Dynamic Programming-Based Plan Generation

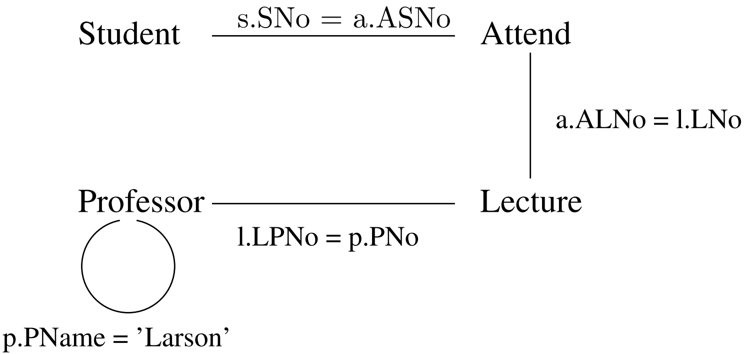
1. Join Ordering Problems
2. Simple Graphs (inner joins only)
3. Hypergraphs (also non-inner joins)
4. Grouping

(Simple) Query Graph

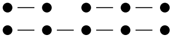
- A *query graph* is an undirected graph with nodes $R_1, \ldots, R_n$.
- For every join predicate $p_{i,j}$ referencing attributes from R_i and R_j there is an edge between R_i and R_j.
  This edge is labeled by the join predicate p_{ij}.

For simple predicates applicable to a single relation, we add a self-edge.
NOTE: We do not consider simple selection predicates, but assume that they have to be pushed down before.

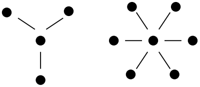
Example Query Graph



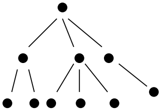
Shapes of Query Graphs



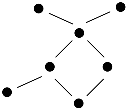
chain queries



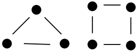
star queries



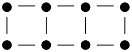
tree query



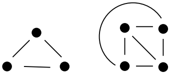
cyclic query



cycle queries



grid query



clique queries

Join Trees (Plans)

are binary trees

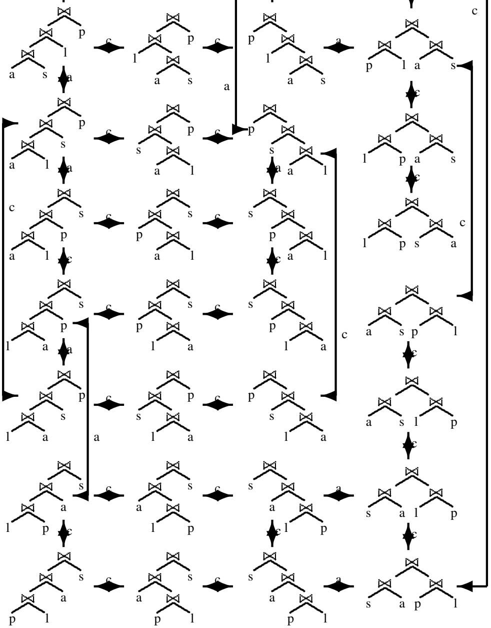
- with relation names attached to leaf nodes and
- join operators as inner nodes.

Some algorithms will produce ordered binary trees, others unordered binary trees.
Distinguish whether cross products are allowed or not.

Join Tree Shapes

- left-deep join trees
- right-deep join trees
- zig-zag trees
- bushy trees

Left-deep, right-deep, and zig-zag trees can be summarized under the notion of *linear trees*



Observations

- Costs differ vastly for different join trees.

Classification of Join Ordering Problems

Query Graph Classes $\times$ *Possible Join Tree Classes* $\times$ *Cost Function Classes*

- Query Graph Classes: *chain*, *star*, *tree*, and *cyclic*
- Join trees: left-deep, zig-zag, or bushy trees: w/o cross products
- Cost functions: w/o ASI property

In total, we have $4 * 3 * 2 * 2 = 48$ different join ordering problems.

