optdbg

query optimizer debugger

David, Yu, Jiaying

Motivation

- No standard tools for evaluating query optimizer performance
- Existing tools are limited in scope
- Detecting cardinality errors is currently done in bespoke ways

The VLDB Journal (2018) 27:643–668
https://doi.org/10.1007/s00778-017-0480-7

CrossMark

SPECIAL ISSUE PAPER

Query optimization through the looking glass, and what we found running the Join Order Benchmark

Viktor Leis[1] · Bernhard Radke[1] · Andrey Gubichev[1] · Atanas Mirchev[1] · Peter Boncz[2] · Alfons Kemper[1] · Thomas Neumann[1]

Received: 22 January 2017 / Revised: 8 August 2017 / Accepted: 11 August 2017 / Published online: 18 September 2017

Abstract Finding a good join order is crucial for query performance. In this paper, we introduce the Join Order Benchmark that works on real-life data riddled with correlations and introduces 113 complex join queries. We experimentally revisit the main components in the classic query optimizer architecture using a complex, real-world data set and realistic multi-join queries. For this purpose, we describe cardinality-estimate injection and extraction techniques that allow us to compare the cardinality estimators of multiple industrial SQL implementations on equal footing, and to characterize the value of having perfect cardinality estimates. Our investigation shows that all industrial-strength cardinality estimators routinely produce large errors: though cardinality estimation using table samples solves the problem for single-table queries, there are still no techniques in industrial systems that can deal accurately with join-crossing correlated query predicates. 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. We investigate plan enumeration techniques comparing exhaustive dynamic programming with heuristic algorithms and find that exhaustive enumeration improves performance despite the suboptimal cardinality estimates. Finally, we extend our investigation from main-memory only, to also include disk-based query processing. Here, we find that though accurate cardinality estimation should be the first priority, other aspects such as modeling random versus sequential I/O are also important to predict query runtime.

Keywords Query optimization · Join ordering · Cardinality estimation · Cost models

✉ Viktor Leis
leis@in.tum.de

Bernhard Radke
radke@in.tum.de

Andrey Gubichev
gubichev@in.tum.de

Atanas Mirchev
mirchev@in.tum.de

Peter Boncz
p.boncz@cwi.nl

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

1 Technische Universität München, Garching, Germany

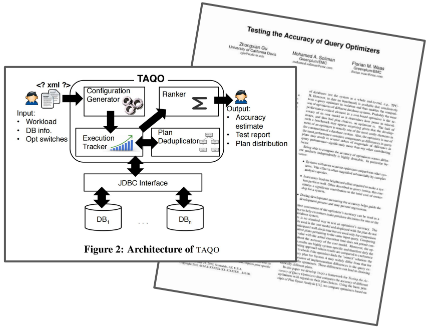
2 CWI, Amsterdam, The Netherlands

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 [45]. 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-world data sets, these assump-

Can we automate this?



TAQO (Greenplum)

\+ Hint-based sampling, hardware-independent

− Purely numerical evaluation, questionable benchmarking

OptMark (Li et al.)

\+ Intelligent sample sizing, efficiency metrics

− Purely focused on numerical evaluation

AutoDI (Lan et al.)

\+ Static analysis of plans to explain regressions

− Requires known regression, loose analysis

Goals

75% – end-to-end product combining evaluation with static analysis

(originally: also support multiple databases – axed by status update)

