Surprisingly Effective: an Evaluation of Static Decision Rules for Query Re-Optimization in DuckDB

Laurens Kuiper
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
laurens.kuiper@cwi.nl

Arjen de Vries
Radboud University
Nijmegen, Netherlands
a.devries@cs.ru.nl

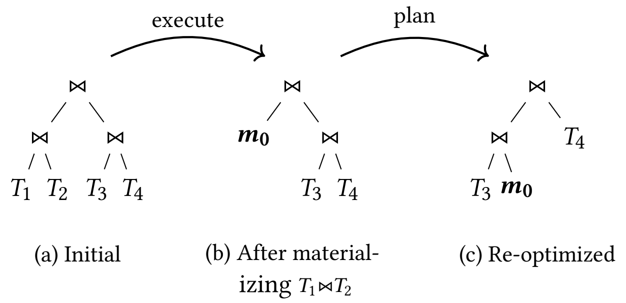

Figure 1: Query plan re-optimization. $T_1 \bowtie T_2$ is materialized into a temporary table m_0. The optimizer can generate a more efficient query plan using exact information about the join.

Traditional query optimizers rely on cost models to assess plan cost and pick an 'optimal' plan for execution. This architecture is known to suffer from serious problems. These problems are not identified by widely used benchmarks like TPC-H, due to its simple data generation method. To address this, the Join Order Benchmark (JOB) was introduced, consisting of analytical multi-join queries over IMDB. JOB demonstrated evidence for the conventional wisdom that cost models perform poorly as the number of joined tables increases, due to poor cardinality estimates.

Estimates are poor, especially for joined relations, because optimizers use simplifying assumptions to estimate cardinality, which rarely hold true in practice. Estimation errors propagate through the query plan, increasing exponentially. As a result, poor join orders are generated. Furthermore, optimizers that rely too heavily on these estimates when deciding which join algorithm to use suffer an even bigger performance loss.

Since its inception, JOB has re-incited the interest in query optimization. Novel sampling techniques to improve cardinality estimation have been proposed. Deep learning has been employed for the same, and also for a full end-to-end cost model. These approaches are effective at improving query plan quality, but have limitations. Sampling causes overhead, and sampling many different joined relations is not feasible, therefore exponential error propagation is still a problem. Deep learning does not suffer as much from this problem, because it can memorize many different joined relations. However, deep learning does not generalize well to unseen data, and can takes several hours to train, making it unsuitable in many use cases.

The approaches discussed have in common that they stick to the traditional plan-then-execute paradigm of query processing. Another approach is to delay optimization decisions until runtime. Re-optimization is such an approach, which interleaves the planning and execution phases, possibly multiple times. Re-optimization is illustrated in Figure 1. Statistics are collected during execution in order to detect whether the current plan is sub-optimal. Based on these statistics, a re-optimization scheme can decide to halt execution, and generate a new plan for the remainder of the query. This ameliorates the problem of exponential error propagation, and does not suffer from being unable to generalize to unseen data.

In this talk, we present experiments with our implementation of re-optimization [1] in DuckDB. DuckDB is an embedded analytical database system, newly developed at CWI. DuckDB was chosen for our experiments because it is an open-source analytical DBMS with full join order optimization. Our results show that a *simple* static re-optimization scheme can reduce the runtime of the 20 longest queries by up to 34%. For these queries, the benefit of an improved query plan is worth the costs of intermediate result materialization and additional planning time. By comparing plan cost, it is clear that the reduction in runtime can be attributed to improved plan quality.

For comparison, we re-implement a *dynamic* re-optimization scheme in DuckDB, that was recently simulated in PostgreSQL and evaluated on JOB. The dynamic scheme has a clear advantage over our static schemes, because it is more effective at deciding when re-optimization is worthwhile. It is especially more effective at deciding not to re-optimize short running queries. However, we argue that the simulation is unrealistic, because their scheme cannot be implemented in a system with a Volcano-style execution model, like those of PostgreSQL and DuckDB. The reason for this is that it the scheme requires full materialization of intermediate results when a cardinality threshold is reached, which is not possible when data is streamed through operators.

We show that the shortcomings of our simple static re-optization schemes regarding short running queries can be easily overcome with a slightly more sophisticated scheme that takes the structure of the query plan into account. With this scheme, runtime of the longest 20 queries is reduced by almost 30%, while only slowing down the runtime of the 20 shortest running queries by 8%. The runtime of average queries in the benchmark remains virtually the same with re-optimization.

REFERENCES

[1] Laurens Kuiper and Arjen de Vries. 2020. Exploring Query Re-Optimization in a Modern Database System. *Master Thesis* (2020).