Lecture #07: Join Ordering: Bottom-Up

15-799 Special Topics in Databases: Query Optimization (Spring 2025)
https://15799.courses.cs.cmu.edu/spring2025/
Carnegie Mellon University
Prepared By: Jiaying Li & Guan-Ru Chen

1 Background

Joins are the backbone of query processing. They occur in nearly every query, and they can affect query runtime dramatically. But most queries with joins involve only two tables.

However, there are ridiculous outlier queries with hundreds or thousands of tables in systems like SAP and Tableau. For instance, the largest known query joins 5000 tables in SAP due its layering of views. Most complex queries are computer generated (i.e., not written by humans).

As a result, an optimizer must be able to handle the common-case "easy" queries but still support the occasional freak queries.

2 Adaptive Join Optimization

In *adaptive join optimization* [2], instead of using a single search strategy for all queries, the optimizer selects a suitable algorithm for each query based on its logical complexity. Figure 1 shows an example:

- **Small Queries:** For small queries, which make up the bulk of most workloads, the goal is to find the optimal join order, achievable through dynamic programming.
- **Medium-Sized Queries:** For medium-sized queries, while optimality can't be guaranteed, the optimizer combines dynamic programming with *search space linearization* to get close to optimal.
- **Larger Queries:** For larger queries, finding the best plan is impractical, so the focus shifts to achieving good results, using a greedy approach to ensure that the plan quality degrades gracefully.

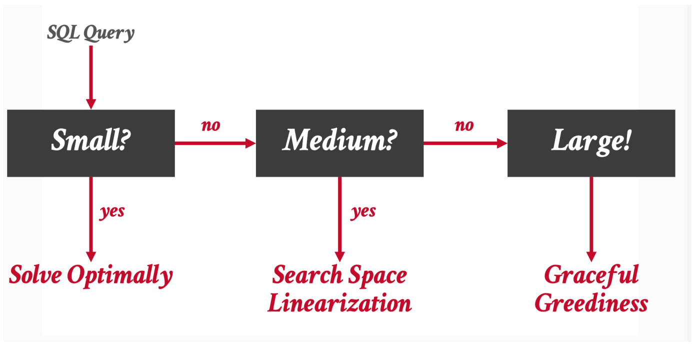
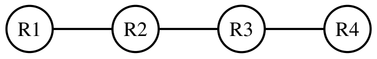


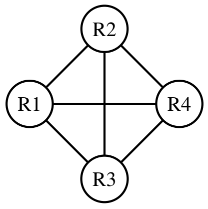
Figure 1: Adaptive Optimization Decision Tree

The logical complexity of a query plan is not determined solely by the number of relations it references. Instead, the complexity depends on the *structure* of its graph, specifically how different relations join with each other.

2.1 Measuring Complexity through Query Graph Structures



(a) Chain Graph: Linear Join Precedence



(b) Clique Graph: Fully Connected Relations

Figure 2: Query Graph Structures

We can use the complexity and size of the **DPHyp** dynamic programming (DP) algorithm to measure a query's complexity. A query's complexity depends on the structure of its query graph. Two important structures are:

- **Chain Graph:** (Figure 2a)
  - Each relation connects to at most two other relations.
  - Allows for linear ordering of join precedence.
  - Represents the **best-case** scenario.
  - DPHyp complexity is $O(n^3)$ with a DP table size of $O(n^2)$: feasible to solve queries with up to **1,000** relations exactly.
- **Clique Graph:** (Figure 2b)
  - Every relation connects to all other relations.
  - These queries are rare but difficult to optimize.
  - Represents the **worst-case** scenario.
  - DPHyp complexity is $O(3^n)$ with a DP table size of $O(2^n)$: exact solutions limited to queries with about **14** relations.

2.2 Small Queries

Small queries are defined to have a DP table with up to 10,000 entries, in which case the *DPHyp* algorithm can efficiently generate the optimal join ordering. This algorithm adapts to the query's graph structure, ensuring complete and minimal enumeration of all possible join orders without cross products.

