UNCERTAINTY-AWARE QUERY EXECUTION TIME PREDICTION

Wentao Wu[1,2], Xi Wu[2], Hakan Hacigumus[3], Jeff Naughton[2]
[1]Microsoft Research
[2]University of Wisconsin-Madison
[3]Google

Query Execution Time Estimation

- 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, …

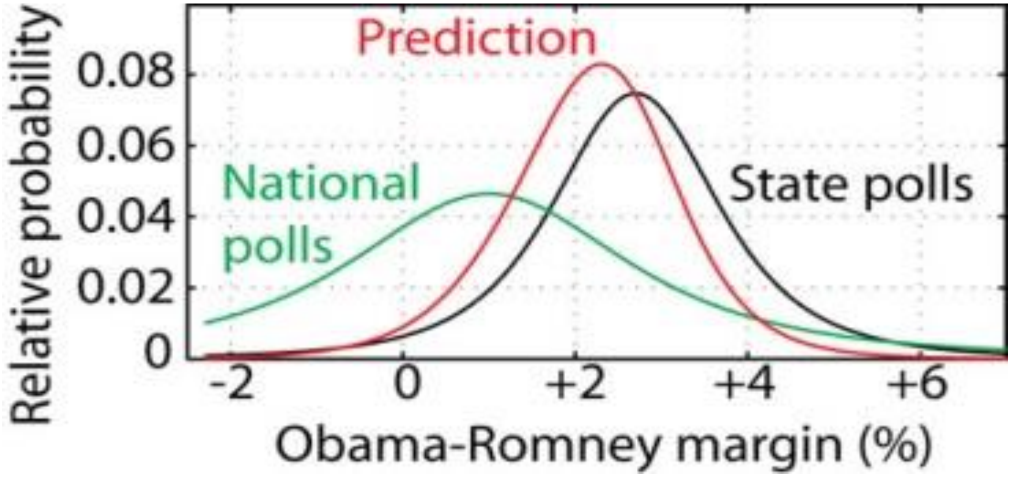
Previous Work

- 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 => distribution

Caveat

- 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

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

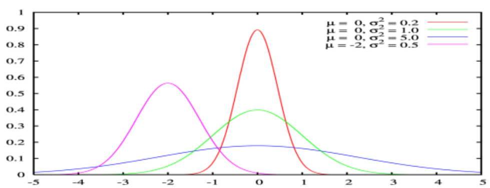
In our previous work [Wu ICDE'13], we proposed a framework that *calibrates* the c's and *refines* the *n*'s to get *better* query execution time estimates.

Our Idea (Cont.)

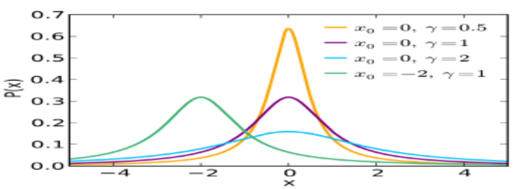
$$\boxed{t = n_s c_s + n_r c_r + n_t c_t + n_i c_i + n_o c_o}$$

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

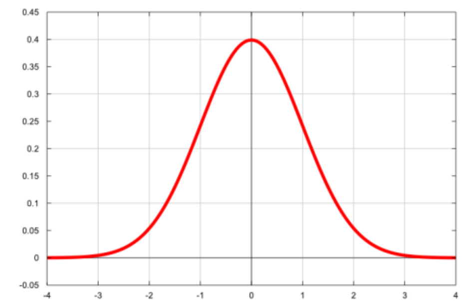
Pr (***c***)



Pr (***n***)



