Analyzing loop hoisting optimizations for recursive SQL queries

Olaf Van Bylen [1]

[1] Data Science Institute, UHasselt, Belgium
olaf.vanbylen@uhasselt.be

Recursion in SQL is famously restricted to linear recursion, with transitive closure over a graph being a prominent example of such a query. These queries are typically evaluated through a semi-naive algorithm. First, the non-recursive term produces an initial set of rows, after which the recursive term repeatedly processes the current working table to produce the next one. For duplicate-eliminating `UNION`, previously discovered rows are discarded, and evaluation terminates when a fixpoint is reached, i.e., when no new rows are derived.

Traditionally, SQL systems evaluate such queries by treating each iteration as a separate, non-recursive query, evaluated from scratch over the input database and current working table. In particular, although the recursive query remains the same, the execution plan can differ between iterations because of the altered working table. Unsurprisingly, parts of these execution plans are repeated across iterations, leading to potentially redundant work. In the case of transitive closure, each iteration might build a hash table over the edge relation as part of a join, even though the edge relation is fixed. Similar to loop hoisting, we can reuse this hash table across iterations. As of version 2.0.0, DuckDB includes this optimization, retaining eligible recursion-independent hash builds across iterations to avoid repeated scans and reconstruction of the same state.

However, reusing execution state across iterations is not limited to subqueries over the static input tables. If the execution plan is fixed, incrementally maintaining modified results can be beneficial as well. In the case of transitive closure, a hash table over this recursive relation can be maintained incrementally to avoid repeated scans over the intermediate results. Contrasting the previous reuse of static data, this optimization is more challenging, as the performance depends on the efficiency of the underlying incremental data structure. In fact, it is not even guaranteed that the overhead of maintaining such a data structure is lower than the cost of recomputing it from scratch.

Additionally, we may fix the physical relation layout across iterations, to avoid e.g. costly shuffling to exploit parallel processing.

Currently, it remains unclear how effective these optimizations are in practice. To this end, we experimentally evaluate different optimization strategies over the transitive closure SQL query on several graph datasets and compare the obtained performance gain against the baseline that recomputes each iteration from scratch. One of our main focuses is on how graph structure influences the effectiveness of these optimizations. Dataset classes can differ in iteration count, volume of newly discovered reachable pairs, input size, ... By relating these characteristics to execution behavior, we study which particular combinations of optimizations work especially well for certain classes and which provide little to no benefit. The central aim of this work is to understand better where current optimizations have an effect and where significant remaining execution time and therefore potential for future optimizations lies.

This is ongoing research, in collaboration with Frank Neven (UHasselt), Brecht Vandevoort (UHasselt) and Stijn Vansummeren (UHasselt).