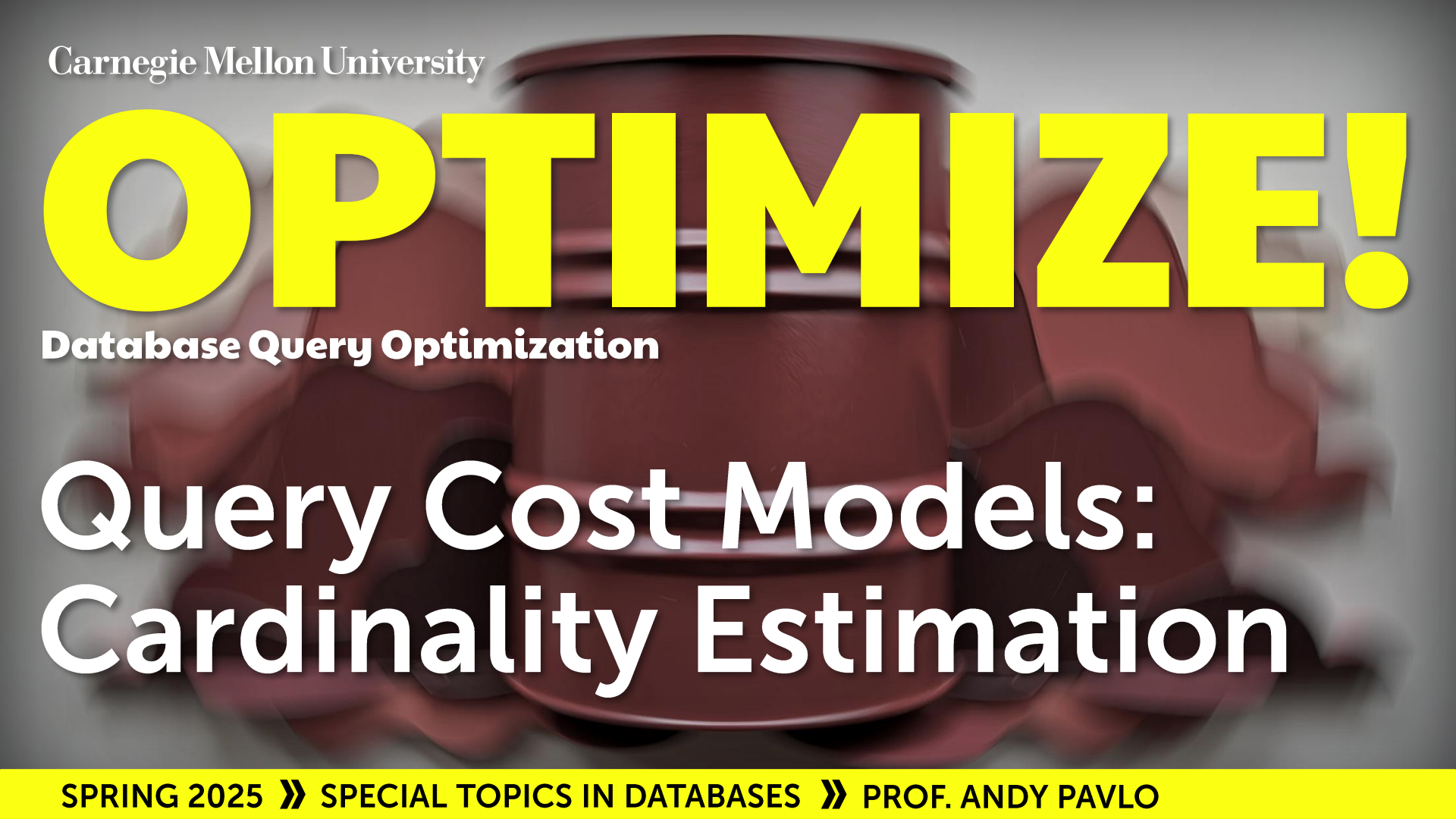

Carnegie Mellon University
OPTIMIZE!
Database Query Optimization
Query Cost Models:
Cardinality Estimation
SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #1 is due Friday Feb 28th.

Project #2 proposals are due Monday Mar 10th.

LAST CLASS

We discussed the data structures that DBMSs maintain to summarize the contents of tables.

→ Most Common: Equi-Depth, End-Bias Histograms (Heavy Hitters), HyperLogLog

Today's class is about how the optimizer's cost model uses these data structures…

CARDINALITY ESTIMATION

Cardinality estimation is the process of predicting the number of rows that will be returned by a query operation, such as a filter or join, to help the optimizer choose the most efficient execution plan.

The number of tuples that will be generated per operator is computed from its selectivity multiplied by the number of tuples in its input.

TODAY'S AGENDA

Cardinality Estimation Basics

The Germans' Survey

Implementations

CARDINALITY ESTIMATION

There are three cardinality estimations on logical expressions an optimizer must support as the core of its cost model:

→ Selection Conditions (filters)
→ Join Size Estimation
→ Distinct Value Estimation

These will serve as building blocks for more complex expressions:

→ Multiple Joins
→ Group By

Source: EQOP Book

DERIVABLE STATISTICS

For each relation ***R***, the DBMS maintains statistics to approximate the following information:

→ N_R: Number of tuples in ***R***.

→ ***V(A,R)***: Number of distinct values for attribute ***A***.

The **selection cardinality** ***SC(A,R)*** is the average number of records with a value for an attribute ***A*** given $N_R / V(A,R)$

SINGLE SELECTION CONDITION

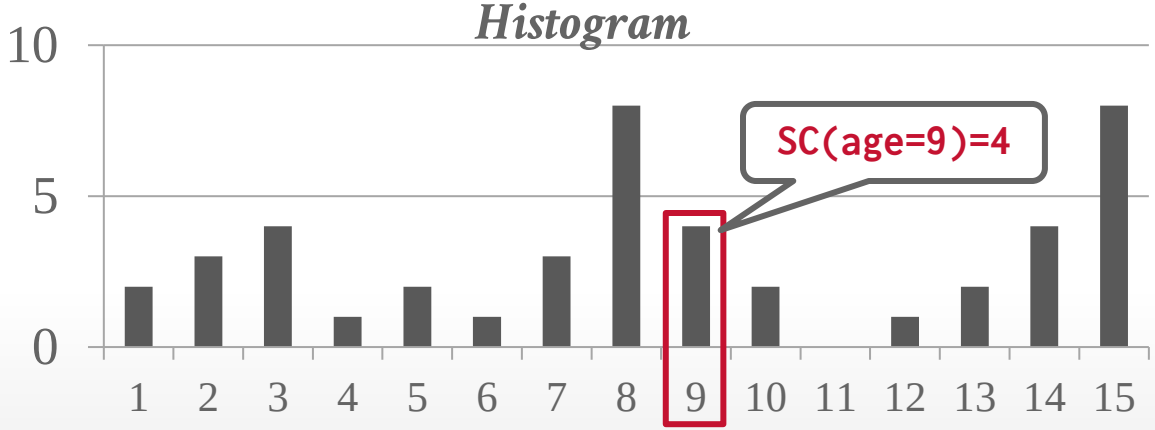
The **selectivity** (***sel***) of a predicate ***P*** is the fraction of tuples that qualify.

```
SELECT * FROM people
 WHERE age = 9
```

Equality Predicate: ***A=constant***

→ ***sel(A=constant) = #occurrences / |R|***

→ Example: ***sel(age=9) = 4 /45 = 0.088***



SINGLE SELECTION CONDITION

The **selectivity** (*sel*) of a predicate *P* is the fraction of tuples that qualify.

