Carnegie Mellon University

An Extensible Framework for Improving Postgres Plans via Hints

MAY 2, 2025

Bobby Norwood, Wenda Fu, Xueqi Li

Image: ChatGPT, prompt: “Can you create a logo for me for a software called AutoHint? AutoHint is an extension that automatically adds hints to Postgres. The logo should be database themed”

Agenda

1. Goals (What we wanted to do)
2. Design (What we actually made)
3. Results! (Pretty colors that show what happened)

Motivation

Research question: **Can we use hints to enable an external query optimizer to influence Postgres plans without modifying Postgres?**

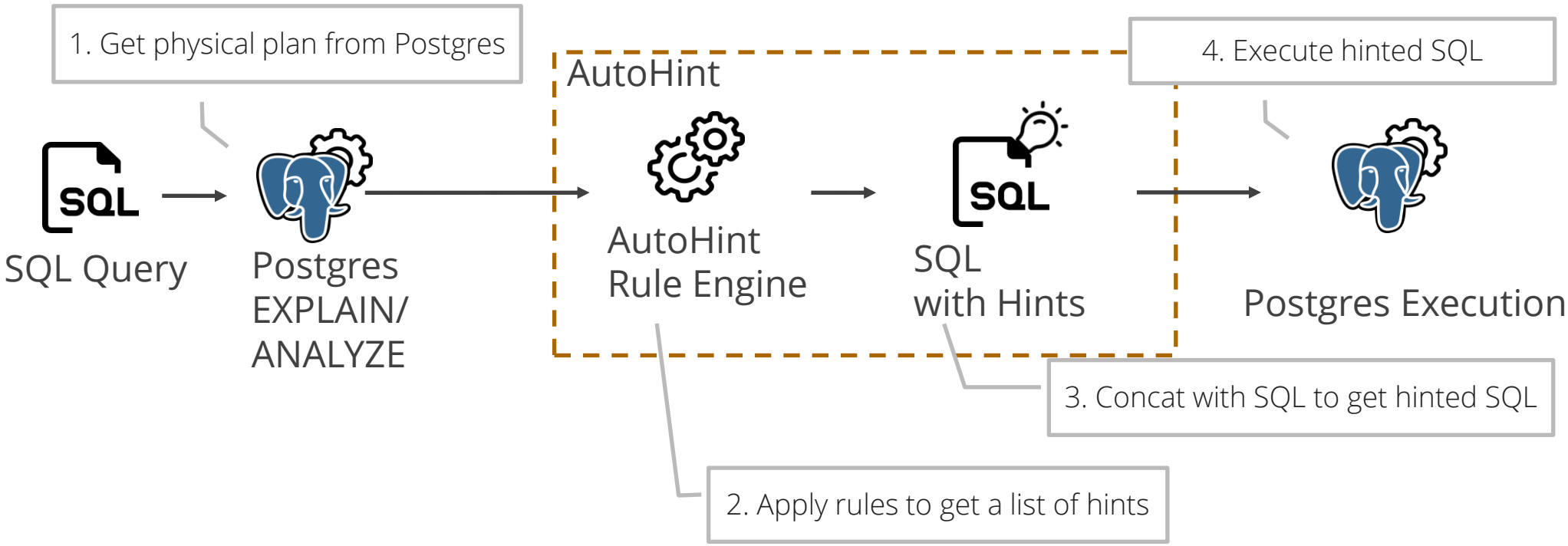
1. Perform preliminary work on converting a Physical Plan back to SQL, in preparation for adding a SQL adapter to optd in the Fall.
2. Experiment with automatically generating hints based on a Physical Plan
3. Combine these motivations to explore using hints to improve bad Postgres query plans

Goals

- 75%
  - Manual examples of improving plan by changing hints - Done
  - Parse Postgres json to sql, no hints - LIMIT, UNION ALL, INDEX SCAN, Hash Join, NLJ, Seq Scan, Projects

  100%
  - Parse postgres internal plan json to sql with hints (join order, access method, join algorithm) - Done (we add hints to original SQL without actually converting plan JSON to SQL)
  - Optimize NLJ to hash join (Done), SELECT X + 0 (Can't change with hints)
  - Results easily replicable, can work with any SQL query - Done
- 125%
  - Solve additional Postgres problems besides those mentioned above (Inject join cardinality)
  - Package as a Postgres extension - No
  - Improve plan parser to also convert to other SQL dialects, from different plan types - No

AutoHint Architecture

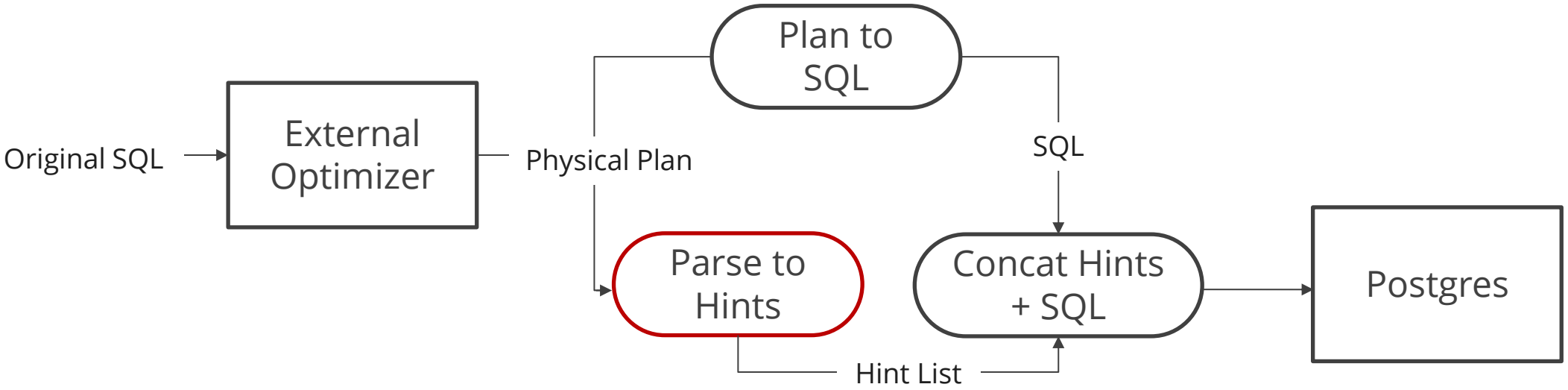


Features

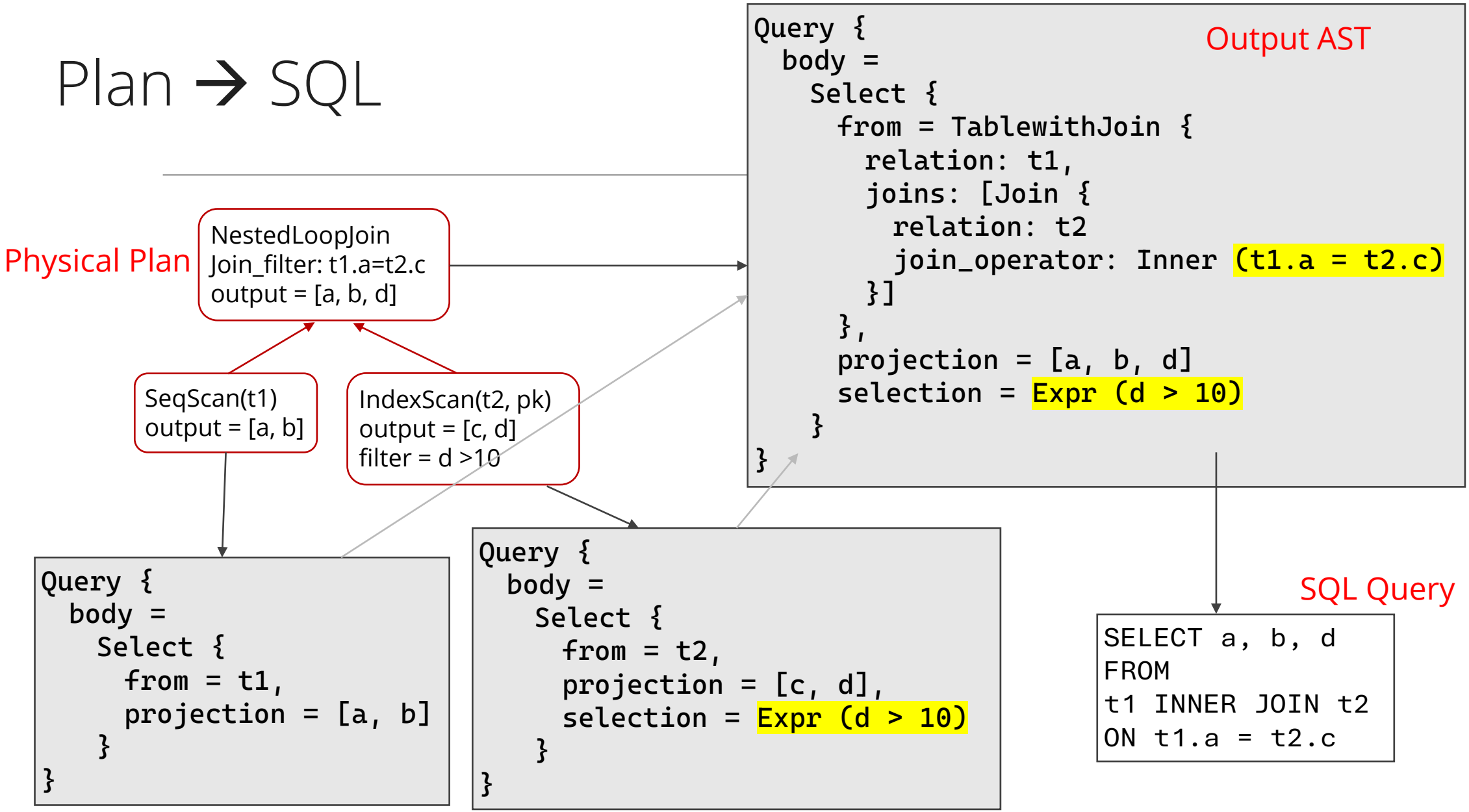
- Can save hints to hint table – automatically applied to similar queries
- Can connect to any Postgres DB with ability to install pg_hint_plan
- Uses Postgres default Rust crate for results and connections, so easy to add into existing workflows
- Two fully implemented rules:
  - Cardinality injection → for each join, create a hint that says how many results it produces (`Rows(A B C D E #123)`)
  - NLJ to HashJoin → Convertes NLJ to hashjoin when NLJ yields more than some given rows (`HashJoin(A)`)
  - Trait defined to easily implement more rules

