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

Wentao Wu[1], Yun Chi[2], Shenghuo Zhu[2], Junichi Tatemura[2], Hakan Hacigumus[2], Jeffrey Naughton[1]

[1]Dept of Computer Sciences, University of Wisconsin-Madison

[2]NEC Laboratories America

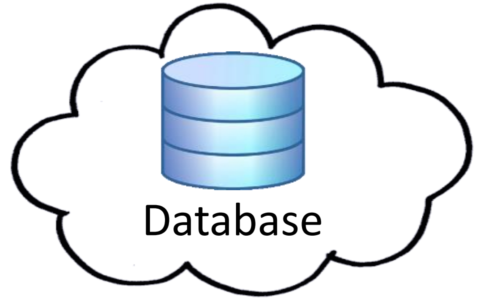
# Motivation

- Database as a service (DaaS)

User


Service Level
Agreement (SLA)

Service Provider




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., $T = a \cdot C$

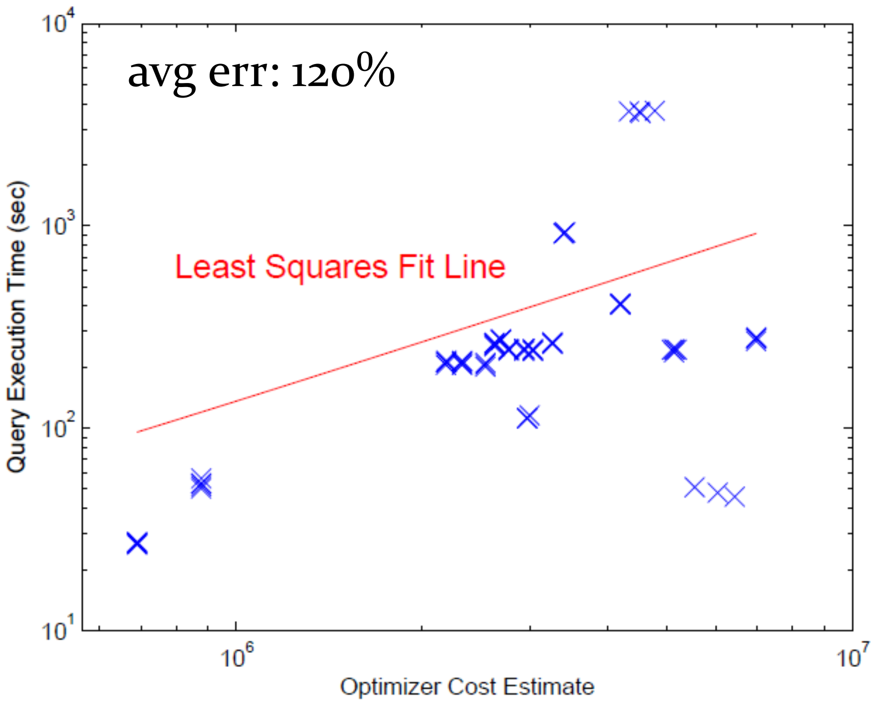



Fig. 5 of [Akdere ICDE'12]

# Why Does Naïve Scaling Fail?

| Cost Unit | Value |
|---|---|
| $c_s$: seq_page_cost | 1.0 |
| $c_r$: rand_page_cost | 4.0 |
| $c_t$: cpu_tuple_cost | 0.01 |
| $c_i$: cpu_index_tuple_cost | 0.005 |
| $c_o$: cpu_operator_cost | 0.0025 |

- PostgreSQL's cost model

$$\boxed{C = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o}$$

Naïve Scaling

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

$$\boxed{c_s' = a \cdot c_s = a \cdot 1.0 = a}$$

Should be correct!

