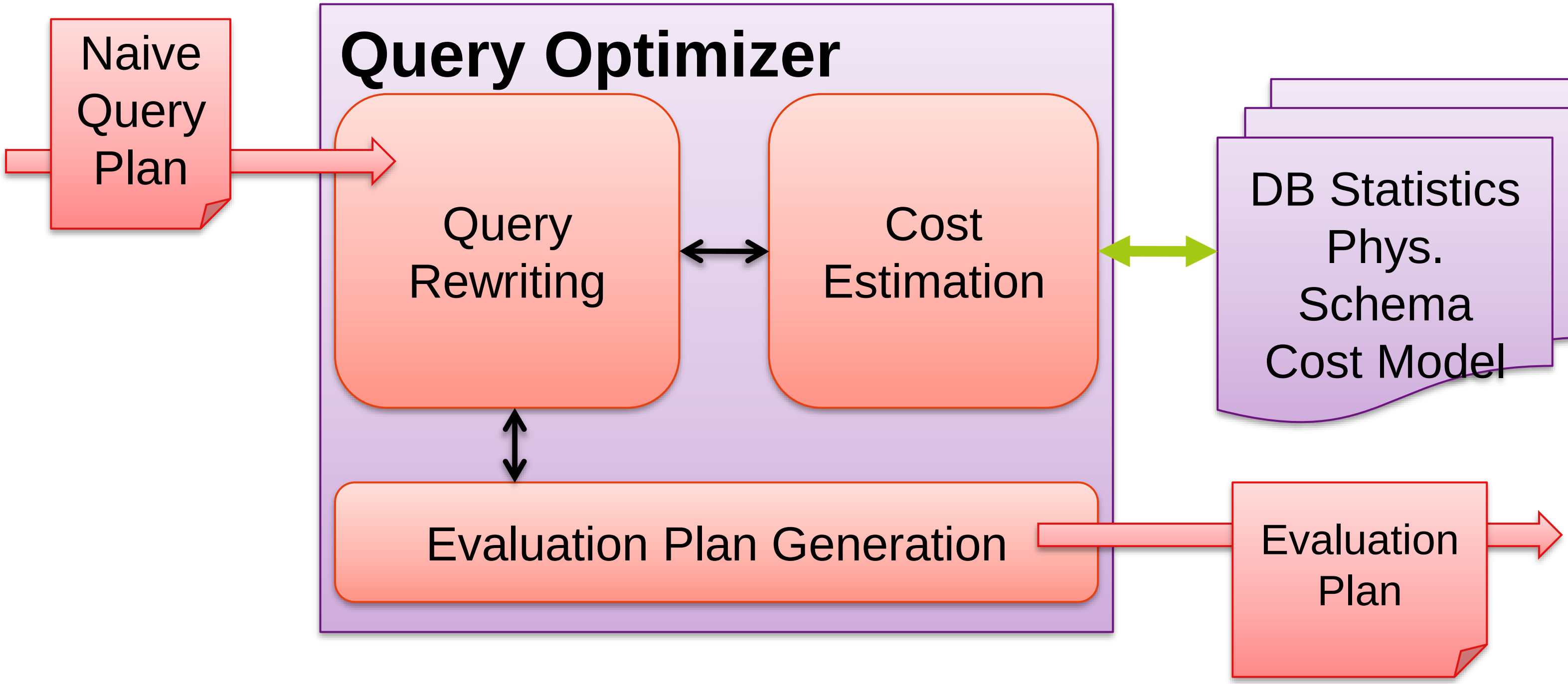
Information & Data Modelling

Query Optimization – Join Orders

Christoph Lofi

Query Optimization

- Query optimizer rewrites the naïve (canonical) query plan into a more efficient evaluation plan