Initial Approach: Convert and Hint

- Rebuild a new SQL query from optimized plan, and add hints.
- Difficult, but can capture additional nuance from the plan structure.



Plan → SQL


Physical Plan
NestedLoopJoin
Join_filter: t1.a=t2.c
output = [a, b, d]
SeqScan(t1)
output = [a, b]
IndexScan(t2, pk)
output = [c, d]
filter = d >10
Output AST
Query {
body =
Select {
from = TablewithJoin {
relation: t1,
joins: [Join {
relation: t2
join_operator: Inner (t1.a = t2.c)
}]
},
projection = [a, b, d]
selection = Expr (d > 10)
}
}
Query {
body =
Select {
from = t1,
projection = [a, b]
}
}
Query {
body =
Select {
from = t2,
projection = [c, d],
selection = Expr (d > 10)
}
}
SQL Query
SELECT a, b, d
FROM
t1 INNER JOIN t2
ON t1.a = t2.c

Problem: EXPLAIN plans lose data!

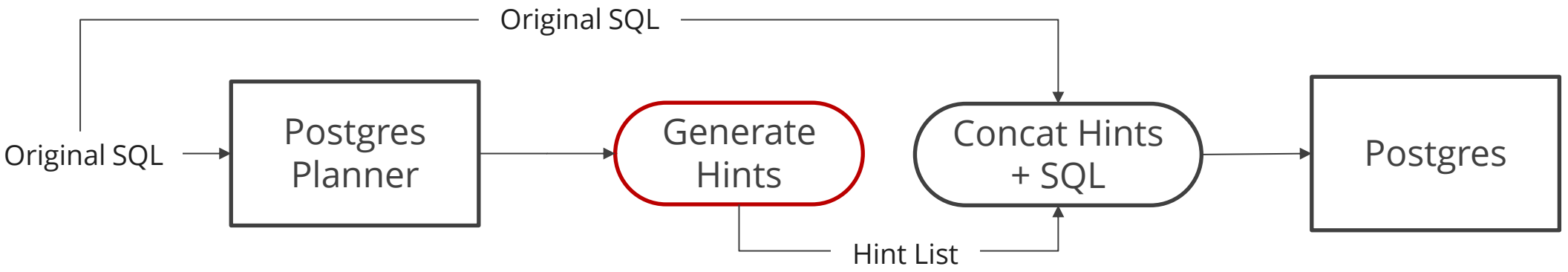
Original SQL: `SELECT primarytitle FROM title_basics ORDER BY startyear, primarytitle LIMIT 10;`

Reconstructed SQL: `SELECT primarytitle, startyear FROM title_basics AS title_basics ORDER BY title_basics.startyear ASC, title_basics.primarytitle ASC LIMIT 10`

Problem: Postgres EXPLAIN plains add columns to output they shouldn't

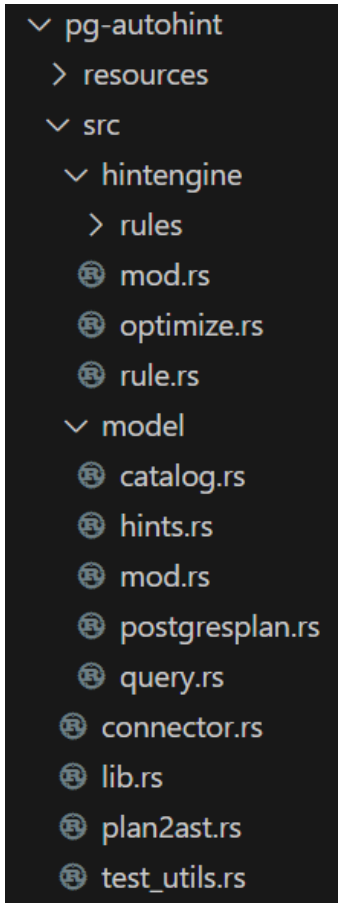
Final Approach: Hint Only

- Create a list of hints from the plan, concatenate with original SQL.
- Sufficient when we aren't really using an external optimizer



Code Quality

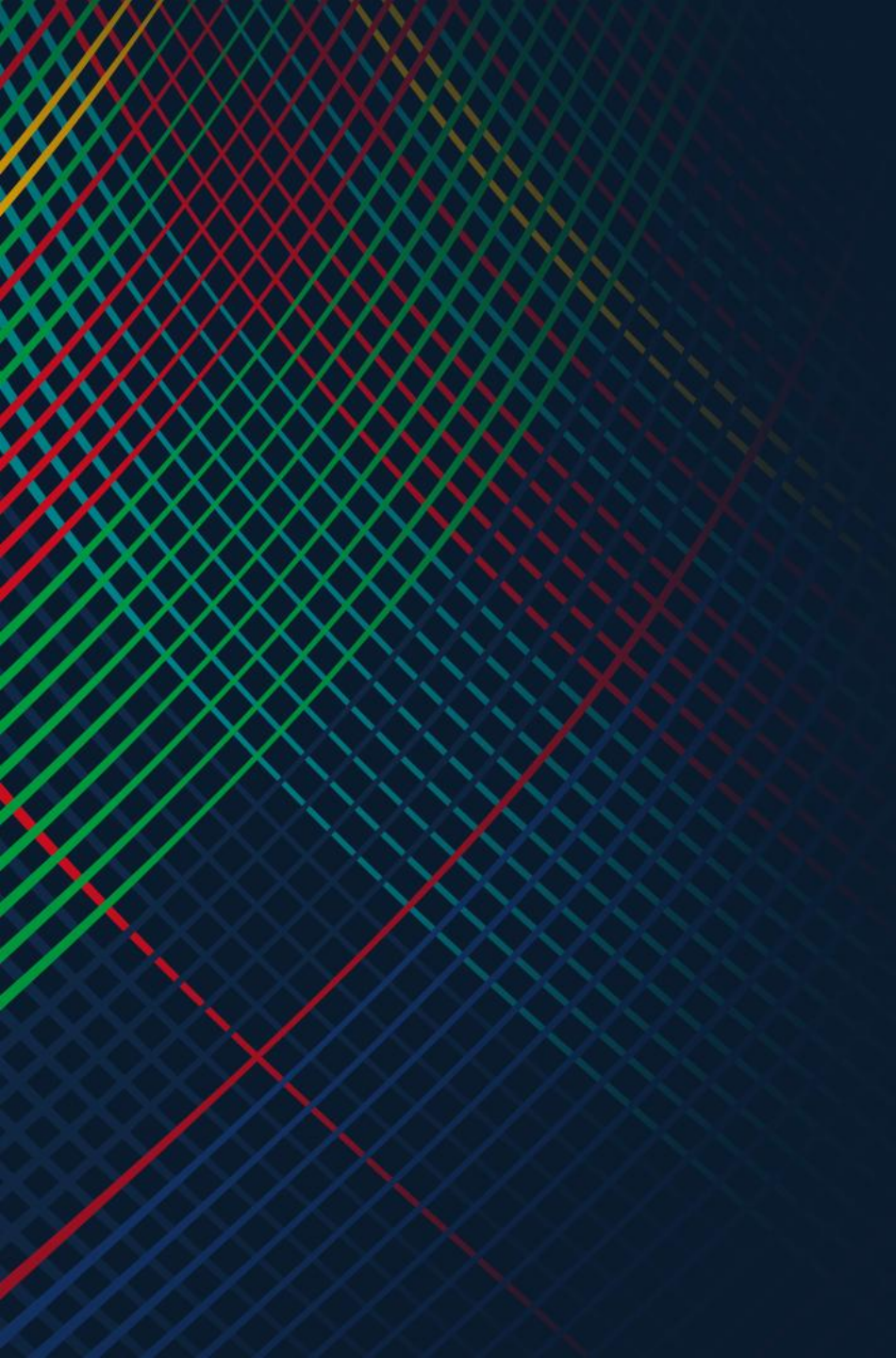
- Strengths
  - Clearly defined implementation for adding new rules
  - Interface for accessing catalog information like indexes and including in rules
  - Scripts to run JOB, scripts to setup test DB (mostly)
- Weaknesses:
  - Converting Postgres plans does not cover all node types – some half-built conversions left in code – but works without due to our implementation choice
  - Not at 100% code coverage
  - Tests require loading the test database, a few manual steps