we are here

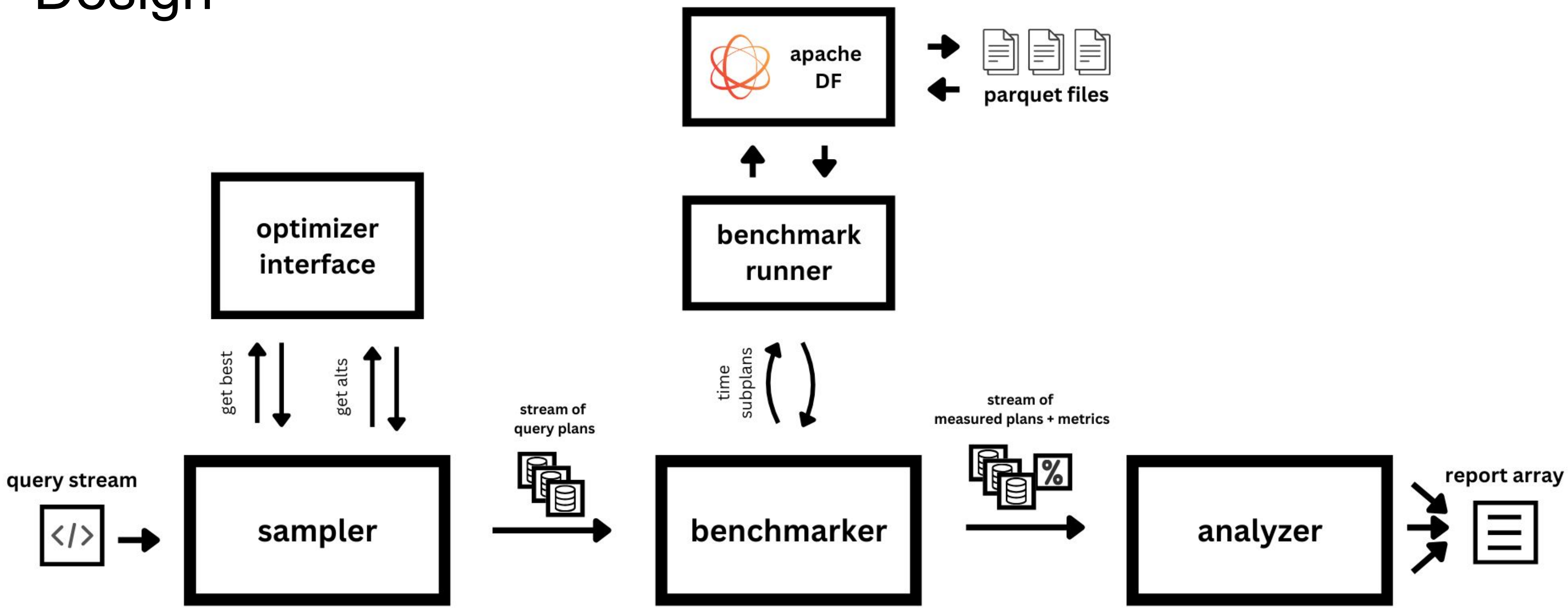
100% – incremental improvements for each component

(e.g. better plan sampling, more rigorous benchmarking, better analysis, …)

125% – fuzzing, fast execution via covering query optimization

(SQL Server optimization that runs min # of queries to get true cards)

Design



apache
DF
parquet files
optimizer
interface
benchmark
runner
get best
get alts
time
subplans
stream of
query plans
stream of
measured plans + metrics
query stream
</>
sampler
benchmarker
analyzer
report array
%

Sampler (Rule based)

```
for rule_subset in powerset(default_rules):
    optimizer = DataFusionOptimizer(rules=rule_subset)
    plan = optimizer.optimize(logical_plan)
    runtime = execute(plan)
    record(rule_subset, runtime, plan)
```

