Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?

Wentao Wu\textsuperscript{1}, Yun Chi\textsuperscript{2}, Shenghuo Zhu\textsuperscript{2}, Junichi Tatemura\textsuperscript{2}, Hakan Hacigumus\textsuperscript{2}, Jeffrey Naughton\textsuperscript{1}

\textsuperscript{1}Dept of Computer Sciences, University of Wisconsin-Madison
\textsuperscript{2}NEC Laboratories America
Motivation

- Database as a service (DaaS)

How to predict the execution time of a query before it runs?
Applications

- Admission control
  - Run this query or not?

- Query scheduling
  - If we decide to run it, when?

- Progress monitoring
  - How long should we wait if something is wrong?

- System sizing
  - How much hardware does it require to run in the given time?
Use Optimizers’ Cost Estimates?

- Query optimizers have cost estimates for queries.
  - Can we just use them?

- Previous work ([Ganapathi ICDE’09], [Akdere ICDE’12])
  - Query optimizers’ cost estimates are unusable.

Naïve Scaling:
Predict the execution time $T$ by scaling the cost estimate $C$, i.e., $T = a \cdot C$

Fig. 5 of [Akdere ICDE’12]
Why Does Naïve Scaling Fail?

- PostgreSQL’s cost model

\[ C = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o \]

**Naïve Scaling**

\[ T = a \cdot C = c_s' \cdot \left( n_s + n_r \cdot \frac{c_r}{c_s} + n_t \cdot \frac{c_t}{c_s} + n_i \cdot \frac{c_i}{c_s} + n_o \cdot \frac{c_o}{c_s} \right) \]

\[ c_s' = a \cdot c_s = a \cdot 1.0 = a \]

- The assumptions required (for naïve scaling to work)
  - The *ratios* between the c’s are correct.
  - The n’s are correct.
Beat Naïve Scaling

- PostgreSQL’s cost model
  \[ C = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o \]

Unfortunately, both the c’s and the n’s could be incorrect!

- To beat naïve scaling
  - Use machine learning ([Ganapathi ICDE’09], [Akdere ICDE’12])
  - **Calibrate** the c’s and the n’s! (our work)
What if We Use Calibrated $c$’s and $n$’s?

- Cost models become much more effective.

Prediction by Naïve Scaling:

\[ T_{pred} = a \cdot (\sum c \cdot n) \]

Prediction by Calibration:

\[ T_{pred} = \sum c' \cdot n' \]
Main Idea

- How can we calibrate the $c$’s and the $n$’s?
  - Calibrate the $c$’s: *use profiling queries.*
  - Calibrate the $n$’s: *refine cardinality estimates.*
Contribution of This Work

- We proposed a systematic framework to calibrate the cost models used by the query optimizer.

- We showed that the calibrated cost model is much better than naively scaling the cost estimates.

- We further showed that the calibrated cost model is also much better than the state-of-the-art machine-learning based approaches.
Calibrating The c’s

- Basic idea (an example)
  - Want to know the true $c_t$ and $c_o$
  
```markdown
q₁: select * from R
q₂: select count(*) from R
```

- General case
  - $k$ cost units (i.e., $k$ unknowns) => $k$ queries (i.e., $k$ equations)
  - $k = 5$ in the case of PostgreSQL

<table>
<thead>
<tr>
<th>Cost Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_s$: seq_page_cost</td>
</tr>
<tr>
<td>$c_r$: rand_page_cost</td>
</tr>
<tr>
<td>$c_t$: cpu_tuple_cost</td>
</tr>
<tr>
<td>$c_i$: cpu_index_tuple_cost</td>
</tr>
<tr>
<td>$c_o$: cpu_operator_cost</td>
</tr>
</tbody>
</table>

$R$ in memory

\[
t_1 = c_t \cdot n_t \\
t_2 = c_t \cdot n_t + c_o \cdot n_o
\]
How to Pick Profiling Queries?

- Completeness
  - Each $c$ should be covered by at least one query.

- Conciseness
  - The set of queries is *incomplete* if any query is removed.

