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: θ must be '$=$'
  (equi-joins only)

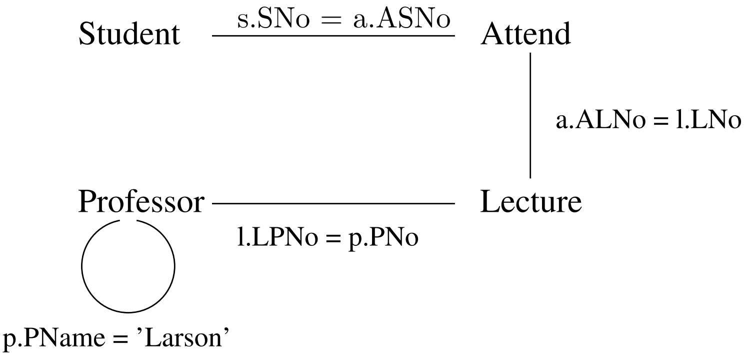
We join relations $R_1, \ldots, R_n$ where R_i can be

- a base relation
- a base relation to which a selection has been applied
- a more complex building block or access path

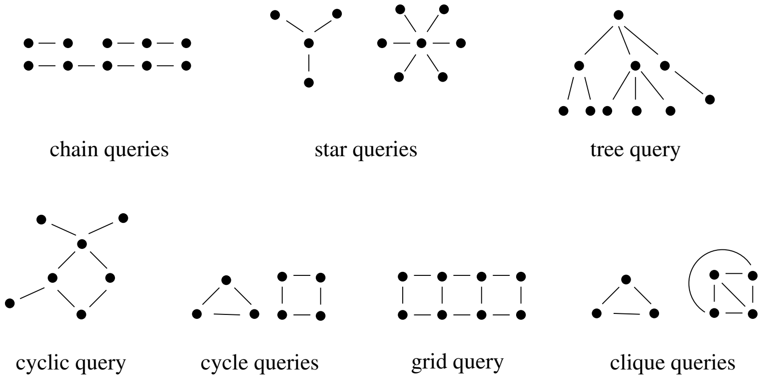
Query Graph

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



Shapes of Query Graphs



Join Trees

are binary trees

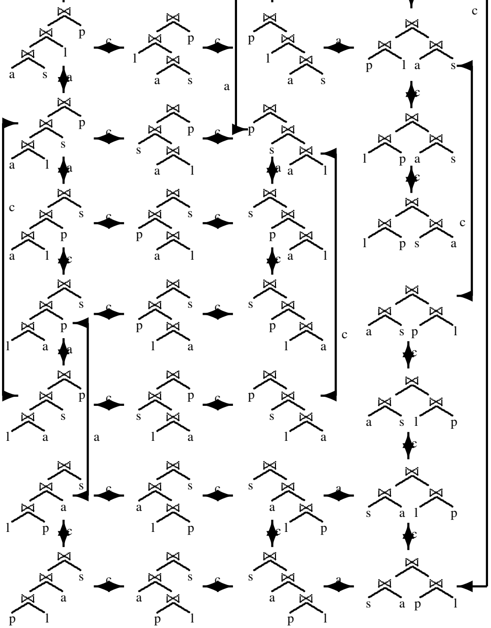
- with relation names attached to leaf nodes and
- join operators as inner nodes.

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

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i| * |R_j|}$$

Calculate:

- result cardinality:

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

Result Cardinalities of Join Trees

Consider a join tree $T = T_1 \bowtie T_2$.
Then, $|T|$ can be calculated as follows:

- If T is a leaf R_i, then $|T| = |R_i|$.
- Otherwise,

$$|T| = (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) \; |T_1| \; |T_2|.$$

This formula assumes independence.

Example

The table

$$
\begin{aligned}
|R_1| &= 10 \\
|R_2| &= 100 \\
|R_3| &= 1000 \\
f_{1,2} &= 0.1 \\
f_{2,3} &= 0.2
\end{aligned}
$$

implicitly defines the query graph $R_1 - - R_2 - - R_3$.
(We assume $f_{i,j} = 1$ for all i, j for which $f_{i,j}$ is not explicitly given.)