Prefilter rules & bailing strategies

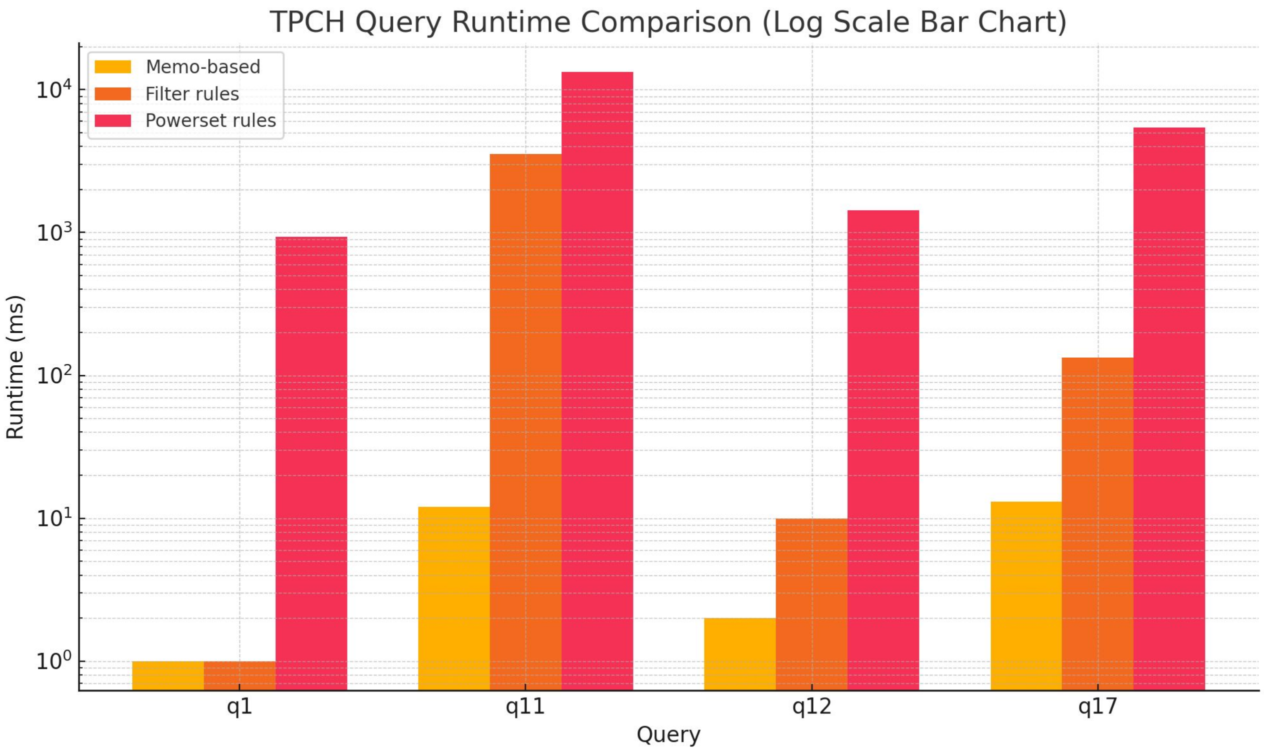
Limitation: Limit diversity on Join ordering

Sampler (Memo based)

```
fn get_alts(group_id):
    exprs = memo.get_group(group_id).physical_exprs
    alts = []
    for expr in exprs:
        child_alts = [get_alts(child_gid) for child_gid in expr.children]
        for combo in cartesian_product(child_alts):
            plan = build_plan(expr, combo)
            cost = compute_cost(plan)
            alts.append((plan, cost))
    return alts
```

Must recompute cost/stat for alternatives

Runtime of different sampling methods

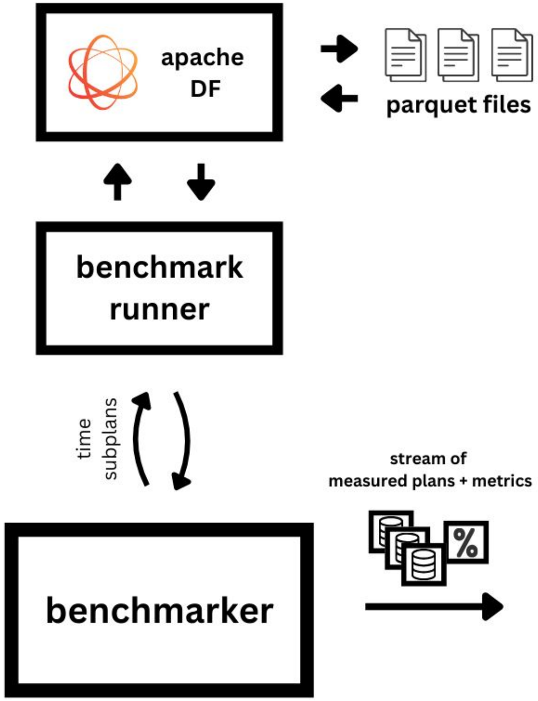


run on TPC-H queries 1, 11, 12, 17 on data with scale factor 0.2

Benchmark Execution Engine

Efficient **top-down** measurement with timeout and caching support

- Recursively run benchmarks for each subplan in the plans returned by sampler
- Subplans run in separate subprocesses via runner
  - Serialize physical plans via `datafusion_proto`
- Collect true cardinalities for each subplan
- Run multiple times to collect metrics, with outlier filtering applied



Benchmark Execution Engine

Early Stopping

- Skip running subplans of children of failed/timeout plans (configurable)

