Relational Database Systems 2
8. Join Order Optimization

Christoph Lofi
Benjamin Köhncke
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
Introduction into heuristic query optimization
Simple heuristics commonly used
Heuristics in action
Complex heuristics
Optimizer hints
8 Join Order Optimization

8.1 Basic join order optimization
8.2 Join cost and size estimations
8.3 Left-deep join trees
8.4 Dynamic programming
8.5 Greedy strategy
8.6 Randomized algorithms
8.1 Introduction

• Joins are **commutative** and **associative**
  – \( R \bowtie S \equiv S \bowtie R \)
  – \( R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T \)

• This allows to evaluate individual joins in any order
  – Results in **join trees**
    • Different join trees may show very different evaluation performance
  – Join trees have different **shapes**
  – Within a shape, there are different relation **assignments** possible

• Example: \( R \bowtie S \bowtie T \bowtie U \)
8.1 Shapes of Join Trees

- Number of possible join trees grows rapidly with number of join relations
  - For $n$ relations, there are $T(n)$ different tree shapes

  - $T(1) = 1$
  - $T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$

- “Any number of $1 \leq i \leq n-1$ relations may be in the left subtree and ordered in $T(i)$ shapes while the remaining $n-i$ relations form the right subtree and can be arranged in $T(n-i)$ shapes.”
This number sequence is called Catalan Numbers

- Named after Belgian mathematician Eugène Charles Catalan (1814–1894)
- Can be rewritten as

\[
T(n) = C(n) = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}
\]
8.1 Shapes of Join Trees

– Example: Shapes for \( n=4 \)

– Example: The 21 first Catalan Numbers:
  
  • 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020
  
  • Grows quite quickly…. 
8.1 Sequence of Relations

- For each shape, the relations can be assigned in $n!$ ways to the tree nodes
  - Example: Left-deep tree shape for $n=3$

- There are $T(n)*n!$ different join trees for $n$ relations!

<table>
<thead>
<tr>
<th>$n$</th>
<th>$1*2! = 2$</th>
<th>$6 : 42*6! = 30,240$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n=2$</td>
<td>$2*3! = 12$</td>
<td>$9 : 1,430*12! = 518E6$</td>
</tr>
<tr>
<td>$n=3$</td>
<td>$5*4! = 120$</td>
<td>$12 : 58,786*12! = 28E12$</td>
</tr>
<tr>
<td>$n=4$</td>
<td>$14*5! = 1,680$</td>
<td>$15 : 2,674,440*15! = 3.49E18$</td>
</tr>
</tbody>
</table>
Finding the “most efficient” join tree and join implementation is a challenging problem.

- Number of possible join trees grows extremely with the number of join relations.
  - Problem was shown to be NP-hard in the general case.
  - \( O(n!) \), with \( n \) as number of join relations.
  - Estimating cost of all trees is not feasible for larger joins.

- Some join implementations are asymmetric.
  - Performance varies greatly depending on relation order.

Query optimizer has to find a good plan in sensible time.
8.1 Basic Join Order Optimization

• Naming convention
  – Left: Build Relation
  – Right: Probe Relation

• Desirable Join Cases
  – Attention: Role (inner/outer relation) of build and probe depends on chosen algorithm
  – **Block Nested Loop Join**
    • Build relation is in *inner loop*, probe relation is in *outer loop*
    • Build relation **significantly** smaller than probe
  – **“Single Pass Join”**
    • Best case
    • **Nested Loop Join** where build relation fits completely into main memory
  – **Index Join**
    • Build relation is in *outer loop*, probe relation is in *inner loop*
    • Index on probe relation
    • Build relation small
8.1 Basic Join Order Optimization

• Optimizer has 3 choices
  – Consider all possible join trees
    • Usually not possible
  – Consider a subset of all trees
    • i.e. restrict to trees of certain shapes
  – Use heuristics to pick a certain shape
8.2 Join Metrics

• For optimizing joins, metrics are necessary
  – Estimated **Join Result Size**
    • “What is the expected size of the result set?”
    • Needed by the query optimizer for global query optimization
    • May be used within the cost metric
    • Is **the same** for all different join orders
  – Estimated **Join Cost**
    • Represents the actual costs for performing the join
    • May consider CPU, I/O, buffer statistics, etc. and varies with join algorithm implementation
8.2 Join Metrics – Size

• **Size Estimation:**
  
  – A join selects tuples fulfilling a join condition from a Cartesian product: \( R \bowtie_c S \equiv \sigma_c (R \times S) \)
    
    • \( |R \bowtie_c S| \leq |R \times S| \)
    • \( |R \bowtie_c S| \leq |R|*|S| \)
    • \( |R \bowtie_c S| = r_{\sigma} *|R|*|S| \)
      
      – \( r_{\sigma} \) the reduction factor of the selection with the join condition wrt. the Cartesian product (fraction of remaining tuples after selection)

  – We need to estimate the reduction factor of the selection!
8.2 Join Metrics – Size

