UNCERTAINTY-AWARE QUERY EXECUTION TIME PREDICTION

Wentao Wu\textsuperscript{1,2}, Xi Wu\textsuperscript{2}, Hakan Hacigumus\textsuperscript{3}, Jeff Naughton\textsuperscript{2}

\textsuperscript{1}Microsoft Research
\textsuperscript{2}University of Wisconsin-Madison
\textsuperscript{3}Google
Problem Definition

- Given a query, estimate its running time *before* it runs.
- Focus on OLAP style, *long-running* queries.

Applications

- Traditionally, cost-based query optimization.
- Recently, database as a service (DaaS): admission control, query scheduling, system sizing, …
Previous Work

- Single-Query Workload
  - [Ganapathi ICDE’09], [Xiong SoCC’11], [Akdere ICDE’12], [Li VLDB’12], [Wu ICDE’13]

- Multi-Query Workload
  - [Ahmad EDBT’11], [Duggan SIGMOD’11], [Wu VLDB’13]

None of them is perfect, but none of them tried to quantify the uncertainty in the estimated query execution time.
Motivation

- Estimates are more useful with confidence intervals.

Measure Uncertainty: point estimate $\Rightarrow$ distribution
What do we mean by “distribution of likely query execution times”?

**Interpretation 1:** If we run the query 100 times, what will be the distribution of its running times?

**Interpretation 2:** If we run the query now, what is the likelihood that it can finish between 100s and 200s?
Applications

- Query optimization
  - Least-Expected-Cost query optimization [Chu PODS’99]
  - Robust Query Optimization [Babcock SIGMOD’05]

- Query progress monitoring
  - Provide error bars for the “remaining” query running time.

- Database as a service
  - Distribution-based query scheduling [Chi VLDB’13].
Our Idea

- PostgreSQL's cost model

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

- `c's`: cost units
- `n's`: functions of *cardinality* estimates

In our previous work [Wu ICDE’13], we proposed a framework that *calibrates* the `c's` and *refines* the `n's` to get *better* query execution time estimates.

<table>
<thead>
<tr>
<th>Cost Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c_s</code>: seq_page_cost</td>
<td>1.0</td>
</tr>
<tr>
<td><code>c_r</code>: rand_page_cost</td>
<td>4.0</td>
</tr>
<tr>
<td><code>c_t</code>: cpu_tuple_cost</td>
<td>0.01</td>
</tr>
<tr>
<td><code>c_i</code>: cpu_index_tuple_cost</td>
<td>0.005</td>
</tr>
<tr>
<td><code>c_o</code>: cpu_operator_cost</td>
<td>0.0025</td>
</tr>
</tbody>
</table>
Our Idea (Cont.)

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

View the c’s and the n’s as random variables rather than constants!

\[
\begin{align*}
Pr (c) & \quad \text{Pr (n)} \\
Pr (t) & \quad t = n \cdot c
\end{align*}
\]
Outlines

- The calibration framework
- Distributions of the c’s
- Distributions of the n’s
- Summary
The Calibration Framework

- Calibrate the $c$’s: use calibration queries.
- Refine the $n$’s: refine cardinality estimates.

Sampling is used only once for the final chosen plan!
Outlines

- The calibration framework
- Distributions of the c’s
- Distributions of the n’s
- Summary
Calibrate The c’s

Basic idea (an example)
- Want to know the true values of \( c_t \) and \( c_o \) via calibration queries.

| q₁: select * from R
| q₂: select count(*) from R

<table>
<thead>
<tr>
<th>R in memory</th>
</tr>
</thead>
</table>

<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>

General case
- \( k \) cost units (i.e., \( k \) unknowns) \( \Rightarrow \) \( k \) queries (i.e., \( k \) equations)
- \( k = 5 \) in the case of PostgreSQL
Calibration Queries For PostgreSQL

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

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁: select * from R</td>
<td>R in memory</td>
<td>( t₁ = c_τ \cdot n_{t_1} )</td>
</tr>
<tr>
<td>q₂: select count(*) from R</td>
<td>R in memory</td>
<td>( t₂ = c_τ \cdot n_{t_2} + c_o \cdot n_{o_2} )</td>
</tr>
<tr>
<td>q₃: select * from R where R.A &lt; a (R.A with an Index)</td>
<td>R in memory</td>
<td>( t₃ = c_τ \cdot n_{t_3} + c_i \cdot n_{i_3} + c_o \cdot n_{o_3} )</td>
</tr>
<tr>
<td>q₄: select * from R</td>
<td>R on disk</td>
<td>( t₄ = c_s \cdot n_{s_4} + c_τ \cdot n_{t_4} )</td>
</tr>
<tr>
<td>q₅: select * from R where R.B &lt; b (R.B unclustered Index)</td>
<td>R on disk</td>
<td>( t₅ = c_s \cdot n_{s_5} + c_r \cdot n_{r_5} + c_τ \cdot n_{t_5} + c_i \cdot n_{i_5} + c_o \cdot n_{o_5} )</td>
</tr>
</tbody>
</table>

For each c, use *multiple* queries and take the *average*. 
Distributions of the c’s

- Assumption: $c \sim N(\mu, \sigma^2)$

$m$ calibration queries for each $c$ $(\mu_m, \sigma_m^2)$
Outlines

- The calibration framework
- Distributions of the c’s
- Distributions of the n’s
- Summary
Refine The $n$’s

- The $n$’s are *functions* of $N$’s (i.e., input cardinalities).

**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)

\[
sc = \text{sc of outer child} + \text{sc of inner child} \\
rc = c_t \cdot N_t^o \cdot N_t^i + N_t^o \cdot rc \text{ of inner child}
\]

- $sc$: start-cost
- $rc$: run-cost
- $tc$: total-cost
- $N_t$: # of input tuples
Distributions of the $n$’s

- We need to model two quantities:
  - The selectivities;
  - The cost functions for different physical operators.

- Using mathematics we get distributions of the $n$’s.
A Sampling-Based Selectivity Estimator

- Estimate the selectivity $\rho_q$ of a join query $q = R_1 \bowtie R_2$.


Do a "cross product" over the samples: $\rho(i, j) = 0 \text{ or } 1$.

The $\rho$'s are not independent, but the estimator $\hat{\rho}_q$ is still unbiased and strongly consistent.
Distributions of Selectivities

- Selectivity \( \sim N(\mu_n, S^2_n) \): by the Central Limit Theorem.

\[
S^2_n = \sum_{k=1}^{K} \left( \frac{1}{n-1} \sum_{j=1}^{n} \left( \frac{Q_{k,j,n}}{n^{K-1}} - \mu_n \right)^2 \right)
\]
Modeling Cost Functions

(Different Implementations)

Nested-Loop Join

**Generic Cost Function**: \( f = a_0 N_l N_r + a_1 N_l + a_2 N_r + a_3 \)

\( N_l \) and \( N_r \) are the left and right input cardinality of the operator.
Distributions of the $n$’s and $t$

- The $n$’s are \textit{asymptotically} Gaussian.
  - More samples $\Rightarrow$ more close to Gaussian

- The running time $t$ is also \textit{asymptotically} Gaussian.

\textbf{Example:} Nested Loop Join

$$t = n_r c_r + n_t c_t \quad \Rightarrow \quad t \sim N(E[t], \text{Var}[t])$$

\begin{align*}
  n_r &= a_0 N_l N_r + a_1 N_l + a_2 N_r + a_3 \\
  n_t &= b_0 N_l N_r + b_1 N_l + b_2 N_r + b_3
\end{align*}

$n_r$ and $n_t$ are \textit{not} independent!

\textbf{Should consider covariances when computing \text{Var}[t]!}
Outlines

- The calibration framework
- Distributions of the c’s
- Distributions of the n’s
- Summary
Put It Together (Review)

**Pr (selectivity) (Gaussian)**

**Cost Model**

\[ t = c \cdot n \]

**Pr(t) (Asymptotically Gaussian)**

**Generic Cost Functions**

**Pr(n) (Asymptotically Gaussian)**

**Convergence Theorems**
The idea: larger variances $\rightarrow$ larger estimation errors

We thus measure the *correlation* between the two.

We observed *strong* correlations (i.e., correlation coefficient $> 0.7$) on almost all the queries tested in our experiments.
Thank you 😊
The Cardinality Refinement Algorithm (Example)

Plan of q:

\[ J_1 = R_1 \bowtie R_2 \]
\[ J_2 = R_1 \bowtie R_2 \bowtie R_3 \]

Rewrite

\[ J_1 = R_1 \bowtie R_2 \]
\[ J_2 = R_1 \bowtie R_2 \bowtie R_3 \]

Run

\[ \hat{\rho}_{J_1} = \frac{|R_1^s \bowtie R_2^s|}{|R_1^s| \times |R_2^s|} \]
\[ \hat{\rho}_{J_2} = \frac{|R_1^s \bowtie R_2^s \bowtie R_3^s|}{|R_1^s| \times |R_2^s| \times |R_3^s|} \]

Reuse

\[ R_1^s, R_2^s, R_3^s \] are samples (as tables) of \( R_1, R_2, R_3 \).

For \textit{agg}, use PostgreSQL’s estimates based on the \textit{refined} input estimates from \( J_2 \).
Distributions of Selectivities (Cont.)

- Implementation of $S^2_n$ in PostgreSQL

$$S^2_n = \sum_{k=1}^{K} \left( \frac{1}{n-1} \sum_{j=1}^{n} (Q_{k,j,n} / n^{K-1} - \mu_n)^2 \right)$$

Example: $K = 2$ (i.e., $R_1 \bowtie R_2$)

$$\begin{align*}
Q_{1,j,n} &= |\{r_{1j}\} \bowtie R^s_2| \\
Q_{2,j,n} &= |R^s_1 \bowtie \{r_{2j}\}| \\
\end{align*}$$

$r_{1i}$ and $r_{2i}$ are the $j$-th row of $R^s_1$ and $R^s_2$.

If: $r_{11} \bowtie r_{21} \in R^s_1 \bowtie R^s_2$

Then: $++Q_{1,1,n}$

$++Q_{2,1,n}$

2n joins here!

Only 1 join!