Understanding PostgreSQL's Genetic Query Optimizer (GEQO)

From Traveling Salesman Problem to Join Order Selection
For Students

Qihan Zhang qihanzha@usc.edu

CSCI 698 Fall 2025

Query optimizers are responsible for choosing efficient execution plans for SQL queries. One of the hardest tasks they face is deciding the *join order* when a query contains many tables. PostgreSQL's *Genetic Query Optimizer* (GEQO) tackles large join queries using a genetic algorithm inspired by the Traveling Salesman Problem (TSP). These notes aim to explain GEQO in a detailed, but still more accessible and informal way. We will review the join-order problem, recall the basics of TSP, and then show how PostgreSQL uses a TSP-style genetic algorithm to search for good join orders.

Contents

1 Motivation: Why Join Order Matters

Consider a complex SQL query that joins many tables. Even if we use the same physical operators (e.g., hash joins, index scans) on the same tables, simply changing the *order* of joins can drastically change the running time.

Intuitively, joining highly selective tables earlier can make later joins cheaper (because intermediate results are smaller). On the other hand, a bad join order may first produce huge intermediate results and thus be extremely slow.

Empirically, on benchmarks such as the Join-Order-Benchmark (JOB), we can see that two different join orders for the same query can lead to execution times differing by factors of 4×, 10×, or even more. This motivates a careful search for a good join order rather than a random or fixed one.

1.1 The Combinatorial Explosion

Suppose a query needs to join n tables. Even if we only consider *left-deep* join trees (which we will define later), the number of possible join orders grows quickly with n. For example:

- For $n = 4$ tables, there are already dozens of possible join trees.
- For $n = 10$ or 12, the number of possibilities becomes huge.
- For full *bushy* trees, the count grows even faster.

Traditional optimizers use dynamic programming to search this space, but the complexity is roughly $O(n^2 2^n)$ for bushy trees. This is fine for small n (say, under 10–12), but becomes impractical when n is large.

1.2 Trade-off: Optimality vs. Planning Time

There is a fundamental trade-off:

- Exhaustive dynamic programming can find the *optimal* plan in the search space, but may take too long for many joins.
- Heuristic or randomized search can find a *good enough* plan faster, but gives up worst-case optimality guarantees.

PostgreSQL's GEQO chooses the second approach when the number of join relations is large. It uses a genetic algorithm to search over possible join orders, trying to balance:

1. **Planning overhead** (time spent in the optimizer).
2. **Execution time** of the final plan.

2 Background: Join Trees and Plan Spaces

Before talking about genetic algorithms, we need a clear picture of what the optimizer is searching over.

2.1 Logical vs. Physical Plans

Given a set of base relations

$$R = \{R_1, R_2, \ldots, R_n\}$$

and a set of join predicates Θ (equality joins, for simplicity), a typical SQL query can be written as:

$$Q = \pi_{\text{proj}}\big(\sigma\big(R_1 \bowtie_\Theta R_2 \bowtie_\Theta \cdots \bowtie_\Theta R_n\big)\big),$$

where σ stands for selection and π for projection.

- A *logical* plan describes the algebraic structure of the joins (which relations are joined, and in what tree shape).
- A *physical* plan additionally specifies physical operators (e.g., hash join vs. sort merge join) and scan methods (e.g., index scan vs. sequential scan).

GEQO focuses primarily on the join order at the logical level, but when evaluating a candidate, PostgreSQL still considers the physical operators to compute a cost.

2.2 Join Tree Shapes

A join tree over R is a binary tree whose leaves are the relations R_i and whose internal nodes are join operators. There are several typical shapes:

- **Bushy trees:** internal nodes can have subtrees containing arbitrary subsets of relations on both sides.
- **Left-deep trees:** each right child is always a leaf.
- **Right-deep trees:** each left child is always a leaf.

Visually, for four relations R_1, R_2, R_3, R_4:

- A *left-deep* tree repeatedly joins one new base table to the growing intermediate result on the left.
- A *bushy* tree can join two intermediate results, each of which might already combine several base tables.

PostgreSQL's GEQO, as implemented today, essentially works with (left-)deep or near-left-deep trees. This is important because it limits the diversity of join shapes that GEQO explores.

