. while $|V_k^P| < |V|$, choose a $R_i \in V \setminus V_k^P$ such that $\exists R_j \in V_k^P : (R_j, R_i) \in E$. Add R_i to V_k^P and $R_j \rightarrow R_i$ to E_k^P.

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

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 $S_1 \longrightarrow S_2$ if S_1 must occur before S_2. More precisely $S_1 \longrightarrow S_2$ iff:

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

Further, we write

$$\begin{aligned} R_{1,2,\ldots,k} &= R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k \\ n_{1,2,\ldots,k} &= |R_{1,2,\ldots,k}| \end{aligned}$$

Selectivities

For a given precedence graph, let R_i be a relation and $\mathcal{R}_i$ be the set of a 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:

$$\begin{aligned} n_{1,2,\ldots,k} &= s_k * |R_k| * |R_{1,2,\ldots,k-1}| \\ &= |s_k| * n_k * n_{1,2,\ldots,k-1} \end{aligned}$$

As a closed form, we can write

$$n_{1,2,\ldots,k} = \prod_{i=1}^{k} s_i n_i$$

as $s_1 = 1$

Costs

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

$$\begin{aligned} C_H(G) &= \sum_{i=2}^{n} [n_{1,2,\ldots,i-1} h_i(n_i)] \\ &= \sum_{i=2}^{n} [(\prod_{j=1}^{i} s_j n_j) h_i(n_i)] \end{aligned}$$

- 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*

Costs (2)

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

$$\begin{aligned} C_H(\epsilon) &= 0 \\ C_H(R_i) &= 0 \text{ if } R_i \text{ is the root} \\ C_H(R_i) &= h_i(n_i) \text{ else} \\ C_H(S_1 S_2) &= C_H(S_1) + T(S_1) * C_H(S_2) \end{aligned}$$

where

$$\begin{aligned} T(\epsilon) &= 1 \\ T(S) &= \prod_{R_i \in S} s_i n_i \end{aligned}$$

ASI 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

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

such that the following holds

$$C(AUVB) \leq C(AVUB) \Leftrightarrow rank(U) \leq rank(V)$$

if AUVB and AVUB satisfy the precedence constraints imposed by a given precedence graph.