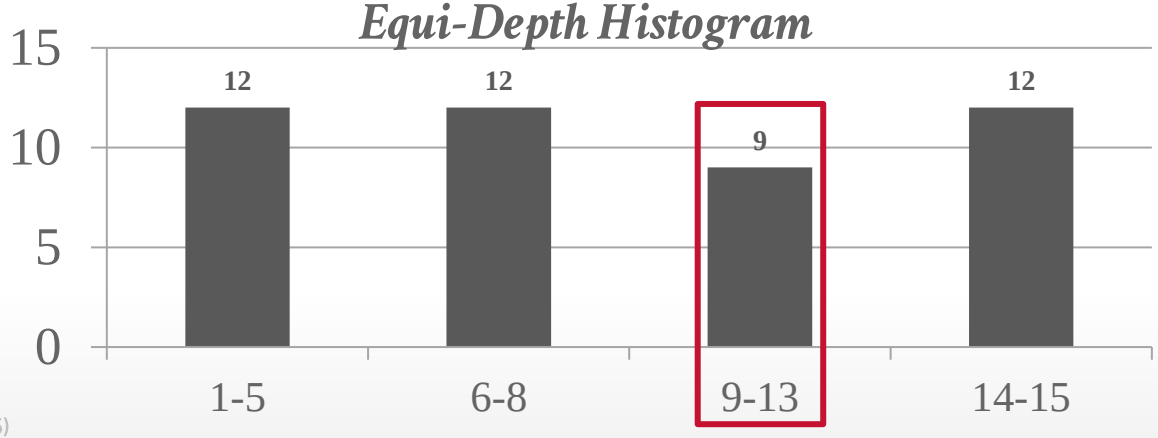
```
SELECT * FROM people
 WHERE age = 9
```

Equality Predicate: *A=constant*

→ *sel(A=constant) = #occurrences / |R|*

→ Example: *sel(age=9) = 4/45 = 0.088*

≈(9/5) /45 ≈ 1.8 /45 ≈0.04



ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

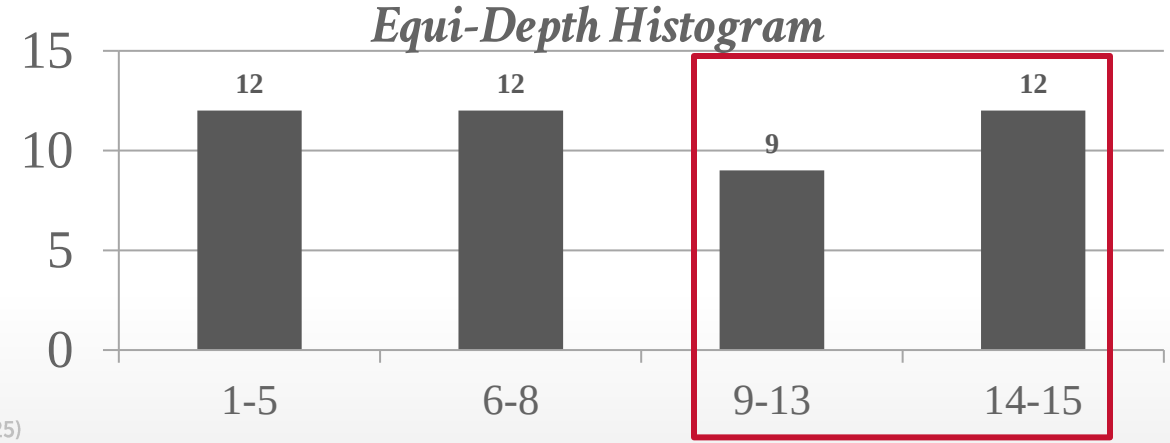
SINGLE SELECTION CONDITION

Range Predicate:

→ *sel(A >= a) = (#RANGE-ROWS + #EQ-ROWS) / |R|*

→ Example: *sel(age >= 7) ≈ ((9+12)*

```
SELECT * FROM people
 WHERE age >= 7
```



SINGLE SELECTION CONDITION

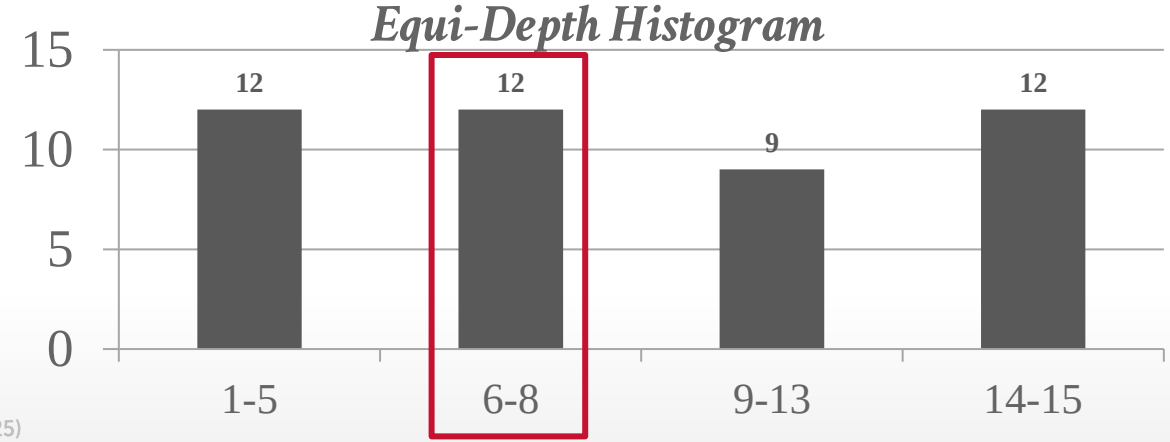
Range Predicate:

→ *sel(A >= a) = (#RANGE-ROWS + #EQ-ROWS) / |R|*

→ Example: *sel(age >= 7) ≈ ((9+12) + (2 × (12/3))) / 45*

≈ 29 / 45 ≈ 0.6444

```
SELECT * FROM people
 WHERE age >= 7
```



SINGLE SELECTION CONDITION

Range Predicate:

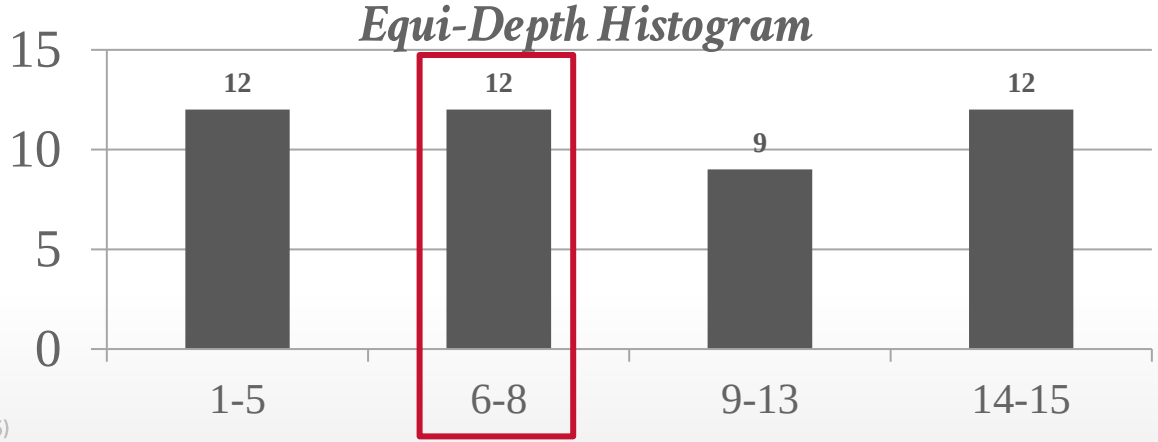
→ *sel(A >= a) = (#RANGE-ROWS + #EQ-ROWS) / |R|*

→ Example: *sel(age >= 7) ≈ ((9+12) + (2 × (12/3))) / 45*

≈ 29 / 45 ≈ 0.6444

```
SELECT * FROM people
 WHERE age >= 7
```

⚠ ***This assumes continuous distribution of values.***



SINGLE SELECTION CONDITION

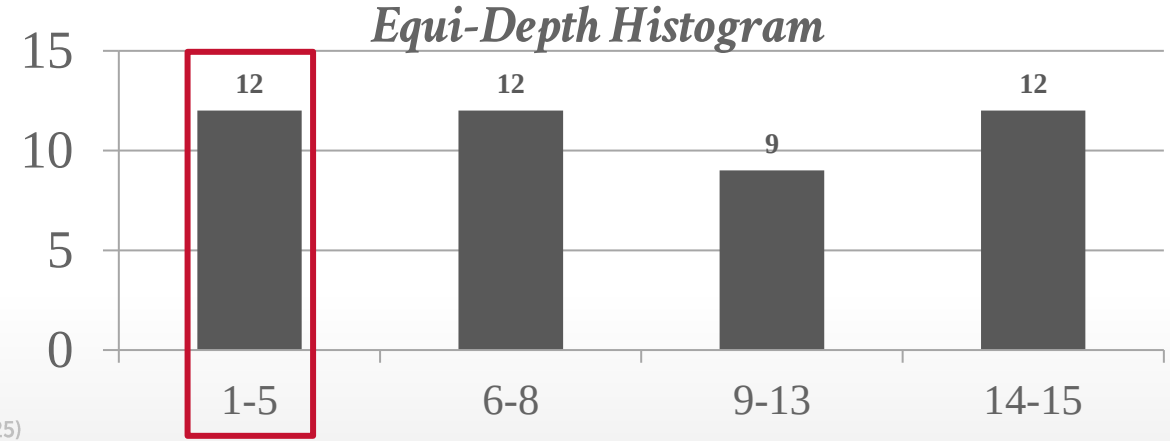
Negation Query:

→ *sel(not P) = 1 – sel(P)*

→ Example: *sel(age != 2) ≈ 1 - ((12/5) /45)*
≈ 1 - (2.4 /45) ≈ 1 - 0.05 ≈0.95

```
SELECT * FROM people
 WHERE age != 2
```

⚠ ***Observation: Selectivity ≈ Probability***



OBSERVATION

We now can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.

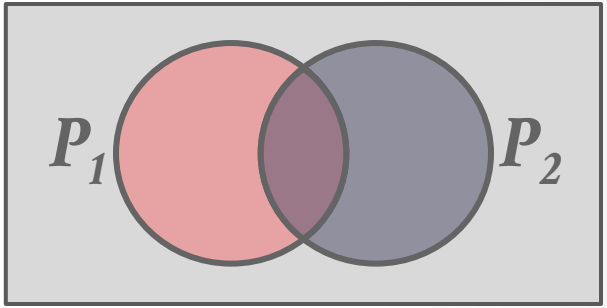
```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

OBSERVATION

We now can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```



Example:

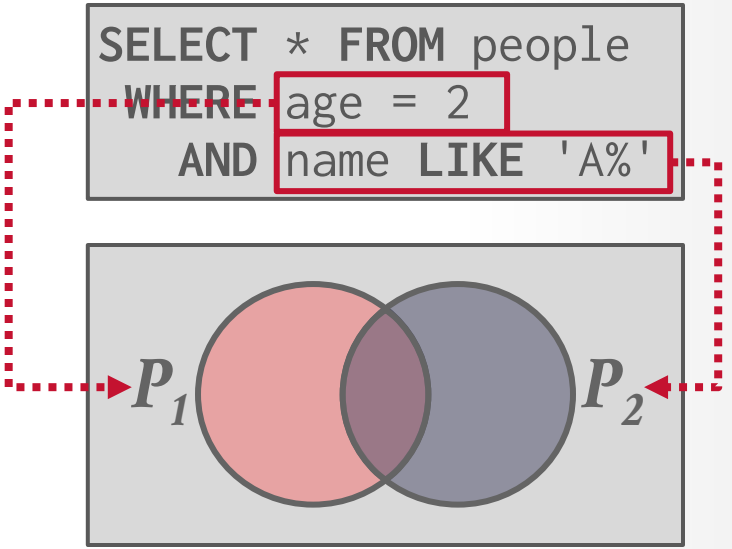
→ *sel(age = 2) ≈0.053*

→ *sel(name LIKE 'A%') ≈0.1*

OBSERVATION

We now can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.



Example:

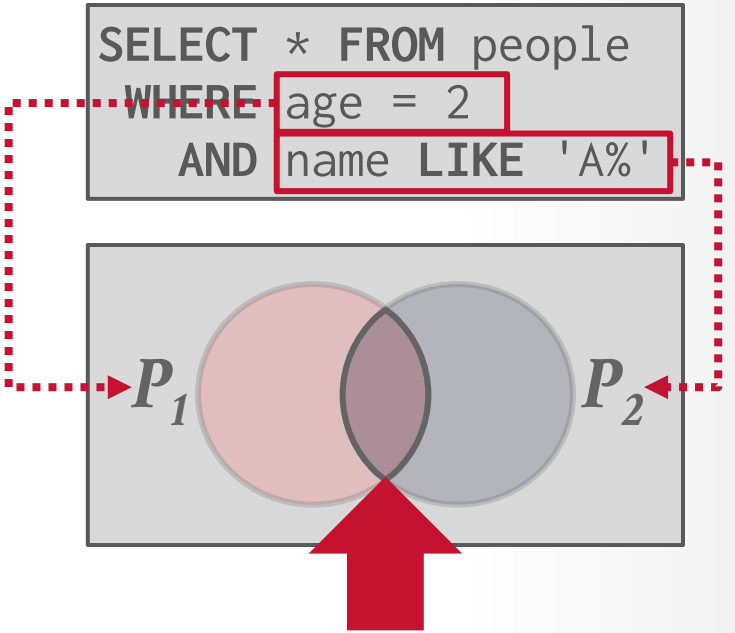
→ *sel(age = 2) ≈ 0.053*

→ *sel(name LIKE 'A%') ≈ 0.1*

OBSERVATION

We now can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.



Example:

→ *sel(age = 2) ≈ 0.053*

→ *sel(name LIKE 'A%') ≈ 0.1*

ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

Assumption #2: Independent Predicates

→ The predicates on attributes are independent. The selectivity of the conjunction of two or more predicates is estimated as the product of their individual selectivities.

MULTIPLE SELECTION CONDITION

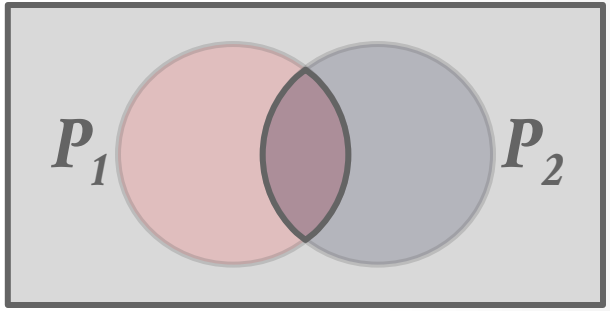
Conjunction:

→ $sel(P1 \wedge P2) = sel(P1) \times sel(P2)$

→ Example: $sel(age=2 \wedge name\ LIKE\ 'A\%') \approx sel(age=2) \times sel(name\ LIKE\ 'A\%') \approx 0.053 \times 0.1 \approx 0.0053$

This assumes that the predicates are **independent**.

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```



MULTIPLE SELECTION CONDITION

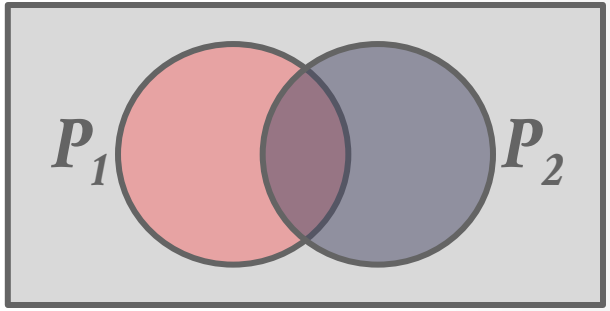
Disjunction:

→ $sel(P1 \vee P2)$

$\approx sel(P1) + sel(P2) - sel(P1 \wedge P2)$

$\approx sel(P1) + sel(P2) - sel(P1) \times sel(P2)$

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



MULTIPLE SELECTION CONDITION

Disjunction:

→ $sel(P1 \vee P2)$

$\approx sel(P1) + sel(P2) - sel(P1 \wedge P2)$

$\approx sel(P1) + sel(P2) - sel(P1) \times sel(P2)$

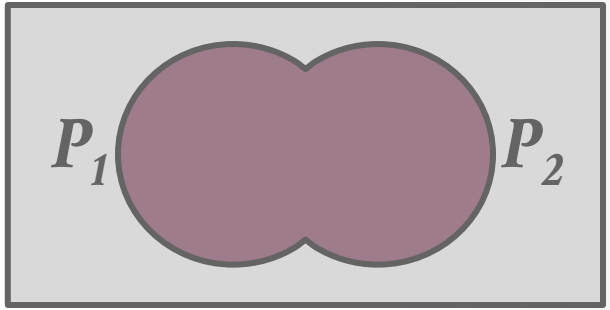
→ Example: $sel(age=2 \vee name\ LIKE\ 'A\%')$

$\approx 0.053 + 0.1 - (0.053 \times 0.1)$

≈ 0.1477

This again assumes that the selectivities are **independent**.

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



CORRELATED ATTRIBUTES

Consider a database of automobiles:
→ # of Makes = 10, # of Models = 100

Then the following query shows up:
→ `(make="Honda" AND model="Accord")`

With the independence and uniformity assumptions, the selectivity is:
→ 1
/10 × 1/100 ≈ 0.001

But since only Honda makes Accords the real selectivity is ***1/100 = 0.01***

Source: Guy Lohman

JOIN SIZE ESTIMATION

Given a join of ***R*** and ***S***, what is the range of possible result sizes in # of tuples?

→ In other words, for a given tuple of ***R***, how many tuples of ***S*** will it match?

Assume each key in the inner relation will exist in the outer table.

ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

Assumption #2: Independent Predicates

→ The predicates on attributes are independent. The selectivity of the conjunction of two or more predicates is estimated as the product of their individual selectivities.

Assumption #3: Containment Principle

→ The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

JOIN SIZE ESTIMATION

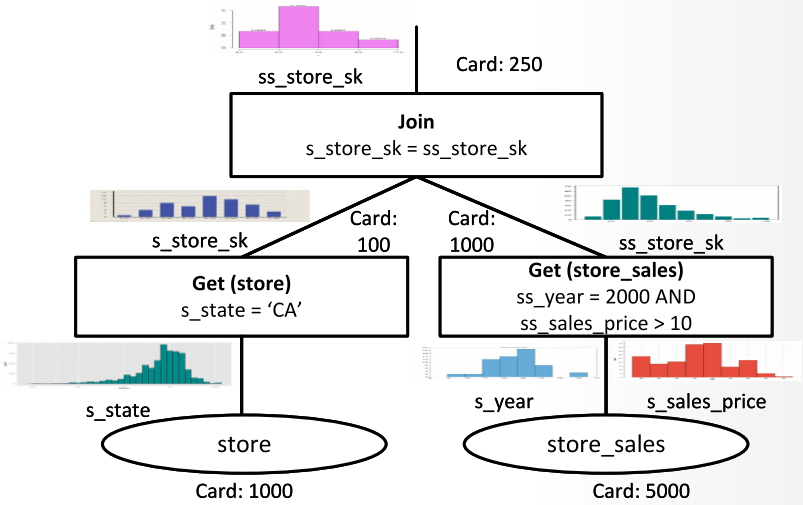
Align the buckets of the histograms so that their boundaries agree.

Compute a per bucket estimation of join size.

→ Containment assumes that for each group g_R of tuples in R, it has a corresponding group g_S in S.