2.3 Cost Model (High-Level View)

PostgreSQL assigns to each physical plan P a non-negative *cost* $C(P) \geq 0$. The cost approximates the expected execution time, factoring in:

- Estimated input and output cardinalities for each operator.
- The type of join operator (hash, merge, nested-loop, etc.).
- I/O cost, CPU cost, and sometimes other physical properties.

Formally, you can think of

$$C(P) = \sum_{u \in \text{nodes}(P)} c(u; \widehat{\text{card}}, \widehat{\text{width}}, \text{props}),$$

where $c(\cdot)$ is a local cost function for each node. Importantly, the cost of a node depends on the estimated cardinalities of its inputs, which in turn depend on the earlier joins in the tree.

The optimizer's goal is:

$$P^\star \in \arg\min_{P \in \mathcal{P}} C(P),$$

where $\mathcal{P}$ is the space of admissible plans (e.g., all left-deep trees with certain physical operators).

3 Traveling Salesman Problem (TSP): A Quick Review

PostgreSQL's GEQO treats the join-order problem using ideas from the Traveling Salesman Problem (TSP). Let's briefly recall what TSP is and how we formally define it.

3.1 Optimization Formulation

Let $G = (V, E, w)$ be a complete undirected graph:

- V is the set of *cities*, with $|V| = n$.
- E contains an edge for every unordered pair $\{u, v\}$.
- $w : E \to \mathbb{R}_{\geq 0}$ assigns a non-negative distance or cost to each edge.

A *tour* is a cyclic permutation of the vertices:

$$\pi = (v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}).$$

Its cost is

$$\text{cost}_{\text{TSP}}(\pi) = \sum_{i=1}^{n-1} w(\{v_{\pi_i}, v_{\pi_{i+1}}\}) + w(\{v_{\pi_n}, v_{\pi_1}\}).$$

The optimization problem is: find a tour $\pi^\star$ of minimum cost,

$$\pi^\star \in \arg\min_{\pi} \text{cost}_{\text{TSP}}(\pi).$$

3.2 Decision Formulation

In the decision version, we are given a number $B \geq 0$ and we ask: *Does there exist a tour π whose cost is at most B?*

Formally, the decision problem is:

Given (G, B), decide whether there exists a tour π such that $\text{cost}_{\text{TSP}}(\pi) \leq B$.

Both the optimization and decision versions are NP-hard (and the decision version is NP-complete), which means we do not expect a polynomial-time algorithm that always finds the exact optimal tour for arbitrary instances.

3.3 Permutation Encoding and Adjacencies

The key representation idea is:

- A tour is represented as a permutation of the n cities.
- The edges used in the tour are the adjacencies in that permutation.

Many genetic algorithms for TSP work directly on permutations and try to preserve useful adjacencies from parents to offspring. This will be crucial when we move to join-order optimization.

4 From TSP to Join Ordering

Now we connect TSP to our join-order problem.

4.1 Encoding Join Orders as Permutations

Assume we have n relations $R_1, \ldots, R_n$. A simple encoding of a join order is:

$$\pi = (\pi_1, \pi_2, \ldots, \pi_n),$$

where π is a permutation of $\{1, \ldots, n\}$. This means we consider the relations in the order

$$R_{\pi_1}, R_{\pi_2}, \ldots, R_{\pi_n}.$$

A common way to map this permutation to a left-deep join tree is:

1. Start with R_{π_1}.
2. At step k, join the current result with R_{π_k}.

Of course, we must ensure that join predicates are respected (we should not create early Cartesian products if possible). PostgreSQL's decoder logic takes care of this feasibility issue.

4.2 Adjacency Intuition

The analogy with TSP is:

- In TSP, edges between consecutive cities in the permutation are important: a good tour tends to reuse certain edges.

- In join ordering, we care about *which relations are adjacent* in the permutation, because it affects which joins happen early.

For example, if a fact table F and a highly selective dimension D should be joined early, then having F and D adjacent in the permutation is usually a good thing. A TSP-style genetic operator that preserves adjacencies between permutations can help retain such useful structures during the search.

5 GEQO in PostgreSQL

We are now ready to describe PostgreSQL's GEQO more concretely.

