Helping the Planner Help You - Extended Statistics in PostgreSQL

Alexandra Wang
PostgreSQL Developer at EnterpriseDB
SCaLE 23x - 03/05/2026

Agenda

- Why Cardinality Estimation Matters
- When Estimates Work - and When they Don’t
- A Practical Approach: Join Statistics
- Other Approaches
- What PostgreSQL Should Do Next

Why Cardinality Estimation Matters

The Achilles' Heel of Cost-Based Optimization

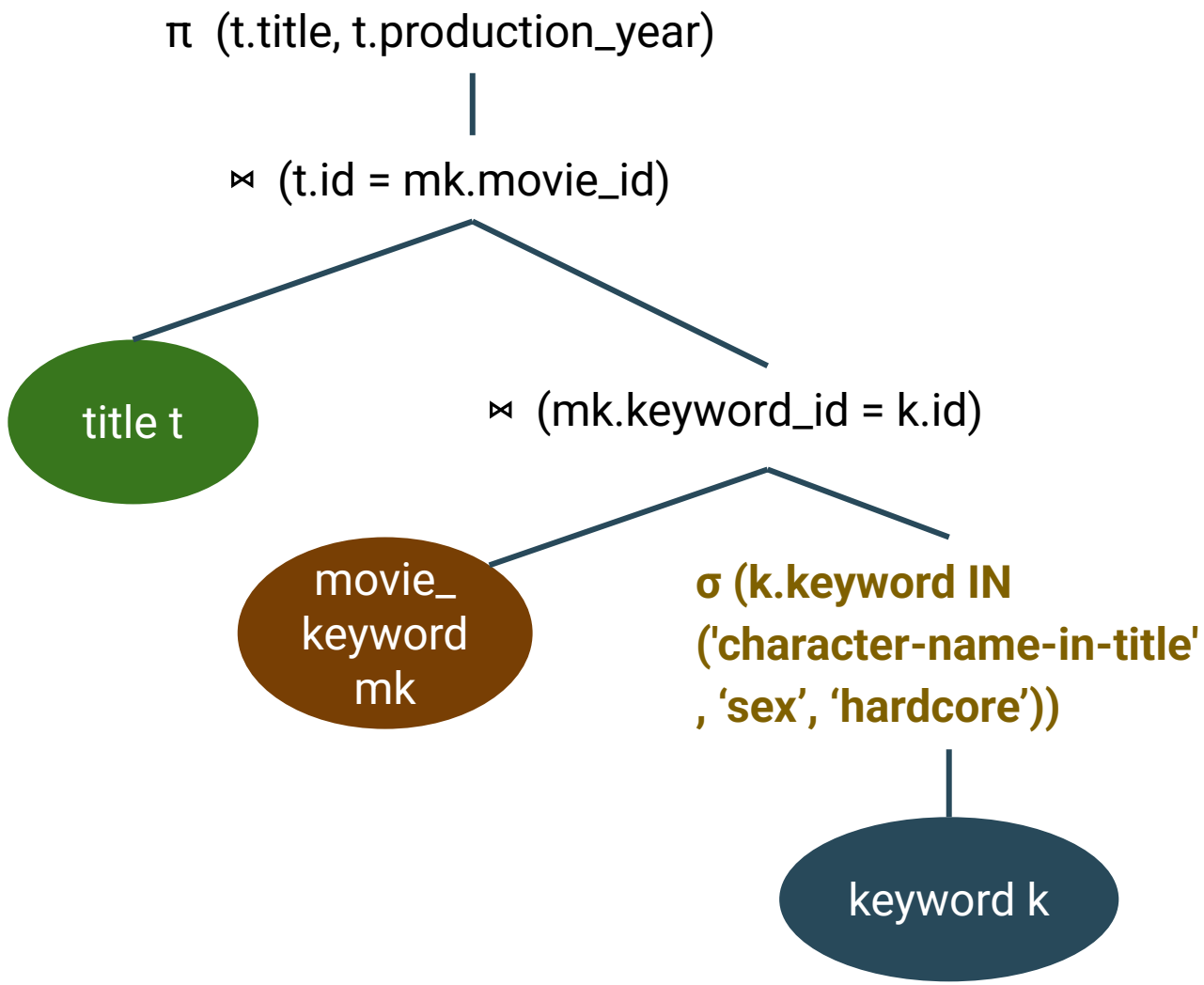
“The root of all evil,
the Achilles Heel of query optimization,
is the estimation of the size of intermediate results,
known as cardinalities.”

“...the cost model may introduce errors of at most 30% for a given cardinality,
but the cardinality model can quite easily introduce errors of many **orders of magnitude**!”