The cost function C_{out}

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

Other cost functions

For single join operators:

$$
\begin{aligned}
C_{\mathrm{nlj}}(e_1 \bowtie_p e_2) &= |e_1||e_2| \\
C_{\mathrm{hj}}(e_1 \bowtie_p e_2) &= h|e_1| \\
C_{\mathrm{smj}}(e_1 \bowtie_p e_2) &= |e_1| log(|e_1|) + |e_2| log(|e_2|)
\end{aligned}
$$

For sequences of join operators (relations):

$$
\begin{aligned}
C_{\mathrm{hj}}(s) &= \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}| \\
C_{\mathrm{smj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| \log(|s_1, \ldots, s_{i-1}|) + \sum_{i=2}^{n} |s_i| \log(|s_i|) \\
C_{\mathrm{nlj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| * s_i
\end{aligned}
$$

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

	C_{out}	C_{nlj}	C_{hj}	C_{smj}
$R_1 \bowtie R_2$	100	1000	12	697.61
$R_2 \bowtie R_3$	20000	100000	120	10630.26
$R_1 \times R_3$	10000	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	132	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	24120	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	22000	143542.00

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

Join Tree	C_{out}
$R_1 \bowtie R_2$	200
$R_2 \times R_3$	4
$R_1 \bowtie R_3$	200
$(R_1 \bowtie R_2) \bowtie R_3$	240
$(R_2 \times R_3) \bowtie R_1$	44
$(R_1 \bowtie R_3) \bowtie R_2$	240

Plan with cross product is best.

Yet Another Example

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

Join Tree	C_{out}
$R_1 \bowtie R_2$	2
$R_2 \bowtie R_3$	200
$R_3 \bowtie R_4$	2
$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$	24
$((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \times R_2) \bowtie (R_3 \bowtie R_4)$	6

Bushy tree better than any left-deep tree

Properties of Cost Functions: Symmetry and ASI

A cost function C_{impl} is called *symmetric*, if $C_{\text{impl}}(R_1 \bowtie^{\text{impl}} R_2) = C_{\text{impl}}(R_2 \bowtie^{\text{impl}} R_1)$ 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)

	ASI	$\neg$ ASI
symmetric	C_{out}	C_{smj}
$\neg$ symmetric	C_{hj}	(listen)

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

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 $\mathcal{C}(n-1)$ where $\mathcal{C}(n)$ is defined by the recurrence

$$\mathcal{C}(n) = \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1)$$

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

$$\mathcal{C}(n) = \frac{1}{n+1}\binom{2n}{n}.$$

The Catalan Numbers grow in the order of $\Theta(4^n/n^{3/2})$.

Number of Bushy Trees with Cross Products

$$
\begin{aligned}
n!\,\mathcal{C}(n-1) &= n!\,\frac{1}{n}\binom{2(n-1)}{n-1} \\
&= n!\,\frac{1}{n}\,\frac{(2n-2)!}{(n-1)!\,((2n-2)-(n-1))!} \\
&= \frac{(2n-2)!}{(n-1)!}
\end{aligned}
$$

Chain Queries, Left-Deep Join Trees, No Cross Product

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

Recurrence

Now we can compute the $f(n)$ as

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

for $n > 1$.
Solving the recurrence gives us

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

(Exercise)

Chain Queries, Bushy Join Trees, No Cross Product

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

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

(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: $2 * (n-1)!$
- right-deep trees: $2 * (n-1)!$
- zig-zag trees: $2 * (n-1)! * 2^{n-2} = 2^{n-1} * (n-1)!$

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				
	Chain Query			Star Query	
	Left-Deep	Zig-Zag	Bushy	Left-Deep	Zig-Zag/Bushy
n	2^{n-1}	2^{2n-3}	$2^{n-1}\mathcal{C}(n-1)$	$2*(n-1)!$	$2^{n-1}(n-1)!$
1	1	1	1	1	1
2	2	2	2	2	2
3	4	8	8	4	8
4	8	32	40	12	48
5	16	128	224	48	384
6	32	512	1344	240	3840
7	64	2048	8448	1440	46080
8	128	8192	54912	10080	645120
9	256	32768	366080	80640	10321920
10	512	131072	2489344	725760	185794560