→ Each tuple in g_R will match with tuples in g_S

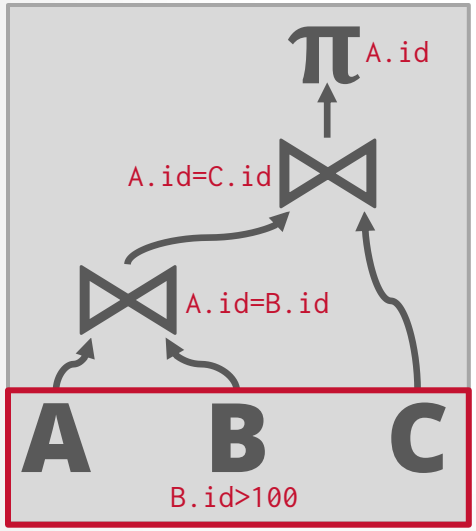
Aggregate partial frequencies from joining each pair of buckets to get cardinality of the whole join.



Source: EQOP Book

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

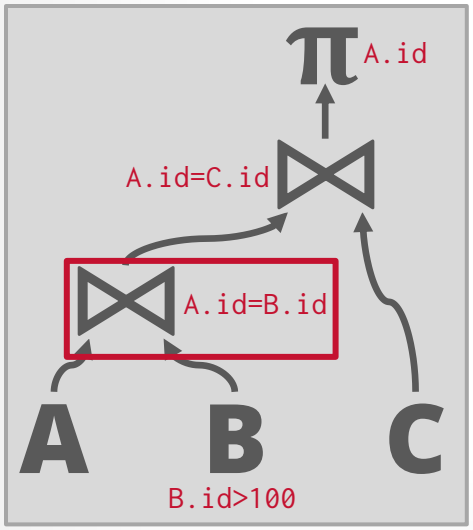
$$A \rightarrow |A|$$

$$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$$

$$C \rightarrow |C|$$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$A \rightarrow |A|$

$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$

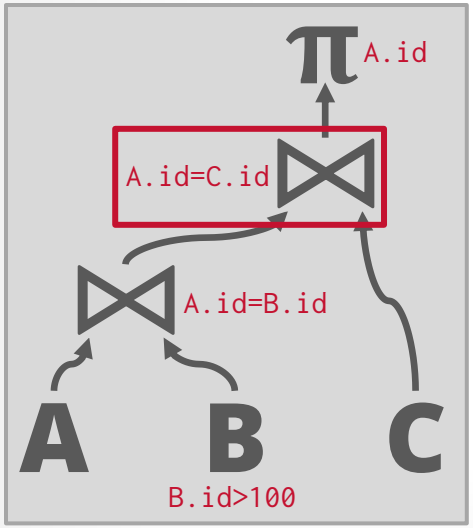
$C \rightarrow |C|$

Compute the cardinality of join results

$$A \bowtie B \approx (|A| \times |B|) / max(sel(A.id=B.id), sel(B.id>100))$$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$$A \rightarrow |A|$$

$$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$$

$$C \rightarrow |C|$$

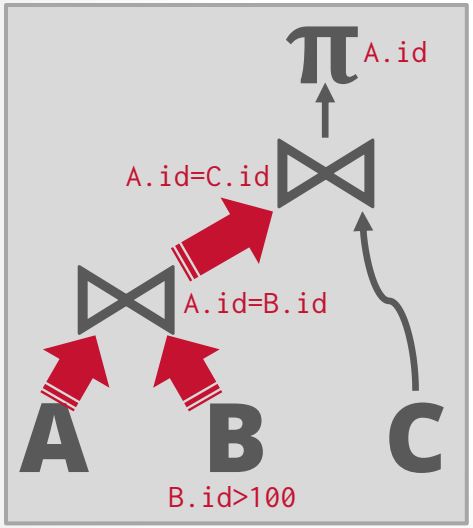
Compute the cardinality of join results

$$A \bowtie B \approx (|A| \times |B|) / max(sel(A.id=B.id), sel(B.id>100))$$

$$(A \bowtie B) \bowtie C \approx (|A| \times |B| \times |C|) / max(sel(A.id=B.id), sel(B.id>100), sel(A.id=C.id))$$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$A \rightarrow |A|$

$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$

$C \rightarrow |C|$

Compute the cardinality of join results

$$A \bowtie B \approx (|A| \times |B|) / max(sel(A.id=B.id), sel(B.id>100))$$

$$(A \bowtie B) \bowtie C \approx (|A| \times |B| \times |C|) / max(sel(A.id=B.id), sel(B.id>100), sel(A.id=C.id))$$

ESTIMATOR QUALITY

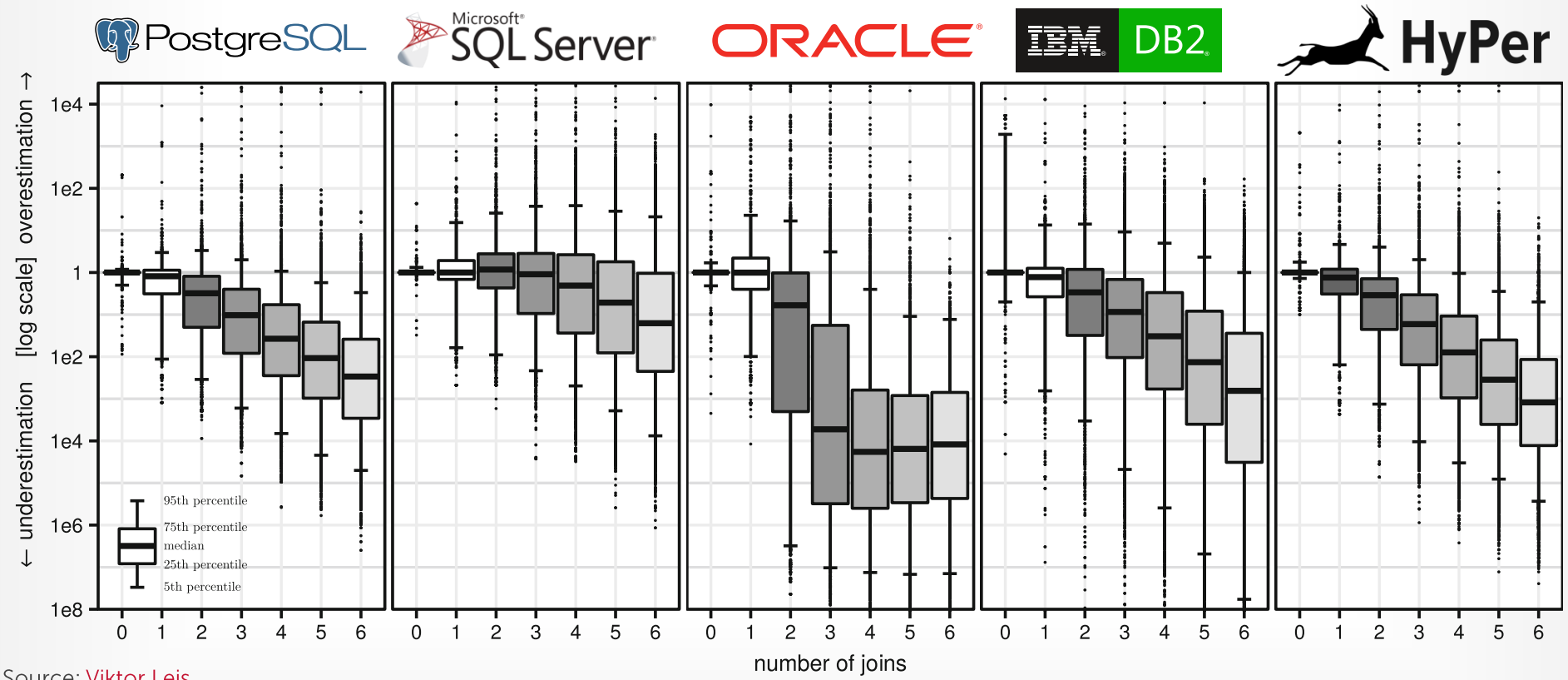
Evaluate the correctness of cardinality estimates generated by DBMS optimizers as the number of joins increases.

→ Let each DBMS perform its stats collection.
→ Extract measurements from query plan.

Compared five DBMSs using 100k queries from the JOB workload based on IMDB data.

HOW GOOD ARE QUERY OPTIMIZERS, REALLY?
VLDB 2015

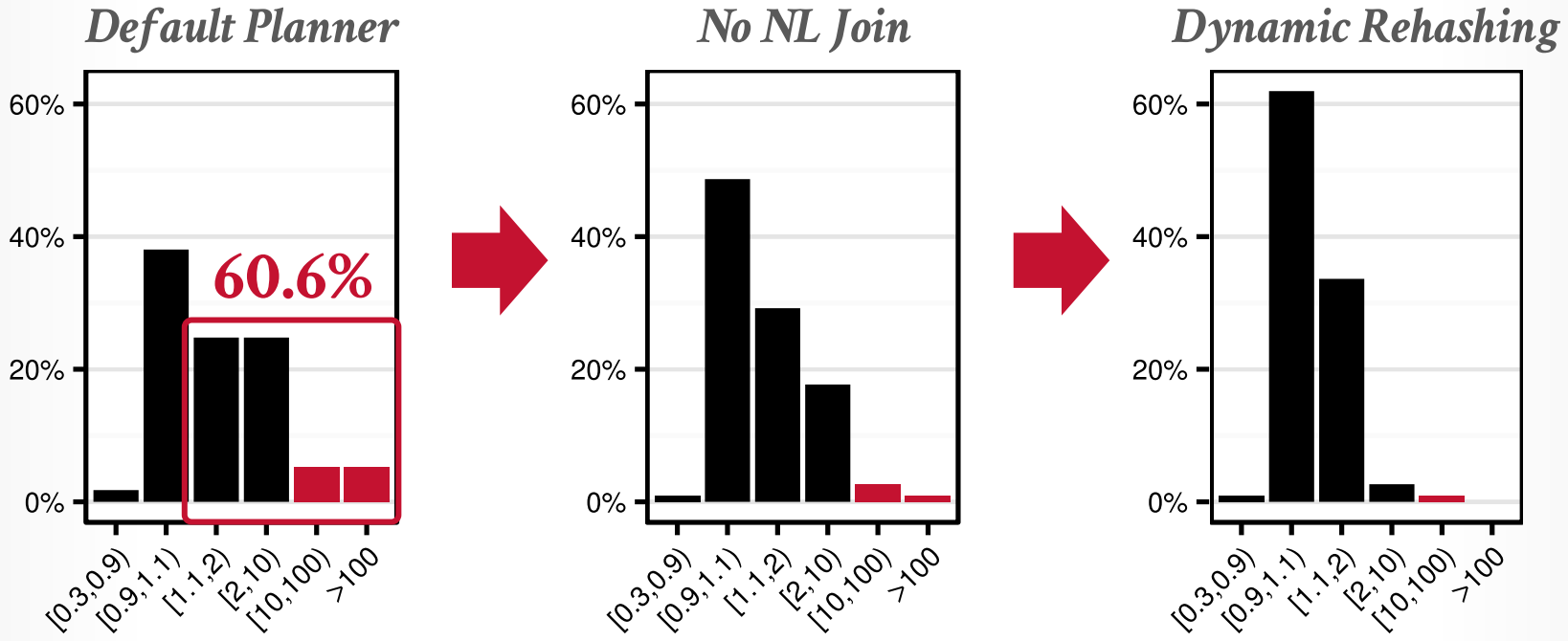
ESTIMATOR QUALITY



Source: Viktor Leis

EXECUTION SLOWDOWN

PostgreSQL v9.4 – JOB Workload



Slowdown compared to using true cardinalities

Source: Viktor Leis

LESSONS FROM THE GERMANS

Query opt is more important than a fast engine
→ Cost-based join ordering is necessary

Cardinality estimates are routinely wrong
→ Try to use operators that do not rely on estimates

Hash joins + seq scans are a robust exec model
→ The more indexes that are available, the more brittle the plans become (but also faster on average)

Working on accurate models is a waste of time
→ Better to improve cardinality estimation instead

Source: Viktor Leis

IMPLEMENTATIONS

Postgres

Microsoft SQL Server

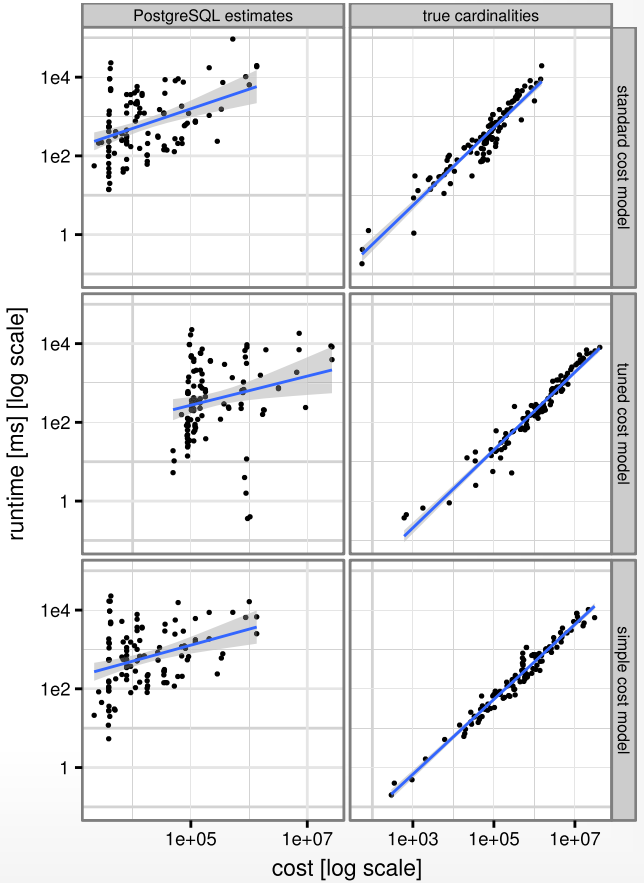
POSTGRES COST MODEL

Cost of an operator is a weighted sum of the # of accessed disk pages and the amount of data processed in memory.

→ Distinguishes between sequential and random I/O.

→ Requires manual tuning the weights based on hardware characteristics.

The Germans replaced Postgres' cost model with simple cost model to see what happens with query plans…



Source: Viktor Leis

MODEL

master postgres / src / backend / optimizer / path / costsize.c

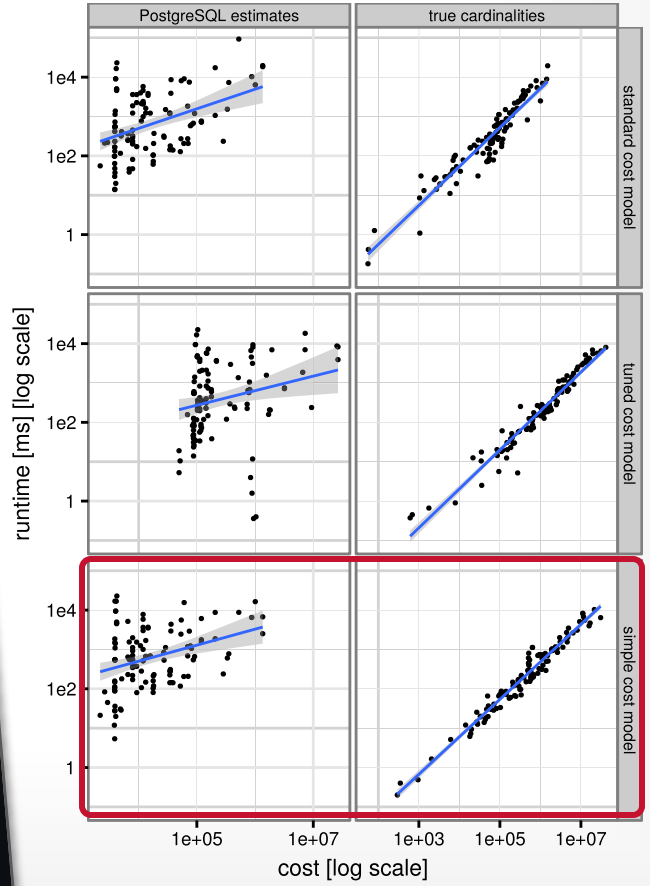
Go to file

6605 lines (5909 loc) · 214 KB

Raw

Code Blame