Test Coverage

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
connector.rs	100.00% (2/2)	95.45% (21/22)	80.00% (4/5)	- (0/0)
hintengine/optimize.rs	85.00% (17/20)	88.00% (220/250)	81.94% (59/72)	- (0/0)
hintengine/rules/card_correction_rule.rs	88.89% (8/9)	75.31% (61/81)	63.16% (24/38)	- (0/0)
hintengine/rules/nlj_to_hashjoin_rule.rs	76.92% (10/13)	73.39% (80/109)	63.04% (29/46)	- (0/0)
hintengine/rules/order_by_incorrect_index_rule.rs	100.00% (6/6)	84.85% (56/66)	84.85% (28/33)	- (0/0)
model/catalog.rs	100.00% (4/4)	100.00% (85/85)	100.00% (17/17)	- (0/0)
model/hints.rs	100.00% (11/11)	98.28% (114/116)	95.65% (44/46)	- (0/0)
model/postgresplan.rs	61.76% (21/34)	36.98% (98/265)	32.31% (74/229)	- (0/0)
model/query.rs	69.57% (16/23)	81.94% (118/144)	63.51% (47/74)	- (0/0)
plan2ast.rs	73.44% (47/64)	78.26% (468/598)	63.16% (180/285)	- (0/0)
test_utils.rs	27.78% (5/18)	51.26% (61/119)	45.28% (24/53)	- (0/0)
Totals	**72.06% (147/204)**	**74.50% (1382/1855)**	**59.02% (530/898)**	**- (0/0)**

Most of the uncovered code is either partially implemented Postgres to SQL converter nodes or error cases



Experiments

Setup

- Experiment run on AWS EC2 z1d.2xlarge (8 Intel Xeon vCPU, 64GB memory), using single core and all cores.
- Postgres 17.4. Query timeout of 40 minutes. Each setting is run 3 times and we take average of the execution time.
- JOB benchmark
- "Single core" vs "multi core": max_parallel_workers_per_gather = 0 vs 2

- Single core Result summaries (excluding timeouts):
  - Average Speedup: 20.95x
  - Max Speedup: 471.34x
  - Timeout queries: 9 -> 11

Fork JOB repo: https://github.com/danolivo/jo-bench

Results

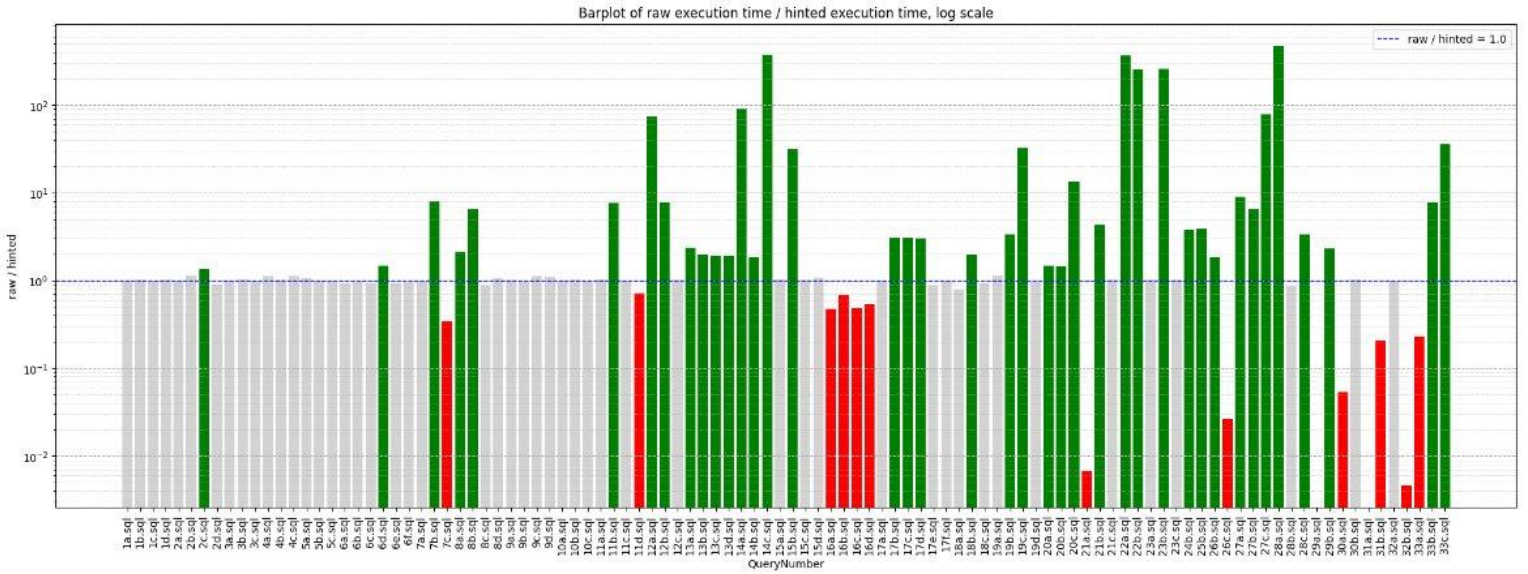
Average Execution Time on Whole JOB Workload (Single Core)

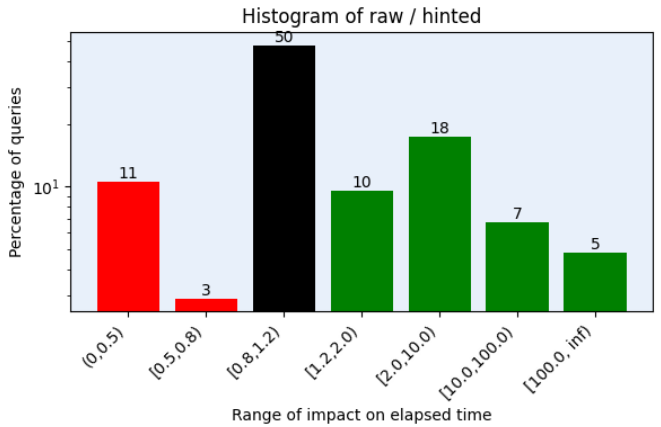
Setting	Average Execution Time (ms)	Average Standard Deviation	Number of Timeout Queries
Autohint-2-rules	47,477.85	81.22	11
Autohint-card-only	31,346.74	62.55	13
Postgres	114,184.99	1,056.78	9

Average Execution Time on Whole JOB Workload (8 Cores)

Setting	Average Execution Time (ms)	Average Standard Deviation	Number of Timeout Queries
Autohint-2-rules	2,561.99	32.80	0
Autohint-card-only	2,617.72	30.45	0
Postgres	1,852.03	44.41	0