- Remember lecture 6.4: reduction factors for selections
  - Simplified: Two useful cases for joins
  - $\text{rel}_1.\text{col}_1 = \text{rel}_2.\text{col}_2$ (natural join, equijoin) shows a reduction factor: $\frac{1}{\max(\#dV(\text{rel}_1, \text{col}_1), \#dV(\text{rel}_2, \text{col}_2))}$
    - Assuming that every tuple in the smaller relation has a match in the larger relation
    - $\#dV(R, A)$ is number of distinct values of attribute $A$ in relation $R$
  - $\text{rel}_1.\text{col}_1 \theta \text{rel}_2.\text{col}_2$, $\theta \in \{\leq, <, =, >, \geq, \neq\}$ (theta join) shows a reduction factor of about $0.5$
    - Assuming that each value in $\text{rel}_1$ is joined with about half the values in $\text{rel}_2$
8.2 Join Metrics – Size

• **Observations:** $R \bowtie S$ on common attribute $A$
  
  – Natural join: $R.A = S.A$
  
  – Join size depends heavily on the relation of values of $A$ in $R$ and $S$
    
    • $R$ and $S$ may have **disjoint** $A$ values
      
      – $rf_\sigma = 0 \implies |R \bowtie S| = 0$
    
    • $A$ might be **key** of $S$ and **foreign key** of $R$
      
      – Each tuple of $R$ joins with exactly one tuple of $S$
        
        – $\implies |R \bowtie S| = |R|$
    
    • Most tuples of $R$ and $S$ could have **equal** values for $A$
      
      – $rf_\sigma \approx 1 \implies |R \bowtie S| \approx |R|^*|S|$
8.2 Join Metrics – Size

• Idealized case with keys and foreign keys
  – Usually, tuples of one relation match a tuple in the other
    • Attribute A is key of S and foreign key of R
      ⇒ rf_σ = 1 / #dV(R, A)
    • Attribute A is key of R and foreign key of S
      ⇒ rf_σ = 1 / #dV(S, A)
    • You don’t know which relation contains key and which foreign key
      ⇒ rf_σ = 1 / max(#dV(R, A), #dV(S, A))
  – | R⋈S | = |R|*|S| /max(#dV(R, A), #dV(S, A))
8.2 Join Metrics – Size

• Join Result Sizes
  – For a single equality join condition on $A$:
    • $|R \bowtie S| = |R| \times |S| / \text{max}(\#dV(R, A), \#dV(S, A))$
  – For multiple equality join conditions on $A_1, \ldots, A_n$:
    • Multiply reduction factors
    • $|R \bowtie S| = |R| \times |S| / \Pi_i \text{max}(\#dV(R, A_i), \#dV(S, A_i))$
  – For multiple join relations $R_1, \ldots, R_n$
    • Cascade formula for two relations
    • i.e. $|R_1 \bowtie \cdots \bowtie R_n| = |(||R_1 \bowtie R_2| \bowtie \cdots | \bowtie R_n|)$
    • Order of relations does not matter for total size estimation
8.2 Join Metrics – Size

– Estimation accuracy can be improved using more sophisticated statistics
  • Histograms
  • Dynamic sampling
  • Simulating common queries
  • Correlation Statistics
  • Incorporating previous query results

– Are more complex statistics worth it?
  • Keeping statistics is expensive in databases with high change rate
  • Which statistics to create?
    – Adapting statistics to queries?
• **Execution Cost Estimation:**
  – For selecting a good join tree, we have to minimize the actual costs for computing the result
  – Easiest cost metric: *size of intermediate results*
    • Creating intermediate results is costly (writing to disk), costs increase with size of relations
    • Final result is *not* an intermediate result
    • **Example:**
      – Costs \((R \bowtie S)\) = 0
      – Costs \(((R \bowtie S) \bowtie T)\) = \(|R \bowtie S|\)
    • Metric does not consider real I/O, memory and CPU costs
    • Metric ignores actual join algorithm implementation
8.2 Join Cost Metrics – Access Costs

• **Cost metric: block accesses**
  
  – Block accesses are the major performance bottlenecks
  
  – Depends on the used join implementation
  
  – **Costs for writing the result**
    
    • $\text{Costs}_{\text{Result}} (R \bowtie S) = |R \bowtie S| / \text{blockingFactor}_{\text{Result}}$
    
    • Use size estimation for $|R \bowtie S|$

  
  – **Block Nested Loop**
    
    • $\text{Costs}_{\text{BNL}} (R \bowtie S) = b_R + (b_R \cdot b_S) + \text{Costs}_{\text{Result}} (R \bowtie S)$
    
    • $b_R$ is number of blocks in $R$
    
    • $b_S$ is number of blocks in $S$
8.2 Join Cost Metrics – Access Costs

- **Block Access Costs:** *Index Loop Join*
  
  \[ \text{Costs}_{\text{IXL}}(R \bowtie S) = b_R + (|R| \times (C_{\text{ix}} + 1)) + \text{Costs}_{\text{Result}}(R \bowtie S) \]

- Costs depend on **index retrieval cost** \( C_{\text{ix}} \)
  
