15721 FINAL PRESENTATION Stats & Cost Model

Allison Wang, Xiaoyou Wang, Xian Zhang

Motivation

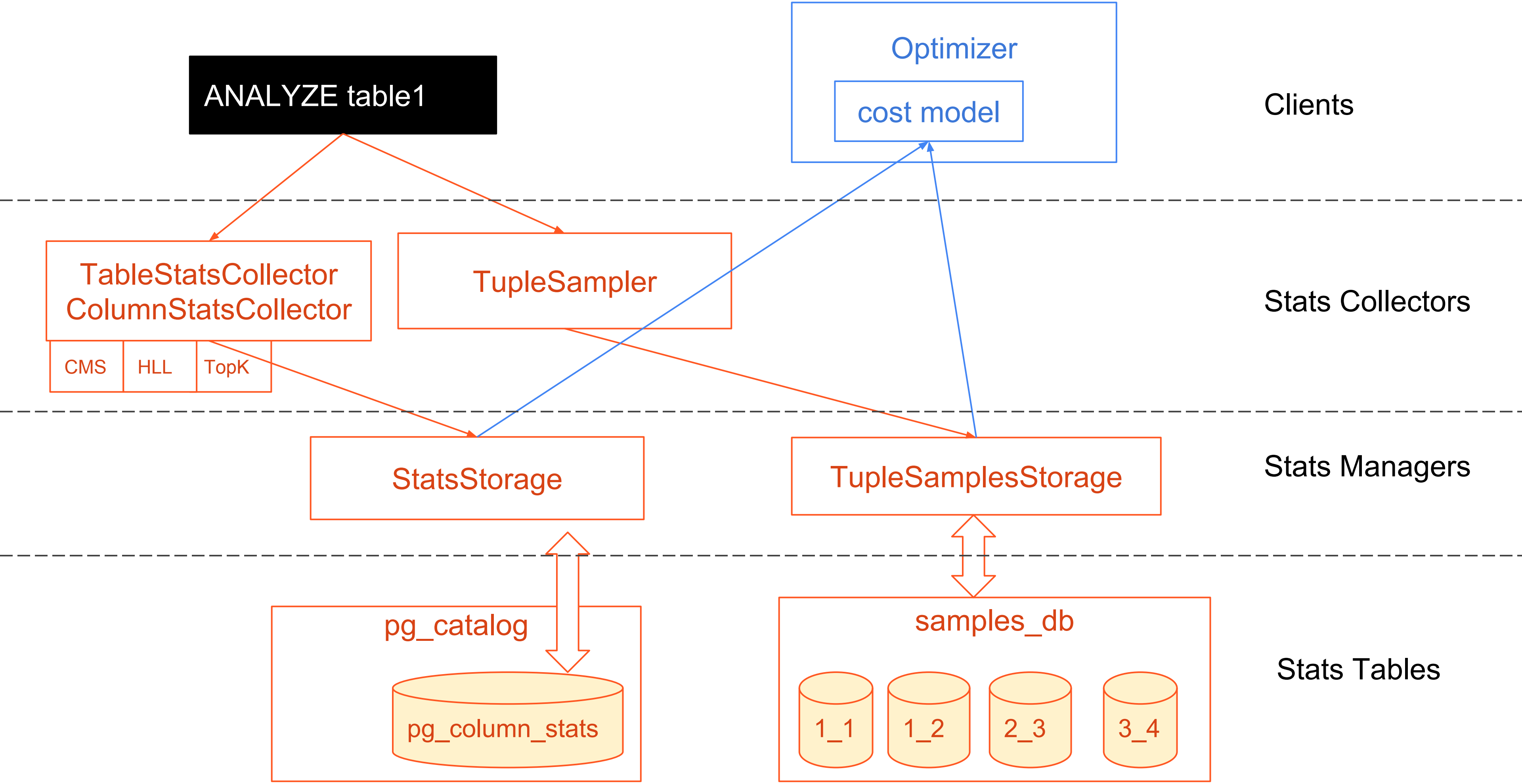
1. Enable the collection of table statistics and cardinality estimation
2. Store stats in Catalog
3. Build cost model for optimizer to predict the optimal query plan

Goals

- 75%: Collect basic table statistics ✓
- 100%: Estimate cardinality and build cost model using statistics ✓
    - Handle a single table query
- 125%: Estimate join query cardinality

We’ve completed the 100% - 5% GOAL !!!

