Mid-Query Re-Optimization

Navin Kabra
David J. DeWitt

Ivan Terziev
University of Pennsylvania
Query Optimization

- Ideally an optimal plan should be found. However, this is not the case especially for complex queries.
- Optimizers are unable to accurately estimate the cost of a complex execution plan. Why?
  - Simplified cost model
  - Out-of-date statistics
  - Exponential error
  - Insufficient information about the runtime system
  - OOD or user-defined datatypes do not fit in the cost model.
Query Optimization

- Solutions:
  - Competition model: start with multiple execution plans and leave the best one.
  - Dynamic query plans: statistics during optimization are stored in the plan. Before execution check against the statistics catalog.
  - Query Scrambling: re-optimizes only if data from a source arrives slowly, not relevant
Query Optimization

– More solutions:
  • Parametric query optimization algorithms: one plan that is a combination of a number of subplans each of which is optimal for a given set of values. In runtime decide which precompiled plan to choose. Sounds good but hard to create from a great space of possibilities.
  • Mid-query re-optimization: gather runtime statistics and fix the remainder of the query (our discussion).
Complex Queries

• Mid-Query optimization mainly addresses complex queries

• Why are they evil?
  – They are long and complex
    (Q: Find all climbers below average age that are not authors with at least one publication and have rented at least one boat, red or green in the past 2 weeks.)
  – Nested operators cause cost model errors to grow exponentially
  – It is hard to predict their behavior based on initial estimates (which is what optimizers have)
  – Operators share memory
  – Pipeline stalls due to binary operators
The problem: Evil Queries

<table>
<thead>
<tr>
<th>Value</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..3</td>
<td>50%</td>
</tr>
<tr>
<td>4..9</td>
<td>20%</td>
</tr>
<tr>
<td>10..20</td>
<td>30%</td>
</tr>
</tbody>
</table>

Actual result on a1

<table>
<thead>
<tr>
<th>Range</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1…3</td>
<td>0%</td>
</tr>
<tr>
<td>4…9</td>
<td>20%</td>
</tr>
<tr>
<td>10…20</td>
<td>80%</td>
</tr>
</tbody>
</table>

Select C.a2 = 'red' or 'green'
The Algorithm

• Detects if a query is suboptimal and re-optimizes the remainder of the query.

• Features of the algorithm:
  – Execution plan modification
  – Resource reallocation (memory, scheduling?, others?)
  – Keeping the overhead low
  – Annotated execution plan: maintains statistics at key points in the tree structure
  – Runtime statistics: gather statistics during query execution
Runtime Statistics

• Tools:
  – Statistic collector operators: physical logic operators just like selections, joins etc.

• Decisions:
  – What kind of statistics to collect
  – Where to insert statistics collector operators

• Limitations:
  – Information must be gathered with only one pass over the input
  – Pipelined operators cannot benefit runtime statistics.
  – Execution overhead
Dynamic Resource Reallocation

• Why? Not all operators receive the memory requirements, e.g. operators running in parallel

• How? Reallocate memory based on current statistics. Done by the memory management module
Modifying The Query Plan

• Ok, we want to modify the query plan. We can:
  – discard current execution plan and build a new one. Yeah, sure!
  – stop execution. Re-optimize operators that have not yet started. Sounds easy but hard to implement. If you have a good solution email navin@cs.wisc.edu and dewitt@cs.wisc.edu
  – stop the execution before output reaches next (parent) operator. Save current results (might by I/O costly) and generate a new SQL query. A compromised solution. Easy to implement.
Optimization

• Re-optimization includes:
  – Query plan modification
  – Dynamic resource reallocation

• We optimize when:

\[ \frac{|T_{\text{improved}} - T_{\text{initial}}|}{T_{\text{initial}}} > \theta \]

• Theta is empirically chosen and accounts for overheads
Maintaining Low Overhead

• Getting hyped with gathering statistics and re-optimizing to death might (will) cause a query to run longer.

• Decide on what statistics to use during query execution time

• Process the current plan to insert statistics collector operators with a heuristic algorithm

• Do not insert statistics operators in simple queries or queries expected to run fast.
Algorithm For Inserting Statistics Collectors

• Input: annotated execution plan, maximum accepted overhead fraction
• Output: annotated execution plan extended with statistics operators
• Heuristic approach determines inaccuracy potential (low, medium, high) of the statistics of an annotated plan
  – Determine effectiveness of possible statistic collectors based on inaccuracy potential
  – Sort possible statistics operators on effectiveness and iteratively delete the lowest effective operator until expected computing time drops below maximum accepted overhead
Rules of Thumb

- Statistics collectors are inserted after filtering operators and before the join operators.

- The inaccuracy potential for non-equi-joins is always high.
- Detailed (serial) histograms have low inaccuracy potential.
- The inaccuracy potential of a selection with two inputs is one level higher than its inputs.
- The inaccuracy potential of an equi-join not over a key attribute is one level higher than its inputs.
All Comes Together

• Step by step:
  – Conventional optimizer generates a conventional plan
  – Statistics collector algorithm inserts statistics operators ensuring does not overload the plan by some given fraction
  – The final plan (annotated) is then executed
  – Data from statistics operators is used to generate a better cost estimate $E$.
  – $E$ is compared to the optimizer’s estimate $C$. If $E +$ overhead is much better than $C$ then generate a new plan and repeat from the beginning
Implementation

• The algorithm is implemented in the Paradise database system
• Components: query optimizer, memory manager, scheduler & dispatcher, data server.
• Algorithm uses dynamic programming (all cool algorithms do so!)
• Statistics are based on histograms. One page is reserved for histograms updates on per-tuple basis.
More
Even More

The Query optimizer or the memory manager can be called at any level

Level 0

Level 1

Level 2
Results

- Simple queries do a little worse (5% overhead)
- Medium queries have some to none benefit
- Complex queries benefit a lot
- Results are consider to be excellent and are as expected. The algorithm was never supposed to be a moon walker.
Conclusion

- It works!