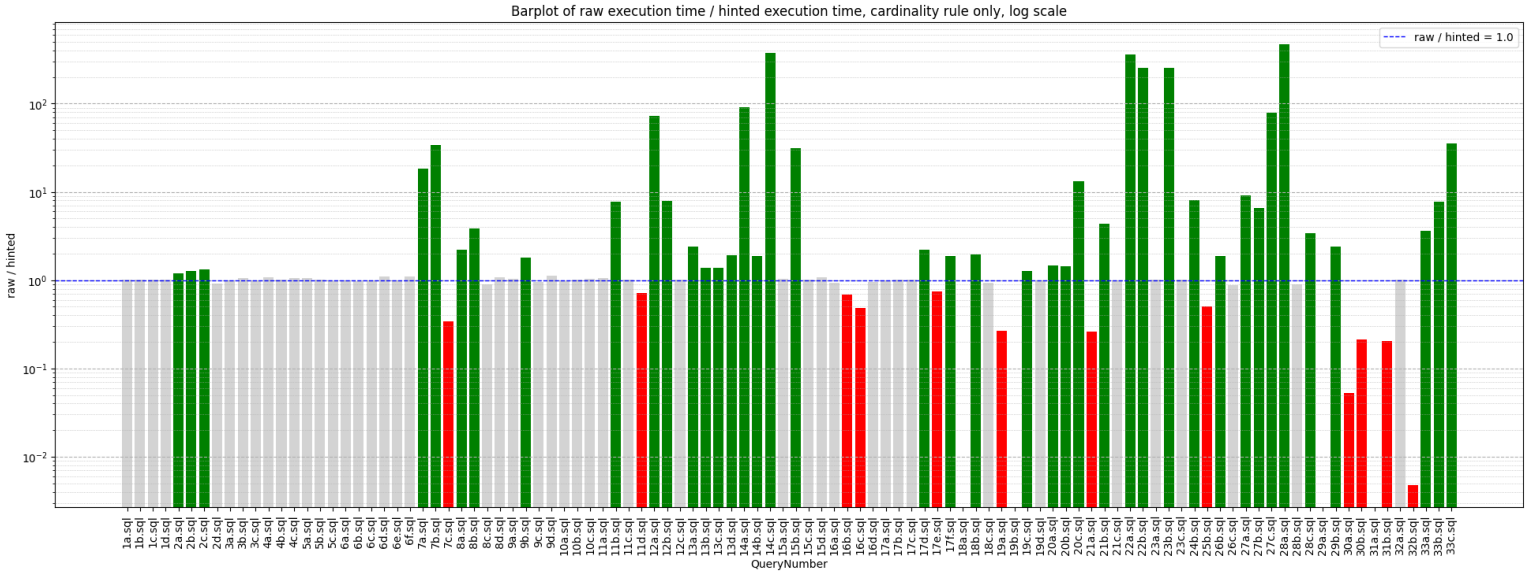
Single Core, both rules

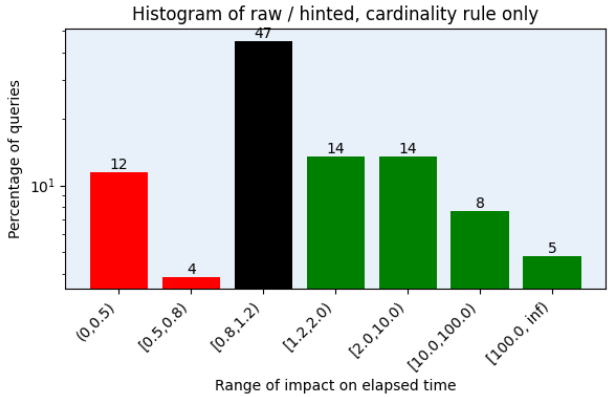




Setting	Average Execution Time (ms) Excluding Timeout Queries	Timeout Queries
Hinted, Both Rules	47477.85	11
Postgres	114184.99	9

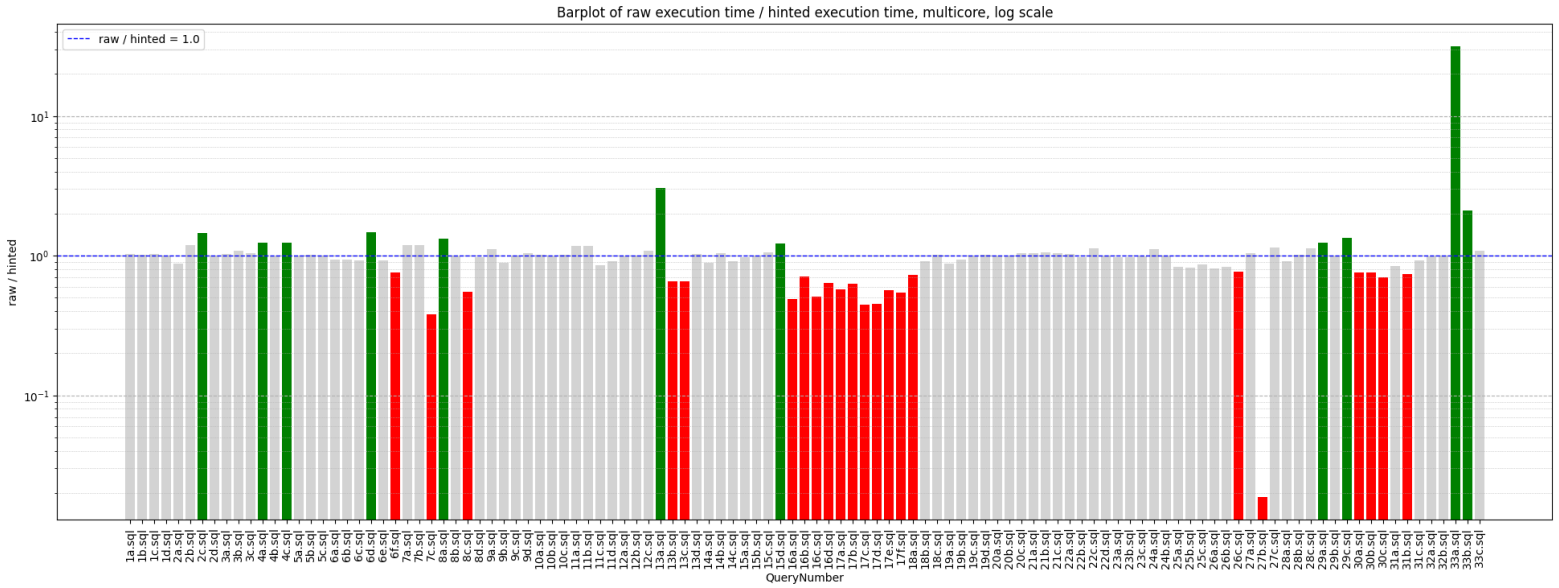
Single Core, cardinality only

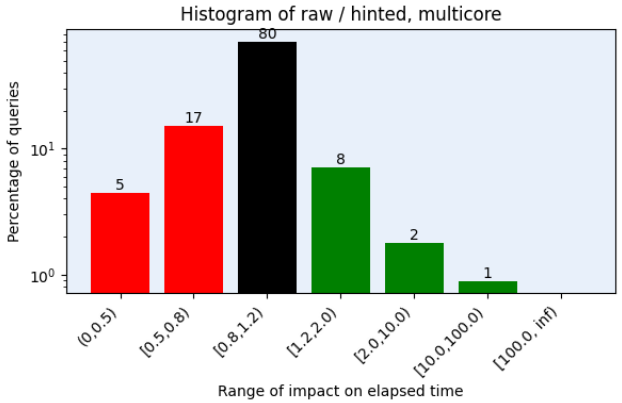




Setting	Average Execution Time (ms) Excluding Timeout Queries	Timeout Queries
Hinted, Cardinality Rule Only	31346.74	13
Postgres	114184.99	9