$t = \boldsymbol{n} \cdot \boldsymbol{c}$



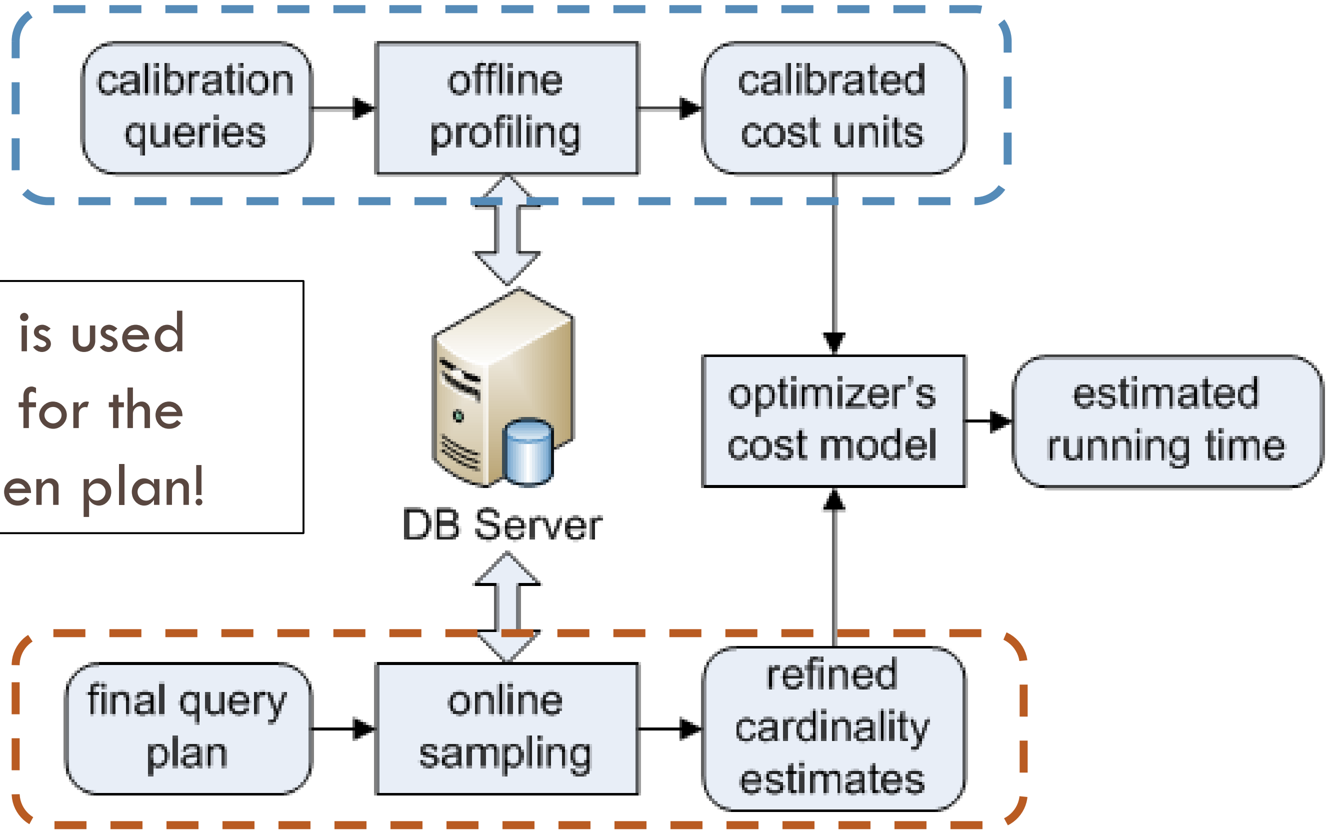
Pr (*t*)

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.*



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

- Refine the *n*'s: *refine cardinality estimates.*

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.