Numbers

	With Cross Products/Clique		
	Left-Deep	Zig-Zag	Bushy
n	$n!$	$2^{n-2} * n!$	$n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	928972800	17643225600

Complexity

Query Graph	Join Tree	$\times$ s	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 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

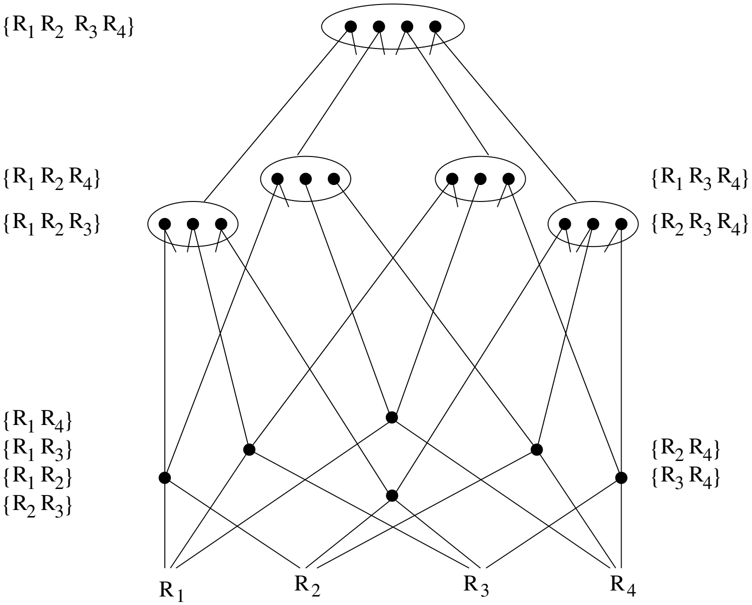
```
CreateJoinTree(T₁, T₂)
```

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 = $T_1 \bowtie^{impl} T_2$
    if (BestTree == NULL || cost(BestTree) > cost(Tree))
      BestTree = Tree;
  if(!LeftDeepOnly)
    Tree = $T_2 \bowtie^{impl} T_1$
    if (BestTree == NULL || cost(BestTree) > cost(Tree))
      BestTree = Tree;
}
return BestTree;

Search space with sharing under Optimality Principle



```
DP-Linear-1({R_1, ..., 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 ⊆ {R_1, ..., R_n}, |S| = i do {
    for all R_i ∈ {R_1, ..., R_n}, R_i ∉ S do {
      if (NoCrossProducts && !connected({R_i}, S)) { continue; }
      CurrTree = CreateJoinTree(BestTree(S),R_i);
      S' = S ∪ {R_i};
      if (BestTree(S') == NULL
            || cost(BestTree(S')) > cost(CurrTree)) {
         BestTree(S') = CurrTree;
      }
    }
  }
}
return BestTree({R_1, ..., 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

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

```
DP-Linear-2({R_1,...,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, ..., 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 ⊂ S do
    S_2 = S \ 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;
do {
  /* do something with subset S_1 */
  S_1 = S & (S_1 - S);
} **while** (S_1 != 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

	without cross products			with cross products	
	chain		star	any query graph	
	linear	bushy	linear	linear	bushy
n	$(n-1)^2$	$(n^3-n)/6$	$(n-1)2^{n-2}$	$n2^{n-1}-n(n+1)/2$	$(3^n-2^{n+1}+1)/2$
2	1	1	1	1	1
3	4	4	4	6	6
4	9	10	12	22	25
5	16	20	32	65	90
6	25	35	80	171	301
7	36	56	192	420	966
8	49	84	448	988	3025
9	64	120	1024	2259	9330
10	81	165	2304	5065	28501