Subplan Caching

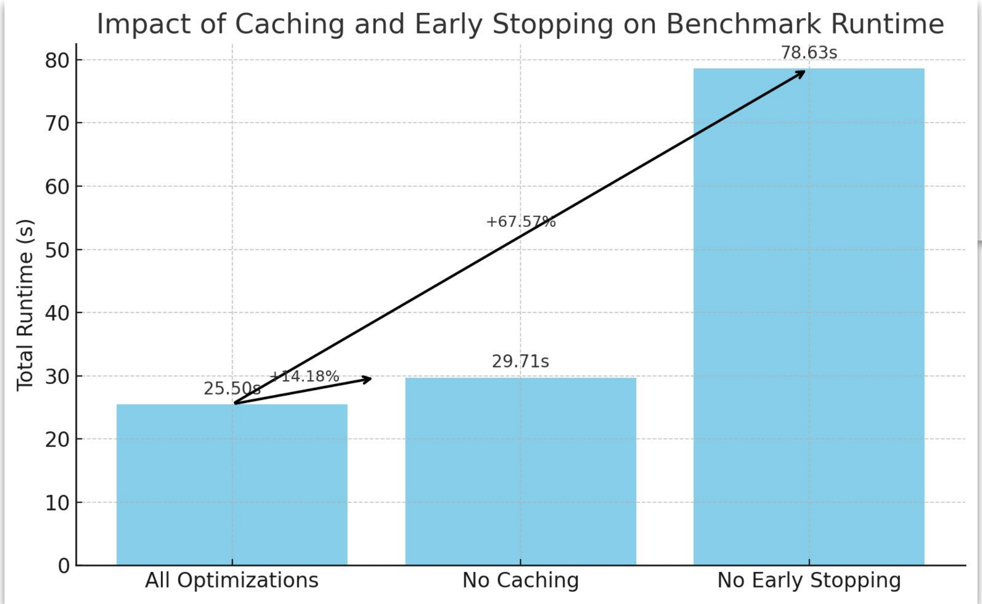
- Avoid re-running subplans across queries
- Maintain small hash set of metrics for small subplans
    - Runtime / space usage tradeoff

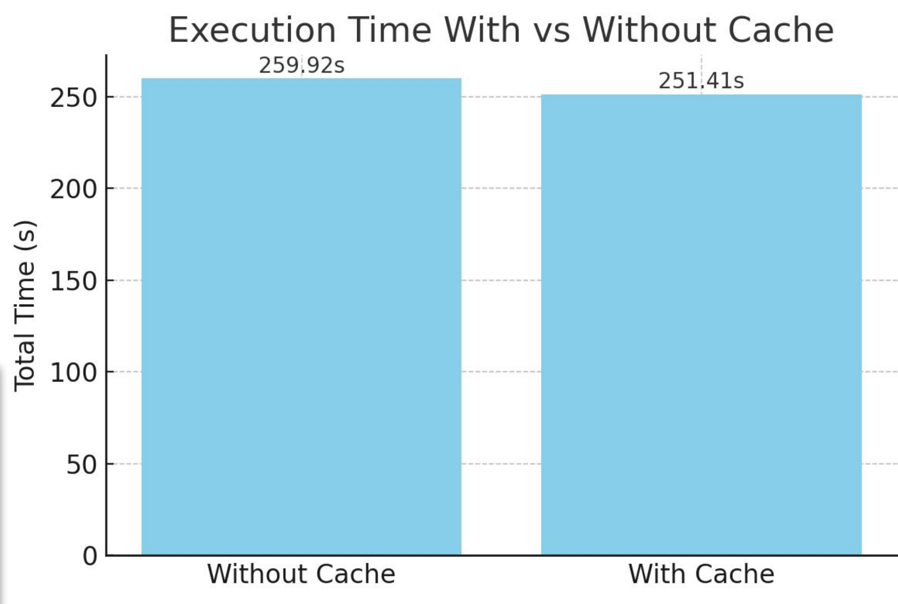
Confidence-aware Comparison

- Plans are compared not just by mean time, but also stddev overlap
- Two plans are "equal" if runtime ranges significantly overlap
- Less sensitivity to noise in final results

Performance Metrics

Cache hit rate: 59.8%





run on TPC-H queries 1, 11, 12, 17 on data with scale factor 0.2

Benchmark Metrics

Accuracy — *Does the optimizer rank plan correctly?*

Metric	Meaning	Formula / Definition
TAQO Score (s)	Measures agreement between cost & runtime rankings	Weighted Kendall's Tau
TAQO Accuracy (%)	Normalized TAQO score (higher is better)	100 × exp(-|s|)
Performance Factor	Fraction of plans slower than the optimizer's pick	$\mathrm{PF} = \frac{\lvert\{p \in P \mid T(p) \geq T(\mathrm{opt})\}\rvert}{\lvert P \rvert}$
Optimality Frequency	% of queries where optimizer picked the fastest plan	$\mathrm{OF} = \frac{\lvert\{\text{queries with PF=1}\}\rvert}{\#\text{queries}}$

Benchmark Metrics

Quality — *Are the estimates close to reality?*

Metric	Meaning	Formula / Definition
Average Q-Error	Cardinality estimation error (lower is better)	$Q\text{-Error} = \max\left(\frac{\text{estimated}}{\text{actual}}, \frac{\text{actual}}{\text{estimated}}\right)$
Q-Error Distribution	Q-error percentiles (p90, p95, max) & error buckets	Binned as Perfect / Good / Poor / etc.
Cardinality Accuracy	Actual vs estimated rows per subplan (preorder)	Measured during execution

Analysis

- Holistic method rather than pairwise like AutoDI

- Rank all relevant subplans to see cost model errors

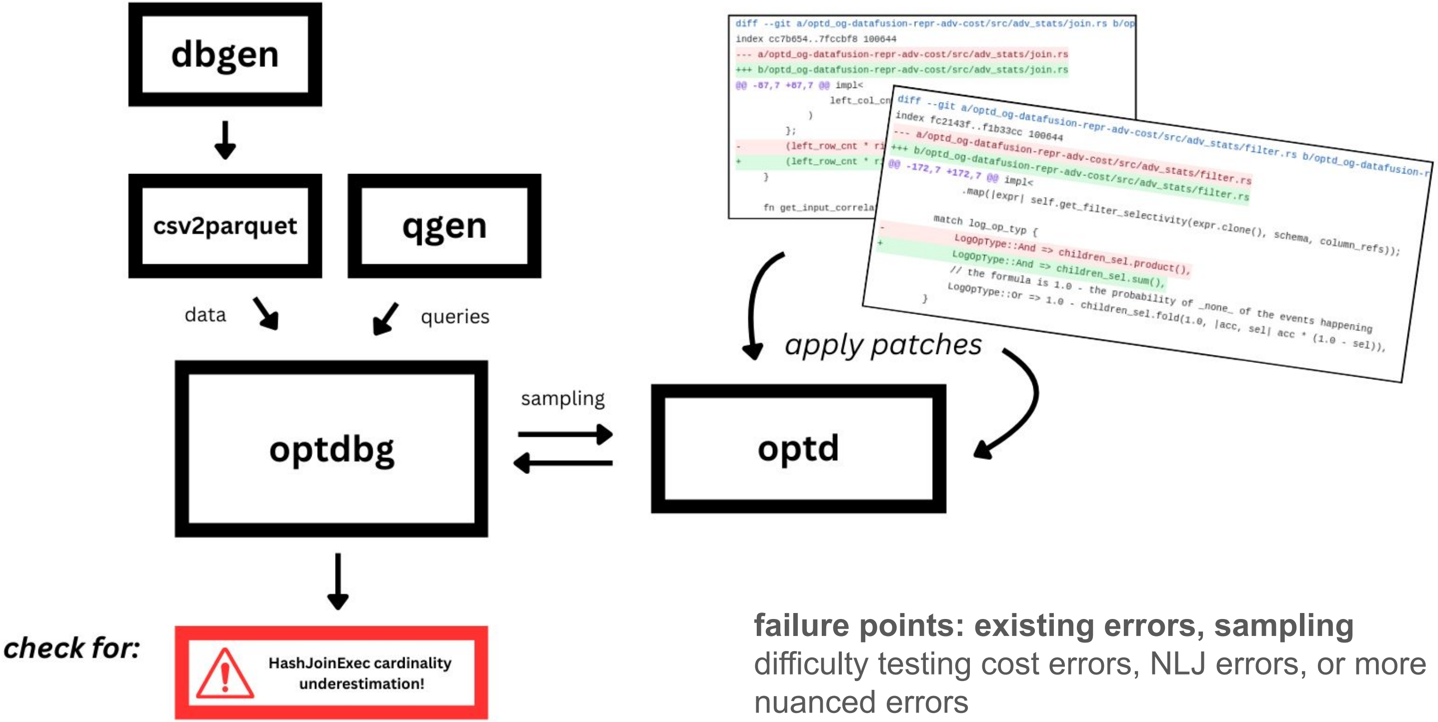
- Detect “problems” by extracting most frequently problematic nodes / predicate types across workload

(sometimes only consider leaf errors)

```
SortExec | [CardinalityMisestimation(1, 115, 115.0), CostMisestimation(47, 10767.0, 0, 1)]
  ProjectionExec | [CardinalityMisestimation(1, 115, 115.0)]
       FilterExec | [CardinalityMisestimation(1, 115, 115.0)]
       CrossJoinExec | [CardinalityMisestimation(1, 6466, 6466.0)]
       AggregateExec | [CardinalityMisestimation(10, 6466, 646.6)]
       ProjectionExec | [CardinalityMisestimation(10, 6880, 688.0)]
              HashJoinExec | [CardinalityMisestimation(10, 6880, 688.0)]
              HashJoinExec |
              FilterExec |
                     DataSourceExec | [CardinalityMisestimation(1000, 25, 40.0)]
              DataSourceExec |
              DataSourceExec | [CardinalityMisestimation(1000, 160000, 160.0)]
       ProjectionExec |
       AggregateExec |
              ProjectionExec | [CardinalityMisestimation(10, 6880, 688.0)]
              HashJoinExec | [CardinalityMisestimation(10, 6880, 688.0)]
              HashJoinExec |
                     FilterExec |
                     DataSourceExec | [CardinalityMisestimation(1000, 25, 40.0)]
                     DataSourceExec |
              DataSourceExec | [CardinalityMisestimation(1000, 160000, 160.0)]
```

```
Evaluated 3 queries.
Optimality Frequency (OF): 100.00 (higher is better)
Average TAQO Score (s): 0.3957 (lower is better)
Average TAQO Accuracy: 76.84% (higher is better)
Average Performance Factor (PF): 100.00%
Global node perf data
{"AggregateExec": ({"cardinality_misestimation": (6, 0)}, 16),
 "ProjectionExec": ({"cardinality_misestimation": (9, 0)}, 23),
 "SortExec": ({"cardinality_misestimation": (4, 0)}, 10),
 "HashJoinExec": ({"cardinality_misestimation": (5, 0)}, 9),
 "FilterExec": ({"cardinality_misestimation": (4, 4)}, 22),
 "NestedLoopJoinExec": ({}, 9),
 "CrossJoinExec": ({"cardinality_misestimation": (2, 0)}, 15),
 "DataSourceExec": ({"cardinality_misestimation": (7, 4)}, 43)}
```

Correctness Testing



failure points: existing errors, sampling
difficulty testing cost errors, NLJ errors, or more nuanced errors

Unit Testing & Code Coverage

Current Coverage: **34%** overall (measured via **cargo-tarpaulin**)

- Project centers on runtime benchmarking, so unit testability is inherently limited.
- Sampler logic is tightly coupled with:
  - Requires memo table traversal, optimizer state, or rule combinations, making it hard to isolate in pure unit tests
- Benchmark layer hard to test:
  - Involves subprocess execution, IPC serialization, timeouts, and runtime measurements — all difficult to mock and not practical for unit tests

Code Quality

- Strong point: design and interfaces reasonably good
  - Loosely designed with 125% extensions in mind
- Strong point: reasonable for a DBMS to implement support
  - Even patched datafusion-dolomite a bit!
- Notably hacky components include:
  - Memo-based plan sampling
  - Physical expression extraction
    - This would complicate covering queries optimization!
  - Problem deduction (prefer a more general pattern matching system)

Future Work

Sampling: correctly extract costs and fully flesh out memo-based sampler

far future: implement optimizer hint support in optd?

Benchmarking: integrate with existing Rust benchmarking libraries

Analysis: implement more robust pattern matching, track error growth (and cancellation)

Testing: set up cardinality injection for optd to isolate inserted bugs

+ Implementing 125% goals!