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

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

R in memory ⇒

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

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

Calibration Queries For PostgreSQL

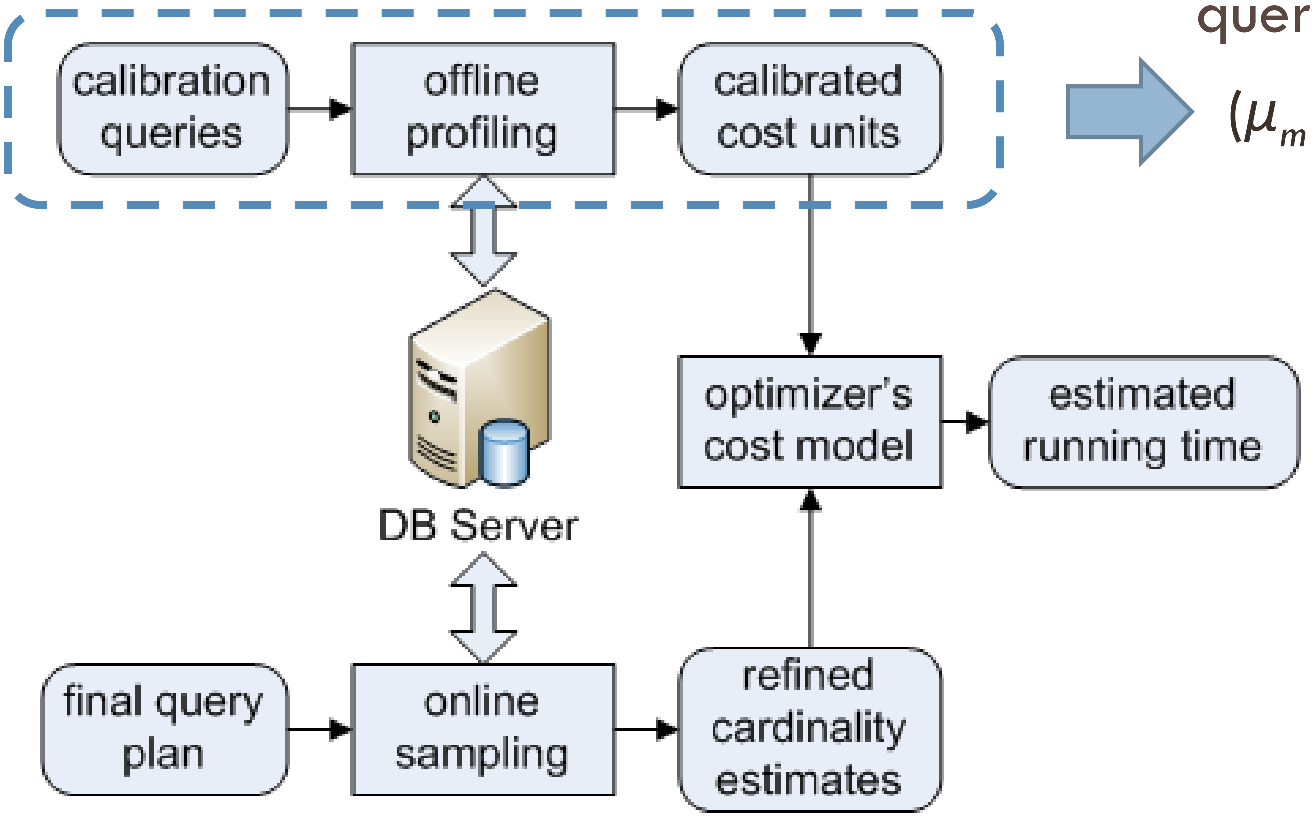
Isolate the unknowns and solve them *one per equation*!

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	$t_5 = c_s \cdot n_{s5} + c_r \cdot n_{r5} + c_t \cdot n_{t5} + c_i \cdot n_{i5} + c_o \cdot n_{o5}$

For each c, use *multiple* queries and take the *average*.

Distributions of the c's

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



m calibration queries for each c

(μ_m, σ_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)

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

$$rc = c_t \cdot N_t$$

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*: 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.

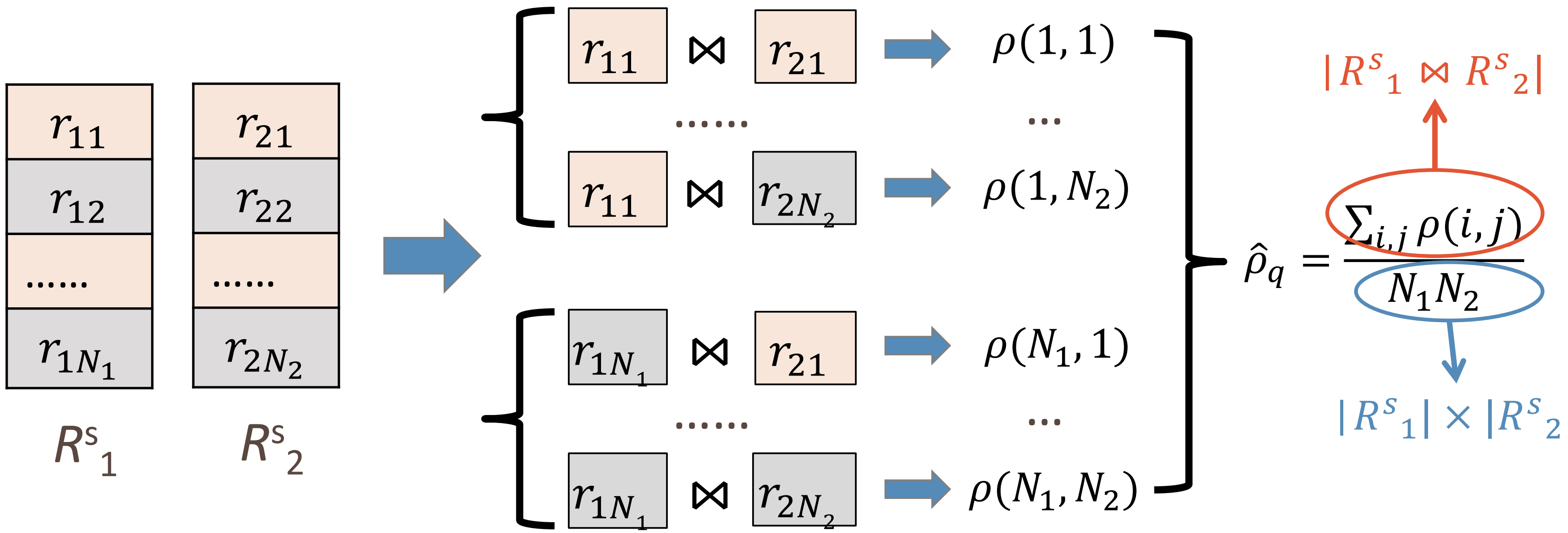
- Using mathematics we get distributions of the *n*'s.