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 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 Ri ∈ R BestPlan({Ri}) = Ri;
for all 1 < s ≤ n ascending // size of plan
for all 1 ≤ s1 < s // size of left subplan
  s2 = s − s1; // size of right subplan
  for all p1 = BestPlan(S1 ⊂ R : |S1| = s1)
      p2 = BestPlan(S2 ⊂ R : |S2| = s2)
    ++InnerCounter;
    if (∅ ≠ S1 ∩ S2) continue;
    if not (S1 connected to S2) continue;
    ++CsgCmpPairCounter;
    CurrPlan = CreateJoinTree(p1,p2);
    if (cost(BestPlan(S1 ∪ S2)) > cost(CurrPlan)) BestPlan
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R0,...,Rn−1});
```

Analysis: DPsize

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

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

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

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

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

DPsub

for all $R_i \in R$ `BestPlan({`R_i`}) = `R_i`;`
for $1 \leq i < 2^n - 1$ **ascending**
  $S = \{R_j \in R | (\lfloor i/2^j \rfloor \mathit{mod}\, 2) = 1\}$
  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

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

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;
forall 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

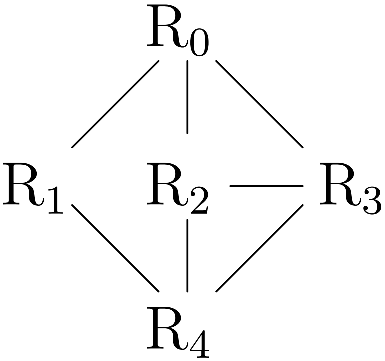
`EnumerateCsg`
Input: a connected query graph $G = (V, E)$
Precondition: nodes in V are numbered according to a breadth-first search
Output: emits all subsets of V inducing a connected subgraph of G

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

Example



S	X	N	emit$/S$
$\{4\}$	$\{0,1,2,3,4\}$	$\emptyset$	
$\{3\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{3,4\}$
$\{2\}$	$\{0,1,2\}$	$\{3,4\}$	
			$\{2,3\}$
			$\{2,4\}$
			$\{2,3,4\}$
$\{1\}$	$\{0,1\}$	$\{4\}$	
			$\{1,4\}$
$\rightarrow \{1,4\}$	$\{0,1,4\}$	$\{2,3\}$	
			$\{1,2,4\}$
			$\{1,3,4\}$
			$\{1,2,3,4\}$

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

$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \boldsymbol{N}(S_1) \setminus X$;
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. preliminaries
2. reorderability
3. conflict detection
4. enumeration

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 $\mathrm{SES}(\circ_p) = \{R, S, T\}$.

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 $\texttt{REL}(\text{left}(\circ)) \cap \mathcal{F}_\mathsf{T}(p) = \emptyset \;\vee\; \texttt{REL}(\text{right}(\circ)) \cap \mathcal{F}_\mathsf{T}(p) = \emptyset$ 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 (Necessity of Hypergraphs)

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

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:

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

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)

$\circ$	
$\times$	+
$\bowtie$	+
$\ltimes$	-
$\triangleright$	-
$⟕$	-
$⟗$	+
$\bowtie^{\Gamma}$	-

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)

$\circ^a$	$\circ^b$						
	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$⟕$	$⟗$	$\bowtie\!\Gamma$
$\times$	+	+	+	+	+	-	+
$\bowtie$	+	+	+	+	+	-	+
$\ltimes$	-	-	-	-	-	-	-
$\triangleright$	-	-	-	-	-	-	-
$⟕$	-	-	-	-	$+^1$	-	-
$⟗$	-	-	-	-	$+^1$	$+^2$	-
$\bowtie\!\Gamma$	-	-	-	-	-	-	-

(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 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:

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

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)

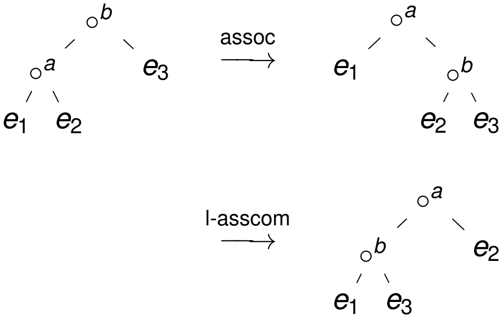
Conflict Detector CD-A: $\texttt{SES}$

$$
\begin{aligned}
\texttt{SES}(R) &= \{R\} \\
\texttt{SES}(T) &= \{T\} \\
\texttt{SES}(\circ_p) &= \bigcup_{R \in \mathcal{F}_{\mathrm{T}}(p)} \texttt{SES}(R) \cap \texttt{REL}(\circ_p) \\
\texttt{SES}(\bowtie\!\!\ulcorner_{p;a_1:e_1,\ldots,a_n:e_n}) &= \bigcup_{R \in \mathcal{F}_{\mathrm{T}}(p) \cup \mathcal{F}_{\mathrm{T}}(e_i)} \texttt{SES}(R) \cap \texttt{REL}(gj)
\end{aligned}
$$

$$
\begin{tikzcd}[column sep=huge, row sep=large]
(e_2\ o^a_{12}\ e_1)\ o^b_{13}\ e_3 \arrow[r, dash, "\text{comm }(o^b)"] \arrow[d, dash, "\text{comm }(o^a)"] \arrow[dr, dash, bend left=60, "\text{assoc }(o^a,\ o^b)"] & e_3\ o^b_{13}\ (e_2\ o^a_{12}\ e_1) \arrow[d, dash, "\text{comm }(o^a)"'] \arrow[ddd, dash, bend left=40, "\text{r-asscomm }(o^a,\ o^b)"'] \\
(e_1\ o^a_{12}\ e_2)\ o^b_{13}\ e_3 \arrow[r, dash, "\text{comm }(o^b)"] \arrow[d, dash, "\text{l-asscomm }(o^a,\ o^b)"] & e_3\ o^b_{13}\ (e_1\ o^a_{12}\ e_2) \\
(e_1\ o^b_{13}\ e_3)\ o^a_{12}\ e_2 \arrow[r, dash, "\text{comm }(o^a)"] \arrow[d, dash, "\text{comm }(o^b)"] & e_2\ o^a_{12}\ (e_1\ o^b_{13}\ e_3) \arrow[d, dash, "\text{comm }(o^b)"'] \\
(e_3\ o^b_{13}\ e_1)\ o^a_{12}\ e_2 \arrow[r, dash, "\text{comm }(o^a)"] \arrow[uur, dash, bend right=60, "\text{assoc }(o^b,\ o^a)"'] & e_2\ o^a_{12}\ (e_3\ o^b_{13}\ e_1)
\end{tikzcd}
$$

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

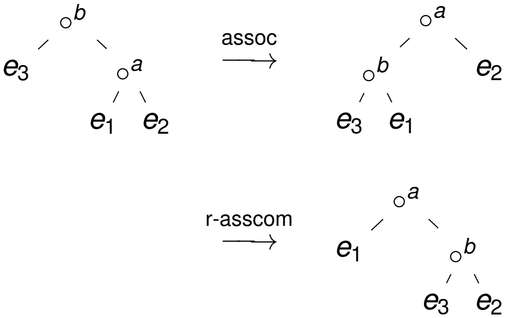
initially: $\mathtt{TES}(\circ_p) := \mathtt{SES}(\circ_p)$



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

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

Conflict Detector CD-A: TES: right conflict



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

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

Conflict Detector CD-A: Remarks

- correct
- not complete

Conflict Detector CD-A: applicability test

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

where

$$\begin{aligned} \texttt{L-TES}(\circ) &:= \texttt{TES}(\circ) \cap \texttt{REL}(\text{left}(\circ)) \\ \texttt{R-TES}(\circ) &:= \texttt{TES}(\circ) \cap \texttt{REL}(\text{right}(\circ)) \end{aligned}$$

Query Hypergraph Construction

The nodes V are the relations.
For every operator $\circ$, we construct a hyperedge (l, r) such that
$r = \mathtt{TES}(\circ) \cap \mathtt{REL}(\text{right}(\circ)) = \mathtt{R\text{-}TES}(\circ)$ and
$l = \mathtt{TES}(\circ) \setminus r = \mathtt{L\text{-}TES}(\circ)$.

DP-PlanGen

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

BUILDPLANS$(S_1, S_2, \circ_p)$

$\mathit{OptimalCost} \leftarrow \infty$
$S \leftarrow S_1 \cup S_2$
$T_1 \leftarrow \mathit{DPTable}[S_1]$
$T_2 \leftarrow \mathit{DPTable}[S_2]$
if $\mathit{DPTable}[S] \neq \mathit{NULL}$
    $\mathit{OptimalCost} \leftarrow \mathrm{COST}(\mathit{DPTable}[S])$
if $\mathrm{COST}(T_1 \circ_p T_2) < \mathit{OptimalCost}$
    $\mathit{OptimalCost} \leftarrow \mathrm{COST}(T_1 \circ_p T_2)$
    $\mathit{DPTable}[S] \leftarrow (T_1 \circ_p T_2)$

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.

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 ⊆ 𝒩(S_1, X): N ≠ ∅
    if DpTable[S_1 ∪ N] ≠ ∅
        EmitCsg(S_1 ∪ N)
for each N ⊆ 𝒩(S_1, X): N ≠ ∅
    EnumerateCsgRec(S_1 ∪ N, X ∪ 𝒩(S_1, X))
```