Stats Collection


ANALYZE table1
Optimizer
cost model
Clients
TableStatsCollector
ColumnStatsCollector
TupleSampler
Stats Collectors
CMS
HLL
TopK
StatsStorage
TupleSamplesStorage
Stats Managers
pg_catalog
pg_column_stats
samples_db
1_1
1_2
2_3
3_4
Stats Tables

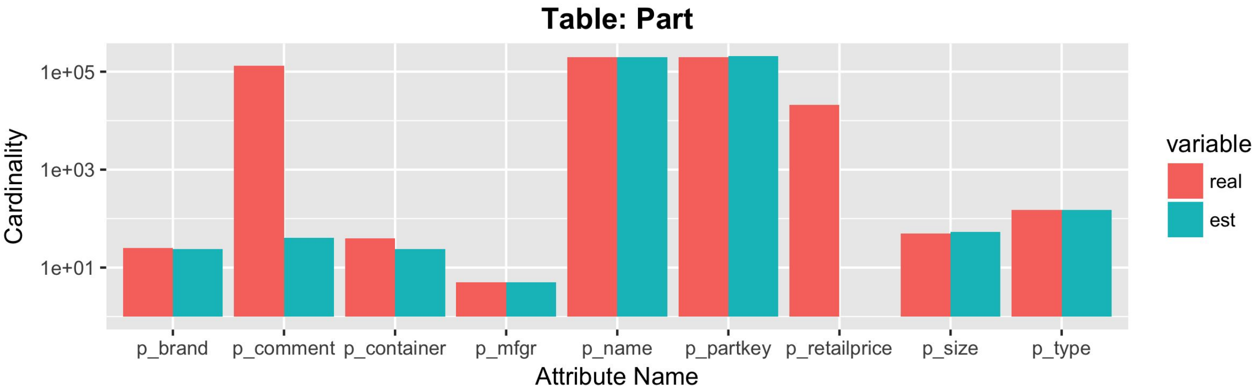
Stats Collectors

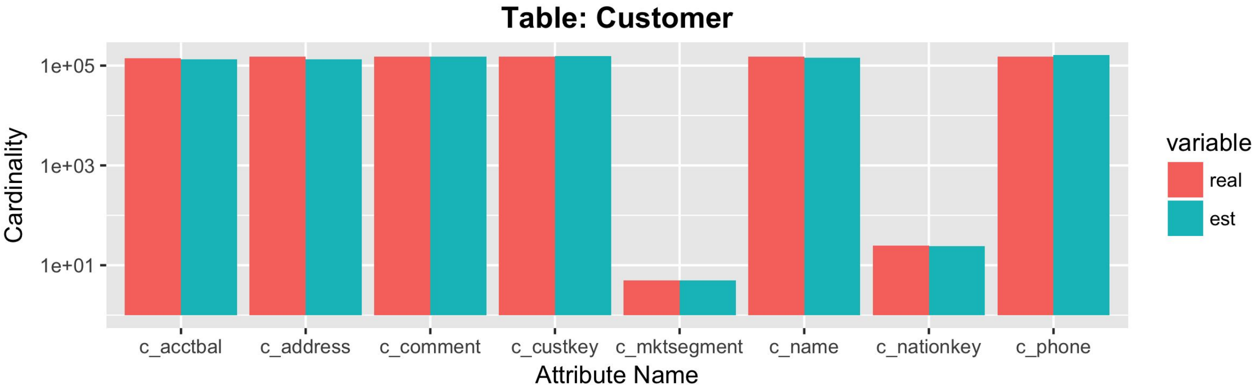
- **cardinality**: hyperloglog
- **most common <values, frequency>** : count-min sketch + topk (priority queue)
- **histogram bounds**: “A Streaming Parallel Decision Tree Algorithm”
- More stats: **num_rows**, **frac_null**

Testing

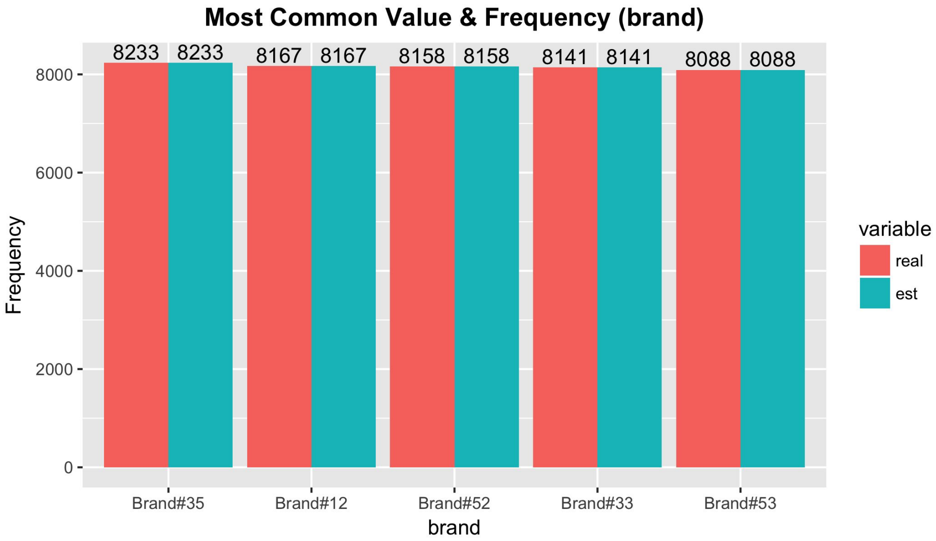
- 2 types of tests: correctness and performance
- Full coverage: comprehensive unit tests suite for each class
  - 12 test files under optimizer folder
- Testing on **TPC-H** benchmark (1GB dataset)
  - extra work: fixed some TPC-H bugs in Peloton
- Customer Table: 24MB (150000 tuples)
- Part Table: 24MB (200000 tuples)
- Lineitem Table: 725MB (6001215 tuples)

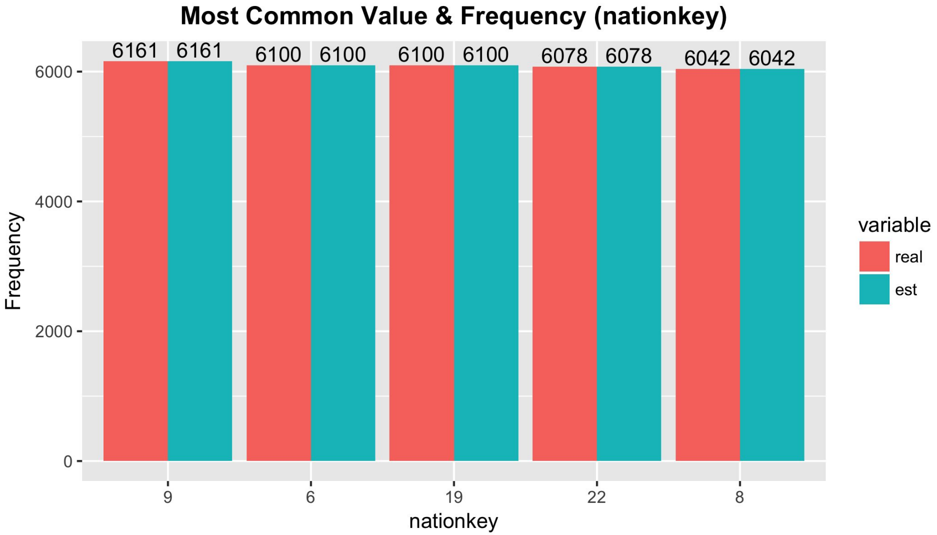
Result: Cardinality Estimation



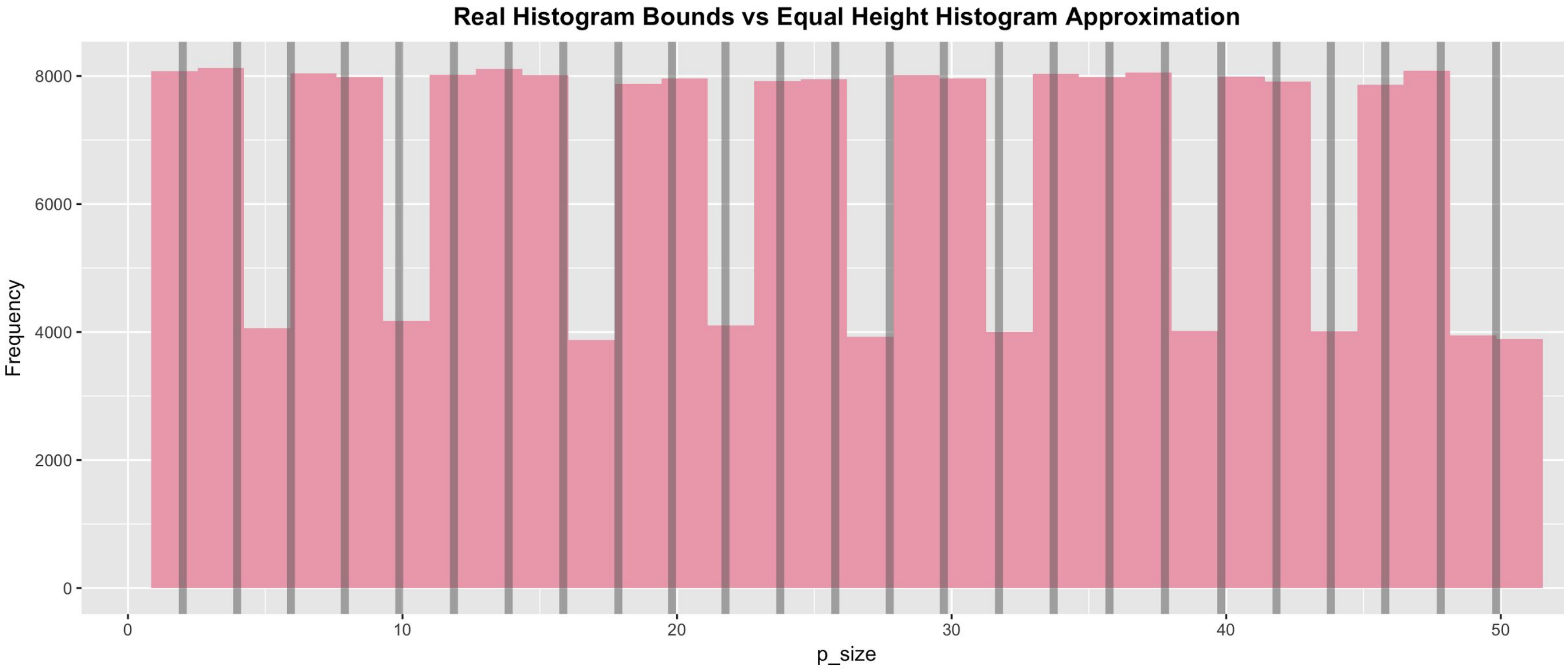


Result: Most Frequent Value & Freq





Result: Histogram Bounds



ANALYZE

- Two interfaces for collecting table stats
  - AnalyzeStatsForTable, AnalzeStatsForAllTables
- Implement command ANALYZE to collect stats for a specific table
- Demo:

```
postgres=#
```

Cost Model

Cost Model (Single Table)

SELECT id, name, COUNT(project_id)

FROM table

WHERE class_id = 15721 AND year < 2017

GROUP BY id, name

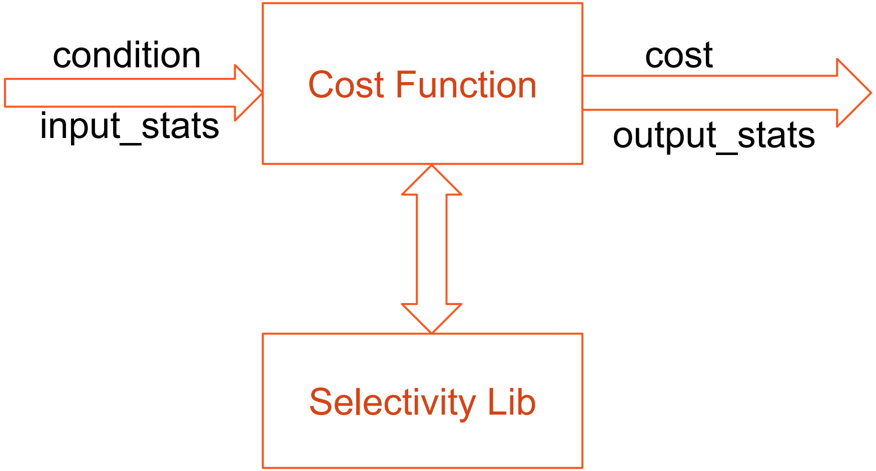
LIMIT 10

$\text{sel}\ (P1 \wedge P2) = \text{sel}(P1) * \text{sel}(P2)$

$\text{sel}\ (P1 \vee P2) = \text{sel}(P1) + \text{sel}(P2) - \text{sel}\ (P1 \wedge P2)$

Standalone Cost Calculator Lib

- SeqScanCost
- IndexScanCost
- CombineConjunctionStats
- SortGroupByCost
- HashGroupByCost
- AggregateCost
- DistinctCost
- ProjectCost
- LimitCost



Standalone Selectivity Lib

- LessThan
- GreaterThan
- Equal
- NotEqual
- LessThanOrEqualTo
- GreaterThanOrEqualTo
- Like
- NotLike

Estimation Performance

🟦 Actual # rows

⬛ Postgres’ estimation

🟧 Our estimation

```
SELECT * FROM part WHERE p_partkey < 10000;
```

- Actual: **9999** | Postgres: **9906** | Ours: **8080**

```
SELECT * FROM part WHERE p_size = 49;
```

- Actual: **3945** | Postgres: **4213** | Ours: **3695**

```
SELECT * FROM part WHERE p_partkey < 10000 AND p_size = 49;
```

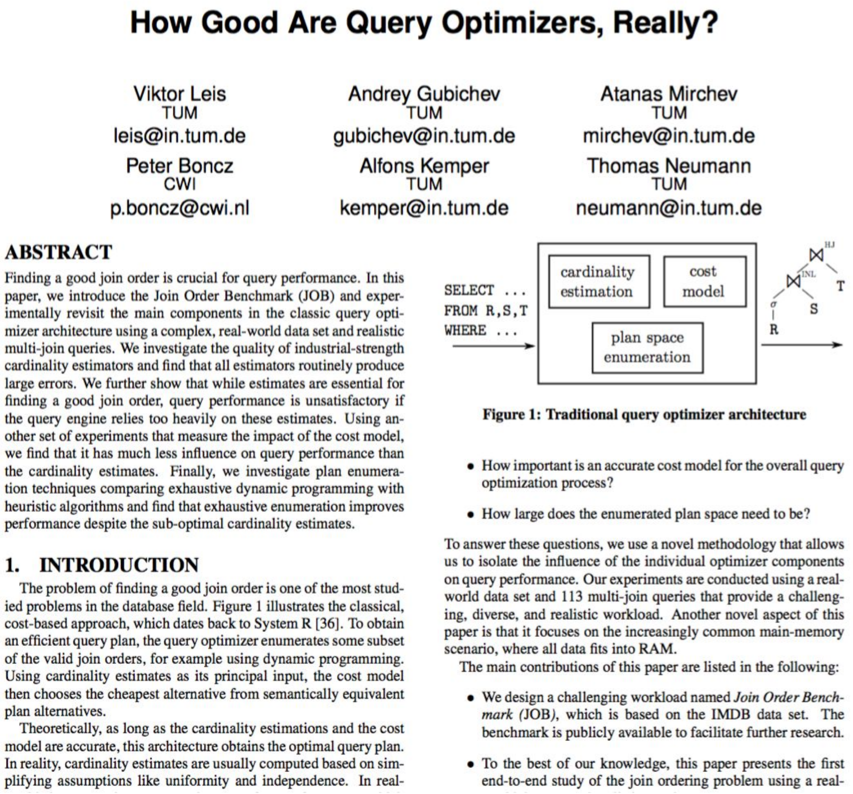
- Actual: **174** | Postgres: **209** | Ours: **149**

```
SELECT * FROM part WHERE p_partkey < 10000 OR p_size = 49;
```

- Actual: **13770** | Postgres: **13911** | Ours: **11625**

Parting Thoughts

- Accurate cardinality estimation is hard
- Good base table statistics lead to better estimation for complicated predicates and joins



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 the cardinality estimates. Finally, we investigate plan enumeration techniques comparing exhaustive dynamic programming with heuristic algorithms and find that exhaustive enumeration improves performance despite the sub-optimal cardinality estimates.

1. INTRODUCTION

The problem of finding a good join order is one of the most studied problems in the database field. Figure 1 illustrates the classical, cost-based approach, which dates back to System R [36]. To obtain an efficient query plan, the query optimizer enumerates some subset of the valid join orders, for example using dynamic programming. Using cardinality estimates as its principal input, the cost model then chooses the cheapest alternative from semantically equivalent plan alternatives.

Theoretically, as long as the cardinality estimations and the cost model are accurate, this architecture obtains the optimal query plan. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. In real-

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-

Next Steps

1. Integrate cost model into optimizer cost and stats calculator
2. Support string equality and cardinality using ARRAY type
3. Support LIKE operator selectivity using sampling
4. Cost model for join operators
5. EXPLAIN + new optimizer to fully test cost model performance

Code Quality and Stats

- **8000+** new lines of code
- **43** new files (excluding third-party)
- Group internal PR and code review: **36** PRs
- Well tested
- Highly modularized

Q & A

Q: What makes you feel heartbroken?

A: make clean