A Sampling-Based Selectivity Estimator

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

[Haas et al., J. Comput. Syst. Sci. 1996]

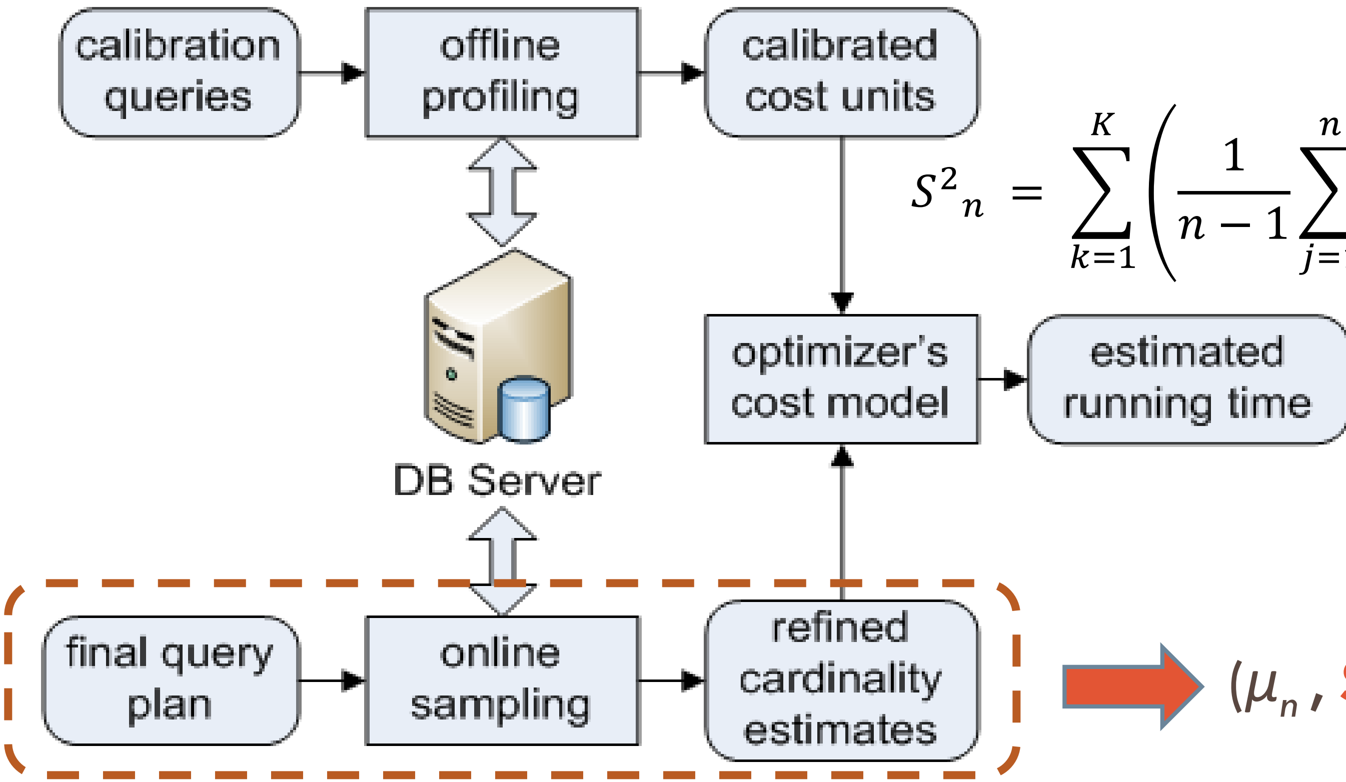
Do a "*cross product*" over the samples: $\rho(i,j) = 0 \; or \; 1$.