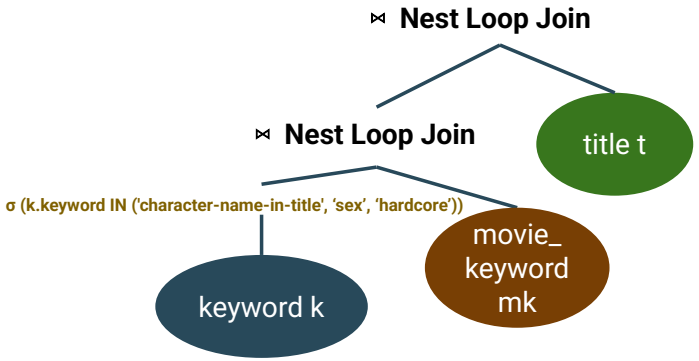
– Guy Lohman, IBM Research
"Is Query Optimization a 'Solved' Problem?" (2014)

The Achilles' Heel of Cost-Based Optimization

```
EXPLAIN (ANALYZE)
SELECT t.title, t.production_year
FROM keyword k
JOIN movie_keyword mk ON mk.keyword_id = k.id
JOIN title t ON t.id = mk.movie_id
WHERE k.keyword IN ('character-name-in-title',
                    'sex',
                    'hardcore');
```



EXPLAIN ANALYZE - Default Setting



Nested Loop (rows=101) (actual … rows=172784.00 loops=1)

-> Nested Loop (rows=101) (actual …rows=172784.00 loops=1)

-> Seq Scan on keyword k (rows=3) (actual … rows=3)

Filter: (keyword = ANY ('{character-name-in-title,sex,hardcore}'::text[]))

-> Bitmap Heap Scan on movie_keyword mk (**rows=307**) (**actual … rows=57594.67 loops=3**)

-> Bitmap Index Scan on keyword_id_movie_keyword

Index Cond: (keyword_id = k.id)

-> Index Scan using title_pkey on title t (rows=1) (actual … rows=1.00 loops=172784)

Index Cond: (id = mk.movie_id)

Planning Time: 11.314 ms

Execution Time: 1340.905 ms

**Omitted irrelevant details*

A ⋈ B - Nest Loop Join vs. Hash Join

Nested Loop Join:

for each row in A

  lookup matches in B

A → lookup in B

A → lookup in B

A → lookup in B

best when:

- outer side is small
- inner lookup is cheap (index)

Hash Join:

- build hash table on B
- probe for each row in A

build hash(B)

A → probe

A → probe

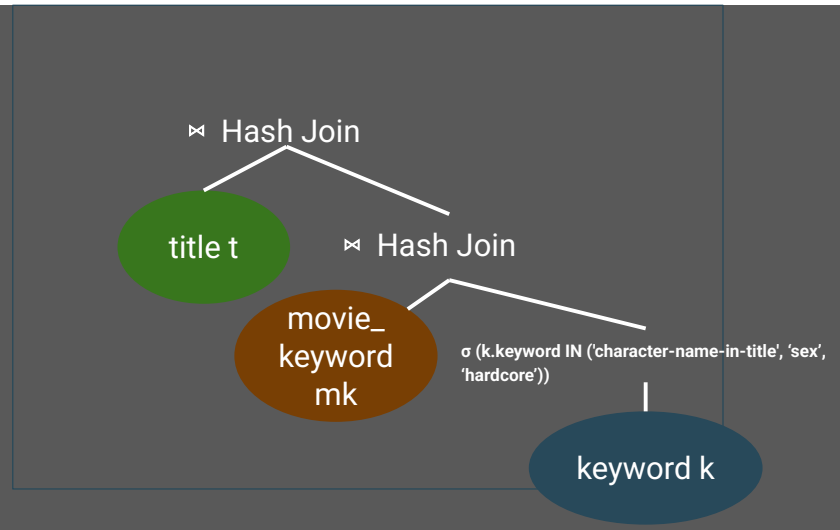
A → probe

best when:

- both sides are large
- parallel scan is possible

EXPLAIN ANALYZE - Set enable_nestloop = off

```
Hash Join  (rows=101) (actual rows=172784.00)
  Hash Cond: (t.id = mk.movie_id)
  ->  Seq Scan on title t
  ->  Hash  (rows=101) (actual rows=172784.00)
        ->  Hash Join  (rows=101) (actual rows=172784.00)
              Hash Cond: (mk.keyword_id = k.id)
              ->  Seq Scan on movie_keyword mk
              ->  Hash
                    ->  Seq Scan on keyword k  (rows=3) (actual rows=3.00 loops=1)
                          Filter: (keyword = ANY ('{character-name-in-title,sex,hardcore}'::text[]))
Planning Time: 0.435 ms
Execution Time: 581.273 ms
```



**Omitted irrelevant details*
**Also SET max_parallel_workers_per_gather = 0; to make plan more readable and easier to compare*

The Achilles' Heel of Cost-Based Optimization

	Planner's Estimate	Reality
Rows	307	172784
Plan Chosen	2 Nest Loop Joins	2 Hash Joins
Executions Time	**1,341 ms**	**581 ms** **(252 ms when parallel)**

When Estimates Work - and When They Don’t

When Estimates Work - Single Table, Single Column

A pg_stats entry

schemaname	public
tablename	t
attname	col
null_frac	0
n_distinct	14832
most_common_vals	{335,16264,117,2488,...}
most_common_freqs	{0.015,0.013,0.009,0.008,...}
histogram_bounds	**{4,72,141,..., 994,** 1050,...,134170}
correlation	0.024
most_common_elems	NULL
(+ 5 more fields for array/range types)	...

<u>Single-column selectivity is often very accurate.</u>

Example query: ***WHERE col < 1000***

MCV contribution:
sum frequencies of MCV values below 1000 = 0.122

Histogram contribution:
histogram_bounds = {4, 72, 141, ..., 994, 1050, ...}
|←—— 13 full buckets ——→| ↑
Sum of their frequencies = 0.106

Add them up
MCV: 0.122 + Histogram: 0.106 = 0.228
Actual: 0.226 (0.6% off)

The Independence Assumption - Multiple Columns

P(A AND B) = P(A) × P(B) PostgreSQL assumes independence until told otherwise.