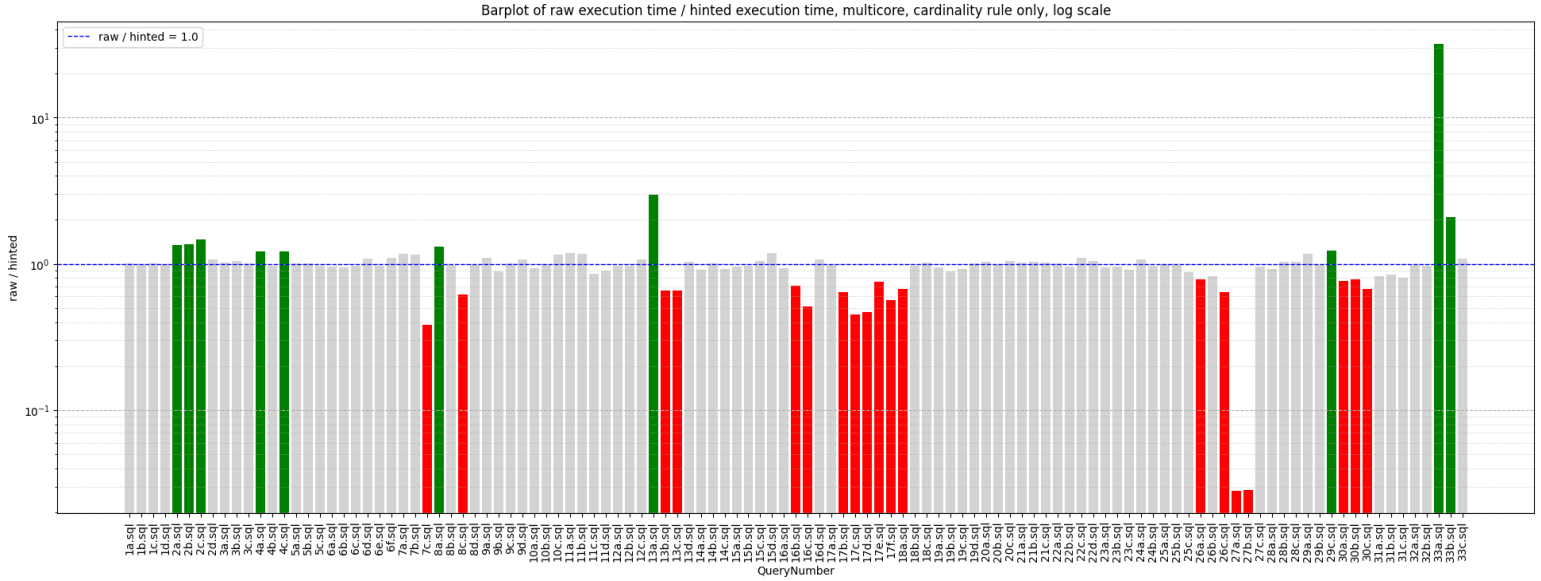
Multicore, both rules

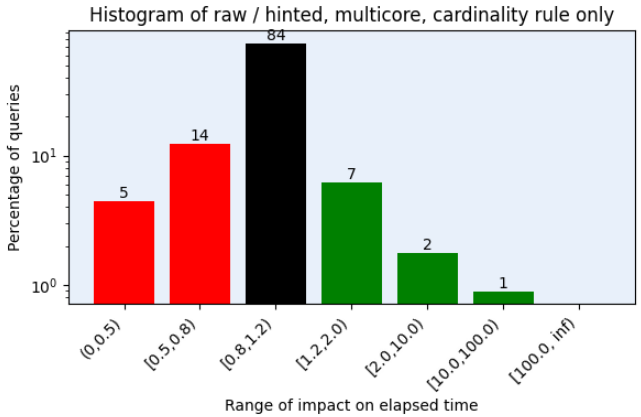




Setting	Average Execution Time (ms) Excluding Timeout Queries	Timeout Queries
Hinted, Both Rules	2561.99	0
Postgres	1852.03	0

Multicore, cardinality only





Setting	Average Execution Time (ms) Excluding Timeout Queries	Timeout Queries
Hinted, Cardinality Rule Only	2617.72	0
Postgres	1852.03	0

Plan Comparison – 22a (single core hint improve)