Csg-Cmp-Enumeration: `EmitCsg`

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

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

Csg-Cmp-Enumeration: `EnumerateCmpRec`

```
EnumerateCmpRec(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)
```

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

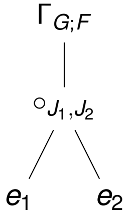
Observation:

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

Goal:

- extend DPhyp to handle push-down of grouping operators

Notation



- grouping operator Γ
- set of grouping attributes G
- vector of aggregate functions $F = (f_1, \ldots, f_k)$
- join operator $\circ \in \{\bowtie, ⟗, ⟕, \triangleright, \ltimes, \bowtie\!\!\ulcorner\}$
- 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 expresseion 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

$\Gamma_{G;\mathrm{sum}(s)}(\Gamma_{G \cup G';s:\mathrm{sum}(a)} R)$

$\to \mathrm{sum}(X \cup Y) = \mathrm{sum}(\mathrm{sum}(X), \mathrm{sum}(Y))$

Decomposable functions: *sum*, *count*, *avg*, *min*, *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

- min, max, sum(distinct), count(distinct), avg(distinct) are duplicate agnostic and
- sum, count, 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($*$).
Then, we define $F \otimes c$ as

$$F \otimes c := (b_1 : \underset{1}{\text{agg}}'(e_1), \ldots, b_m : \underset{m}{\text{agg}}'(e_m))$$

with

$$\underset{i}{\text{agg}}'(e_i) = \begin{cases} \text{agg}_i(e_i) & \text{if agg}_i \text{ is duplicate agnostic,} \\ \text{agg}_i(e_i * c) & \text{if agg}_i \text{ is sum,} \\ \text{sum}(c) & \text{if agg}_i(e_i) = \text{count}(*), \end{cases}$$

and if $\text{agg}_i(e_i)$ is $\text{count}(e_i)$, then
$\text{agg}_i'(e_i) := \text{sum}(e_i = \text{NULL} ? 0 : c)$.

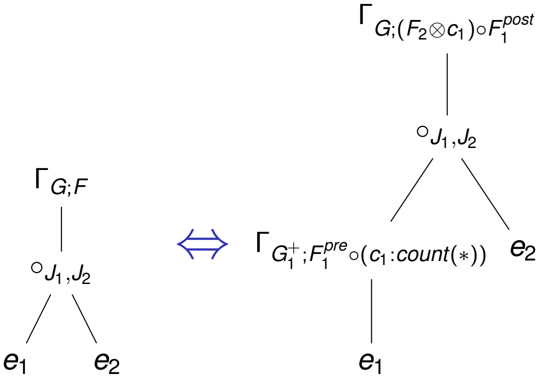
Transformations [1]

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

[1] W. Yan and P.-A. Larson, "Eager Aggregation and Lazy Aggregation", VLDB, 1995

Eager/Lazy Groupby-Count



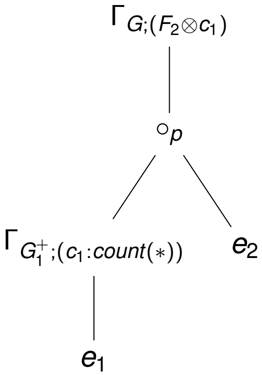
- G_i: Grouping attributes from e_i
- G_i^+ : $G_i \cup J_i$
- Split F into F_1 and F_2
- Decompose F_1 into F_1^{pre} and F_1^{post}