*SELECT * FROM customers WHERE city = 'Seattle' AND state = 'WA';*

P(city = 'Seattle') = 0.056 ← from pg_stats MCV
P(state = 'WA') = 0.05 ← from pg_stats MCV
P(Seattle AND WA) = 0.056 × 0.05 = 0.0028

Estimated rows: 0.0028 × 200,000 = 560
Actual rows: 5,000 ← 9x off

But almost every Seattle row IS in Washington — they are functionally dependent.

When Estimates Work - Single Table, Multiple Columns

Extended Statistics For Correlated Columns Within A Table:

CREATE STATISTICS *s_customers* ON *city, state, zip* FROM *customers*;
ANALYZE *customers*;

-- Now: estimated 4,950. Actual: 5,000. → within 1%.

Extended Statistics - Functional Dependencies

Functional dependencies capture correlations between columns in the same table.

f = 0: independence

f = 1: full dependence

Extended Statistics - MCV List

The planner stores the real frequency of (crypto, Japan)

Query with multi-column WHERE

→ MCV match? → use joint frequency

→ Dependencies → interpolate

→ Fallback → independence (multiply)

Extended Statistics - NDistinct

Benefits GROUP BY

50 states × 5,000 cities = 250,000 estimated groups. Actual: 5,000.

When Estimates Don’t Work

Currently, PostgreSQL’s extended statistics solve the **within-table** problem.

But what happens when the correlation **crosses a join boundary**?

```
SELECT *
FROM A JOIN B
 ON (B.id = A.foreign_key)
WHERE B.filter_col = 'popular-filter-value';
```

- Filter on dimension table (B)
- Join key correlation in fact table (A)
- Stats stored per relation
- Cross-table correlation invisible

A Practical Fix: Join Statistics

Join Statistics

Goal:

- Capture **join correlations** between tables
- Start with Join **MCV**, extend later to **NDistinct / dependencies**
- **Low overhead** during ANALYZE
- Simple, **extensible design**

Not Goal:

- Automatically detect candidate columns
- Automatically create the join statistics

Join Statistics - Proposed Syntax for MCV Stats

```
CREATE STATISTICS [ [ IF NOT EXISTS ] statistics_name ]
    [ ( mcv ) ]
    ON { table_name1.column_name1 [, { table_name1.column_name2 } [, ...]]
    FROM table_name1 JOIN table_name2 ON table_name1.column_name3 = table_name2.column_name4
```

Example:

```
-- Create cross-table MCV statistics on a single filter column (keyword)
CREATE STATISTICS movie_keywords2_keyword_stats (mcv)
ON k.keyword
FROM movie_keywords2 mk JOIN keywords2 k ON (mk.keyword_id = k.id);
ANALYZE movie_keywords2;
-- Create cross-table MCV statistics on multiple filter columns (keyword + phonetic_code)
CREATE STATISTICS movie_keywords2_multi_stats (mcv)
ON k.keyword, k.phonetic_code
FROM movie_keywords2 mk JOIN keywords2 k ON (mk.keyword_id = k.id);
ANALYZE movie_keywords2;
```

