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

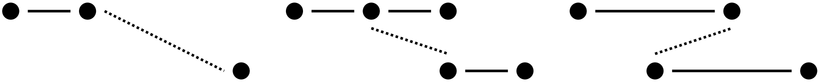
good algorithm for cliques, but adapts badly:

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

Observation

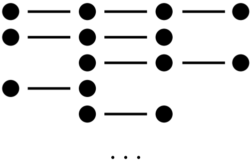
DPsize and DPsub generate many pairs that are pruned anyway (connectedness, overlap).

Typical pruned pairs (chain with 4 relations):



not connected not disjoint invalid subproblems

last example ⇒ 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 $n-1$

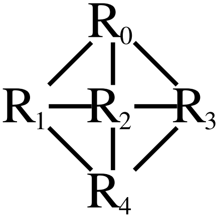
Preliminaries, given query graph $G = (V, E)$:

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

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, $(S \cup S')$, $(X \cup N)$);
$\}$

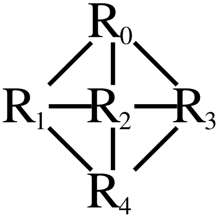


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

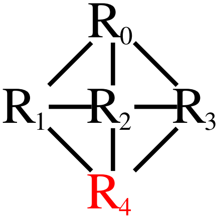


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

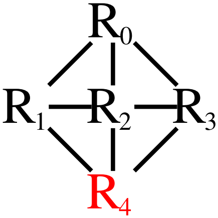


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

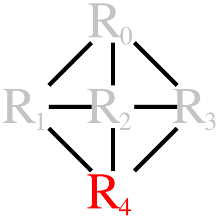


Enumerating Connected Subgraphs (2)

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

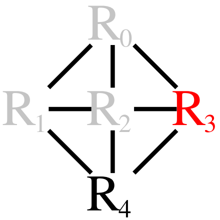
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

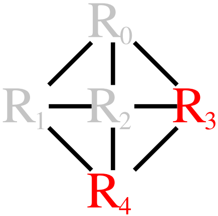
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

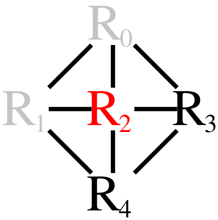
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



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, $(S \cup S')$, $(X \cup N)$);
}

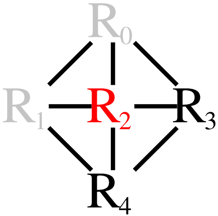


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

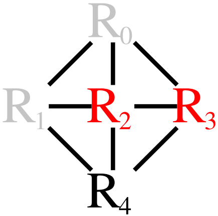


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

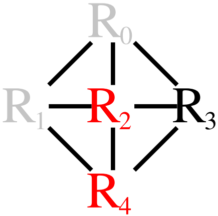


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

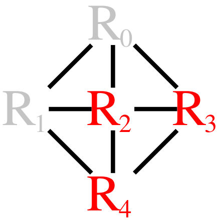


Enumerating Connected Subgraphs (2)

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, $(S \cup S')$, $(X \cup N)$);
}

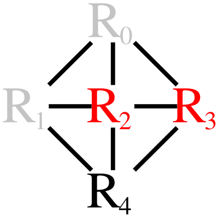


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \dots, 0]$ **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, $(S \cup S')$, $(X \cup N)$);
}

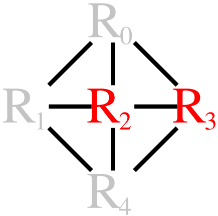


Enumerating Connected Subgraphs (2)

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

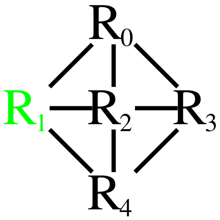
The neighborhood is prohibited during recursion, preventing duplicates

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 Complementary Subgraphs

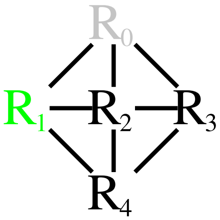
EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
  emit $\{v_i\}$;
  EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}



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, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

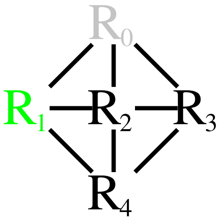
Prohibit all nodes that will be start nodes later on and the primary subgraph



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, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

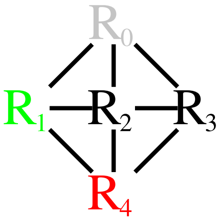
Find all neighboring nodes that are not prohibited



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, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

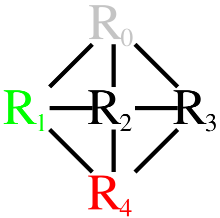
Consider each of the nodes



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, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

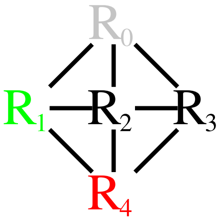
Choose the node as complementary subgraph and emit it



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, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

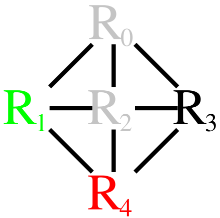
Recursively increase the subgraph re-using EnumerateCsgRec



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

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 $R_1R_2R_3R_4$. If we know that $R_1R_3R_2$ is cheaper than $R_1R_2R_3$, we do not have to consider $R_1R_2R_3R_4$.

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 $\forall R_i \in R$ {
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 $\forall R_i \in R$ {
    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 of ties occur
- worst-case runtime of no ties occur is an open problem

Often fast, can be stopped anytime, but can perform poor.

Transformative Approaches

Main idea: [6]

- 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 $\forall\ T \in$ Trees {
        if $T \notin$ ToDo $\cup$ Done
            ToDo = ToDo $\cup \{T\}$
    }
}
return $\arg\min_{T \in \text{Done}} C(T)$

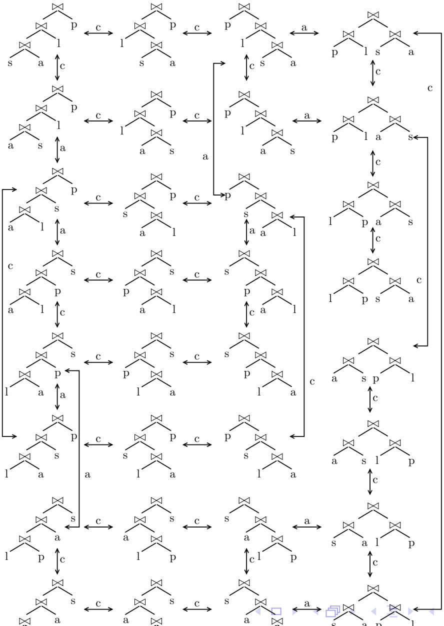
Basic Algorithm (2)

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 $\forall\ S \in$ 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 $\forall$ transformation $\mathcal{T}$ and class member of child classes {
  for $\forall T'$ 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 $R_1 \bowtie_0 R_2 \leadsto R_2 \bowtie_1 R_1$
Here, the R_i are restricted to classes with exactly one relation. T_1 is disabled for $\bowtie_1$.

T_2 Right Join Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \leadsto (C_1 \bowtie_2 C_3) \bowtie_3 C_2$
Disable T_2 for $\bowtie_3$.