  - **IndexAccessCosts** vary on the type of index
    
    - Assume 0 for in-memory index
    - Increasing costs per index level for disk residing indexes
8.2 Join Cost Metrics – Access Costs

- Retrieval costs for some different indexes
  - With $S_\sigma$, the selection cardinality of $S$: Estimated number of records in $S$ fulfilling the join condition for a given record from $R$; see lecture 6.4
    - **Cluster index**
      - $C_{ix} = \text{indexAccessCost} + \left( \frac{S_\sigma}{\text{blockingFactor}_{\text{index}}} \right)$
    - **Secondary Index**:
      - $C_{ix} = \text{indexAccessCost} + S_\sigma$
    - **Hash Index**:
      - $C_{ix} \geq 1$; Average costs for retrieving a record, depending of hash size and key collision
8.2 Join Cost Metrics – Access Costs

• Block access costs: **Sort-Merge-Join**
  – Assume that relations are already sorted on the join attribute:

  \[
  \text{Costs}_{\text{SMJ}} (R \bowtie S) = b_R + b_S + \text{Costs}_{\text{Result}} (R \bowtie S)
  \]

  • Very efficient when already sorted
  • If not, additional costs for sorting have to be considered
A simple heuristic for reducing the search space size is using **left-deep join trees**

- Introduced by System R optimizer
- Considers **only one tree shape**: left-deep tree
  - In left-deep trees, all right children are leaves
8.3 Left-deep Join Trees

• Left-deep join trees cooperate well with most join algorithms as they aim for decreasing the build relation
  – Usually, left-deep join tree yield good performance
  – Optimized buffer usage

• Left-deep plans allow output of each operator to be pipelined into the next operator
  – No need to store results in a temporary relation
  – Careful: not for sort-merge joins
8.3 Left-deep Join Trees

• The number of possible left-deep join trees is significantly smaller than the number of all join trees

<table>
<thead>
<tr>
<th>n</th>
<th>All join trees</th>
<th>Left-deep trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>30,240</td>
<td>720</td>
</tr>
<tr>
<td>9</td>
<td>518E6</td>
<td>362,880</td>
</tr>
<tr>
<td>12</td>
<td>28E12</td>
<td>479E6</td>
</tr>
</tbody>
</table>

• But…
  – **Still** a considerable amount (impractical for >15 joins)
  – Parallel execution of joins is **not possible**!
8.4 Finding the Best Tree

- Exploring all possible join orders is not possible
  - Employ techniques for reducing search space which still deliver **best solution**
    - Dynamic Programming
    - Branch and Bound
  - Employ **approximate techniques** that deliver a sufficiently good solution
    - Greedy Strategies
    - Randomized Strategies
    - Genetic Algorithms
8.4 Dynamic Programming

• **Dynamic programming** techniques are frequently used to explore the search space more efficiently
  - Break the problem into smaller subproblems
  - Solve these subproblems optimally recursively and remember the best solutions
  • Memorization
  - Use these optimal solutions to construct an optimal solution for the original problem
8.4 Dynamic Programming

• For finding a join plan, DP is often implemented with a **cost table**
  – Table stores lowest costs for joins of subsets of all relations
    • Only good sub-solutions are remembered
    • Use an adequate cost function for joins
      – In the following we assume intermediate result sizes as costs
  – Storing the table uses up buffer space!
8.4 Dynamic Programming

– The table contains columns for
  
  • The relation subset described by the row
  • The estimated size of the join result
  • The estimated lowest costs for performing the join
    – i.e. estimated intermediate result size, estimated IO cost, estimated CPU cost, etc.
  • The expression (i.e. tree shape and assignment) which produced the lowest costs

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{R,S,T,U}</td>
<td>2,500</td>
<td>25,750</td>
<td>(U ⋈ (S ⋈ T)) ⋈ R</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Table is build *inductively* on the subsets of relations

• *Claim:*
  – Table always contains *join expressions* with *lowest costs* for given *relation subsets*
• Basics:
  – For each single relation subset \( \{R_a\} \), table contains one row with size of \( R_a \), with size \( |R_a| \), costs 0 and expression \( R_a \)
  – For each relation subset of size two \( \{R_a, R_b\} \), table contains one row
    • Estimated size as described in previous section
    • Costs 0 (⇒ no temp files!)
    • Either expression \( R_a \bowtie R_b \) or \( R_b \bowtie R_a \); use heuristic to choose which expression is better: usually, order smaller relation to the left

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R_1}</td>
<td>2000</td>
<td>0</td>
<td>( R_1 )</td>
</tr>
<tr>
<td>{R_2}</td>
<td>1000</td>
<td>0</td>
<td>( R_2 )</td>
</tr>
<tr>
<td>{R_1, R_2}</td>
<td>500</td>
<td>0</td>
<td>( R_2 \bowtie R_1 )</td>
</tr>
</tbody>
</table>
• **Induction:**
  
  – For each *relation subset of size* $n$ $Rs = \{R_a, R_b, \ldots, R_z\}$, create a table row
  
  – Find two subsets $Rs_1 \cup Rs_2 = Rs$ within the table such that $\text{Cost}(Rs_1 \bowtie Rs_2)$ are minimal
    