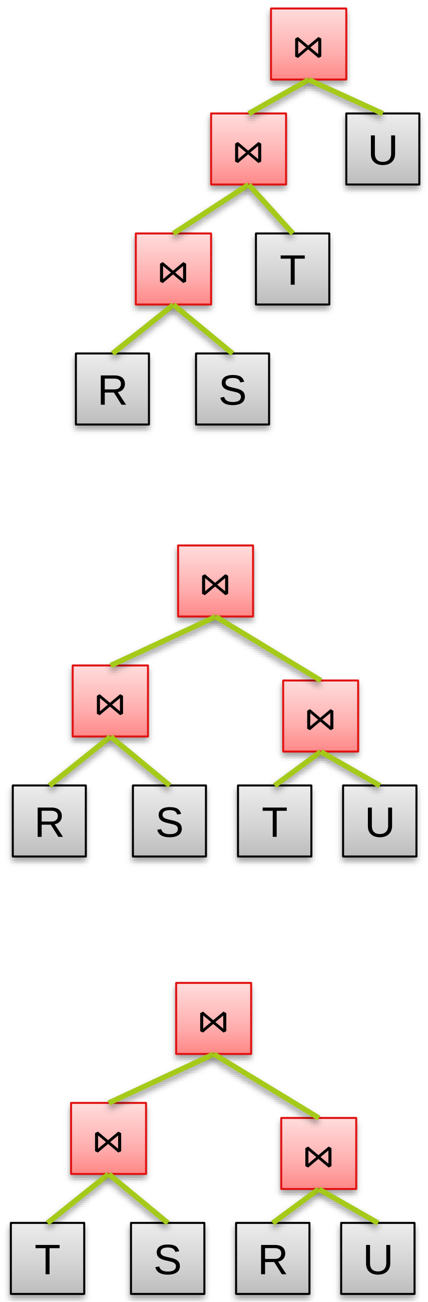


Join Order Optimization

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

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: $R \bowtie S \bowtie T \bowtie U$



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 ≤ i ≤ 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."

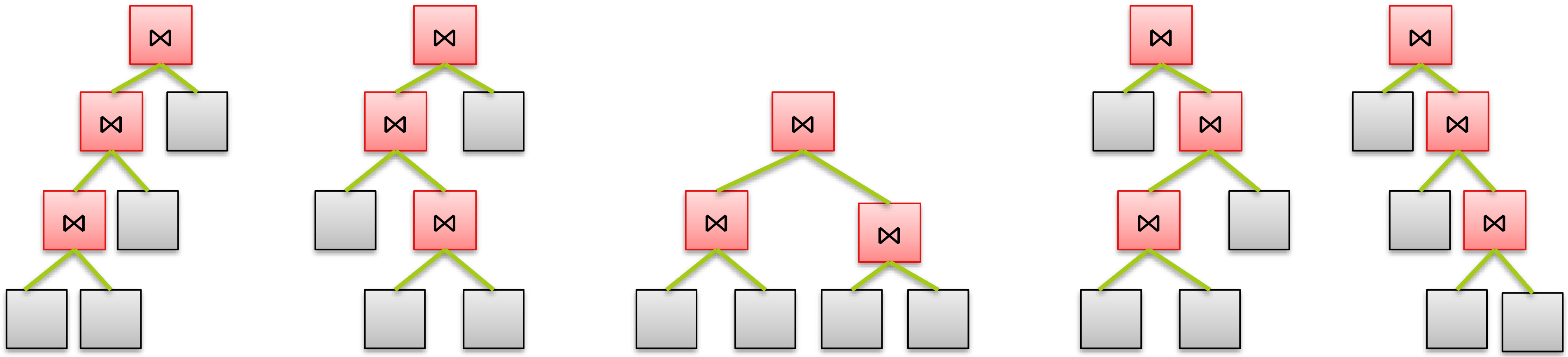
Shapes of Join Trees

- This number sequence is called **Catalan Numbers**
  - Named after Belgian mathematician Eugène Charles Catalan (1814–1894)
  - Can be rewritten as

$$T(n) = C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

Shapes of Join Trees

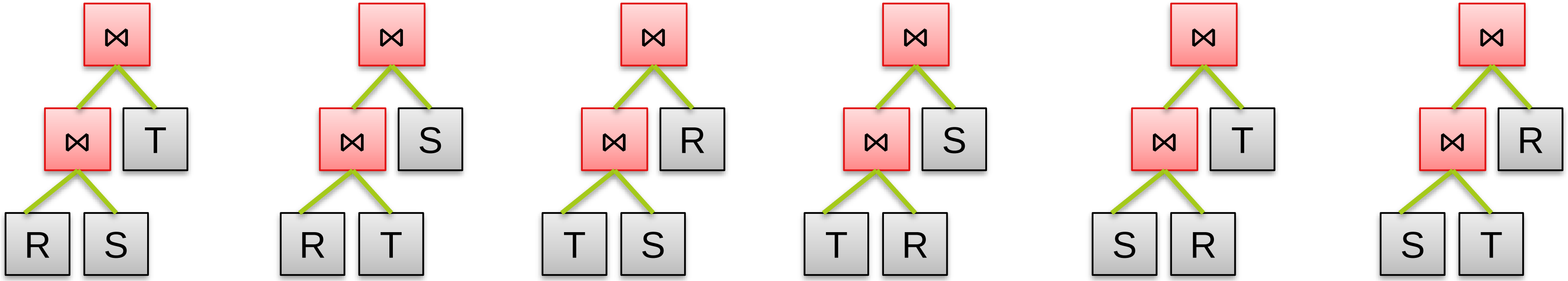
- **Example**: Shapes for *n=4*



- **Example:** The 22 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….