Search Space Size (No Cross Product)

	Join Trees Without Cross Products			
	Chain Query		Star Query	
	Left-Deep	Bushy	Left-Deep	Zig-Zag/Bushy
n	2^{n-1}	$2^{n-1}\mathcal{C}(n-1)$	$2*(n-1)!$	$2^{n-1}(n-1)!$
1	1	1	1	1
2	2	2	2	2
3	4	8	4	8
4	8	40	12	48
5	16	224	48	384
6	32	1344	240	3840
7	64	8448	1440	46080
8	128	54912	10080	645120
9	256	366080	80640	10321920
10	512	2489344	725760	185794560

Search Space Size (With Cross Products)

	With Cross Products/Clique	
	Left-Deep	Bushy
n	$n!$	$n!\mathcal{C}(n-1)$
1	1	1
2	2	2
3	6	12
4	24	120
5	120	1680
6	720	30240
7	5040	665280
8	40320	17297280
9	362880	518918400
10	3628800	17643225600

Complexity

Query Graph	Join Tree	$\times$	Cost Function	Complexity
general	left-deep	no	ASI	NP-hard
tree/star/chain	left-deep	no	one join method (ASI)	P
star	left-deep	no	two join methods (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

Dynamic Programming

- Optimality Principle
- Avoid duplicate work

Optimality Principle

Consider the two join 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 tree is cheaper than the second join tree. Hence, we could avoid generating the second alternative and still won't miss the optimal join tree.

Optimality Principle

Optimality Principle 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.*

Remark: Optimality principle does not hold in the presence of physical properties.

Dynamic Programming

- Generate optimal join trees bottom up
- Start from optimal join trees of size one
- Build larger join trees for sizes $n > 1$ by (re-) using those of smaller sizes
- We use subroutine `CreateJoinTree` that joins two (sub-) trees

Optimal Bushy Trees without Cross Products

Given: Connected join graph

Problem: Generate optimal bushy trees without cross products

Csg-Cmp-Pairs (ccp)

Let S_1 and S_2 be subsets of the nodes (relations) of the query graph. We say (S_1, S_2) is a *csg-cmp-pair*, if and only if

1. S_1 induces a connected subgraph of the query graph,
2. S_2 induces a connected subgraph of the query graph,
3. S_1 and S_2 are disjoint, and
4. there exists at least one edge connectinG A NODE in S_1 to a node in S_2.

If (S_1, S_2) is a csg-cmp-pair, then (S_2, S_1) is a valid csg-cmp-pair.

Csg-Cmp-Pairs and Join Trees

Let (S_1, S_2) be a csg-cmp-pair and T_i be a join tree for S_i. Then we can construct two valid join tree:

$$T_1 \bowtie T_2 \text{ and } T_2 \bowtie T_1$$

Hence, the number of csg-cmp-pairs coincides with the search space DP explores. In fact, the number of csg-cmp-pairs is a lower bound for the complexity of DP.

If `CreateJoinTree` considers commutativity of joins, the number of calls to it is precisely expressed by the count of non-symmetric csg-cmp-pairs. In other implementations `CreateJoinTree` might be called for all csg-cmp-pairs and, thus, may not consider commutativity.

The Number of Csg-Cmp-Pairs

Let us denote the number of non-symmetric csg-cmp-pairs by $\#\texttt{ccp}$. Then

$$\begin{aligned}
\#\texttt{ccp}^{\text{chain}}(n) &= \frac{1}{6}(n^3 - n) \\
\#\texttt{ccp}^{\text{cycle}}(n) &= (n^3 - 2n^2 + n)/2 \\
\#\texttt{ccp}^{\text{star}}(n) &= (n-1)2^{n-2} \\
\#\texttt{ccp}^{\text{clique}}(n) &= (3^n - 2^{n+1} + 1)/2
\end{aligned}$$

These numbers have to be multiplied by two if we want to count all csg-cmp-pairs.

DPsize

```
for all R_i ∈ R BestPlan({R_i}) = R_i;
for all 1 < s ≤ n ascending // size of plan
for all 1 ≤ s_1 < s // size of left subplan
  s_2 = s − s_1; // size of right subplan
  for all p_1 = BestPlan(S_1 ⊂ R : |S_1| = s_1)
  for all p_2 = BestPlan(S_2 ⊂ R : |S_2| = s_2)
    ++InnerCounter;
    if (∅ ≠ S_1 ∩ S_2) continue;
    if not (S_1 connected to S_2) continue;
    ++CsgCmpPairCounter;
    CurrPlan = CreateJoinTree(p_1, p_2);
    if (cost(BestPlan(S_1 ∪ S_2)) > cost(CurrPlan))
      BestPlan(S_1 ∪ S_2) = CurrPlan;
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n−1}});
```