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

# Beat Naïve Scaling

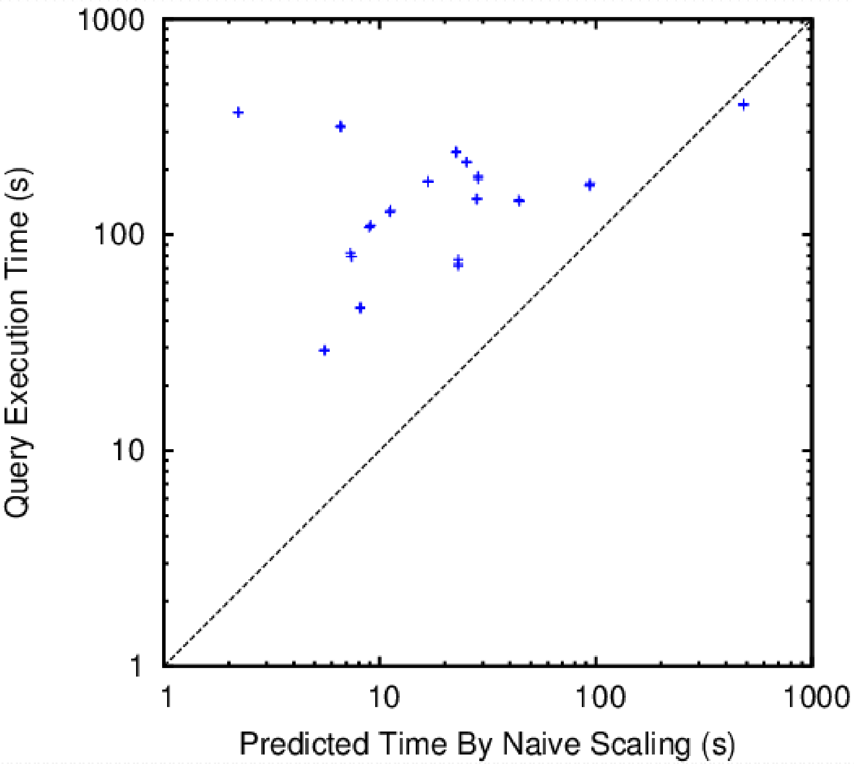
- PostgreSQL's cost model $C = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o$

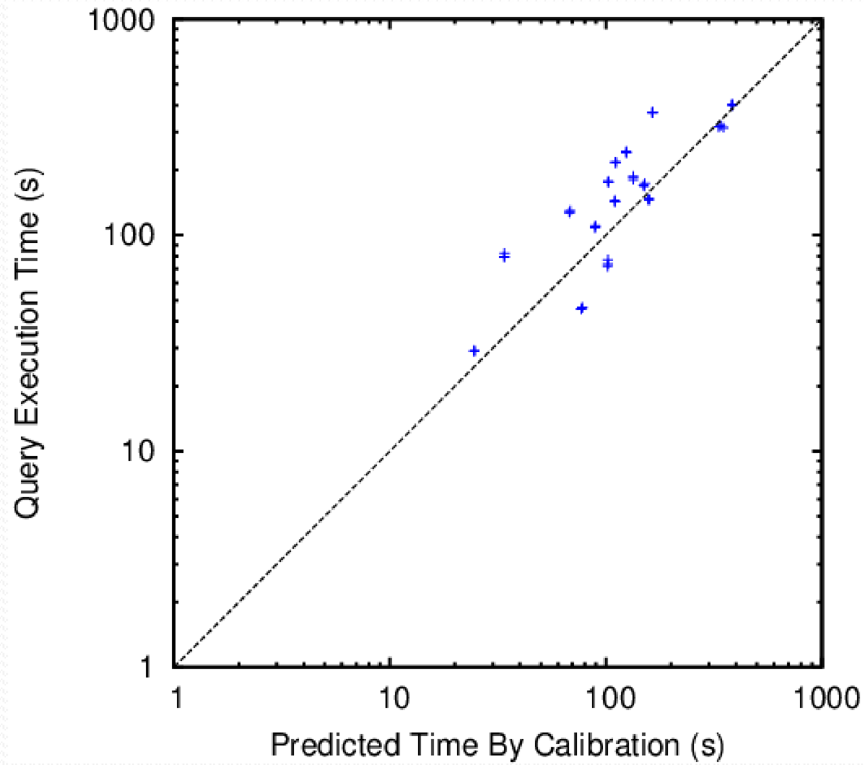
**Unfortunately, *both* the *c*'s and the *n*'s could be *incorrect*!**