Shapes of Join Trees

- 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!

n = 2 : 1*2! = 2	*n* = 6 : 42*6! = 30,240
n = 3 : 2*3! = 12	*n* = 9 : 1,430*12! = 518E6
n = 4 : 5*4! = 120	*n* = 12 : 58,786*12! = 28E12
n = 5 : 14*5! = 1 ,680	*n* = 15 : 2,674,440*15! = 3.49E18

Basic Join Order Optimization

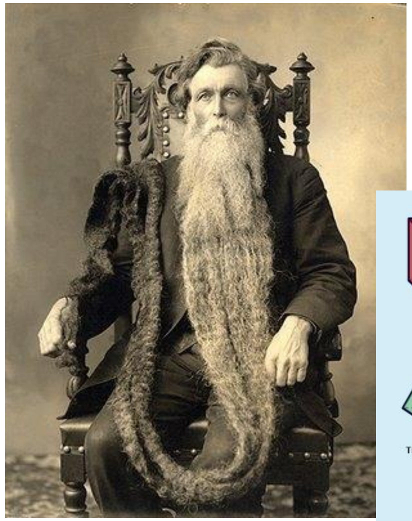
- Finding the "most efficient" **join tree** and **join implementation** is a challenging problem
  - Number of possible join trees grows extremely with number of join relations
    - Problem was shown to be **NP-hard** in the general case
    - ***O(n!)***, with *n* as number of join relations
    - Estimating cost of all trees is not feasible for larger joins
  - Some join implementations are **asymmetric**
    - Performance varies greatly depending on relation order
      - E.g., BNL join
- Query optimizer has to find a **good plan** in **sensible time**

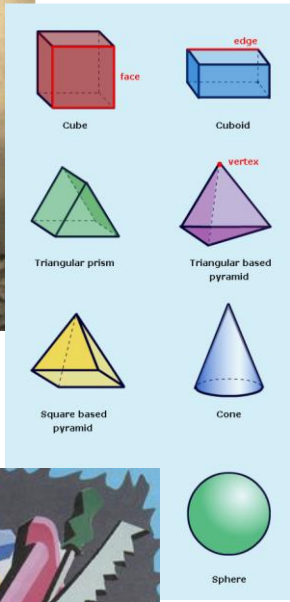
Basic Join Order Optimization

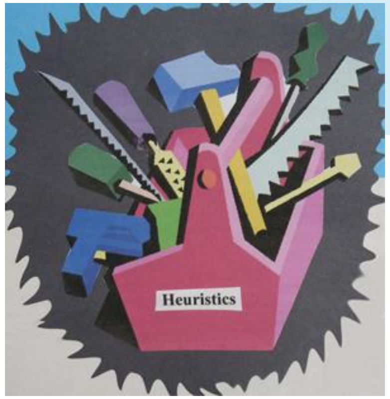
- Naming convention
  - Left: **Build Relation**
  - Right: **Probe Relation**
- Desirable Join Cases
  - Attention: Role (inner/outer relation) of build and probe depends on chosen algorithm
  - **Block Nested Loop Join**
    - Build relation is in **inner loop**, probe relation is in **outer loop**
    - Good when build relation **significantly** smaller than probe
  - **“Single Pass Join”**
    - Deteriorated case if BNL Join
    - **Block Nested Loop Join** for which 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 (probe relation = the one where you are “probing” for a match)
    - Build relation small (because you probe for each row in build relation)