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

Distributions of Selectivities

- Selectivity ~ $N(\mu_n, S^2{}_n)$: by the Central Limit Theorem.



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

$(\mu_n, S^2{}_n)$

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 *asymptotically* Gaussian.
  - More samples => more close to Gaussian

- The running time *t* is also *asymptotically* Gaussian.

Example: Nested Loop Join

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

$$\begin{cases} n_r = a_0 N_l N_r + a_1 N_l + a_2 N_r + a_3 \\ n_t = b_0 N_l N_r + b_1 N_l + b_2 N_r + b_3 \end{cases} \Rightarrow$$

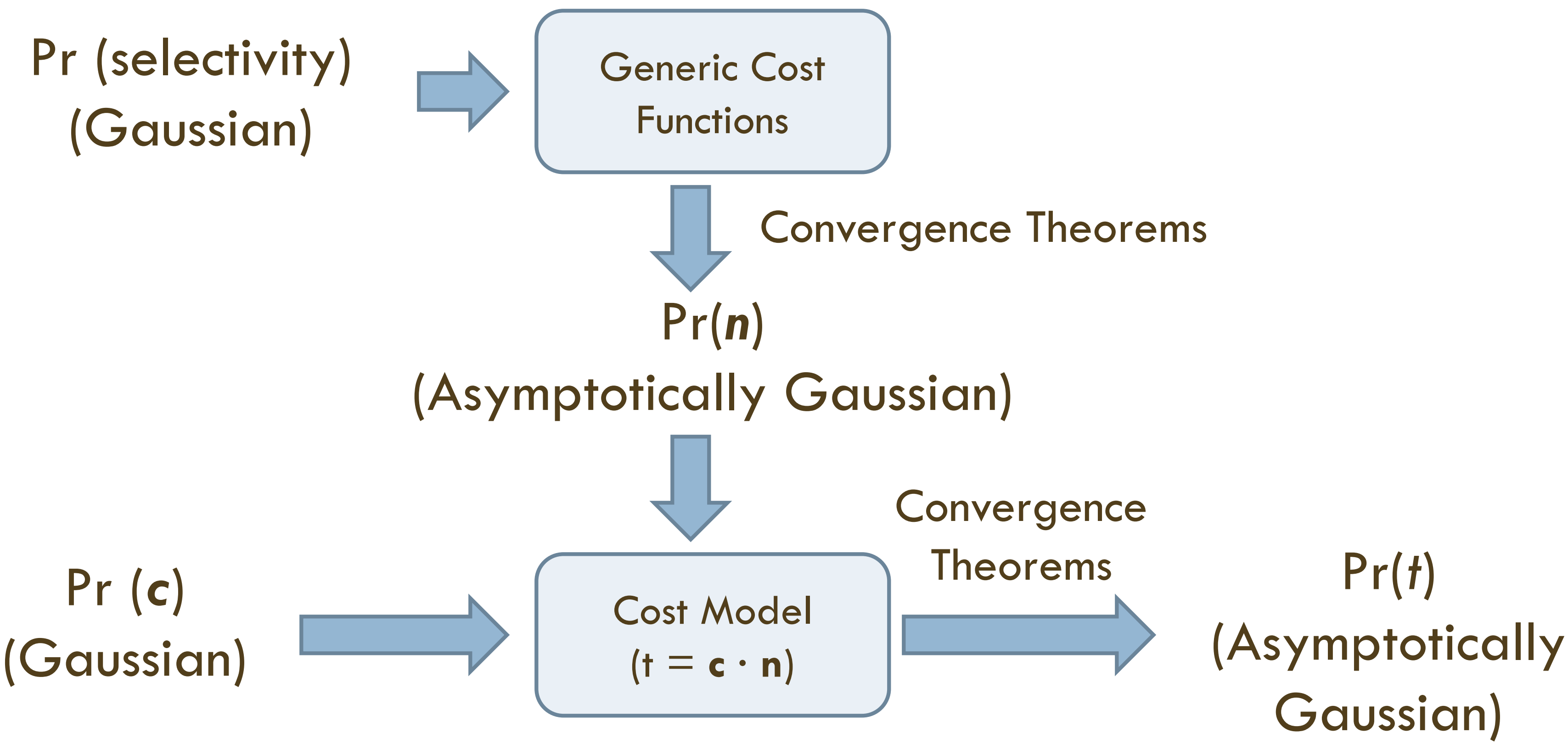
n_r and n_t are *not* independent!