- 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:

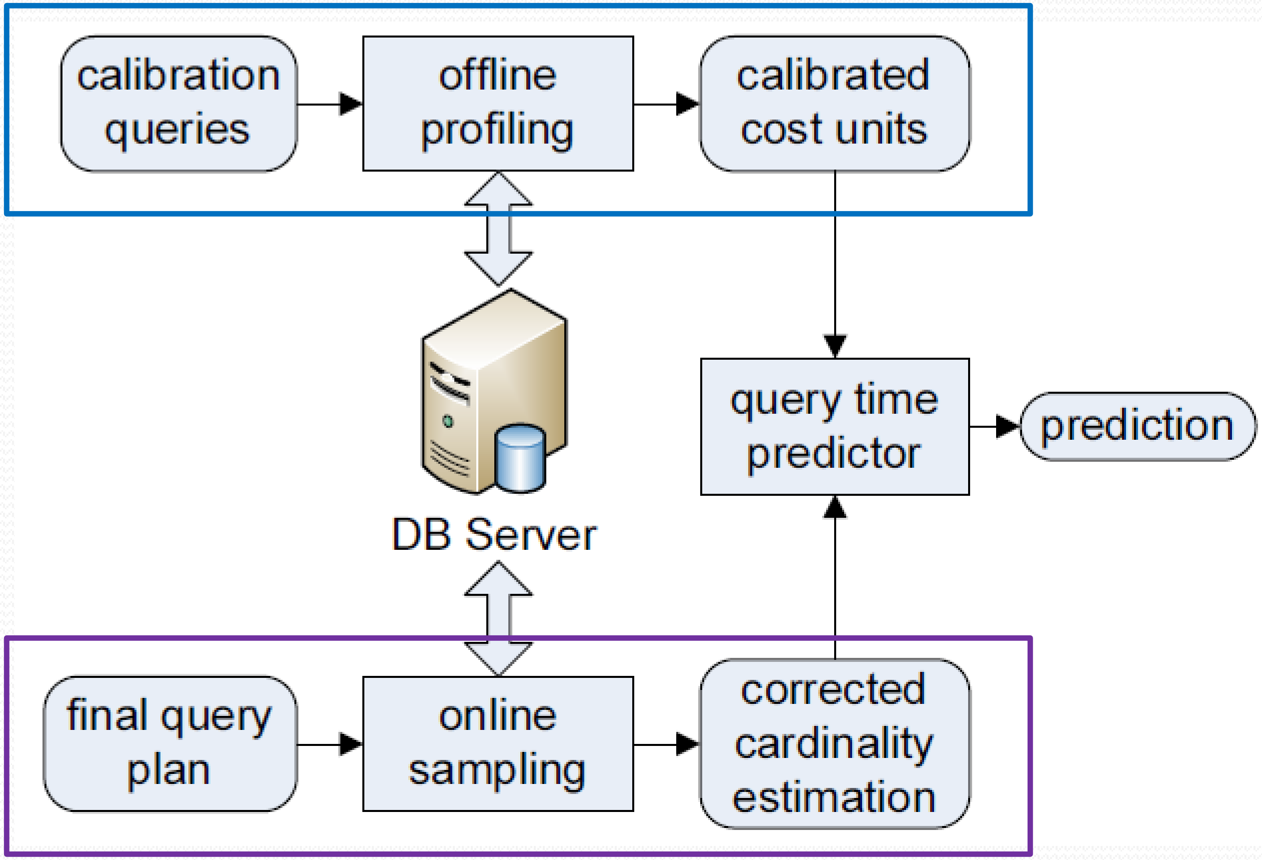
$$T_{pred} = a \cdot (\sum c \cdot n)$$

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 *naïvely 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

| Cost Unit |
|---|
| $c_s$: seq_page_cost |
| $c_r$: rand_page_cost |
| $c_t$: cpu_tuple_cost |
| $c_i$: cpu_index_tuple_cost |
| $c_o$: cpu_operator_cost |