```
/*-------------------------------------------------------------------------
 *
 * costsize.c
 *    Routines to compute (and set) relation sizes and path costs
 *
 * Path costs are measured in arbitrary units established by these basic
 * parameters:
 *
 *  seq_page_cost       Cost of a sequential page fetch
 *  random_page_cost    Cost of a non-sequential page fetch
 *  cpu_tuple_cost      Cost of typical CPU time to process a tuple
 *  cpu_index_tuple_cost  Cost of typical CPU time to process an index tuple
 *  cpu_operator_cost   Cost of CPU time to execute an operator or function
 *  parallel_tuple_cost Cost of CPU time to pass a tuple from worker to leader backend
 *  parallel_setup_cost Cost of setting up shared memory for parallelism
 *
 * We expect that the kernel will typically do some amount of read-ahead
 * optimization; this in conjunction with seek costs means that seq_page_cost
 * is normally considerably less than random_page_cost.  (However, if the
 * database is fully cached in RAM, it is reasonable to set them equal.)
 *
 * We also use a rough estimate "effective_cache_size" of the number of
 * disk pages in Postgres + OS-level disk cache.  (We can't simply use
 * NBuffers for this purpose because that would ignore the effects of
 * the kernel's disk cache.)
 *
 * Obviously, taking constants for these values is an oversimplification,
 * but it's tough enough to get any useful estimates even at this level of
 * detail.  Note that all of these parameters are user-settable, in case
 * the default values are drastically off for a particular platform.
 *
 * seq_page_cost and random_page_cost can also be overridden for an individual
 * tablespace, in case some data is on a fast disk and other data is on a slow
 * disk.  Per-tablespace overrides never apply to temporary work files such as
 * an external sort or a materialize node that overflows work_mem.
 *
 * We compute two separate costs for each path:
```



MODEL

master postgres / src / backend / optimizer / path / costsize.c

Go to file

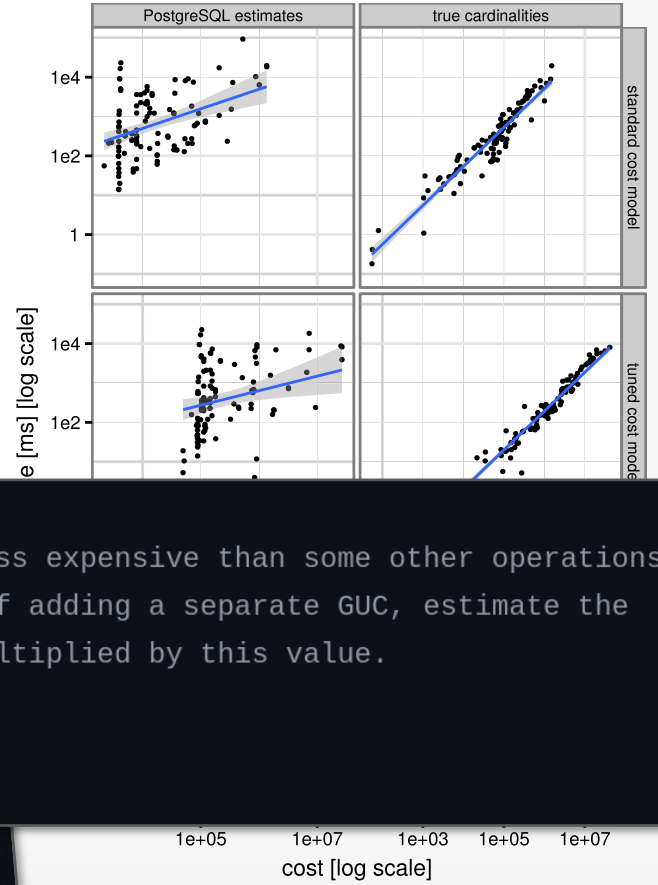
6605 lines (5909 loc) · 214 KB

Raw

Code Blame

```
/*-------------------------------------------------------------------------
 *
 * costsize.c
 *    Routines to compute (and set) relation sizes and path costs
 *
 * Path costs are measured in arbitrary units established by these basic
 * parameters:
 *
 *  seq_page_cost       Cost of a sequential page fetch
 *  random_page_cost    Cost of a non-sequential page fetch
 *  cpu_tuple_cost      Cost of typical CPU time to process a tuple
 *  cpu_index_tuple_cost  Cost of typical CPU time to process an index tuple
 *  cpu_operator_cost   Cost of CPU time to execute an operator or function
 *  parallel_tuple_cost Cost of CPU time to pass a tuple from worker to leader backend
 *  parallel_setup_cost Cost of setting up shared memory for parallelism
 *
 * We expect that the kernel will typically do some amount of read-ahead
 * optimization; this in conjunction with seek costs means that seq_page_cost
 * is normally consi
 * database is fully
 *
 * We also use a ro
 * disk pages in Po
 * NBuffers for thi
 * the kernel's di
 *
 * Obviously, taki
 * but it's tough
 * detail.  Note
 * the default va
 *
 * seq_page_cost
 * tablespace, i
 * disk.  Per-tab
 * an external sort or a materialize node that
 *
 * We compute two separate costs for each path:
```