Should consider *covariances* when computing *Var*[*t*]!

Outlines

- The calibration framework
- Distributions of the c's
- Distributions of the *n*'s
- Summary

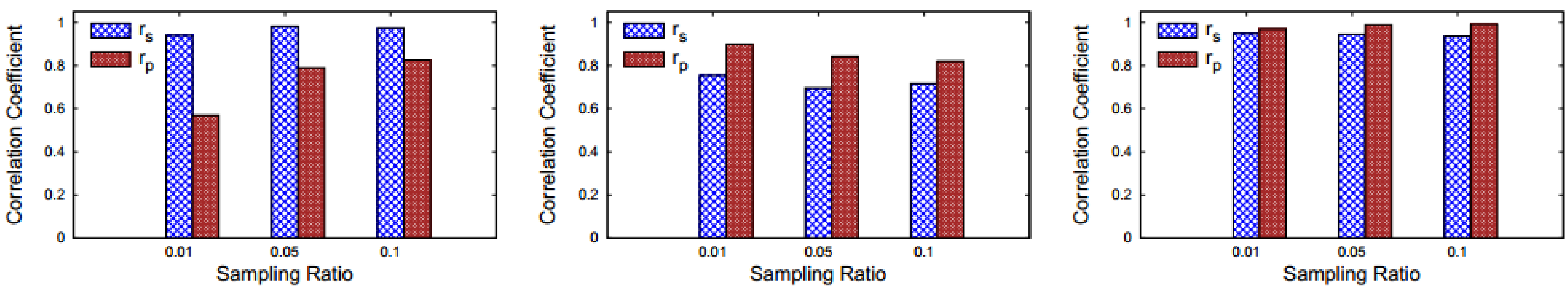
Put It Together (Review)



Experimental Evaluation

- The idea: larger variances => larger estimation errors

We thus measure the *correlation* between the two.



(a) **MICRO**, Uniform 1GB, PC2 (b) **SELJOIN**, Uniform 1GB, PC1 (c) **TPCH**, Skewed 10GB, PC1

We observed *strong* correlations (i.e., correlation coefficient > 0.7) on almost all the queries tested in our experiments.

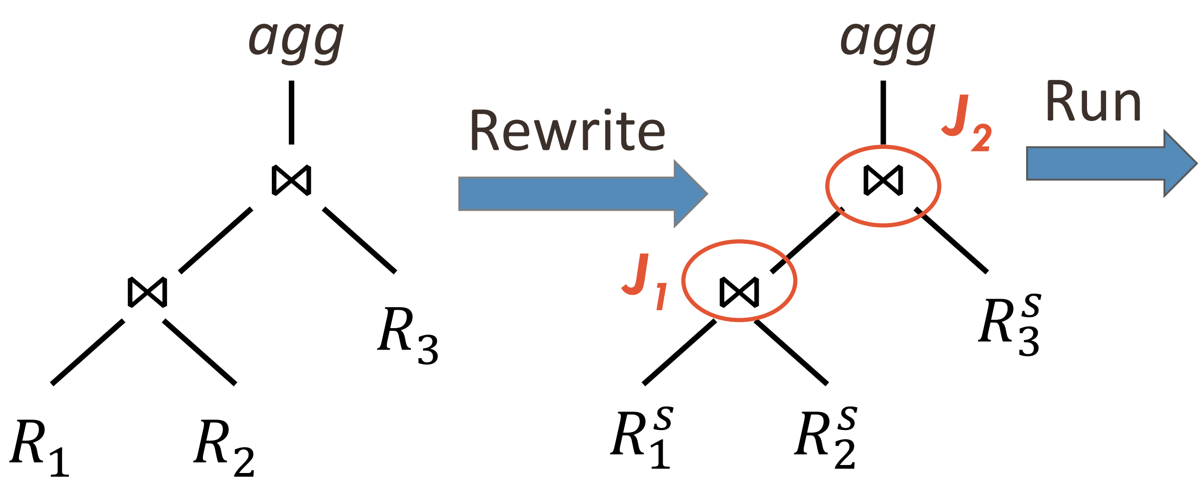
Q & A

- Thank you☺

The Cardinality Refinement Algorithm (Example)