2.2.1 DHyp: Basic Algorithm

The *DHyp* algorithm is a dynamic programming-based approach for join optimization. It serves as the foundation for *DPHyp*. The algorithm works as follows:

- Enumerate all connected subgraphs of the query graph.
- For each subgraph, enumerate all other connected subgraphs that are disjoint but still connected to it. Start with one node and expand recursively by following edges.

Figure 3 illustrates the recursive expansion process. Starting from R_3, the algorithm enumerates all connected subgraphs by adding R_2, then expands further by incorporating R_1 or R_4. Each subgraph grows by following edges while maintaining connectivity, ensuring systematic exploration of all valid join plans.

To explore join orders more effectively, the optimizer can consider *node groupings* instead of individual nodes by using *hypergraphs*.

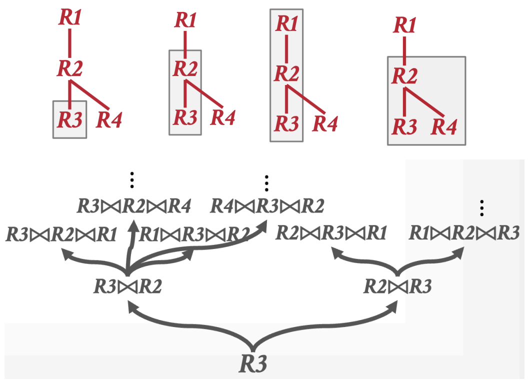


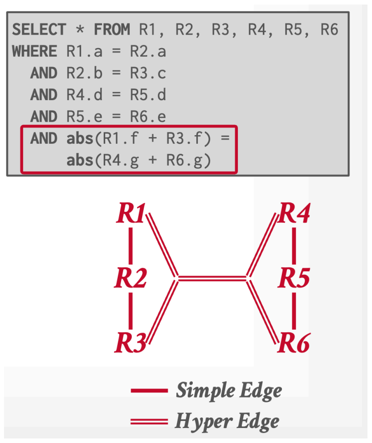
Figure 3: DHyp Example

2.2.2 Hypergraphs

A hypergraph is defined as a pair $H = (V, E)$ such that:

- V is a non-empty set of nodes.
- E is a set of hyperedges, where a hyperedge is an unordered pair (u, v) of non-empty subsets of V $(u \subset V, v \subset V)$ with the additional condition that $u \cap v = \emptyset$.

This representation allows the search algorithm to consider *node groupings* instead of individual nodes, which is crucial for efficiently optimizing the join orders of complex queries.



```
SELECT * FROM R1, R2, R3, R4, R5, R6
WHERE R1.a = R2.a
  AND R2.b = R3.c
  AND R4.d = R5.d
  AND R5.e = R6.e
  AND abs(R1.f + R3.f) =
      abs(R4.g + R6.g)
```

Figure 4: Hypergraph Representation and Query Example

Figure 4 shows an example query with six relations. In the hypergraph representation, the left subgraph $\{R_1, R_2, R_3\}$ forms a chain, and the right subgraph $\{R_4, R_5, R_6\}$ follows a similar structure. These two groups are connected by a *hyperedge*, reflecting the complex predicate that spans multiple relations. This representation allows the optimizer to first optimize each group independently before determining how to join them while respecting the hyperedge constraint.

But because hyperedges represent $n : m$ relationships, adding them to a subgraph connects multiple nodes at once. This introduces new challenges for using the DHyp algorithm:

- When expanding from a subset of relations, it is necessary to ensure that all nodes in a hyperedge are included in the subgraph.
- Careful selection of expansion order is required to maintain the correctness of the dynamic programming algorithm.

For example, Figure 4 shows that the left portion $\{R_1, R_2, R_3\}$ and the right portion $\{R_4, R_5, R_6\}$ are connected by a hyperedge. If R_4 is added prematurely, it causes R_6 to become disconnected, violating the DHyp algorithm's constraints. Therefore, the expansion order must be carefully controlled to ensure that the full hyperedge is considered at each step. To address this limitation, **DPHyp** integrates hypergraph structures into *DHyp*'s optimization process.