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





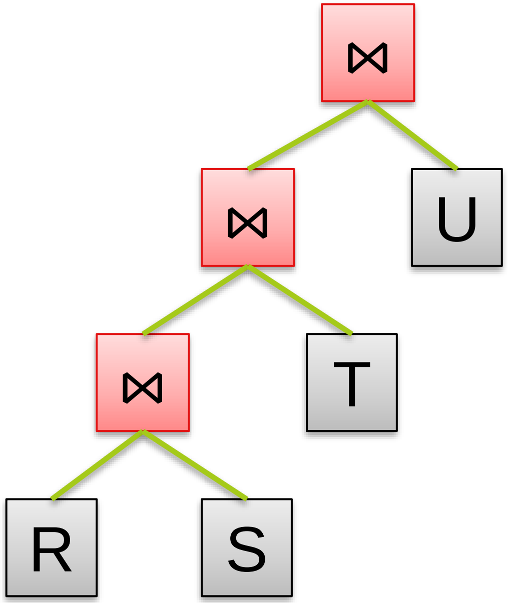


Join Metrics

- Remember the join costs from last week:
  - Size Estimate:
    - **$| R \bowtie S | = |R| * |S| \,/\, \Pi_i\, (\max(\#dv(R, A_i), \#dv(S, A_i))$**
  - Block Access Costs:
    - Block-Nested-Loop Join
      - $Costs_{BNL}\,(R \bowtie S) = b_R + (b_R * b_S) + Costs_{Result}(R \bowtie S)$
    - **Indexed Block-Nested Loop Join**
      - $Costs_{IXL}\,(R \bowtie S) = b_R + (|R| * (C_{ix}+1)) + Costs_{Result}(R \bowtie S)$
    - **Merge-Join**
      - Only when sorted correctly!
      - $Costs_{SMJ}\,(R \bowtie S) = b_R + b_S + Costs_{Result}(R \bowtie S)$

Left-deep Join Trees

- 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 leafs



Left-deep Join Trees

- Left-deep join trees incorpate well with most **join algorithms** as they aim for decreasing the size of 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

Left-deep Join Trees

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

n	All join trees	Left-deep trees
3	12	6
6	30,240	720
9	518E6	362,880
12	28E12	479E6

- But…
  - **Still** a considerable amount (impractical for >15 joins)
  - Parallel execution of joins is **not possible!**

Exploring Join Order Options

- 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

Dynamic Programming

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

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!

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.
    - We use intermediate result set size in the following
  - The **expression** (i.e. tree shape and assignment) which produced the lowest costs

Subset	Size	Costs	Expression
…			
{R,S,T,U}	2,500	25,750	(U⋈ (S⋈T))⋈R
…			

Dynamic Programming

- Table is build **inductively** on the subsets of relations
- ***Claim***:
  - Table always contains **join expressions** with **lowest costs** for given **relation subsets**

Dynamic Programming

- **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** $\{R_a, R_b\}$, the table contains one row
    - Estimated size as described in previous section
      - In the following examples, the size is simply given; it results from the reduction factors of the join which are not shown in the example
    - Costs 0 ($\Rightarrow$ 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

Subset	Size	Costs	Expression
$\{R_1\}$	2000	0	R_1
$\{R_2\}$	1000	0	R_2
$\{R_1, R_2\}$	500	0	$R_2 \bowtie R_1$

Dynamic Programming

- **Induction**:
  - For each **relation subset of size *n*** $\mathbf{Rs}$={R_a, R_b, …, R_z}, create a table row
  - Find two subsets $\mathbf{Rs_1} \cup \mathbf{Rs_2} = \mathbf{Rs}$ within the table such that Cost($\mathbf{Rs_1} \bowtie \mathbf{Rs_2}$) are minimal
    - For **deep-left trees**, only subsets with $|\mathbf{Rs_1}|=n-1$ and $|\mathbf{Rs_2}|=1$ need to be considered

Dynamic Programming

- Fill row with
  - Rs as subset identifier
  - Estimated size $|\mathbf{Rs_1} \bowtie \mathbf{Rs_2}|$
  - Estimated costs $\text{Cost}(\mathbf{Rs_1} \bowtie \mathbf{Rs_2})$
  - Concatenation of the expressions of $\mathbf{Rs_1}$ and $\mathbf{Rs_2}$
    - For deep-left join trees, always place expression of $\mathbf{Rs_1}$ to the left (no nesting)
    - otherwise, place expression with smaller result size to the left

Subset	Size	Costs	Expression
$\{R_1\}$	2000	0	R_1
$\{R_2\}$	1000	0	R_2
$\{R_3\}$	**3000**	**0**	**R_3**
$\{R_1, R_2\}$	**500**	**0**	**$R_2 \bowtie R_1$**
$\{R_1, R_3\}$	1200	0	$R_1 \bowtie R_3$
$\{R_2, R_3\}$	1800	0	$R_2 \bowtie R_3$
$\{R_1, R_2, R_3\}$	200	500	$(R_2 \bowtie R_1) \bowtie R_3$

Here:
$\mathbf{Rs_1} = \{R_1, R_2\}$
$\mathbf{Rs_2} = \{R_3\}$

Dynamic Programming

- Find optimal join order restricted to **left-deep join trees**
  - **WE ONLY CONSIDER NATURAL JOINS**
- 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(Relation, attribute)**
    - Number of distinct values for attributes and relations

Note: These are made-up numbers and can be used for the size estimation formula
| R⋈S | = |R|*|S| / Π_i (max(#dv(R, A_i), #dv(S, A_i))

