Dynamic Programming-Based Plan Generation

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

Student → Attend → Lecture

s.SNo = a.ASNo
a.ALNo = l.LNo
l.LPNo = p.PNo
p.PName = 'Larson'

Professor

1.LPNo = p.PNo

p.PName = 'Larson'
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 × Possible Join Tree Classes × 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 $4 \times 3 \times 2 \times 2 = 48$ different join ordering problems.
### Search Space Size (No Cross Product)

<table>
<thead>
<tr>
<th>n</th>
<th>Chain Query</th>
<th>Star Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left-Deep</td>
<td>Bushy</td>
</tr>
<tr>
<td>1</td>
<td>$2^{n-1}$</td>
<td>$2^{n-1}C(n-1)$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>224</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>1344</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>8448</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>54912</td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td>366080</td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td>2489344</td>
</tr>
</tbody>
</table>
## Search Space Size (With Cross Products)

<table>
<thead>
<tr>
<th>n</th>
<th>Left-Deep</th>
<th>With Cross Products/Clique</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n!</td>
<td>n!C(n − 1)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>1680</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>30240</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>665280</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>17297280</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>518918400</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>17643225600</td>
</tr>
</tbody>
</table>
## Complexity

<table>
<thead>
<tr>
<th>Query Graph</th>
<th>Join Tree</th>
<th>Cost Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>general tree/star/chain star</td>
<td>left-deep left-deep left-deep</td>
<td>ASI one join method (ASI) two join methods (NLJ+SMJ)</td>
<td>NP-hard P NP-hard</td>
</tr>
<tr>
<td>general/tree/star chain</td>
<td>left-deep left-deep</td>
<td>ASI —</td>
<td>NP-hard open</td>
</tr>
<tr>
<td>general tree star</td>
<td>bushy bushy bushy</td>
<td>ASI — ASI</td>
<td>NP-hard open P</td>
</tr>
<tr>
<td>chain</td>
<td>bushy</td>
<td>any</td>
<td>P</td>
</tr>
<tr>
<td>general tree/star/chain</td>
<td>bushy bushy</td>
<td>ASI</td>
<td>NP-hard NP-hard</td>
</tr>
</tbody>
</table>
Dynamic Programming

- Optimality Principle
- Avoid duplicate work
Optimality Principle

Consider the two join trees

\[((((R_1 \Join R_2) \Join R_3) \Join R_4) \Join R_5)\]

and

\[((((R_3 \Join R_1) \Join R_2) \Join R_4) \Join R_5)\].

If we know that \(((R_1 \Join R_2) \Join R_3)\) is cheaper than \(((R_3 \Join R_1) \Join R_2)\), 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 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 $\text{CreateJoinTree}$ 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.
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 trees:

\[
T_1 \Join T_2 \quad \text{and} \quad T_2 \Join T_1
\]

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 \texttt{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 \texttt{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 $\#_{ccp}$. Then

\[
\begin{align*}
\#_{ccp}^{\text{chain}}(n) &= \frac{1}{6}(n^3 - n) \\
\#_{ccp}^{\text{cycle}}(n) &= \frac{(n^3 - 2n^2 + n)}{2} \\
\#_{ccp}^{\text{star}}(n) &= (n - 1)2^{n-2} \\
\#_{ccp}^{\text{clique}}(n) &= \frac{(3^n - 2^{n+1} + 1)}{2}
\end{align*}
\]