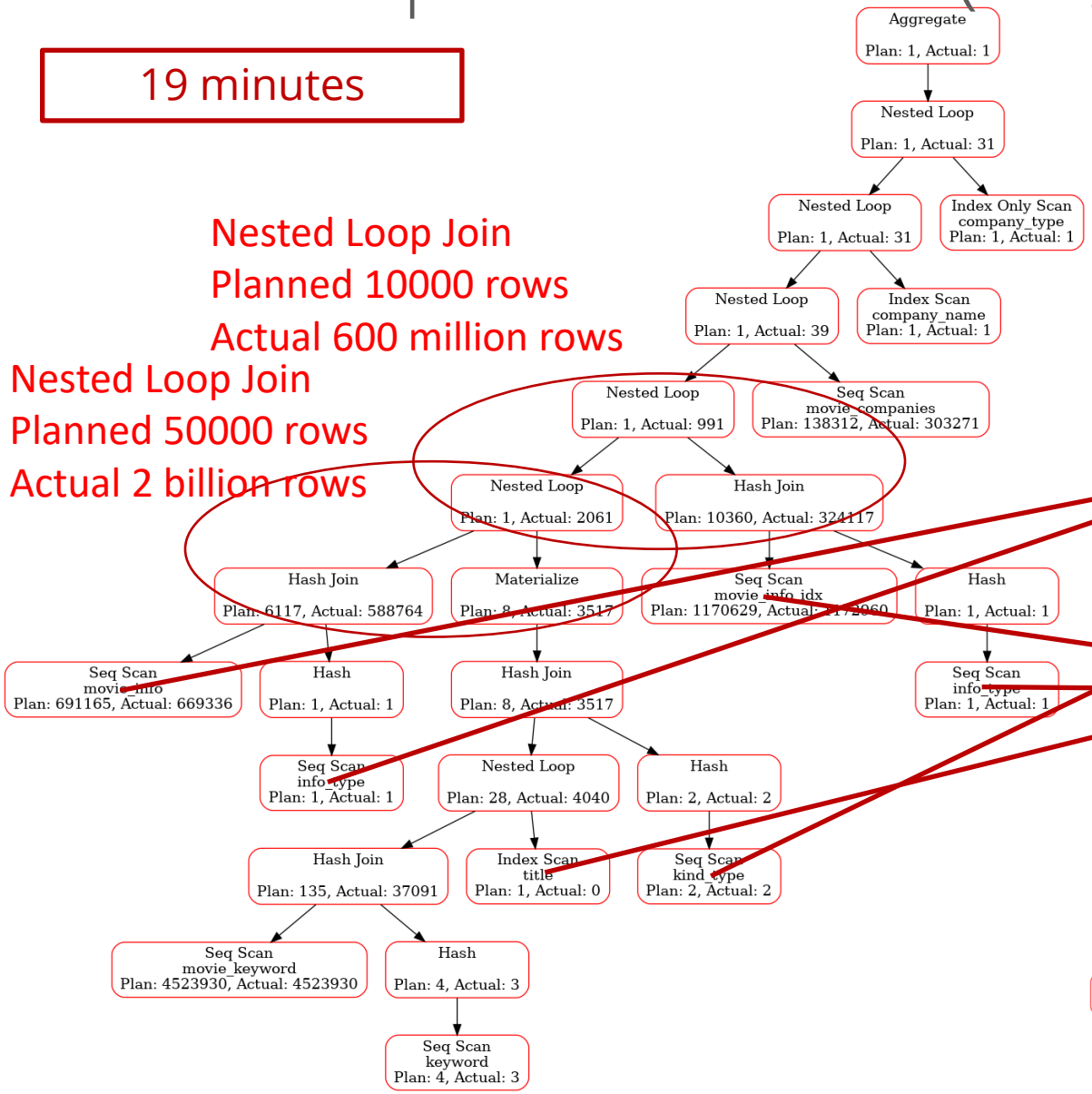
19 minutes

3.17 seconds

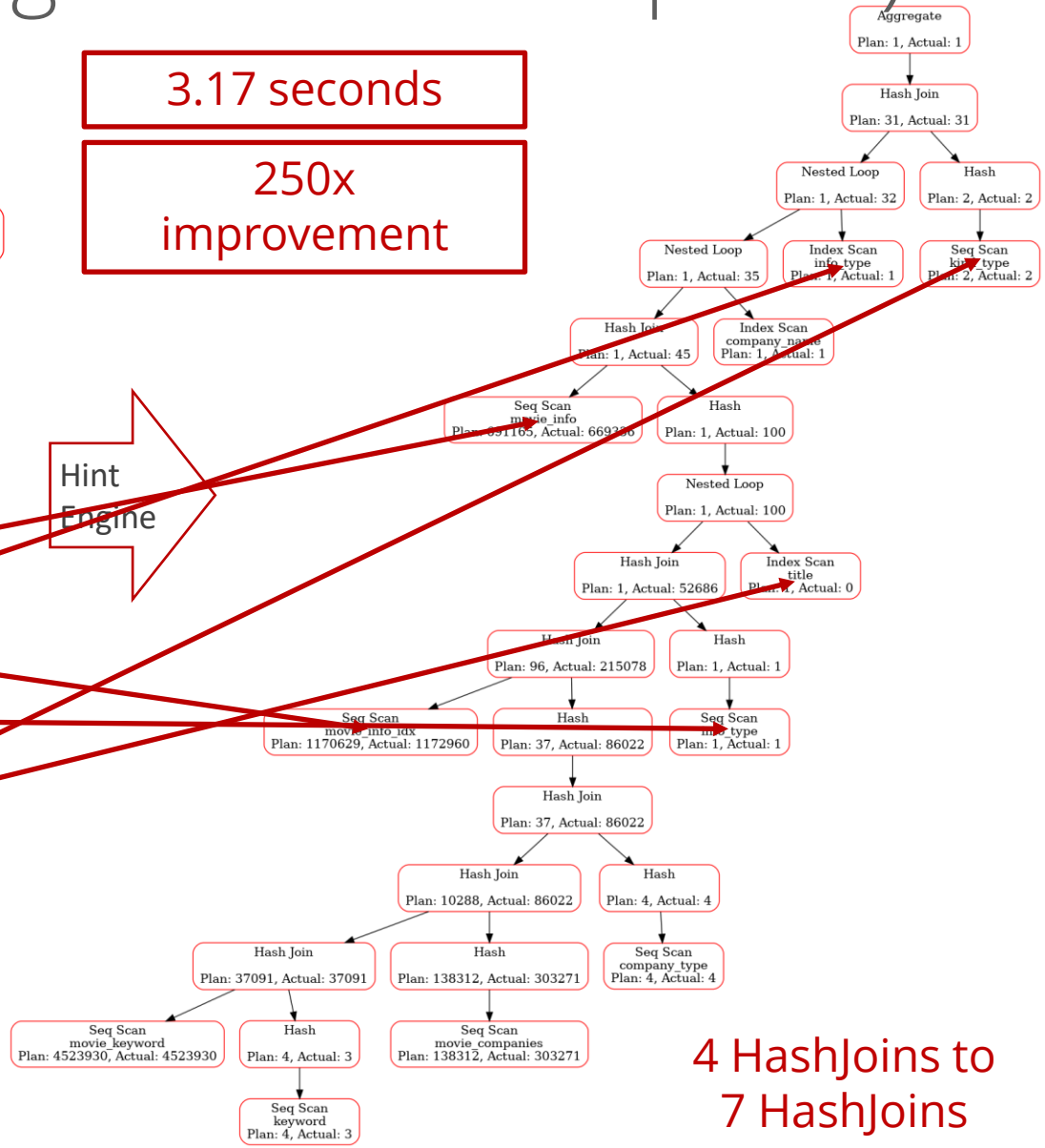
250x improvement

Nested Loop Join
Planned 10000 rows
Actual 600 million rows

Nested Loop Join
Planned 50000 rows
Actual 2 billion rows



Hint Engine



4 HashJoins to 7 HashJoins

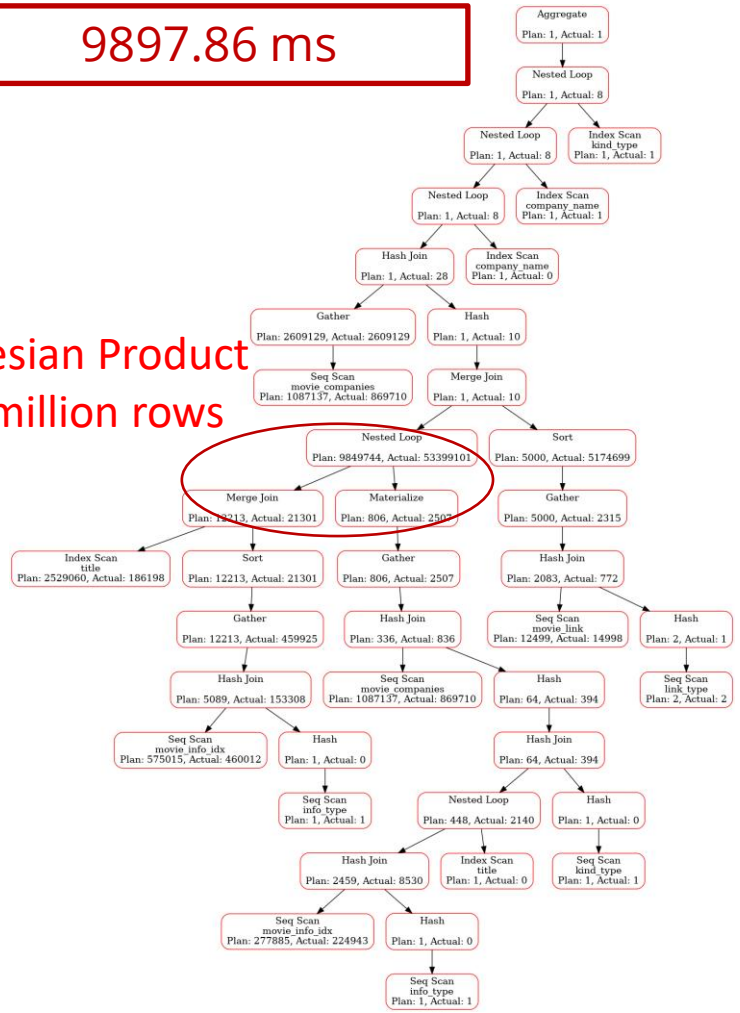
ChatGPT prompt: <plan json> Visualize this plan as a tree, only keep "Node Type", "Relation Name", "Plan Rows", "Actual Rows" for each node. Make the visualization a png, similar to the style of this <screenshot of previous the slide>

Plan Comparison – 33a (multicore hint improve)

Multicore-raw

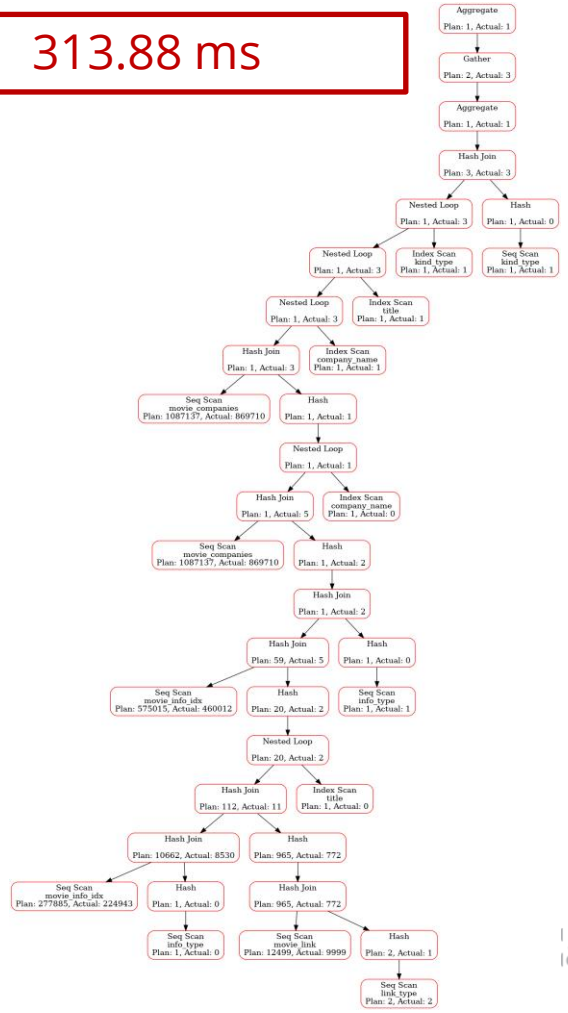
9897.86 ms

Cartesian Product
500 million rows



Multicore-hinted

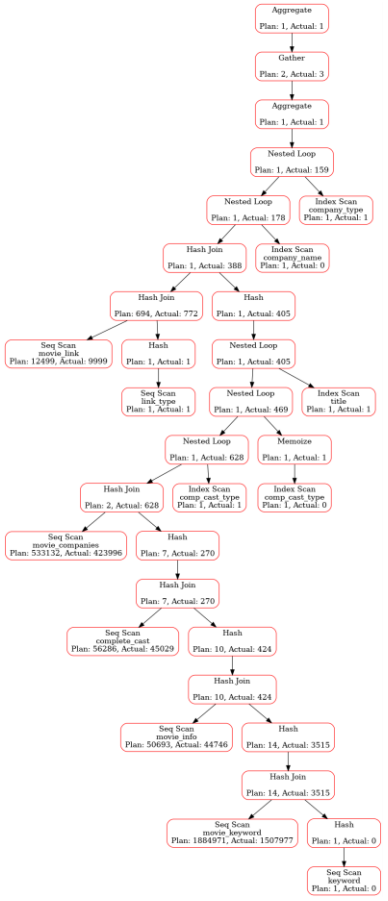
313.88 ms



Plan Comparison – 27a (multicore - card only worse)