#dV	R	S	T	U
a	100			50
b	200	100		
c		500	20	
d			50	1000

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

Subset	Size	Costs	Expression
{R}	1,000	0	R
{S}	1,000	0	S
{T}	1,000	0	T
{U}	1,000	0	U

{R, S}	5,000	0	R ⋈ S
{R, T}	1 M	0	R ⋈ T
{R, U}	10,000	0	R ⋈ U
{S, T}	2,000	0	S ⋈ T
{S, U}	1 M	0	S ⋈ U
{T, U}	1,000	0	T ⋈ U

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**

Subset	Size	Costs	Expression
{R}	1,000	0	R
{S}	1,000	0	S
{T}	1,000	0	T
{U}	1,000	0	U
{R, S}	5,000	0	$R \bowtie S$
{R, T}	1 M	0	$R \bowtie T$
{R, U}	10,000	0	$R \bowtie U$
{S, T}	2,000	0	$S \bowtie T$
{S, U}	1 M	0	$S \bowtie U$
{T, U}	1,000	0	$T \bowtie U$
{R, S, T}	10,000	2,000	$(S \bowtie T) \bowtie R$
{R, S, U}	50,000	5,000	$(R \bowtie S) \bowtie U$
{R, T, U}	10,000	1,000	$(T \bowtie U) \bowtie R$
{S, T, U}	2,000	1,000	$(T \bowtie U) \bowtie S$

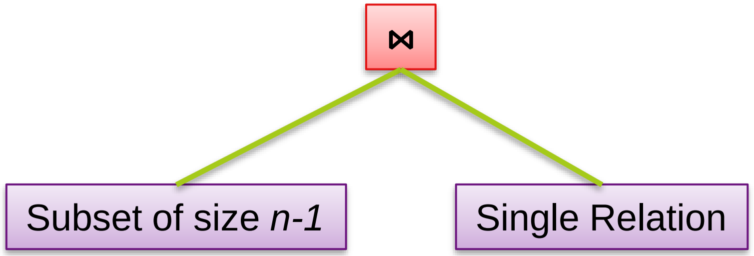
Dynamic Programming

- 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⋈T)⋈R)⋈U : 12,000
    - ((R⋈S)⋈U)⋈T : 55,000
    - ((T⋈U)⋈R)⋈S : 11,000
    - ((T⋈U)⋈S)⋈R : 3,000

Subset	Size	Costs	Expression
{R}	1,000	0	R
{S}	1,000	0	S
{T}	1,000	0	T
{U}	1,000	0	U
{R, S}	5,000	0	R ⋈ S
{R, T}	1 M	0	R ⋈ T
{R, U}	10,000	0	R ⋈ U
{S, T}	2,000	0	S ⋈ T
{S, U}	1 M	0	S ⋈ U
{T, U}	1,000	0	T ⋈ U
{R, S, T}	10,000	2,000	(S⋈T)⋈R
{R, S, U}	50,000	5,000	(R⋈S)⋈U
{R, T, U}	10,000	1,000	(T⋈U)⋈R
{S, T, U}	2,000	1,000	(T⋈U)⋈S

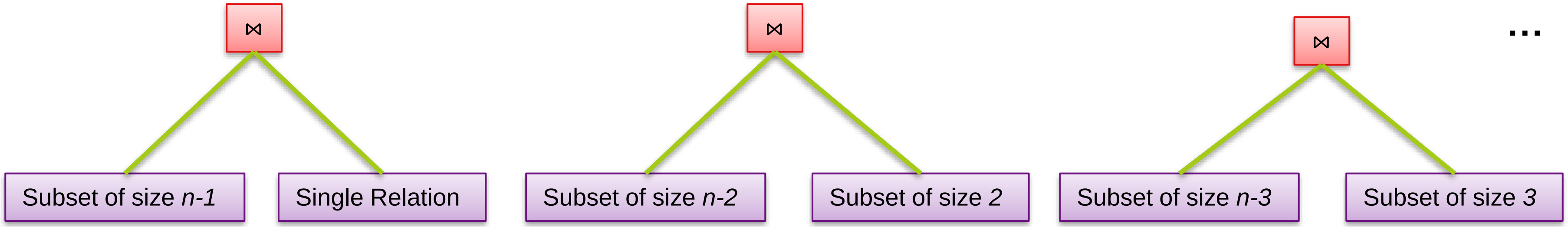
Dynamic Programming

- 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⋈U)⋈S)⋈R**
    - Significantly reduced search space per step



Dynamic Programming

- If any shape of join tree is possible, for computing a 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