2.2.3 DPHyp: Dynamic Programming Hypergraph Algorithm

DPHyp models the query as a hypergraph and incrementally expands it to enumerate new plans. It is implemented in several modern database systems (e.g., HyPer, Umbra, DuckDB). In *DPHyp*, the hypergraph is traversed in a fixed order, recursively producing larger connected subgraphs. Key aspects include:

- Incrementally expand connected subgraphs by considering new nodes in the neighborhood of a subgraph. Both the primary connected subgraph and its connected complement are created through recursive graph traversals.
- Identify reachable nodes from a subgraph, excluding certain nodes based on constraints. To avoid redundant exploration, some nodes are forbidden during traversal. Specifically, when a function performs a recursive call, it forbids all nodes that it will investigate itself.
- Treat hypernodes as single instances when choosing subsets. Hyperedges are interpreted as $n : 1$ edges, leading from n nodes on one side to a single canonical node on the other side. This reduces the complexity of join enumeration by allowing the optimizer to process grouped relations instead of individual ones.

DPHyp handles complex join predicates and non-inner joins. The extension to hypergraphs enables the optimizer to process queries with non-inner joins much more efficiently than before, even for queries with binary join predicates. Furthermore, *DPHyp* is designed to handle complex join predicates effectively, allowing for robust optimization in a broader range of query scenarios.

2.3 Medium Queries

For queries with more than 100 relations, the search strategy depends on the structure of the query graph. The primary goal is to simplify the problem by converting every query into a chain query. Through linearization, the optimizer only needs to consider *associativity* and not commutativity when enumerating join orderings.

2.3.1 Search Space Linearization

Queries are simplified into chain queries through *Search Space Linearization*. Assume that the order of relations in the optimal plan is known (see Section 2.3.2). A polynomial DP algorithm then generates an optimal plan from this linearization, where the algorithm combines optimal solutions for sub-chains of increasing size.

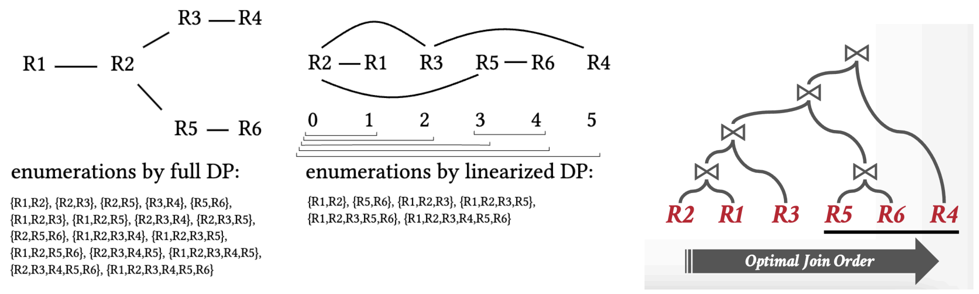

R3 — R4
R1 — R2
R5 — R6
enumerations by full DP:
{R1,R2}, {R2,R3}, {R2,R5}, {R3,R4}, {R5,R6}, {R1,R2,R3}, {R1,R2,R5}, {R2,R3,R4}, {R2,R3,R5}, {R2,R5,R6}, {R1,R2,R3,R4}, {R1,R2,R3,R5}, {R1,R2,R5,R6}, {R2,R3,R4,R5}, {R1,R2,R3,R4,R5}, {R2,R3,R4,R5,R6}, {R1,R2,R3,R4,R5,R6}
R2 — R1 R3 R5 — R6 R4
0 1 2 3 4 5
enumerations by linearized DP:
{R1,R2}, {R5,R6}, {R1,R2,R3}, {R1,R2,R3,R5}, {R1,R2,R3,R5,R6}, {R1,R2,R3,R4,R5,R6}
R2 R1 R3 R5 R6 R4
Optimal Join Order