Multicore-raw

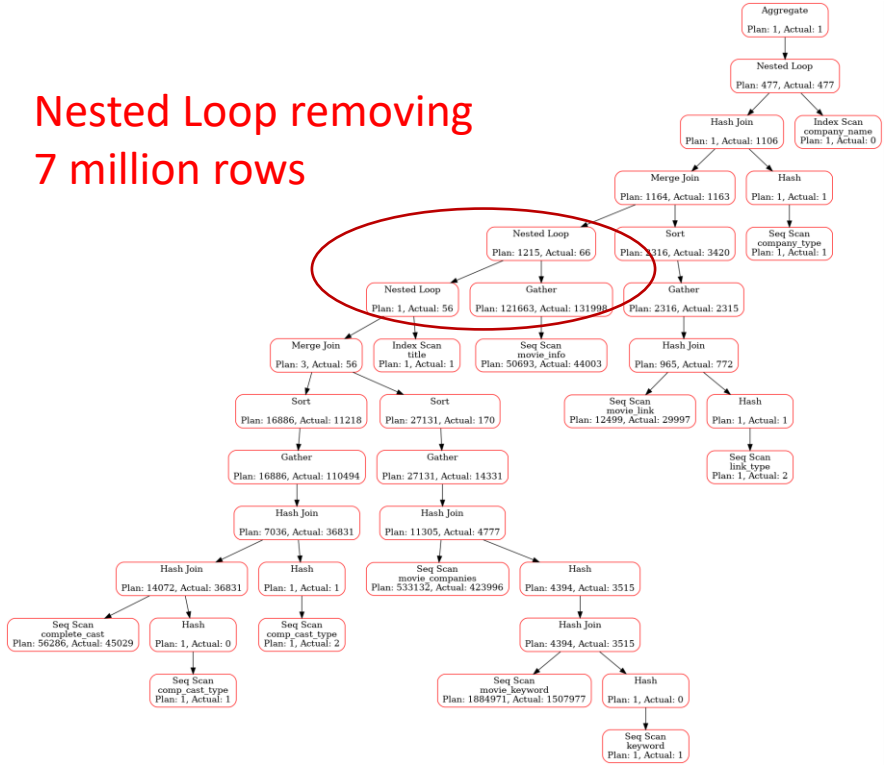
967.5413333 ms



Multicore-hinted-cardonly

34410.52333 ms

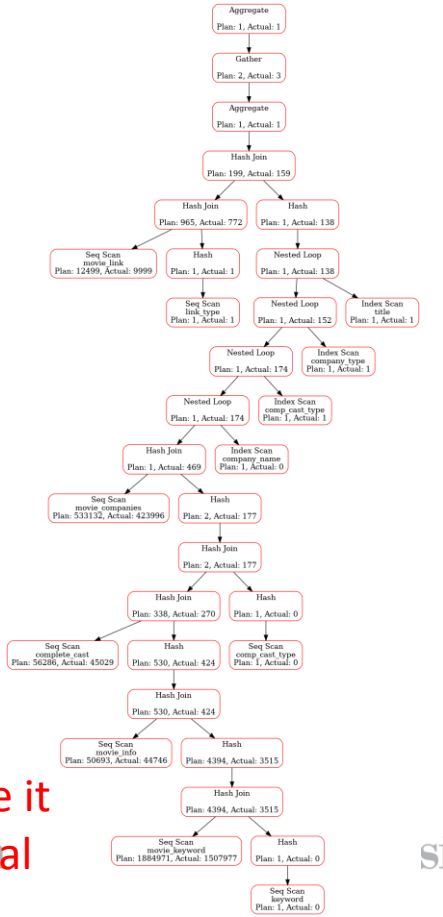
Nested Loop removing 7 million rows



Enforcing hash join made it switch back to the original plan

Multicore-hinted

930.0106667 ms

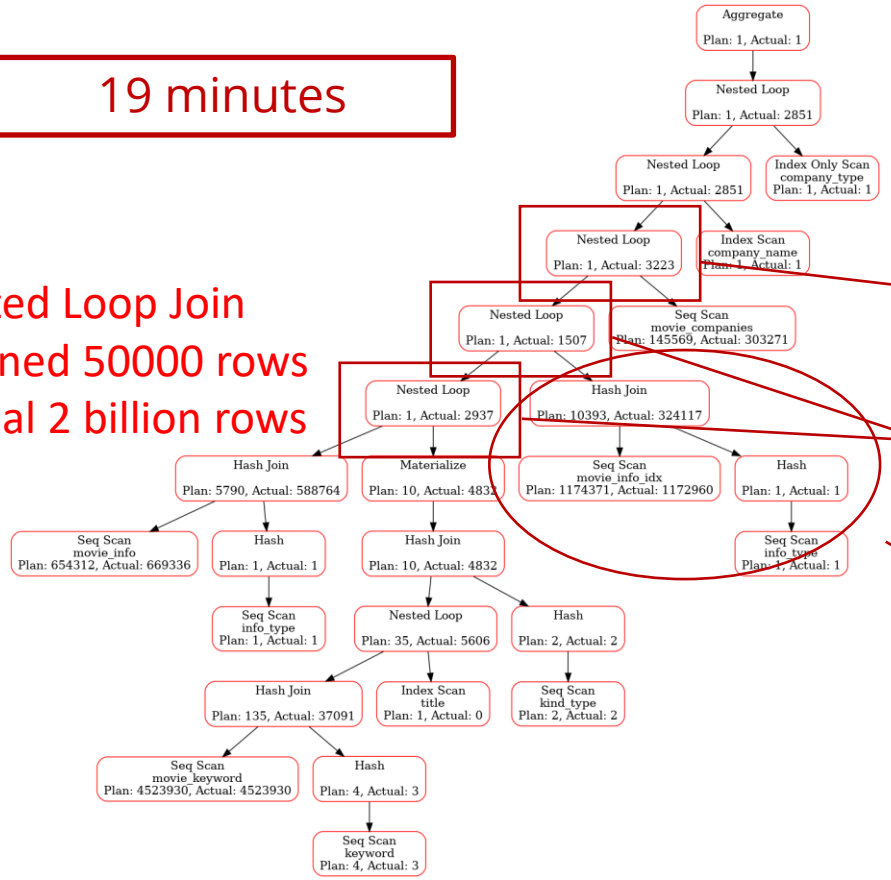


Plan Comparison – 22a (single vs multi)

Singlecore-raw

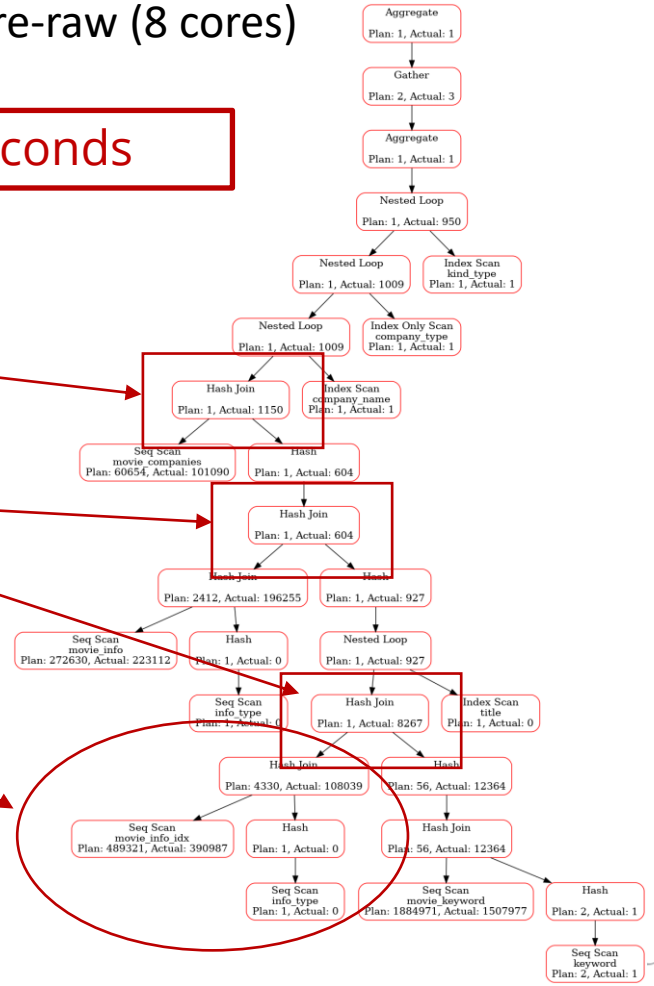
19 minutes

Nested Loop Join
Planned 50000 rows
Actual 2 billion rows



Multicore-raw (8 cores)