5.1 When Does PostgreSQL Use GEQO?

Let n be the number of relations in the largest join-connected component of the query (roughly, the size of the largest cluster of tables that are all linked via join predicates). PostgreSQL has a configuration parameter `geqo_threshold` (default around 12).

- If $n <$ `geqo_threshold`, PostgreSQL uses its traditional dynamic programming-based optimizer.
- If $n \geq$ `geqo_threshold` and `geqo` is turned on, PostgreSQL uses GEQO for join-order search.

In other words, GEQO is a fallback method for *large* join queries where exhaustive search would be too expensive.

5.2 Search Space and Representation

GEQO searches over a subset X of the permutation space S_n:

$$X \subseteq S_n, \quad x = \pi = (\pi_1, \ldots, \pi_n).$$

There is a decoding function

$$\text{decode} : X \to \mathcal{P}$$

that maps each permutation π to a valid physical plan $P = \text{decode}(\pi)$, usually a left-deep join tree. During this decoding, PostgreSQL:

- Ensures that join predicates are respected (or fixes the order to avoid Cartesian products where possible).
- Lets the standard planner choose physical operators for each step (*hash join* vs. *merge join*, etc.).

5.3 Fitness Function

Once we have a plan $P = \text{decode}(\pi)$, we can compute its cost $C(P)$ using PostgreSQL's usual cost model.

GEQO defines a *fitness* function $F(\pi)$ that is monotonically related to $C(P)$, for example:

$$F(\pi) = \frac{1}{1 + C(\text{decode}(\pi))} \quad \text{or} \quad F(\pi) = -C(\text{decode}(\pi)).$$

Only the ranking of candidates by fitness matters:

- Lower cost ⇒ better fitness.
- Higher fitness ⇒ more likely to be selected as a parent.

5.4 Population and Evolution Parameters

GEQO maintains a population of candidate permutations. Let:

- $N =$ `geqo_pool_size` be the population size.
- $G =$ `geqo_generations` be the number of generations (or derived from `geqo_effort` if not explicitly set).

The overall number of cost evaluations is roughly:

$$\#\text{cost calls} \approx N + G,$$

ignoring constant factors, because:

- We evaluate the initial population of size N.
- Each generation produces one new offspring, whose cost is computed once.

Each cost evaluation can be relatively expensive, since it runs the cost estimator on a full join plan.

5.5 Evolutionary Loop (Genitor-Style)

GEQO uses a *steady-state* evolutionary algorithm inspired by Genitor. At a high level, each generation does:

1. **Selection:** pick two parent permutations from the current population, with a bias toward higher fitness (lower cost).
2. **Crossover:** apply a TSP-style recombination operator, *Edge Recombination Crossover* (ERX), to create one offspring.
3. **Mutation (optional):** occasionally apply a small random change to the offspring (PostgreSQL historically uses mutation lightly).
4. **Replacement:** insert the new offspring into the population and remove the worst individual.

This process repeats for G generations. At the end, GEQO returns the best permutation (and its decoded plan) found so far.

5.6 Edge Recombination Crossover (ERX)

ERX is specifically designed for permutation problems like TSP and join ordering.

Assume we have two parent permutations p and q over the set $\{1, \ldots, n\}$:

$$p = (p_1, \ldots, p_n), \quad q = (q_1, \ldots, q_n).$$

For each gene (relation index) i, we define its adjacency set:

$$A_i = \text{Nbrs}_p(i) \cup \text{Nbrs}_q(i),$$

where $\text{Nbrs}_p(i)$ is the set of predecessors and successors of i in p (treated cyclically), and similarly for q.

ERX constructs an offspring $o = (o_1, \ldots, o_n)$ as follows:

1. Choose o_1 as a random starting gene from $\{1, \ldots, n\}$.
2. For each next position $k + 1$:
   - Look at the adjacency set A_{o_k} and remove any genes that have already been used in $o_1, \ldots, o_k$.
   - If the remaining set is non-empty, pick the gene j in this set that has the *smallest adjacency list* (fewest neighbors). If there are ties, break them randomly.
   - If the set is empty, pick a random unused gene.
   - Set $o_{k+1} = j$ and continue.