    • For *deep-left trees*, only subsets with $|Rs_1| = n-1$ and $|Rs_2| = 1$ need to be considered
8.4 Dynamic Programming

- Fill row with
  - $Rs$ as subset identifier
  - Estimated size $|Rs_1 \bowtie Rs_2|$
  - Estimated costs $Cost(Rs_1 \bowtie Rs_2)$
  - Concatenation of the expressions of $Rs_1$ and $Rs_2$
    - For deep-left join trees, always place expression of $Rs_1$ to the left
    - otherwise, place expression with smaller result size to the left

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>${R_1}$</td>
<td>2000</td>
<td>0</td>
<td>$R_1$</td>
</tr>
<tr>
<td>${R_2}$</td>
<td>1000</td>
<td>0</td>
<td>$R_2$</td>
</tr>
<tr>
<td>${R_3}$</td>
<td>3000</td>
<td>0</td>
<td>$R_2$</td>
</tr>
<tr>
<td>${R_1, R_2}$</td>
<td>500</td>
<td>0</td>
<td>$R_2 \bowtie R_1$</td>
</tr>
<tr>
<td>${R_1, R_3}$</td>
<td>1200</td>
<td>0</td>
<td>$R_1 \bowtie R_3$</td>
</tr>
<tr>
<td>${R_2, R_3}$</td>
<td>1800</td>
<td>0</td>
<td>$R_2 \bowtie R_3$</td>
</tr>
<tr>
<td>${R_1, R_2, R_3}$</td>
<td>200</td>
<td>500</td>
<td>$(R_2 \bowtie R_1) \bowtie R_3$</td>
</tr>
</tbody>
</table>

Here:
- $Rs_1 = \{R_1, R_2\}$
- $Rs_2 = \{R_3\}$
8.4 Dynamic Programming

- Find optimal join order restricted to left-deep join trees
- 4 Relations
  - \( R \) with attributes \( a \) and \( b \)
  - \( S \) with attributes \( b \) and \( c \)
  - \( T \) with attributes \( c \) and \( d \)
  - \( U \) with attributes \( d \) and \( a \)
  - Each relation has size of 1000
  - Following Table: \#dV(\text{Relation, attribute})
    - Number of distinct values for attributes and relations

<table>
<thead>
<tr>
<th>#dV</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>100</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>b</td>
<td>200</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td>50</td>
<td>1000</td>
</tr>
</tbody>
</table>
8.4 Dynamic Programming

- Start with subsets of size one
  - Use intermediate result set size as cost metric
- Fill table with subsets of size two
  - Still no costs because of intermediate result cost metric
  - Heuristic: Smaller relation to the left side of join

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R}</td>
<td>1,000</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>{S}</td>
<td>1,000</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>{T}</td>
<td>1,000</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>{U}</td>
<td>1,000</td>
<td>0</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R, S}</td>
<td>5,000</td>
<td>0</td>
<td>R \bowtie S</td>
</tr>
<tr>
<td>{R, T}</td>
<td>1 M</td>
<td>0</td>
<td>R \bowtie T</td>
</tr>
<tr>
<td>{R, U}</td>
<td>10,000</td>
<td>0</td>
<td>R \bowtie U</td>
</tr>
<tr>
<td>{S, T}</td>
<td>2,000</td>
<td>0</td>
<td>S \bowtie T</td>
</tr>
<tr>
<td>{S, U}</td>
<td>1 M</td>
<td>0</td>
<td>S \bowtie U</td>
</tr>
<tr>
<td>{T, U}</td>
<td>1,000</td>
<td>0</td>
<td>T \bowtie U</td>
</tr>
</tbody>
</table>
8.4 Dynamic Programming

- Fill table with subsets of size three
  - Use previous table entries and combine a subset result of size two with a result of size one
    - Always select pairs smallest size
    - Single relation to the right side due to left-deep join tree restriction
  - For \{R, S, T\} consider:
    - \((R \bowtie S) \bowtie T\) : Costs 5,000
    - \((R \bowtie T) \bowtie S\) : Costs 1,000,000
    - \((S \bowtie T) \bowtie R\) : Costs 2,000

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R}</td>
<td>1,000</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>{S}</td>
<td>1,000</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>{T}</td>
<td>1,000</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>{U}</td>
<td>1,000</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>{R, S}</td>
<td>5,000</td>
<td>0</td>
<td>R \bowtie S</td>
</tr>
<tr>
<td>{R, T}</td>
<td>1 M</td>
<td>0</td>
<td>R \bowtie T</td>
</tr>
<tr>
<td>{R, U}</td>
<td>10,000</td>
<td>0</td>
<td>R \bowtie U</td>
</tr>
<tr>
<td>{S, T}</td>
<td>2,000</td>
<td>0</td>
<td>S \bowtie T</td>
</tr>
<tr>
<td>{S, U}</td>
<td>1 M</td>
<td>0</td>
<td>S \bowtie U</td>
</tr>
<tr>
<td>{T, U}</td>
<td>1,000</td>
<td>0</td>
<td>T \bowtie U</td>
</tr>
<tr>
<td>{R, S, T}</td>
<td>10,000</td>
<td>2,000</td>
<td>(S \bowtie T) \bowtie R</td>
</tr>
<tr>
<td>{R, S, U}</td>
<td>50,000</td>
<td>5,000</td>
<td>(R \bowtie S) \bowtie U</td>
</tr>
<tr>
<td>{R, T, U}</td>
<td>10,000</td>
<td>1,000</td>
<td>(T \bowtie U) \bowtie R</td>
</tr>
<tr>
<td>{S, T, U}</td>
<td>2,000</td>
<td>1,000</td>
<td>(T \bowtie U) \bowtie S</td>
</tr>
</tbody>
</table>
8.4 Dynamic Programming