Analysis: DPsize

$$I^{\text{chain}}_{\text{DPsize}}(n) = \begin{cases} 1/48(5n^4 + 6n^3 - 14n^2 - 12n) & n \text{ even} \\ 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & n \text{ odd} \end{cases}$$

$$I^{\text{cycle}}_{\text{DPsize}}(n) = \begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases}$$

$$I^{\text{star}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases}$$

$$\text{with } q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$$

$$I^{\text{clique}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 & n \text{ odd} \end{cases}$$

DPsub

for all $R_i \in R$ `BestPlan(`$\{R_i\}$`) =` R_i;
for $1 \leq i < 2^n - 1$ **ascending**
  $S = \{R_j \in R | (\lfloor i/2^j \rfloor \, mod \, 2) = 1\}$
  if not `(connected` S`)` **continue**;
  for all $S_1 \subset S$, $S_1 \neq \emptyset$ **do**
    `++InnerCounter;` $S_2 = S \setminus S_1$;
    if ($S_2 = \emptyset$) **continue**;
    if not `(connected` S_1`)` **continue**;
    if not `(connected` S_2`)` **continue**;
    if not (S_1 `connected to` S_2) **continue**;
    `++CsgCmpPairCounter;`
    p_1 `= BestPlan(`S_1`),` p_2 `= BestPlan(`S_2`);`
    `CurrPlan = CreateJoinTree(`p_1`,` p_2`);`
    if `(cost(BestPlan(`S`)) > cost(CurrPlan))`
      `BestPlan(`S`) = CurrPlan;`
`OnoLohmanCounter = CsgCmpPairCounter / 2;`
return `BestPlan(`$\{R_0, \ldots, R_{n-1}\}$`);`

Analysis: DPsub

$$
\begin{aligned}
I_{\text{DPsub}}^{\text{chain}}(n) &= 2^{n+2} - n^2 - 3n - 4 \\
I_{\text{DPsub}}^{\text{cycle}}(n) &= n2^n + 2^n - 2n^2 - 2 \\
I_{\text{DPsub}}^{\text{star}}(n) &= 2 * 3^{n-1} - 2^n \\
I_{\text{DPsub}}^{\text{clique}}(n) &= 3^n - 2^{n+1} + 1
\end{aligned}
$$

Sample Numbers

	Chain			Cycle		
n	`#ccp`	`DPsub`	`DPsize`	`#ccp`	`DPsub`	`DPsize`
5	20	84	73	40	140	120
10	165	3962	1135	405	11062	2225
15	560	130798	5628	1470	523836	11760
20	1330	4193840	17545	3610	22019294	37900
	Star			Clique		
n	`#ccp`	`DPsub`	`DPsize`	`#ccp`	`DPsub`	`DPsize`
5	32	130	110	90	180	280
10	2304	38342	57888	28501	57002	306991
15	114688	9533170	57305929	7141686	14283372	307173877
20	4980736	2323474358	59892991338	1742343625	3484687250	309338182241

Algorithm DPccp

```
for all (R_i ∈ ℛ) BestPlan({R_i}) = R_i;
for all csg-cmp-pairs (S_1, S_2), S = S_1 ∪ S_2
  ++InnerCounter;
  ++OnoLohmanCounter;
  p_1 = BestPlan(S_1);
  p_2 = BestPlan(S_2);
  CurrPlan = CreateJoinTree(p_1, p_2);
  if (cost(BestPlan(S)) > cost(CurrPlan))
    BestPlan(S) = CurrPlan;
  CurrPlan = CreateJoinTree(p_2, p_1);
  if (cost(BestPlan(S)) > cost(CurrPlan))
    BestPlan(S) = CurrPlan;
CsgCmpPairCounter = 2 * OnoLohmanCounter;
return BestPlan({R_0, ..., R_{n-1}});
```

Notation

Let $G = (V, E)$ be an undirected graph.
For a node $v \in V$ define the *neighborhood* $\boldsymbol{N}(v)$ of v as

$$\boldsymbol{N}(v) := \{v' | (v, v') \in E\}$$

For a subset $S \subseteq V$ of V we define the *neighborhood* of S as

$$\boldsymbol{N}(S) := \cup_{v \in S} \boldsymbol{N}(v) \setminus S$$

The neighborhood of a set of nodes thus consists of all nodes reachable by a single edge.
Note that for all $S, S' \subset V$ we have
$\boldsymbol{N}(S \cup S') = (\boldsymbol{N}(S) \cup \boldsymbol{N}(S')) \setminus (S \cup S')$. This allows for an efficient bottom-up calculation of neighborhoods.
$\mathcal{B}_i = \{v_j | j \leq i\}$