- Basic idea (an example)
  - Want to know the true $c_t$ and $c_o$

**q**$_1$: select * from R
**q**$_2$: select count(*) from R

R in memory $\Longrightarrow$

$$t_1 = c_t \cdot n_t$$
$$t_2 = c_t \cdot n_t + c_o \cdot n_o$$

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

# 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*!

| Query | Setting | Equation |
|---|---|---|
| **q₁**: select * from R | R in memory | $t_1 = c_t \cdot n_{t1}$ |
| **q₂**: select count(*) from R | R in memory | $t_2 = c_t \cdot n_{t2} + c_o \cdot n_{o2}$ |
| **q₃**: select * from R where R.A < a (R.A with an Index) | R in memory | $t_3 = c_t \cdot n_{t3} + c_i \cdot n_{i3} + c_o \cdot n_{o3}$ |
| **q₄**: select * from R | R on disk | $t_4 = c_s \cdot n_{s4} + c_t \cdot n_{t4}$ |
| **q₅**: select * from R where R.B < b (R.B *unclustered* Index) | R on disk | $t_5 = c_s \cdot n_{s5} + c_r \cdot n_{r5} + c_t \cdot n_{t5} + c_i \cdot n_{i5} + c_o \cdot n_{o5}$ |

# Calibrating The *n*'s

- The *n*'s are *functions* of *N*'s (i.e., input cardinalities).
  - Calibrating the *n*'s => Calibrating the *N*'s

**Example 1** (In-Memory Sort)

$$sc = \underbrace{[2 \cdot N_t \cdot \log N_t]}_{n_o} \cdot c_o + tc\ of\ child$$

$$rc = c_t \cdot N_t$$

**Example 2** (Nested-Loop Join)

$$sc = sc\ of\ outer\ child + sc\ of\ inner\ child$$

$$rc = c_t \cdot \underbrace{N_t^o \cdot N_t^i}_{n_t} + N_t^o \cdot rc\ of\ inner\ child$$