Collect Join Statistics - Index-base Sampling

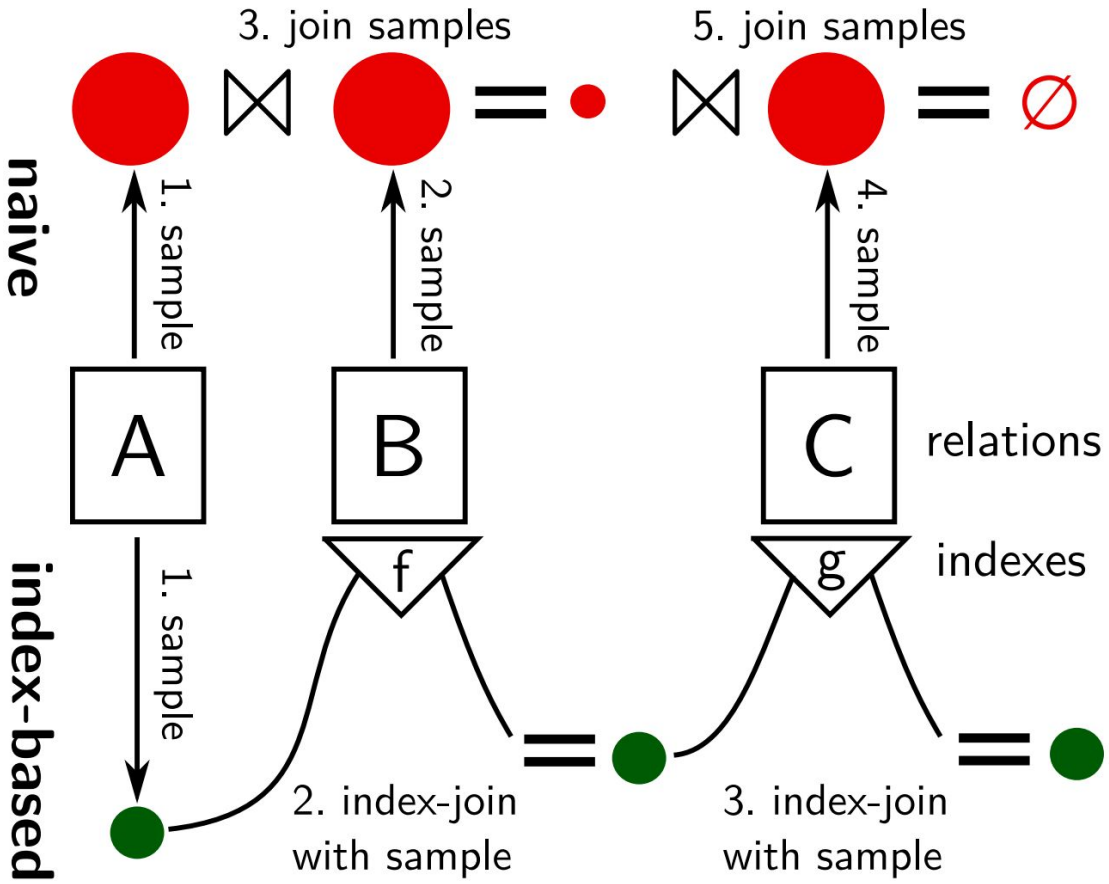


Figure 1: Naive (top) vs. index-based (bottom) join sampling

Cardinality Estimation Done Right: Index-Based Join Sampling

Viktor Leis, Bernhard Radke, Andrey Gubichev†; Alfons Kemper, Thomas Neumann
Technische Universität München
{leis,radke,kemper,neumann}@in.tum.de
Google†
gubichev@google.com†

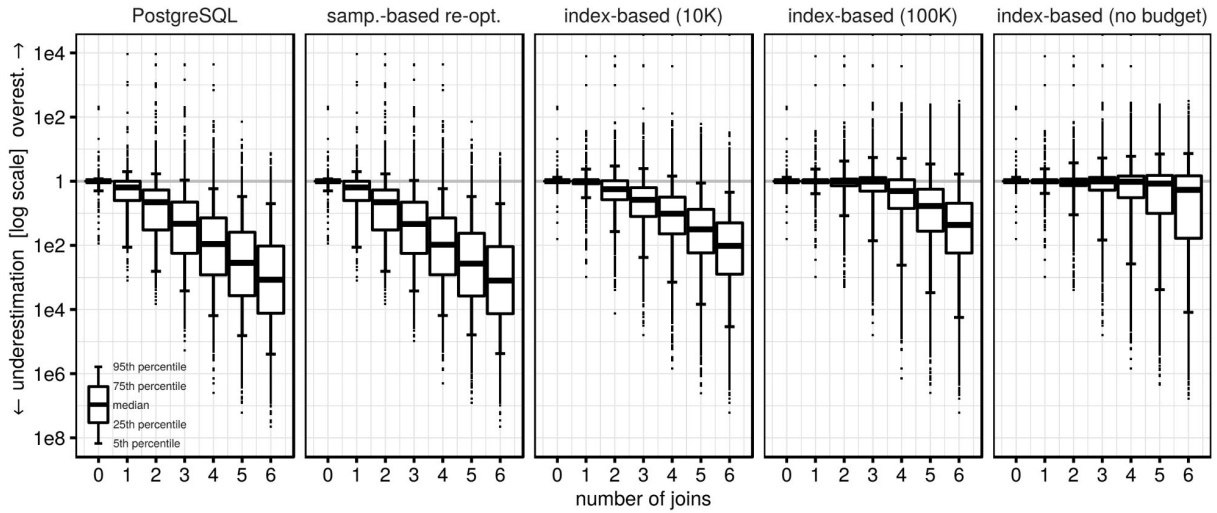
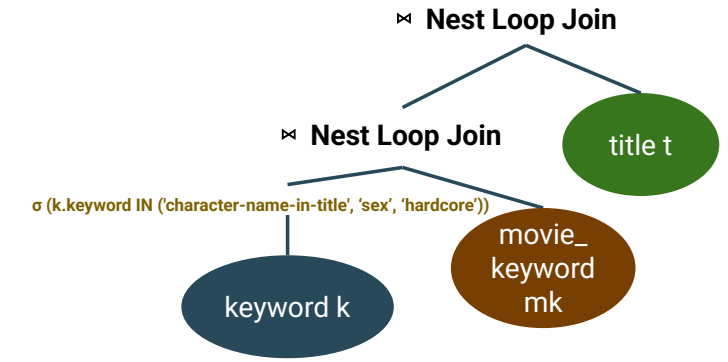


Figure 4: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

My PoC Patch for Join Statistics

Re: Is there value in having optimizer stats for joins/foreignkeys?

Minimal Query - Before



Nested Loop (... rows=101...) (actual ... rows=172784.00 loops=1)

-> Nested Loop (... rows=101...) (actual ...rows=172784.00 loops=1)

-> Seq Scan on keyword k

Filter: (keyword = ANY ('{character-name-in-title,sex,hardcore}'::text[]))

-> Bitmap Heap Scan on movie_keyword mk (... **rows=307**...) (**actual ... rows=57594.67 loops=3**)

-> Bitmap Index Scan on keyword_id_movie_keyword

Index Cond: (keyword_id = k.id)

-> Index Scan using title_pkey on title t (... rows=1..) (actual ... rows=1.00 loops=172784)