1.2 seconds



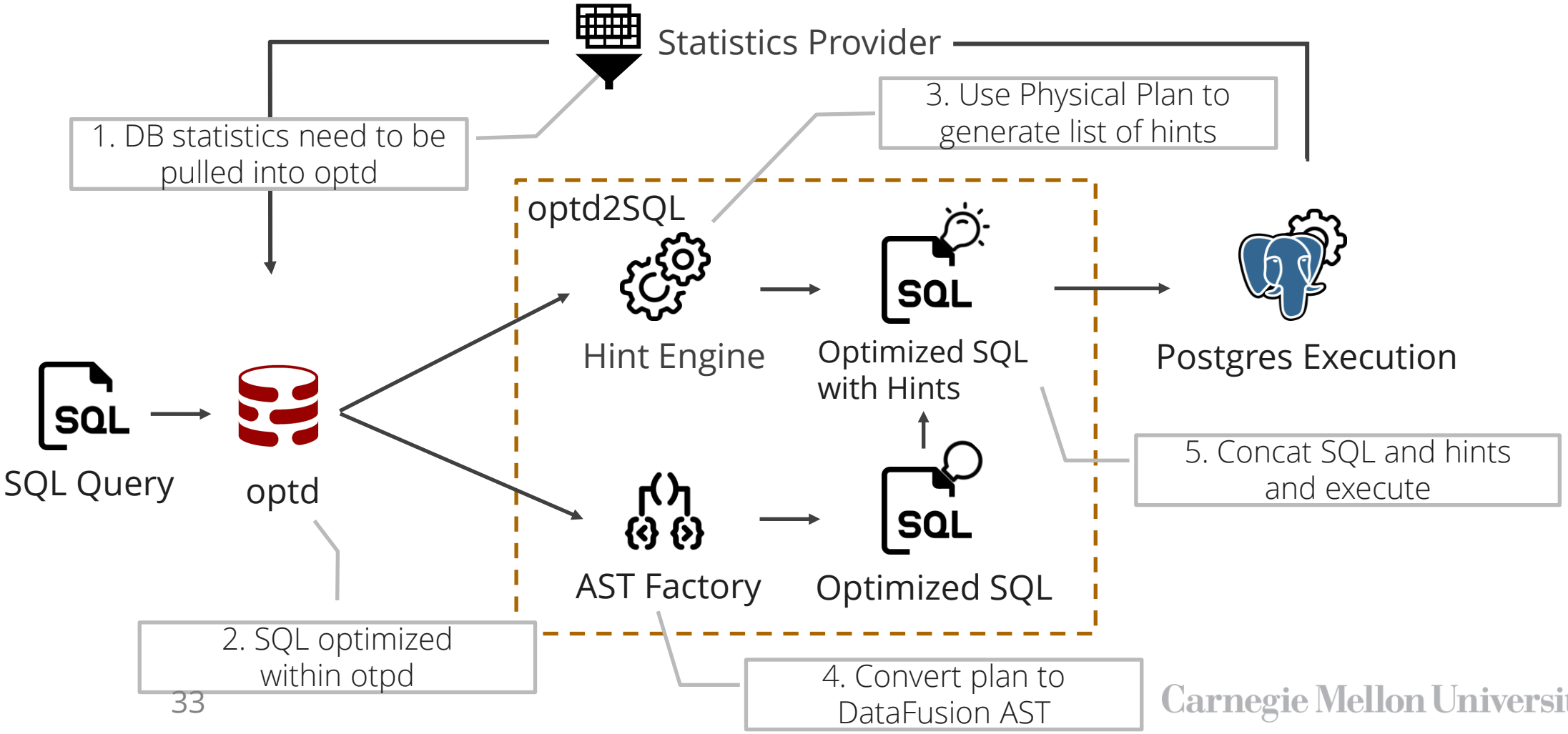
Observations

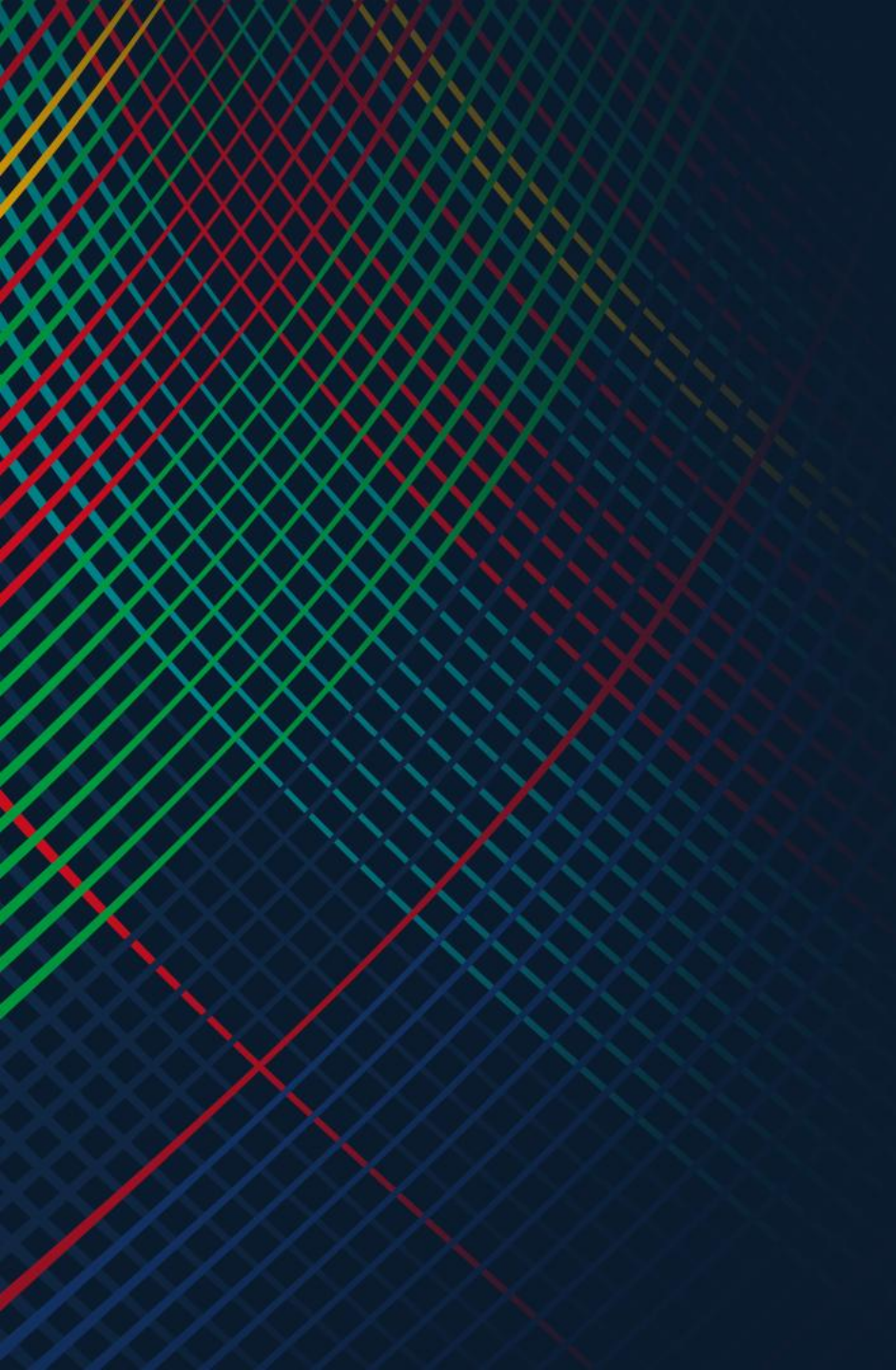
- Injecting correct join cardinality does show significant impact → Potential benefit of using an external optimizer to better estimate cardinality
- Injected cardinalities doesn't cover all join orderings. Injected cardinalities are sometimes ineffective. Consider:
  - Truth: plan A better than plan B,
  - Cost: estimated A < estimated B < truth A < truth B
  - Original Plan: chooses plan A :)
  - Cardinality Injection: gives truth for plan A, it is worse than you thought!
  - New plan: chooses plan B over plan A :(
- Enforcing nested loop join to hash join on top of cardinality injection brings some improvement and reduces the number of timeout queries
- No real benefit on multi-core
- Related, performance doesn’t scale linearly from single to multi core. Postgres sometimes chooses more reasonable plans under multicore setting

Future work

- Use hints to inject information from an entirely different optimizer into Postgres
  - Can theoretically dictate join order and cardinality of those joins
  - Can specific which indexes to use
- Support Postgres 16's hint table – change insert function
- Explore why Postgres make such different optimization decisions on single vs multi-core environments

Future Work: Next Semester Architecture





Questions?