Algorithm EnumerateCsg

`EnumerateCsg`
Input: a connected query graph $G = (V, E)$
Precondition: nodes in V are numbered according to a breadth-first search
Output: emits all subsets of V inducing a connected subgraph of G

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

Subroutine EnumerateCsgRec

EnumerateCsgRec(G, S, X)
$N = \boldsymbol{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
$\}$

Algorithm EnumerateCmp

`EnumerateCmp`

Input: a connected query graph $G = (V, E)$, a connected subset S_1

Precondition: nodes in V are numbered according to a breadth-first search

Output: emits all complements S_2 for S_1 such that (S_1, S_2) is a csg-cmp-pair

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

where $\min(S_1) := \min(\{i | v_i \in S_1\})$.

A DP-Based Plan Generator for Hypergraphs

outline

1. motivation
2. basic algorithm
3. preliminaries
4. reorderability
5. conflict detection
6. enumeration

Necessity of Hypergraphs

1. non-inner joins: conflict detectors introduce hypergraphs
2. general join predicates: $R.a + S.b = S.c + T.d$

DPHYP

$\triangleright$ **Input:** a set of relations $R = \{R_0, \ldots, R_{n-1}\}$
a set of operators O with associated predicates
a query hypergraph H

$\triangleright$ **Output:** an optimal bushy operator tree

for all $R_i \in R$
    $DPTable[R_i] \leftarrow R_i$ $\triangleright$ initial access paths
for all csg-cmp-pairs (S_1, S_2) of H
    for all $\circ_p \in O$
        if APPLICABLE$(S_1, S_2, \circ_p)$
            BUILDPLANS$(S_1, S_2, \circ_p)$
            if $\circ_p$ is commutative
                BUILDPLANS$(S_2, S_1, \circ_p)$
return $DPTable[R]$

Preliminaries (strict predicates)

Definition

A predicate is null rejecting for a set of attributes A if it evaluates to `false` *or* `unknown` *on every tuple in which all attributes in A are* `null`*.*

Synonyms for null rejecting are used: *null intolerant*, *strong*, and *strict*.

Preliminaries (initial operator tree)

We assume that we have an initial operator tree, e.g., by a canonical translation of a SQL query.

Preliminaries (accessors)

For a set of attributes A, $\mathtt{REL}(A)$ denotes the set of tables to which these attributes belong. We abbreviate $\mathtt{REL}(\mathcal{F}(e))$ by $\mathcal{F}_{\mathsf{T}}(e)$. Let $\circ$ be an operator in the initial operator tree. We denote by left($\circ$) (right($\circ$)) its left (right) child. $\mathtt{STO}(\circ)$ denotes the operators contained in the operator subtree rooted at $\circ$. $\mathtt{REL}(\circ)$ denotes the set of tables contained in the subtree rooted at $\circ$.

Preliminaries (SES)

Then, for each operator we define its *syntactic eligibility sets* as its set of tables referenced by its predicate.
If $p \equiv R.a + S.b = S.c + T.d$, then $\mathcal{F}(p) = \{R.a, S.b, S.c, T.d\}$ and $\texttt{SES}(\circ_p) = \{R, S, T\}$.

Preliminaries (degenerate predicates)

Definition

Let p be a predicate associated with a binary operator $\circ$ and $\mathcal{F}_{\mathsf{T}}(p)$ the tables referenced by p. Then, p is called *degenerate* if $\mathtt{REL}(\text{left}(\circ)) \cap \mathcal{F}_{\mathsf{T}}(p) = \emptyset \ \vee \ \mathtt{REL}(\text{right}(\circ)) \cap \mathcal{F}_{\mathsf{T}}(p) = \emptyset$ holds.

Here, we exclude degenerate predicates.

Preliminaries (hypergraph)

Definition

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 $u \cap v = \emptyset$.

We call any non-empty subset of V a *hypernode*.

Preliminaries (Neighborhood)

$$\min(S) = \{s | s \in S, \forall s' \in S\ s \neq s' \implies s \prec s'\}$$

Let S be a current set, which we want to expand by adding further relations. Consider a hyperedge (u, v) with $u \subseteq S$. Then, we will add $\min(v)$ to the neighborhood of S. We thus define

$$\overline{\min}(S) = S \setminus \min(S)$$

Note: we have to make sure that the missing elements of v, i.e. $v \setminus \min(v)$, are also contained in any set emitted.