$sc$: start-cost  $rc$: run-cost  $tc = sc + rc$: total-cost

$N_t$: # of input tuples

# Refine Cardinality Estimates

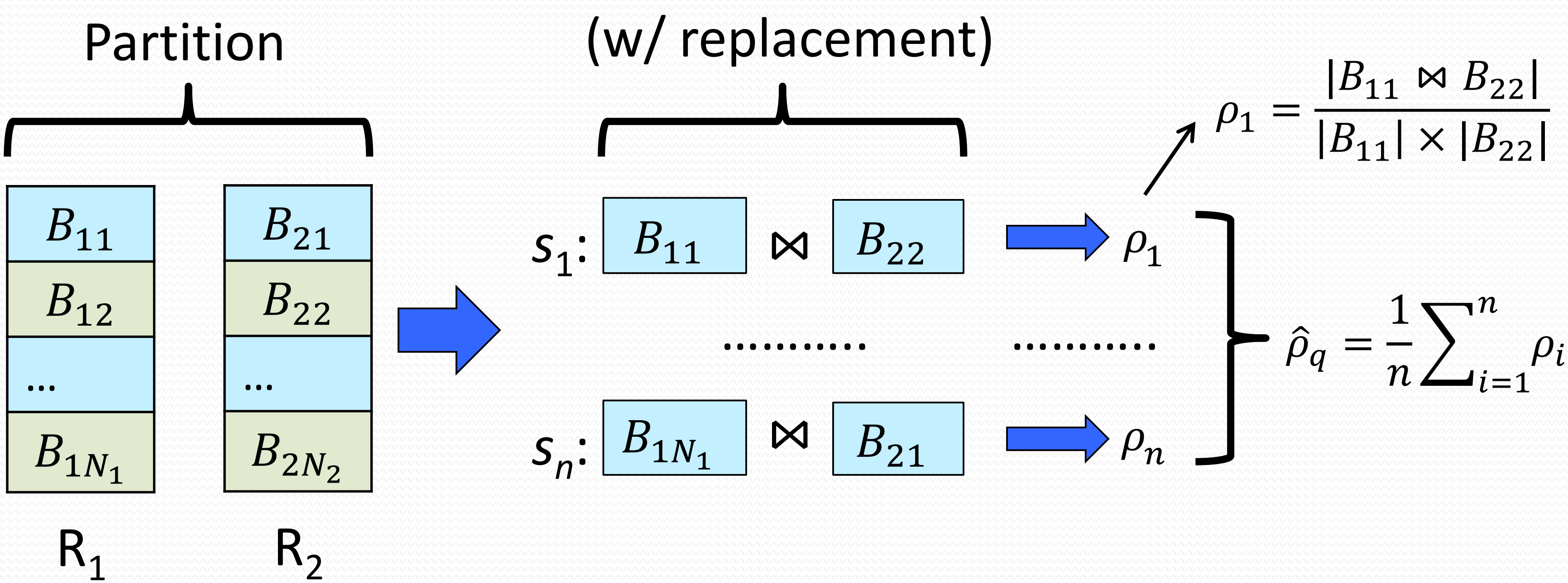
- Cardinality Estimation

| | Traditional Role (Query Optimization) | Our Case (Execution Time Prediction) |
|---|---|---|
| **# of Plans** | Hundreds/Thousands of | 1 |
| **Time per Plan** | Must be very short | Can be a bit *longer* |
| **Precision** | Important | *Critical* |
| **Approach** | Histograms (dominant) | *Sampling* (one option) |

# A Sampling-Based Estimator

- Estimate the *selectivity* $\rho_q$ of a select-join query $q$.
  [Haas et al., J. Comput. Syst. Sci. 1996]

$q : \mathrm{R}_1 \bowtie \mathrm{R}_2$




The estimator $\hat{\rho}_q$ is *unbiased* and *strongly consistent*!

# The Cardinality Refinement Algorithm

- Design the algorithm based on the previous estimator.

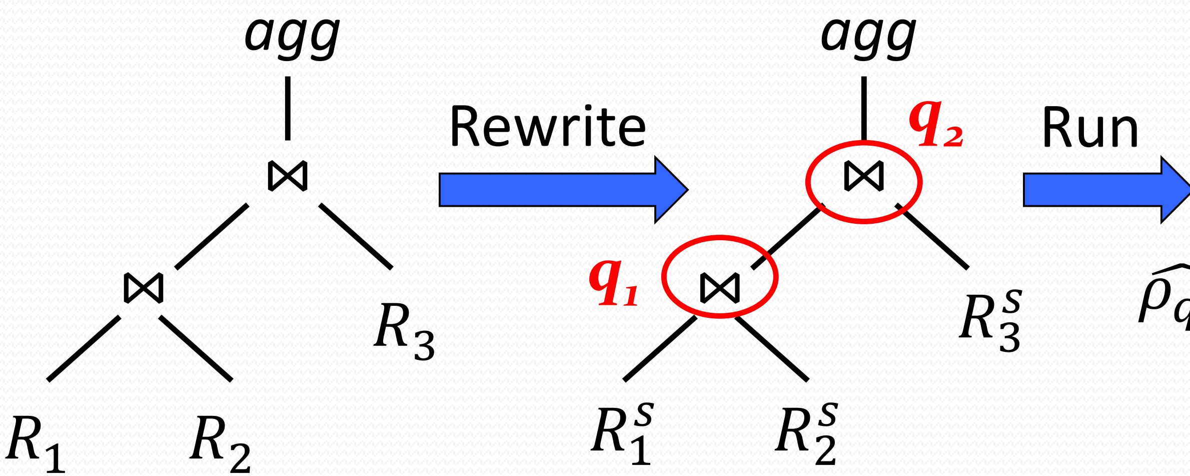
| Problem | Our Solution |
|---|---|
| 1. The estimator needs *random* I/Os at *runtime* to take samples. | 1. Take samples *offline* and store them as tables in the database. |
| 2. Query plans usually contain *more than one* operators. | 2. Estimate multiple operators in a *single* run, by *reusing* partial results. |
| 3. The estimator only works for *select/join* operators. | 3. Rely on PostgreSQL's cost models for *aggregates*.<br><br>*Future work*: Add estimators for aggregates ([Charikar PODS'00]). |