```
115     /*
116      * Append and MergeAppend nodes are less expensive than some other operations
117      * which use cpu_tuple_cost; instead of adding a separate GUC, estimate the
118      * per-tuple cost as cpu_tuple_cost multiplied by this value.
119      */
120     #define APPEND_CPU_COST_MULTIPLIER 0.5
121
```



Sourc

SQL SERVER

Each operator's cost is based on producing all its output and the cost producing the first output.

→ Example: Nested-Loop Join vs. Index Nested-Loop Join

Maintains "row goals" to track the number of tuples are needed above in the query plan.

→ Example: Top-K / LIMIT clause

Combine predicates using an exponential backoff. Given a table R and predicate selectivities S_1, S_2, S_3, …S_n, where S_1 is the most selective and S_n the least:

$$Estimate = |R| \times S_1 \times \sqrt{(S_2)} \times \sqrt{(\sqrt{(S_3)})} \times \sqrt{(\sqrt{(\sqrt{(S_4)})}} \ldots$$

Source: EQOP Book

PARTING THOUGHTS

Cardinality estimation is the hardest part of building a query optimizer.

→ Lots of simplifying assumptions to make the problem more tractable.

→ But developers often don't revisit those assumptions…

Research suggests that accurate cardinality estimation is more important than sophisticated cost models.

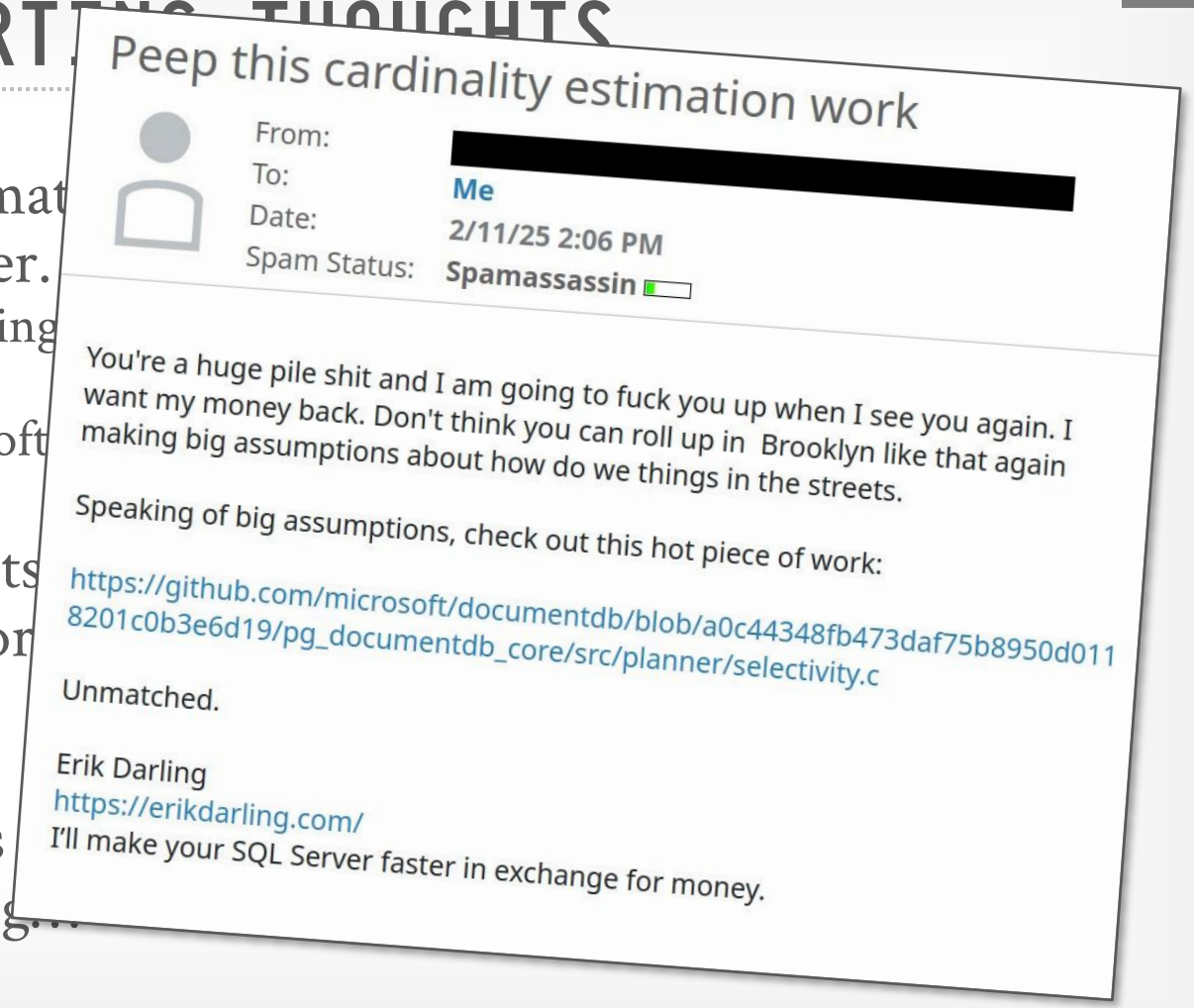
All of this seems like it is the perfect usecase for machine learning…

PART[illegible] THOUGHTS

Cardinality estimat[illegible]
a query optimizer.
→ Lots of simplifying[illegible]
tractable.
→ But developers oft[illegible]

Research suggests[illegible]
estimation is mor[illegible]
cost models.

All of this seems[illegible]
machine learning…



main documentdb / pg_documentdb_core / src / planner / selectivity.c

Code Blame Raw

```c
/*-------------------------------------------------------------------------
 * Copyright (c) Microsoft Corporation.  All rights reserved.
 *
 * src/planner/selectivity.c
 *
 * Implementation of selectivity functions for BSON operators.
 *
 *-------------------------------------------------------------------------
 */
#include <postgres.h>
#include <fmgr.h>


PG_FUNCTION_INFO_V1(bson_operator_selectivity);


/*
 * bson_operator_selectivity returns the selectivity of a BSON operator
 * on a relation.
 */
Datum
bson_operator_selectivity(PG_FUNCTION_ARGS)
{
        /* dumbest possible implementation: assume 1% of rows are returned */
        PG_RETURN_FLOAT8(0.01);
}
```

TS

y estimation work

[redacted]

25 2:06 PM

assassin

ping to fuck you up when I see you again. I
you can roll up in Brooklyn like that again
w do we things in the streets.

k out this hot piece of work:

mentdb/blob/a0c44348fb473daf75b8950d011
ore/src/planner/selectivity.c

change for money.

NEXT CLASS

Project #2 Proposals