Preliminaries (Neighborhood)

We define the set of non-subsumed hyperedges as the minimal subset $E{\downarrow}$ of E such that for all $(u, v) \in E$ there exists a hyperedge $(u', v') \in E{\downarrow}$ with $u' \subseteq u$ and $v' \subseteq v$.

$$E{\downarrow}'(S, X) = \{v | (u, v) \in E, u \subseteq S, v \cap S = \emptyset, v \cap X = \emptyset\}$$

Define $E{\downarrow}(S, X)$ to be the minimal set of hypernodes such that for all $v \in E{\downarrow}'(S, X)$ there exists a hypernode v' in $E{\downarrow}(S, X)$ such that $v' \subseteq v$.
Neighborhood:

$$N(S, X) = \bigcup_{v \in E{\downarrow}(S,X)} \min(v) \quad (1)$$

where X is the set of forbidden nodes.

Preliminaries (csg-cmp-pair)

Definition

Let $H = (V, E)$ be a hypergraph and S_1, S_2 two non-empty subsets of V with $S_1 \cap S_2 = \emptyset$. Then, the pair (S_1, S_2) is called a *csg-cmp-pair* if the following conditions hold:

1. S_1 and S_2 induce a connected subgraph of H, and
2. there exists a hyperedge $(u, v) \in E$ such that $u \subseteq S_1$ and $v \subseteq S_2$.

Reorderability (properties)

- commutativity (comm)
- associativity (assoc)
- l/r-asscom

Reorderability (comm)

$\circ$	
$\times$	+
$\bowtie$	+
$\ltimes$	-
$\triangleright$	-
⟕	-
⟗	+
$\bowtie\!\ulcorner$	-

Reorderability (assoc)

assoc:

$$(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 \equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3) \tag{2}$$

Reorderability (assoc)

$\circ^a$	$\circ^b$						
	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$\leftouterjoin$	$\fullouterjoin$	$\bowtie^{\Gamma}$
$\times$	+	+	+	+	+	-	+
$\bowtie$	+	+	+	+	+	-	+
$\ltimes$	-	-	-	-	-	-	-
$\triangleright$	-	-	-	-	-	-	-
$\leftouterjoin$	-	-	-	-	+[1]	-	-
$\fullouterjoin$	-	-	-	-	+[1]	+[2]	-
$\bowtie^{\Gamma}$	-	-	-	-	-	-	-

(1) if p_{23} rejects nulls on $\mathcal{A}(e_2)$ (Eqv. 2)

(2) if p_{12} and p_{23} reject nulls on $\mathcal{A}(e_2)$ (Eqv. 2)

Reorderability (l/r-asscom)

Consider the following truth about the semijoin:

$$(e_1 \ltimes_{12} e_2) \ltimes_{13} e_3 \equiv (e_1 \ltimes_{13} e_3) \ltimes_{12} e_2.$$

This is not expressible with associativity nor commutativity (in fact the semijoin is neither).

Reorderability (l/r-asscom)

We define the *left asscom property* (l-asscom for short) as follows:

$$(e_1 \circ^a_{12} e_2) \circ^b_{13} e_3 \equiv (e_1 \circ^b_{13} e_3) \circ^a_{12} e_2. \tag{3}$$

We denote by l-asscom($\circ^a, \circ^b$) the fact that Eqv. 3 holds for $\circ^a$ and $\circ^b$.

Analogously, we can define a *right asscom property* (r-asscom):

$$e_1 \circ^a_{13} (e_2 \circ^b_{23} e_3) \equiv e_2 \circ^b_{23} (e_1 \circ^a_{13} e_3). \tag{4}$$

First, note that l-asscom and r-asscom are symmetric properties, i.e.,

$$\begin{aligned} \text{l-asscom}(\circ^a, \circ^b) &\leftrightarrow \text{l-asscom}(\circ^b, \circ^a), \\ \text{r-asscom}(\circ^a, \circ^b) &\leftrightarrow \text{r-asscom}(\circ^b, \circ^a). \end{aligned}$$

Reorderability (l/r-asscom)