Eager/Lazy Group-By

$$
\begin{tikzcd}
& \Gamma_{G;F_1^{post}} \arrow[d, dash] & \\
& \circ_p \arrow[dl, dash] \arrow[dr, dash] & \\
\Gamma_{G_1^+;F_1^{pre}} \arrow[d, dash] & & e_2 \\
e_1 & &
\end{tikzcd}
$$

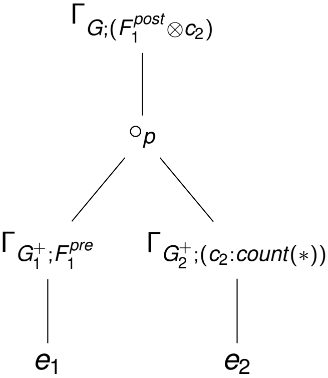
- If $F_2 = ()$

Eager/Lazy Count



- If $F_1 = ()$

Double Eager/Lazy



- If $F_2 = ()$

Eager/Lazy Split

$$
\begin{tikzcd}
& \Gamma_{G;(F_1^{post} \otimes c_2)\circ(F_2^{post} \otimes c_1)} \arrow[d, dash] & \\
& \circ_p \arrow[dl, dash] \arrow[dr, dash] & \\
\Gamma_{G_1^+;F_1^{pre}\circ(c_1:count(*))} \arrow[d, dash] & & \Gamma_{G_2^+;F_2^{pre}\circ(c_2:count(*))} \arrow[d, dash] \\
e_1 & & e_2
\end{tikzcd}
$$

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

$$\Gamma_{G;F}(e_1 ⟗_{J_1,J_2} e_2)$$
$$\equiv$$
$$\Gamma_{G;(F_2 \otimes c_1) \circ F_1^{post}}(\Gamma_{G_1^+;F_1^{pre} \circ (c_1:\text{count}(*))}(e_1) ⟗_{J_1,J_2}^{F_1^{pre}(\{\perp\}) \circ (c_1:1);-} e_2)$$

$e_1 ⟗_p^{D_1;D_2} e_2$: Outer join with default values D_1, D_2

$F_1^{pre}(\perp)$: Application of $F_1^{pre}(\perp)$ to a bag of null-tuples

Example (1/7)

e_1

g_1	j_1	a_1
1	1	2
1	2	4
1	2	8
1	3	7

e_2

g_2	j_2	a_2
1	1	2
1	1	4
1	2	8
1	4	9

$e_3 := e_1 ⟗_{j_1=j_2} e_2$

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

Example (2/7)

$$e_3 := e_1 ⟗_{j_1=j_2} e_2$$

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

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

g_1	g_2	s_1	s_2
1	1	16	22
1	-	7	-
-	1	-	9

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

Eager/Lazy Groupby-Count with Full Outer Join

$$\Gamma_{G;F}(e_1 \bowtie_{J_1,J_2} e_2)$$
$$\equiv$$
$$\Gamma_{G;(F_2 \otimes c_1) \circ F_1^{post}}(\Gamma_{G_1^+;F_1^{pre} \circ (c_1:\mathrm{count}(*))}(e_1) \bowtie_{J_1,J_2}^{F_1^{pre}(\{\perp\}) \circ (c_1:1);-} e_2)$$

Example (3/7)

	e_1	
g_1	j_1	a_1
1	1	2
1	2	4
1	2	8
1	3	7

$e_5 := \Gamma_{g_1, j_1; F_1^{pre} \circ (c_1 : count(*))}(e_1)$

g_1	j_1	c_1	s_1'
1	1	1	2
1	2	2	12
1	3	1	7

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

- Split F:
  - $F_1 = s_1 : sum(a_1)$
  - $F_2 = s_2 : sum(a_2)$
- Decompose F_1:
  - $F_1^{pre} = s_1' : sum(a_1)$
  - $F_1^{post} = s_1 : sum(s_1')$

Example (4/7)

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

$e_5 := \Gamma_{g_1, j_1; F_1^{pre} \circ (c_1 : count(*))}(e_1)$

g_1	j_1	c_1	s'_1
1	1	1	2
1	2	2	12
1	3	1	7

e_2

g_2	j_2	a_2
1	1	2
1	1	4
1	2	8
1	4	9

$e_6 := e_5 ⟗_{j_1 = j_2} e_2$

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	4	9

Example (5/7)

$e_6 := e_5 \bowtie_{j_1=j_2} e_2$

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	4	9

$e_7 := \Gamma_{g_1,g_2;F_X}(e_5)$

g_1	g_2	s_1	s_2
1	1	16	22
1	-	7	-
-	1	-	-

$\neq$

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

g_1	g_2	s_1	s_2
1	1	16	22
1	-	7	-
-	1	-	9

$$\begin{aligned} F &= s_1 : sum(a_1), s_2 : sum(a_2) \\ F_X &= (F_2 \otimes c_1) \circ F_1^{post} \\ &= s_2 : \mathrm{sum}(c_1 * a_2), s_1 : \mathrm{sum}(s'_1) \end{aligned}$$

Example (6/7)

Using the full outer join with default values:

$e_5 := \Gamma_{g_1, j_1; F_1^{pre} \circ (c_1 : count(*))}(e_1)$

g_1	j_1	c_1	s'_1
1	1	1	2
1	2	2	12
1	3	1	7

e_2

g_2	j_2	a_2
1	1	2
1	1	4
1	2	8
1	4	9

$e'_6 := e_5 \mathbin{⟗}_{j_1 = j_2}^{F_1^{pre}(\{\perp\}) \circ (c_1 : 1); -} e_2$

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

Example (7/7)

$$e_6' := e_5 \bowtie^{F_1^{pre}(\{\perp\})\circ(c_1:1);-}_{j_1=j_2} e_2$$

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

$$e_7' := \Gamma_{g_1,g_2;F_X}(e_6')$$

g_1	g_2	s_1	s_2
1	1	16	22
1	-	7	-
-	1	-	9

=

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

g_1	g_2	s_1	s_2
1	1	16	22
1	-	7	-
-	1	-	9

$$\begin{aligned} F &= s_1 : sum(a_1), s_2 : sum(a_2) \\ F_X &= (F_2 \otimes c_1) \circ F_1^{post} \\ &= s_2 : \mathrm{sum}(c_1 * a_2), s_1 : \mathrm{sum}(s_1') \end{aligned}$$

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($S_1, S_2, \circ_p$)

```
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

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

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

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

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

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

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

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

if $\textsc{Valid}(\mathit{NewTree})$
 $\wedge \textsc{NeedsGrouping}(G_1^+, \mathit{NewTree})$
 $\wedge \textsc{NeedsGrouping}(G_2^+, \mathit{NewTree}))$

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

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

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

return Trees

Cost and Cardinality Estimation

- 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

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

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_{opt} the plan which minimizes $\mathcal{M}P$.
- Denote by P_{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

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

θ, q-acceptability

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:

θ, q-acceptability

θ, 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 θ, q-acceptable if

itemsep=2pt $f \leq \theta \ \wedge \ \hat{f} \leq \theta$ or

iitemsep=2pt $||\hat{f}/f||_Q \leq q$.

We define that a binary estimation $\hat{f}^+$ for f^+ is θ, 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 θ, q-acceptable.
Remark: If θ is chosen carefully, the plan generator does not make incorrect choices.

Testing θ, q-acceptability

Directly testing θ,q-acceptability for a given bucket for a continuous domain is impossible since it would involve testing θ,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

Discretization

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

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

or

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

then the estimation function is θ,q-acceptable for the bucket. □

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

- itemsep=2pt the cumulated bucket frequency is less than or equal to θ or
- iitemsep=2pt $\max_i f_i / \min_i f_i \leq q^2$,

then there exists an estimation function $\hat{f}^+$ that is θ,q-acceptable. $\square$

Testing θ, q-acceptability

`isThetaQAcc`(l, u)
if the pretest succeeds
    then return true
if `MaxSize` ¡ ($l - u$)
    then return false
return the result of the subquadratic test

where `MaxSize` is a tuning parameter.

Building Histograms

- Given the test of θ, 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