# The Cardinality Refinement Algorithm (Example)

$$q_1 = R_1 \bowtie R_2$$

$$q_2 = R_1 \bowtie R_2 \bowtie R_3$$

Plan for $q$:




$$\widehat{\rho_{q_1}} = \frac{|R_1^s \bowtie R_2^s|}{|R_1^s| \times |R_2^s|}$$

$$\widehat{\rho_{q_2}} = \frac{|R_1^s \bowtie R_2^s \bowtie R_3^s|}{|R_1^s| \times |R_2^s| \times |R_3^s|}$$

Reuse

$R_1^s$, $R_2^s$, $R_3^s$ are samples (as tables) of $R_1$, $R_2$, $R_3$

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

# 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:

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

PC2:

| Cost Unit | Calibrated (ms) | Calibrated (normalized to $c_s$) | Default |
|---|---|---|---|
| $c_s$: seq_page_cost | 5.03e-2 | 1.0 | 1.0 |
| $c_r$: rand_page_cost | 4.89e-1 | 9.7 | 4.0 |
| $c_t$: cpu_tuple_cost | 1.41e-4 | 0.0028 | 0.01 |
| $c_i$: cpu_index_tuple_cost | 3.34e-5 | 0.00066 | 0.005 |
| $c_o$: cpu_operator_cost | 7.10e-5 | 0.0014 | 0.0025 |

# Prediction Precision

- Metric of precision
  - Mean Relative Error (MRE)

$$\frac{1}{M}\sum_{i=1}^{M}\frac{|T_i^{pred} - T_i^{act}|}{T_i^{act}}$$

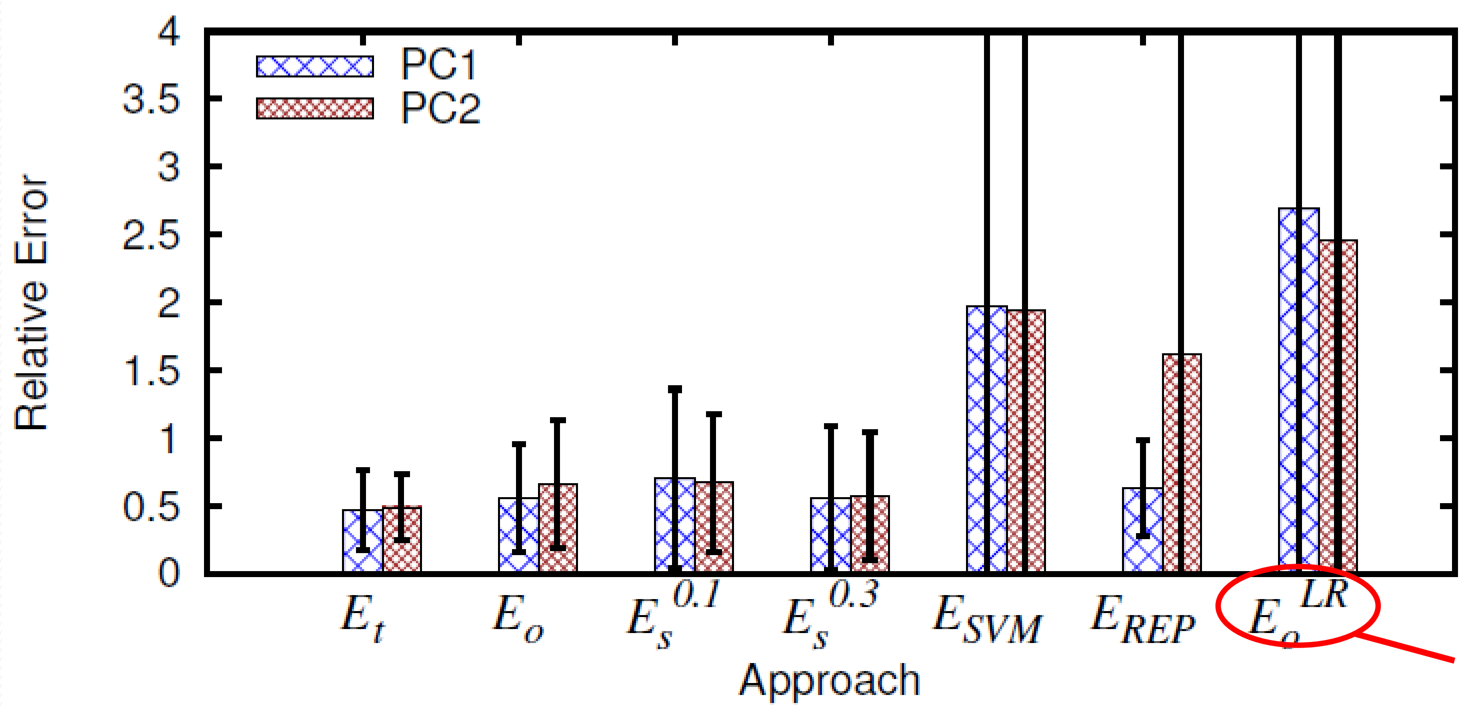
- Dynamic database workloads
  - Unseen queries frequently occur.
- Compare with existing approaches
  - Naive scaling
  - More complex machine learning approaches

# 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:

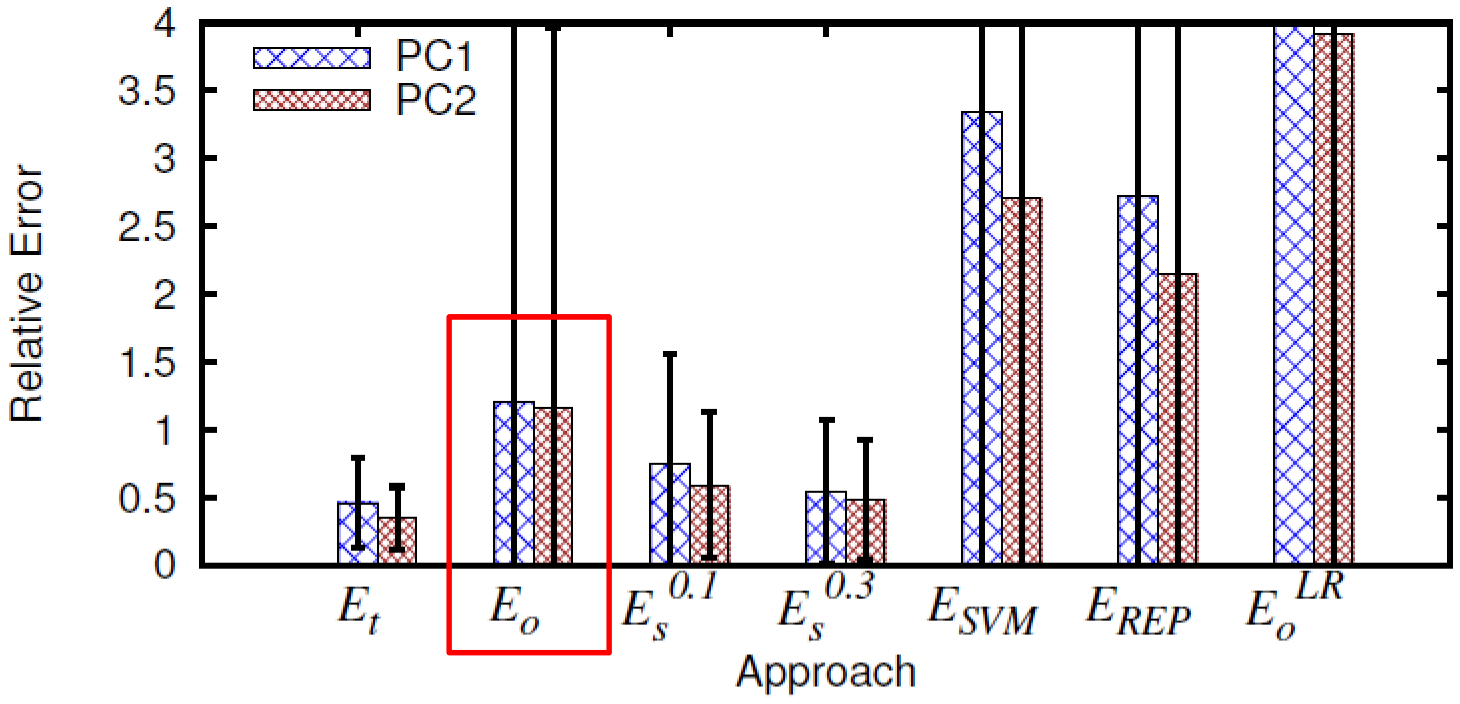



Naïve Scaling

$E_t$: $c$'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*)

# Precision on TPC-H 1GB DB (Cont.)

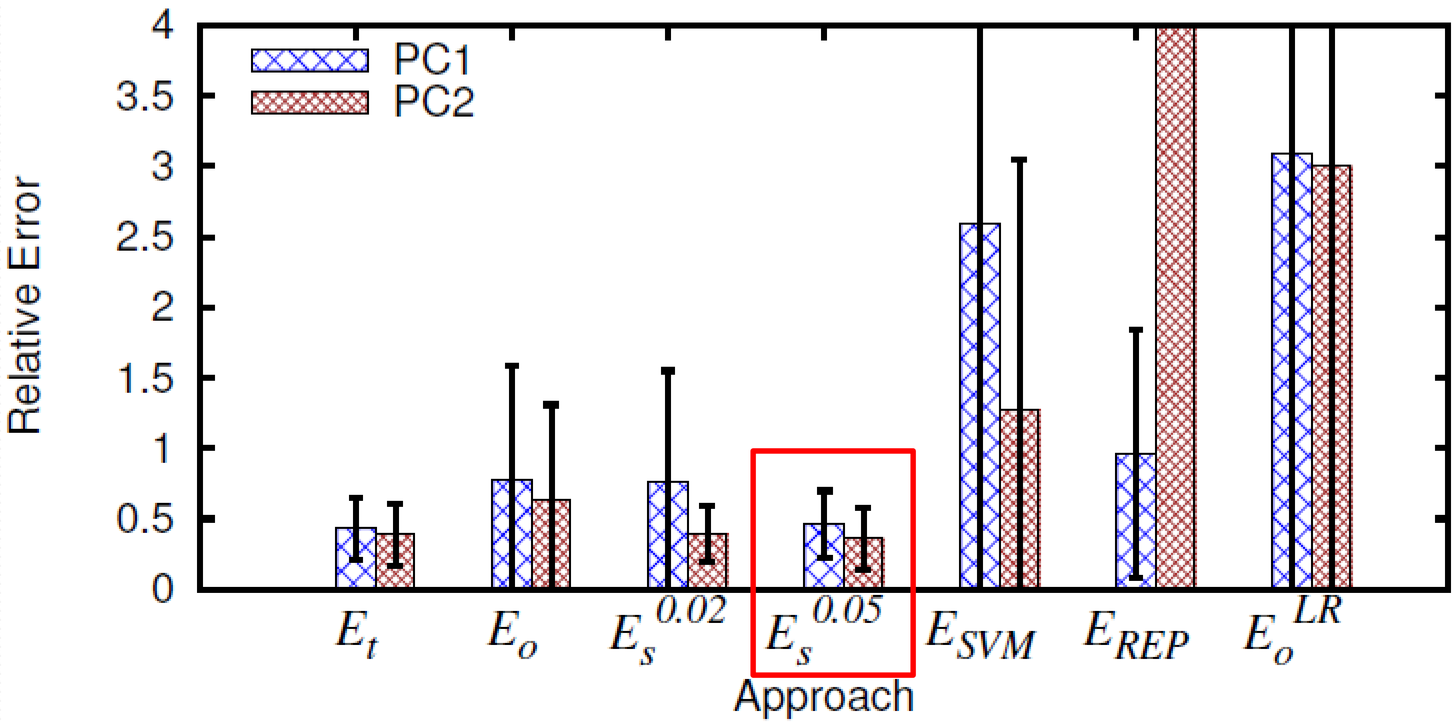
Skewed data:




$E_t$: *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*)

# Precision on TPC-H 10GB DB

Uniform data (similar results on 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*)

# Overhead of Sampling

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