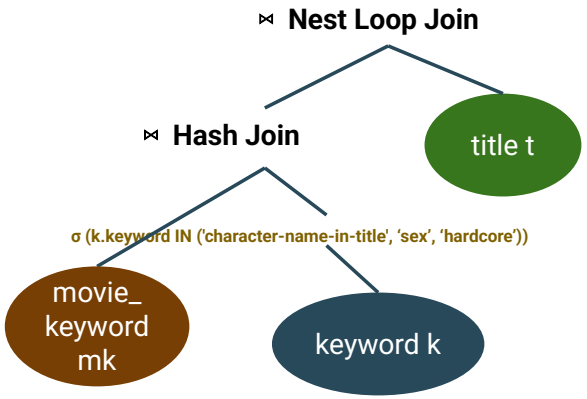
Index Cond: (id = mk.movie_id)

Planning Time: 11.314 ms

Execution Time: 1340.905 ms

**Omitted irrelevant details*

Minimal Query - After



CREATE STATISTICS movie_keyword_keyword_stats (mcv) ON k.keyword

FROM movie_keyword mk JOIN keyword k ON (mk.keyword_id = k.id);

Gather (cost... rows=177187 ...) (actual ... rows=172784.00 loops=1)

Workers Launched: 2

-> Nested Loop (rows=73828 ...) (actual ... rows=57594.67 loops=3)

-> Hash Join (rows=73828 ...) (actual ... rows=57594.67 loops=3)

Hash Cond: (mk.keyword_id = k.id)

-> Parallel Seq Scan on movie_keyword mk (rows=1884970) (actual rows=1507976.33 loops=3)

-> Hash (rows=3) (actual rows=3.00 loops=3)

-> Seq Scan on keyword k

Filter: (keyword = ANY ('{character-name-in-title,sex,hardcore}'::text[]))

-> Index Scan using title_pkey on title t

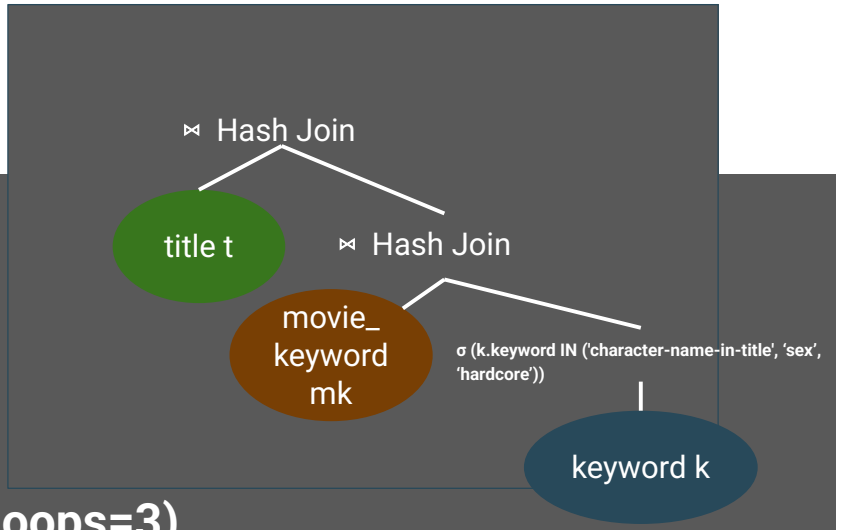
Index Cond: (id = mk.movie_id)

Planning Time: 15.559 ms

Execution Time: 251.653 ms

**Omitted irrelevant details*

Minimal Query - Set enable_nestloop = off




```
Gather
  Workers Launched: 2
  -> Parallel Hash Join  (rows=73828) (actual rows=57594.67 loops=3)
       Hash Cond: (t.id = mk.movie_id)
       -> Parallel Seq Scan on title t  (rows=1056548) (actual rows=842770.33 loops=3)
       -> Parallel Hash  (rows=73828) (actual rows=57594.67 loops=3)
            -> Hash Join  (rows=73828) (actual rows=57594.67 loops=3)
                 Hash Cond: (mk.keyword_id = k.id)
                 -> Parallel Seq Scan on movie_keyword mk  (rows=1884970) (actual rows=1507976.33 loops=3)
                 -> Hash  (cost=2852.82..2852.82 rows=3 width=4) (actual time=20.040..20.040 rows=3.00 loops=3)
                      -> Seq Scan on keyword k  (rows=3) (actual rows=3.00 loops=3)
                           Filter: (keyword = ANY ('{character-name-in-title,sex,hardcore}'::text[]))
Planning Time: 1.687 ms
Execution Time: 285.266 ms
```

**Omitted irrelevant details*

A 2015 VLDB paper: https://www.vldb.org/pvldb/vol9/p204-leis.pdf

The Join Order Benchmark (JOB)

- Uses the IMDB data set
- Queries have between 3 and 16 joins, with an average of 8 joins per query

