3. Join Ordering

- Basics
- Search Space
- Greedy Heuristics
- IKKBZ
- MVP
- Dynamic Programming
- Simplifying the Query Graph
- Generating Permutations
- Transformative Approaches
- Randomized Approaches
- Metaheuristics
- Iterative Dynamic Programming
- Order Preserving Joins

Queries Considered

Concentrate on join ordering, that is:

- conjunctive queries
- simple predicates
- predicates have the form $a_1 = a_2$ where a_1 is an attribute and a_2 is either an attribute or a constant
- even ignore constants in some algorithms

We join relations $R_1, \ldots, R_n$, where R_i can be

- a base relation
- a base relation including selections
- a more complex building block or access path

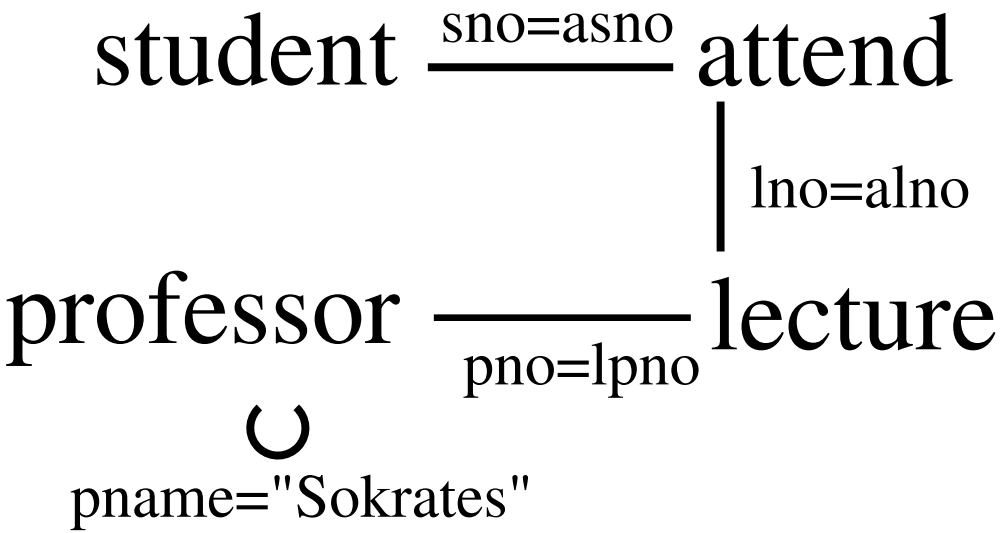
Pretending to have a base relation is ok for now.

Query Graph

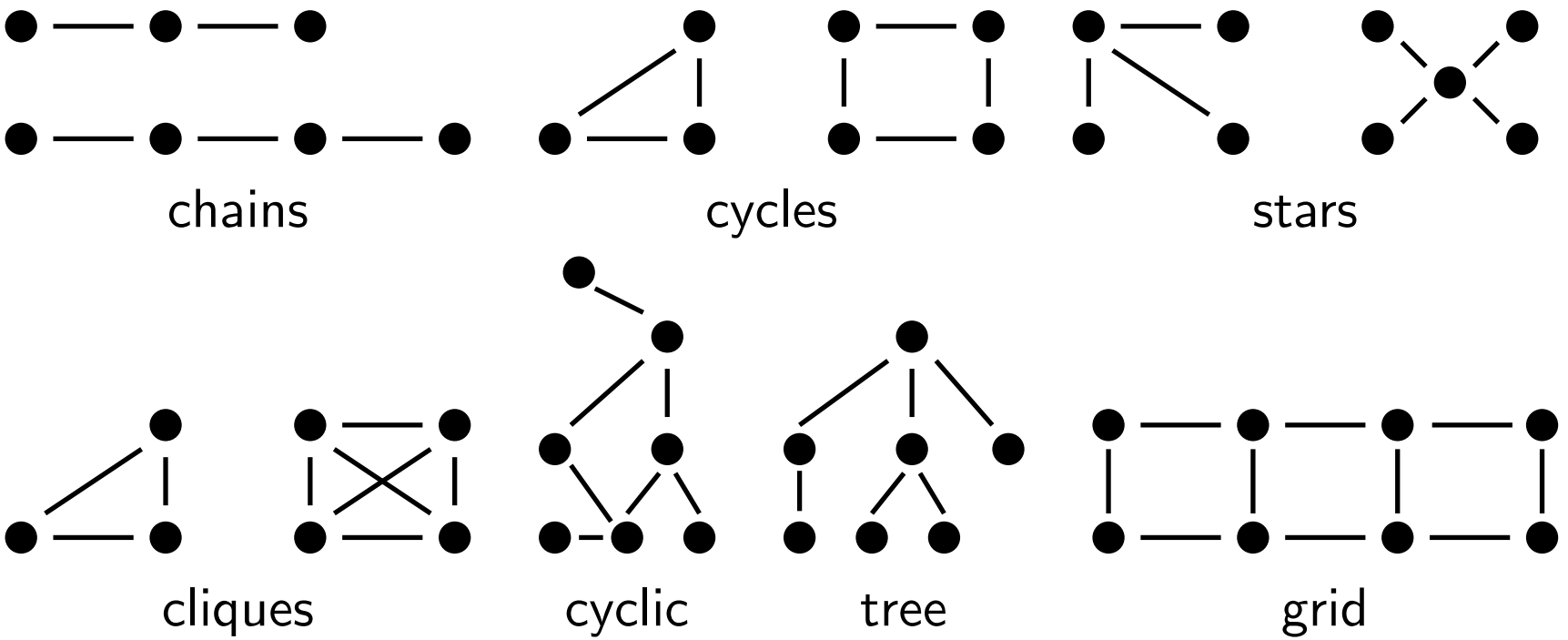
Queries of this type can be characterized by their query graph:

- the query graph is an undirected graph with $R_1, \ldots, R_n$ as nodes
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and $a_2 \in R_j$ forms an edge between R_i and R_j labeled with the predicate
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and a_2 is a constant forms a self-edge on R_i labeled with the predicate
- most algorithms will not handle self-edges, they have to be pushed down

Sample Query Graph



Shapes of Query Graphs



- real world queries are somewhere in-between
- chain, cycle, star and clique are interesting to study
- they represent certain kind of problems and queries

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

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

Calculate:

- result cardinality:

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

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

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

- 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:

$$\begin{aligned} |R_1| &= 10 \\ |R_2| &= 100 \\ |R_3| &= 1000 \\ f_{1,2} &= 0.1 \\ f_{2,3} &= 0.2 \end{aligned}$$

- 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

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

- 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:

$$\begin{aligned} C_{nlj}(e_1 \bowtie e_2) &= |e_1||e2| \\ C_{hj}(e_1 \bowtie e_2) &= 1.2|e_1| \\ C_{smj}(e_1 \bowtie e_2) &= |e_1|\log(|e_1|) + |e_2|\log(|e_2|) \end{aligned}$$

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

	C_{out}	C_{nl}	C_{hj}	C_{smj}
$R_1 \bowtie R_2$	100	1000	12	697.61
$R_2 \bowtie R_3$	20000	100000	120	10630.26
$R_1 \times R_3$	10000	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	132	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	24120	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	22000	143542.00

- 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:

	C_{out}
$R_1 \bowtie R_2$	200
$R_2 \times R_3$	4
$R_1 \bowtie R_3$	200
$(R_1 \bowtie R_2) \bowtie R_3$	240
$(R_2 \times R_3) \bowtie R_1$	44
$(R_1 \bowtie R_3) \bowtie R_2$	240

- 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:

	C_{out}
$R_1 \bowtie R_2$	2
$R_2 \bowtie R_3$	200
$R_3 \bowtie R_4$	2
$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$	24
$((R_2 \times R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$	6

- 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 $C_{impl}(e_1 \bowtie^{impl} e_2) = C_{impl}(e_2 \bowtie^{impl} e_1)$
- 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:

	ASI	¬ASI
symmetric	C_{out}	C_{smj}
¬symmetric	C_{hj}	-

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

Classification of Join Ordering Problems

We distinguish four different dimensions:

1. query graph class: *chain*, *cycle*, *star*, and *clique*
2. join tree structure: *left-deep*, *zig-zag*, or *bushy* trees
3. join construction: *with* or *without* cross products
4. cost function: *with* or *without* ASI property

In total, 48 different join ordering problems.

Reminder: Catalan Numbers

The number of binary trees with n leave nodes is given by $\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ is defined as

$$\mathcal{C}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1) & \text{if } n > 0 \end{cases}$$

It can be written in a closed form as

$$\mathcal{C}(n) = \frac{1}{n+1}\binom{2n}{n}$$

The Catalan Numbers grown in the order of $\Theta(4^n/n^{\frac{3}{2}})$

Number Of Join Trees with Cross Products

left deep		$n!$
right deep		$n!$
zig-zag		$n!2^{n-2}$
bushy		$n!\mathcal{C}(n-1)$
	$=$	$\frac{(2n-2)!}{(n-1)!}$

- rational: number of leaf combinations ($n!$) $\times$ number of unlabeled trees (varies)
- grows exponentially
- increases even more with a flexible tree structure

Chain Queries, no Cross Products

Let us denote the number of left-deep join trees for a chain query $R_1 - \ldots - R_n$ as $f(n)$

- obviously $f(0) = 1, f(1) = 1$
- for $n > 1$, consider adding R_n to all join trees for $R_1 - \ldots - R_{n-1}$
- R_n can be added at any position following R_{n-1}
- lets denote the position of R_{n-1} from the bottom with k ($[1, n-1]$)
- there are $n - k$ join trees for adding R_n after R_{n-1}
- one additional tree if $k = 1$, R_n can also be added before R_{n-1}
- for R_{n-1} to be at k, $R_{n-k} - \ldots R_{n-2}$ must be below it. $f(k-1)$ trees

for $n > 1$:

$$f(n) = 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k)$$

Chain Queries, no Cross Products (2)

The number of left-deep join trees for chain queries of size n is

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k) & \text{if } n \geq 2 \end{cases}$$

solving the recurrence gives the closed form

$$f(n) = 2^{n-1}$$

- generalization to zig-zag as before

Chain Queries, no Cross Products (3)

The generalization to bushy trees is not as obvious

- each subtree must contain a subchain to avoid cross products
- thus do not add single relations but subchains
- whole chain must be $R_1 - \ldots - R_n$, cut anywhere
- consider commutativity (two possibilities)

This leads to the formula

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ \sum_{k=1}^{n-1} 2f(k)f(n-k) & \text{if } n \geq 2 \end{cases}$$

solving the recurrence gives the closed form

$$f(n) = 2^{n-1}\mathcal{C}(n-1)$$

Star Queries, no Cross Products

Consider a star query with R_1 at the center and $R_2, \ldots, R_n$ as satellites.

- the first join must involve R_1
- afterwards all other relations can be added arbitrarily

This leads to the following formulas:

- left-deep: $2 * (n-1)!$
- zig-zag: $2 * (n-1)! * 2^{n-2} = (n-1)! * 2^{n-1}$
- bushy: no bushy trees possible (R_1 required), same as zig-zag

Clique Queries, no Cross Products

- in a clique query, every relation is connected to each other
- thus no join tree contains cross products
- all join trees are valid join trees, the number is the same as with cross products

Sample Numbers, without Cross Products

	Chain Queries			Star Queries	
n	Left-Deep 2^{n-1}	Zig-Zag 2^{2n-3}	Bushy $2^{n-1}\mathcal{C}(n-1)$	Left-Deep $2(n-1)!$	Zig-Zag/Bushy $2^{n-1}(n-1)!$
1	1	1	1	1	1
2	2	2	2	2	2
3	4	8	8	4	8
4	8	32	40	12	48
5	16	128	224	48	384
6	32	512	1344	240	3840
7	64	2048	8448	1440	46080
8	128	8192	54912	10080	645120
9	256	32768	366080	80640	10321920
10	512	131072	2489344	725760	18579450

Sample Numbers, with Cross Products

n	Left-Deep $n!$	Zig-Zag $n!2^{n-2}$	Bushy $n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	968972800	17643225600

Problem Complexity

query graph	join tree	cross products	cost function	complexity
general	left-deep	no	ASI	NP-hard
tree/star/chain	left-deep	no	ASI, 1 joint.	P
star	left-deep	no	NLJ+SMJ	NP-hard
general/tree/star	left-deep	yes	ASI	NP-hard
chain	left-deep	yes	-	open
general	bushy	no	ASI	NP-hard
tree	bushy	no	-	open
star	bushy	no	ASI	P
chain	bushy	no	any	P
general	bushy	yes	ASI	NP-hard
tree/star/chain	bushy	yes	ASI	NP-hard

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($R = \{R_1, \ldots, R_n\}$, $w : R \to \mathbb{R}$)
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 each $R_i \in R$ {
  $R' = R \setminus \{R_i\}$
  $S' =< R_i >$
  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

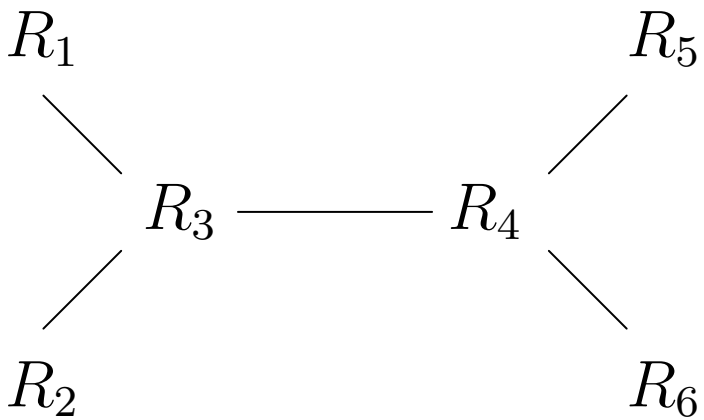
Precedence Graph

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:

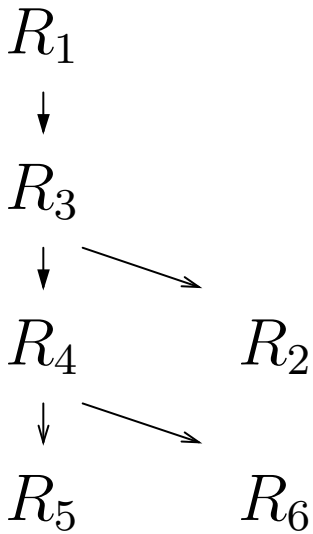
1. choose R_k as the root node of G_k^P, $V_k^P = \{R_k\}$
2. while $|V_k^P| < |V|$, choose a $R_i \in V \setminus V_k^P$ such that $\exists R_j \in V_k^P : (R_j, R_i) \in E$. Add R_i to V_k^P and $R_j \rightarrow R_i$ to E_k^P.

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 $S_1 \rightarrow S_2$ 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. $\exists v_i, v_j \in V : v_i \in S_1 \wedge v_j \in S_2 \wedge (v_i, v_j) \in E$
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.

First Lemma

Lemma: The cost function C_h has the ASI-Property.
Proof: The proof can be derived from the definition of C_H:

$$\begin{aligned} C_H(AUVB) &= C_H(A) \\ &\quad + T(A)C_H(U) \\ &\quad + T(A)T(U)C_H(V) \\ &\quad + T(A)T(U)T(V)C_H(B) \end{aligned}$$

and, hence,

$$\begin{aligned} C_H(AUVB) - C_H(AVUB) &= T(A)[C_H(V)(T(U)-1) - C_H(U)(T(V)-1 \\ &= T(A)C_H(U)C_H(V)[rank(U) - rank(V)] \end{aligned}$$

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 $rank(B) \leq rank(A)$, 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 $rank(V) \leq rank(A)$, we can exchange V and A without increasing the costs.
- if $rank(A) \leq rank(V)$, $rank(B) \leq rank(V)$ 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 $rank(B) \leq rank(A)$, 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 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 each $R_i \in R$ {
  G_i = the precedence graph derived from G rooted at R_i
  S_i = IKKBZ-Sub(G_i, C_H)
  $S = S \cup \{S_i\}$
}
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 $\exists r, c \in T, (r, c) \in E : rank(r) > rank(c)$ {
  replace r and c by a compound relation r' that represent rc
}
return R

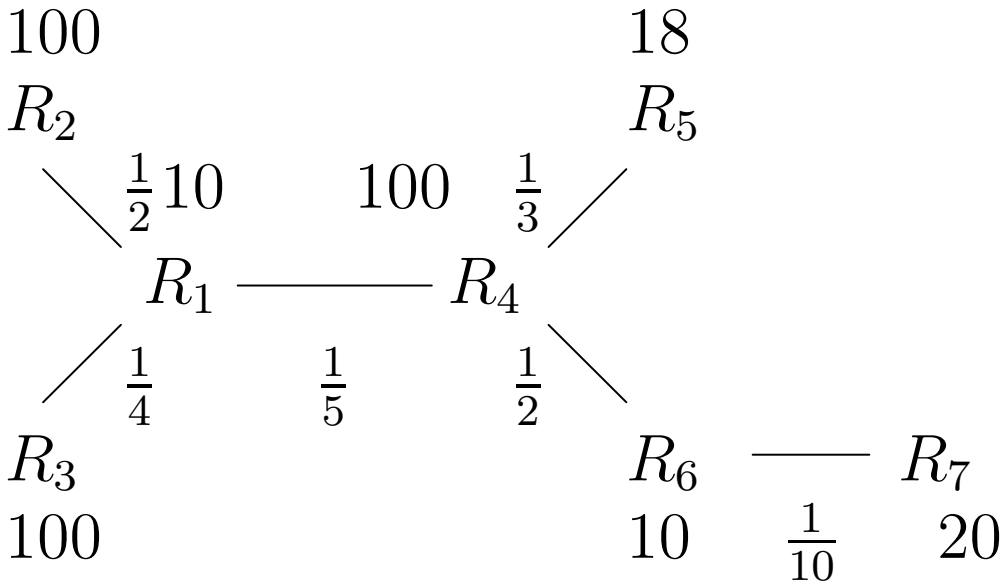
- merges relations that would have been reorder if only considering the rank
- guarantees that the rank is ascending in each subchain

Algorithm (4)

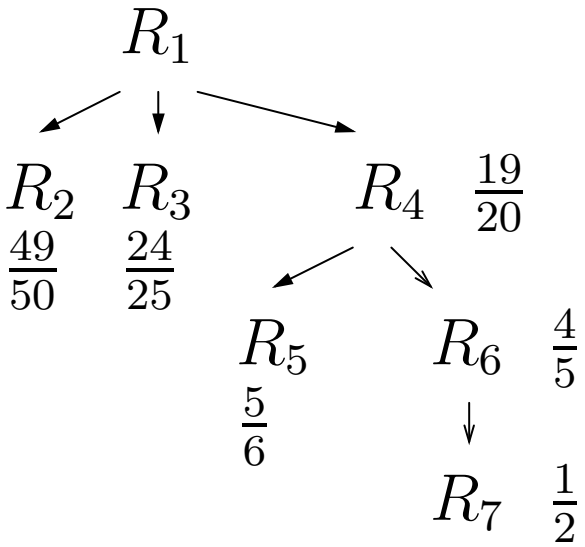
IKKBZ-Denormalize(R)
Input: a precedence graph R containing relations and compound relations
Output: a denormalized precedence graph, containing only relations
while $\exists r \in R : r$ is a compound relation $\{$
  replace r by the sequence of relations it represents
$\}$
return R

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

Sample Algorithm Execution



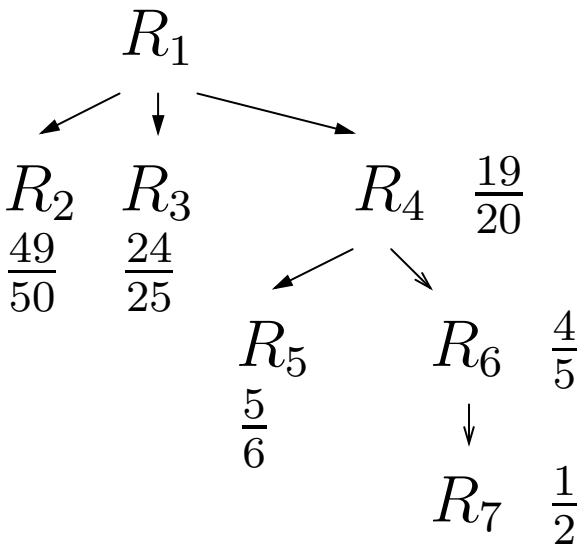
Input: query graph



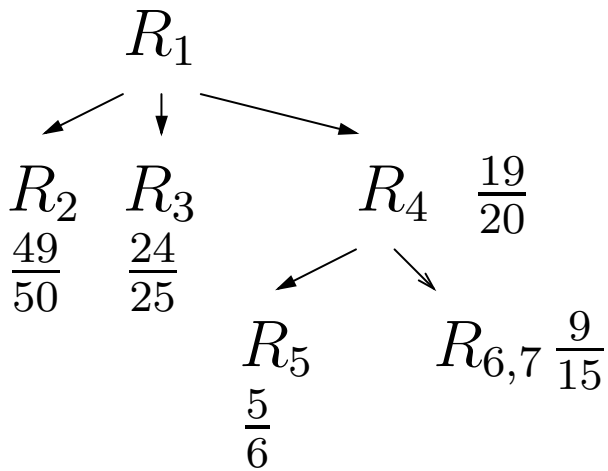
Step 1: precedence graph for R_1

the precedence graph includes the ranks

Sample Algorithm Execution (2)



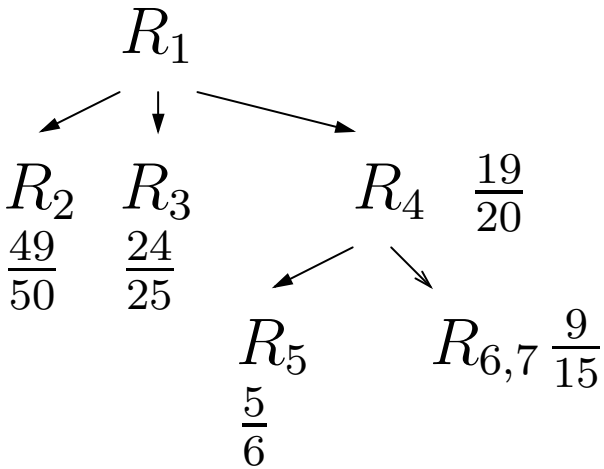
Step 1: precedence graph for R_1



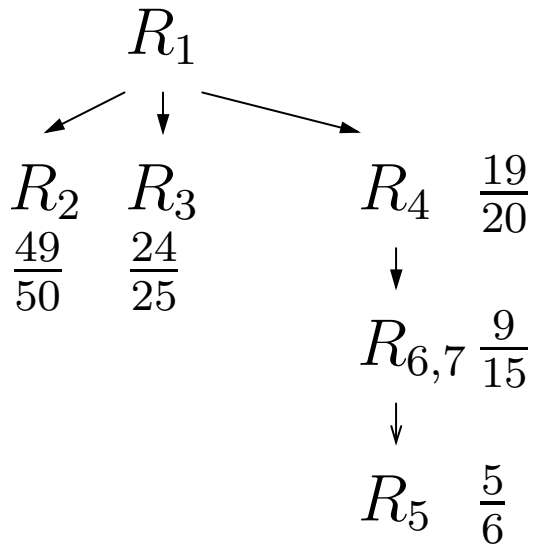
Step 2: normalization

$rank(R_6) > rank(R_7)$, but $R_6 \rightarrow R_7$

Sample Algorithm Execution (3)



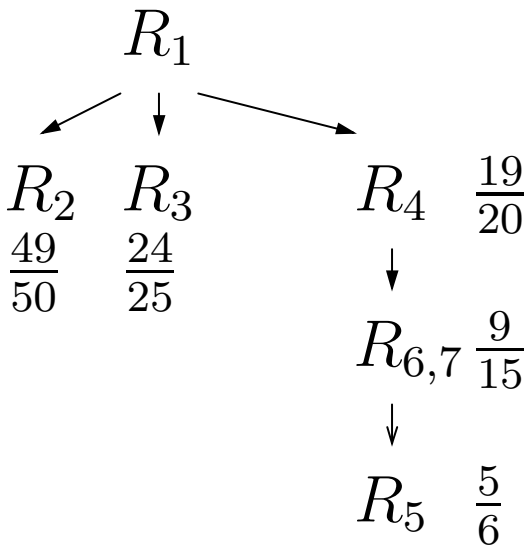
Step 2: normalization



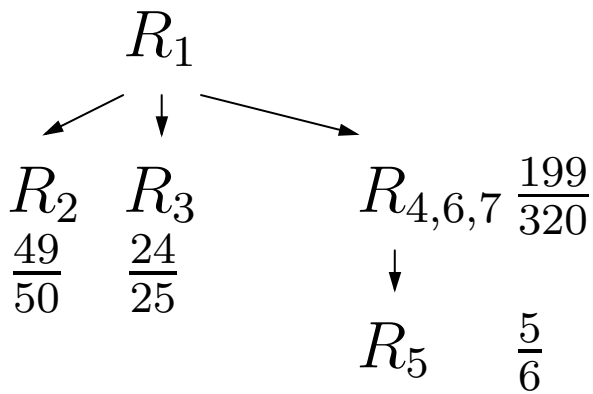
Step 3: merging subchains

$rank(R_5) < rank(R_{6,7})$

Sample Algorithm Execution (3)



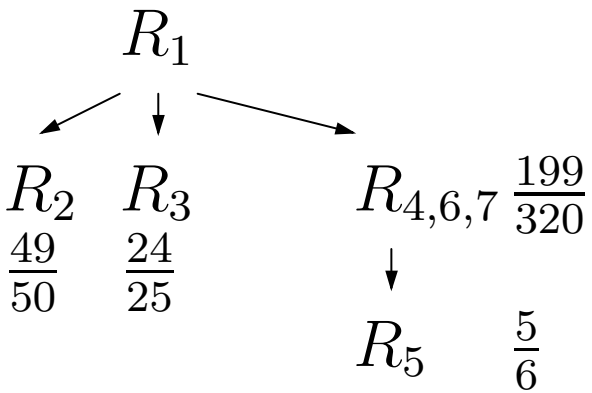
Step 3: merging subchains



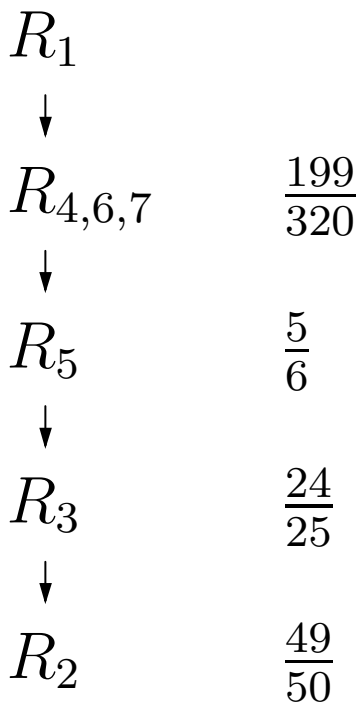
Step 4: normalization

$rank(R_4) > rank(R_5)$, but $R_4 \rightarrow R_5$

Sample Algorithm Execution (4)



Step 4: normalization



Step 5: merging subchains

$rank(R_{4,6,7}) < rank(R_5) < rank(R_3) < rank(R_2)$

Sample Algorithm Execution (5)

$$
\begin{array}{ll}
R_1 & \\
\downarrow & \\
R_{4,6,7} & \frac{199}{320} \\
\downarrow & \\
R_5 & \frac{5}{6} \\
\downarrow & \\
R_3 & \frac{24}{25} \\
\downarrow & \\
R_2 & \frac{49}{50}
\end{array}
$$

Step 5: merging subchains

$$
\begin{array}{c}
R_1 \\
\downarrow \\
R_4 \\
\downarrow \\
R_6 \\
\downarrow \\
R_7 \\
\downarrow \\
R_5 \\
\downarrow \\
R_3 \\
\downarrow \\
R_2
\end{array}
$$

Step 6: 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

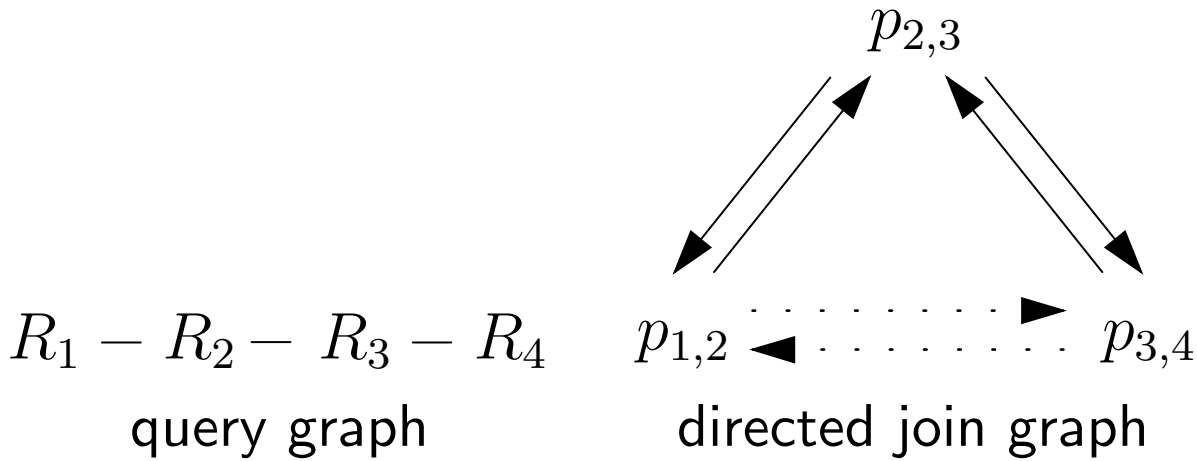
Given a conjunctive query with predicates P.

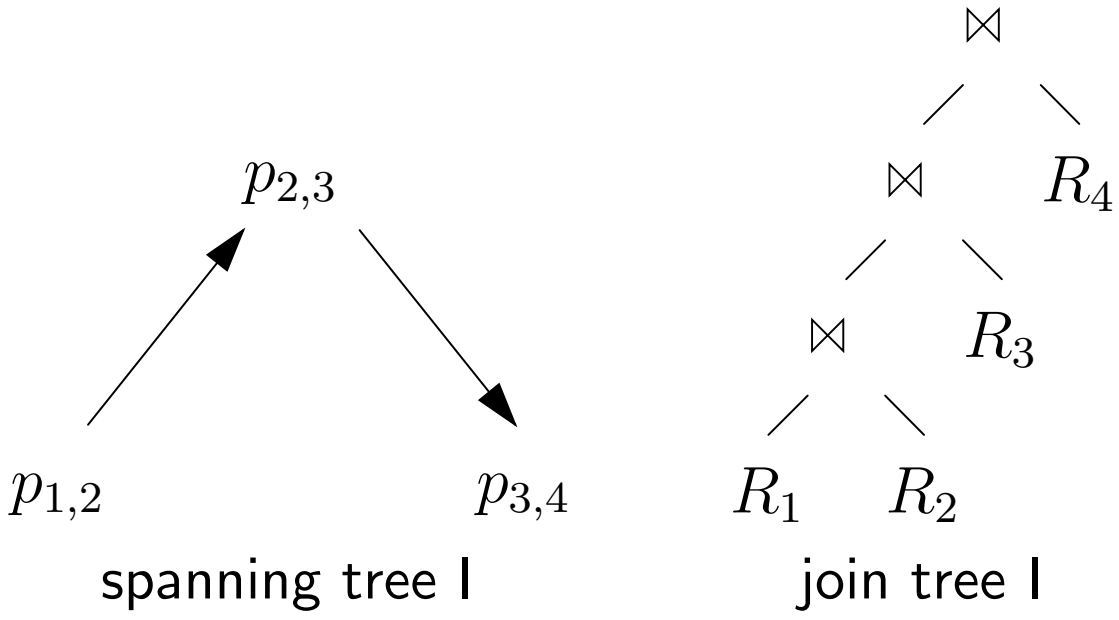
- for all join predicates $p \in P$, we denote by $\mathcal{R}(p)$ the relations whose attributes are mentioned in p.
- the *directed join graph* of the query is a triple $G = (V, E_p, E_v)$, where V is the set of predicates and E_p and E_v are sets of directed edges defined as follows
- for any nodes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

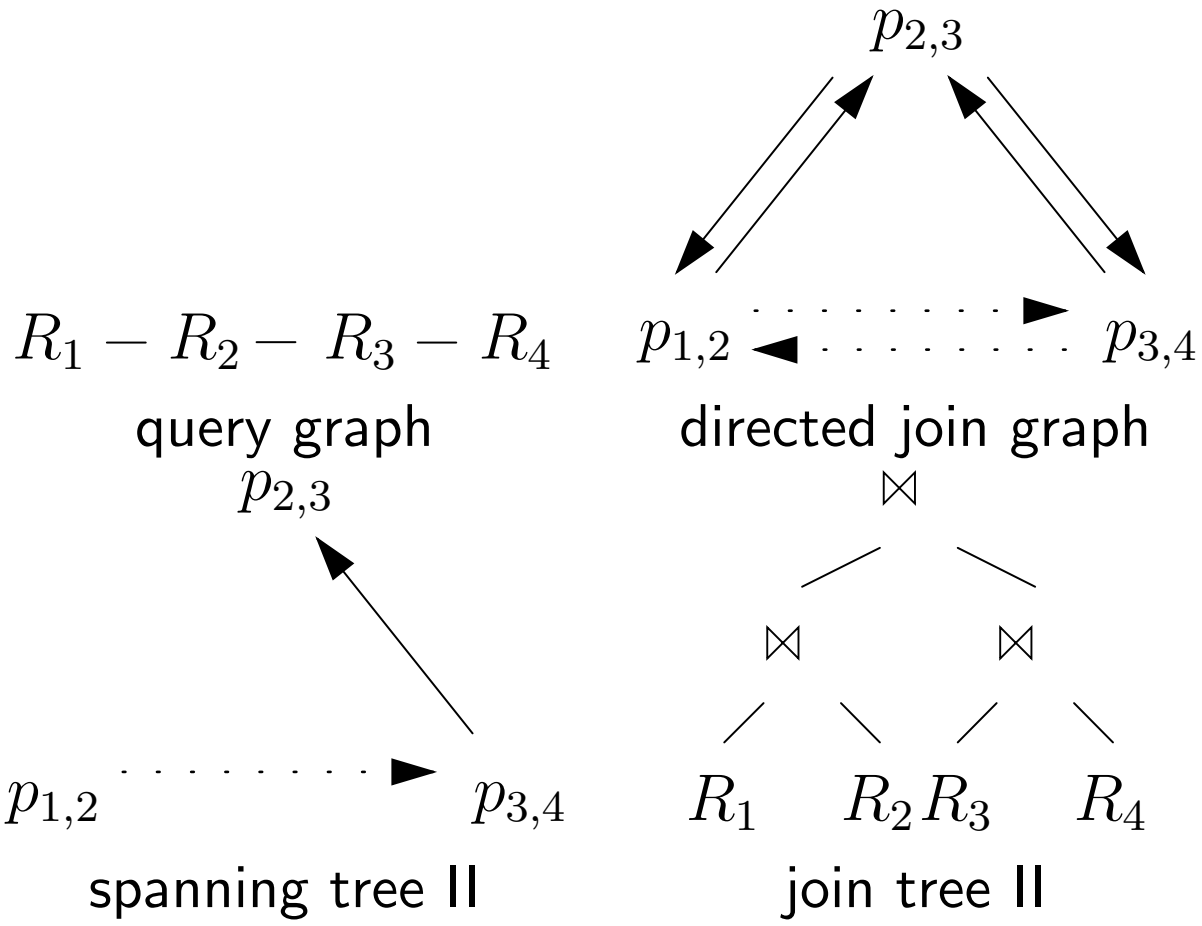
Examples: Spanning Tree and Join Tree

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





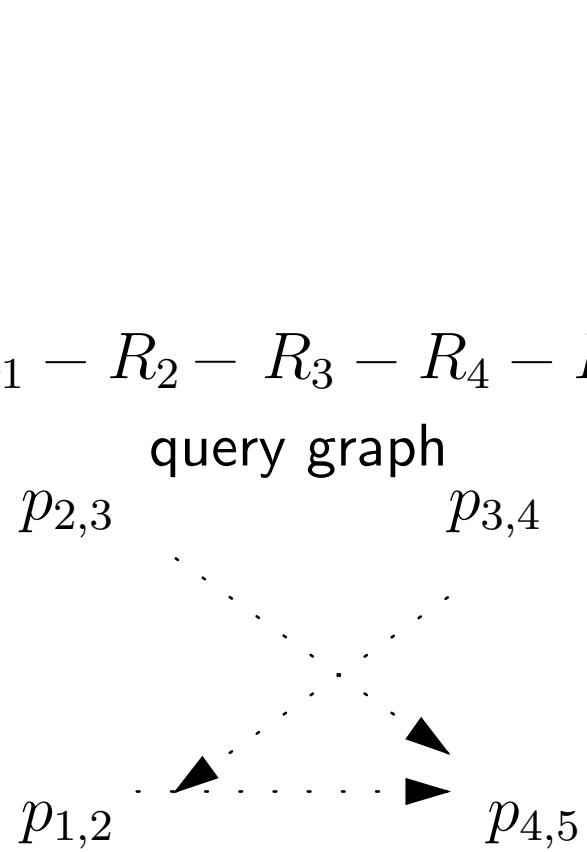
Examples: Spanning Tree and Join Tree (2)



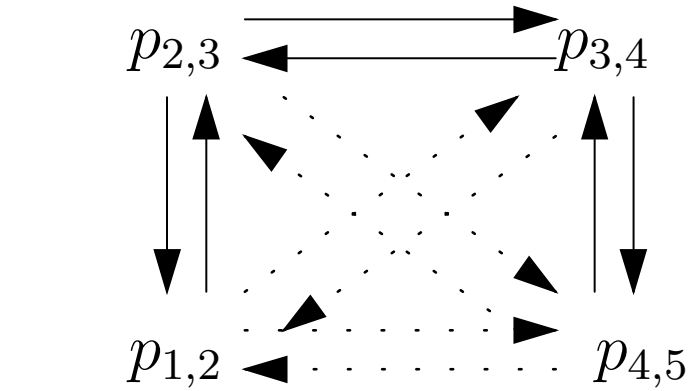
Examples: Spanning Tree and Join Tree (3)

$R_1 - R_2 - R_3 - R_4 - R_5$

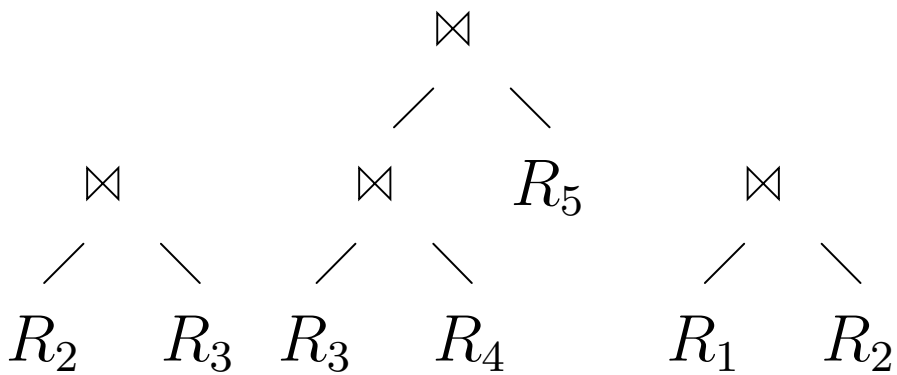
query graph



spanning tree III



directed join graph



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 $(\mathcal{R}(T(u_1)) = \mathcal{R}(v)) \vee (\mathcal{R}(T(u_2)) = \mathcal{R}(v))$

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 $R_i \bowtie_v R_j$ 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

$$w_{u,v} = \frac{|\bowtie_u|}{|u \sqcap v|}$$

- 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

$$\begin{aligned} w_{u,v} &= \frac{|\bowtie_u|}{|u \sqcap v|} \\ &= \frac{|R_1 \bowtie R_2|}{|R_2|} \\ &= \frac{f_{1,2}|R_1||R_2|}{|R_2|} \\ &= f_{1,2}|R_1| \end{aligned}$$

Hence, if the join $R_1 \bowtie_u R_2$ is executed before the join $R_2 \bowtie_v R_3$, 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 $\bowtie^S_{p_{i,j}}$ the result of the join $\bowtie_{p_{i,j}}$ if all joins preceding $p_{i,j}$ in S have been executed. Then the weight attached to node $p_{i,j}$ is defined as

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

For empty sequences we define $w(p_{i,j}, \epsilon) = |R_i \bowtie_{p_{i,j}} R_j|$.
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, largest w first
Q_2 = a priority queue for nodes, smallest 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 $|Q_1| > 0 \wedge |E_s| < |V| - 1$ {
  v = head of Q_1
  $U = \{u | (u, v) \in E' \wedge w_{u,v} < 1 \wedge (V, E_S \cup \{(u, v)\})$ is acyclic and effective$\}$
  if $U = \emptyset$ {
    $Q_1 = Q_1 \setminus \{v\}$
    $Q_2 = Q_2 \cup \{v\}$
  } **else** {
    $u = \arg\max_{u \in U} C(\bowtie_v, S) - C(\bowtie_v, (V, E_S \cup \{(u, v)\}))$
    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 $|Q_2| > 0 \wedge |E_s| < |V| - 1$ {
  v = head of Q_2
  $U = \{(x,y) | (x,y) \in E' \wedge (x = v \vee y = v) \wedge (V, E_S \cup \{(x,y)\})$ is acyclic and effective$\}$
  $(x,y) = \arg\min_{(x,y) \in U} C(\bowtie_v, (V, E_S \cup \{(x,y)\})) - C(\bowtie_v, S)$
  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

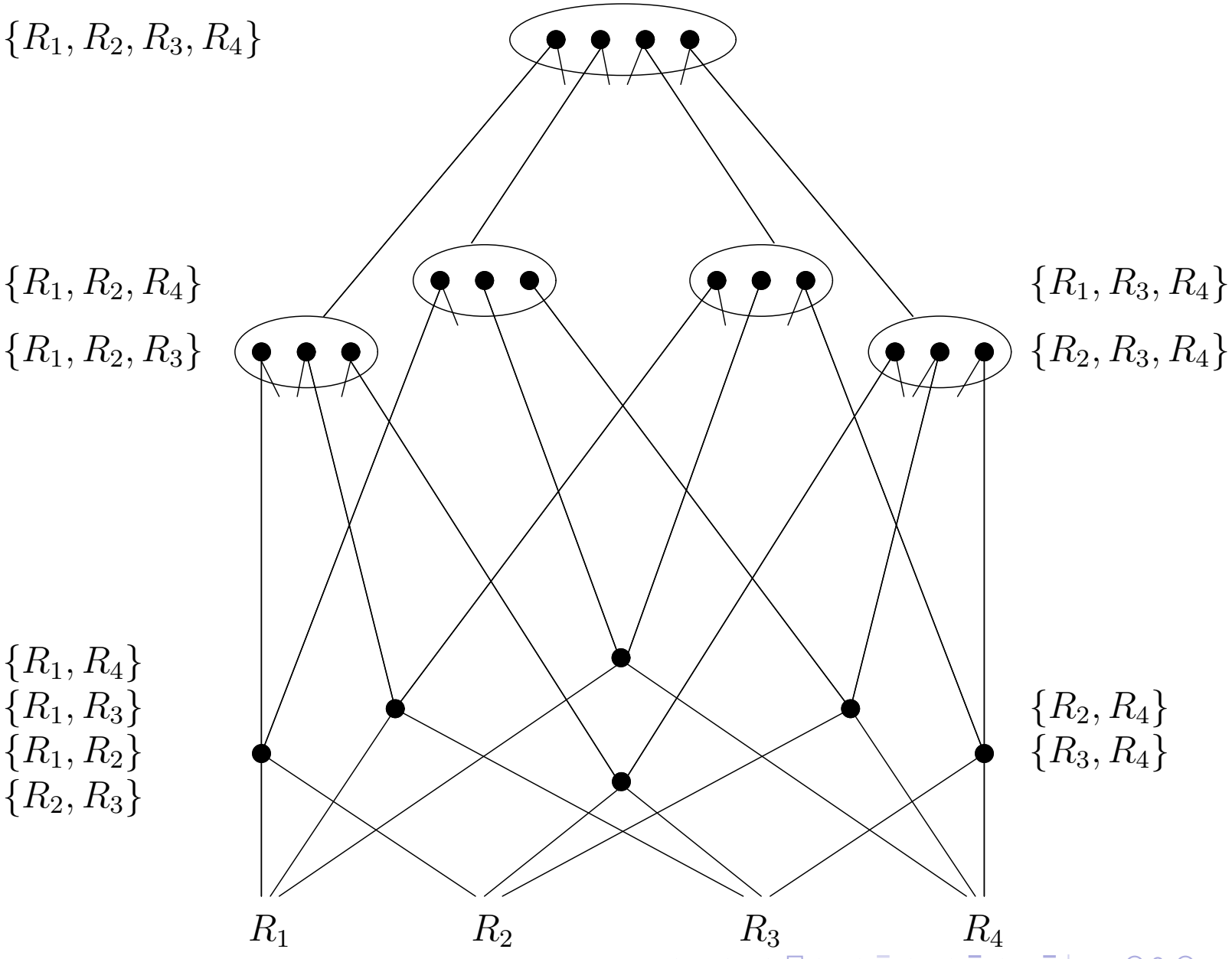
- generate optimal join trees bottom up
- start from optimal join trees of size one (relations)
- build larger join trees by (re-)using those of smaller sizes

To keep the algorithms concise, we use a subroutine *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 *impl* $\in$ { applicable join implementations } {
  if $\neg$right-deep only {
    $B = B \cup \{ T_1 \bowtie^{impl} T_2 \}$
  }
  if $\neg$left-deep only {
    $B = B \cup \{ T_2 \bowtie^{impl} T_1 \}$
  }
}
return $\arg\min_{T \in B} C(T)$

Search Space with Sharing under Optimality Principle



Generating Linear Trees

- a (left-deep) linear tree T with $|T| > 1$ has the form $T' \bowtie R_i$, 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 $S \subset R, R_i \in R : |S| = s - 1 \wedge R_i \notin S$ {
    if $\neg$cross products $\wedge \neg S$ connected to R_i **continue**
    $p_1 = B[S], p_2 = B[\{R_i\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S \cup \{R_i\}] = \epsilon \vee C(B[S \cup \{R_i\}]) > C(P)$
      $B[S \cup \{R_i\}] = P$
  }
}
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

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

- 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 $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < i \leq 2^n - 1$ **ascending** {
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for each $R_j \in S$ {
    if ¬cross products $\wedge \neg S \setminus \{R_j\}$ connected to R_j **continue**
    $p_1 = B[S \setminus \{R_j\}], p_2 = B[\{R_j\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
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
$B =$ 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 $S_1, S_2 \subset R : |S_1| + |S_2| = s$ {
    if ($\neg$cross products $\wedge \neg S_1$ connected to S_2) $\vee$ ($S_1 \cap S_2 \neq \emptyset$) **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S_1 \cup S_2] = \epsilon \vee C(B[S_1 \cup S_2]) > C(P)$
      $B[S_1 \cup S_2] = P$
  }
}
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** $\{$
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for each $S_1 \subset S, S_2 = S \setminus S_1$ $\{$
    if $\neg$cross products $\wedge \neg S_1$ connected to S_2 **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  $\}$
$\}$
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** $(S_1 != S)$

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

Remarks

- DPsize/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)
- DPsub/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
if $B[S] = \epsilon$ {
  for each $S_1 \subset S, S_2 = S \setminus S_1$
    p_1 =MemoizationRec(B, S_1), p_2 =MemoizationRec(B, S_2)
    P=CreateJoinTree(p_1, p_2)
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[S]$

- checks for connectedness omitted

Dynamic Programming - Connected Subgraphs

- DP a very versatile strategy
- common usage scenario: bushy, no cross produts
- DPsize and DPsub support it, of course, but not optimal
- enumeration order does not consider the query graph
- many pairs have to be pruned due to conectedness
- especially bad for DPsub

Solution: consider the query graph structure during DP enumeration [5]

Asymptotic Search Space

DPsize:

- organize DP by the size of the join tree
- problem: only few DP slots, many pairs considered

good algorithm for chains, very bad for cliques:

	chains	cycles	stars	cliques
pairs	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$

DPsub:

- organize DP by the set of relations involved
- problem: always 2^n DP slots, fixed enumeration

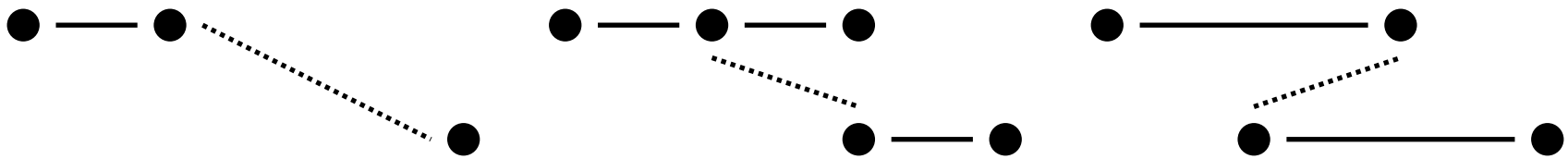
good algorithm for cliques, but adapts badly:

	chains	cycles	stars	cliques
pairs	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$

Observation

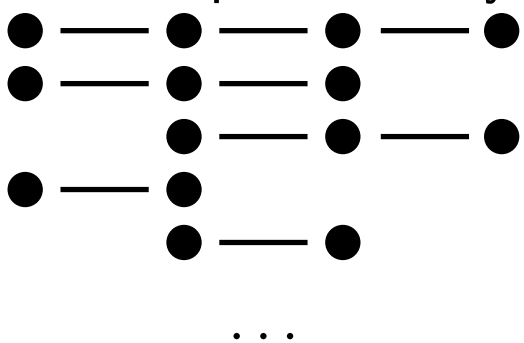
DPsize and DPsub generate many pairs that are pruned anyway (connectedness, overlap).

Typical pruned pairs (chain with 4 relations):



not connected | not disjoint | invalid subproblems

last example $\Rightarrow$ every join partner must be a connected subgraph:



Graph Theoretic Approach

- reformulation as graph theoretic problem:
- enumerate all connected subgraphs of the query graph
- for each subgraph enumerate all other connected subgraphs that are disjoint but connected to it
- each connected subgraph - complement pair (ccp) can be joined
- enumerate them suitable for DP $\Rightarrow$ DP algorithm

algorithm adapts naturally to the graph structure:

	chains	cycles	stars	cliques
pairs	$O(n^3)$	$O(n^3)$	$O(n2^n)$	$O(3^n)$

Lohman et al: #ccp is a lower bound for all DP enumeration algorithms

DP Algorithm using Connected Subgraphs

If we can efficiently enumerate all connected subgraphs/connected complement pairs, the resulting DP algorithm is:

DPccp(R)
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
B = an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall$ csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ {
  $p_1 = B[S_1]$, $p_2 = B[S_2]$
  P = CreateJoinTree(p_1, p_2);
  if $B[S] = \epsilon \vee C(B[S]) > C(P)$
    $B[S] = P$
}
return $B[\{R_0, \ldots, R_{n-1}\}]$

The main problem is enumerating the pairs.

Effect on Search Space

Absolute number of generated pairs

	Chain			Star		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	32	130	110
10	165	3,962	1,135	2,304	38,342	57,888
15	560	130,798	5,628	114,688	9,533,170	57,305,929
20	1,330	4,193,840	17,545	4,980,736	2,323,474,358	59,892,991,338
	Cycle			Clique		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	40	140	120	90	180	280
10	405	11,062	2,225	28,501	57,002	306,991
15	1,470	523,836	11,760	7,141,686	14,283,372	307,173,877
20	3,610	22,019,294	37,900	1,742,343,625	3,484,687,250	309,338,182,241

Enumerating Connected Subgraphs

- two steps: enumerate all connected subgraphs, enumerate disjoint but connected subgraphs for a given one $\Rightarrow$ pairs
- enumerate all pairs, enumerate no duplicates, enumerate for DP
- if (a, b) is enumerated, do not enumerate (b, a)
- requires total ordering of connected subgraphs
- preparation: label nodes breadth-first from 0 to $n - 1$

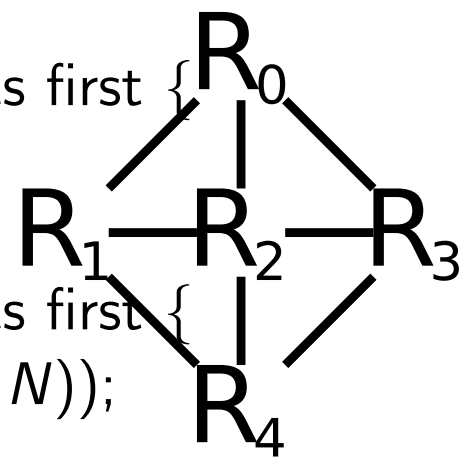
Preliminaries, given query graph $G = (V, E)$:

$$\begin{aligned} V &= \{v_0, \ldots, v_{n-1}\} \\ \mathcal{N}(V') &= \{v' | v \in V' \wedge (v, v') \in E\} \\ \mathcal{B}_i &= \{v_j | j \leq i\} \end{aligned}$$

Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

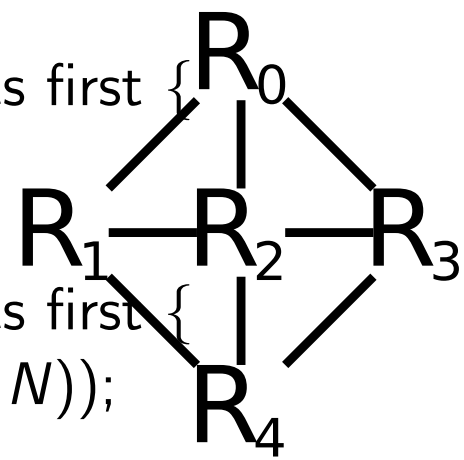


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Choose all nodes as enumeration start node once

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

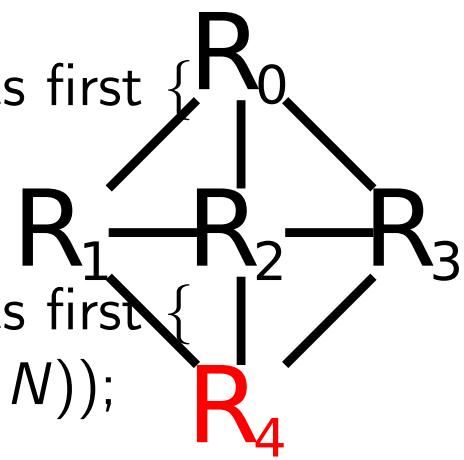


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

First emit only the node itself as subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

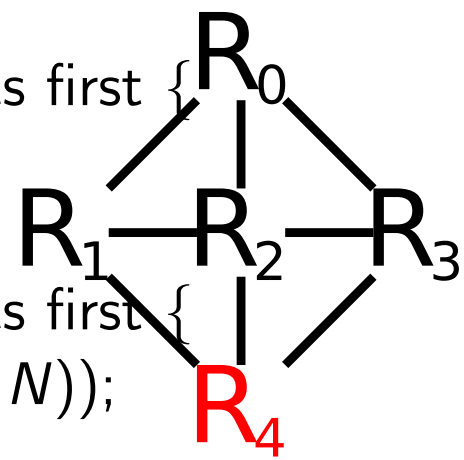


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Then enlarge the subgraph recursively

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

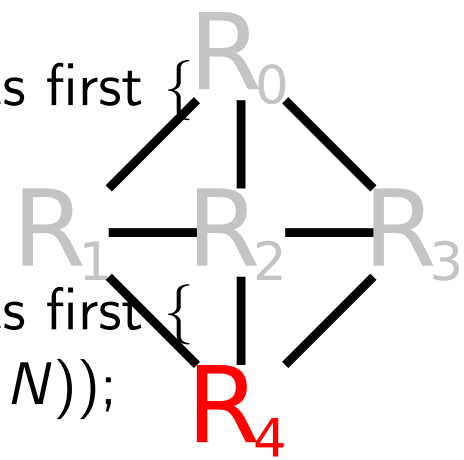


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
  emit $\{v_i\}$;
  EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Prohibit nodes with smaller labels. Thus the set of valid nodes increases over time

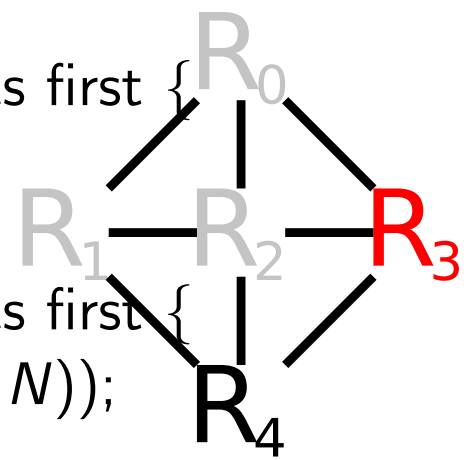
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

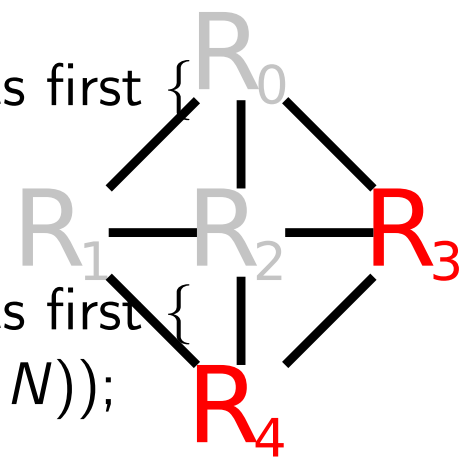
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

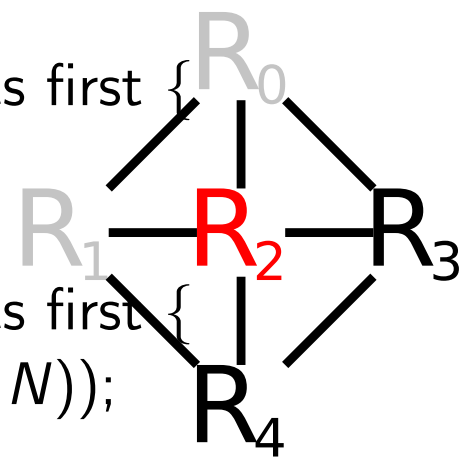
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

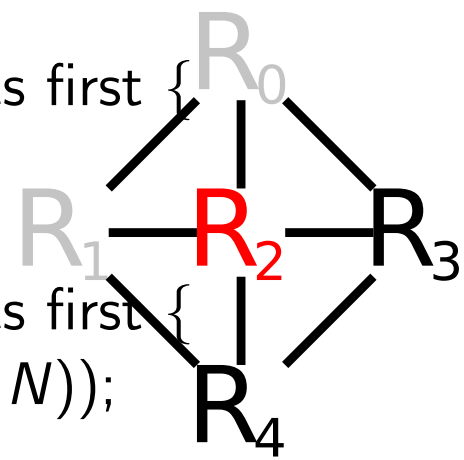


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
  emit $\{v_i\}$;
  EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

In each recursion, find all neighboring nodes that are not prohibited

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

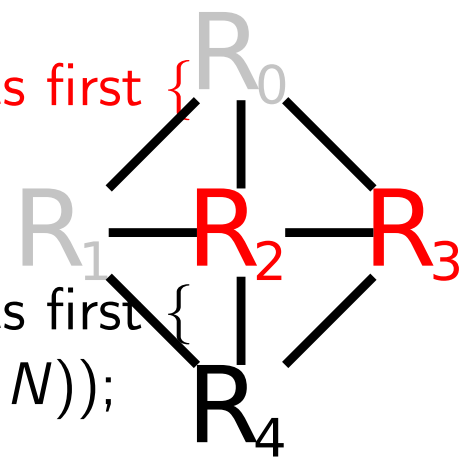


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Add all combinations to the subgraph and emit the new subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

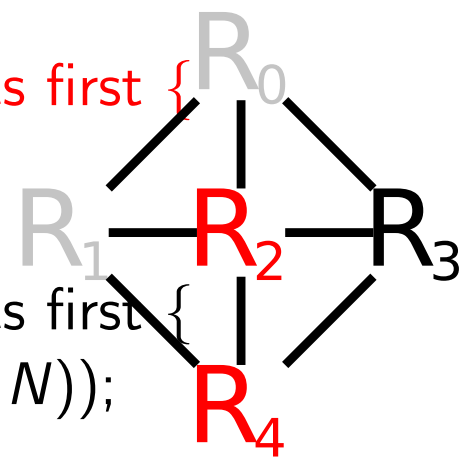


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Add all combinations to the subgraph and emit the new subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

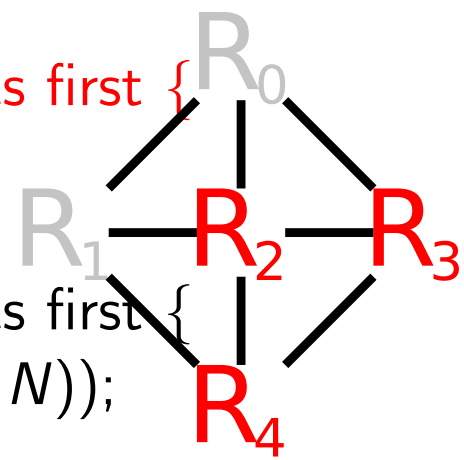


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Add all combinations to the subgraph and emit the new subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

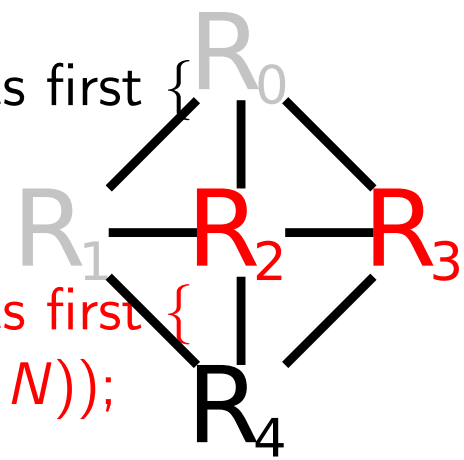


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Then, add all combinations to the subgraph and increase recursively

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

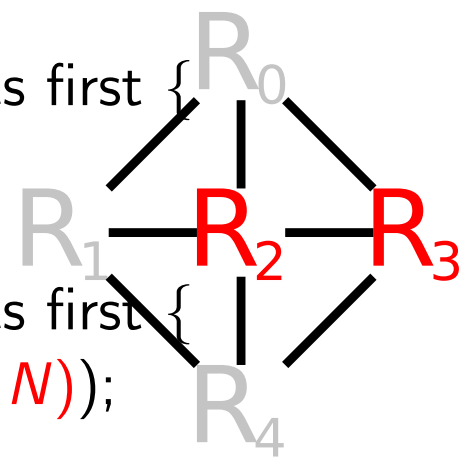


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

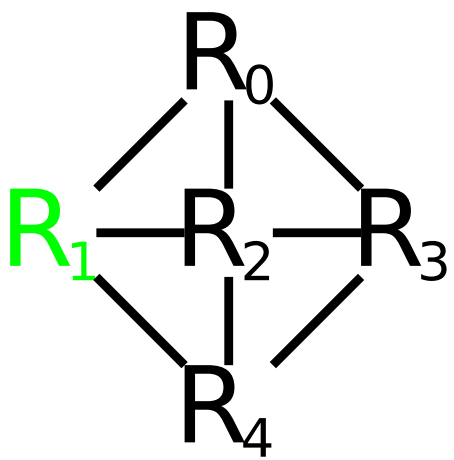
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

The neighborhood is prohibited during recursion, preventing duplicates



Enumerating Complementary Subgraphs

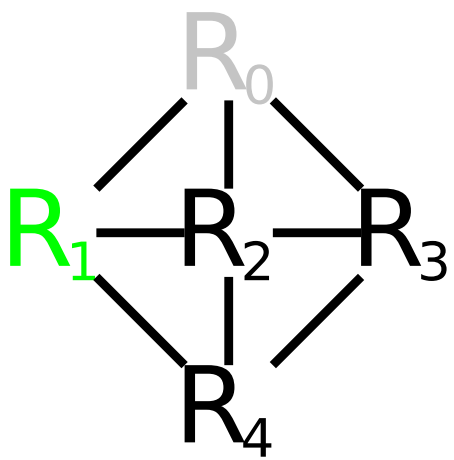
EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

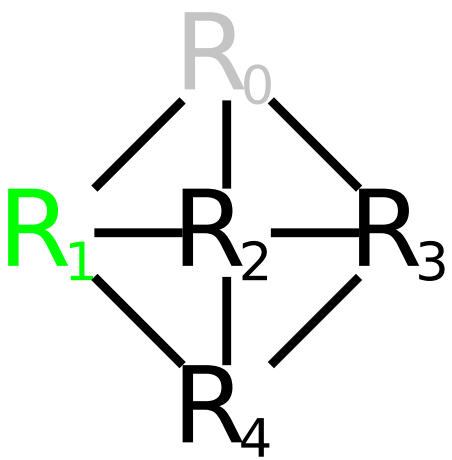
Prohibit all nodes that will be start nodes later on and the primary subgraph



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

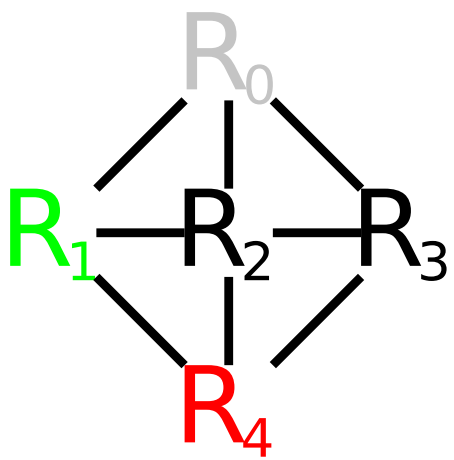
Find all neighboring nodes that are not prohibited



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

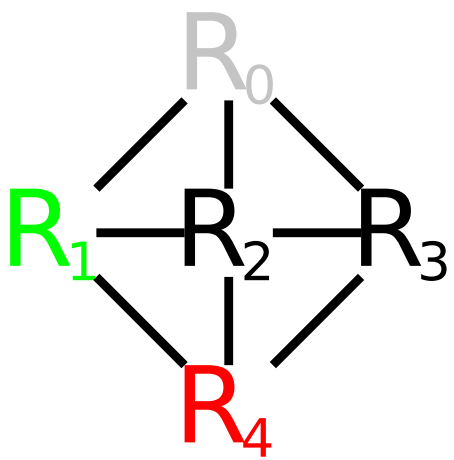
Consider each of the nodes



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

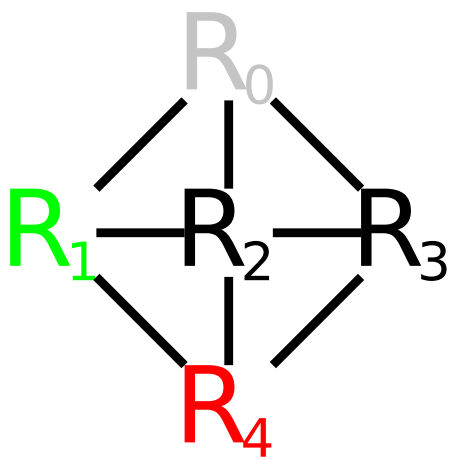
Choose the node as complementary subgraph and emit it



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

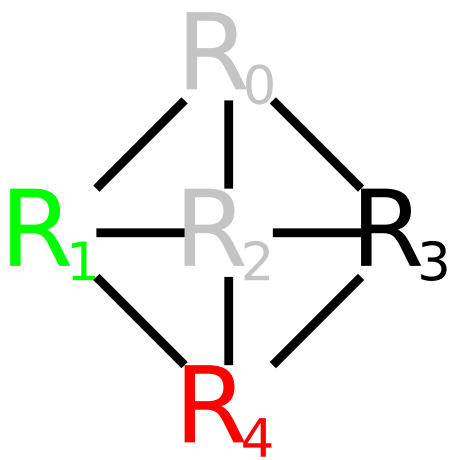
Recursively increase the subgraph re-using EnumerateCsgRec



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

Again prohibit nodes with a smaller label to prevent duplicates



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

- EnumerateCsg+EnumerateCmp produce all ccp
- resulting algorithm DPccp considers exactly #ccp pairs
- which is the lower bound for all DP enumeration algorithms

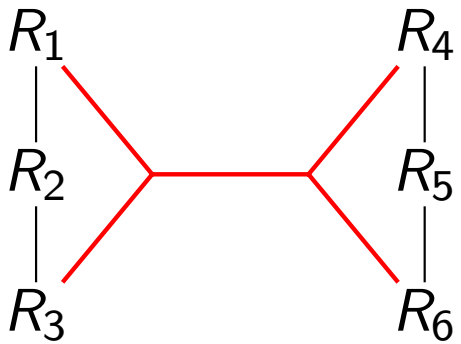
Remarks

- DPsize is good for chains, DPsub for cliques
- implementation of DPccp is more involved
- each enumeration step must be fast (ideally $O(1)$, at most $O(n)$, where n is the number of relations)
- but benefits are huge
- DPccg adopts to query graph structure
- considers minimal number of pairs
- especially for "in-between queries" (e.g. stars) much faster

Beyond (Regular) Query Graphs

Some queries are more complex

select *
from $R_1\ r_1,\ R_2\ r_2,\ R_3\ r_3,$
$R_4\ r_4,\ R_5\ r_5,\ R_6\ r_6$
where $r_1.a{=}r_2.a$ **and** $r_2.b{=}r_3.c$ **and**
$r_4.d{=}r_5.d$ **and** $r_5.e{=}r_6.e$ **and**
$\text{abs}(r_1.f + r_3.f)$
$= \text{abs}(r_4.g + r_6.g)$



- does not induce a graph but a hyper-graph
- graph based DP algorithm not directly applicable
- generic DP algorithms work, but not as efficient

Handling Hypergraphs

A *hypergraph* is a pair $H = (V, E)$ such that

1. V is a non-empty set of nodes and
2. E is a set of hyperedges, where a *hyperedge* is an unordered pair (u, v) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that $u \cap v = \emptyset$.

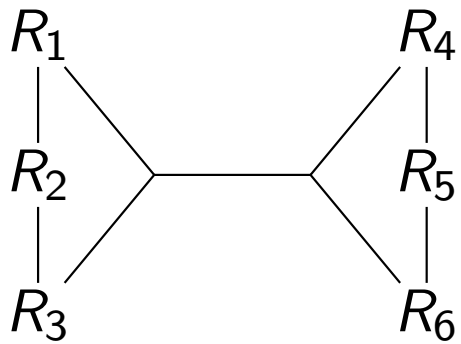
Nodes in V are totally ordered via an (arbitrary) relation $\prec$.

- enumeration is performed by decreasing $\prec$
- $\prec$ orders the search space (DP order, duplicates)

Handling Hypergraphs (2)

In principle same approach as for regular graphs:

- start with one node
- expand recursively by following edges



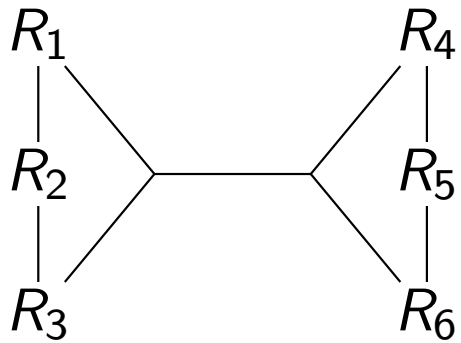
Problem:

- hyperedges are n:m edges
- where to expand to from $\{R_1, R_2, R_3\}$?
- must still guarantee DP order

Handling Hypergraphs - Neighborhood

When computing the neighborhood, choose representatives:

- a hyperedge "leads" to the least node (regarding $\prec$)
- therefore $N(\{R_1, R_2, R_3\}) = \{R_4\}$
- ensures DP order (and prevents duplicates)



But:

- leads to (temporarily) disconnected graphs
- $\{R_1, R_2, R_3, R_4\}$ is not connected
- must expand further until R_6 reached

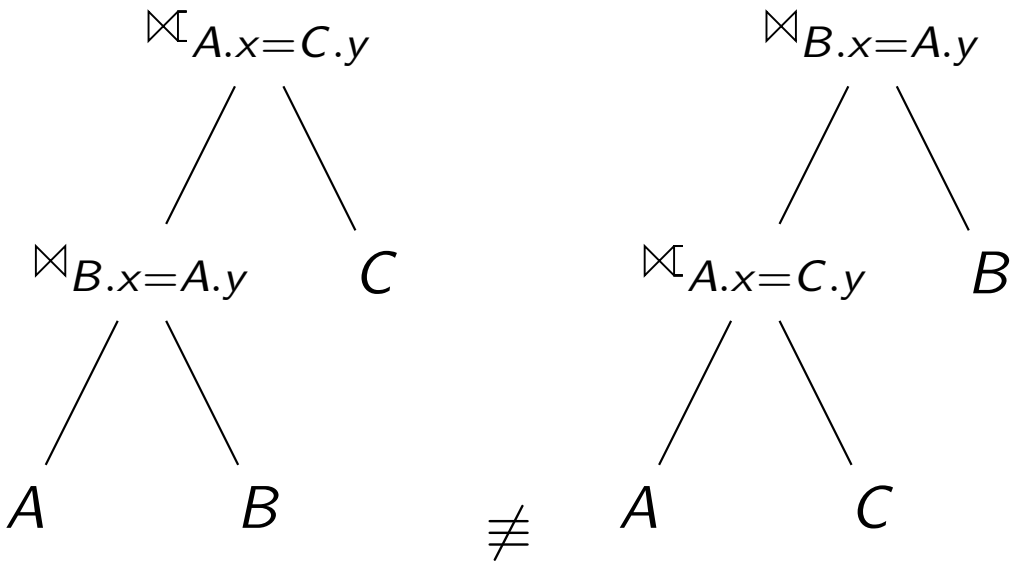
Requires checks for connectedness

- can exploit the DP table for cheap tests
- if it is connected, a DP entry must exist

Non-Inner Joins

Some queries use non-inner joins:

- either explicitly (*OUTER JOIN* etc.) or implicitly (unnesting etc.)
- are not freely reorderable



Must be taken into account during join ordering.

Non-Inner Joins - Reordering Constraints

Examine pair-wise reorderings of operators

- for all $\circ_1, \circ_2$, check if $(R \circ_1 S) \circ_2 T \equiv R \circ_1 (S \circ_2 T)$
- assume syntax constraints are satisfied

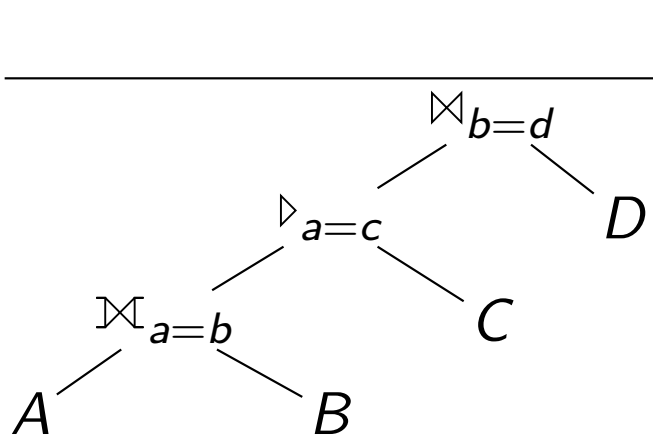
Gives a big compatibility matrix

	$\bowtie$	$\overrightarrow{\bowtie}$	$\overleftrightarrow{\bowtie}$	$\triangleright$	$\ltimes$	$\overrightarrow{\ltimes}$	...
$\bowtie$	+	+	-	+	+	+	...
$\overrightarrow{\bowtie}$	-	+	-	-	-	-	...
$\overleftrightarrow{\bowtie}$	-	+	+	-	-	-	...
$\triangleright$	-	-	-	-	-	-	...
$\ltimes$	-	-	-	-	-	-	...
$\overrightarrow{\ltimes}$	-	-	-	-	-	-	...
...							

Non-Inner Joins - TESs

Extract reordering constraints from operator tree in two steps:

1. build the *syntactic eligibility set* (SES) for each operator
   - set of relations that has to be in the input

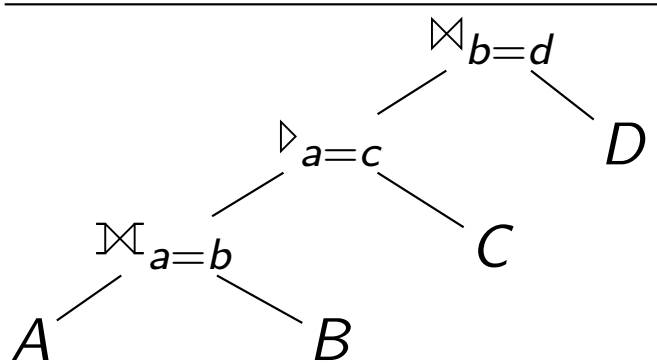


	SES
$\bowtie_{b=d}$	$\{B, D\}$
$\triangleright_{a=c}$	$\{A, C\}$
$\fullouterjoin_{a=b}$	$\{A, B\}$

Non-Inner Joins - TESs

Extract reordering constraints from operator tree in two steps:

1. build the *syntactic eligibility set* (SES) for each operator
2. bottom up traversal, build the *total eligibility set* (TES)
   - initialize TES with SES
   - check for conflicts with other operators (can be in subtrees!)
   - if conflict, add other TES to own TES



	SES	TES
$\bowtie_{b=d}$	$\{B, D\}$	$\{A, B, D\}$
$\triangleright_{a=c}$	$\{A, C\}$	$\{A, B, C\}$
$\bowtie_{a=b}$	$\{A, B\}$	$\{A, B\}$

TESs capture reordering restrictions by requiring relations, which imply operators.

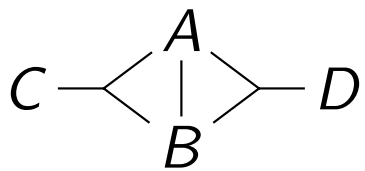
Non-Inner Joins - Using TESs

Add the TES to the join edge

- operator "requires" certain relations, so encode it like this
- constructs hyperedges (n:m)
- eliminates invalid reorderings from the search space

Original query graph from previous example: $C - A - B - D$

After adding TESs to the edges:



Simplifying the Query Graph

The graph-based DP algorithm considers the minimal number of join-pairs

- we therefore cannot expect to get a better runtime for exact solutions
- many problems can be solved exactly, but not all
- depends on the structure of the query graph
- chains are simple, others, e.g., stars, are hard
- how to cope with these queries?

Greedy heuristics would work, but results are much worse than DP solutions.

Simplifying the Query Graph - General Idea

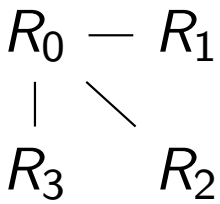
If the problem is too complex to solve exactly, simplify the query graph until it gets tractable.

- the query graph describes all join possibilities
- by modifying the query graph we can rule out some possibilities
- this reduces the search space and the optimization time
- we prefer modifications that are "safe"
- uses greedy steps only for the "easy" problems, then use DP

Note: "simplifying" means simpler for the optimizer.
For a human the query graph tends to get strange.

Simplifying a Star Query

graph

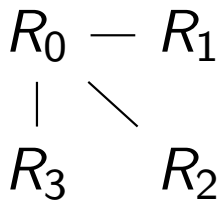


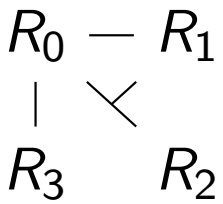
joins

$R_0 \bowtie R_1$
$R_0 \bowtie R_2$
$R_0 \bowtie R_3$
original

search space size
6

Simplifying a Star Query

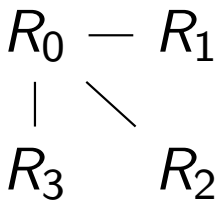
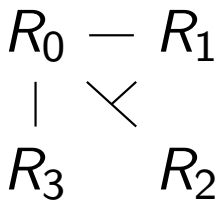
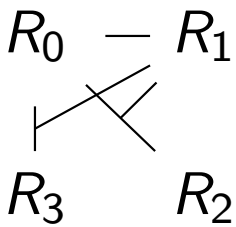




graph			
joins	$R_0 \bowtie R_1$	$R_0 \bowtie R_1$	search space size
	$R_0 \bowtie R_2$	$\{R_0, R_1\} \bowtie R_2$	$\not{6}$
	$R_0 \bowtie R_3$	$R_0 \bowtie R_3$	3
	original	1st step	

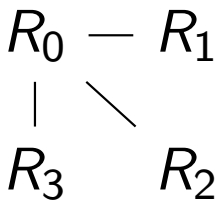
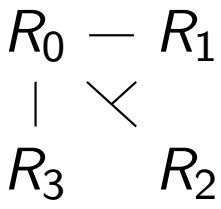
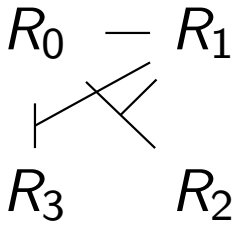
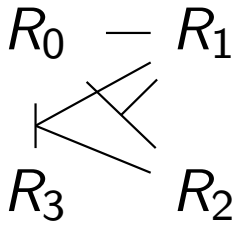
We decide to order $R_0 \bowtie R_1$ before $R_0 \bowtie R_2$ (introduces hyperedge)

Simplifying a Star Query

			search space size
graph			
joins	$R_0 \bowtie R_1$ $R_0 \bowtie R_2$ $R_0 \bowtie R_3$	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $R_0 \bowtie R_3$	
	original	1st step	
			$\not{6}$
			$\not{3}$
			2
graph			
joins	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1\} \bowtie R_3$		
	2nd step		

We decide to order $R_0 \bowtie R_1$ before $R_0 \bowtie R_3$ (introduces hyperedge)

Simplifying a Star Query

			search space size
graph			
joins	$R_0 \bowtie R_1$ $R_0 \bowtie R_2$ $R_0 \bowtie R_3$	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $R_0 \bowtie R_3$	
	original	1st step	
			~~6~~
graph			~~3~~
			~~2~~
joins	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1\} \bowtie R_3$	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1, R_2\} \bowtie R_3$	1
	2nd step	3rd step	

We decide to order $\{R_0, R_1\} \bowtie R_2$ before $R_0 \bowtie R_3$ (introduces hyperedge)

Performing A Simplification Step

Given a query graph $G = (V, E)$

1. examine all joins $\bowtie_1, \bowtie_2 \in E$ that are *neighboring*
   - neighboring $\approx$ have a relation in common (see [6])
2. make sure that $\bowtie_2$ could be ordered before $\bowtie_1$
   - checks for contradictions, requires a fast cycle checker
3. compute the *orderingBenefit*($\bowtie_1$,$\bowtie_2$)
   - this is the heuristical part, different benefit heuristics could be used
4. retain the $S_1^L \bowtie_1 S_1^R, S_2^L \bowtie_2 S_2^R$ with the maximal orderingBenefit
   - maintain priority queues to speed up repeated simplification
5. return $G' = (V, E \setminus \{\bowtie_1\} \cup \{(S_1^L \cup S_2^L \cup S_2^R) \bowtie_1 S_1^R\})$

The resulting query graph is more restrictive, i.e., simpler.

(there are more cases due to different possible ways of neighboring)

Estimating the Ordering Benefit

We want to prefer orderings that are almost certainly a good idea. Therefore one approach is to maximize

$$\text{orderingBenefit}(X \bowtie_1 R_1, X \bowtie_2 R_2) = \frac{C((X \bowtie_1 R_1) \bowtie_2 R_2)}{C((X \bowtie_2 R_2) \bowtie_1 R_1)}$$

If we cannot compute C due to missing information, use C_{out}.

Adjusting the Problem Complexity

How much should we simplify?

- until optimization fits into resource constraints (memory or time)

How do we know when to stop simplifying?

- count the number of **connected subgraphs** of the query graph
- directly determines memory, indirectly optimization time
- stop counting when the limit is reached

Counting is fast, but not instantaneous

- counting 10,000 subgraphs in a query with 100 relations took $\approx$ 5ms
- we cannot do this after every simplification

Exact limit depends on hardware, a reasonable choice is 10,000 connected subgraphs.

Full Optimization Algorithm

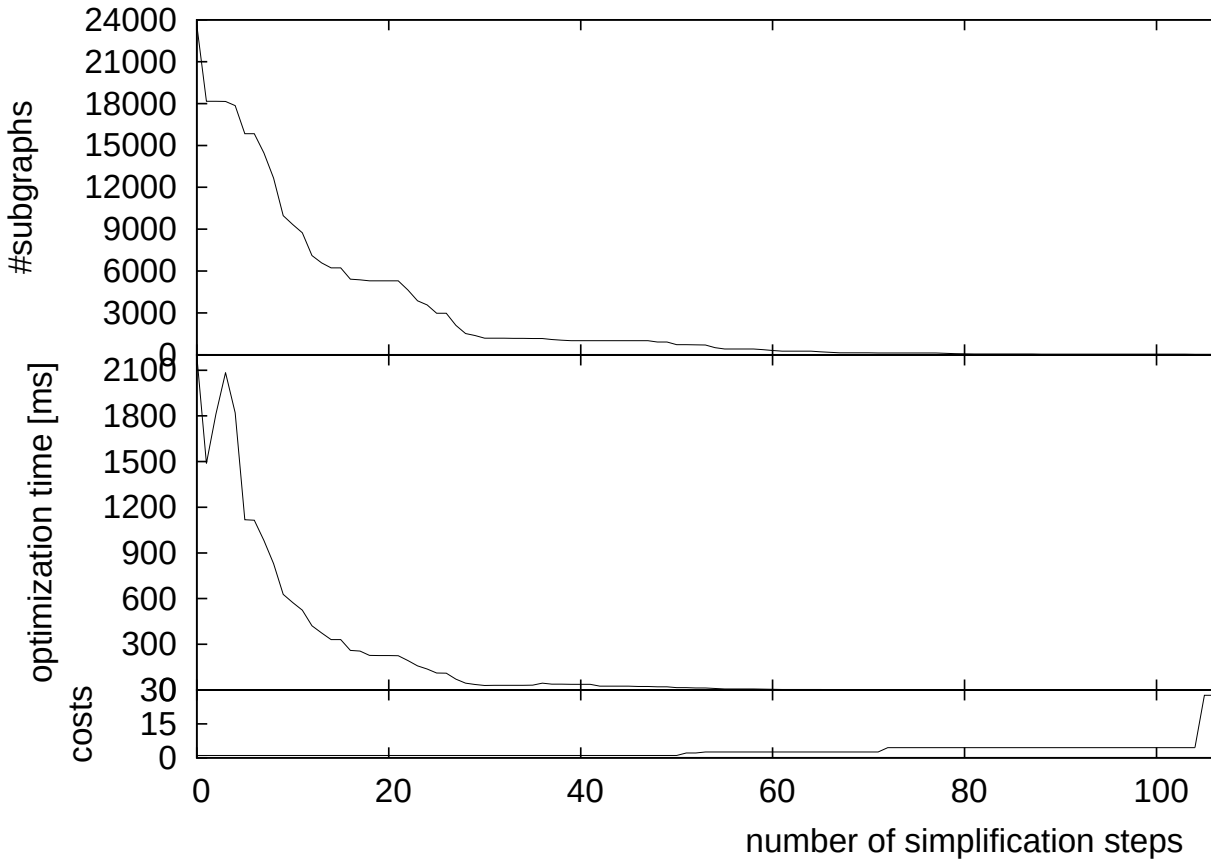
Given a Query Graph $G = (V, E)$ and a complexity budget b

1. compute a list $\bar{G}$ of query graphs
   - repeatedly call the simplification step, stop when no change
2. perform binary search over $\bar{G}$, find G_b
   - for the current element G', $c = \#$connected subgraphs in G' (count at most $b + 1$)
   - if $c > b$ increase, otherwise decrease
3. return $DPhyp(G_b)$

Simplifies as much as needed to meet the constraints, than uses DP.

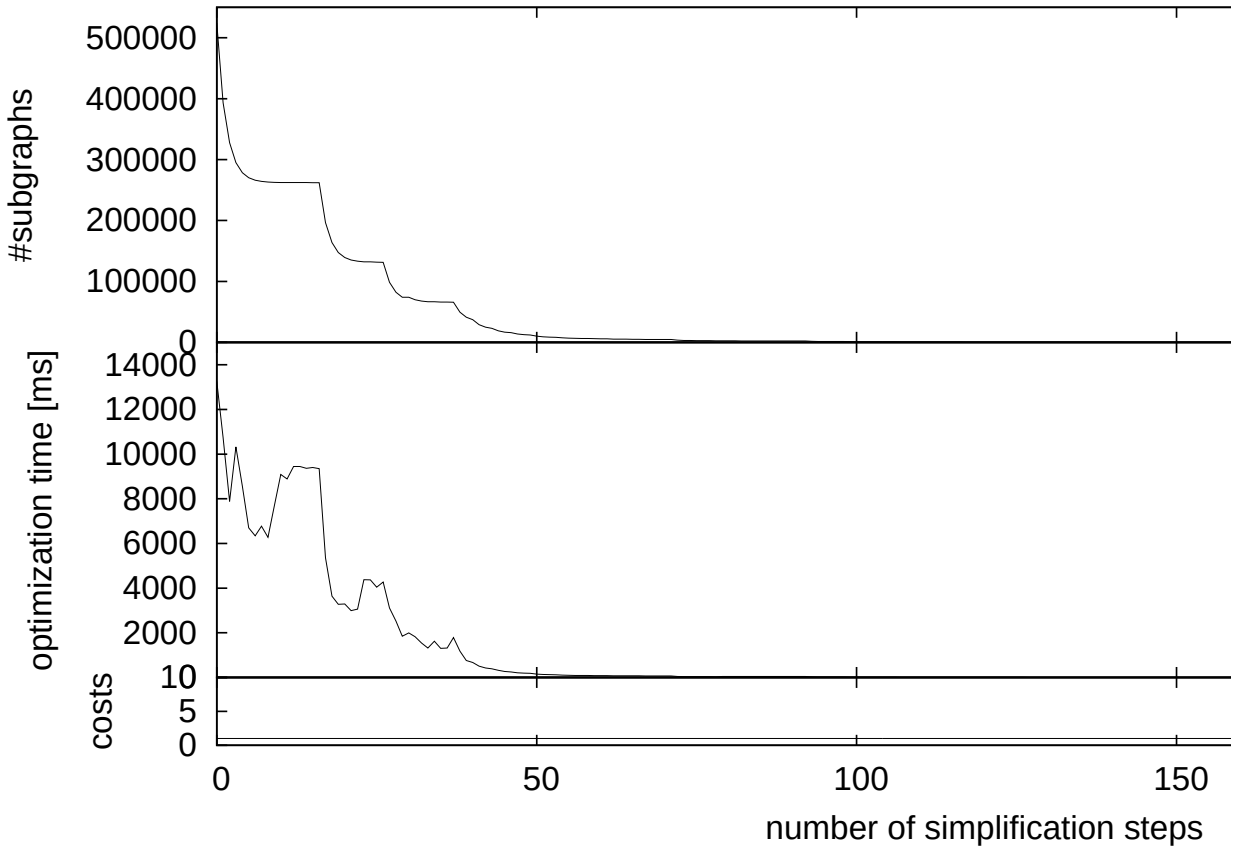
(the algorithm does not materialize $\bar{G}$ explicitly, see [6])

Time/Quality Trade-off - Grid with 20 Relations



- as expected plan quality degrades at some point
- but optimization times drops off much earlier

Time/Quality Trade-off - Star with 20 Relations



- same optimization time behavior, but plan quality remains perfect

Generating Permutations

The algorithms so far have some drawbacks:

- greedy heuristics only heuristics
- will probably not find the optimal solution
- DP algorithms optimal, but very heavy weight
- especially memory consumption is high
- find a solution only after the complete search

Sometimes we want a more light-weight algorithm:

- low memory consumption
- stop if time runs out
- still find the optimal solution if possible

Generating Permutations (2)

We can achieve this when only considering left-deep trees:

- left-deep trees are permutations of the relations to be joined
- permutations can be generated directly
- generating all permutations is too expensive
- but some permutations can be ignored:
  Consider the join sequence $R_1 R_2 R_3 R_4$. If we know that $R_1 R_3 R_2$ is cheaper than $R_1 R_2 R_3$, we do not have to consider $R_1 R_2 R_3 R_4$.

Idea: successively add a relation. An extended sequence is only explored if exchanging the last two relations does not result in a cheaper sequence.

Recursive Search

ConstructPermutations(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep join tree
$B = \epsilon$
$P = \epsilon$
for each $R_i \in R$ {
  ConstructPermutationsRec($P \circ < R_i >, R \setminus \{R_i\}, B$)
} **return** B

- algorithm considers a prefix P and the rest R
- keeps track of the best tree found so far B
- increases the prefix recursively

Recursive Search (2)

ConstructPermutationsRec(P, R, B)
Input: a prefix P, remaining relations R, best plan B
Output: side effects on B
if $|R| = 0$ {
  if $B = \epsilon \vee C(B) > C(P)$ {
    $B = P$
  }
} **else** {
  for each $R_i \in R$ {
    if $C(P \circ < R_i >) \leq C(P[1 : |P| - 1] \circ < R_i, P[|P|] >)$ {
      ConstructPermutationsRec($P \circ < R_i >, R \setminus \{R_i\}, B$)
    }
  }
}

Remarks

Good:

- linear memory
- immediately produces plan alternatives
- anytime algorithm
- finds the optimal plan eventually

Bad:

- worst-case runtime if ties occur
- worst-case runtime if no ties occur is an open problem

Often fast, can be stopped anytime, but may perform poorly.

Transformative Approaches

Main idea: [7]

- use equivalences directly (associativity, commutativity)
- would make integrating new equivalences easy

Problems:

- how to navigate the search space
- equivalences have no order
- how to guarantee finding the optimal solution
- how to avoid exhaustive search

Rule Set

$R_1 \bowtie R_2$	$\rightsquigarrow$	$R_2 \bowtie R_1$	Commutativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$R_1 \bowtie (R_2 \bowtie R_3)$	Right Associativity
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$(R_1 \bowtie R_2) \bowtie R_3$	Left Associativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$(R_1 \bowtie R_3) \bowtie R_2$	Left Join Exchange
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$R_2 \bowtie (R_1 \bowtie R_3)$	Right Join Exchange

Two more rules are often used to transform left-deep trees:

- *swap* exchanges two arbitrary relations in a left-deep tree
- *3Cycle* performs a cyclic rotation of three arbitrary relations in a left-deep tree.

To try another join method, another rule called *join method exchange* is introduced.

Rule Set RS-0

- commutativity
- left-associativity
- right-associativity

Basic Algorithm

ExhaustiveTransformation($\{R_1, \ldots, R_n\}$)
Input: a set of relations
Output: an optimal join tree
Let T be an arbitrary join tree for all relations
Done $= \emptyset$ // contains all trees processed
ToDo $= \{T\}$ // contains all trees to be processed
while $|\text{ToDo}| > 0$ {
    T = an arbitrary tree in ToDo
    ToDo = ToDo $\setminus T$;
    Done = Done $\cup \{T\}$;
    Trees = ApplyTransformations(T);
    for each $T \in$ Trees {
        if $T \notin$ ToDo $\cup$ Done
            ToDo = ToDo $\cup \{T\}$
    }
}
return $\arg\min_{T \in \text{Done}} C(T)$

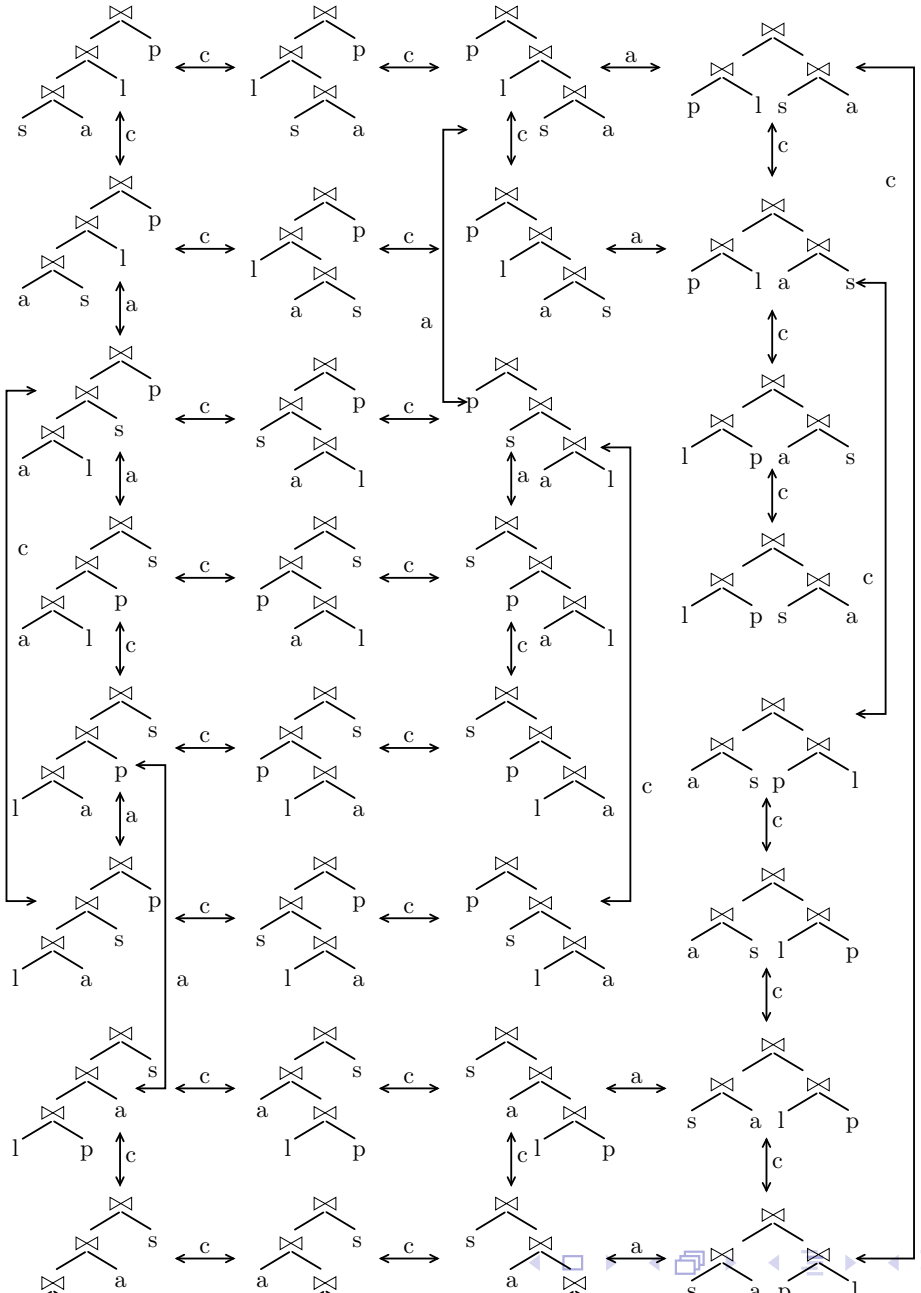
Basic Algorithm (2)

ApplyTransformations(T)
Input: join tree
Output: all trees derivable by associativity and commutativity
Trees $= \emptyset$
Subtrees $=$ all subtrees of T rooted at inner nodes
for each $S \in$ Subtrees $\{$
    if S is of the form $S_1 \bowtie S_2$
        Trees $=$ Trees $\cup \{S_2 \bowtie S_1\}$
    if S is of the form $(S_1 \bowtie S_2) \bowtie S_3$
        Trees $=$ Trees $\cup \{S_1 \bowtie (S_2 \bowtie S_3)\}$
    if S is of the form $S_1 \bowtie (S_2 \bowtie S_3)$
        Trees $=$ Trees $\cup \{(S_1 \bowtie S_2) \bowtie S_3\}$
$\}$
return Trees;

Remarks

- if no cross products are to be considered, extend **if** conditions for associativity rules.
- problem 1: explores the whole search space
- problem 2: generates join trees more than once
- problem 3: sharing of subtrees is non-trivial

Search Space



Introducing the Memo Structure

A memoization strategy is used to keep the runtime reasonable:

- for any subset of relations, dynamic programming remembers the best join tree.
- this does not quite suffice for the transformation-based approach.
- instead, we have to keep all join trees generated so far including those differing in the order of the arguments of a join operator.
- however, subtrees can be shared.
- this is done by keeping pointers into the data structure (see next slide).

Memo Structure Example

$\{R_1, R_2, R_3\}$	$\{R_1, R_2\} \bowtie R_3, R_3 \bowtie \{R_1, R_2\},$ $\{R_1, R_3\} \bowtie R_2, R_2 \bowtie \{R_1, R_3\},$ $\{R_2, R_3\} \bowtie R_1, R_1 \bowtie \{R_2, R_3\}$
$\{R_2, R_3\}$	$\{R_2\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_2\}$
$\{R_1, R_3\}$	$\{R_1\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_1\}$
$\{R_1, R_2\}$	$\{R_1\} \bowtie \{R_2\}, \{R_2\} \bowtie \{R_1\}$
$\{R_3\}$	R_3
$\{R_2\}$	R_2
$\{R_1\}$	R_1

- in Memo Structure: arguments are pointers to classes
- Algorithm: `ExploreClass` expands a class
- Algorithm: `ApplyTransformation2` expands a member of a class

Memoizing Algorithm

ExhaustiveTransformation2(Query Graph G)
Input: a query specification for relations $\{R_1, \ldots, R_n\}$.
Output: an optimal join tree
initialize MEMO structure
ExploreClass($\{R_1, \ldots, R_n\}$)
return $\arg\min_{T \in \text{class } \{R_1, \ldots, R_n\}} C(T)$

- stored an arbitrary join tree in the memo structure
- explores alternatives recursively

Memoizing Algorithm (2)

ExploreClass(C)
Input: a class $\mathcal{C} \subseteq \{R_1, \ldots, R_n\}$
Output: none, but has side-effect on MEMO-structure
while not all join trees in C have been explored {
    choose an unexplored join tree T in $\mathcal{C}$
    ApplyTransformation2(T)
    mark T as explored
}

- considers all alternatives within one class
- transformations themselves are done in ApplyTransformation2

Memoizing Algorithm (3)

ApplyTransformations2(T)
Input: a join tree of a class $\mathcal{C}$
Output: none, but has side-effect on MEMO-structure
ExploreClass(left-child(T))
ExploreClass(right-child(T));
for each transformation $\mathcal{T}$ and class member of child classes {
    for each T' resulting from applying $\mathcal{T}$ to T {
        if T' not in MEMO structure {
            add T' to class $\mathcal{C}$ of MEMO structure
        }
    }
}

- first explores subtrees
- then applies all known transformations to the tree
- stores new trees in the memo structure

Remarks

- Applying `ExhaustiveTransformation2` with a rule set consisting of Commutativity and Left and Right Associativity generates $4^n - 3^{n+1} + 2^{n+2} - n - 2$ duplicates
- Contrast this with the number of join trees contained in a completely filled MEMO structure: $3^n - 2^{n+1} + n + 1$
- Solve the problem of duplicate generation by disabling applied rules.

Rule Set RS-1

T_1: Commutativity $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$

Disable all transformations T_1, T_2, and T_3 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$

Disable transformations T_2 and T_3 for $\bowtie_2$ and enable all rules for $\bowtie_3$.

T_3: Left associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$

Disable transformations T_2 and T_3 for $\bowtie_3$ and enable all rules for $\bowtie_2$.

Example for chain $R_1 - R_2 - R_3 - R_4$

Class	Initialization	Transformation	Step
$\{R_1, R_2, R_3, R_4\}$	$\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$	$\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$	3
		$R_1 \bowtie_{100} \{R_2, R_3, R_4\}$	4
		$\{R_1, R_2, R_3\} \bowtie_{100} R_4$	5
		$\{R_2, R_3, R_4\} \bowtie_{000} R_1$	8
		$R_4 \bowtie_{000} \{R_1, R_2, R_3\}$	10
$\{R_2, R_3, R_4\}$		$R_2 \bowtie_{111} \{R_3, R_4\}$	4
		$\{R_3, R_4\} \bowtie_{000} R_2$	6
		$\{R_2, R_3\} \bowtie_{100} R_4$	6
		$R_4 \bowtie_{000} \{R_2, R_3\}$	7
$\{R_1, R_3, R_4\}$			
$\{R_1, R_2, R_4\}$			
$\{R_1, R_2, R_3\}$		$\{R_1, R_2\} \bowtie_{111} R_3$	5
		$R_3 \bowtie_{000} \{R_1, R_2\}$	9
		$R_1 \bowtie_{100} \{R_2, R_3\}$	9
		$\{R_2, R_3\} \bowtie_{000} R_1$	9
$\{R_3, R_4\}$	$R_3 \bowtie_{111} R_4$	$R_4 \bowtie_{000} R_3$	2
$\{R_2, R_4\}$			
$\{R_2, R_3\}$			
$\{R_1, R_4\}$			
$\{R_1, R_3\}$			
$\{R_1, R_2\}$	$R_1 \bowtie_{111} R_2$	$R_2 \bowtie_{000} R_1$	1

Rule Set RS-2

Bushy Trees: Rule set for clique queries and if cross products are allowed:

T_1: Commutativity $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2, T_3, and T_4 for $\bowtie_2$.

T_3: Left Associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2, T_3 and T_4 for $\bowtie_3$.

T_4: Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 (C_3 \bowtie_2 C_4) \rightsquigarrow (C_1 \bowtie_3 C_3) \bowtie_4 (C_2 \bowtie_5 C_4)$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_4$.

If we initialize the MEMO structure with left-deep trees, we can strip down the above rule set to Commutativity and Left Associativity. Reason: from a left-deep join tree we can generate all bushy trees with only these two rules

Rule Set RS-3

Left-deep trees:

T_1 Commutativity $R_1 \bowtie_0 R_2 \rightsquigarrow R_2 \bowtie_1 R_1$

Here, the R_i are restricted to classes with exactly one relation. T_1 is disabled for $\bowtie_1$.

T_2 Right Join Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow (C_1 \bowtie_2 C_3) \bowtie_3 C_2$

Disable T_2 for $\bowtie_3$.

Generating Random Join Trees

Generating a random join tree is quite useful:

- allows for cost sampling
- randomized optimization procedures
- basis for Simulated Annealing, Iterative Improvement etc.
- easy with cross products, difficult without
- we consider with cross products first

Main problems:

- generating all join trees (potentially)
- creating all with the same probability

Ranking/Unranking

Let S be a set with n elements.

- a bijective mapping $f : S \to [0, n[$ is called *ranking*
- a bijective mapping $f : [0, n[\to S$ is called *unranking*

Given an unranking function, we can generate random elements in S by generating a random number in $[0, n[$ and unranking this number.
Challenge: making unranking fast.

Random Permutations

Every permutation corresponds to a left-deep join tree possibly with cross products.
Standard algorithm to generate random permutations is the starting point for the algorithm:

for each $k \in [0, n[$ **descending**
  swap$(\pi[k], \pi[\text{random}(k)])$

Array π initialized with elements $[0, n[$.
`random`(k) generates a random number in $[0, k]$.

Random Permutations

- Assume the random elements produced by the algorithm are $r_{n-1}, \ldots, r_0$ where $0 \leq r_i \leq i$.
- Thus, there are exactly $n(n-1)(n-2)\ldots 1 = n!$ such sequences and there is a one to one correspondance between these sequences and the set of all permutations.
- Unrank $r \in [0, n![$ by turning it into a unique sequence of values $r_{n-1}, \ldots, r_0$.
  Note that after executing the swap with r_{n-1} every value in $[0, n[$ is possible at position $\pi[n-1]$.
  Further, $\pi[n-1]$ is never touched again.
- Hence, we can unrank r as follows. We first set $r_{n-1} = r \mod n$ and perform the swap. Then, we define $r' = \lfloor r/n \rfloor$ and iteratively unrank r' to construct a permutation of $n-1$ elements.

Generating Random Permutations

Unrank(n, r)
Input: the number n of elements to be permuted
and the rank r of the permutation to be constructed
Output: a permutation π
for each $0 \leq i < n$
  $\pi[i] = i$
for each $n \geq i > 0$ **descending** $\{$
  swap($\pi[i-1], \pi[r \mod i]$)
  $r = \lfloor r/i \rfloor$
$\}$
return π;

Generating Random Bushy Trees with Cross Products

Steps of the algorithm:

1. Generate a random number b in $[0, C(n)[$.
2. Unrank b to obtain a bushy tree with $n-1$ inner nodes.
3. Generate a random number p in $[0, n![$.
4. Unrank p to obtain a permutation.
5. Attach the relations in order p from left to right as leaf nodes to the binary tree obtained in Step 2.

The only step that we have still to discuss is Step 2.

Tree Encoding

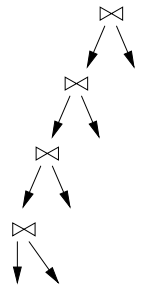
- Preordertraversal:
  - Inner node: '('
  - Leaf Node: ')'

  Skip last leaf node.
- Replace '(' by 1 and ')' by 0
- Just take positions of 1s.

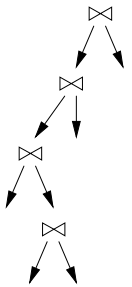
Example: all trees with four inner nodes:

- The ranks are in $[0, 14[$

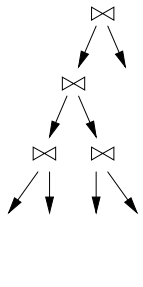
Tree Ranking Example



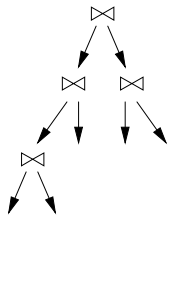
(((( ))))
11110000
1, 2, 3, 4
0



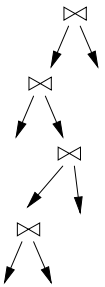
((( )( )))
11101000
1, 2, 3, 5
1



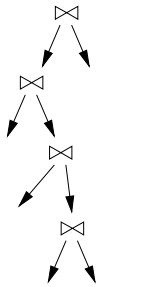
((( )) ( ))
11100100
1, 2, 3, 6
2



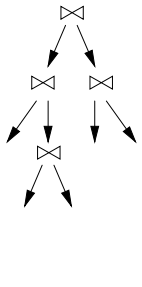
((( ))) ( )
11100010
1, 2, 3, 7
3



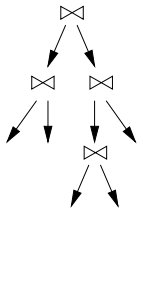
(( ) (( )))
11011000
1, 2, 4, 5
4



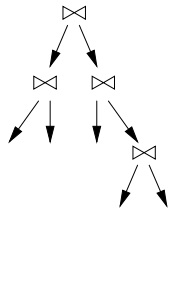
(( ) ( ) ( ))
11010100
1, 2, 4, 6
5



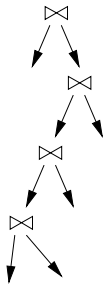
(( ) ( )) ( )
11010010
1, 2, 4, 7
6



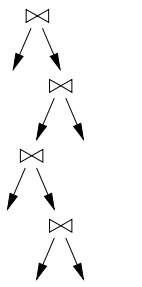
(( )) (( ))
11001100
1, 2, 5, 6
7



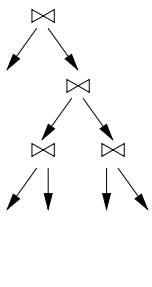
(( )) ( ) ( )
11001010
1, 2, 5, 7
8



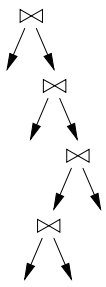
( ) ((( )))
10111000
1, 3, 4, 5
9



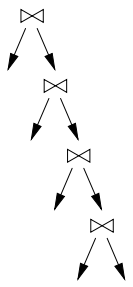
( ) (( ) ( ))
10110100
1, 3, 4, 6
10



( ) (( )) ( )
10110010
1, 3, 4, 7
11



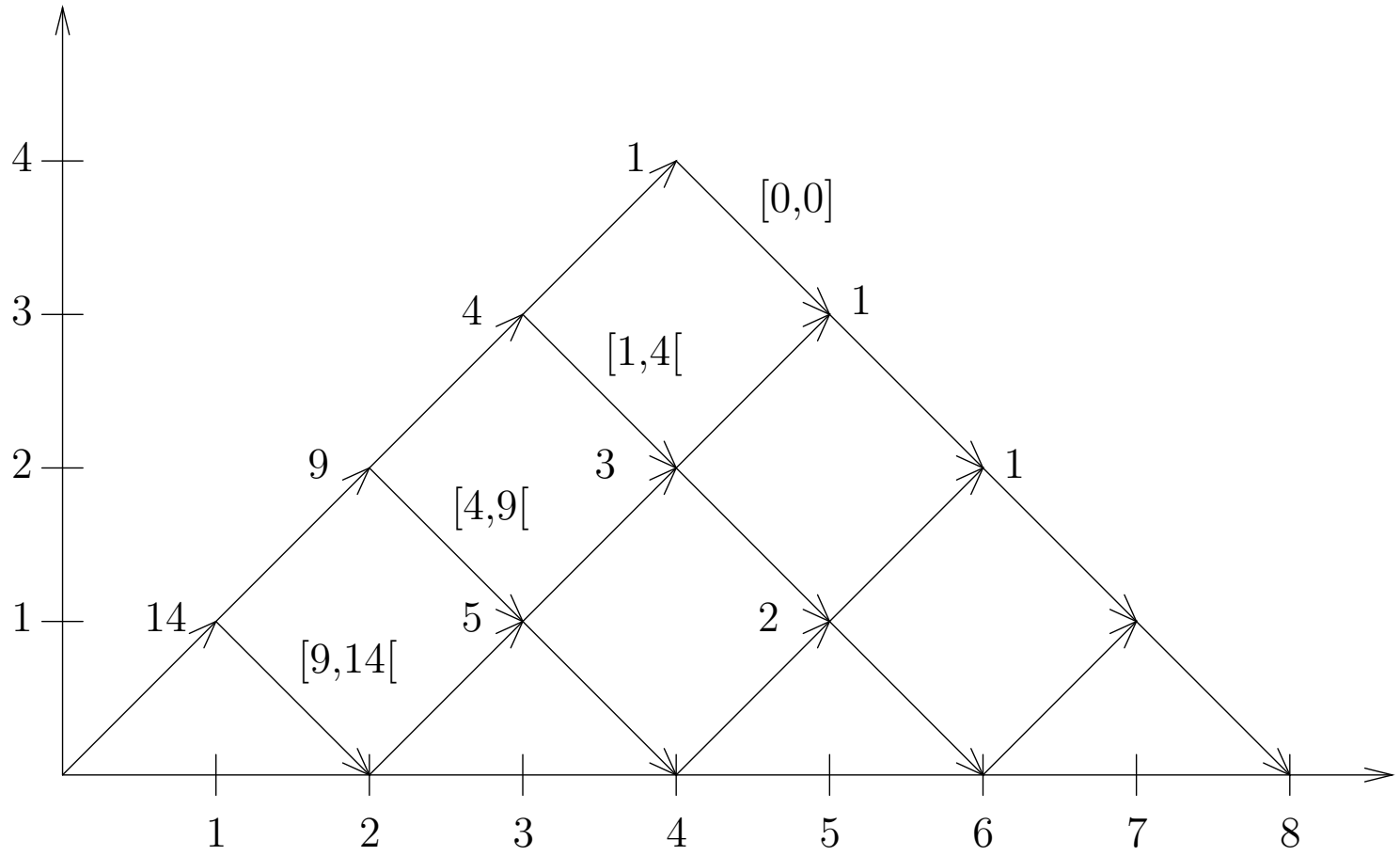
( ) ( ) (( ))
10101100
1, 3, 5, 6
12



( ) ( ) ( ) ( )
10101010
1, 3, 5, 7
13

Unranking Binary Trees

We establish a bijection between Dyck words and paths in a grid:



Every path from $(0, 0)$ to $(2n, 0)$ uniquely corresponds to a Dyck word.

Counting Paths

The number of different paths from $(0,0)$ to (i,j) can be computed by

$$p(i,j) = \frac{j+1}{i+1}\binom{i+1}{\frac{1}{2}(i+j)+1}$$

These numbers are the *Ballot numbers*.
The number of paths from (i,j) to $(2n,0)$ can thus be computed as:

$$q(i,j) = p(2n-i,j)$$

Note the special case $q(0,0) = p(2n,0) = C(n)$.

Unranking Outline

- We open a parenthesis (go from (i, j) to $(i + 1, j + 1)$) as long as the number of paths from that point does no longer exceed our rank r.
- If it does, we close a parenthesis (go from (i, j) to $(i + 1, j - 1)$).
- Assume, that we went upwards to (i, j) and then had to go down to $(i + 1, j - 1)$.
  We subtract the number of paths from $(i + 1, j + 1)$ from our rank r and proceed iteratively from $(i + 1, j - 1)$ by going up as long as possible and going down again.
- Remembering the number of parenthesis opened and closed along our way results in the required encoding.

Generating Bushy Trees

UnrankTree(n, r)
Input: a number of inner nodes n and a rank $r \in [0, C(n)[$
Output: encoding of the inner leafes of a tree
open $= 1$, close $= 0$
pos $= 1$, encoding $= < 1 >$
while $|$encoding$| < n$ {
  $k = q$(open+close,open-close)
  if $k \leq r$ {
    $r = r - k$, close=close+1
  } **else** {
    encoding=encoding$\circ < $ pos $>$, open=open+1
  }
  pos=pos+1
}
return encoding

Generating Random Trees Without Cross Products

Tree queries only!

- query graph: $G = (V, E)$, $|V| = n$, G must be a tree.
- level: root has level 0, children thereof 1, etc.
- $\mathcal{T}_G$: join trees for G

[8]

Partitioning $\mathcal{T}_G$

$\mathcal{T}_G^{v(k)} \subseteq \mathcal{T}_G$: subset of join trees where the leaf node (i.e. relation) v occurs at level k.

Observations:

- $n = 1$: $|\mathcal{T}_G| = |\mathcal{T}_G^{v(0)}| = 1$
- $n > 1$: $|\mathcal{T}_G^{v(0)}| = 0$ (top is a join and no relation)
- The maximum level that can occur in any join tree is $n - 1$. Hence: $|\mathcal{T}_G^{v(k)}| = 0$ if $k \geq n$.
- $\mathcal{T}_G = \cup_{k=0}^{n} \mathcal{T}_G^{v(k)}$
- $\mathcal{T}_G^{v(i)} \cap \mathcal{T}_G^{v(j)} = \emptyset$ for $i \neq j$
- Thus: $|\mathcal{T}_G| = \sum_{k=0}^{n} |\mathcal{T}_G^{v(k)}|$

The Specification

- The algorithm will generate an unordered tree with n leaf nodes.
- If we wish to have a random ordered tree, we have to pick one of the 2^{n-1} possibilities to order the $(n-1)$ joins within the tree.

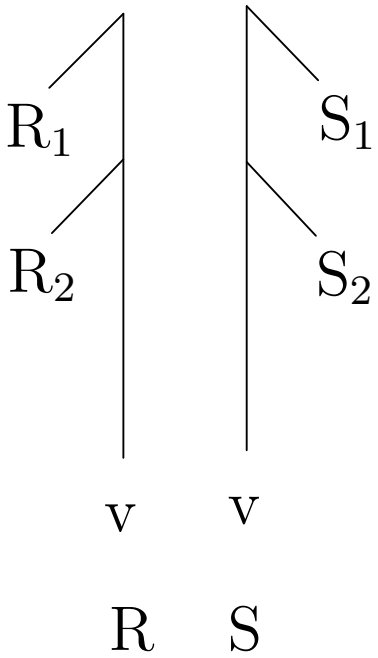
The Procedure

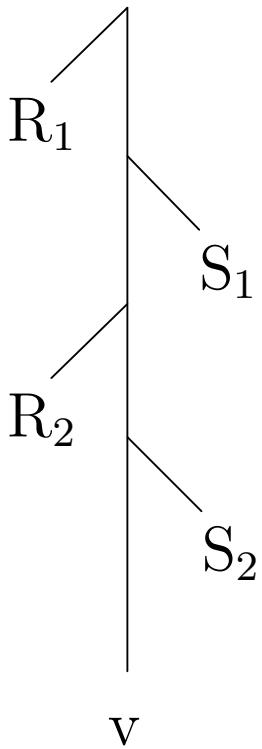
1. List merges (notation, specification, counting, unranking)
2. Join tree construction: leaf-insertion and tree-merging
3. Standard Decomposition Graph (SDG): describes all valid join trees
4. Counting
5. Unranking algorithm

List Merge

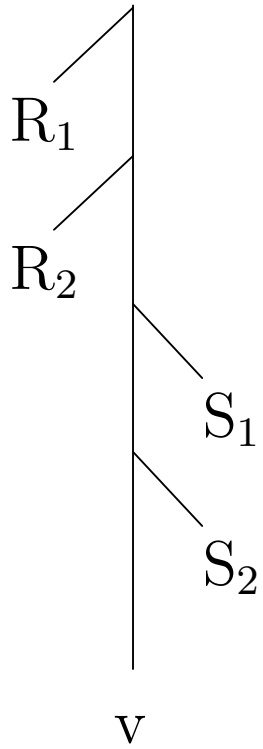
- Lists: Prolog-Notation: $< a|t >$
- Property P on elements
- A list l' is the *projection* of a list L on P, if L' contains all elements of L satisfying the property P.
  Thereby, the order is retained.
- A list L is a *merge* of two disjoint lists L_1 and L_2, if L contains all elements from L_1 and L_2 and both are projections of L.

Example

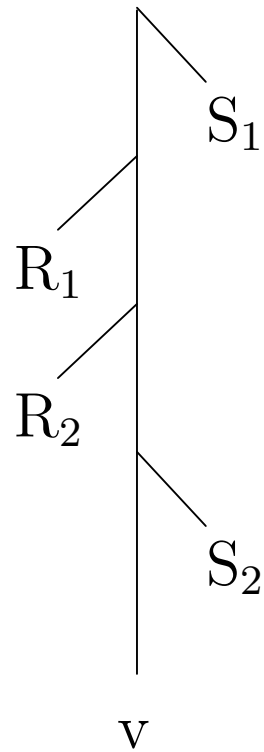




(R, S, [1, 1, 0])



(R, S, [2, 0, 0])



(R, S, [0, 2, 0])

List Merge: Specification

A merge of a list L_1 with a list L_2 whose respective lengths are l_1 and l_2 can be described by an array $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$ of non-negative integers whose sum is equal to l_1, i.e. $\sum_{i=0}^{l_2} \alpha_i = |l_1|$.

- We obtain the merged list L by first taking α_0 elements from L_1.
- Then, an element from L_2 follows. Then follow α_1 elements from L_1 and the next element of L_2 and so on.
- Finally follow the last α_{l_2} elements of L_1.

List Merge: Counting

Non-negative integer decomposition:

- What is the number of decompositions of a non-negative integer n into k non-negative integers α_i with $\sum_{i=1}^{k} \alpha_k = n$.

Answer: $\binom{n+k-1}{k-1}$

List Merge: Counting (2)

Since we have to decompose l_1 into $l_2 + 1$ non-negative integers, the number of possible merges is $M(l_1, l_2) = \binom{l_1+l_2}{l_2}$.
The observation $M(l_1, l_2) = M(l_1 - 1, l_2) + M(l_1, l_2 - 1)$ allows us to construct an array of size $n * n$ in $O(n^2)$ that materializes the values for M.
This array will allow us to rank list merges in $O(l_1 + l_2)$.

List Merge: Unranking: General Idea

The idea for establishing a bijection between $[1, M(l_1, l_2)]$ and the possible αs is a general one and used for all subsequent algorithms of this section. Assume we want to rank the elements of some set S and $S = \cup_{i=0}^{n} S_i$ is partitioned into disjoint S_i.

1. If we want to rank $x \in S_k$, we first find the *local rank* of $x \in S_k$.
2. The rank of x is then $\sum_{i=0}^{k-1} |S_i| + \texttt{local-rank}(x, S_k)$.
3. To unrank some number $r \in [1, N]$, we first find k such that $k = \min_j r \leq \sum_{i=0}^{j} |S_i|$.
4. We proceed by unranking with the new local rank $r' = r - \sum_{i=0}^{k-1} |S_i|$ within S_k.

List Merge: Unranking

We partition the set of all possible merges into subsets.

- Each subset is determined by α_0.
  For example, the set of possible merges of two lists L_1 and L_2 with length $l_1 = l_2 = 4$ is partitioned into subsets with $\alpha_0 = j$ for $0 \leq j \leq 4$.
- In each partition, we have $M(l_1 - j, l_2 - 1)$ elements.
- To unrank a number $r \in [1, M(l_1, l_2)]$ we first determine the partition by computing $k = \min_j r \leq \sum_{i=0}^{j} M(j, l_2 - 1)$.
  Then, $\alpha_0 = l_1 - k$.
- With the new rank $r' = r - \sum_{i=0}^{k} M(j, l_2 - 1)$, we start iterating all over.

Example

k	α_0	$(k, l_2 - 1)$	$M(k, l_2 - 1)$	rank intervals
0	4	$(0, 3)$	1	$[1, 1]$
1	3	$(1, 3)$	4	$[2, 5]$
2	2	$(2, 3)$	10	$[6, 15]$
3	1	$(3, 3)$	20	$[16, 35]$
4	0	$(4, 3)$	35	$[36, 70]$

Decomposition

UnrankDecomposition(r, l_1, l_2)
Input: a rank r, two list sizes l_1 and l_2
Output: encoding of the inner leafes of a tree
alpha $= <>$, $k = 0$
while $l_1 > 0 \wedge l_2 > 0$ {
  $m = M(k, l_2 - 1)$
  if $r \leq m$ {
    alpha=alphae$\circ < l_1 - k >$
    $l_1 = k, k = 0, l_2 = l_2 - 1$
  } **else** {
    $r = r - m, k = k + 1$
  }
}
return alpha$\circ < l_1 > \circ \bigcirc_{|alpha|+1 \leq i < l_2} < 0 >$

Anchored List Representation of Join Trees

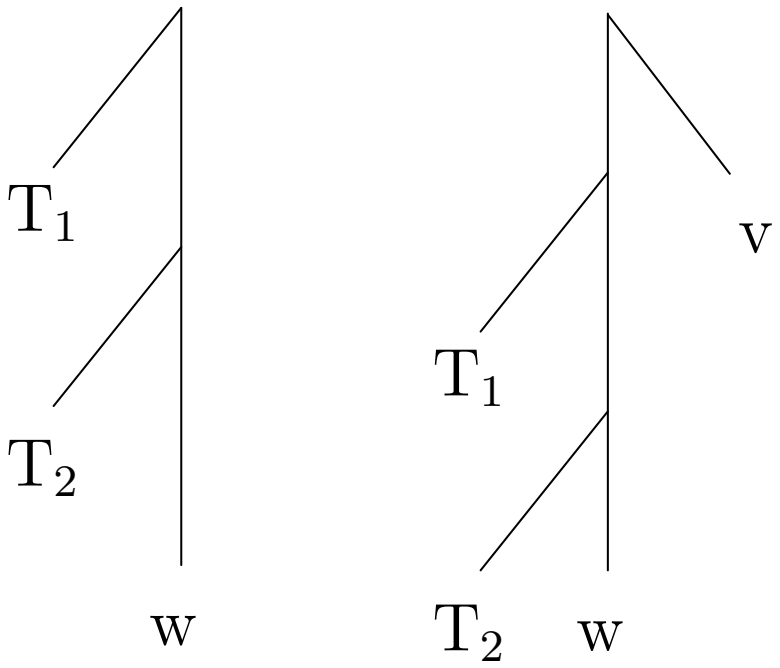
Definition Let T be a join tree and v be a leaf of T. The *anchored list representation* L of T is constructed as follows:

- If T consists of the single leaf node v, then $L =<>$.
- If $T = (T_l \bowtie T_2)$ and without loss of generality v occurs in T_2, then $L =< T_1 | L_2 >$ where L_2 is the anchored list representation of T_2.

We then write $T = (L, v)$.

Observation If $T = (L, v) \in \mathcal{T}_G$ then $T \in \mathcal{T}_G^{v(k)} \prec\succ |L| = k$

Leaf-Insertion: Example



T (T, 1) (T, 2) (T, 3)

Leaf-Insertion

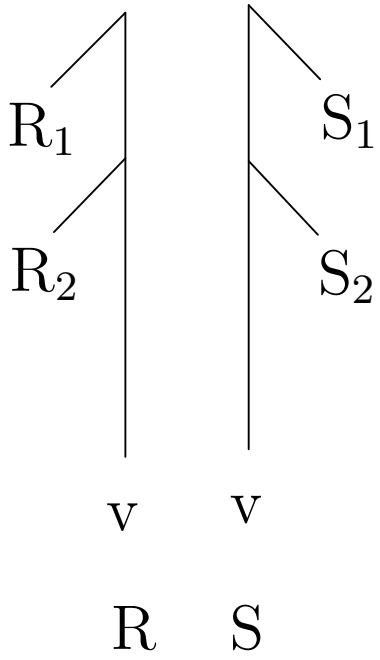
Definition Let $G = (V, E)$ be a query graph, T a join tree of G. $v \in V$ be such that $G' = G|_{V \setminus \{v\}}$ is connected, $(v, w) \in E$, $1 \leq k < n$, and

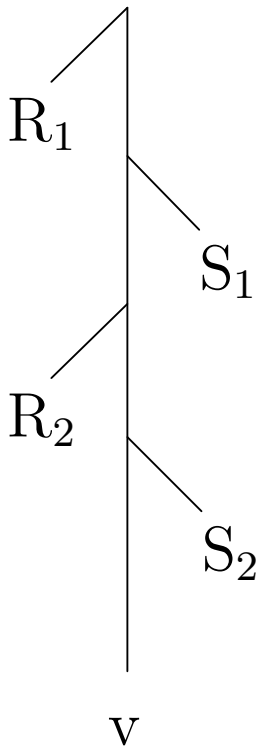
$$\begin{aligned} T &= (< T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n >, w) \\ T' &= (< T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n >, w). \end{aligned}$$

Then we call (T', k) an *insertion pair* on v and say that T is *decomposed into* (or *constructed from*) the pair (T', k) on v.

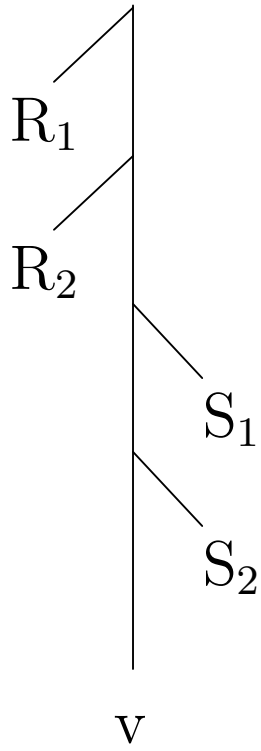
Observation: Leaf-insertion defines a bijective mapping between $\mathcal{T}_G^{v(k)}$ and insertion pairs (T', k) on v, where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

Tree-Merging: Example

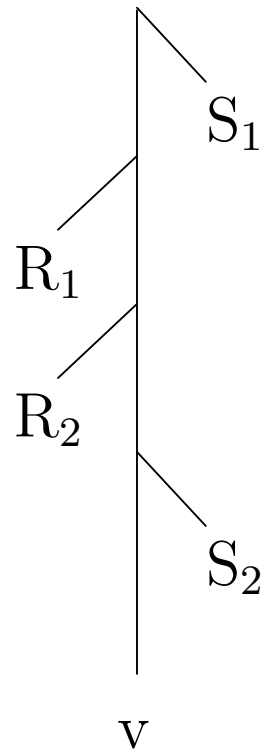




(R, S, [1, 1, 0])



(R, S, [2, 0, 0])



(R, S, [0, 2, 0])

Tree-Merging

Two trees $R = (L_R, w)$ and $S = (L_S, w)$ on a common leaf w are merged by merging their anchored list representations.

Definition. Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{w\}$. For $i = 1, 2$:

- Define the property P_i to be "every leaf of the subtree is in V_i",
- Let L_i be the projection of L on P_i.
- $T_i = (L_i, w)$.

Let α be the integer decomposition such that L is the result of merging L_1 and L_2 on α. Then, we call (T_1, T_2, α) a *merge triplet*. We say that T is *decomposed into* (*constructed from*) (T_1, T_2, α) on V_1 and V_2.

Observation

Tree-Merging defines a bijective mapping between $\mathcal{T}_G^{w(k)}$ and merge triplets (T_1, T_2, α), where $T_1 \in \mathcal{T}_{G_1}^{w(i)}$, $T_2 \in \mathcal{T}_{G_2}^{w(k-i)}$, and α specifies a merge of two lists of sizes i and $k-i$. Further, the number of these merges (i.e. the number of possibilities for α) is $\binom{i+(k-i)}{k-i} = \binom{k}{i}$.

Standard Decomposition Graph (SDG)

A *standard decomposition graph* of a query graph describes the possible constructions of join trees.
It is not unique (for $n > 1$) but anyone can be used to construct all possible unordered join trees.
For each of our two operations it has one kind of inner nodes:

- A unary node labeled $+_v$ stands for leaf-insertion of v.
- A binary node labeled $*_w$ stands for tree-merging its subtrees whose only common leaf is w.

Constructing a Standard Decomposition Graph

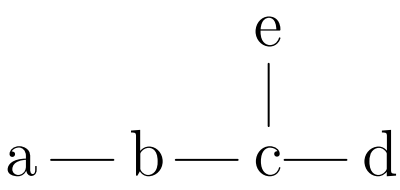
The standard decomposition graph of a query graph $G = (V, E)$ is constructed in three steps:

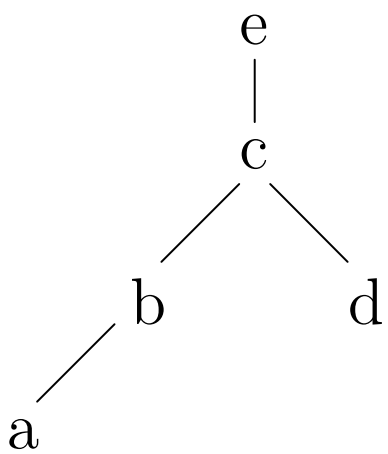
1. pick an arbitrary node $r \in V$ as its root node
2. transform G into a tree G' by directing all edges away from r;
3. call `QG2SDG`(G', r)

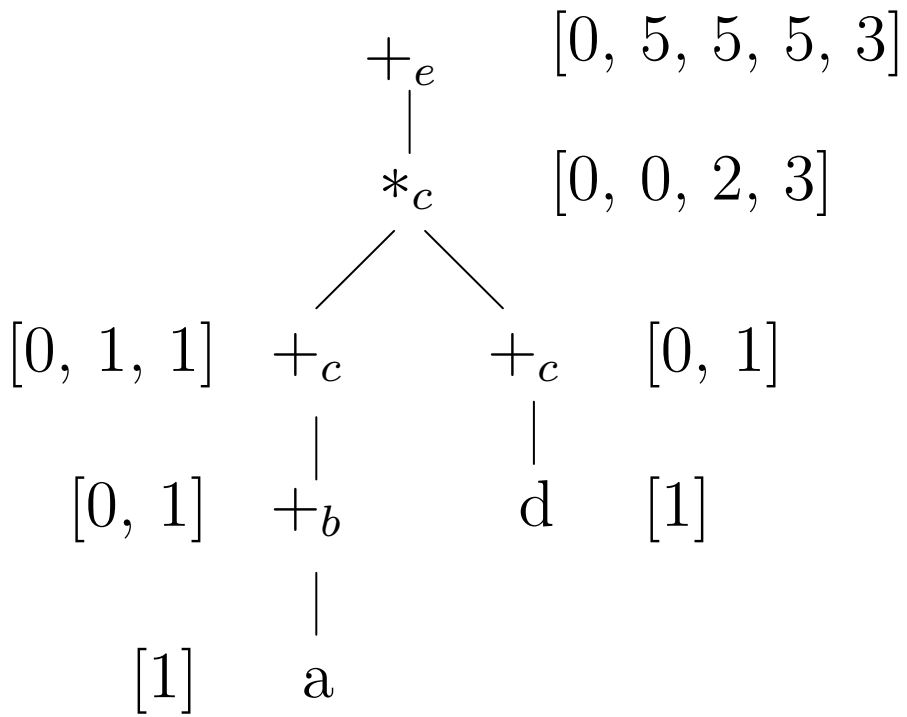
Constructing a Standard Decomposition Graph (2)

QG2SDG(G', r)
Input: a query tree $G' = (V, E)$ and its root r
Output: a standard query decomposition tree of G'
Let $\{w_1, \ldots, w_n\}$ be the children of v
switch n {
  case 0: label v with "v"
  case 1:
    label v as "$+_v$"
    QG2SDG(G', w_1)
  otherwise:
    label v as "$*_v$"
    create new nodes l, r with label $+_v$
    $E = E \setminus \{(v, w_i) | 1 \leq i \leq n\}$
    $E = E \cup \{(v, l), (v, r), (l, w_1)\} \cup \{(r, w_i) | 2 \leq i \leq n\}$
    QG2SDG(G', l), QG2SDG(G', r)
}
return G'

Constructing a Standard Decomposition Graph (3)







Counting

For efficient access to the number of join trees in some partition $\mathcal{T}_G^{v(k)}$ in the unranking algorithm, we materialize these numbers.
This is done in the `count` array.
The semantics of a `count` array $[c_0, c_1, \ldots, c_n]$ of a node u with label $\circ_v$ ($\circ \in \{+, *\}$) of the SDG is that

- u can construct c_i different trees in which leaf v is at level i.

Then, the total number of trees for a query can be computed by summing up all the c_i in the `count` array of the root node of the decomposition tree.

Counting (2)

To compute the `count` and an additional `summand` adornment of a node labeled $+_v$, we use the following lemma:

Lemma. Let $G = (V, E)$ be a query graph with n nodes, $v \in V$ such that $G' = G|_{V \setminus v}$ is connected, $(v, w) \in E$, and $1 \leq k < n$. Then

$$|\mathcal{T}_G^{v(k)}| = \sum_{i \geq k-1} |\mathcal{T}_{G'}^{w(i)}|$$

Counting (3)

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of the former Lemma directly correspond to subsets $\mathcal{T}_{G}^{v(k),i}$ $(k-1 \leq i \leq n-2)$ defined such that $T \in \mathcal{T}_{G}^{v(k),i}$ if

1. $T \in \mathcal{T}_{G}^{v(k)}$,
2. the insertion pair on v of T is (T', k), and
3. $T' \in \mathcal{T}_{G'}^{w(i)}$.

Further, $|\mathcal{T}_{G}^{v(k),i}| = |\mathcal{T}_{G'}^{w(i)}|$. For efficiency, we materialize the summands in an array of arrays `summands`.

Counting (4)

To compute the `count` and `summand` adornment of a node labeled $*_v$, we use the following lemma.

Lemma. Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$. Then

$$|\mathcal{T}_G^{v(k)}| = \sum_i \binom{k}{i} |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$$

Counting (5)

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of the previous Lemma directly correspond to subsets $\mathcal{T}_G^{v(k),i}$ $(0 \leq i \leq k)$ defined such that $T \in \mathcal{T}_G^{v(k),i}$ if

1. $T \in \mathcal{T}_G^{v(k)}$,
2. the merge triplet on V_1 and V_2 of T is (T_1, T_2, α), and
3. $T_1 \in \mathcal{T}_{G_1}^{v(i)}$.

Further, $|\mathcal{T}_G^{v(k),i}| = \binom{k}{i} \, |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$.

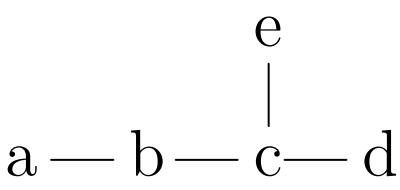
Counting (6)

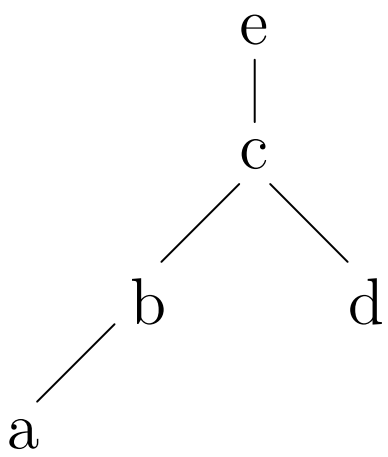
Observation: Assume a node v whose `count` array is $[c_1, \ldots, c_m]$ and whose `summands` is $s = [s^0, \ldots, s^n]$ with $s_i = [s_0^i, \ldots, s_m^i]$, then

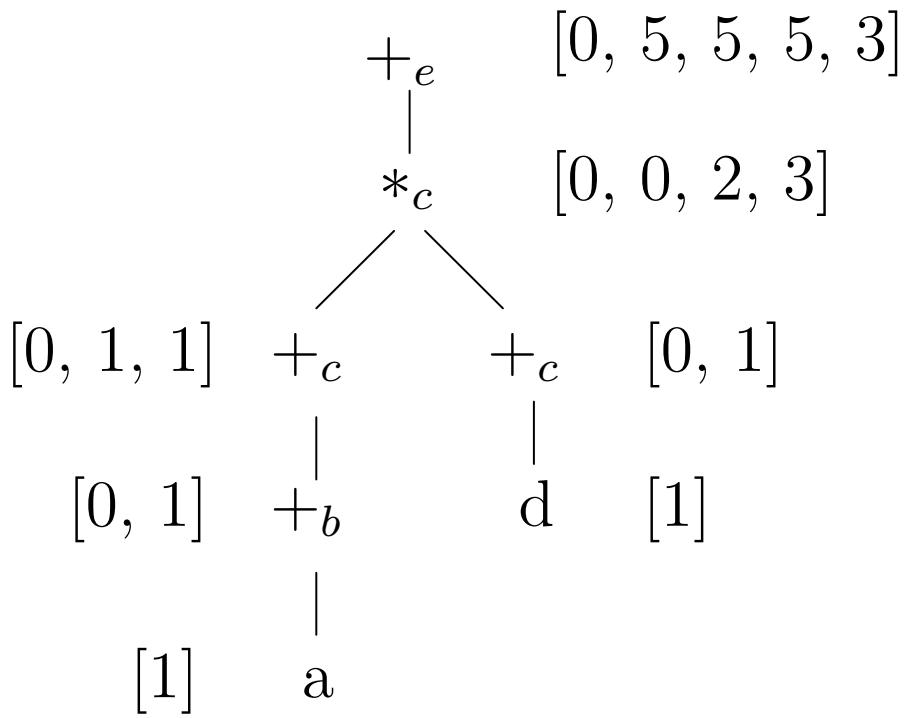
$$c_i = \sum_{j=0}^{m} s_j^i$$

holds.
The following algorithm has worst-case complexity $O(n^3)$.
Looking at the `count` array of the root node of the following SDG, we see that the total number of join trees for our example query graph is 18.