Dynamic Programming

- Based on the previous example:
  - For {R, S, T, U} consider:
    - Triple with Single
      - {S, T, R}⋈ {U}
      - {R,S,U} ⋈ {T}
      - {T,U,R} ⋈ {S}
      - {T,U,S} ⋈ {R}
    - Pair with Pair
      - {T, U} ⋈ {R, S}
      - {R, T} ⋈ {S, U}
      - {S, T} ⋈ {R, U}
    - Single with Triple
      - {U} ⋈ {S, T, R}
      - {T} ⋈ {R,S,U}
      - {S} ⋈ {T,U,R}
      - {R} ⋈ {T,U,S}

  - Optimal solution for join order is not a deep-left tree, but **R⋈((T⋈U)⋈S)**
    - Same intermediate result costs, but lower estimated execution costs as **build** and **probe** relations are ordered better (smaller to the left)

Dynamic Programming

- **Summary Dynamic Programming**
  - Guarantees "**best**" join order
    - Use of "" because it is the best order based on our very rough cost estimates
  - 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

Greedy Join Strategies

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

- Idea: **Heuristic Greedy Join Order Algorithm**
  - Quickly construct only left-deep join trees
  - Result not necessarily optimal

Greedy Join Strategies

- **Algorithm**
  - **Start** with tree containing a join pair with cheapest costs
    - Smaller relation to the left
  - **While** not all relations in tree
    - Join current tree with relation promising cheapest join costs by attaching new relation to the right side of the tree

Greedy Join Strategies

- Find "good" 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(Relation, attribute)**
    - Number of distinct values for attributes and values

#dV	R	S	T	U
a	100			50
b	200	100		
c		500	20	
d			50	1000

Greedy Join Strategies

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

Note: These are made-up numbers and can be used for the size estimation formula
| R⋈S | = |R|*|S| / Π_i (max(#dv(R, A_i), #dv(S, A_i))

#dV	R	S	T	U
a	100			50
b	200	100		
c		500	20	
d			50	1000

Randomized Algorithms

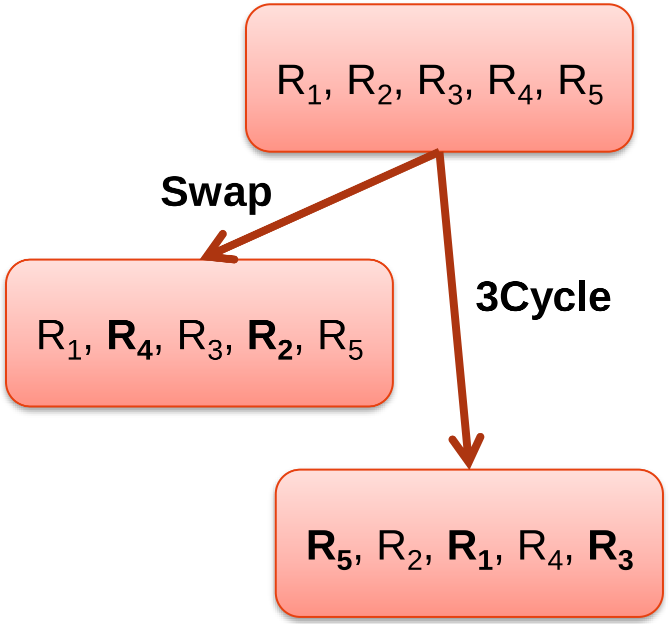
- These algorithms so far have some **drawbacks**:
  - DP algorithms are optimal, but very heavy weight
    - Especially memory consumption is high
  - Greedy heuristics are still only extremely simple heuristics
    - Will probably not find the optimal solution
- Sometimes a **light-weight** algorithm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

Randomized Algorithms

- 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

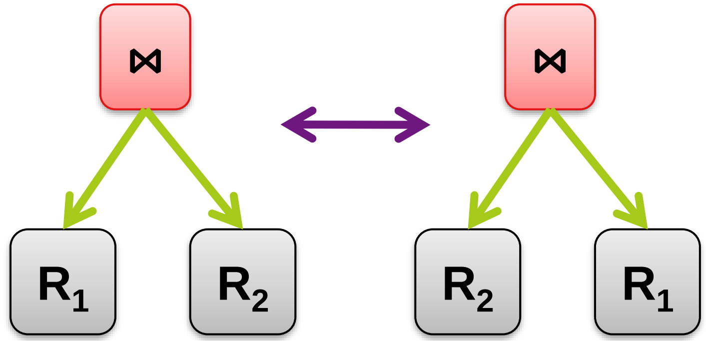
Randomized Algorithms

- If the search is restricted to **left-deep plans only**, the solutions are simple sequences of the relations $\mathbf{R_1, \ldots, R_n}$
- 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