These numbers have to be multiplied by two if we want to count all csg-cmp-pairs.
for all  $R_i \in R$  $\text{BestPlan} \{ \{R_i\} \} = R_i$;
for all  $1 < s \leq n$ ascending // size of plan
for all  $1 \leq s_1 < s$ // size of left subplan
  $s_2 = s - s_1$; // size of right subplan
for all  $p_1 = \text{BestPlan} \{ S_1 \subset R : |S_1| = s_1 \}$
for all  $p_2 = \text{BestPlan} \{ S_2 \subset R : |S_2| = s_2 \}$
  $\text{++InnerCounter}$;
if  $(\emptyset \neq S_1 \cap S_2)$ continue;
if not  $(S_1$ connected to $S_2)$ continue;
  $\text{++CsgCmpPairCounter}$;
CurrPlan = CreateJoinTree($p_1$, $p_2$);
if  $(\text{cost}(\text{BestPlan} \{ S_1 \cup S_2 \}) > \text{cost}(\text{CurrPlan}))$
  $\text{BestPlan} \{ S_1 \cup S_2 \} = \text{CurrPlan}$;
OnoLohmanCounter = CsgCmpPairCounter / 2;
return  $\text{BestPlan} \{ \{R_0, \ldots, R_{n-1}\} \}$;
Analysis: DPsize

\[
\begin{align*}
\text{chain } & \quad \text{DPsize } (n) = \\
& \begin{cases} 
1/48(5n^4 + 6n^3 - 14n^2 - 12n) & \text{if } n \text{ even} \\
1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & \text{if } n \text{ odd}
\end{cases} \\
\text{cycle } & \quad \text{DPsize } (n) = \\
& \begin{cases} 
1/4 (n^4 - n^3 - n^2) & \text{if } n \text{ even} \\
1/4 (n^4 - n^3 - n^2 + n) & \text{if } n \text{ odd}
\end{cases} \\
\text{star } & \quad \text{DPsize } (n) = \\
& \begin{cases} 
2^{2n-4} - 1/4({2(n-1) \choose n-1}) + q(n) & \text{if } n \text{ even} \\
2^{2n-4} - 1/4({2(n-1) \choose n-1}) + 1/4({n-1 \choose (n-1)/2}) + q(n) & \text{if } n \text{ odd}
\end{cases} \\
\text{q}(n) & = n2^{n-1} - 5 \times 2^{n-3} + 1/2(n^2 - 5n + 4) \\
\text{clique } & \quad \text{DPsize } (n) = \\
& \begin{cases} 
2^{2n-2} - 5 \times 2^{n-2} + 1/4({2n \choose n}) - 1/4({n \choose n/2}) + 1 & \text{if } n \text{ even} \\
2^{2n-2} - 5 \times 2^{n-2} + 1/4({2n \choose n}) + 1 & \text{if } n \text{ odd}
\end{cases}
\end{align*}
\]
for all $R_i \in R$ BestPlan($\{R_i\}$) = $R_i$;
for $1 \leq i < 2^n - 1$ ascending
$S = \{R_j \in R|([i/2^j] \mod 2) = 1\}$
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

$I^\text{chain}_{\text{DPsub}}(n) = 2^{n+2} - n^2 - 3n - 4$

$I^\text{cycle}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2$

$I^\text{star}_{\text{DPsub}}(n) = 2 \times 3^{n-1} - 2^n$

$I^\text{clique}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1$
## Sample Numbers

<table>
<thead>
<tr>
<th>$n$</th>
<th>#ccp</th>
<th>DPsub</th>
<th>DPsize</th>
<th>#ccp</th>
<th>DPsub</th>
<th>DPsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>84</td>
<td>73</td>
<td>40</td>
<td>140</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>165</td>
<td>3962</td>
<td>1135</td>
<td>405</td>
<td>11062</td>
<td>2225</td>
</tr>
<tr>
<td>15</td>
<td>560</td>
<td>130798</td>
<td>5628</td>
<td>1470</td>
<td>523836</td>
<td>11760</td>
</tr>
<tr>
<td>20</td>
<td>1330</td>
<td>4193840</td>
<td>17545</td>
<td>3610</td>
<td>22019294</td>
<td>37900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>#ccp</th>
<th>DPsub</th>
<th>DPsize</th>
<th>#ccp</th>
<th>DPsub</th>
<th>DPsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
<td>130</td>
<td>110</td>
<td>90</td>
<td>180</td>
<td>280</td>
</tr>
<tr>
<td>10</td>
<td>2304</td>
<td>38342</td>
<td>57888</td>
<td>28501</td>
<td>57002</td>
<td>306991</td>
</tr>
<tr>
<td>15</td>
<td>114688</td>
<td>9533170</td>
<td>57305929</td>
<td>7141686</td>
<td>14283372</td>
<td>307173877</td>
</tr>
<tr>
<td>20</td>
<td>4980736</td>
<td>2323474358</td>
<td>59892991338</td>
<td>1742343625</td>
<td>3484687250</td>
<td>309338182241</td>
</tr>
</tbody>
</table>
Algorithm \text{DPccp}

\begin{align*}
\text{for all } & (R_i \in \mathcal{R}) \text{ BestPlan} \{\{R_i\}\} = R_i; \\
\text{for all } & \text{csg-cmp pairs } (S_1, S_2), \ S = S_1 \cup S_2 \\
& \quad \quad +\text{InnerCounter} \\
& \quad \quad +\text{OnoLohmanCounter} \\
& \quad p_1 = \text{BestPlan} (S_1) \\
& \quad p_2 = \text{BestPlan} (S_2) \\
& \quad \text{CurrPlan} = \text{CreateJoinTree} (p_1, p_2); \\
& \quad \text{if } (\text{cost} (\text{BestPlan} (S)) > \text{cost} (\text{CurrPlan})) \\
& \quad \quad \text{BestPlan} (S) = \text{CurrPlan} \\
& \quad \text{CurrPlan} = \text{CreateJoinTree} (p_2, p_1); \\
& \quad \text{if } (\text{cost} (\text{BestPlan} (S)) > \text{cost} (\text{CurrPlan})) \\
& \quad \quad \text{BestPlan} (S) = \text{CurrPlan}; \\
& \text{CsgCmpPairCounter} = 2 \times \text{OnoLohmanCounter}; \\
\text{return } & \text{BestPlan} \{\{R_0, \ldots, R_{n-1}\}\};
\end{align*}
Notation

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

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

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

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

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

$$N(S \cup S') = (N(S) \cup N(S')) \setminus (S \cup S').$$

This allows for an efficient bottom-up calculation of neighborhoods.

$$B_i = \{v_j | j \leq i\}$$
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($G$, $\{v_i\}$, $B_i$);
}
Subroutine EnumerateCsgRec

EnumerateCsgRec(G, S, X)

\[ N = N(S) \setminus X; \]

for all \( S' \subseteq N, S' \neq \emptyset \), enumerate subsets first { 
  \[ \text{emit } (S \cup S'); \]
}

for all \( S' \subseteq N, S' \neq \emptyset \), enumerate subsets first { 
  \[ \text{EnumerateCsgRec}(G, (S \cup S'), (X \cup N)); \]
}
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

$$X = \mathcal{B}_{\min(S_1)} \cup S_1;$$
$$N = \mathcal{N}(S_1) \setminus X;$$

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

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

▷ **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

1. **for all** \( R_i \in R \)
2. \( DPTable[R_i] \leftarrow R_i \) ▷ initial access paths
3. **for all** csg-cmp-pairs \( (S_1, S_2) \) of \( H \)
4. **for all** \( \circ p \in O \)
5. \[ \text{if } \text{APPLICABLE}(S_1, S_2, \circ p) \]
6. \( \text{BUILDPANS}(S_1, S_2, \circ p) \)
7. \[ \text{if } \circ p \text{ is commutative} \]
8. \( \text{BUILDPANS}(S_2, S_1, \circ p) \)
9. **return** \( DPTable[R] \)
Preliminaries (strict predicates)

Definition
A predicate is null rejecting for a set of attributes $A$ if it evaluates to $\text{false}$ or $\text{unknown}$ on every tuple in which all attributes in $A$ are $\text{null}$.

Synonyms for null rejecting are used: $\text{null intolerant}$, $\text{strong}$, and $\text{strict}$. 
We assume that we have an initial operator tree, e.g., by a canonical translation of a SQL query.
For a set of attributes $A$, $\text{REL}(A)$ denotes the set of tables to which these attributes belong. We abbreviate $\text{REL}(\mathcal{F}(e))$ by $\mathcal{F}_T(e)$. Let $\circ$ be an operator in the initial operator tree. We denote by $\text{left}(\circ)$ ($\text{right}(\circ)$) its left (right) child. $\text{STO}(\circ)$ denotes the operators contained in the operator subtree rooted at $\circ$. $\text{REL}(\circ)$ denotes the set of tables contained in the subtree rooted at $\circ$. 
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 $\text{SES}(\circ_p) = \{R, S, T\}$. 
Preliminaries (degenerate predicates)

Definition
Let $p$ be a predicate associated with a binary operator $\circ$ and $\mathcal{F}_T(p)$ the tables referenced by $p$. Then, $p$ is called degenerate if $\text{REL}(\text{left}(\circ)) \cap \mathcal{F}_T(p) = \emptyset \lor \text{REL}(\text{right}(\circ)) \cap \mathcal{F}_T(p) = \emptyset$ 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:

\[
\mathcal{N}(S, X) = \bigcup_{v \in E \downarrow(S, X)} \min(v) \tag{1}
\]

where \(X\) is the set of forbidden nodes.
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)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>X</td>
<td>+</td>
</tr>
<tr>
<td>x</td>
<td>X</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reorderability (assoc)

assoc:

\[(e_1 \circ_{12}^a e_2) \circ_{23}^b e_3 \equiv e_1 \circ_{12}^a (e_2 \circ_{23}^b e_3)\]  \hspace{1cm} (2)
Reorderability (assoc)

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

1. if \(p_{23}\) rejects nulls on \(A(e_2)\) (Eqv. 2)
2. if \(p_{12}\) and \(p_{23}\) reject nulls on \(A(e_2)\) (Eqv. 2)
Reorderability (l/r-asscom)

Consider the following truth about the semijoin:

\[(e_1 \bowtie_{12} e_2) \bowtie_{13} e_3 \equiv (e_1 \bowtie_{13} e_3) \bowtie_{12} e_2.\]

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:

\[(e_1 \circ_{12}^a e_2) \circ_{13}^b e_3 \equiv (e_1 \circ_{13}^b e_3) \circ_{12}^a e_2.\]  \hspace{1cm} (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):

\[e_1 \circ_{13}^a (e_2 \circ_{23}^b e_3) \equiv e_2 \circ_{23}^b (e_1 \circ_{13}^a e_3).\]  \hspace{1cm} (4)

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

\[
\text{l-asscom}(\circ^a, \circ^b) \iff \text{l-asscom}(\circ^b, \circ^a),
\]

\[
\text{r-asscom}(\circ^a, \circ^b) \iff \text{r-asscom}(\circ^b, \circ^a).
\]
Reorderability (l/r-asscom)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>×</td>
<td>∨</td>
<td>×</td>
<td>∨</td>
<td>×</td>
<td>∨</td>
</tr>
<tr>
<td>×</td>
<td>++</td>
<td>++</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>−/ −</td>
</tr>
<tr>
<td>∨</td>
<td>++</td>
<td>++</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>−/ −</td>
</tr>
<tr>
<td>×</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>−/ −</td>
</tr>
<tr>
<td>∨</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ −</td>
</tr>
<tr>
<td>▽</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ -</td>
<td>+/ −</td>
</tr>
<tr>
<td>△</td>
<td>−/ −</td>
<td>−/ −</td>
<td>−/ −</td>
<td>−/ −</td>
<td>+/ 2/ −</td>
<td>+/ 3/ + 4</td>
</tr>
<tr>
<td>△'</td>
<td>+/ −</td>
<td>+/ −</td>
<td>+/ −</td>
<td>+/ −</td>
<td>+/ −</td>
<td>+/ −</td>
</tr>
</tbody>
</table>

1. if $p_{12}$ rejects nulls on $A(e_1)$ (Eqv. 3)
2. if $p_{13}$ rejects nulls on $A(e_3)$ (Eqv. 3)
3. if $p_{12}$ and $p_{13}$ rejects nulls on $A(e_1)$ (Eqv. 3)
4. if $p_{13}$ and $p_{23}$ reject nulls on $A(e_3)$ (Eqv. 4)
Conflict Detector CD-A: $\text{SES}$

\[
\begin{align*}
\text{SES}(R) &= \{ R \} \\
\text{SES}(T) &= \{ T \} \\
\text{SES}(\circ_p) &= \bigcup_{R \in \mathcal{F}_T(p)} \text{SES}(R) \cap \text{REL}(\circ_p) \\
\text{SES}(\nabla_p; a_1:e_1, ..., a_n:e_n) &= \bigcup_{R \in \mathcal{F}_T(p) \cup \mathcal{F}_T(e_i)} \text{SES}(R) \cap \text{REL}(g_j)
\end{align*}
\]
initially: \( \text{TES}(\circ_p) := \text{SES}(\circ_p) \)
Conflict Detector CD-A: \textit{TES}: right conflict

\begin{align*}
\neg \text{assoc}(\circ b, \circ a) & \quad \neg \text{r-asscom}(\circ a, \circ b) \\
\text{TES}(\circ b) \cup = \text{REL}(e_2) & \quad \text{TES}(\circ b) \cup = \text{REL}(e_1)
\end{align*}
Conflict Detector CD-A: Remarks

- correct
- not complete
applicable(\circ, S_1, S_2) := \text{tesl}(\circ) \subseteq S_1 \land \text{tesr}(\circ) \subseteq S_2.

where

\text{tesl}(\circ) := \text{TES}(\circ) \cap \text{REL}(\text{left}(\circ))

\text{tesr}(\circ) := \text{TES}(\circ) \cap \text{REL}(\text{right}(\circ))
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 $r = TES(\circ) \cap REL(right(\circ)) = R-TES(\circ)$ and $l = TES(\circ) \setminus r = L-TES(\circ)$. 
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 \texttt{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 $B_v = \{w | w \prec v\} \cup \{v\}$. 
EnumerateCsgRec($S_1, X$)

for each $N \subseteq N(S_1, X): N \neq \emptyset$

    if DpTable[$S_1 \cup N] \neq \emptyset$

        EmitCsg($S_1 \cup N$)

for each $N \subseteq N(S_1, X): N \neq \emptyset$

    EnumerateCsgRec($S_1 \cup N, X \cup N(S_1, X)$)
Csg-Cmp-Enumeration: EmitCsg

\[
\text{EmitCsg}(S_1) \\
X = S_1 \cup B_{\min}(S_1) \\
N = N(S_1, X) \\
\text{for each } v \in N \text{ descending according to } \prec \\
S_2 = \{v\} \\
\text{if } \exists (u, v) \in E : u \subseteq S_1 \land v \subseteq S_2 \\
\quad \text{EmitCsgCmp}(S_1, S_2) \\
\quad \text{EnumerateCmpRec}(S_1, S_2, X \cup B_v(N))
\]

where \( B_v(W) = \{w | w \in W, w \leq v\} \) is defined in DPccp.
Csg-Cmp-Enumeration: \texttt{EnumerateCmpRec}

\begin{itemize}
\item \texttt{EnumerateCmpRec}(S_1, S_2, X) \\
\textbf{for each} \( N \subseteq N(S_2, X) : N \neq \emptyset \) \\
\textbf{if} \ \text{DpTable}[S_2 \cup N] \neq \emptyset \land \\
\exists (u, v) \in E : u \subseteq S_1 \land v \subseteq S_2 \cup N \\
\text{EmitCsgCmp}(S_1, S_2 \cup N) \\
X = X \cup N(S_2, X) \\
\textbf{for each} \( N \subseteq N(S_2, X) : N \neq \emptyset \) \\
\texttt{EnumerateCmpRec}(S_1, S_2 \cup N, X)
\end{itemize}
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
If outerjoins with defaults other than padding with NULL values are used, grouping can be pushed down any join operator.
Algorithm: Top-Level

\[ \text{DPHYPE} \]

▷ **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

1. **for all** \( R_i \in R \)
2. \( \text{DPTable}[R_i] \leftarrow R_i \) ▷ initial access paths
3. **for all** csg-cmp-pairs \((S_1, S_2)\) of \( H \)
4. **for all** \( \circ_p \in O \)
5. \( \text{if} \ \text{APPLICABLE}(S_1, S_2, \circ_p) \)
6. \( \text{BUILD TREE}(S_1, S_2, \circ_p) \)
7. \( \text{if} \ \circ_p \text{ is commutative} \)
8. \( \text{BUILD TREE}(S_2, S_1, \circ_p) \)
9. **return** \( \text{DPTable}[R] \)
Algorithm: BuildTree

$$\text{BUILD TREE}(S_1, S_2, \circ_p)$$

1 $S \leftarrow S_1 \cup S_2$
2 \text{for each } $T_1 \in DPTable[S_1]$
3 \quad \text{for each } $T_2 \in DPTable[S_2]$
4 \quad \quad \text{for each } $T \in \text{OPTREES}(T_1, T_2, \circ_p)$
5 \quad \quad \quad \text{if } S == R
6 \quad \quad \quad \quad \text{INSERT TOP LEVEL PLAN}(S, T)$
7 \quad \quad \quad \text{else}
8 \quad \quad \quad \quad \text{INSERT NON TOP LEVEL PLAN}(S, T)$
OpTrees

\textbf{OpTrees}(T_1, T_2, \circ p)

1. \( S_1 \leftarrow T(T_1) \)
2. \( S_2 \leftarrow T(T_2) \)
3. \( S \leftarrow S_1 \cup S_2 \)
4. \( \text{Trees} \leftarrow \emptyset \)
5. \( \text{NewTree} \leftarrow (T_1 \circ p T_2) \)
6. \textbf{if} \( S = R \land \text{NEEDSGROUPING}(G, \text{NewTree}) \)
7. \( \text{NewTree} \leftarrow (\Gamma_G(\text{NewTree})) \)
8. \( \text{Trees.insert(\text{NewTree})} \)
9. \( \text{NewTree} \leftarrow \Gamma_{G_1^+}(T_1) \circ p T_2 \)
10. \textbf{if} \ \text{VALID}(\text{NewTree}) \land \text{NEEDSGROUPING}(G_1^+, \text{NewTree}) \)
11. \textbf{if} \( S = R \land \text{NEEDSGROUPING}(G, \text{NewTree}) \)
12. \( \text{NewTree} \leftarrow (\Gamma_G(\text{NewTree})) \)
13. \( \text{Trees.insert(\text{NewTree})} \)
14. \( \text{NewTree} \leftarrow T_1 \circ p \Gamma_{G_2^+}(T_2) \)
1  if $\text{VALID}(\text{NewTree}) \land \text{NEEDSGROUPING}(G^+_2, \text{NewTree})$
2      if $S == R \land \text{NEEDSGROUPING}(G, \text{NewTree})$
3          $\text{NewTree} \leftarrow (\Gamma_G(\text{NewTree}))$
4            $\text{Trees.insert(\text{NewTree})}$
5      $\text{NewTree} \leftarrow \Gamma_{G_1^+}(T_1) \circ p \Gamma_{G_2^+}(T_2)$
6  if $\text{VALID}(\text{NewTree})$
7      $\land \text{NEEDSGROUPING}(G^+_1, \text{NewTree})$
8          $\land \text{NEEDSGROUPING}(G^+_2, \text{NewTree})$
9      if $S == R \land \text{NEEDSGROUPING}(G, \text{NewTree})$
10         $\text{NewTree} \leftarrow (\Gamma_G(\text{NewTree}))$
11            $\text{Trees.insert(\text{NewTree})}$
12  return $\text{Trees}$
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