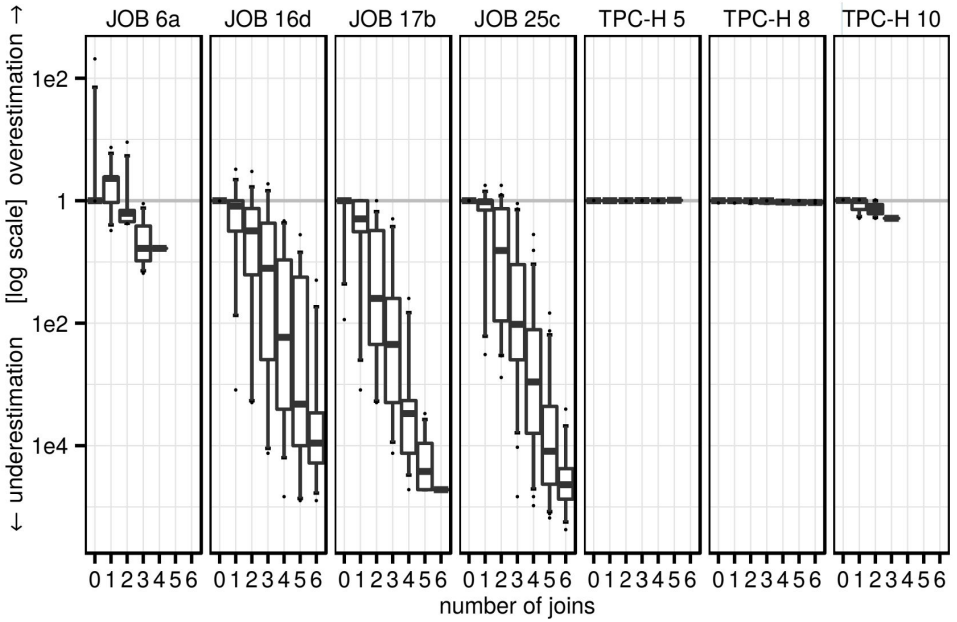


Figure 4: PostgreSQL cardinality estimates for 4 JOB queries and 3 TPC-H queries

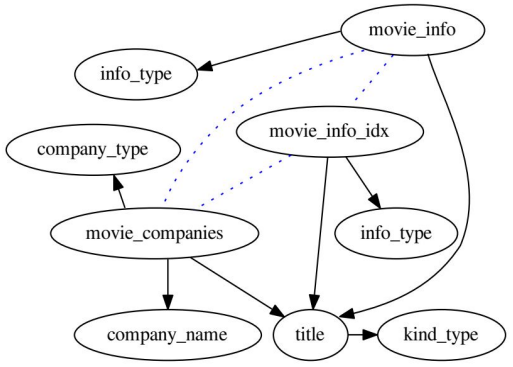


Figure 2: Typical query graph of our workload

How Good Are Query Optimizers, Really?

Viktor Leis
TUM
leis@in.tum.de

Andrey Gubichev
TUM
gubichev@in.tum.de

Atanas Mirchev
TUM
mirchev@in.tum.de

Peter Boncz
CWI
p.boncz@cwi.nl

Alfons Kemper
TUM
kemper@in.tum.de

Thomas Neumann
TUM
neumann@in.tum.de

ABSTRACT

Finding a good join order is crucial for query performance. In this paper, we introduce the Join Order Benchmark (JOB) and experimentally revisit the main components in the classic query optimizer architecture using a complex, real-world data set and realistic multi-join queries. We investigate the quality of industrial-strength cardinality estimators and find that all estimators routinely produce large errors. We further show that while estimates are essential for finding a good join order, query performance is unsatisfactory if the query engine relies too heavily on these estimates. Using another set of experiments that measure the impact of the cost model, we find that it has much less influence on query performance than rdinality estimates. Finally, we investigate plan enumera- chniques comparing exhaustive dynamic programming with tic algorithms and find that exhaustive enumeration improves mance despite the sub-optimal cardinality estimates.

INTRODUCTION

problem of finding a good join order is one of the most studoblems in the database field. Figure 1 illustrates the classical, ased approach, which dates back to System R [36]. To obtain cient query plan, the query optimizer enumerates some subset valid join orders, for example using dynamic programming. cardinality estimates as its principal input, the cost model hooses the cheapest alternative from semantically equivalent lternatives.

oretically, as long as the cardinality estimations and the cost are accurate, this architecture obtains the optimal query plan. lity, cardinality estimates are usually computed based on simg assumptions like uniformity and independence. In real- data sets, these assumptions are *frequently* wrong, which ead to sub-optimal and sometimes disastrous plans.

his *experiments and analyses* paper we investigate the three components of the classical query optimization architecture er to answer the following questions:

- How good are cardinality estimators and when do bad estimates lead to slow queries?

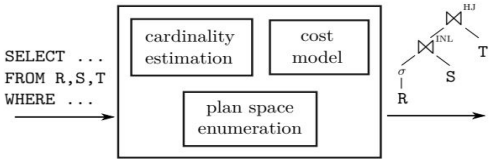


Figure 1: Traditional query optimizer architecture

- How important is an accurate cost model for the overall query optimization process?
- How large does the enumerated plan space need to be?

To answer these questions, we use a novel methodology that allows us to isolate the influence of the individual optimizer components on query performance. Our experiments are conducted using a real-world data set and 113 multi-join queries that provide a challenging, diverse, and realistic workload. Another novel aspect of this paper is that it focuses on the increasingly common main-memory scenario, where all data fits into RAM.

The main contributions of this paper are listed in the following:

- We design a challenging workload named *Join Order Benchmark* (JOB), which is based on the IMDB data set. The benchmark is publicly available to facilitate further research.
- To the best of our knowledge, this paper presents the first end-to-end study of the join ordering problem using a real-world data set and realistic queries.
- By quantifying the contributions of cardinality estimation, the cost model, and the plan enumeration algorithm on query performance, we provide guidelines for the complete design of a query optimizer. We also show that many disastrous plans can easily be avoided.

The rest of this paper is organized as follows: We first discuss important background and our new benchmark in Section 2. Section 3 shows that the cardinality estimators of the major relational database systems produce bad estimates for many realistic queries, in particular for multi-join queries. The conditions under which these bad estimates cause slow performance are analyzed in Section 4. We show that it very much depends on how much the query engine relies on these estimates and on how complex the physical database design is, i.e., the number of indexes available. Query engines that mainly rely on hash joins and full table scans,

Proceedings of the VLDB Endowment, Vol. 9, No. 3

POSTGRES.AI

The Join Order Benchmark (JOB) - PostgreSQL in 2026

Setup:

- On my Macbook: ./configure 'CFLAGS=-g -O3'
- Cold runs: restarted database & cleared system cache
- Warm runs: repeat run of the entire test suite
- Average of 3 runs per test
- Add a single join statistics object:

```
CREATE STATISTICS movie_keyword_keyword_stats (mcv)
ON k.keyword
FROM movie_keyword mk
JOIN keyword k ON (mk.keyword_id = k.id);
```

JOB Results - Total Execution Time

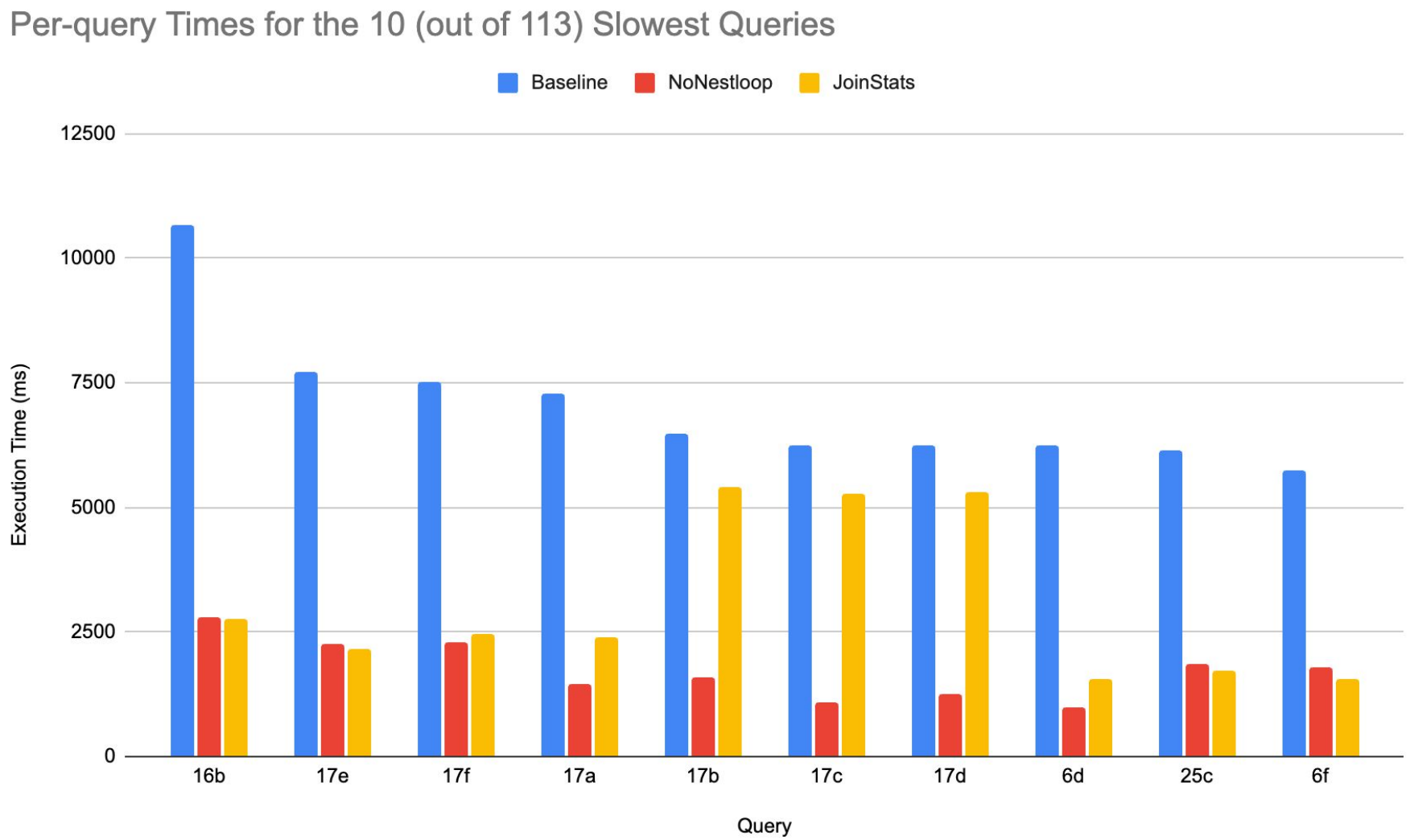
Configuration	Cold Runs (s)	Warm Runs (s)
Baseline (pg19devel)	140.75 ± 2.1	117.72 ± 0.8
No Nestloop	87.16 ± 0.2	86.68 ± 0.4
Join Stats	**85.81 ± 3.4**	**65.24 ± 0.5**

A Single Join Statistic Object:

Cold run: **39% faster than baseline**

Warm run: **45% faster than baseline**

JOB Results - Per-query Execution Times



Other Approaches

Learned & Feedback-Based Optimization

Learn Cardinality from Data

Train model -> predict row counts

Examples:

