Technical Perspective: Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems

Andreas Kipf
Amazon Web Services
kipf@amazon.com

ABSTRACT

Query optimization is the process of finding an efficient query execution plan for a given SQL query. The runtime difference between a good and a bad plan can be tremendous. For example, in the case of TPC-H query 5, a query with 5 joins, the difference between the best and the worst plan is more than 10,000×. Therefore, it is vital to avoid bad plans. The dominating factor which differentiates a good from a bad plan is their join order and whether this join order avoids large intermediate results.

There are two fundamental approaches to query optimization that were both introduced back in the 70s. The first approach, cost-based query optimization, also referred to as the Selinger optimizer, performs an exhaustive search using dynamic programming (DP), and uses cost functions to compare two plans enumerated by the DP algorithm. While this approach can theoretically find the optimal plan, it depends on good cost functions and on well-calibrated cardinality estimates. However, estimating the size of intermediate results is a notoriously hard problem, especially in the presence of correlations found in real-world data. For example, a French actor is more likely to participate in a French movie than a German actor. In contrast, the INGRES optimizer uses runtime optimization to re-optimize the remaining query at each intermediate step. While the Selinger approach allows for pipelined query execution, where intermediate results are streamed to the next operator, the INGRES approach requires materialization at each intermediate step.

The paper "Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems" revisits the runtime re-optimization approach in the context of big data systems. Specifically, the authors follow the INGRES approach and materialize intermediate results as new tables, which are considered by the optimizer to optimize the remaining query in a greedy fashion by adding one table at a time. In contrast to INGRES, which bases its decisions solely on dataset cardinalities, the authors additionally build two types of sketches: HyperLogLog sketches to estimate unique values in join columns and Greenwald-Khanna to estimate selectivities. They integrate their approach into AsterixDB, an open-source big data management system that comes with a rule-based optimizer and a distributed query execution engine. The authors interleave planning and execution: First, predicates are pushed down to base tables. Then the corresponding scans are executed and their results and statistics are materialized, before the planner is invoked again to find the next best join to be scheduled for execution. They repeat this process until only two joins remain. In addition to the join order, the authors use the runtime statistics to choose between different physical operators, such as a shuffle-based and a broadcast-based join.

The authors evaluate their approach using TPC-H and TPC-DS with scale factors of up to 1,000. They show that statistics collection adds less than 20% of overhead. The paper also shows that the materialization overhead increases with dataset size, since intermediate results tend to be larger. They compare against the cost-based approach (Selinger optimizer) and show that the materialization overhead of their approach is amortized in almost all cases. While this is an impressive result on its own, it would be illuminating to see how the system compares with state-of-the-art systems such as Umbra.

Overall, this paper tackles an important problem faced by real-world optimizers to date: avoiding bad query plans. Runtime re-optimization can avoid bad plans in most cases. On the flip side, materializing intermediate results can be very expensive on its own, especially when they spill to disk. Also, re-optimization is not compatible with execution engines that operate in a pipelined fashion. In such cases, one can only re-optimize at full pipeline breakers. Or throw away work and start over when decisions turn out to be unfortunate at runtime, such as which input to use as the build input of a join. This requires careful engineering and is therefore only implemented by a few systems.

The authors' idea of collecting additional statistics for intermediate results to be used to determine the next best join is very interesting and may have a significant impact on the quality of the resulting query plan. A promising avenue of future work could be to extend these statistics even further or use join sampling techniques to better estimate the size of the next best join.

While their approach can avoid bad plans, it is still sensitive to the first plan or rather to the greedy decisions taken along the way. The pathway forward seems to be to combine the best of both worlds: First, finding a good initial plan based on cost-based techniques and then re-optimize at runtime whenever possible and worthwhile. That is, when reality differs from expectation (cost-based plan). Recent work on ML-based cardinality and cost estimation can help to find a good starting plan that can be re-optimized later.