- Subsets of size four
  - Subsets of size four can be found by combining a triple and a single relation
  - Again, single to the right
- For \( \{R, S, T, U\} \) consider:
  - \((S \bowtie T) \bowtie R) \bowtie U : 12,000\)
  - \((R \bowtie S) \bowtie U) \bowtie T : 55,000\)
  - \((T \bowtie U) \bowtie R) \bowtie S : 11,000\)
  - \((T \bowtie U) \bowtie S) \bowtie R : 3,000\)

<table>
<thead>
<tr>
<th>Subset</th>
<th>Size</th>
<th>Costs</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>{R}</td>
<td>1,000</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>{S}</td>
<td>1,000</td>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>{T}</td>
<td>1,000</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>{U}</td>
<td>1,000</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>{R, S}</td>
<td>5,000</td>
<td>0</td>
<td>R \bowtie S</td>
</tr>
<tr>
<td>{R, T}</td>
<td>1 M</td>
<td>0</td>
<td>R \bowtie T</td>
</tr>
<tr>
<td>{R, U}</td>
<td>10,000</td>
<td>0</td>
<td>R \bowtie U</td>
</tr>
<tr>
<td>{S, T}</td>
<td>2,000</td>
<td>0</td>
<td>S \bowtie T</td>
</tr>
<tr>
<td>{S, U}</td>
<td>1 M</td>
<td>0</td>
<td>S \bowtie U</td>
</tr>
<tr>
<td>{T, U}</td>
<td>1,000</td>
<td>0</td>
<td>T \bowtie U</td>
</tr>
<tr>
<td>{R, S, T}</td>
<td>10,000</td>
<td>2,000</td>
<td>(S \bowtie T) \bowtie R</td>
</tr>
<tr>
<td>{R, S, U}</td>
<td>50,000</td>
<td>5,000</td>
<td>(R \bowtie S) \bowtie U</td>
</tr>
<tr>
<td>{R, T, U}</td>
<td>10,000</td>
<td>1,000</td>
<td>(T \bowtie U) \bowtie R</td>
</tr>
<tr>
<td>{S, T, U}</td>
<td>2,000</td>
<td>1,000</td>
<td>(T \bowtie U) \bowtie S</td>
</tr>
<tr>
<td>{R, S, T, U}</td>
<td>100</td>
<td>3,000</td>
<td>((T \bowtie U) \bowtie S) \bowtie R</td>
</tr>
</tbody>
</table>
Adapting DP to arbitrary join orders

- Previously, a larger relation set of size $n$ was computed by finding the optimal solution for size $n-1$ and joining another relation
  - The new relation is always placed to the right side of the join to form a **deep-left tree**, e.g., $((T \bowtie U) \bowtie S) \bowtie R$
  - Significantly reduced search space per step
– If any shape of join tree is possible, for computing an solution for subset of size \( n \), all combinations of smaller subsets have to be considered

• e.g., for \( n=5 \) consider
  – All subsets of size 4 with all valid subsets of size 1
  – All subsets of size 3 with all valid subsets of size 2
  – All subsets of size 2 with all valid subsets of size 3
  – All subsets of size 1 with all valid subsets of size 4
Based on the previous example:

- For \{R, S, T, U\} consider:
  - Triple with Single
    - \{S, T, R\} \bowtie \{U\}
    - \{R, S, U\} \bowtie \{T\}
    - \{T, U, R\} \bowtie \{S\}
    - \{T, U, S\} \bowtie \{R\}
  - Pair with Pair
    - \{T, U\} \bowtie \{R, S\}
    - \{R, T\} \bowtie \{S, U\}
    - \{S, T\} \bowtie \{R, U\}
  - Single with Triple
    - \{U\} \bowtie \{S, T, R\}
    - \{T\} \bowtie \{R, S, U\}
    - \{S\} \bowtie \{T, U, R\}
    - \{R\} \bowtie \{T, U, S\}

- Optimal solution for join order is not a deep-left tree, but \(R \bowtie (T \bowtie U) \bowtie S\)
  - Same intermediate result costs, but lower estimated execution costs as \textit{build} and \textit{probe} relations are ordered better (smaller to the left)
8.4 Dynamic Programming

• Summary Dynamic Programming
  – Guarantees “best” join order
  – Search effort still exponential, but strongly limited compared to exhaustive search
    • Complexity $O(2^n)$
    • Useful up to 10-15 joins only
  – Additional space consumption for storing the cost table
8.5 Greedy Strategy