$\circ$	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	⟕	⟗	$\bowtie^{\Gamma}$
$\times$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\bowtie$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\ltimes$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
$\triangleright$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
⟕	+/-	+/-	+/-	+/-	+/-	+[1] /-	+/-
⟗	-/-	-/-	-/-	-/-	+[2] /-	+[3] /+[4]	-/-
$\bowtie^{\Gamma}$	+/-	+/-	+/-	+/-	+/-	-/-	+/-

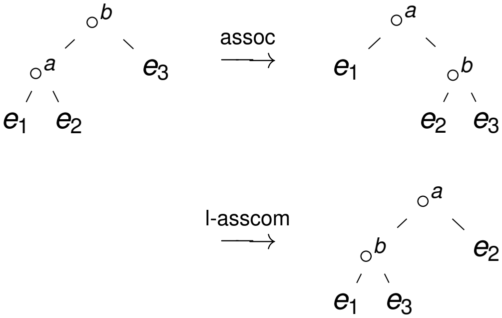
1. if p_{12} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 3)
2. if p_{13} rejects nulls on $\mathcal{A}(e_3)$ (Eqv. 3)
3. if p_{12} and p_{13} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 3)
4. if p_{13} and p_{23} reject nulls on $\mathcal{A}(e_3)$ (Eqv. 4)

Conflict Detector CD-A: $\mathtt{SES}$

$$
\begin{aligned}
\mathtt{SES}(R) &= \{R\} \\
\mathtt{SES}(T) &= \{T\} \\
\mathtt{SES}(\circ_p) &= \bigcup_{R \in \mathcal{F}_\mathsf{T}(p)} \mathtt{SES}(R) \cap \mathtt{REL}(\circ_p) \\
\mathtt{SES}(\Join^{\Gamma}_{p;a_1:e_1,\ldots,a_n:e_n}) &= \bigcup_{R \in \mathcal{F}_\mathsf{T}(p) \cup \mathcal{F}_\mathsf{T}(e_i)} \mathtt{SES}(R) \cap \mathtt{REL}(gj)
\end{aligned}
$$

Conflict Detector CD-A: $\mathtt{TES}$: left conflict

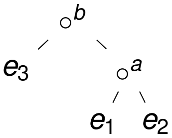
initially: $\mathtt{TES}(\circ_p) := \mathtt{SES}(\circ_p)$



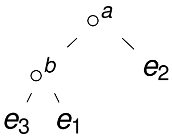
$\neg\text{assoc}(\circ^a, \circ^b)$
$\mathtt{TES}(\circ^b) \cup= \mathtt{REL}(e_1)$

$\neg\text{l-asscom}(\circ^a, \circ^b)$
$\mathtt{TES}(\circ^b) \cup= \mathtt{REL}(e_2)$

Conflict Detector CD-A: `TES`: right conflict

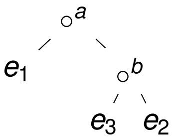


$\xrightarrow{\text{assoc}}$



$\neg\text{assoc}(\circ^b, \circ^a)$
$\mathtt{TES}(\circ^b) \cup= \mathtt{REL}(e_2)$

$\xrightarrow{\text{r-asscom}}$



$\neg\text{r-asscom}(\circ^a, \circ^b)$
$\mathtt{TES}(\circ^b) \cup= \mathtt{REL}(e_1)$

Conflict Detector CD-A: Remarks

- correct
- not complete

Conflict Detector CD-A: applicability test

$$\text{applicable}(\circ, S_1, S_2) := \text{tesl}(\circ) \subseteq S_1 \wedge \text{tesr}(\circ) \subseteq S_2.$$

where

$$\begin{aligned} \text{tesl}(\circ) &:= \mathtt{TES}(\circ) \cap \mathtt{REL}(\text{left}(\circ)) \\ \text{tesr}(\circ) &:= \mathtt{TES}(\circ) \cap \mathtt{REL}(\text{right}(\circ)) \end{aligned}$$

Other Conflict Detectors

- CD-A is correct but not complete.
- CD-B is correct but not complete.
- CD-C is correct and complete

Query Hypergraph Construction

The nodes V are the relations.

For every operator $\circ$, we construct a hyperedge (l, r) such that

$r = \mathrm{TES}(\circ) \cap \mathrm{REL}(\text{right}(\circ)) = \text{R-}\mathrm{TES}(\circ)$ and

$l = \mathrm{TES}(\circ) \setminus r = \text{L-}\mathrm{TES}(\circ)$.

