Generating Permutations

The algorithms so far have some drawbacks:

- greedy heuristics only heuristics
- will probably not find the optimal solution
- DP algorithms optimal, but very heavy weight
- especially memory consumption is high
- find a solution only after the complete search

Sometimes we want a more light-weight algorithm:

- low memory consumption
- stop if time runs out
- still find the optimal solution if possible
Generating Permutations (2)

We can achieve this when only considering left-deep trees:

- left-deep trees are permutations of the relations to be joined
- permutations can be generated directly
- generating all permutations is too expensive
- but some permutations can be ignored:
  Consider the join sequence $R_1 R_2 R_3 R_4$. If we know that $R_1 R_3 R_2$ is cheaper than $R_1 R_2 R_3$, we do not have to consider $R_1 R_2 R_3 R_4$.

Idea: successively add a relation. An extended sequence is only explored if exchanging the last two relations does not result in a cheaper sequence.
Recursive Search

ConstructPermutations\((R)\)

**Input:** a set of relations \(R = \{R_1, \ldots, R_n\}\) to be joined

**Output:** an optimal left-deep join tree

\[ B = \epsilon \]

\[ P = \epsilon \]

for \(\forall R_i \in R\) {
    ConstructPermutationsRec\((P \circ < R_i >, R \setminus \{R_i\}, B)\)
}

return \(B\)

- algorithm considers a prefix \(P\) and the rest \(R\)
- keeps track of the best tree found so far \(B\)
- increases the prefix recursively
Recursive Search (2)

ConstructPermutationsRec(\(P, R, B\))

**Input:** a prefix \(P\), remaining relations \(R\), best plan \(B\)

**Output:** side effects on \(B\)

if \(|R| = 0\) {
    if \(B = \epsilon \lor C(B) > C(P)\) {
        \(B = P\)
    }
} else {
    for \(\forall R_i \in R\) {
        if \(C(P \circ < R_i >) \leq C(P[1 : |P| - 1] \circ < R_i, P[|P|] >)\) {
            ConstructPermutationsRec(\(P \circ < R_i >, R \setminus \{R_i\}, B\))
        }
    }
}
Remarks

Good:
- linear memory
- immediately produces plan alternatives
- anytime algorithm
- finds the optimal plan eventually

Bad:
- worst-case runtime of ties occur
- worst-case runtime of no ties occur is an open problem

Often fast, can be stopped anytime, but can perform poor.
Transformative Approaches

Main idea: [6]
- use equivalences directly (associativity, commutativity)
- would make integrating new equivalences easy

Problems:
- how to navigate the search space
- equivalences have no order
- how to guarantee finding the optimal solution
- how to avoid exhaustive search
Rule Set

\[ R_1 \Join R_2 \quad \leadsto \quad R_2 \Join R_1 \quad \text{Commutativity} \]
\[ (R_1 \Join R_2) \Join R_3 \quad \leadsto \quad R_1 \Join (R_2 \Join R_3) \quad \text{Right Associativity} \]
\[ R_1 \Join (R_2 \Join R_3) \quad \leadsto \quad (R_1 \Join R_2) \Join R_3 \quad \text{Left Associativity} \]
\[ (R_1 \Join R_2) \Join R_3 \quad \leadsto \quad (R_1 \Join R_3) \Join R_2 \quad \text{Left Join Exchange} \]
\[ R_1 \Join (R_2 \Join R_3) \quad \leadsto \quad R_2 \Join (R_1 \Join R_3) \quad \text{Right Join Exchange} \]

Two more rules are often used to transform left-deep trees:

- **swap** exchanges two arbitrary relations in a left-deep tree
- **3Cycle** performs a cyclic rotation of three arbitrary relations in a left-deep tree.

To try another join method, another rule called *join method exchange* is introduced.
Rule Set RS-0

- commutativity
- left-associativity
- right-associativity
Basic Algorithm

ExhaustiveTransformation(\{R_1, \ldots, R_n\})

**Input:** a set of relations

**Output:** an optimal join tree

Let $T$ be an arbitrary join tree for all relations

Done = $\emptyset$ // contains all trees processed

ToDo = \{ $T$ \} // contains all trees to be processed

while $|\text{ToDo}| > 0$ {
    $T =$ an arbitrary tree in ToDo
    ToDo = ToDo \ $T$;
    Done = Done $\cup$ \{ $T$ \};
    Trees = ApplyTransformations($T$);
    for $\forall$ $T \in$ Trees {
        if $T \notin$ ToDo $\cup$ Done
            ToDo = ToDo $\cup$ \{ $T$ \}
    }
}

return $\arg \min_{T \in \text{Done}} C(T)$
Basic Algorithm (2)

ApplyTransformations( T )

**Input:** join tree

**Output:** all trees derivable by associativity and commutativity

Trees = ∅

Subtrees = all subtrees of T rooted at inner nodes

**for** ∀ S ∈ Subtrees {
  **if** S is of the form S₁ ⨂ S₂
  Trees = Trees ∪{ S₂ ⨂ S₁ }
  **if** S is of the form ( S₁ ⨂ S₂ ) ⨂ S₃
  Trees = Trees ∪{ S₁ ⨂ ( S₂ ⨂ S₃ ) }
  **if** S is of the form S₁ ⨂ ( S₂ ⨂ S₃ )
  Trees = Trees ∪{ ( S₁ ⨂ S₂ ) ⨂ S₃ }
}

**return** Trees;
Remarks

- if no cross products are to be considered, extend \textbf{if} conditions for associativity rules.
- problem 1: explores the whole search space
- problem 2: generates join trees more than once
- problem 3: sharing of subtrees is non-trivial
Search Space
Introducing the Memo Structure

A memoization strategy is used to keep the runtime reasonable:

• for any subset of relations, dynamic programming remembers the best join tree.
• this does not quite suffice for the transformation-based approach.
• instead, we have to keep all join trees generated so far including those differing in the order of the arguments of a join operator.
• however, subtrees can be shared.
• this is done by keeping pointers into the data structure (see next slide).
## Memo Structure Example

<table>
<thead>
<tr>
<th>{R_1, R_2, R_3}</th>
<th>{R_1, R_2} \times R_3, R_3 \times {R_1, R_2}, {R_1, R_3} \times R_2, R_2 \times {R_1, R_3}, {R_2, R_3} \times R_1, R_1 \times {R_2, R_3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R_2, R_3}</td>
<td>{R_2} \times {R_3}, {R_3} \times {R_2}</td>
</tr>
<tr>
<td>{R_1, R_3}</td>
<td>{R_1} \times {R_3}, {R_3} \times {R_1}</td>
</tr>
<tr>
<td>{R_1, R_2}</td>
<td>{R_1} \times {R_2}, {R_2} \times {R_1}</td>
</tr>
<tr>
<td>{R_3}</td>
<td>R_3</td>
</tr>
<tr>
<td>{R_2}</td>
<td>R_2</td>
</tr>
<tr>
<td>{R_1}</td>
<td>R_1</td>
</tr>
</tbody>
</table>

- in Memo Structure: arguments are pointers to classes
- Algorithm: `ExploreClass` expands a class
- Algorithm: `ApplyTransformation2` expands a member of a class
Memoizing Algorithm

ExhaustiveTransformation2(Query Graph $G$)

**Input:** a query specification for relations $\{R_1, \ldots, R_n\}$.

**Output:** an optimal join tree

initialize MEMO structure

ExploreClass($\{R_1, \ldots, R_n\}$)

return $\arg\min_{T \in \text{class } \{R_1, \ldots, R_n\}} C(T)$

- stored an arbitrary join tree in the memo structure
- explores alternatives recursively
Memoizing Algorithm (2)

ExploreClass(C)

**Input:** a class $C \subseteq \{R_1, \ldots, R_n\}$

**Output:** none, but has side-effect on MEMO-structure

while not all join trees in C have been explored {
    choose an unexplored join tree $T$ in $C$
    ApplyTransformation2($T$)
    mark $T$ as explored
}

- considers all alternatives within one class
- transformations themselves are done in ApplyTransformation2
Memoizing Algorithm (3)

ApplyTransformations2(\(T\))

**Input:** a join tree of a class \(C\)

**Output:** none, but has side-effect on MEMO-structure

ExploreClass(left-child(\(T\)))
ExploreClass(right-child(\(T\)))

for \(\forall\) transformation \(T\) and class member of child classes {

for \(\forall T'\) resulting from applying \(T\) to \(T\) {

if \(T'\) not in MEMO structure {

add \(T'\) to class \(C\) of MEMO structure

}

}

}

- first explores subtrees
- then applies all known transformations to the tree
- stores new trees in the memo structure
Remarks

- Applying ExhaustiveTransformation2 with a rule set consisting of Commutativity and Left and Right Associativity generates $4^n - 3^{n+1} + 2^{n+2} - n - 2$ duplicates.
- Contrast this with the number of join trees contained in a completely filled MEMO structure: $3^n - 2^{n+1} + n + 1$.
- Solve the problem of duplicate generation by disabling applied rules.
Rule Set RS-1

\( T_1: \text{Commutativity} \quad C_1 \Join_0 C_2 \rightsquigarrow C_2 \Join_1 C_1 \)

Disable all transformations \( T_1, T_2, \) and \( T_3 \) for \( \Join_1 \).

\( T_2: \text{Right Associativity} \quad (C_1 \Join_0 C_2) \Join_1 C_3 \rightsquigarrow C_1 \Join_2 (C_2 \Join_3 C_3) \)

Disable transformations \( T_2 \) and \( T_3 \) for \( \Join_2 \) and enable all rules for \( \Join_3 \).

\( T_3: \text{Left associativity} \quad C_1 \Join_0 (C_2 \Join_1 C_3) \rightsquigarrow (C_1 \Join_2 C_2) \Join_3 C_3 \)

Disable transformations \( T_2 \) and \( T_3 \) for \( \Join_3 \) and enable all rules for \( \Join_2 \).
## Example for chain $R_1 - R_2 - R_3 - R_4$

<table>
<thead>
<tr>
<th>Class</th>
<th>Initialization</th>
<th>Transformation</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>${R_1, R_2, R_3, R_4}$</td>
<td>${R_1, R_2} \bowtie_{111} {R_3, R_4}$</td>
<td>$R_1 \bowtie_{100} {R_2, R_3, R_4}$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${R_1, R_2, R_3} \bowtie_{100} R_4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>${R_2, R_3, R_4} \bowtie_{000} R_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R_4 \bowtie_{000} {R_1, R_2, R_3}$</td>
<td></td>
</tr>
<tr>
<td>${R_2, R_3, R_4}$</td>
<td>$R_2 \bowtie_{111} {R_3, R_4}$</td>
<td>$R_2 \bowtie_{111} {R_3, R_4}$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${R_3, R_4} \bowtie_{000} R_2$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${R_2, R_3} \bowtie_{100} R_4$</td>
<td>4</td>
</tr>
<tr>
<td>${R_1, R_3, R_4}$</td>
<td>$R_4 \bowtie_{000} {R_1, R_2}$</td>
<td>${R_1, R_2} \bowtie_{111} R_3$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R_3 \bowtie_{000} {R_1, R_2}$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R_1 \bowtie_{100} {R_2, R_3}$</td>
<td>5</td>
</tr>
<tr>
<td>${R_1, R_2, R_3}$</td>
<td>${R_2, R_3} \bowtie_{000} R_1$</td>
<td>${R_2, R_3} \bowtie_{000} R_1$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R_4 \bowtie_{000} R_3$</td>
<td>5</td>
</tr>
<tr>
<td>${R_3, R_4}$</td>
<td>$R_3 \bowtie_{111} R_4$</td>
<td>$R_1 \bowtie_{111} R_2$</td>
<td>1</td>
</tr>
<tr>
<td>${R_2, R_4}$</td>
<td></td>
<td>$R_2 \bowtie_{000} R_1$</td>
<td>1</td>
</tr>
<tr>
<td>${R_2, R_3}$</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>${R_1, R_4}$</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>${R_1, R_3}$</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>${R_1, R_2}$</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Rule Set RS-2

Bushy Trees: Rule set for clique queries and if cross products are allowed:

\( T_1: \) Commutativity \( C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1 \)
Disabling all transformations \( T_1, T_2, T_3, \) and \( T_4 \) for \( \bowtie_1 \).

\( T_2: \) Right Associativity \( (C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3) \)
Disabling transformations \( T_2, T_3, \) and \( T_4 \) for \( \bowtie_2 \).

\( T_3: \) Left Associativity \( C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3 \)
Disabling transformations \( T_2, T_3 \) and \( T_4 \) for \( \bowtie_3 \).

\( T_4: \) Exchange \( (C_1 \bowtie_0 C_2) \bowtie_1 (C_3 \bowtie_2 C_4) \rightsquigarrow (C_1 \bowtie_3 C_3) \bowtie_4 (C_2 \bowtie_5 C_4) \)
Disabling all transformations \( T_1, T_2, T_3, \) and \( T_4 \) for \( \bowtie_4 \).

If we initialize the MEMO structure with left-deep trees, we can strip down the above rule set to Commutativity and Left Associativity. Reason: from a left-deep join tree we can generate all bushy trees with only these two rules.
Rule Set RS-3

Left-deep trees:

$T_1$ Commutativity  $R_1 \bowtie_0 R_2 \leadsto R_2 \bowtie_1 R_1$

Here, the $R_i$ are restricted to classes with exactly one relation. $T_1$ is disabled for $\bowtie_1$.

$T_2$ Right Join Exchange  $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \leadsto (C_1 \bowtie_2 C_3) \bowtie_3 C_2$

Disable $T_2$ for $\bowtie_3$. 
Generating Random Join Trees

Generating a random join tree is quite useful:

- allows for cost sampling
- randomized optimization procedures
- basis for Simulated Annealing, Iterative Improvement etc.
- easy with cross products, difficult without
- we consider with cross products first

Main problems:

- generating all join trees (potentially)
- creating all with the same probability
Ranking/Unranking

Let $S$ be a set with $n$ elements.

- a bijective mapping $f : S \rightarrow [0, n[$ is called ranking
- a bijective mapping $f : [0, n[ \rightarrow S$ is called unranking

Given an unranking function, we can generate random elements in $S$ by generating a random number in $[0, n[$ and unranking this number. Challenge: making unranking fast.
Random Permutations

Every permutation corresponds to a left-deep join tree possibly with cross products.
Standard algorithm to generate random permutations is the starting point for the algorithm:

\[
\textbf{for } \forall k \in [0, n[ \text{ \textbf{descending } swaps: } \\
(\pi[k], \pi[\text{random}(k)])
\]

Array \( \pi \) initialized with elements \([0, n[\).
\text{random}(k) generates a random number in \([0, k[\).
Random Permutations

- Assume the random elements produced by the algorithm are $r_{n-1}, \ldots, r_0$ where $0 \leq r_i \leq i$.
- Thus, there are exactly $n(n-1)(n-2)\ldots1 = n!$ such sequences and there is a one to one correspondence between these sequences and the set of all permutations.
- Unrank $r \in [0, n!]$ by turning it into a unique sequence of values $r_{n-1}, \ldots, r_0$.
  Note that after executing the swap with $r_{n-1}$ every value in $[0, n]$ is possible at position $\pi[n-1]$.
  Further, $\pi[n-1]$ is never touched again.
- Hence, we can unrank $r$ as follows. We first set $r_{n-1} = r \mod n$ and perform the swap. Then, we define $r' = \lfloor r/n \rfloor$ and iteratively unrank $r'$ to construct a permutation of $n-1$ elements.
Generating Random Permutations

Unrank($n, r$)

**Input:** the number $n$ of elements to be permuted and the rank $r$ of the permutation to be constructed

**Output:** a permutation $\pi$

for $\forall 0 \leq i < n$

$\pi[i] = i$

for $\forall n \geq i > 0 \text{ descending}$ {

swap($\pi[i - 1], \pi[r \mod i]$)

$r = \left\lfloor r/i \right\rfloor$

}

return $\pi$;