• For larger joins dynamic programming will be too expensive…
  – Remember: $O(2^n)$

• Idea: Use a **Heuristic Greedy Algorithm**
  – Quickly construct only left-deep join trees
  – Result not necessarily optimal
8.5 Greedy Strategy

• Algorithm
  – **Start** with tree containing a join pair with cheapest costs
    • Smaller relation to the left
  – **While** not all relations on tree
    • Join current tree with relation promising cheapest join costs by attaching new relation to the right side of the tree
8.5 Greedy Strategy

• Find “good” join order restricted to left-deep join trees
• 4 Relations
  – \( \textbf{R} \) with attributes \( a \) and \( b \)
  – \( \textbf{S} \) with attributes \( b \) and \( c \)
  – \( \textbf{T} \) with attributes \( c \) and \( d \)
  – \( \textbf{U} \) with attributes \( d \) and \( a \)
  – Each relation has size of 1000
  – Following Table: \#dV(\text{Relation, attribute})
  
  • Number of distinct values for attributes and values

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{\#dV} & \text{R} & \text{S} & \text{T} & \text{U} \\
\hline
a & 100 & & & 50 \\
\hline
b & 200 & 100 & & \\
\hline
c & & 500 & 20 & \\
\hline
d & & & 50 & 1000 \\
\hline
\end{array}
\]
8.5 Greedy Strategy

- Start with \( T \bowtie U \) promising the smallest result
  - Cost 1000
- Consider \((T \bowtie U) \bowtie R\) and \((T \bowtie U) \bowtie S\)
  - \((T \bowtie U) \bowtie S\) better with costs 2000
- Join in \( R \)
  - Result \(((T \bowtie U) \bowtie S) \bowtie R\) with costs 3000

<table>
<thead>
<tr>
<th>#dV</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>100</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>b</td>
<td>200</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td>50</td>
<td>1000</td>
</tr>
</tbody>
</table>
8.6 Randomized Algorithms

• The algorithms so far have some **drawbacks**:  
  – DP algorithms are optimal, but very heavy weight  
    • Especially memory consumption is high  
  – Greedy heuristics are still only heuristics  
    • Will probably not find the optimal solution  
  – Both find a solution only after the complete search  
• Sometimes a **light-weight** algorithm is needed  
  – Low memory consumption  
  – Can stop when time runs out and still has an result  
  – Usually finds a good solution
8.6 Motivation

• Solutions to the join order problems can be seen as points in a solution space
  – Connect these point by a set of edges transforming the solutions into each other
  – Edges are called moves

• Randomized algorithms perform a random walk through the solution space along the edges
  – Random walk moves into the direction of better solutions
  – The walk can be stopped at any time, or if a (local) minimum is reached

Relational Database Systems 2 – Christoph Lofi - Benjamin Köhncke – Institut für Informationssysteme
8.6 Typical Moves

• If the search is restricted to **left-deep plans only**, the solutions are simple sequences of the relations \( R_1, \ldots, R_n \).

• Sequences can be transformed into each other by **two different moves**
  
  – **Swap**: exchange the positions of two arbitrary positions in the sequence
  
  – **3Cycle**: cyclic rotations of three arbitrary positions in the sequence
8.6 Typical Moves

• If also **bushy trees** are considered, add four moves can be applied:
  
  – **Commutativity**

  – **Associativity**
8.6 Typical Moves

- **Left Join Exchange**
  - $R_1 \bowtie R_3 \bowtie R_2$  
  - $R_2 \bowtie R_1 \bowtie R_3$

- **Right Join**
  - $R_1 \bowtie R_2 \bowtie R_3$  
  - $R_2 \bowtie R_3 \bowtie R_1$
8.6 Randomized Algorithms

• Typical algorithms are
  – Iterative Improvement
  – Simulated Annealing

• Each of these algorithms can return some result at all times, but can improve them with more time
  – i.e. optimize until a good enough solution is reached and stop
  – Either stop after a certain time span, or once a local minimum is detected
8.6 Iterative Improvement

• The set of solutions will **not contain** only a single global cost minimum reachable via all paths
  – But local minima are often sufficient
  – Remember: The optimizer does not need the optimal plan, but has to avoid crappy ones

• Simple **hill climbing** would
  – Start at some random point
  – Determine the neighboring node with smallest costs
  – Carry out the respective move
  – Until no smaller neighbor can be found
• But finding the minimum cost of all possible neighbors is expensive

• **Iterative improvement**
  – Starts at some random point
  – Randomly applies a move
  – Checks whether the new solution is less costly
    • If yes, start new iteration from current solution
    • If no, undo last move start new iteration
      – If no better move is found for several iteration, the solution is considered a local minimum; algorithm stops
Iterative improvement performs a random walk through the solution space by taking every possible improvement

- Quite efficient procedure
- Constant improvement during the walk
- No possibility to leave local minima, even if there is a global minimum near
  - Local minima may still have high cost
8.6 Simulated Annealing

• **Simulated annealing** is a refinement of iterative improvement
  
  – Moves do **not always** have to result in lower costs
  
  – Simulated annealing does not get caught in local minima so easily
8.6 Simulated Annealing

• The algorithm simulates the natural annealing process of crystals
  – simply stated: first **heating** and then **slowly cooling** a liquid will result in **crystallization**
  – One large crystal is of lower energy than several smaller ones combined
  – The system eventually reaches a state of minimum energy
    • The slower the cool down, the lower the final energy
8.6 Simulated Annealing

- Basic algorithm with cost function $c$
  - Start with a random tree and a high temperature
  - Apply a random move
  - Proceed with the new solution, if it is less expensive
  - **Proceed with the new solution anyway with a probability of**
    \[
    e^{-\frac{(c(\text{newsolution}) - c(\text{oldsolution}))}{\text{temperature}}}
    \]
  - Reduce temperature and apply new random move until an equilibrium is reached or the temperature is at freezing point
• It is very hard to determine the best parameters
  – Starting temperature, temperature reduction, stopping condition, etc.
• Often a two-phase version is used
  – Do iterative improvements for several random solutions
  – Use the least expensive result solution for a simulated annealing process
    • Since the initial solution is already better, the process can start with a lower temperature
8.6 Randomized Algorithms

• If the solution space cannot be enumerated, **randomized algorithms** are generally most appropriate

  – If **good solutions** are of primary importance use simulated annealing
  
  – If **short optimization times** are of primary importance use iterative improvement
  
  – Results for both are **far better** than in the heuristic case
8.6 Randomized Trees

• Problem: **How to generate a random join tree?**

• Generating a Random Join Tree has two phases
  – Generate a random **tree shape**
  – Generate a random **relation assignment** to the shape
8.6 Randomized Trees

• **Easiest Case:** Generate a Random Deep-Left Tree for \( n \) relations
  
  – Deep-Left Tree has only one shape
  – Relations can be assigned in any order (permutation) to the shape
  – Need to find a random permutation of the \( n \) relations
• Generating a real random permutation efficiently is tricky

  – We use a technique named 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

– Consider $S$ as being the set of all permutations of relations

– 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.
8.6 Randomized Trees

• An efficient **unranking** for permutations
  – Unranking between integers \([0,n!]\) and permutations
    • Based on factoradic numbers
  – Array *elements* contains relations \([R_1, R_2, R_3, \ldots, R_n]\)
  – Algorithm returns the *k*’s permutation of *s*

```plaintext
function permutation(k, elements) {
    for j = 2 to length(elements) {
        k := k / (j - 1);  // integer division
        swap elements[(k mod j) + 1] with elements[j]; }
    return elements;
}
```
Loop iterates over all elements of the array but the first

The red code fragments will generate a unique corresponding sequence of \( n \) integers
- First is in \( \{0, 1\} \), second in \( \{0, 1, 2\} \), third in \( \{0, 1, 2, 3\} \), ...
- Sequence depends on \( k \)

The green fragments swaps the current element with one of the previous elements based on the sequence

Result: Uniformly distributed random permutations of the element array

```plaintext
function permutation(k, elements) {
    for j = 2 to length(elements) {
        k := k / (j-1);
        swap elements[(k mod j)+ 1] with elements[j];
    }
    return elements;
}
```
8.6 Randomized Trees

- **More Difficult Case**: Generate arbitrarily shaped Random Trees for \( n \) relations
  - Generate a random shape
    - To be done..
  - Assign a random permutation of relations to the shape
    - Learned already
8.6 Randomized Trees

- How to generate a random tree shape?
  - Generating random trees is tricky

- Usually, not the tree itself is generated but an equivalent code word
  - Example: Dyck words (words of balanced number of characters, usually parenthesis)
    - e.g. (), (()), (())(), (()((()))), ...
    - There is an bijection between all Dyck words and all binary trees
• **Encoding Binary Tree with Dyck Words**
  
  – Traverse the tree in Pre-Order
    
    • Pre-Order
      – Visit node
      – Traverse left subtree
      – Traverse right subtree
    
    • Skip last leaf node
    
    • For each encountered *inner node*, write a ‘(‘
    • For each encountered *leaf node* write a ‘)’
    • For binary Dyck Representation, replace ‘(‘ with ‘1’ and ‘)’ with ‘0’
      
      – so called Zaks sequence

```
(()())() = 11010010
```
• Dyck words can be mapped to a triangle grid
  – Start at \((0,0)\); end at \((2n,0)\)
  – For each digit move one hop to the right, move up for 1 and
down for 0
• Number of paths described by Catalan Numbers

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

\(((())()) = 11010010\)
• **Unranking Binary Trees**
  
  – We try to create an unranking function for binary trees
    
    • Translates an integer number to a tree
      
      – i.e. generate a Zaks/Dyck sequence word from an integer
    
    – A tree with \( n+1 \) leaves has \( n \) inner nodes
      
      • For each relation in the join tree, one leaf is needed
      • We need Dyck words of the length \( 2n \) for \( n-1 \) relations
  