Csg-Cmp-Enumeration: Overview

1. The algorithm constructs ccps by enumerating connected subgraphs from an increasing part of the query graph;
2. both the primary connected subgraphs and its connected complement are created by recursive graph traversals;
3. during traversal, some nodes are *forbidden* to avoid creating duplicates. More precisely, when a function performs a recursive call it forbids all nodes it will investigate itself;
4. connected subgraphs are increased by following edges to neighboring nodes. For this purpose hyperedges are interpreted as $n : 1$ edges, leading from n of one side to one (specific) canonical node of the other side (cmp. Eq. 1).

The last point is like selecting a representative.

Csg-Cmp-Enumeration: Complications

- “starting side” of an edge may contain multiple nodes
- neighborhood calculation more complex, no longer simply bottom-up
- choosing representative: loss of connectivity possible

Last point: use `DpTable` lookup as connectivity test

Csg-Cmp-Enumeration: Routines

1. top-level: `BuEnumCcpHyp`
2. `EnumerateCsgRec`
3. `EmitCsg`
4. `EnumerateCmpRec`

Csg-Cmp-Enumeration: `BuEnumCcpHyp`

`BuEnumCcpHyp`()
for each $v \in V$ // initialize DpTable
    DpTable[$\{v\}$] = plan for v
for each $v \in V$ **descending** according to $\prec$
    `EmitCsg`($\{v\}$) // process singleton sets
    `EnumerateCsgRec`($\{v\}, B_v$) // expand singleton sets
return DpTable[V]

where $B_v = \{w | w \prec v\} \cup \{v\}$.

Csg-Cmp-Enumeration: EnumerateCsgRec

`EnumerateCsgRec`(S_1, X)
for each $N \subseteq \mathcal{N}(S_1, X)$: $N \neq \emptyset$
    if DpTable$[S_1 \cup N] \neq \emptyset$
        `EmitCsg`$(S_1 \cup N)$
for each $N \subseteq \mathcal{N}(S_1, X)$: $N \neq \emptyset$
    `EnumerateCsgRec`$(S_1 \cup N, X \cup \mathcal{N}(S_1, X))$

Csg-Cmp-Enumeration: `EmitCsg`

`EmitCsg`(S_1)
$X = S_1 \cup \mathbf{B}_{min(S_1)}$
$N = \mathcal{N}(S_1, X)$
for each $v \in N$ **descending** according to $\prec$
    $S_2 = \{v\}$
    if $\exists (u, v) \in E : u \subseteq S_1 \wedge v \subseteq S_2$
        `EmitCsgCmp`(S_1, S_2)
    `EnumerateCmpRec`$(S_1, S_2, X \cup B_v(N))$

where $B_v(W) = \{w | w \in W, w \leq v\}$ is defined in `DPccp`.

Csg-Cmp-Enumeration: `EnumerateCmpRec`

`EnumerateCmpRec`(S_1, S_2, X)
for each $N \subseteq \mathcal{N}(S_2, X)$: $N \neq \emptyset$
  if DpTable[$S_2 \cup N$]$\neq \emptyset \wedge$
    $\exists (u, v) \in E : u \subseteq S_1 \wedge v \subseteq S_2 \cup N$
   `EmitCsgCmp`($S_1, S_2 \cup N$)
$X = X \cup \mathcal{N}(S_2, X)$
for each $N \subseteq \mathcal{N}(S_2, X)$: $N \neq \emptyset$
  `EnumerateCmpRec`($S_1, S_2 \cup N, X$)

Csg-Cmp-Enumeration: `EmitCsgCmp`

The procedure `EmitCsgCmp`(S_1,S_2) is called for every S_1 and S_2 such that (S_1, S_2) forms a csg-cmp-pair.
important. Since it is called for either (S_1, S_2) or (S_2, S_1), somewhere the symmetric pairs have to be considered.

Csg-Cmp-Enumeration: Neighborhood Calculation

Let $G = (V, E)$ be a hypergraph not containing any subsumed edges.
For some set S, for which we want to calculate the neighborhood, define the set of reachable hypernodes as

$$W(S, X) := \{w | (u, w) \in E, u \subseteq S, w \cap (S \cup X) = \emptyset\},$$

where X contains the forbidden nodes. Then, any set of nodes N such that for every hypernode in $W(S, X)$ exactly one element is contained in N can serve as the neighborhood.

```
calcNeighborhood(S, X)
N := ∅
if isConnected(S)
    N = simpleNeighborhood(S) \ X
else
    foreach s ∈ S
        N ∪= simpleNeighborhood(s)
F = (S ∪ X ∪ N) // forbidden since in X or already handled
foreach (u, v) ∈ E
    if u ⊆ S
        if v ∩ F = ∅
            N += min(v)
            F ∪= N
    if v ⊆ S
        if u ∩ F = ∅
            N += min(u)
            F ∪= N
```

Including Grouping

1. Pushing grouping down joins may result in much better plans
2. Pulling grouping up may result in much better plans

Equivalences

If outerjoins with defaults other than padding with NULL values are used, grouping can be pushed down any join operator.