Figure 5: Example for Search Space Linearization

Figure 5 shows an example where the query graph on the left is transformed into its linearized representation in the middle. This process assumes that the order of relations in the optimal plan is already known. Beginning with the leftmost relation, joins are sequentially added to construct increasingly larger subqueries, as illustrated in the right portion of the figure. The steps are as follows:

1. Begin by examining combinations of 2 relations. From the sequence: $\{R1, R2\}$ and $\{R5, R6\}$.
2. Next, consider combinations of 3 relations. Relation $R3$ can join with the result of $\{R1, R2\}$, forming $\{R1, R2, R3\}$.
3. Continue expanding to larger subqueries by adding joins from left to right.
4. Finally, the entire query plan is constructed.

This step-by-step approach significantly reduces complexity compared to a full DP algorithm. While a full DP would require a table with 17 entries, linearized DP requires only 6 entries, as shown in the figure. This difference grows exponentially with the size of the query: the complexity of a full DP algorithm is $O(2^n)$, whereas for linearized DP, it is $O(n^2)$.

2.3.2 IKKBZ ALGORITHM

The *IKKBZ* algorithm was developed in 1984/1986. It generates an optimal left-deep plan in $O(n^2)$, efficiently linearizing the query graph. This left-deep plan's join order is used as the initial join order for Search Space Linearization. By fixing the relation order, the optimizer will only need to consider associativity rather than full join reordering.

The algorithm follows these steps:

- Transform the precedence graph into a linear order.
- If the query graph contains cycles, generate a minimum-spanning tree.
- Assign ranks to nodes based on a cost/benefit ratio.
- Merge child chains successively in increasing rank order.
- Resolve contradictory sequences by merging conflicting nodes into a single unit.

As shown in Figure 6, the *IKKBZ* algorithm operates as follows:

- **Build a precedence graph for each individual relation.** Each relation serves as a potential root (e.g., A, B, C), generating different join order sequences. Each node represents a relation and is assigned a cost/benefit ratio, calculated as the selectivity of join predicates divided by the estimated cost of executing the join (e.g., number of input tuples).
- **Resolve contradictory sequences in child chains by merging into a single node.** Moving from top to bottom, E and F are not ranked in ascending order of their cost/benefit ratio, as rank$(E) >$ rank(F). According to the precedence graph, F should be joined before E, but logically, this is impossible since E precedes F. To resolve this, E and F are merged into a single node, summing their ranks to form $7/10$.
- **Merge child chains based on increasing rank until a linear form is obtained.** All nodes are sequentially merged, producing the initial linearized join order.

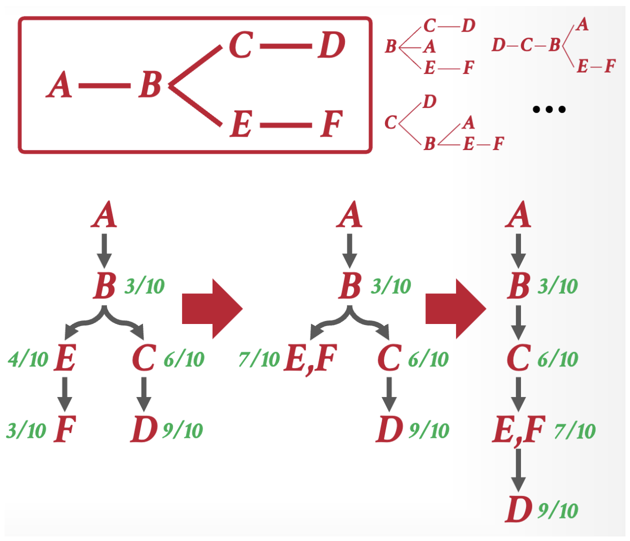


Figure 6: IKKBZ Algorithm Example

Thus, we can use *IKKBZ* to generate an optimal left-deep plan in $O(n^2)$ and linearize the query graph.

2.3.3 Processing Medium Queries

To process medium queries, the query graph is first linearized using *IKKBZ*. Then, the *best bushy plan* is built based on this linearized structure. This algorithm runs in $O(n^3)$ time and produces results that are at least as good as the optimal left-deep plan. With proper linearization, it can discover the globally optimal bushy plan.

2.4 Large Queries

The optimizer handles the most complex queries with an iterative dynamic programming approach. First, it greedily builds an initial query plan. Then, it iteratively refines the plan by optimizing the most expensive subtrees up to size k using DP. By applying the linearization trick, the optimizer can go from $k = 7$ to $k = 100$.

The greedy algorithms used include *Min-sel* (Minimum Selectivity) and *GOO* (Greedy Operator Ordering). *Min-sel* chooses the next join based on the lowest selectivity value. We now discuss *GOO* in detail.

2.4.1 Greedy Operator Ordering

Greedy Operator Ordering (GOO) is a heuristic-based optimization algorithm designed to efficiently determine the join order in large query plans. Instead of exhaustively exploring all possible join orders, GOO iteratively selects the best join candidates based on a cost function. The algorithm consists of below steps:

1. **Identify the Best Pair to Join**: it finds the pair of connected relations (i, j) that minimizes:

   $$\text{cost} = \text{size}(i) \times \text{size}(j) \times \text{selectivity}(i, j)$$

   Intuitively, this cost function prioritizes lower selectivity values and smaller relation sizes to minimize intermediate result sizes. In Figure 7, we choose the pair (B, C).
2. **Merge the Selected Pair into a New Node**: it combines the selected pair into a single node and recomputes the selectivities of edges to other nodes. In Figure 7, (B, C) merges into one node with size 2000, with recomputed costs for pairs (A, BC) and (BC, D).
3. **Repeat Until Only One Node Remains**: the algorithm continues merging nodes until only a single node remains, producing either bushy trees or left-deep trees depending on the join order. Figure 7 shows (A, BC) merging first, followed by (ABC, D), resulting in the final left-deep tree.

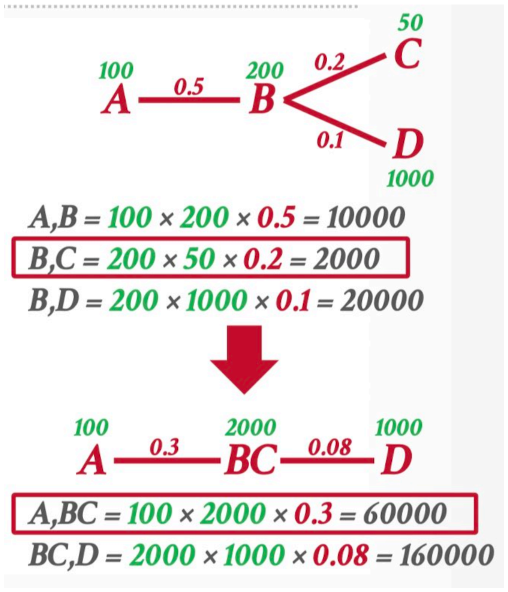

50
C
100
A
0.5
200
B
0.2
0.1
D
1000
A,B = 100 × 200 × 0.5 = 10000
B,C = 200 × 50 × 0.2 = 2000
B,D = 200 × 1000 × 0.1 = 20000
100
A
0.3
2000
BC
0.08
1000
D
A,BC = 100 × 2000 × 0.3 = 60000
BC,D = 2000 × 1000 × 0.08 = 160000

Figure 7: Example of Greedy Operator Ordering

2.5 Experimental Results

The experimental results in Figure 8 show the performance of different database management systems (DBMSs) and query optimization techniques on randomly generated queries with an increasing number of relations. Traditional DBMSs and optimization algorithms struggle significantly as the size and complexity of queries increase. Adaptive approaches, which combine heuristics and targeted dynamic programming, show promising results, significantly outperforming traditional methods in both compilation time and execution efficiency.

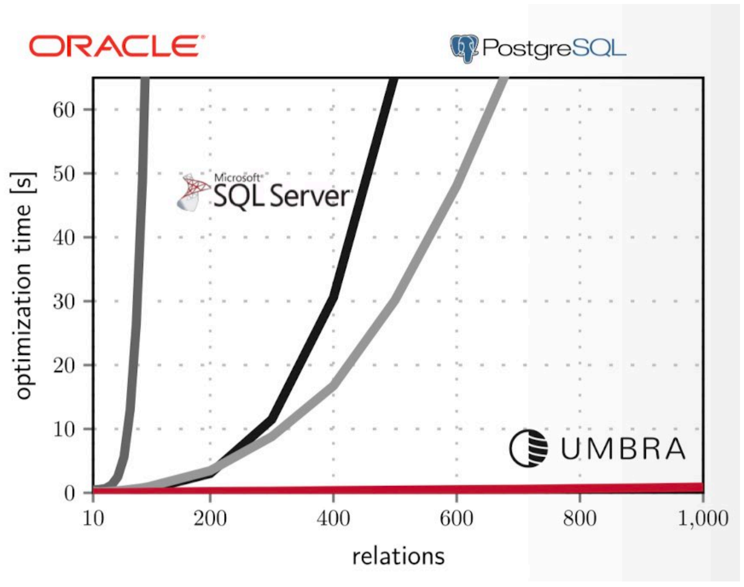


Figure 8: Comparison with Existing Systems.

3 Randomized Algorithms

For large queries or queries that have complex structures, it is computationally impractical to find the globally optimal plan with exhaustive search. *Randomized algorithms* provide an alternative by exploring the solution space randomly, allowing the optimizer to find a good enough plan within a reasonable time. These methods do not guarantee optimality but can efficiently handle cases where traditional dynamic programming or heuristic-based approaches fail due to excessive computation time.

We discuss three randomized algorithms as examples:

3.1 QuickPick

QuickPick[3] is a randomized algorithm that incrementally builds random join trees and selects the one with the lowest cost. It works by randomly selecting and removing edges from the query graph, iteratively adding joins or predicates to the plan. If a newly generated plan has a lower cost than the best one seen so far, it is kept; otherwise, it is discarded, and the process restarts. To improve efficiency, the sampling function is biased toward edges with lower selectivities, leading to better join orderings.

3.2 Simulated Annealing

Simulated Annealing [1] is a randomized optimization technique inspired by the annealing process in metallurgy. It starts with an initial query plan generated using heuristics and iteratively modifies the plan by randomly swapping operators (e.g., join order of tables).

- If the change reduces cost, it is always accepted.
- If the change increases cost, it may still be accepted with some probability, which decreases over time.
- Invalid changes (e.g., breaking sort order) are always rejected.

This method helps to escape local minima and allows the optimizer to explore a larger solution space, improving the chance of finding a better query plan compared to purely greedy approaches.

3.3 Postgres Genetic Optimizer

PostgreSQL's Genetic Query Optimizer (GEQO) is an alternative query planner designed to handle very complex queries with many joins. Based on ideas from genetic algorithms, it uses evolutionary techniques to find cheaper plans when the normal System R style optimizer is too computationally expensive. In practice, PostgreSQL switches to GEQO when the number of relations exceeds the `geqo_threshold` parameter (default value is 12). It works as follows:

1. **Initial Population**: A set of random join orders is generated.
2. **Fitness Evaluation**: Each join order is evaluated based on the estimated query cost.
3. **Selection**: Higher-quality (lower-cost) join orders are more likely to be selected for reproduction.
4. **Crossover**: Selected join orders are combined to create new join orders.
5. **Mutation**: Random changes are applied to introduce diversity and prevent local optima.
6. **Iteration**: The algorithm repeats until a stopping condition is met (e.g., a maximum number of generations or convergence).

In Postgres, there are several parameters to control GEQO's behavior:

- **geqo_threshold:** Determines when GEQO is used, based on the number of tables in a query. Lower values make GEQO handle smaller queries, while higher values restrict its usage to larger queries.
- **geqo_effort:** Controls the effort spent on optimizing query plans. Does not do anything directly, used to set default values for other GEQO parameters. Increasing this value leads to better plans but takes more time.
- **geqo_pool_size:** Specifies the number of join orders considered in each generation. A larger pool size improves exploration but increases memory and computational cost.
- **geqo_generations:** Determines how many generations GEQO will iterate through. More generations allow better refinement but take longer to converge.
- **geqo_selection_bias:** Controls the selection bias (i.e., the selective pressure within the population).

Figure 9 shows an example of GEQO in action.

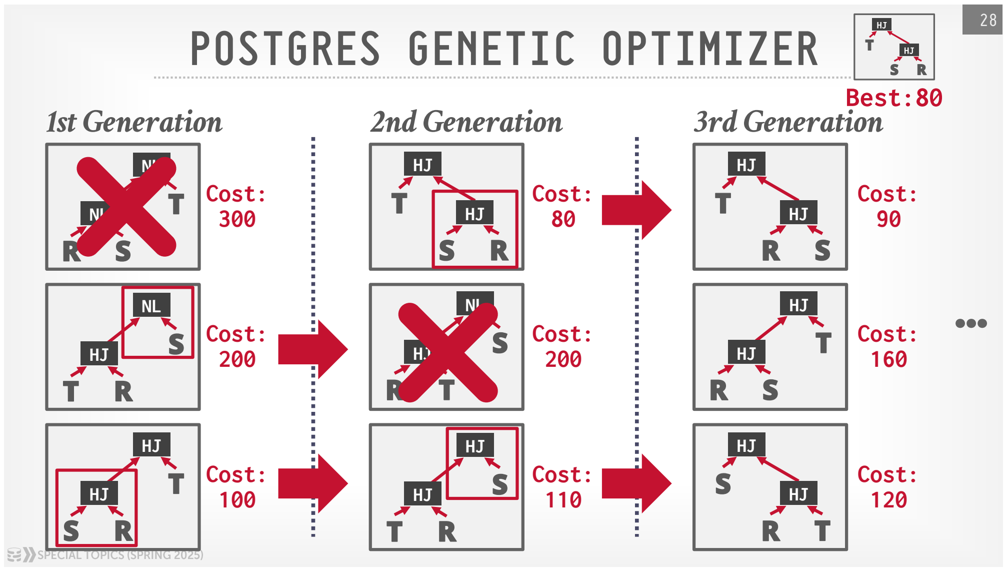


Figure 9: Example of PostgreSQL's Genetic Query Optimizer

First Generation (Initial Plans)

- Three different query plans are generated:

  - **Plan 1:** Uses nested loops (NL) with a cost of 300.
  - **Plan 2:** Uses hash join (HJ) + NL, improving cost to 200.
  - **Plan 3:** Uses only HJ, further reducing the cost to 100.

Second Generation (Mutation and Crossover)

- The optimizer selects the best plans (lower cost) and mutates them:
  - New plans are generated by rearranging joins and combining parts of different plans.
  - The best plan now has a cost of 80.

Third Generation and Beyond

- The process repeats, refining join orders through mutation and crossover.
- The optimizer gradually improves the join plan until no significant cost reductions occur.
- The final best plan achieves the lowest execution cost among all iterations.

By using heuristics and randomness, GEQO finds a reasonably good join order much faster than exhaustive search.

Randomized algorithms offer a practical approach to query optimization when exhaustive or heuristic-based methods become computationally infeasible. They excel in escaping local minima by exploring the search space non-linearly and require low memory overhead, making them efficient for large or complex queries. However, their inherent randomness introduces challenges, such as a lack of explainability in plan selection and the need for additional effort to ensure deterministic query plans.

References

[1] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 9–22, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912365. doi: 10.1145/38713.38722. URL `https://doi.org/10.1145/38713.38722`.

[2] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 677–692, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3183733. URL `https://doi.org/10.1145/3183713.3183733`.

[3] F. Waas and A. Pellenkoft. Join order selection (good enough is easy). In B. Lings and K. Jeffery, editors, *Advances in Databases*, pages 51–67, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45033-7. doi: 10.1007/3-540-45033-5_5.