  – In the following: sketch of the canonical version
8.6 Randomized Trees

• For unranking, we work on a **triangle grid**
  – Number of possible paths from (0,0) to any position in the grid
    • \( p(i, j) = \frac{j+1}{i+1} \left( \frac{i+1}{2(i+j)+1} \right) \)
    • so called **Ballot number**
  – Number of possible paths from any position to (2n,0)
    • \( q(i, j) = p(2n-i, j) \)

• **Algorithm Sketch** (without proofs and derivation...)
  – Work on a triangle grid
  – Generate random tree rank \( r \) from \([0, C(n)]\)
    • Maximum number of possible trees (again) expressed by Catalan numbers
  – Start on (0,0)
– While **number of paths** from current point to \((2n,0)\) exceeds rank \(r\) (i.e. \(q(i,j)>r\)), **or** baseline \((x,0)\) is reached move a step **top-right**
  • i.e. go from \((i,j)\) to \((i+1,j+1)\)
  • Write an ‘(‘ or ‘1’ for each upward movement

– Otherwise
  • Write an ‘)’ or ‘0’, move to the **lower-right**
    – i.e. go from \((i,j)\) to \((i+1,j-1)\)
  • **Subtract** number of paths of the overlaying coordinate (i.e. the one which we had reached if we had gone top-right) from the **rank** and **resume** moving top-right
    – i.e. if we just went from \((i,j)\) to \((i+1,j-1)\), subtract number of paths from \((i+1,j+1)\)

– **Stop** when \((2n,0)\) is reached
8.6 Randomized Trees

- Example: \( n=4 \) (join trees for 3 relations)
  - \( C(4) = 14 \)
  - Generate random rank in \([0,14]\)

  - e.g. \( r=9 \)

---

Result:
8.6 Randomized Trees

- Start at (0,0), rank \( r=9 \)
- Number of Paths at (0,0): 14
  - \( q=14 > r=9 \) ⇒ Move up
- Reach (1,1)

\[
\begin{align*}
\text{Result:} & \\
( & \\
1 & )
\end{align*}
\]
8.6 Randomized Trees

- Position (1,1), rank $r=9$
- Number of Paths at (1,1): 14
  - $q=14 > r=9 \Rightarrow$ Move up
- Reach (2,2)

Result: (11)
8.6 Randomized Trees

- Position (2,2), rank $r=9$
- Number of Paths at (2,2): 9
  - $\text{not } (q=9 > r=9) \Rightarrow \text{Move down}$
  - Subtract $q(3,3)=4$ from rank $r; r:=5$
- Reach (3,1)

Result: 
(()

110
8.6 Randomized Trees

- Position (3,1), rank \( r=5 \)
- Number of Paths at (3,1): 5
  - not (\( q=5 \) > \( r=5 \)) \( \Rightarrow \) Move down
  - Subtract \( q(4,2)=3 \) from rank \( r; r:=2 \)
- Reach (4,0)

Result: 
\((()\)

1100
8.6 Randomized Trees

- Position (4,0), rank $r=2$
- Number of Paths at (4,0): 2
  - not $(q=2 \geq r=2)$, but reached base line ⇒ Move up
- Reach (5,1)

Result: 

11001
8.6 Randomized Trees

- Position (5,1), rank \( r = 2 \)
- Number of Paths at (5,1): 2
  - not (\( q = 2 > r = 2 \)) ⇒ Move down
  - Subtract \( q(6,2) = 1 \) from rank \( r; r := 1 \)

- Reach (6,0)

Result:
((()())
110010
8.6 Randomized Trees

- Position (6,0), rank $r=1$
- Number of Paths at (6,0): 1
  - not $(q=2 \geq r=2)$ but reached base line ⇒ Move up
- Reach (7,1)

Result:

```
(()())()
1100101
```
8.6 Randomized Trees

- Position (7,1), rank $r=1$
- Number of Paths at (7,1): 1
  - not ($q=1 \geq r=1$) ⇒ Move down
  - Try to subtract $q(8,2)=\text{NaN}$ from rank $r$...

- Reach (8,0)

Result:
(()(()))
11001010
8.6 Randomized Trees

- **In General:**
  - **Red:** Number of possible paths \( q(i,j) \)
  - **Green:** Interval of remainder ranks choose the annotated path
8.6 Randomized Trees

• Example:
  \[ R = 11 \]
• Canonical unranking performs badly
  – Generating and working with Catalan numbers is expensive
    • $C(5000)$ has already 2000 digits…
• We need algorithms which work in space/time complexity of $O(n)$
  – Generate Dyck words/Zaks sequences directly without Catalan numbers
  – There are already such wonderful algorithms
    • Arnold and Sleep Algorithm
    • Atkinson and Sack Algorithm
    • Martin and Orr Algorithm
    • Johnson and Zergling Algorithm
    • …
8 Join Order Optimization

Basic join order optimization
Join cost and size estimations
Left-deep join trees
Dynamic programming
Greedy strategy
Randomized algorithms
Outlook: Transaction Processing

Basic database transactions
The ACID principle
Transaction schedules
Conflict serializability
Locking schedulers