Algorithm: Top-Level

DPHYPE

▷ **Input:** a set of relations $R = \{R_0, \ldots, R_{n-1}\}$
a set of operators O with associated predicates
a query hypergraph H

▷ **Output:** an optimal bushy operator tree

for all $R_i \in R$
    $DPTable[R_i] \leftarrow R_i$ ▷ initial access paths
for all csg-cmp-pairs (S_1, S_2) of H
    for all $\circ_p \in O$
        if APPLICABLE$(S_1, S_2, \circ_p)$
            BUILDTREE$(S_1, S_2, \circ_p)$
            if $\circ_p$ is commutative
                BUILDTREE$(S_2, S_1, \circ_p)$
return $DPTable[R]$

Algorithm: BuildTree

```
BUILDTREE(S1, S2, ∘p)
S ← S1 ∪ S2
for each T1 ∈ DPTable[S1]
    for each T2 ∈ DPTable[S2]
        for each T ∈ OPTREES(T1, T2, ∘p)
            if S == R
                INSERTTOPLEVELPLAN(S, T)
            else
                INSERTNONTOPLEVELPLAN(S, T)
```

OpTrees

$\textsc{OpTrees}(T_1, T_2, \circ_p)$

$S_1 \leftarrow \mathcal{T}(T_1)$
$S_2 \leftarrow \mathcal{T}(T_2)$
$S \leftarrow S_1 \cup S_2$
$\mathit{Trees} \leftarrow \emptyset$
$\mathit{NewTree} \leftarrow (T_1 \circ_p T_2)$
if $S == R \wedge \textsc{NeedsGrouping}(G, \mathit{NewTree})$
    $\mathit{NewTree} \leftarrow (\Gamma_G(\mathit{NewTree}))$
$\mathit{Trees}.\mathit{insert}(\mathit{NewTree})$
$\mathit{NewTree} \leftarrow \Gamma_{G_1^+}(T_1) \circ_p T_2$
if $\textsc{Valid}(\mathit{NewTree}) \wedge \textsc{NeedsGrouping}(G_1^+, \mathit{NewTree})$
    if $S == R \wedge \textsc{NeedsGrouping}(G, \mathit{NewTree})$
        $\mathit{NewTree} \leftarrow (\Gamma_G(\mathit{NewTree}))$
    $\mathit{Trees}.\mathit{insert}(\mathit{NewTree})$
$\mathit{NewTree} \leftarrow T_1 \circ_p \Gamma_{G_2^+}(T_2)$

if $\textsc{Valid}(\mathit{NewTree}) \wedge \textsc{NeedsGrouping}(G_2^+, \mathit{NewTree})$

  if $S == R \wedge \textsc{NeedsGrouping}(G, \mathit{NewTree})$

    $\mathit{NewTree} \leftarrow (\Gamma_G(\mathit{NewTree}))$

  $\mathit{Trees.insert}(\mathit{NewTree})$

$\mathit{NewTree} \leftarrow \Gamma_{G_1^+}(T_1) \circ_p \Gamma_{G_2^+}(T_2)$

if $\textsc{Valid}(\mathit{NewTree})$

 $\wedge\ \textsc{NeedsGrouping}(G_1^+, \mathit{NewTree})$

 $\wedge\ \textsc{NeedsGrouping}(G_2^+, \mathit{NewTree}))$

  if $S == R \wedge \textsc{NeedsGrouping}(G, \mathit{NewTree})$

    $\mathit{NewTree} \leftarrow (\Gamma_G(\mathit{NewTree}))$

  $\mathit{Trees.insert}(\mathit{NewTree})$

return Trees

Conclusion

What I did not talk about:

1. alternatives to bottom-up plan generation
   1.1 top-down plan generation
   1.2 rule-based plan generation
   1.3 hybrids
2. cardinality estimation
3. cost functions