Efficient mid-query re-optimization of sub-optimal query execution plans

Authors: Navin Kabra & David J. DeWitt
Motivation - what problems are solved?

**Query plan creation**

Query optimizer creates a query plan for the query based on estimates

Cardinalities give estimates of query selectivity and table-sizes

Based on statistics/histograms

Must be kept up-to-date

---

**Problem**

Inaccurate cardinality estimates

+ User-defined data-types

+ User-defined functions

= **Inaccurate cost-estimates**

➡ Sub-optimal query plans

➡ **Performance degradation**
Method

- Query plan is annotated with statistics and estimates from the optimizer
- During execution, actual runtime statistics are collected
- Estimates and actual cardinalities are compared

- If significant mismatch (estimates < actual cardinalities):
  - Re-optimize ➡ New plan created
  - Actual found cardinalities are used
  - Execute new plan if lower cost than current plan, re-optimization overhead-cost is also taken into account
Architecture

- Statistics Collector operators are inserted right after query optimization (SCIA)
- At a collector, statistics such as cardinalities, sizes, and histograms are recorded
  - Available for usage in case of re-optimization
- The collected statistics are also used to perform dynamic memory-allocation
  - Effective during execution without re-optimizing
Keeping overheads low

- **Statistics-collectors insertion algorithm**
  - Determines “most effective” statistics to collect
  - Maximum acceptable overhead ➡ limit amount of statistics-collectors

**Inaccuracy potential**

- **Low**
  - Serial histogram
  - Index
  - Unique values estimate in base table

- **Medium**
  - Equi-width and equi-depth histograms

- **High**
  - No histogram
  - Estimates not in base table
  - User-defined methods

- Rules determine how inaccuracy potential levels are propagated upwards in the query plan
- Order collectors by:
  - Higher potential
  - Largest portion of query plan affected
- Remove least effective collectors until below maximum acceptable overhead

- **Memory allocation**
  - Only improved for future operator
  - Only re-invoked if improved estimates are available
Test results

- Implemented in Paradise Database System
- TPC-D dataset with 3GB database
- Queries: Q1, Q3, Q5, Q6, Q7, Q8, Q10 from TPC-D specification
- Ran each query with and without the use of Dynamic Re-Optimization
- Each query was executed 5 times and the average execution time was reported
Test results

- The queries could be classified into 3 categories based on expected effects
  - Simple (0-1 joins - Never re-optimized)
  - Medium (2-3 joins)
  - Complex (4+ joins - Primary target of Dynamic Re-Optimization)
Test results

- The effect of skew on the performance of Dynamic Re-Optimization was also tested by generating skewed data.
- The relative performance of Dynamic Re-Optimization improves slightly as more skew is introduced in the system.

Effect of skew, higher $z = \text{more skew}$
Conclusions

Dynamic Re-Optimization can significantly improve performance of complex queries if their execution plans are sub-optimal without slowing down queries whose plan don’t benefit from re-optimization.

As applications force databases to support complex decision support queries, complex data-types and user-defined methods, it will become more difficult for query optimizers to statically produce good query execution plans. Some form of re-optimization of query execution plans at run-time will become necessary in such cases.