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

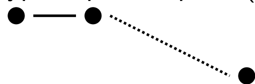
good algorithm for cliques, but adapts badly:

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

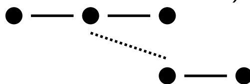
Observation

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

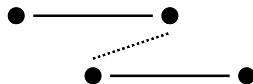
Typical pruned pairs (chain with 4 relations):



not connected

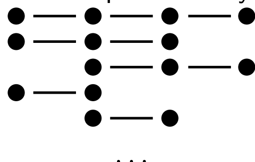


not disjoint



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), $S = S_1 \cup S_2$ {
  $p_1 = B[S_1]$, $p_2 = B[S_2]$
  $P = \text{CreateJoinTree}(p_1, p_2)$;
  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 $n-1$

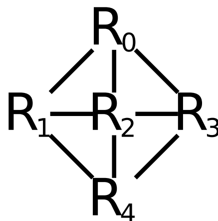
Preliminaries, given query graph $G = (V, E)$:

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

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

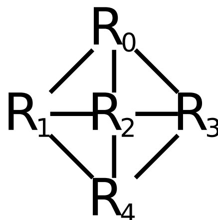


Enumerating Connected Subgraphs (2)

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

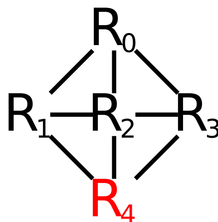


Enumerating Connected Subgraphs (2)

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

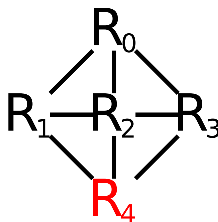


Enumerating Connected Subgraphs (2)

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

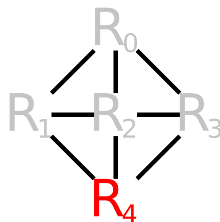


Enumerating Connected Subgraphs (2)

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

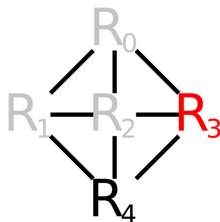
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

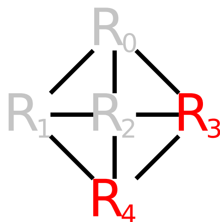
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

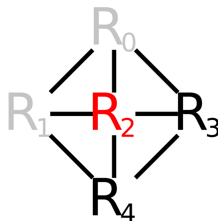
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



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

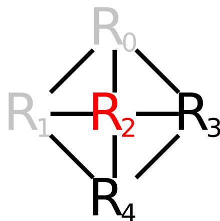


Enumerating Connected Subgraphs (2)

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

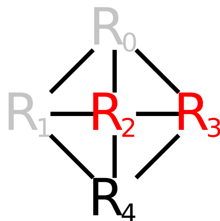


Enumerating Connected Subgraphs (2)

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

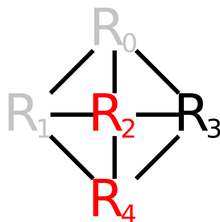


Enumerating Connected Subgraphs (2)

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

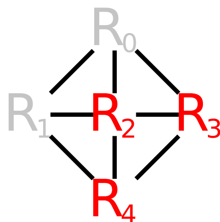


Enumerating Connected Subgraphs (2)

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

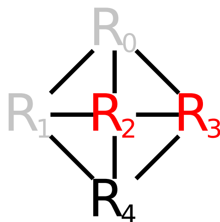


Enumerating Connected Subgraphs (2)

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

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

Then, add all combinations to the subgraph and increase recursively

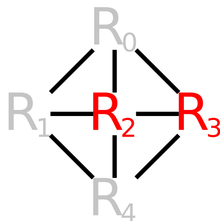


Enumerating Connected Subgraphs (2)

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

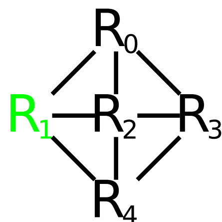
The neighborhood is prohibited during recursion, preventing duplicates

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

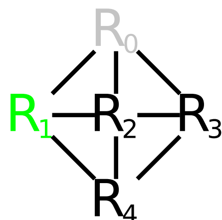
EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}



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

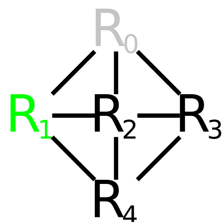
Prohibit all nodes that will be start nodes later on and the primary subgraph



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

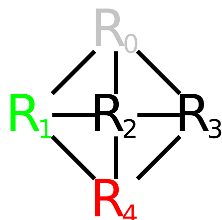
Find all neighboring nodes that are not prohibited



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

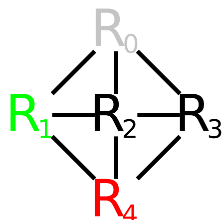
Consider each of the nodes



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

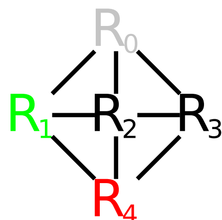
Choose the node as complementary subgraph and emit it



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

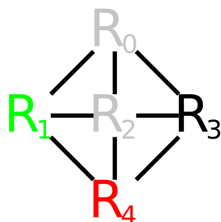
Recursively increase the subgraph re-using EnumerateCsgRec



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

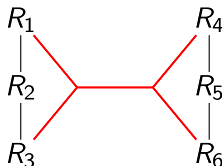
Remarks

- 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

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 $u \cap v = \emptyset$.

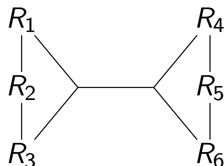
Nodes in V are totally ordered via an (arbitrary) relation $\prec$.

- 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



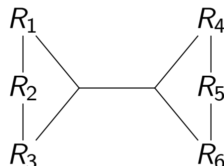
Problem:

- 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

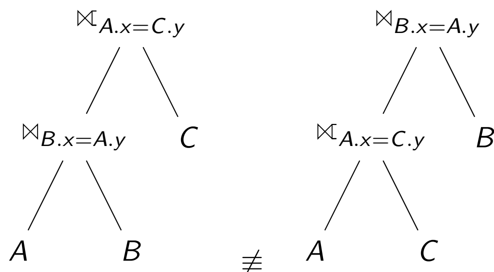
Requires checks for connectedness

- 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

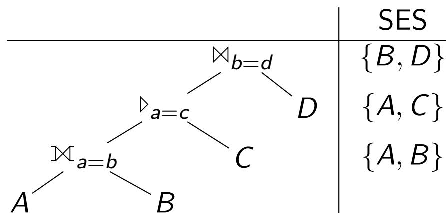
Gives a big compatibility matrix

	⋈	⟕	⟗	▷	⋉	⋈Γ	...
⋈	+	+	-	+	+	+	...
⟕	-	+	-	-	-	-	...
⟗	-	+	+	-	-	-	...
▷	-	-	-	-	-	-	...
⋉	-	-	-	-	-	-	...
⋈Γ	-	-	-	-	-	-	...
...							

Non-Inner Joins - TESs

Extract reordering constraints from operator tree in two steps:

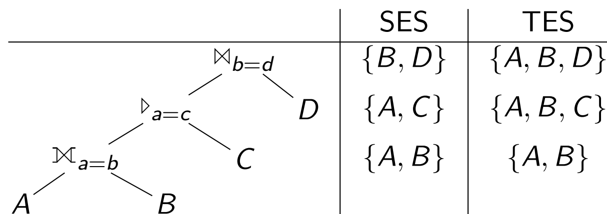
1. build the *syntactic eligibility set* (SES) for each operator
   - set of relations that has to be in the input



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: