Dynamic Programming-Based Plan Generation

1. Preliminaries
2. Simple Graphs, Inner Joins only
3. Hypergraphs
Preliminaries
Queries Considered

- conjunctive queries, i.e.
- conjunctions of simple predicates
- predicates of the form $e_1 \theta e_2$
  - $e_1$ is an attribute, $e_2$ is either an attribute or a constant
- join predicates: $\theta$ must be ‘$=$’ (equi-joins only)

We join relations $R_1, \ldots, R_n$ where $R_i$ can be
- a base relation
- a base relation to which a selection has been applied
- a more complex building block or access path
A *query graph* is an undirected graph with nodes $R_1, \ldots, R_n$. For every join predicate in the conjunction $P$ whose attributes belong to the relations $R_i$ and $R_j$, we add an edge between $R_i$ and $R_j$. This edge is labeled by the join predicate. For simple predicates applicable to a single relation, we add a self-edge. However, our algorithms will not consider simple selection predicates. They have to be pushed down before.
Example Query Graph

Student \( \rightarrow \) Attend \( \rightarrow \) Lecture

Professor

\( s.\text{SNo} = a.\text{ASNo} \)

\( 1.\text{LPNo} = p.\text{PNo} \)

\( p.\text{PName} = '\text{Larson}' \)

\( a.\text{ALNo} = 1.\text{LNo} \)
Shapes of Query Graphs

- **Chain queries**
- **Star queries**
- **Tree query**
- **Cyclic query**
- **Cycle queries**
- **Grid query**
- **Clique queries**
Join Trees

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*
Simple Cost Functions

Input:

- cardinalities: $|R_i|$
- selectivities: $f_{i,j}$ of $p_{i,j}$ is then defined as

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

Calculate:

- result cardinality:

$$|R_i \Join_{p_{i,j}} R_j| = f_{i,j} |R_i| |R_j|$$
Consider a join tree $T = T_1 \Join T_2$. Then, $|T|$ can be calculated as follows:

- If $T$ is a leaf $R_i$, then $|T| = |R_i|$.
- Otherwise,
  $$ |T| = \left( \prod_{R_i \in T_1, R_j \in T_2} f_{i,j} \right) |T_1| |T_2|. $$

This formula assumes independence.
Example

The table

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>= 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_2$</td>
<td>= 100</td>
</tr>
<tr>
<td>$R_3$</td>
<td>= 1000</td>
</tr>
<tr>
<td>$f_{1,2}$</td>
<td>= 0.1</td>
</tr>
<tr>
<td>$f_{2,3}$</td>
<td>= 0.2</td>
</tr>
</tbody>
</table>

implicitly defines the query graph $R_1 \rightarrow \rightarrow R_2 \rightarrow \rightarrow R_3$. (We assume $f_{i,j} = 1$ for all $i, j$ for which $f_{i,j}$ is not explicitly given.)
The cost function $C_{\text{out}}$

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \Join T_2 \end{cases}$$
Other cost functions

For single join operators:

\[
C_{nlj}(e_1 \bowtie_p e_2) = |e_1||e_2|
\]

\[
C_{hj}(e_1 \bowtie_p e_2) = h|e_1|
\]

\[
C_{smj}(e_1 \bowtie_p e_2) = |e_1|\log(|e_1|) + |e_2|\log(|e_2|)
\]

For sequences of join operators (relations):

\[
C_{hj}(s) = \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}|
\]

\[
C_{smj}(s) = \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| \log(|s_1, \ldots, s_{i-1}|) + \sum_{i=2}^{n} |s_i| \log(|s_i|)
\]

\[
C_{nlj}(s) = \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| \ast s_i
\]
Remarks on Cost Functions

- cost functions are simplistic
- cost functions designed for left-deep trees
- $C_{hj}$ and $C_{smj}$ do not work for cross products
  (Fix: define them then to be equal to the output cardinality
  which happens to be the costs of the nested-loop cost function)
- in reality: other parameters besides cardinality play a role
- the above cost functions assume that the same join
  algorithm is chosen throughout the whole plan
## Example Calculations

<table>
<thead>
<tr>
<th></th>
<th>$C_{out}$</th>
<th>$C_{nlj}$</th>
<th>$C_{hj}$</th>
<th>$C_{smj}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \times R_2$</td>
<td>100</td>
<td>1000</td>
<td>12</td>
<td>697.61</td>
</tr>
<tr>
<td>$R_2 \times R_3$</td>
<td>20000</td>
<td>100000</td>
<td>120</td>
<td>10630.26</td>
</tr>
<tr>
<td>$R_1 \times R_3$</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000.00</td>
</tr>
<tr>
<td>$(R_1 \times R_2) \times R_3$</td>
<td>20100</td>
<td>101000</td>
<td>132</td>
<td>11327.86</td>
</tr>
<tr>
<td>$(R_2 \times R_3) \times R_1$</td>
<td>40000</td>
<td>300000</td>
<td>24120</td>
<td>32595.00</td>
</tr>
<tr>
<td>$(R_1 \times R_3) \times R_2$</td>
<td>30000</td>
<td>1010000</td>
<td>22000</td>
<td>143542.00</td>
</tr>
</tbody>
</table>
Observations

- Costs differ vastly
- different cost functions result in different costs
- the cheapest join tree is the cheapest one under all cost functions
- join trees with cross products are expensive
- the order in which relations are joined is essential under all (and other) cost functions
Another Example

Query: $|R_1| = 1000$, $|R_2| = 2$, $|R_3| = 2$, $f_{1,2} = 0.1$, $f_{1,3} = 0.1$

For $C_{out}$ we have costs

<table>
<thead>
<tr>
<th>Join Tree</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \Join R_2$</td>
<td>200</td>
</tr>
<tr>
<td>$R_2 \times R_3$</td>
<td>4</td>
</tr>
<tr>
<td>$R_1 \Join R_3$</td>
<td>200</td>
</tr>
<tr>
<td>$(R_1 \Join R_2) \Join R_3$</td>
<td>240</td>
</tr>
<tr>
<td>$(R_2 \times R_3) \Join R_1$</td>
<td>44</td>
</tr>
<tr>
<td>$(R_1 \Join R_3) \Join R_2$</td>
<td>240</td>
</tr>
</tbody>
</table>

Plan with cross product is best.
Yet Another Example

Query: $|R_1| = 10$, $|R_2| = 20$, $|R_3| = 20$, $|R_4| = 10$, $f_{1,2} = 0.01$, $f_{2,3} = 0.5$, $f_{3,4} = 0.01$

<table>
<thead>
<tr>
<th>Join Tree</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \Join R_2$</td>
<td>2</td>
</tr>
<tr>
<td>$R_2 \Join R_3$</td>
<td>200</td>
</tr>
<tr>
<td>$R_3 \Join R_4$</td>
<td>2</td>
</tr>
<tr>
<td>$((R_1 \Join R_2) \Join R_3) \Join R_4$</td>
<td>24</td>
</tr>
<tr>
<td>$((R_2 \Join R_3) \Join R_1) \Join R_4$</td>
<td>222</td>
</tr>
<tr>
<td>$(R_1 \times R_2) \Join (R_3 \times R_4)$</td>
<td>6</td>
</tr>
</tbody>
</table>

Bushy tree better than any left-deep tree
A cost function $C_{\text{impl}}$ is called *symmetric*, if

$$C_{\text{impl}}(R_1 \Join^{\text{impl}} R_2) = C_{\text{impl}}(R_2 \Join^{\text{impl}} R_1)$$

for all relations $R_1$ and $R_2$.

For symmetric cost functions, it does not make sense to consider commutativity.

ASI: adjacent sequence interchange (see below)

<table>
<thead>
<tr>
<th></th>
<th>ASI</th>
<th>~ASI</th>
</tr>
</thead>
<tbody>
<tr>
<td>symmetric</td>
<td>$C_{\text{out}}$</td>
<td>$C_{\text{smj}}$</td>
</tr>
<tr>
<td>~ symmetric</td>
<td>$C_{\text{hj}}$</td>
<td>(listen)</td>
</tr>
</tbody>
</table>
Classification of Join Ordering Problems

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

- Query Graph Classes: \textit{chain}, \textit{star}, \textit{tree}, and \textit{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

1. with cross products
2. without cross products
Number of Linear Trees with Cross Products

- left-deep: \( n! \)
- right-deep: \( n! \)
- zig-zag: \( 2^{n-2} n! \)
Remember: Catalan Numbers

For \( n \) leave nodes, the number of binary trees is given by \( C(n - 1) \) where \( C(n) \) is defined by the recurrence

\[
C(n) = \sum_{k=0}^{n-1} C(k)C(n - k - 1)
\]

with \( C(0) = 1 \). They can also be computed by the following formula:

\[
C(n) = \frac{1}{n+1} \binom{2n}{n}
\]

The Catalan Numbers grow in the order of \( \Theta(4^n/n^{3/2}) \).
Number of Bushy Trees with Cross Products

\[ n! \ C(n-1) = n! \ \frac{1}{n} \ \binom{2(n-1)}{n-1} = n! \ \frac{1}{n} \ \frac{(2n-2)!}{(n-1)! \ ((2n-2) - (n-1))!} = \frac{(2n-2)!}{(n-1)!} \]
Let us denote the number of join trees for a chain query in $n$ relations with query graph $R_1 - R_2 - \ldots - R_{n-1} - R_n$ as $f(n)$. Obviously: $f(0) = 1$ and $f(1) = 1$. 
... for $n > 1$

For larger $n$:
Consider the join trees for $R_1 - \ldots - R_{n-1}$ where

- $R_{n-1}$ is the $k$-th relation from the bottom
where $k$ ranges from 1 to $n - 1$.

From such a join tree we can derive join trees for all $n$ relations
by adding relation $R_n$ at any position following $R_{n-1}$.
There are $n - k$ such join trees.
Only for $k = 1$, we can also add $R_n$ below $R_{n-1}$. Hence, for
$k = 1$ we have $n$ join trees.
How many join trees with $R_{n-1}$ at position $k$ are there?
For $k = 1$, $R_{n-1}$ must be the first relation to be joined. Since we do not consider cross products, it must be joined with $R_{n-2}$. The next relation must be $R_{n-3}$, and so on. Hence, there is only one such join tree.

For $k = 2$, the first relation must be $R_{n-2}$ which is then joined with $R_{n-1}$. Then $R_{n-3}, \ldots, R_1$ must follow in this order. Again, there is only one such join tree.

For higher $k$, for $R_{n-1}$ to occur savely at position $k$ (no cross products) the $k - 1$ relations $R_{n-2}, \ldots, R_{n-k}$ must occur before $R_{n-1}$. There are exactly $f(k - 1)$ join trees for the $k - 1$ relations. On each such join tree we just have to add $R_{n-1}$ on top of it to yield a join tree with $R_{n-1}$ at position $k$. 
Now we can compute the $f(n)$ as

$$f(n) = n + \sum_{k=2}^{n-1} f(k - 1) \ast (n - k)$$

for $n > 1$.

Solving the recurrence gives us

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

(Exercise)
Let $f(n)$ be the number of bushy trees without cross products for a chain query in $n$ relations with query graph $R_1 - R_2 - \ldots - R_{n-1} - R_n$.

Obvious: $f(0) = 1$ and $f(1) = 1$. 

... for $n > 1$

Every subtree of the join tree must contain a subchain in order to avoid cross products. Every subchain can be joined in either the left or the right argument of the join. Thus:

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

This is equal to

$$2^{n-1} \cdot C(n - 1)$$

(Exercise)
Star Queries, No Cartesian Product

Star: $R_0$ in the center, $R_1, \ldots, R_{n-1}$ as satellites
The first join must involve $R_0$.
The order of the remaining relations does not matter.

- left-deep trees: $2 \times (n-1)!$
- right-deep trees: $2 \times (n-1)!$
- zig-zag trees: $2 \times (n-1)! \times 2^{n-2} = 2^{n-1} \times (n-1)!$
Remark

The numbers for star queries are also upper bounds for tree queries. For clique queries, there is no join tree possible that does contain a cross product. Hence, all join trees are valid join trees and the search space size is the same as the corresponding search space for join trees with cross products.
### Numbers

#### Join Trees Without Cross Products

<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>Zig-Zag</td>
</tr>
<tr>
<td></td>
<td>$2^{n-1}$</td>
<td>$2^{2n-3}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>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>32</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>2048</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>8192</td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td>32768</td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td>131072</td>
</tr>
</tbody>
</table>
## Numbers

<table>
<thead>
<tr>
<th>n</th>
<th>Left-Deep $n!$</th>
<th>Zig-Zag $2^{n-2} \cdot n!$</th>
<th>Bushy $n!C(n-1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>96</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>960</td>
<td>1680</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>11520</td>
<td>30240</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>161280</td>
<td>665280</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>2580480</td>
<td>17297280</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>46448640</td>
<td>518918400</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>928972800</td>
<td>17643225600</td>
</tr>
</tbody>
</table>
## Complexity

<table>
<thead>
<tr>
<th>Query Graph</th>
<th>Join Tree</th>
<th>$\times s$</th>
<th>Cost Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>general tree/star/chain star</td>
<td>left-deep</td>
<td>no</td>
<td>ASI one join method (ASI)</td>
<td>NP-hard</td>
</tr>
<tr>
<td></td>
<td>left-deep</td>
<td>no</td>
<td>two join methods (NLJ+SMJ)</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>left-deep</td>
<td>no</td>
<td></td>
<td>NP-hard</td>
</tr>
<tr>
<td>general/tree/star chain</td>
<td>left-deep</td>
<td>yes</td>
<td>ASI —</td>
<td>NP-hard open</td>
</tr>
<tr>
<td></td>
<td>left-deep</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>general tree star</td>
<td>bushy</td>
<td>no</td>
<td>ASI —</td>
<td>NP-hard open</td>
</tr>
<tr>
<td></td>
<td>bushy</td>
<td>no</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>bushy</td>
<td>no</td>
<td>ASI —</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bushy</td>
<td>no</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>chain</td>
<td>bushy</td>
<td>no</td>
<td>any</td>
<td>P</td>
</tr>
<tr>
<td>general tree/star/chain</td>
<td>bushy</td>
<td>yes</td>
<td>ASI</td>
<td>NP-hard</td>
</tr>
<tr>
<td></td>
<td>bushy</td>
<td>yes</td>
<td>ASI</td>
<td>NP-hard</td>
</tr>
<tr>
<td></td>
<td>bushy</td>
<td>yes</td>
<td>ASI</td>
<td>NP-hard</td>
</tr>
</tbody>
</table>
Dynamic Programming

- Optimality Principle
- Avoid duplicate work
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 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
CreateJoinTree($T_1$, $T_2$)

Input: two (optimal) join trees $T_1$ and $T_2$.
   for linear trees: assume that $T_2$ is a single relation

Output: an (optimal) join tree for joining $T_1$ and $T_2$.

BestTree = NULL;
for all implementations impl do {
   if (!RightDeepOnly)
      Tree = $T_1 \Join^{impl} T_2$
      if (BestTree == NULL || cost(BestTree) > cost(Tree))
         BestTree = Tree;
   if (!LeftDeepOnly)
      Tree = $T_2 \Join^{impl} T_1$
      if (BestTree == NULL || cost(BestTree) > cost(Tree))
         BestTree = Tree;
}
return BestTree;
Search space with sharing under Optimality Principle

\{R_1 R_2 R_3 R_4\}

\{R_1 R_2 R_4\}  \{R_1 R_3 R_4\}

\{R_1 R_2 R_3\}  \{R_2 R_3 R_4\}

\{R_1 R_4\}  \{R_2 R_4\}

\{R_1 R_3\}  \{R_3 R_4\}

\{R_1 R_2\}  \{R_2 R_3\}

\{R_2 R_3\}  \{\}\n
R_1  R_2  R_3  R_4
DP-Linear-1(\{R_1, \ldots, R_n\})

Input: a set of relations to be joined

Output: an optimal left-deep (right-deep, zig-zag) join tree

for (i = 1; i <= n; ++i) BestTree(\{R_i\}) = R_i;

for (i = 1; i < n; ++i) {
    for all \( S \subseteq \{R_1, \ldots, R_n\}, \ |S| = i \) do {
        for all \( R_i \in \{R_1, \ldots, R_n\}, \ R_i \not\in S \) do {
            if (NoCrossProducts && !connected(\{R_i\}, S)) {
                continue;
            }
            CurrTree = CreateJoinTree(BestTree(S), R_i);
            S' = S \cup \{R_i\};
            if (BestTree(S') == NULL
                || cost(BestTree(S')) > cost(CurrTree)) {
                BestTree(S') = CurrTree;
            }
        }
    }
}

return BestTree(\{R_1, \ldots, R_n\});
Order in which subtrees are generated

The order in which subtrees are generated does not matter as long as the following condition is not violated:

Let $S$ be a subset of \{ $R_1, \ldots, R_n$ \}. Then, before a join tree for $S$ can be generated, the join trees for all relevant subsets of $S$ must already be available.

Exercise: fix the semantics of relevant
Generation in integer order

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td>{R_1}</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td>{R_2}</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td>{R_1, R_2}</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>{R_3}</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>{R_1, R_3}</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>{R_2, R_3}</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>{R_1, R_2, R_3}</td>
<td></td>
</tr>
</tbody>
</table>
DP-Linear-2(\{R_1, \ldots, R_n\})

Input: a set of relations to be joined
Output: an optimal left-deep (right-deep, zig-zag)

for (i = 1; i <= n; ++i) { BestTree(i) = R_i; }
for (S = 1; S < 2^n-1; ++S) {
    if (BestTree(S) != NULL) continue;
    for all i ∈ S do {
        S' = S \ \{i\};
        CurrTree = CreateJoinTree(BestTree(S'), R_i);
        if (cost(BestTree(S)) > cost(CurrTree)) {
            BestTree(S) = CurrTree;
        }
    }
}
return BestTree(2^n - 1);
bushy trees with cross products

DP-Bushy($\{R_1, \ldots, R_n\}$)

**Input:** a set of relations to be joined

**Output:** an optimal bushy join tree

for (i = 1; i <= n; ++i)

    BestTree(1 << i) = $R_i$;

for (S = 1; S < $2^n$-1; ++S) {

    if (BestTree(S) != NULL) continue;

    for all $S_1 \subset S$ do

        $S_2 = S \setminus S_1$;

        CurrTree = CreateJoinTree(BestTree($S_1$), BestTree($S_2$));

        if (BestTree(S) == NULL

            || cost(BestTree(S)) > cost(CurrTree))

            BestTree(S) = CurrTree;

    }

return BestTree($2^n - 1$);
Subset Generation for Bushy Trees

\[ S_1 = S \& - S; \]

\[
\text{do } \{
    \text{/* do something with subset } S_1 \text{ */}
    S_1 = S \& (S_1 - S);
\}\text{ while } (S_1 \neq S); \\
\]

\(S\) represents the input set. \(S_1\) iterates through all subsets of \(S\) where \(S\) itself and the empty set are not considered.
### Number of join trees investigated by DP

<table>
<thead>
<tr>
<th>n</th>
<th>without cross products</th>
<th>with cross products</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>chain</td>
<td>star</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>linear</td>
<td>bushy</td>
</tr>
<tr>
<td>n</td>
<td>$(n - 1)^2$</td>
<td>$(n^3 - n)/6$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>36</td>
<td>56</td>
</tr>
<tr>
<td>8</td>
<td>49</td>
<td>84</td>
</tr>
<tr>
<td>9</td>
<td>64</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td>165</td>
</tr>
</tbody>
</table>
Optimal Bushy Trees without Cross Products

**Given:** Connected join graph

**Problem:** Generate optimal bushy trees without cross products

**Steps:**
1. minimal bound for all DP algorithms
2. DPsize
3. DPsub
4. DPccp
Csg-Cmp-Pairs

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 connected 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 trees:

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

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 \( R_i \in R \) 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 \( p_1 = \text{BestPlan}(S_1 \subset R : |S_1| = s_1) \)
\( p_2 = \text{BestPlan}(S_2 \subset R : |S_2| = s_2) \)
++InnerCounter;
if \( \emptyset \neq S_1 \cap S_2 \) continue;
if not \( S_1 \) connected to \( S_2 \) continue;
++CsgCmpPairCounter;
CurrPlan = CreateJoinTree(\( p_1, p_2 \));
if \( \text{cost(\( \text{BestPlan}(S_1 \cup S_2) \)) > \text{cost(CurrPlan)} \}) \) BestPlan
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan(\( \{R_0, \ldots, R_{n-1}\} \));
Analysis: DPsize

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

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

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

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

\[ I_{\text{clique}}^{\text{DPsize}}(n) = \begin{cases} 
2^{2n-2} - 5 \times 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & \text{n even} \\
2^{2n-2} - 5 \times 2^{n-2} + 1/4\binom{2n}{n} + 1 & \text{n odd} 
\end{cases} \]
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

\[
\begin{align*}
I_{\text{chain}}^{\text{DPsub}}(n) &= 2^{n+2} - n^2 - 3n - 4 \\
I_{\text{cycle}}^{\text{DPsub}}(n) &= n2^n + 2^n - 2n^2 - 2 \\
I_{\text{star}}^{\text{DPsub}}(n) &= 2 \times 3^{n-1} - 2^n \\
I_{\text{clique}}^{\text{DPsub}}(n) &= 3^n - 2^{n+1} + 1
\end{align*}
\]
### Sample Numbers

<table>
<thead>
<tr>
<th>Chain</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>#ccp</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>165</td>
</tr>
<tr>
<td>15</td>
<td>560</td>
</tr>
<tr>
<td>20</td>
<td>1330</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Star</th>
<th>Clique</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>#ccp</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>2304</td>
</tr>
<tr>
<td>15</td>
<td>114688</td>
</tr>
<tr>
<td>20</td>
<td>4980736</td>
</tr>
</tbody>
</table>
Algorithm DPccp

for all \((R_i \in \mathcal{R})\) \(\text{BestPlan}(\{R_i\}) = R_i;\)
forall csg-cmp-pairs \((S_1, S_2), \quad S = S_1 \cup S_2\)
  ++InnerCounter;
  ++OnoLohmanCounter;
  \(p_1 = \text{BestPlan}(S_1);\)
  \(p_2 = \text{BestPlan}(S_2);\)
  CurrPlan = CreateJoinTree\((p_1, p_2)\);
  if \((\text{cost}(\text{BestPlan}(S)) > \text{cost}(\text{CurrPlan}))\)
    \(\text{BestPlan}(S) = \text{CurrPlan};\)
  CurrPlan = CreateJoinTree\((p_2, p_1)\);
  if \((\text{cost}(\text{BestPlan}(S)) > \text{cost}(\text{CurrPlan}))\)
    \(\text{BestPlan}(S) = \text{CurrPlan};\)
  CsgCmpPairCounter = 2 * OnoLohmanCounter;
return \(\text{BestPlan}(\{R_0, \ldots, R_{n-1}\});\)
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) \ 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, \( S \cup S' \), \( X \cup N \));
}
Example
<table>
<thead>
<tr>
<th>$S$</th>
<th>$X$</th>
<th>$N$</th>
<th>emit/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>{4}</td>
<td>{0, 1, 2, 3, 4}</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>{3}</td>
<td>{0, 1, 2, 3}</td>
<td>{4}</td>
<td></td>
</tr>
<tr>
<td>{2}</td>
<td>{0, 1, 2}</td>
<td>{3, 4}</td>
<td></td>
</tr>
<tr>
<td>{1}</td>
<td>{0, 1}</td>
<td>{4}</td>
<td></td>
</tr>
<tr>
<td>→ {1, 4}</td>
<td>{0, 1, 4}</td>
<td>{2, 3}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{3, 4}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2, 3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2, 4}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2, 3, 4}</td>
</tr>
</tbody>
</table>
Algorithm EnumerateCmp

EnumerateCmp

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

Precondition: nodes in \( V \) are ordered

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

\[
X = B_{\min(S_1)} \cup S_1;
N = N(S_1) \setminus X;
\]

for all \((v_i \in N \text{ by descending } i)\) {
\[
\text{emit } \{v_i\};
\text{EnumerateCsgRec}(G, \{v_i\}, X \cup B_i(N));
\]
}

where \(\min(S_1) := \min(\{i|v_j \in S_1\})\).
A DP-Based Plan Generator for Hypergraphs

outline

1. preliminaries
2. reorderability
3. conflict detection
4. enumeration
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 $A$, $\text{REL}(A)$ 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 $\text{left}(\circ)$ (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.
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 (Necessity of Hypergraphs)

possible join predicate: $R.a + S.b = S.c + T.d$
We will see later: conflict detectors introduce hypergraphs
Preliminaries (Neighborhood)

\[ \text{min}(S) = \{s| s \in S, \forall s' \in S \; s \neq s' \implies s \prec s' \} \]

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 \( \text{min}(v) \) to the neighborhood of \( S \). We thus define

\[ \text{min}(S) = S \setminus \text{min}(S) \]

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

$$\mathcal{N}(S, X) = \bigcup_{v \in E \downarrow(S, X)} \min(v)$$  \hspace{1cm} (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 \textit{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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reorderability (assoc)

assoc:

\[(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 \equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3)\]  \hspace{1cm} (2)
Reorderability (assoc)

<table>
<thead>
<tr>
<th>$\circ^a$</th>
<th>$\circ^b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\times$</td>
<td>+</td>
</tr>
<tr>
<td>$\circ$</td>
<td>+</td>
</tr>
<tr>
<td>$\times$</td>
<td>+</td>
</tr>
<tr>
<td>$\triangleright$</td>
<td>+</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>-</td>
</tr>
<tr>
<td>$\bowtie'$</td>
<td>-</td>
</tr>
</tbody>
</table>

(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)
Consider the following truth about the semijoin:

\[(e_1 \Join_1 e_2) \Join_3 e_3 \equiv (e_1 \Join_3 e_3) \Join_1 e_2.\]

This is neither 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:

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

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<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>
<td>+/-</td>
</tr>
<tr>
<td>□</td>
<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>
<td>+/-</td>
</tr>
<tr>
<td>▽</td>
<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>
<td>+/-</td>
</tr>
<tr>
<td>▲</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
</tbody>
</table>

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: \( \text{SES} \)

\[
\begin{align*}
\text{SES}(R) &= \{R\} \\
\text{SES}(T) &= \{T\} \\
\text{SES}(\circ_p) &= \bigcup_{R \in \mathcal{F}_T(p)} \text{SES}(R) \cap \text{REL}(\circ_p) \\
\text{SES}(\nabla^p; a_1:e_1,\ldots,a_n:e_n) &= \bigcup_{R \in \mathcal{F}_T(p) \cup \mathcal{F}_T(e_i)} \text{SES}(R) \cap \text{REL}(g_j)
\end{align*}
\]
assoc \((o^a, o^b)\)

\[
\begin{align*}
&\left((e_2 o_{12}^a e_1) o_{13}^b e_3\right) \xrightarrow{\text{comm \((o^b)\)}} (e_3 o_{13}^b (e_2 o_{12}^a e_1)) \\
&\left((e_1 o_{12}^a e_2) o_{13}^b e_3\right) \xrightarrow{\text{comm \((o^a)\)}} (e_3 o_{13}^b (e_1 o_{12}^a e_2)) \\
&\left((e_1 o_{13}^b e_3) o_{12}^a e_2\right) \xrightarrow{\text{comm \((o^a)\)}} (e_2 o_{12}^a (e_1 o_{13}^b e_3)) \\
&\left((e_3 o_{13}^b e_1) o_{12}^a e_2\right) \xrightarrow{\text{comm \((o^a)\)}} (e_2 o_{12}^a (e_3 o_{13}^b e_1))
\end{align*}
\]

\[
\begin{align*}
&\xrightarrow{\text{assoc \((o^b, o^a)\)}} \\
&\xrightarrow{\text{assoc \((o^a, o^b)\)}} \\
&\text{l-asscomm \((o^a, o^b)\)}
\end{align*}
\]

\[
\begin{align*}
&\text{r-asscomm \((o^a, o^b)\)}
\end{align*}
\]
Conflict Detector CD-A: $\text{TES: left conflict}$

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

\[ \text{TES}(\circ_b) \cup = \text{REL}(e_1) \]

\[ \neg \text{assoc}(\circ^a, \circ^b) \]

\[ \text{TES}(\circ^b) \cup = \text{REL}(e_2) \]

\[ \neg \text{l-asscom}(\circ^a, \circ^b) \]
Conflict Detector CD-A: $\text{TES}$: right conflict

\[ \langle e_3, b, e_1, a, e_2 \rangle \xrightarrow{\text{assoc}} \langle e_3, b, e_1, a, e_2 \rangle \]

\[ \langle e_3, b, e_1, a, e_2 \rangle \xrightarrow{\text{r-asscom}} \langle e_3, b, e_1, a, e_2 \rangle \]

\[ \neg \text{assoc}(b, a) \]

\[ \text{TES}(b) \cup e_2 = \text{REL}(e_2) \]

\[ \neg \text{r-asscom}(a, b) \]

\[ \text{TES}(b) \cup e_1 = \text{REL}(e_1) \]
Conflict Detector CD-A: Remarks

- correct
- not complete
Conflict Detector CD-A: applicability test

applicable(\(\circ, S_1, S_2\)) := L-\text{TES}(\circ) \subseteq S_1 \land R-\text{TES}(\circ) \subseteq S_2.

where

\[\begin{align*}
\text{L-\text{TES}}(\circ) & := \text{TES}(\circ) \cap \text{REL(left}(\circ)) \\
\text{R-\text{TES}}(\circ) & := \text{TES}(\circ) \cap \text{REL(right}(\circ))
\end{align*}\]
Query Hypergraph Construction

The nodes $V$ are the relations. For every operator $\circ$, we construct a hyperedge $(l, r)$ such that

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

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

▷ 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. \( \text{DPTable}[R_i] \leftarrow R_i \triangleright \) initial access paths
3. for all csg-cmp-pairs $(S_1, S_2)$ of $H$
4. for all $\circ p \in O$
   5. if \( \text{APPLICABLE}(S_1, S_2, \circ p) \)
      6. \( \text{BUILD PLANS}(S_1, S_2, \circ p) \)
   7. if $\circ p$ is commutative
      8. \( \text{BUILD PLANS}(S_2, S_1, \circ p) \)
9. return \( \text{DPTable}[R] \)
BUILDPLANS\((S_1, S_2, \circ_p)\)

1. \textit{OptimalCost} \leftarrow \infty
2. \textit{S} \leftarrow S_1 \cup S_2
3. \textit{T}_1 \leftarrow \text{DPTable}[S_1]
4. \textit{T}_2 \leftarrow \text{DPTable}[S_2]
5. \textbf{if} \text{DPTable}[S] \neq \text{NULL}
   \hspace{1em} \textit{OptimalCost} \leftarrow \text{COST}(\text{DPTable}[S])
6. \textbf{if} \text{COST}(T_1 \circ_p T_2) < \textit{OptimalCost}
   \hspace{1em} \textit{OptimalCost} \leftarrow \text{COST}(T_1 \circ_p T_2)
7. \textit{DPTable}[S] \leftarrow (T_1 \circ_p T_2)
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.
“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
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 $\text{DpTable}[S_1 \cup N] \neq \emptyset$

        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: \texttt{.EmitCsg}

\begin{align*}
\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 } < \\
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)) \\
\end{align*}

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

EnumerateCmpRec(S₁, S₂, X)
for each N ⊆ N(S₂, X): N ≠ ∅
    if DpTable[S₂ ∪ N] ≠ ∅ ∧ ∃(u, v) ∈ E : u ⊆ S₁ ∧ v ⊆ S₂ ∪ N
        EmitCsgCmp(S₁, S₂ ∪ N)
    X = X ∪ N(S₂, X)
for each N ⊆ N(S₂, X): N ≠ ∅
    EnumerateCmpRec(S₁, S₂ ∪ N, X)
The procedure $\text{EmitCsgCmp}(S_1, S_2)$ 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.
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

Observation:
- Pushing grouping down joins may result in much better plans.

Goal:
- extend DPhyp to handle push-down of grouping operators
Notation

- grouping operator $\Gamma$
- set of grouping attributes $G$
- vector of aggregate functions $F = (f_1, \ldots, f_k)$
- join operator $\circ \in \{\times, \text{IN}, \text{IN}, \triangleright, \triangleright, \times, \times\}$
- sets of join attributes $J_1$ from $e_1$, $J_2$ from $e_2$
- algebraic expressions $e_1$ and $e_2$
Aggregate Functions

We need some properties of aggregate functions:

1. splittability
2. decomposability
Splittability

Definition
An aggregation vector $F$ is splittable into $F_1$ and $F_2$ with respect to arbitrary expressions $e_1$ and $e_2$ if $F = F_1 \circ F_2$, $\mathcal{F}(F_1) \cap \mathcal{A}(e_2) = \emptyset$ and $\mathcal{F}(F_2) \cap \mathcal{A}(e_1) = \emptyset$, with

- $\mathcal{F}(e)$, the set of attributes referenced by some expression $e$ and
- $\mathcal{A}(e)$, the set of attributes provided by some expression $e$.

Example

$F = (\text{sum}(e_1.a), \text{count}(e_2.b))$
$F_1 = (\text{sum}(e_1.a))$
$F_2 = (\text{count}(e_2.b))$
$F = F_1 \circ F_2 = (\text{sum}(e_1.a), \text{count}(e_2.b))$
Decomposability

Definition
An aggregate function $f$ is decomposable if there exist aggregate functions $f^{pre}$ and $f^{post}$ such that $f(Z) = f^{post}(f^{pre}(X), f^{pre}(Y))$, for bags of values $X$, $Y$ and $Z$ where $Z = X \cup Y$.

Example
\[
\Gamma_{G; \text{sum}(s)}(\Gamma_{G \cup G'; s; \text{sum}(a) R})
\rightarrow \text{sum}(X \cup Y) = \text{sum}(\text{sum}(X), \text{sum}(Y))
\]
Decomposable functions: $\text{sum}$, $\text{count}$, $\text{avg}$, $\text{min}$, $\text{max}$
Duplicate Agnostic/Sensitive

An aggregate function $f$ is called *duplicate agnostic* if its result does *not* depend on whether there are duplicates in its argument or not. Otherwise, it is called *duplicate sensitive*. For SQL aggregate functions, we have that

- $\text{min}$, $\text{max}$, $\text{sum}($distinct$)$, $\text{count}($distinct$)$, $\text{avg}($distinct$)$ are duplicate agnostic and
- $\text{sum}$, $\text{count}$, $\text{avg}$ are duplicate sensitive.

We need to be careful in case of duplicate sensitive aggregate functions:
Duplicate Sensitive Aggregate Functions

We encapsulate this by an operator prime (′) as follows. Let $F = (b_1 : agg_1(a_1), \ldots, b_m : agg_m(a_m))$ be an aggregation vector. Further, let $c$ be some other attribute typically holding the result of a count($\ast$).

Then, we define $F \otimes c$ as

$$F \otimes c := (b_1 : agg'(e_1), \ldots, b_m : agg'(e_m))$$

with

$$agg'(e_i) = \begin{cases} 
agg_i(e_i) & \text{if } agg_i \text{ is duplicate agnostic,} \\
agg_i(e_i \ast c) & \text{if } agg_i \text{ is sum,} \\
\text{sum}(c) & \text{if } agg_i(e_i) = \text{count}(\ast),
\end{cases}$$

and if $agg_i(e_i)$ is count($e_i$), then

$$agg'_i(e_i) := \text{sum}(e_i = \text{NULL} ? 0 : c).$$
Transformations

Known transformation to push group-by past regular inner join:

- Eager/Lazy Groupby-Count
- Eager/Lazy Group-By
- Eager/Lazy Count
- Double Eager/Lazy
- Eager/Lazy Split

\^\textsuperscript{1}W. Yan and P.-A. Larson, “Eager Aggregation and Lazy Aggregation”, VLDB, 1995
Eager/Lazy Groupby-Count

- $G_i$: Grouping attributes from $e_i$
- $G_i^+ : G_i \cup J_i$
- Split $F$ into $F_1$ and $F_2$
- Decompose $F_1$ into $F_1^{pre}$ and $F_1^{post}$
Eager/Lazy Group-By

\[ \Gamma_{G;F_1^{post}} \]

\[ \circ p \]

\[ \Gamma_{G_1^+;F_1^{pre}} \]

\[ e_2 \]

\[ e_1 \]

- If \( F_2 = () \)
Eager/Lazy Count

\[ \Gamma_{G_i}(F_2 \otimes c_1) \circ p \]

\[ \Gamma_{G_1^+; (c_1: count(*))} e_2 \]

\[ e_1 \]

- If \( F_1 = () \)
Double Eager/Lazy

\[ \Gamma G_i(F_{post} \otimes c_2) \]
\[ \circ p \]
\[ \Gamma G_1^{+}; F_1^{pre} \]
\[ \Gamma G_2^{+}; (c_2: \text{count}(\ast)) \]

\[ e_1 \]
\[ e_2 \]

- If \( F_2 = () \)
Eager/Lazy Split

\[ \Gamma \cdot (F_{post} \otimes c_2) \cdot (F_{post} \otimes c_1) \]

\[ \circ p \]

\[ \Gamma \cdot G; F_{pre} \circ (c_1: \text{count}(\ast)) \]

\[ \Gamma \cdot G; F_{pre} \circ (c_2: \text{count}(\ast)) \]

\[ e_1 \]

\[ e_2 \]
Important Observations

1. In general, grouping cannot be pushed down outer joins.
2. If outerjoins with defaults other than padding with NULL values are used, grouping can be pushed down any join operator.
Eager/Lazy Groupby-Count with Full Outer Join

\[ \Gamma_{G;F}(e_1 \Join_{J_1,J_2} e_2) \]

\[ \equiv \]

\[ \Gamma_{G_i:(F_2 \otimes c_1) \circ F_{1}^{\text{post}}(\Gamma_{G_1^+;F_{1}^{\text{pre}}(c_1:\text{count}(\star))) (e_1) \Join_{J_1,J_2}^{F_{1}^{\text{pre}}(\{\perp\}) \circ (c_1:1);\neg} e_2) \]

\[ e_1 \Join_{D_1;D_2}^{D_{1};D_{2} \circ } e_2: \text{Outer join with default values } D_1, D_2 \]

\[ F_{1}^{\text{pre}}(\perp): \text{Application of } F_{1}^{\text{pre}}(\perp) \text{ to a bag of null-tuples} \]
Example (1/7)

\[
\begin{array}{c|c|c}
  e_1 & e_2 \\
  \hline
  g_1 & j_1 & a_1 & g_2 & j_2 & a_2 \\
  1 & 1 & 2 & 1 & 1 & 2 \\
  1 & 2 & 4 & 1 & 1 & 4 \\
  1 & 2 & 8 & 1 & 2 & 8 \\
  1 & 3 & 7 & - & - & - \\
 \end{array}
\]

\[
e_3 := e_1 \times_{j_1 = j_2} e_2
\]

\[
\begin{array}{c|c|c|c|c|c}
  g_1 & j_1 & a_1 & g_2 & j_2 & a_2 \\
  1 & 1 & 2 & 1 & 1 & 2 \\
  1 & 1 & 2 & 1 & 1 & 4 \\
  1 & 2 & 4 & 1 & 2 & 8 \\
  1 & 2 & 8 & 1 & 2 & 8 \\
  1 & 3 & 7 & - & - & - \\
  - & - & - & 1 & 4 & 9 \\
\end{array}
\]
Example (2/7)

$$e_3 := e_1 \otimes_{j_1=2} e_2$$

<table>
<thead>
<tr>
<th>$g_1$</th>
<th>$j_1$</th>
<th>$a_1$</th>
<th>$g_2$</th>
<th>$j_2$</th>
<th>$a_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

$$e_4 := \Gamma_{g_1,g_2} F(e_3)$$

<table>
<thead>
<tr>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$s_1$</th>
<th>$s_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>9</td>
</tr>
</tbody>
</table>

$$F = s_1 : \text{sum}(a_1), s_2 : \text{sum}(a_2)$$
Eager/Lazy Groupby-Count with Full Outer Join

\[
\Gamma_{G;F}(\ e_1 \bowtie \ J_1,J_2 \ e_2) \\
\equiv \Gamma_{G_1;F_1}(\Gamma_{G_2;F_2}^\preceq(F_1^\preceq(c_1:count(\ast))(e_1 \bowtie J_1,J_2 \ e_2)) \\
= \Gamma_{G_1;F_1}^\preceq(F_1^\preceq(\perp)^\circ(c_1:1);- \ e_2)
\]
Example (3/7)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$j_1$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

$e_5 := \Gamma_{g_1,j_1;F_{pre}^1(c_1:count(\star))}(e_1)$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$j_1$</td>
<td>$c_1$</td>
<td>$s_1'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

$F = s_1 : \text{sum}(a_1), s_2 : \text{sum}(a_2)$

*Recall: sum$(X \cup Y) = \text{sum}(\text{sum}(X), \text{sum}(Y))$*

- Split $F$:
  - $F_1 = s_1 : \text{sum}(a_1)$
  - $F_2 = s_2 : \text{sum}(a_2)$

- Decompose $F_1$:
  - $F_{pre}^1 = s_1' : \text{sum}(a_1)$
  - $F_{post}^1 = s_1 : \text{sum}(s_1')$
Example (4/7)

What if we use a “normal” full outer join?

\[ e_5 := \Gamma_{g_1, j_1 : F_{pre}^1(c_1 : \text{count}(*))}(e_1) \]

<table>
<thead>
<tr>
<th>( g_1 )</th>
<th>( j_1 )</th>
<th>( c_1 )</th>
<th>( s'_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( g_2 )</th>
<th>( j_2 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ e_6 := e_5 \bowtie_{j_1 = j_2} e_2 \]

<table>
<thead>
<tr>
<th>( g_1 )</th>
<th>( j_1 )</th>
<th>( c_1 )</th>
<th>( s'_1 )</th>
<th>( g_2 )</th>
<th>( j_2 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>
Example (5/7)

\[ e_6 := e_5 \land_{j_1 = j_2} e_2 \]

<table>
<thead>
<tr>
<th>( g_1 )</th>
<th>( j_1 )</th>
<th>( c_1 )</th>
<th>( s'_1 )</th>
<th>( g_2 )</th>
<th>( j_2 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>( \bigcirc )</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

\[ e_7 := \Gamma_{g_1,g_2}:F_X(e_5) \]

\[ e_4 := \Gamma_{g_1,g_2}:F(e_3) \]

\[ F = s_1 : \text{sum}(a_1), s_2 : \text{sum}(a_2) \]

\[ F_X = (F_2 \otimes c_1) \circ F_1^{\text{post}} \]

\[ = s_2 : \text{sum}(c_1 \ast a_2), s_1 : \text{sum}(s'_1) \]
Using the full outer join with default values:

\[ e_5 := \Gamma_{g_1, j_1 : F_1^{\text{pre}} \circ (c_1 : \text{count}(\ast))} (e_1) \]

<table>
<thead>
<tr>
<th>( g_1 )</th>
<th>( j_1 )</th>
<th>( c_1 )</th>
<th>( s'_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

\[ e'_6 := e_5 \bigcirc_{j_1 = j_2} F_1^{\text{pre}} (\{\bot\}) \circ (c_1 : 1) ; - e_2 \]

<table>
<thead>
<tr>
<th>( g_1 )</th>
<th>( j_1 )</th>
<th>( c_1 )</th>
<th>( s'_1 )</th>
<th>( g_2 )</th>
<th>( j_2 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ e_2 \]

<table>
<thead>
<tr>
<th>( g_2 )</th>
<th>( j_2 )</th>
<th>( a_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>
$e'_6 := e^\pre(F_1)_{j_1=j_2} \circ (c_1:1) ; - e_2$

\begin{tabular}{cccccccc}
  $g_1$ & $j_1$ & $c_1$ & $s'_1$ & $g_2$ & $j_2$ & $a_2$ \\
  1 & 1 & 1 & 2 & 1 & 1 & 2 \\
  1 & 1 & 1 & 2 & 1 & 1 & 4 \\
  1 & 2 & 2 & 12 & 1 & 2 & 8 \\
  1 & 3 & 1 & 7 & - & - & - \\
  - & - & 1 & - & 1 & 4 & 9 \\
\end{tabular}

$e'_7 := \Gamma_{g_1,g_2;F_X}(e'_6)$

\begin{tabular}{cccc}
  $g_1$ & $g_2$ & $s_1$ & $s_2$ \\
  1 & 1 & 16 & 22 \\
  1 & - & 7 & - \\
  - & 1 & - & 9 \\
\end{tabular}

$e_4 := \Gamma_{g_1,g_2;F}(e_3)$

\begin{tabular}{cccc}
  $g_1$ & $g_2$ & $s_1$ & $s_2$ \\
  1 & 1 & 16 & 22 \\
  1 & - & 7 & - \\
  - & 1 & - & 9 \\
\end{tabular}

$F = s_1 : sum(a_1), s_2 : sum(a_2)$

$F_X = (F_2 \otimes c_1) \circ F_1^{\text{post}}$

$= s_2 : sum(c_1 \ast a_2), s_1 : sum(s'_1)$
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

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.  **if** \( \text{APPLICABLE}(S_1, S_2, \circ p) \)
6.  \( \text{BUILDTREE}(S_1, S_2, \circ p) \)
7.  **if** \( \circ p \) is commutative
8.  \( \text{BUILDTREE}(S_2, S_1, \circ p) \)
9. **return** \( \text{DPTable}[R] \)
Algorithm: BuildTree

**BUILD TREE**($S_1, S_2, \circ_p$)

1. $S \leftarrow S_1 \cup S_2$
2. for each $T_1 \in DPTable[S_1]$
3.     for each $T_2 \in DPTable[S_2]$
4.         for each $T \in OPTHREE(T_1, T_2, \circ_p)$
5.             if $S == R$
6.                 INSERTTOPLEVELPLAN($S, T$)
7.             else
8.                 INSERTNONTOPLEVELPLAN($S, T$)
**OpTrees**

\[
\text{OP\textsc{TREES}}(T_1, T_2, \circ_p)
\]

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

if \( S == R \land \text{NEEDSGROUPING}(G, \text{NewTree}) \)

\[
\text{NewTree} \leftarrow (\Gamma_G(\text{NewTree}))
\]

\[\text{Trees}.insert(\text{NewTree})\]

\( \text{NewTree} \leftarrow \Gamma_{G_1}(T_1) \circ_p \Gamma_{G_2}(T_2) \)

if \( \text{VALID}(\text{NewTree}) \)

\( \land \text{NEEDSGROUPING}(G^+_1, \text{NewTree}) \land \text{NEEDSGROUPING}(G^+_2, \text{NewTree}) \)

if \( S == R \land \text{NEEDSGROUPING}(G, \text{NewTree}) \)

\( \text{NewTree} \leftarrow (\Gamma_G(\text{NewTree})) \)

\[\text{Trees}.insert(\text{NewTree})\]

return \( \text{Trees} \)
Cost functions and cardinality estimation are not perfect. Thus, some error metrics is needed. Different metrics lead to different best approximations. Thus, the error metrics should be well-suited for query optimization.
Q-Paranorm and Q-Error

For some number $x > 0$ define

$$||x||_Q := \max(x, 1/x)$$

For two numbers $x > 0$ and $y > 0$ define

$$q\text{-error}(x, y) := ||(||_Qx/y)||$$
Cost Functions

Let $P$ be a plan, then

- $\mathcal{M}P > 0$ denotes the measured costs
- $\mathcal{C}P > 0$ denotes the estimated costs via some cost function $\mathcal{C}$.

- Let $\mathcal{P}$ be a set of (equivalent) plans.
- Denote by $P_{\text{opt}}$ the plan which minimizes $\mathcal{M}P$.
- Denote by $P_{\text{best}}$ the plan which minimizes $\mathcal{C}P$.
- For all plans q-error($\mathcal{C}P, \mathcal{M}P) \leq q$.

Then

$$\|\mathcal{M}(e_{\text{best}})/\mathcal{M}(e_{\text{opt}})\|_Q \leq q^2$$
Cardinality Estimation

Under certain circumstances (cost function sort-merge join or hash-join):

- Let $P$ be the optimal plan produced by precise cardinality estimates.
- Let $\hat{P}$ be the optimal plan produced using cardinality estimates.
- All cardinality estimates have a maximum q-error of $q$.

Then

$$||\hat{P}/P||_Q \leq q^4$$
Synopsis:

- must be small
- must be fast to build
- must be fast to evaluate during plan generation
- must be precise

Achieving all this at the same time is rather difficult. Thus, a weaker notion than q-error is used:
$\theta, q$-acceptability

$\theta, q$-acceptability. Let $f \geq 0$ be a number and $\hat{f} \geq 0$ be an estimate for $f$. Let $q \geq 1$ and $\theta \geq 1$ be numbers. We say that $\hat{f}$ is $\theta, q$-acceptable if

$$f \leq \theta \land \hat{f} \leq \theta \lor \|\hat{f}/f\|_Q \leq q.$$ 

We define that a binary estimation $\hat{f}^+$ for $f^+$ is $\theta, q$-acceptable on some interval $[l, u]$, if for all $l \leq c_1 \leq c_2 \leq u$ the estimate $\hat{f}^+(c_1, c_2)$ is $\theta, q$-acceptable.

Remark: If $\theta$ is chosen carefully, the plan generator does not make incorrect choices.
Testing $\theta, q$-acceptability

Directly testing $\theta, q$-acceptability for a given bucket for a continuous domain is impossible since it would involve testing $\theta, q$-acceptability of $\hat{f}^+(c_1, c_2)$ for all $c_1, c_2$ within the bucket.

Data distribution $(x_i, f_i)$ for $i = 1, \ldots, n$:

- $x_i$ domain value
- $f_i$ its frequency
Theorem

Assume $\hat{f}^+$ is monotonic. If for all $i, j$ such that $x_i$ and $x_j$ are in the bucket, we have that

$$\hat{f}^+(x_{i-1}, x_{j+1}) \leq \theta \land f^+(x_{i-1}, x_{j+1}) \leq \theta$$

or

$$\left\| \frac{\hat{f}^+(x_i, x_j)}{f^+(x_i, x_j)} \right\|_Q \leq q \land \left\| \frac{\hat{f}^+(x_{i-1}, x_{j+1})}{f^+(x_i, x_j)} \right\|_Q \leq q,$$

then the estimation function is $\theta, q$-acceptable for the bucket. $\Box$
Subquadratic Test

There exists a faster test for monotonic and additive estimation functions $\hat{f}^+$. 
Cheap Pretest for Dense Buckets

Theorem

If the bucket is dense and

\[ \text{itemsep=2pt the cumulated bucket frequency is less than or} \]
\[ \text{equal to } \theta \text{ or} \]
\[ \text{itemsep=2pt} \max_i f_i / \min_i f_i \leq q^2, \]

then there exists an estimation function \( \hat{f}^+ \) that is \( \theta, q \)-acceptable. □
Testing $\theta$, $q$-acceptability

\[
is\text{Theta}Q\text{Acc}(l, u)
\]
if the pretest succeeds
   then return true
if $\text{MaxSize} \leq (l - u)$
   then return false
return the result of the subquadratic test

where $\text{MaxSize}$ is a tuning parameter.
Building Histograms

- Given the test of $\theta$, $q$-acceptability it is now easy to devise several kinds of histograms.
- Note: Only for dense buckets we have a cheap pretest.
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