Query Optimization

Papers:
Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans (Kabra & DeWitt)
LEO – DB2’s Learning Optimizer (Stillger, et al.)

presented by John Davin
February 18, 2005

Outline
- Introduction + Motivation
- Mid-Query Re-optimization
  - Algorithm
  - Analysis/Results
- LEO
  - Algorithm
  - Analysis/Results
- Comparisons and Conclusions

Query Optimization
- SQL is nice – allows us to write queries that specify what data we want, but not the details of how to retrieve it.
  - So the query optimizer needs to figure that out!
- DBMSs construct a query execution plan (QEP) for executing each query.
- Plan is based on statistics in system catalog (cardinalities, selectivity).
  - Relies on several assumptions...
### Assumptions / Violations

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currency of information</td>
<td>- statistics are not updated after every update/insert/delete.</td>
</tr>
<tr>
<td>Uniformity</td>
<td>- data skew affects joins</td>
</tr>
<tr>
<td>Independence of predicates</td>
<td>- dependence of attributes affects selectivity estimate.</td>
</tr>
<tr>
<td>Principle of inclusion</td>
<td>- selectivity assumes each value in smaller table has match in larger table.</td>
</tr>
</tbody>
</table>

### Related Work

- Query Scrambling (Urhan et al, '98) – wide area networks – reschedule when a data source times out.
- Competition model (Antoshenkov, '93, '96) - start multiple plans until see which is best. Doesn't help with join ordering.
- Derr, et al, '93 – plans are reoptimized right before execution if stats in catalog are different from stats in the plan.
- Aboulnaga + Chaudhuri, '99 – query feedback loop to correct histograms. Doesn't address joins or aggregation.

### Dynamic Re-Optimization: Overview

- Annotated QEP - plans contain optimizer's estimates of all sizes/costs.
- Runtime collection of statistics
- Dynamic resource re-allocation – shared resources like memory.
- Query plan modification
- Low overheads – only collect statistics at certain points.
Collecting statistics

- Interpose a “statistics collector” into QEP after filter operation runs at each relation.

Collecting Statistics

- **Advantages:**
  - Stats can be more specific to the current query, in contrast with standard optimizer stats (e.g., histograms) which are general.
  - Improved estimates are actual observations – possibly better than optimizer’s estimates.

- **Limitations:**
  - only one pass over data
  - when several operators are pipelined concurrently, we won’t have stats until all are done.

Resource Re-allocation

- Joins are memory hogs. Want to maximize efficient use of memory.
- Memory allocation is based on db statistics.

  - When improved estimates are available, memory allocation can be adjusted mid-query.
Query Plan Modification

- QEPs may have sub-optimal choice of join order, or join types
- What should we do?
  - Throw out the current execution and start over?
  - Suspend query in mid-execution and send remainder of plan back to optimizer
  - Allow current operator (eg, hash-join) to complete. Pipe its output to a temp file. Generate a new SQL query for remainder of plan and submit to optimizer.

When should we re-optimize?

- Tcur-plan, optimizer = optimizer’s estimate of plan execution time
- Tcur-plan, improved = improved estimate of execution time
- T(opt, estimated) = time to re-optimize query. (precomputed based on # of joins)
When should we re-optimize?

- $T_{\text{cur-plan, optimizer}}$: optimizer’s estimate of plan execution time
- $T_{\text{cur-plan, improved}}$: improved estimate of execution time
- $T_{\text{opt, estimated}}$: time to re-optimize query.
  (precomputed based on # of joins)

- **DON’T** re-optimize if:
  \[
  \frac{T_{\text{opt, estimated}}}{T_{\text{cur-plan, improved}}} > \theta_1 \ (0.05)
  \]

- **DO** re-optimize if:
  \[
  \frac{T_{\text{cur-plan, improved}} - T_{\text{cur-plan, optimizer}}}{T_{\text{cur-plan, optimizer}}} > \theta_2 \ (0.2)
  \]

**Summary:** re-optimize only if the reoptimization time is less than 5% of the query execution time, and improved execution time estimate is at least 20% different from optimizer’s estimate.

Statistics collectors insertion

- CPU overhead of statistics collection can be significant. Need to make sure we don’t hurt simple queries too much.
- Cardinality, page counts, min/max values are cheap so collect them at every point in QEP. Histograms + estimates of # of unique values are more expensive.

   ➔ Choose a subset of statistics which will fit within the maximum acceptable overhead $\mu$, and are considered “effective.”

Inaccuracy Potential

- Effectiveness based on:
  - Probability that optimizer estimates are inaccurate. (inaccuracy potential of low, medium or high)
  - Fraction of QEP that is affected by this statistic.

- Inaccuracy potential determined by a bunch of rules. Example: for a histogram, “low” if we have a serial histogram, medium for equi-width+equi-depth, high if no histogram.

   ➔ Order statistics by effectiveness (ordering: inaccuracy potential, size of affected portion of QEP). Discard lowest effective operators that won’t fit within acceptable overhead.
Results
TPC-D benchmarks with 3GB database on 4 PCs with dual 133mhz

- 10-30% improvement on complex queries

Is Memory or plan adjustment better?

- Plan modification better when queries are complex.

LEO: DB2's LEarning Optimizer

IBM Research
LEO Overview

- Motivation: Self-tuning DBMS’s reduce cost and improve performance.
- Cardinality estimates in statistics tables can be off by orders of magnitude. LEO seeks to reduce all modeling errors by learning correction factors.
- Takes an idea from feedback control systems: *Make a feedback loop within the DBMS so that statistics are self-correcting.*

LEO Feedback Loop

- LEO’s components:
  - save optimizer’s plan
  - monitoring component
  - analysis component
  - feedback exploitation

Plan Skeleton
Statistics

- Adjustment factors stored separately from original catalog – allows us to easily disable learning, and avoid duplicate re-adjustments.
- Maintain consistency of statistics (indexes, disk page counts)
- When DB2 makes updates to its statistics, LEO adjustments need to be re-adjusted.

Definitions

- **Section** – executable program from code generator.
- **Plan skeleton** – a “road map” of the QEP

Analysis

- Separate process, can be run offline.
- Analyzes feedback data on a *per-query* basis; allows interruptions.
Match counters with operators

- Matching monitor counters with operators in plan skeleton:

Calculating adjustments

- \textit{old}_{\textit{est}} = \text{estimated selectivity from optimizer}
- \textit{old}_{\textit{adj}} = \text{old adjustment factor}
- \textit{act} = \text{actual selectivity based on monitor data}
- \textit{adj} = \text{adjustment factor}

An error is indicated if:
\[
\left| \frac{\textit{old}_{\textit{est}} - \textit{act}}{\textit{act}} \right| > 0.05
\]

- Compute adjust factor adj:
  \[
  \textit{act} = \text{stats} * \text{adj}
  \]
- But need to account for old adjustment (\text{old}_{\textit{adj}}):
  \[
  \textit{adj} = \text{act} * \left( \frac{\text{old}_{\textit{adj}}}{\textit{old}_{\textit{est}}} \right)
  \]
Example

- \( X.\text{price} > 100: \)
  - \( \text{stats} = 7200 \)
  - \( \text{act} = 7623 \)

  \[ \text{act} = \text{stats} \times \text{adj} \]
  \[ \text{adj} = \frac{\text{act}}{\text{stats}} = \frac{7623}{7200} = 1.06 \]

Example

- \( X.\text{price} > 100: \)
  - \( \text{stats} = 7200 \)
  - \( \text{act} = 7623 \)

  \[ \text{act} = \text{stats} \times \text{adj} \]
  \[ \text{adj} = \frac{\text{act}}{\text{stats}} = \frac{7623}{7200} = 1.06 \]

- Selectivity of TBSCAN X:
  - \( \text{stats} = 1149/7200 = 0.1596 \)
  - \( \text{actual} = 2283/7623 = 0.2991 \)
  - \( \text{Price} > 100 = (1 - \text{Price} < 100) \)
    \[ \text{Adj} = \frac{\text{act} \times \text{old_adj}}{\text{old_est}} \]
    \[ \text{Adj} = (1 - 0.2991) \times 1.0 / (1 - 0.1596) = 0.8340 \]

Challenges to statistics collection

- An index scan may not scan the entire base table, for example if we have a range predicate.
- *Implicit early out* – a merge-join can terminate without looking at all of the tuples of one of the tables.

- Therefore: LEO cannot use monitor counts for cardinality when it does not see entire table.
Using Learned Knowledge

- Base table cardinalities: \( \text{card} = \text{stats\_card} \times \text{adj} \)
  - Consistency: NPAGES must be updated
  - Adding rows does not always increase column cardinality. Ex: adding an employee does not increase card of Sex column (still 2, or 1 at some tech co's).
  - Indexes: FIRSTKEYCARD is card of first index col.

Adjustment Example

- Joins: multiply adjustment factor by optimizer's est.
  - Free bonus: eliminates errors from base table card's.
  - Also takes advantage of correlations between join cols.
- GROUP BY, DISTINCT, UNION, EXCEPT – applies to these operators also, but not implemented currently.
Overhead & Results

- monitoring overhead < 5% (on 2 TPC-H queries)
- Results:
  
  QEP with learning was improved by 14x over QEP without learning.

Future Work: Re-optimize

- Static queries are bound to a plan during query compilation. Dynamic queries stored in statement cache.
- LEO does not rebind static queries or flush statement cache – repeated queries won't get re-optimized.

  - When should cached plans be re-optimized?

Future Work

- Learn other parameters:
  - memory allocation,
  - parameters in the cost model,
  - physical parameters (disk seek time, network latency, etc.)
Conclusions on Dynamic Re-Optimization & LEO

What are the contributions of each?

Advantages / Disadvantages?

Comparison

<table>
<thead>
<tr>
<th>Dynamic Re-optimization</th>
<th>LEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Improves complex queries only (4 or more joins)</td>
<td>- Comprehensive approach, applies to everything. Best improvement on the most used part of the DB</td>
</tr>
<tr>
<td>- Operates on a per-query basis - if it re-optimizes a query, and then later that query is sent again, it will redo all the work to re-optimize again.</td>
<td>- Learn from previous mistakes. Future queries continue to improve.</td>
</tr>
<tr>
<td>- Paper shows clear results on improvements for complex queries.</td>
<td>- Performance analysis in paper is brief and lacking in details</td>
</tr>
</tbody>
</table>
Conclusions

- **Dynamic Re-optimization** is a basis for future research on mid-query re-optimization
  - Practical use: DBMS's with complex queries and rapidly changing data.
- **DB2 Learning Optimizer** – comprehensive approach that can potentially improve all components of a DBMS.
  - But how well does it actually work? No definitive answers yet; more research needed.
  - Practical use: any DBMS, but especially ones with query loads that reuse sections of the data.