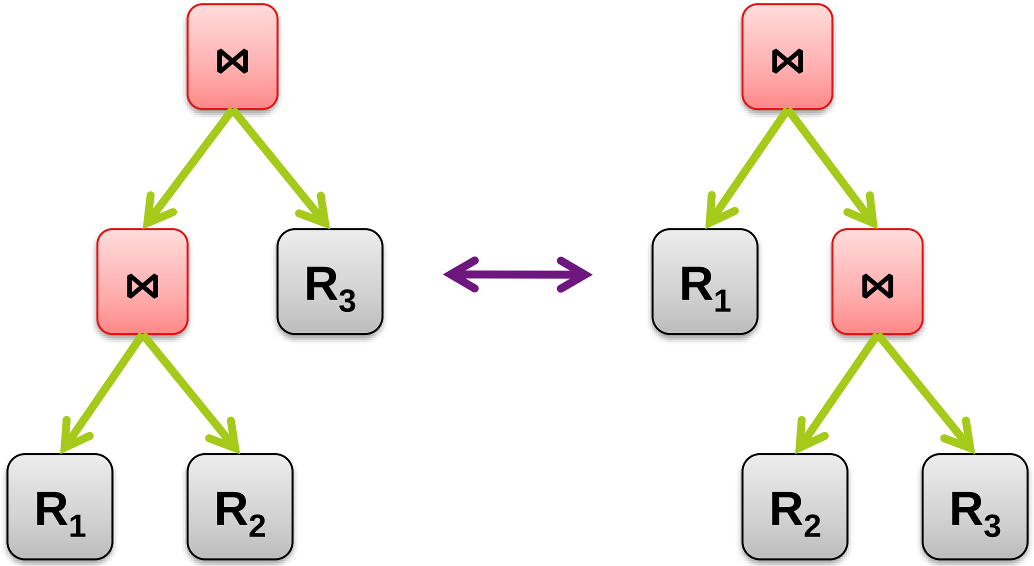


Randomized Algorithms

- If also **bushy trees** are considered, add four additional moves:
  - Commutativity

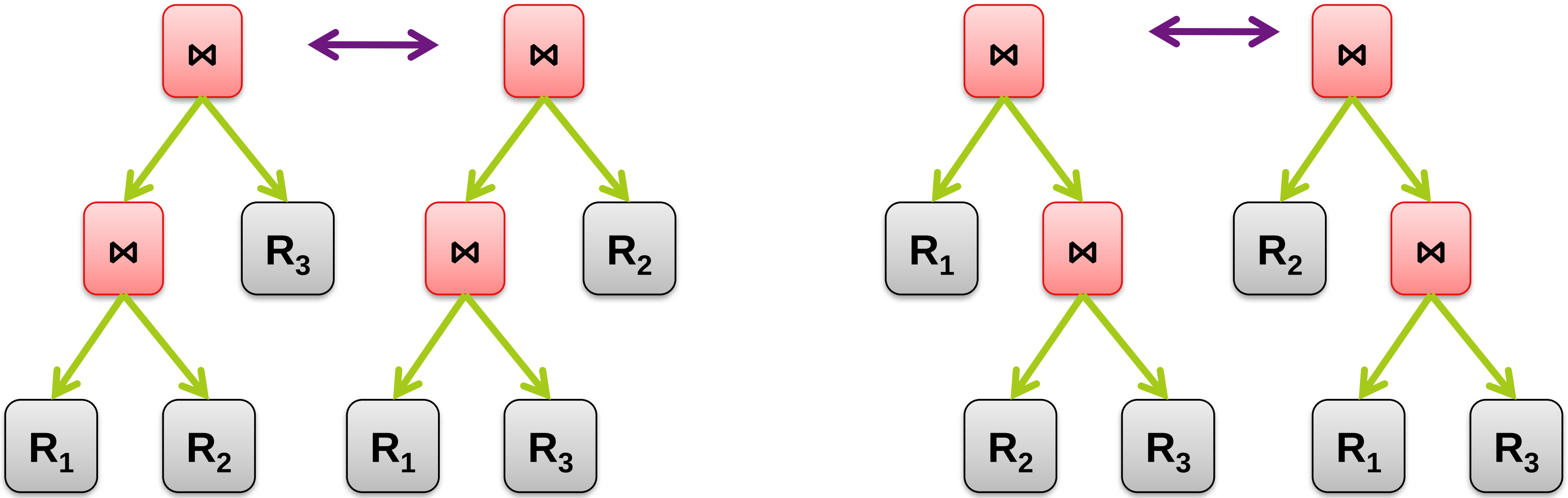


  - Associativity



Typical Moves

- Left Join Exchange Right Join Exchange

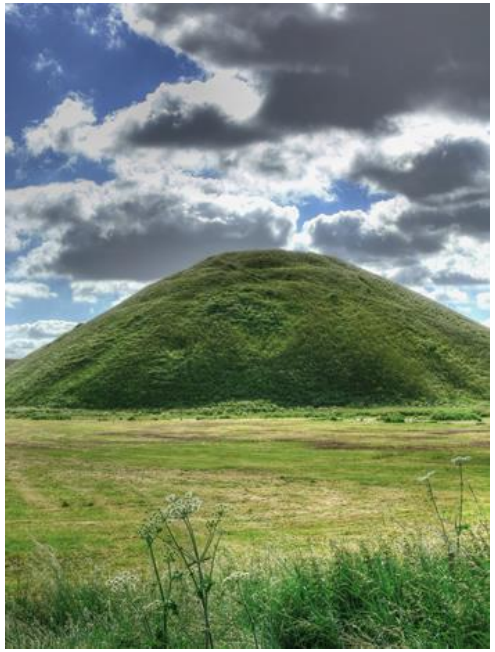


Randomized Algorithms

- 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

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



Iterative Improvement

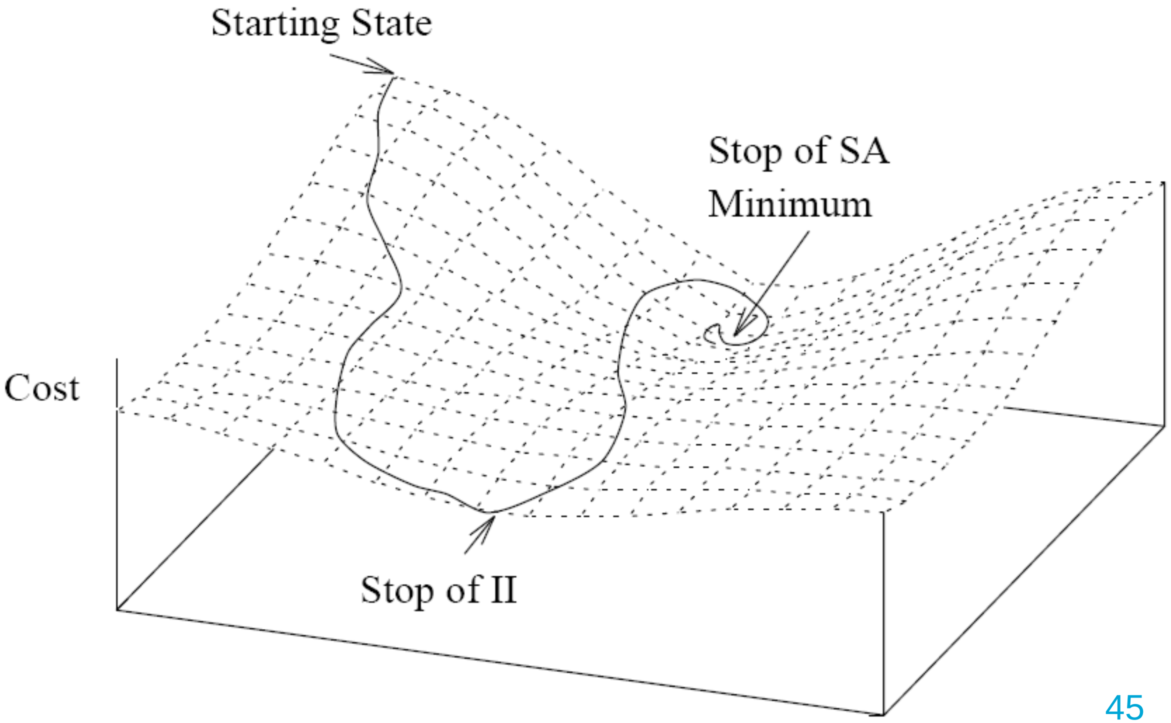
- But finding the minimum cost of **all possible** neighbors is expensive
- **Iterative improvement**
  - Starts at some random point
  - Randomly applies a move
  - Checks whether the new solution is less costly
    - If yes, start new iteration from current solution
    - If no, undo last move and start new iteration
      - If no better move is found for several iterations, the solution is considered a local minimum; algorithm stops

Iterative Improvement

- 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

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



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

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**

$$e^{-\left(\frac{c(newsolution)-c(oldsolution)}{temperature}\right)}$$

  - Reduce temperature and apply new random move until an equilibrium is reached or the temperature is at freezing point

Simulated Annealing

- 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

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

Summary

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

Information and Data Management

See you next time!

TU Delft WIS Web Information Systems