Plan of q:

$$J_1 = R_1 \bowtie R_2$$
$$J_2 = R_1 \bowtie R_2 \bowtie R_3$$



$$\widehat{\rho_{J_1}} = \frac{|R_1^s \bowtie R_2^s|}{|R_1^s| \times |R_2^s|}$$

$$\widehat{\rho_{J_2}} = \frac{|R_1^s \bowtie R_2^s \bowtie R_3^s|}{|R_1^s| \times |R_2^s| \times |R_3^s|}$$

Reuse

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

For *agg*, use PostgreSQL's estimates based on the *refined* input estimates from J_2.

Distributions of Selectivities (Cont.)

- Implementation of $S^2{}_n$ in PostgreSQL

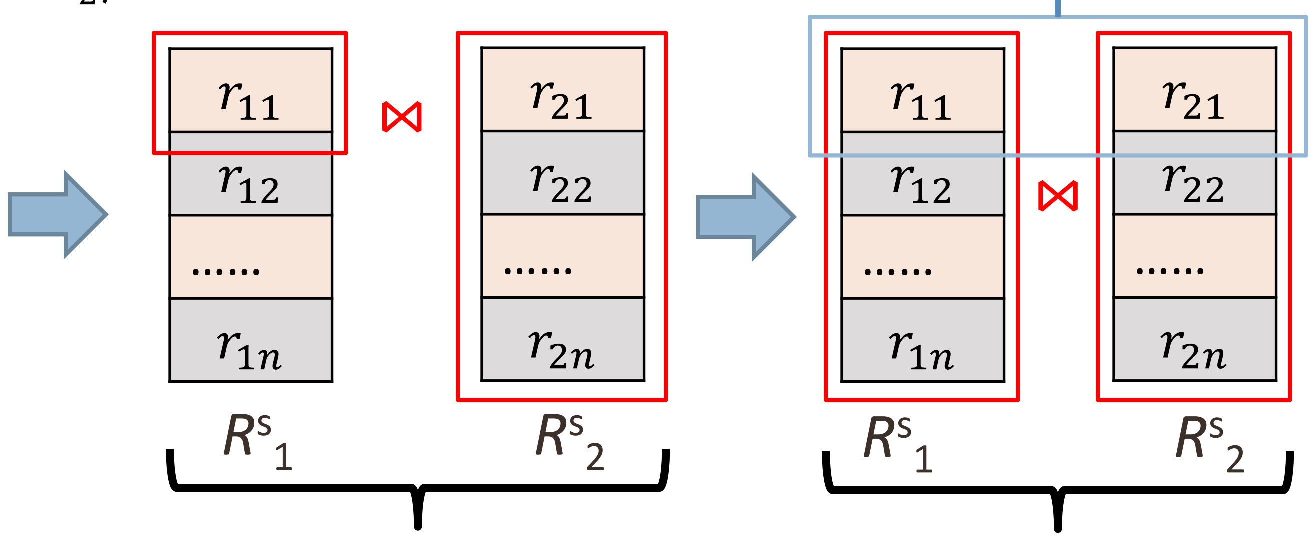
$$S^2{}_n = \sum_{k=1}^{K}\left(\frac{1}{n-1}\sum_{j=1}^{n}\left(Q_{k,j,n} / n^{K-1} - \mu_n\right)^2\right)$$

If: $r_{11} \bowtie r_{21} \in R^s{}_1 \bowtie R^s{}_2$
Then: $++Q_{1,1,n}$
$++Q_{2,1,n}$

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

$$\begin{cases} Q_{1,j,n} = \left|\{r_{1j}\} \bowtie R^s{}_2\right| \\ Q_{2,j,n} = \left|R^s{}_1 \bowtie \{r_{2j}\}\right| \end{cases}$$

r_{1j} and r_{2j} are the j-th row of $R^s{}_1$ and $R^s{}_2$.



2n joins here!

Only 1 join!