SDG example







Annotating the SDG

Adorn(v)
Input: a node v of the SDG
Output: v and nodes below are adorned by count and summands
Let $\{w_1, \ldots, w_n\}$ be the children of v
switch (n) {
case 0: count(v) = [1] // no summands for v
case 1:
Adorn(w_1)
assume count(w_1) = $[c_0^1, \ldots, c_{m_1}^1]$;
count(v) = $[0, c_1, \ldots, c_{m_1+1}]$ where $c_k = \sum_{i=k-1}^{m_1} c_i^1$
summands(v) = $[s^0, \ldots, s^{m_1+1}]$ where $s^k = [s_0^k, \ldots, s_{m_1+1}^k]$ and

$$s_i^k = \begin{cases} c_i^1 & \text{if } 0 < k \text{ and } k-1 \leq i \\ 0 & \text{else} \end{cases}$$

Annotating the SDG (2)

case 2:

Adorn(w_1)

Adorn(w_2)

assume count(w_1) $= [c_0^1, \ldots, c_{m_1}^1]$

assume count(w_2) $= [c_0^2, \ldots, c_{m_2}^2]$

count(v) $= [c_0, \ldots, c_{m_1+m_2}]$ where

$c_k = \sum_{i=0}^{m_1} \binom{k}{i} c_i^1 c_{k-i}^2$; // $c_i^2 = 0$ for $i \notin \{0, \ldots, m_2\}$

summands(v) $= [s^0, \ldots, s^{m_1+m_2}]$ where $s^k = [s_0^k, \ldots, s_{m_1}^k]$ and

$$s_i^k = \begin{cases} \binom{k}{i} c_i^1 c_{k-i}^2 & \text{if } 0 \leq k - i \leq m_2 \\ 0 & \text{else} \end{cases}$$

}

Unranking: top-level procedure

The algorithm `UnrankLocalTreeNoCross` called by `UnrankTreeNoCross` adorns the standard decomposition graph with `insert-at` and `merge-using` annotations. These can then be used to extract the join tree.

UnrankTreeNoCross(r,v)
Input: a rank r and the root v of the SDG
Output: adorned SDG
let count$(v) = [x_0, \ldots, x_m]$
$k = \min_j r \leq \sum_{i=0}^{j} x_i$
$r' = r - \sum_{i=0}^{k-1} x_i$
UnrankLocalTreeNoCross(v, r', k)

Unranking: Example

The following table shows the intervals associated with the partitions $\mathcal{T}_G^{e(k)}$ for our standard decomposition graph:

Partition	*Interval*
$\mathcal{T}_G^{e(1)}$	$[1, 5]$
$\mathcal{T}_G^{e(2)}$	$[6, 10]$
$\mathcal{T}_G^{e(3)}$	$[11, 15]$
$\mathcal{T}_G^{e(4)}$	$[16, 18]$

Unranking: the last utility function

The unranking procedure makes use of unranking decompositions and unranking triples. For the latter and a given X, Y, Z, we need to assign each member in

$$\{(x, y, z) | 1 \leq x \leq X, 1 \leq y \leq Y, 1 \leq z \leq Z\}$$

a unique number in $[1, XYZ]$ and base an unranking algorithm on this assignment. We call the function `UnrankTriplet`(r, X, Y, Z). r is a rank and X, Y, and Z are the upper bounds for the numbers in the triplets.

Unranking Without Cross Products

UnrankingTreeNoCrossLocal(v, r, k)
Input: an SDG node v, a rank r, a number k identifying a partition
Output: adornments of the SDG as a side-effect
Let $\{w_1, \ldots, w_n\}$ be the children of v
switch n {
  case 0:
    // no additional adornment for v

Unranking Without Cross Products (2)

case 1:

let $\text{count}(v) = [c_0, \ldots, c_n]$

let $\text{summands}(v) = [s^0, \ldots, s^n]$

$k_1 = \min_j r \leq \sum_{i=0}^{j} s_i^k$

$r_1 = r - \sum_{i=0}^{k_1-1} s_i^k$

$\text{insert-at}(v) = k$

$\text{UnrankingTreeNoCrossLocal}(w_1, r_1, k_1)$

Unranking Without Cross Products (3)

case 2:

let count$(v) = [c_0, \ldots, c_n]$

let summands$(v) = [s^0, \ldots, s^n]$

let count$(w_1) = [c_0^1, \ldots, c_{n_1}^1]$

let count$(w_2) = [c_0^2, \ldots, c_{n_2}^2]$

$k_1 = \min_j r \leq \sum_{i=0}^{j} s_i^k$

$q = r - \sum_{i=0}^{k_1-1} s_i^k$

$k_2 = k - k_1$

$(r_1, r_2, a) = \text{UnrankTriplet}(q, c_{k_1}^1, c_{k_2}^2, \binom{k}{i})$

$\alpha = \text{UnrankDecomposition}(a)$

merge-using$(v) = \alpha$

UnrankingTreeNoCrossLocal(w_1, r_1, k_1)

UnrankingTreeNoCrossLocal(w_2, r_2, k_2)

}

Quick Pick

- problem: build (pseudo-)random join trees fast
- unranking without cross products is quite involved
- idea: randomly select an edge in the query graph
- extend join tree by selected edge

No longer uniformly distributed, but very fast

Quick Pick (2)

QuickPick(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a bushy join tree
$E' = E$;
Trees $= \{R_1, \ldots, R_n\}$;
while |Trees| > 1 {
  choose a random $e \in E'$
  $E' = E' \setminus \{e\}$
  if e connects two relations in different subtrees $T_1, T_2 \in$ Trees
    Trees $=$ Trees$\setminus\{T_1, T_2\} \cup$CreateJoinTree$(T_1, T_2)$
}
return $T \in$Trees

- repeated multiple times to find a good tree

Metaheuristics

- provide a very general optimization strategy
- applicable for many different problems
- work well even for very large problems
- but are often considered a "brute-force" method

We consider the metaheuristics formulated for the join ordering problem.

Iterative Improvement

- Start with random join tree
- Select rule that improves join tree
- Stop when no further improvement possible

Iterative Improvement (2)

IterativeImprovementBase(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
do {
  JoinTree = random tree
  JoinTree = IterativeImprovement(JoinTree)
  if cost(JoinTree) $<$ cost(BestTree) {
    BestTree = JoinTree
  }
} **while** (time limit not exceeded)
return BestTree

Iterative Improvement (3)

IterativeImprovement(JoinTree)
Input: a join tree
Output: improved join tree
do {
  JoinTree' = randomly apply a transformation from the rule set to the JoinT
  if (cost(JoinTree') < cost(JoinTree)) {
    JoinTree = JoinTree'
  }
} **while** local minimum not reached
return JoinTree

- problem: local minimum detection

Simulated Annealing

- II: stuck in local minimum
- SA: allow moves that result in more expensive join trees
- lower the threshold for worsening

Simulated Annealing (2)

SimulatedAnnealing(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
BestTreeSoFar = random tree
Tree = BestTreeSoFar

Simulated Annealing (3)

```
do {
  do {
    Tree' = apply random transformation to Tree
    if (cost(Tree') < cost(Tree)) {
      Tree = Tree'
    } else {
      with probability e^{-(cost(Tree')-cost(Tree))/temperature}
        Tree = Tree'
    }
    if (cost(Tree) < cost(BestTreeSoFar)) {
      BestTreeSoFar = Tree'
    }
  } while equilibrium not reached
  reduce temperature
} while not frozen
return BestTreeSoFar
```

Simulated Annealing (4)

Advantages:

- can escape from local minimum
- produces better results than II

Problems:

- parameter tuning
- initial temperature
- when and how to decrease the temperature

Tabu Search

- Select cheapest reachable neighbor (even if it is more expensive)
- Maintain tabu set to avoid running into circles

Tabu Search (2)

TabuSearch(Query Graph)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
Tree = random join tree
BestTreeSoFar = Tree
TabuSet = $\emptyset$
do {
  Neighbors = all trees generated by applying a transformation to Tree
  Tree = cheapest in Neighbors $\setminus$ TabuSet
  if cost(Tree) $<$ cost(BestTreeSoFar)
    BestTreeSoFar = Tree
  if (|TabuSet| $>$ limit) remove oldest tree from TabuSet
  TabuSet = TabuSet$\cup${Tree}
}
return BestTreeSoFar

Genetic Algorithms

- Join trees seen as population
- Successor generations generated by crossover and mutation
- Only the fittest survive

Problem: Encoding

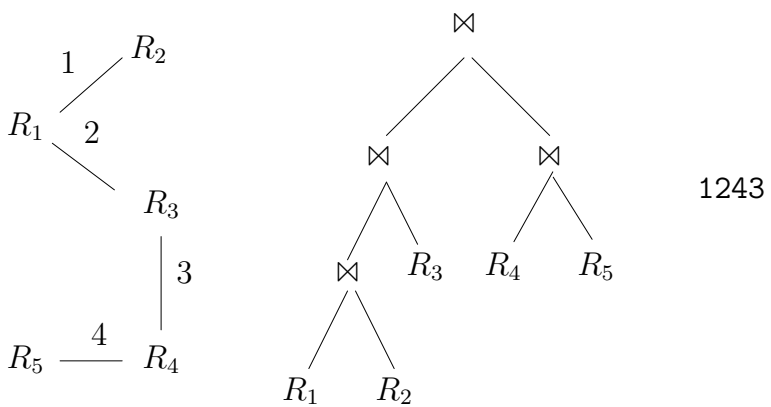
- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

Encoding

We distinguish *ordered list* and *ordinal number* encodings.
Both encodings are used for left-deep and bushy trees.
In all cases we assume that the relations $R_1, \ldots, R_n$ are to be joined and use the index i to denote R_i.

Ordered List Encoding

1. left-deep trees
   A left-deep join tree is encoded by a permutation of $1, \ldots, n$. For instance, $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1423".
2. bushy trees
   A bushy join-tree without cartesian products is encoded as an ordered list of the edges in the join graph. Therefore, we number the edges in the join graph. Then, the join tree is encoded in a bottom-up, left-to-right manner.



Ordinal Number Encoding

In both cases, we start with the list $L =< R_1, \ldots, R_n >$.

- left-deep trees
  Within L we find the index of first relation to be joined. If this relation be R_i then the first character in the chromosome string is i. We eliminate R_i from L. For every subsequent relation joined, we again determine its index in L, remove it from L and append the index to the chromosome string.
  For instance, starting with $< R_1, R_2, R_3, R_4 >$, the left-deep join tree $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1311".

Ordinal Number Encoding (2)

- bushy trees
  We encode a bushy join tree in a bottom-up, left-to-right manner. Let $R_i \bowtie R_j$ be the first join in the join tree under this ordering. Then we look up their positions in L and add them to the encoding. Then we eliminate R_i and R_j from L and push $R_{i,j}$ to the front of it. We then proceed for the other joins by again selecting the next join which now can be between relations and or subtrees. We determine their position within L, add these positions to the encoding, remove them from L, and insert a composite relation into L such that the new composite relation directly follows those already present.
  For instance, starting with the list $< R_1, R_2, R_3, R_4 >$, the bushy join tree $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$ is encoded as "12 23 12".

Crossover

1. Subsequence exchange
2. Subset exchange

Crossover: Subsequence exchange

The subsequence exchange for the ordered list encoding:

- Assume two individuals with chromosomes $u_1 v_1 w_1$ and $u_2 v_2 w_2$.
- From these we generate $u_1 v_1' w_1$ and $u_2 v_2' w_2$ where v_i' is a permutation of the relations in v_i such that the order of their appearence is the same as in $u_{3-i} v_{3-i} w_{3-i}$.

Subsequence exchange for ordinal number encoding:

- We require that the v_i are of equal length ($|v_1| = |v_2|$) and occur at the same offset ($|u_1| = |u_2|$).
- We then simply swap the v_i.
- That is, we generate $u_1 v_2 w_1$ and $u_2 v_1 w_2$.

Crossover: Subset exchange

The subset exchange is defined only for the ordered list encoding. Within the two chromosomes, we find two subsequences of equal length comprising the same set of relations. These sequences are then simply exchanged.

Mutation

A mutation randomly alters a character in the encoding.
If duplicates may not occur— as in the ordered list encoding—swapping two characters is a perfect mutation.

Selection

- The probability of survival is determined by its rank in the population.
- We calculate the costs of the join trees encoded for each member in the population.
- Then, we sort the population according to their associated costs and assign probabilities to each individual such that the best solution in the population has the highest probability to survive and so on.
- After probabilities have been assigned, we randomly select members of the population taking into account these probabilities.
- That is, the higher the probability of a member the higher its chance to survive.

The Algorithm

1. Create a random population of a given size (say 128).
2. Apply crossover and mutation with a given rate.
   For example such that 65% of all members of a population participate in crossover, and 5% of all members of a population are subject to random mutation.
3. Apply selection until we again have a population of the given size.
4. Stop after no improvement within the population was seen for a fixed number of iterations (say 30).

Combinations

- metaheuristics are often not used in isolation
- they can be used to improve existing heurstics
- or heuristics can be used to speed up metaheuristics

Two Phase Optimization

1. For a number of randomly generated initial trees, Iterative Improvement is used to find a local minima.
2. Then Simulated Annealing is started to find a better plan in the neighborhood of the local minima.
   The initial temperature of Simulated Annealing can be lower as is its original variants.

AB Algorithm

1. If the query graph is cyclic, a spanning tree is selected.
2. Assign join methods randomly
3. Apply IKKBZ
4. Apply iterative improvement

Toured Simulated Annealing

The basic idea is that simulated annealing is called n times with different initial join trees, if n is the number of relations to be joined.

- Each join sequence in the set S produced by `GreedyJoinOrdering-3` is used to start an independent run of simulated annealing.

As a result, the starting temperature can be descreased to 0.1 times the cost of the initial plan.

GOO-II

Append an iterative improvement step to GOO

Iterative Dynamic Programming

- Two variants: IDP-1, IDP-2 [9]
- Here: Only IDP-1 base version

Idea:

- create join trees with up to k relations
- replace cheapest one by a compound relation
- start all over again

Iterative Dynamic Programming (2)

IDP-1($\{R_1, \ldots, R_n\}$, k)
Input: a set of relations to be joined, maximum block size k
Output: a join tree
for each $1 \leq i \leq n$ {
  BestTree($\{R_i\}$) $= R_i$;
}
ToDo $= \{R_1, \ldots, R_n\}$

Iterative Dynamic Programming (3)

while $|\text{ToDo}| > 1$ {
  $k = \min(k, |\text{T}oDo|)$
  for each $2 \leq i < k$ **ascending**
    for all $S \subseteq \text{T}oDo$, $|S| = i$ **do**
      for all $O \subset S$ **do**
        $\text{BestTree}(S) = \text{CreateJoinTree}(\text{BestTree}(S \setminus O), \text{BestTree}(O))$;
  find $V \subset \text{T}oDo$, $|V| = k$ **with**
    $\text{cost}(\text{BestTree}(V)) = \min\{\text{cost}(\text{BestTree}(W)) \mid W \subset \text{T}oDo, |W| = k\}$
  generate new symbol T
  $\text{BestTree}(\{T\}) = \text{BestTree}(V)$
  $\text{ToDo} = (\text{ToDo} \setminus V) \cup \{T\}$
  for each $O \subset V$ **do** delete($\text{BestTree}(O)$)
}
return $\text{BestTree}(\{R_1, \ldots, R_n\})$

Iterative Dynamic Programming (4)

- compromise between runtime and optimality
- combines greedy heuristics with dynamic programming
- scales well to large problems
- finds the optimal solution for smaller problems
- approach can be used for different DP strategies

Order Preserving Joins

- some query languages operatore on lists instead of sets/bags
- order of tuples matters
- examples: XPath/XQuery
- alternatives: either add sort operators or use order preserving operators

Here, we define order preserving operators, $list \rightarrow list$

- let L be a list
- $L[1]$ is the first entry in L
- $L[2 : |L|]$ are the remaining entries

Order Preserving Selection

We define the order preserving selection σ^L as follows:

$$\sigma_p^L(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < e[1] > \circ \sigma_p^L(e[2 : |e|]) & \text{if } p(e[1]) \\ \sigma_p^L(e[2 : |e|]) & \text{otherwise} \end{cases}$$

- filters like a normal selection
- preserves the relative ordering (guaranteed)

Order Preserving Cross Product

We define the order preserving cross product $\times^L$ as follows:

$$e_1 \times^L e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ (e[1] \hat{\times}^L e_2) \circ (e_1[2 : |e_1|] \times^L e_2) & \text{otherwise} \end{cases}$$

using the tuple/list product defined as:

$$t \hat{\times}^L e := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < t \circ e[1] > \circ (t \hat{\times}^L e[2 : |e|]) & \text{otherwise} \end{cases}$$

- preserves the order of e_1
- order of e_2 is preserved for each e_1 group

Order Preserving Join

The definition of the order preserving join is analogous to the non-order preserving case:

$$e_1 \bowtie_p^L e_2 := \sigma_p^L(e_1 \times^L e_2)$$

- preserves order of e_1, order of e_2 relative to e_1

Equivalences

$$
\begin{array}{rcll}
\sigma^L_{p_1}(\sigma^L_{p_2}(e)) & \equiv & \sigma^L_{p_2}(\sigma^L_{p_1}(e)) & \\
\sigma^L_{p_1}(e_1 \bowtie^L_{p_2} e_2) & \equiv & \sigma^L_{p_1}(e_1) \bowtie^L_{p_2} e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \\
\sigma^L_{p_2}(e_1 \bowtie^L_{p_2} e_2) & \equiv & e_1 \bowtie^L_{p_2} \sigma^L_{p_1}(e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_2) \\
e_1 \bowtie^L_{p_1} (e_2 \bowtie^L_{p_2} e_3) & \equiv & (e_1 \bowtie^L_{p_1} e_2) \bowtie^L_{p_2} e_3) & \text{if } \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

- swap selections
- push selections down
- associativity

Commutativity

Consider the relations $R_1 =< [a:1], [a:2] >$ and $R_2 =< [b:1], [b:2] >$. Then

$$
\begin{aligned}
R_1 \bowtie^L_{true} R_2 &= < [a:1, b:1], [a:1, b:2], [a:2, b:1], [a:2, b:2] > \\
R_2 \bowtie^L_{true} R_1 &= < [a:1, b:1], [a:2, b:1], [a:1, b:2], [a:2, b:2] >
\end{aligned}
$$

- the order preserving join is not commutative

Algorithm

- similar to matrix multiplication
- in addition: selection push down
- DP table is a $n \times n$ array (or rather 4 arrays)
- algorithm fills arrays p, s, c, t:
  - p: applicable predicates
  - s: statistics (cardinality, perhaps more)
  - c: costs
  - t: split position for larger plans
- plan is extracted from the arrays afterwards

Algorithm (2)

OrderPreservingJoins($R = \{R_1, \ldots, R_n\}$,P)
Input: a set of relations to be joined and a set of predicates
Output: fills p, s, c, t
for each $1 \leq i \leq n$ {
  $p[i,i]$ =predicates from P applicable to R_i
  $P = P \setminus p[i,i]$
  $s[i,i]$ =statistics for $\sigma_{p[i,i]}(R_i)$
  $c[i,i]$ =costs for $\sigma_{p[i,i]}(R_i)$
}

Algorithm (3)

for each $2 \leq l \leq n$ **ascending** {
  for each $1 \leq i \leq n - l + 1$ {
    $j = i + l - 1$
    $p[i,j]$=predicates from P applicable to $R_i, \ldots, R_j$
    $P = P \setminus p[i,j]$
    $s[i,j]$=statistics derived from $s[i,j-1]$ and $s[j,j]$ including $p[i,j]$
    $c[i,j]$=∞
    for each $i \leq k < j$ {
      $q = c[i,k] + c[k+1,j]$+costs for $s[i,k]$ and $s[k+1,j]$ and $p[i,j]$
      if $q < c[i,j]$ {
        c[i,j]=q
        t[i,j]=k
      }
    }
  }
}

Algorithm (4)

ExtractPlan($R = \{R_1, \ldots, R_n\}, t, p$)
Input: a set of relations, arrays t and p
Output: a bushy join tree
return ExtractPlanRec($R, t, p, 1, n$)

ExtractPlanRec($R = \{R_1, \ldots, R_n\}, t, p, i, j$)
if $i < j$ {
  T_1 =ExtractPlanRec($R, t, p, i, t[i, j]$)
  T_2 =ExtractPlanRec($R, t, p, t[i, j] + 1, j$)
  return $T_1 \bowtie^L_{p[i,j]} T_2$
} **else** {
  return $\sigma_{p[i,j]} R_i$
}