The intuition:

- ERX tries to preserve edges (adjacencies) that appear in the parents.
- Relations that are often neighbors in good parents are likely to stay neighbors in the offspring.

In the join-order context, this means that if certain tables tend to appear next to each other in good join orders, ERX tends to keep them adjacent.

5.7 Decoding: From Permutation to Join Plan

After we obtain an offspring permutation o, GEQO must turn it into a legal join tree:

1. Scan o from left to right.
2. At each step, attach the next relation in a way that respects join predicates (using the join graph derived from Θ).
3. Avoid pure Cartesian products if possible, by reordering or deferring attachments.
4. For each join step, call the standard planner to choose the physical join operator with minimum local cost (under the global cost model).

The result is a physical plan $P = \text{decode}(o)$, whose cost can be computed and used as the fitness for o.

6 Limitations and Practical Issues

6.1 Instability

Because GEQO is randomized (its behavior depends on `geqo_seed`), the chosen plan can change if:

- The seed changes.
- Statistics change.
- The search behaves slightly differently (e.g., due to small configuration changes).

This can be problematic in production systems, where DBAs often want stable plans for repeatable performance and easier debugging.

6.2 Limited Plan Shapes

GEQO's current implementation in PostgreSQL mainly explores left-deep or near-left-deep trees. It does not systematically search over full bushy trees. This means:

- Some potentially excellent bushy join orders are never considered.
- The benefit of GEQO is limited to exploring different left-deep orders rather than exploring fundamentally different tree shapes.

6.3 Cost Model Mismatch

GEQO relies on PostgreSQL's cost model to evaluate candidates. However, the cost model is imperfect (due to cardinality estimation errors, simplifying assumptions, etc.). Thus:

- A plan that looks cheap under the cost model might not be fast at runtime.
- Conversely, some genuinely good plans might be assigned higher estimated cost and thus be ignored by GEQO.

7 How to Think About GEQO in Practice

7.1 When Might GEQO Help?

GEQO is most promising when:

- The query involves many joined tables (above the threshold).
- The join graph is complex and the dynamic programming search would be too expensive.
- You are willing to accept a heuristic plan for the sake of faster planning.

For certain large, messy queries (e.g., ad-hoc analytical queries), GEQO can sometimes find significantly better join orders than a limited DP search.

7.2 When Might GEQO Hurt?

GEQO might be problematic when:

- You care a lot about predictable and stable performance, especially in the tail (e.g., 95th/99th percentile latency).
- Your workload mostly contains queries with moderate join size, where the DP approach is still affordable.
- You have observed that GEQO often picks clearly bad join orders for your workload.

In such cases, many practitioners choose to disable GEQO and rely on the traditional optimizer.

7.3 Relevant PostgreSQL Parameters

To control GEQO in PostgreSQL, the main configuration parameters are:

- `geqo` (on/off): master switch for GEQO.
- `geqo_threshold`: minimum number of tables in a join for GEQO to be considered.
- `geqo_pool_size`: population size N.
- `geqo_generations`: number of generations G (or derived from `geqo_effort`).
- `geqo_seed`: random seed for reproducibility.

Tuning these parameters changes the trade-off between planning time and the thoroughness of the search.

8 Conclusion

1. Choosing a good join order is crucial for performance on complex queries.
2. Exhaustive search over all join orders is combinatorially expensive.
3. PostgreSQL uses GEQO (a genetic algorithm) to search for good join orders when there are many tables:
    - Each candidate join order is a permutation.
    - Plans are evaluated with the usual cost model.
    - Edge Recombination Crossover preserves useful table adjacencies.
4. GEQO is inspired by the TSP, but the cost structure in query optimization is more complicated than simply adding edge weights.
5. In practice, GEQO can both help and hurt: it may find excellent plans for some queries but can also increase tail latency and reduce stability.
6. Understanding GEQO helps you reason about optimizer behavior in PostgreSQL and motivates further research on hybrid and learned optimizers.

Acknowledgments

These notes are based on the author's CSCI 670 project on PostgreSQL's Genetic Query Optimizer (GEQO) and accompanying presentation. The author makes it more concise and easier to understand, although they ignore many technical details and experimental results.