- Simplicity
  - Each query should be as *simple* as possible.
**Profiling Queries For PostgreSQL**

*Isolate* the unknowns and solve them *one per equation!*

- **q₁**: select * from R  
  \[ t₁ = cₜ \cdot nₜ₁ \]
- **q₂**: select count(*) from R  
  \[ t₂ = cₜ \cdot nₜ₂ + cₒ \cdot nₒ₂ \]
- **q₃**: select * from R where R.A < a (R.A with an Index)  
  \[ t₃ = cₜ \cdot nₜ₃ + cᵢ \cdot nᵢ₃ + cₒ \cdot nₒ₃ \]
- **q₄**: select * from R  
  \[ t₄ = cₛ \cdot nₛ₄ + cₜ \cdot nₜ₄ \]
- **q₅**: select * from R where R.B < b (R.B *unclustered* Index)  
  \[ t₅ = cₛ \cdot nₛ₅ + cᵣ \cdot nᵣ₅ + cₜ \cdot nₜ₅ + cᵢ \cdot nᵢ₅ + cₒ \cdot nₒ₅ \]
Calibrating The $n$’s

- The $n$’s are functions of $N$’s (i.e., input cardinalities).
- Calibrating the $n$’s $\Rightarrow$ Calibrating the $N$’s

### Example 1 (In-Memory Sort)

$sc = (2 \cdot N_t \cdot \log N_t) \cdot c_o + tc \text{ of child}$

$rc = c_t \cdot N_t$

### Example 2 (Nested-Loop Join)

$sc = \text{sc of outer child} + \text{sc of inner child}$

$rc = c_t \cdot (N_t^o \cdot N_t^i) + N_t^o \cdot \text{rc of inner child}$

$sc$: start-cost  \hspace{1cm} $rc$: run-cost  \hspace{1cm} $tc = sc + rc$: total-cost

$N_t$: # of input tuples
## Refine Cardinality Estimates

- Cardinality Estimation

<table>
<thead>
<tr>
<th></th>
<th>Traditional Role (Query Optimization)</th>
<th>Our Case (Execution Time Prediction)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Plans</td>
<td>Hundreds/Thousands of</td>
<td>1</td>
</tr>
<tr>
<td>Time per Plan</td>
<td>Must be very short</td>
<td>Can be a bit longer</td>
</tr>
<tr>
<td>Precision</td>
<td>Important</td>
<td>Critical</td>
</tr>
<tr>
<td>Approach</td>
<td>Histograms (dominant)</td>
<td>Sampling (one option)</td>
</tr>
</tbody>
</table>
A Sampling-Based Estimator

- Estimate the selectivity $\rho_q$ of a select-join query $q$.


$q : R_1 \bowtie R_2$

Partition

$n$ samples (w/ replacement)

\[ \rho_1 = \frac{|B_{11} \bowtie B_{22}|}{|B_{11}| \times |B_{22}|} \]

\[ \hat{\rho}_q = \frac{1}{n} \sum_{i=1}^{n} \rho_i \]

The estimator $\hat{\rho}_q$ is unbiased and strongly consistent!
The Cardinality Refinement Algorithm

- Design the algorithm based on the previous estimator.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Our Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The estimator needs <em>random</em> I/Os at <em>runtime</em> to take samples.</td>
<td>1. Take samples <em>offline</em> and store them as tables in the database.</td>
</tr>
<tr>
<td>2. Query plans usually contain <em>more than one</em> operators.</td>
<td>2. Estimate multiple operators in a <em>single</em> run, by <em>reusing</em> partial results.</td>
</tr>
<tr>
<td>3. The estimator only works for <em>select/join</em> operators.</td>
<td>3. Rely on PostgreSQL’s cost models for <em>aggregates</em>.</td>
</tr>
<tr>
<td></td>
<td><em>Future work:</em> Add estimators for aggregates ([Charikar PODS’00]).</td>
</tr>
</tbody>
</table>
Plan for $q$: $q_1 = R_1 \bowtie R_2$
$q_2 = R_1 \bowtie R_2 \bowtie R_3$

For $agg$, use PostgreSQL’s estimates based on the *refined* input estimates from $q_2$. 

$R_1^S, R_2^S, R_3^S$ are samples (as tables) of $R_1, R_2, R_3$
Experimental Settings

- PostgreSQL 9.0.4, Linux 2.6.18

- TPC-H 1GB and 10GB databases
  - Both uniform and skewed data distribution

- Two different hardware configurations
  - PC1: 1-core 2.27 GHz Intel CPU, 2GB memory
  - PC2: 8-core 2.40 GHz Intel CPU, 16GB memory
## Calibrating Cost Units

### PC1:

<table>
<thead>
<tr>
<th>Cost Unit</th>
<th>Calibrated (ms)</th>
<th>Calibrated (normalized to $c_s$)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_s$: seq_page_cost</td>
<td>5.53e-2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>$c_r$: rand_page_cost</td>
<td>6.50e-2</td>
<td>1.2</td>
<td>4.0</td>
</tr>
<tr>
<td>$c_t$: cpu_tuple_cost</td>
<td>1.67e-4</td>
<td>0.003</td>
<td>0.01</td>
</tr>
<tr>
<td>$c_i$: cpu_index_tuple_cost</td>
<td>3.41e-5</td>
<td>0.0006</td>
<td>0.005</td>
</tr>
<tr>
<td>$c_o$: cpu_operator_cost</td>
<td>1.12e-4</td>
<td>0.002</td>
<td>0.0025</td>
</tr>
</tbody>
</table>

### PC2:

<table>
<thead>
<tr>
<th>Cost Unit</th>
<th>Calibrated (ms)</th>
<th>Calibrated (normalized to $c_s$)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_s$: seq_page_cost</td>
<td>5.03e-2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>$c_r$: rand_page_cost</td>
<td>4.89e-1</td>
<td>9.7</td>
<td>4.0</td>
</tr>
<tr>
<td>$c_t$: cpu_tuple_cost</td>
<td>1.41e-4</td>
<td>0.0028</td>
<td>0.01</td>
</tr>
<tr>
<td>$c_i$: cpu_index_tuple_cost</td>
<td>3.34e-5</td>
<td>0.00066</td>
<td>0.005</td>
</tr>
<tr>
<td>$c_o$: cpu_operator_cost</td>
<td>7.10e-5</td>
<td>0.0014</td>
<td>0.0025</td>
</tr>
</tbody>
</table>
Prediction Precision

- Metric of precision
  - Mean Relative Error (MRE)

- Dynamic database workloads
  - Unseen queries frequently occur.

- Compare with existing approaches
  - Naive scaling
  - More complex machine learning approaches
Existing Machine-Learning Methods

- The idea
  - Represent a query as a feature vector
  - Train a regression model

- SVM [Akdere ICDE’12]

- REP trees [Xiong SoCC’11]

- KCCA [Ganapathi ICDE’09]
  - Did not compare since [Akdere ICDE’12] is better.
Precision on TPC-H 1GB DB

Uniform data:

- $E_t$: c’s (calibrated) + n’s (true cardinalities)
- $E_o$: c’s (calibrated) + n’s (cardinalities by optimizer)
- $E_s$: c’s (calibrated) + n’s (cardinalities by sampling)

Naïve Scaling
Skewed data:

- $E_t$: $c$’s (calibrated) + $n$’s (true cardinalities)
- $E_o$: $c$’s (calibrated) + $n$’s (cardinalities by optimizer)
- $E_s$: $c$’s (calibrated) + $n$’s (cardinalities by sampling)
Uniform data (similar results on skewed data):

\[ E_t : c's \text{ (calibrated)} + n's \text{ (true cardinalities)} \]
\[ E_o : c's \text{ (calibrated)} + n's \text{ (cardinalities by optimizer)} \]
\[ E_s : c's \text{ (calibrated)} + n's \text{ (cardinalities by sampling)} \]
Overhead of Sampling

- Additional overhead is measured as $\frac{t_{\text{sampling}}}{t_{\text{query}}}$

- More samples mean higher additional overhead

- For close-to-ideal prediction on 1GB DB
  - 30% samples (0.3GB) => 20% additional overhead

- For close-to-ideal prediction on 10GB DB
  - 5% samples (0.5GB) => 4% additional overhead
Conclusion

- We presented a systematic framework to calibrate the cost units and refine the cardinality estimates used by current cost models.

- We showed that current cost models are much more effective in query execution time prediction after proper calibration, and the additional overhead is affordable in practice.