- MSCN (learn: query -> cardinality)
- DeepDB (learn: data distribution)
- NeuroCard (learn: full joint model generatively)
- ByteCard (in production)

Learn from Past Executions

Store execution stats -> improve future plans

Examples:

- SQL Server CE feedback
- PostgreSQL AQO extension
- Presto history-based optimization (HBO)

PostgreSQL challenges:
No pluggable CE interface in planner; Training/feedback infrastructure required

(Leis et al. CIDR 2019; Hilprecht et al. PVLDB 2020; Yang et al. PVLDB 2021, Han et al., SIGMOD 2024)

(Microsoft; PostgresPro; Meta)

Runtime Adaptation

Correct mistakes **during execution**

Nested Loop → Hash Join

(after seeing 40k rows instead of 34)

Production Systems::

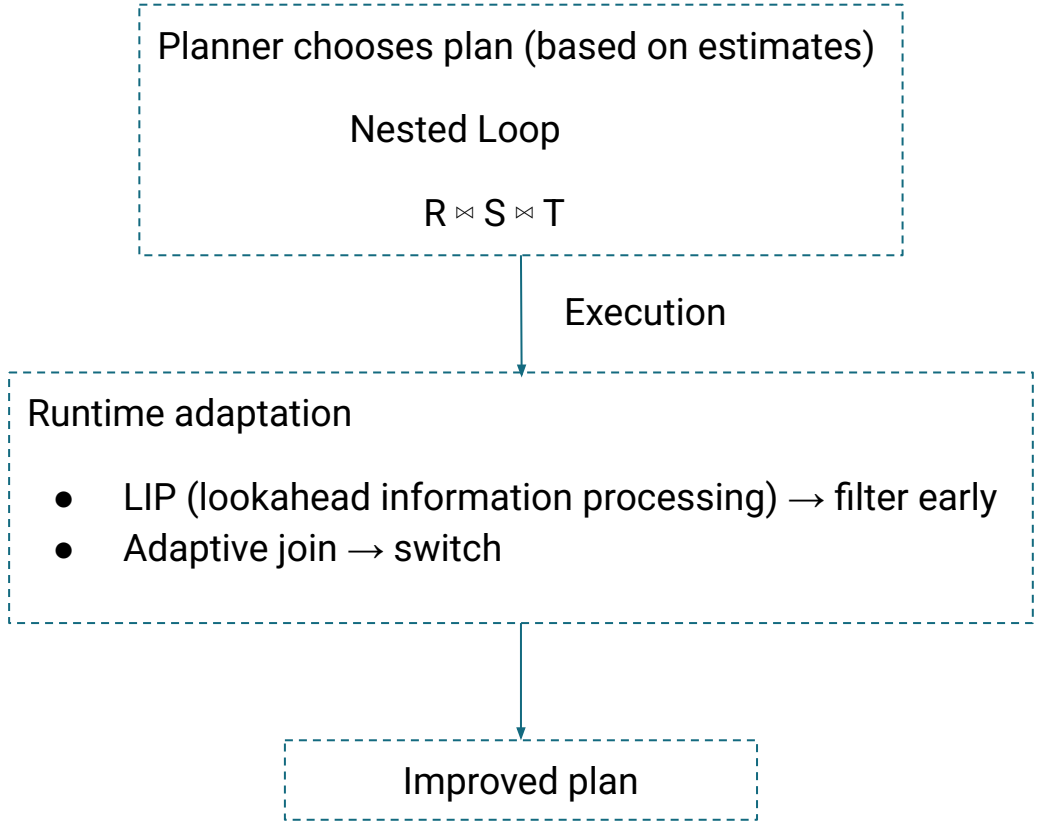
- Oracle adaptive joins
- Spark AQE (Adaptive Query Execution)

PostgreSQL: simple adaptive techniques:

- Early filtering (Bloom filters)
- Adaptive Join switching

≈ or > learned optimizers on JOB workloads

(Zhang et al., VLDB 2023)



Pessimistic Bounds

Underestimate -> nested loop over millions of rows ⟶ catastrophic

Upper bounds -> planner choses a safer plan

Research:

- Safebound
- LpBound

Tested in PostgreSQL:

- 80% lower end-to-end runtimes on JOB
- Nearly as good as using the true cardinalities

Zhang et al., SIGMOD 2025

What PostgreSQL Should Do Next?

The Hardware Timeline

Many optimizer assumptions were designed when **random I/O was expensive**

- ANALYZE sample size (~30k rows, a 1998 paper)
- Cost model: Changing the default random_page_cost value (from 2000)

Era	Medium	Random Read Latency
1990s–2000s	HDD	~10 ms
2010s	SATA SSD	~170 µs
Today	NVMe SSD	~20 µs

~500x faster random reads

Exactly cardinalities are currently used for optimizer testing. (*Chaudhuri et al., VLDB 2009*)

"how do we estimate better?"

OR

"what do we still need to estimate at all?"

Possible Directions for PostgreSQL

- **Better estimation**
  - Join statistics
  - Learned cardinality estimation
- **Adapt during execution**
  - Runtime adaptation
- **Avoid worst plans**
  - Pessimistic bounds
  - pg_plan_advice
  - pg_hint_plan
- **Update assumptions**
  - Cost model for modern hardware

Thank You

Want to Go Deeper?

Check out more resources and Slides: