The Glorious Dead

Building a query optimizer that **learns and evolves**

Yuchen Liang

CMU-DB IAP Visit Day, Sep 2026

Carnegie Mellon University
yuchenl3@cs.cmu.edu

Overview

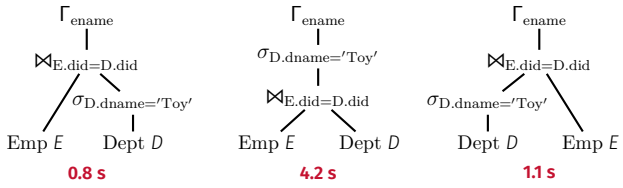
- → Traditional optimizers **learn little** from past planning and execution.
- → Adaptive techniques refine estimates, reuse subplans, or steer plan search.
- → But these techniques operate within a **fixed action space**, without changing optimizer implementations.
- → We are building ***optd***, leveraging query feedback and LLM agents to **improve** existing components, **implement** known techniques, and **synthesize** new capabilities.
- → Query optimizers could **evolve with their workloads**, not repeat the same mistakes.

Query Optimization

Query languages like SQL are **declarative**.

→ The user tells the DBMS what result they want and (usually) not how to compute it.

```
-- unique employee names
-- in the Toy department
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```



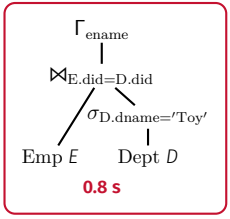
For a given query, the DBMS attempts to find a **correct** execution plan with the **least cost** (exec time, memory usage, etc.).

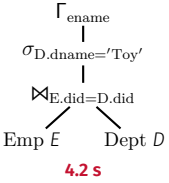
Query Optimization

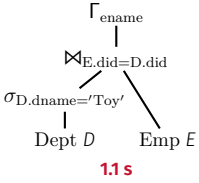
Query languages like SQL are **declarative**.

→ The user tells the DBMS what result they want and (usually) not how to compute it.

```
-- unique employee names
-- in the Toy department
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

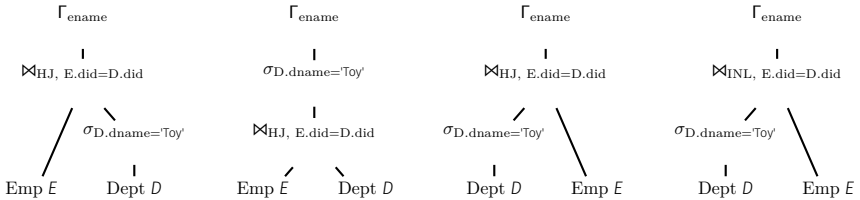






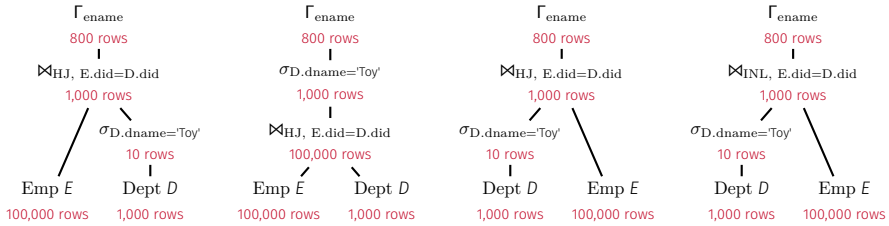
For a given query, the DBMS attempts to find a **correct** execution plan with the **least cost** (exec time, memory usage, etc.).

Query Optimization: Exploring and Costing Plans



→ **Simplify** and **Explore** alternatives, including physical implementations.

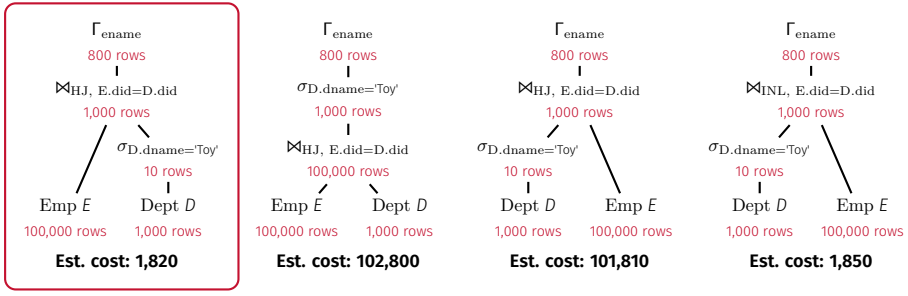
Query Optimization: Exploring and Costing Plans



→ **Simplify** and **Explore** alternatives, including physical implementations.

→ **Estimate cardinalities**: rows produced by each operator.

Query Optimization: Exploring and Costing Plans



→ **Simplify** and **Explore** alternatives, including physical implementations.

→ **Estimate cardinalities**: rows produced by each operator.

→ **Estimate cost** for each alternative and choose a minimum-cost plan.

Optimizer Problems

Reason #1: Inaccurate cost estimates based on a **static** summarization of its contents of the database and its operating environment:

Optimizer Problems

Reason #1: Inaccurate cost estimates based on a **static** summarization of its contents of the database and its operating environment:

Simplifying assumptions

```
SELECT * FROM Cars
WHERE Make = 'Honda'
  AND Model = 'Accord';
```

Attributes are not independent:
only Honda makes Accord.

Guy Lohman, SIGMOD Blog, 2014.

Optimizer Problems

Reason #1: Inaccurate cost estimates based on a **static** summarization of its contents of the database and its operating environment:

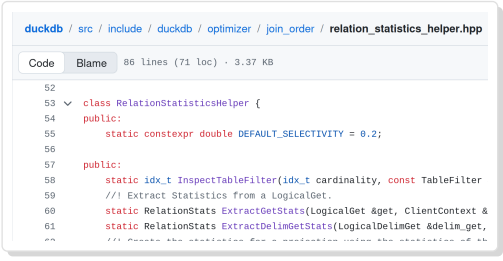
Simplifying assumptions

```
SELECT * FROM Cars
WHERE Make = 'Honda'
  AND Model = 'Accord';
```

Attributes are not independent: only Honda makes Accord.

Guy Lohman, SIGMOD Blog, 2014.

Selectivity fallback



Modern data stacks make this worse: missing catalog statistics force the optimizer to fall back on magic numbers.

Optimizer Problems

Reason #2: Missed optimization opportunities.

Optimizer Problems

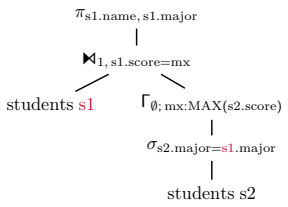
Reason #2: Missed optimization opportunities.

```
-- Find the highest-scoring students in each major. (20,000 students, 80 majors.)
SELECT name, major FROM students AS s1
WHERE score = (SELECT MAX(s2.score) FROM students AS s2
               WHERE s2.major = s1.major);
```

Optimizer Problems

Reason #2: Missed optimization opportunities.

```
-- Find the highest-scoring students in each major. (20,000 students, 80 majors.)
SELECT name, major FROM students AS s1
WHERE score = (SELECT MAX(s2.score) FROM students AS s2
               WHERE s2.major = s1.major);
```

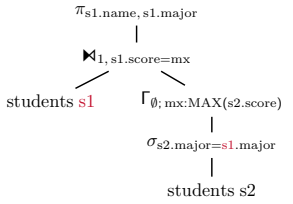


MAX once per student

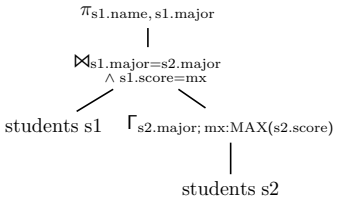
Optimizer Problems

Reason #2: Missed optimization opportunities.

```
-- Find the highest-scoring students in each major. (20,000 students, 80 majors.)
SELECT name, major FROM students AS s1
WHERE score = (SELECT MAX(s2.score) FROM students AS s2
               WHERE s2.major = s1.major);
```



MAX once per student



MAX once per major

Optimizer Problems

Reason

```
-- Find
SELECT
WHERE s
```

...iors.)

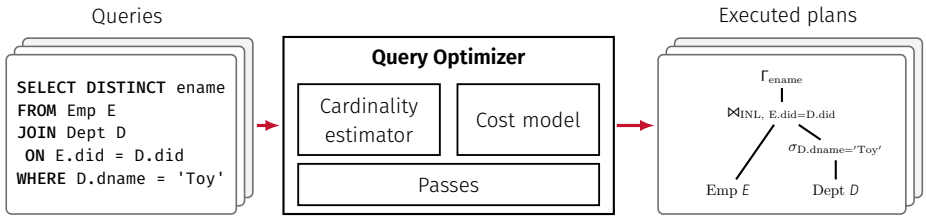
...re)

	Method	UDFs 1	5	6	7	12	13	15	17	18	20a_q1	20a_q2	20b_q1	20b_q2
SQL Server	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	Batched	✗	✓	(✓)	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
Oracle	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	Batched	✗	✗	(✓)	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
DuckDB	Inlined	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Batched	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PostgreSQL	Inlined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Batched	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 1: Subquery Decorrelation – Whether a given UDF's subqueries could be decorrelated by a DBMS after inlining or batching. Symbol (✓) indicates that some, but not all subqueries could be decorrelated.

Optimizer Problems

Optimizers learn little from past planning.



New queries. Same assumptions. **Same mistakes.**

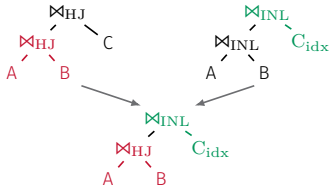
Prior Work: Query Feedback

LEO: Repair bad cardinality estimates using observed row counts.

Estimated 100 rows → Observed 10,000 rows

$$\text{new\_adj} = \text{old\_adj} \times \frac{10{,}000}{100}$$

Plan Stitch: Combine efficient subplans from prior executions.



Limitations: Cannot generalize beyond observed queries and subplans.

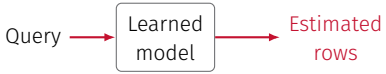
Prior Work: Learned Estimators and Optimizers

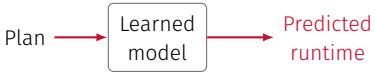
Learned CE: Predict cardinalities from data or query observations.

MSCN, DeepDB, NeuroCard.

Learned Optimizer: Learn plan selection or search-space steering from runtimes.

Neo, Bao, AutoSteer.

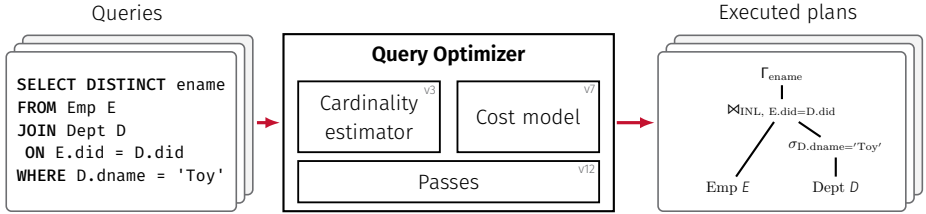




Limitations: Learning stays within a fixed model and decision space; adapting to change requires retraining and engineering.

Agent-in-the-Loop Optimization

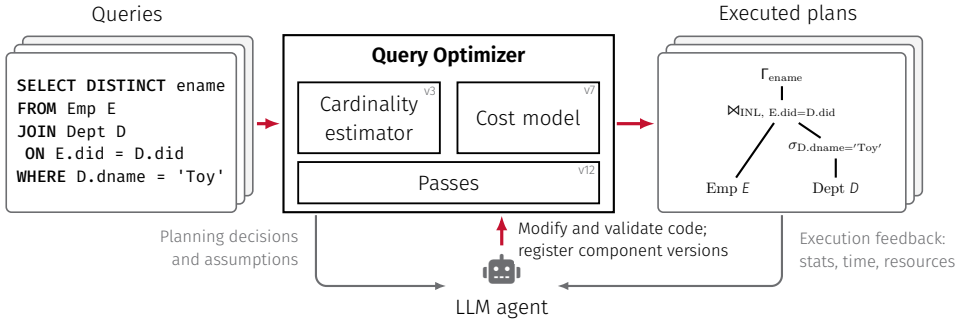
LLM agents offer a new opportunity to evolve optimizer component implementations.



Improve implementations while **preserving query semantics**.

Agent-in-the-Loop Optimization

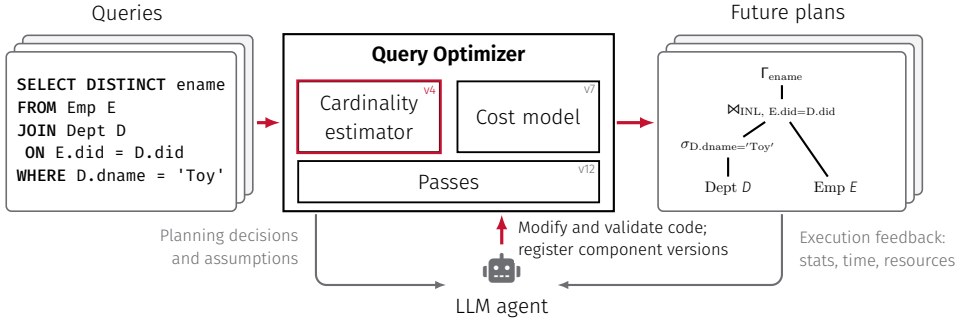
LLM agents offer a new opportunity to evolve optimizer component implementations.



Improve implementations while **preserving query semantics**.

Agent-in-the-Loop Optimization

LLM agents offer a new opportunity to evolve optimizer component implementations.



Improve implementations while **preserving query semantics**.

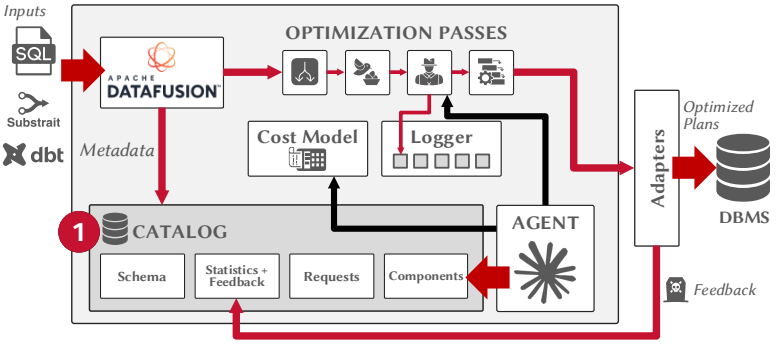
The optd Project

New standalone **agentic** query optimizer service written in Rust.

Learn from the corpses of past queries to improve planning strategies and optimizer implementations.

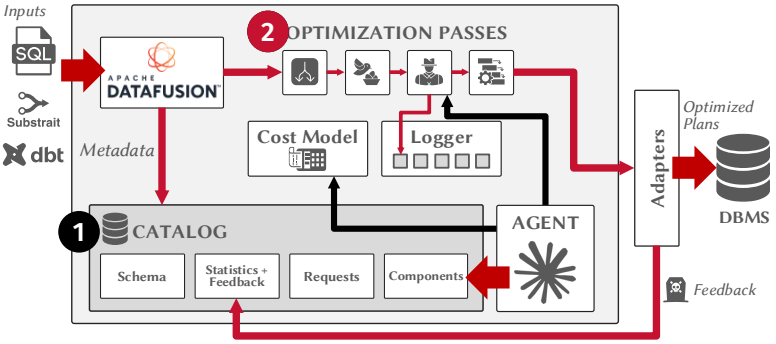
Conjecture: The optimizer will be the linchpin for agentic coding in future database management systems engineering.

Architecture



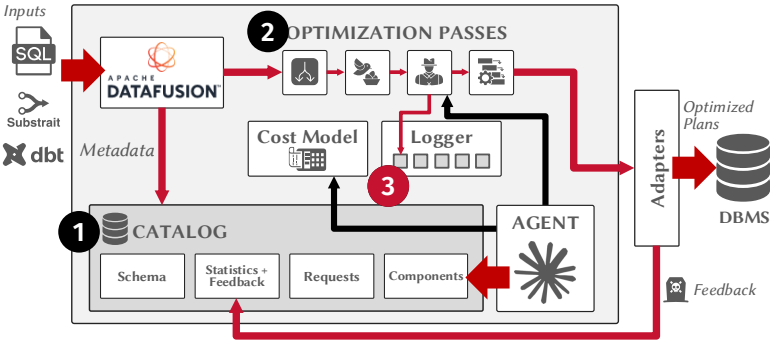
Catalog: Version metadata, statistics, history, and component implementations.

Architecture



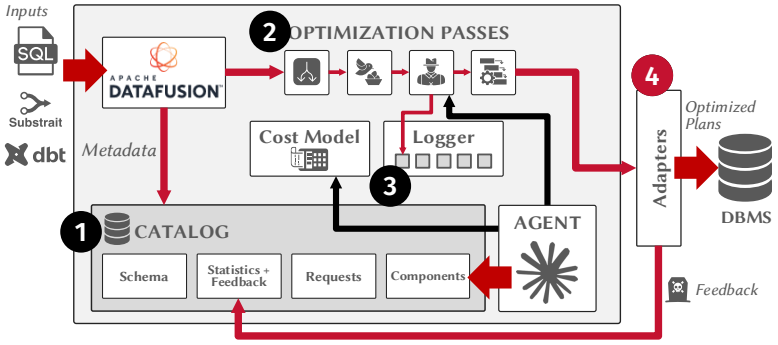
Passes: Transform plans through heuristics and searches guided by the cost model.

Architecture



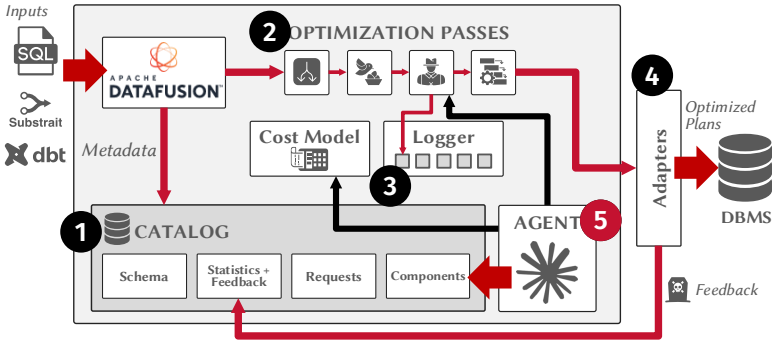
Logger: Record planning decisions and their provenance.

Architecture



Adapters: Translate plans for DBMSs and collect execution feedback.

Architecture



LLM Agent: Use the feedback to improve the service and register new components.

Different Workloads, Different Optimizers

Specialize the optimizer to the workload.

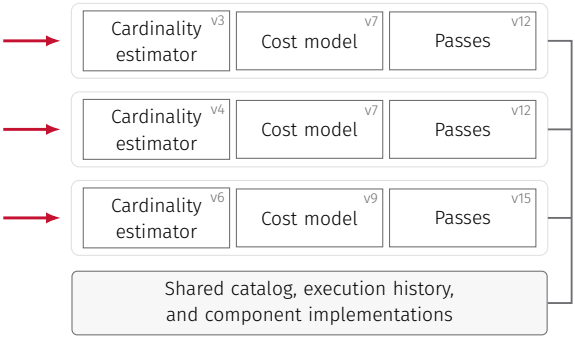
Workloads

String/regex-heavy
Predicate cost and selectivity

Repeated parameterized queries
Parameter-aware plan reuse

Memory-constrained workloads
Avoid spills and large intermediates

Workload-specific optimizer configurations



Three Tiers of Agent-Driven Adaptation

1. **Improve** existing implementations.
   - → Reduce pass overhead or refine an estimator.
   - → Demonstrated in our **two case studies**.
2. **Implement** known techniques.
   - → Adapt published papers to the optimizer and engine.
   - → Implemented DP-based join ordering and unnesting with agents and manually verified.
3. **Synthesize** new capabilities.
   - → Develop new passes, operators, and optimization methods.

Objectives

Performance: Do the same optimization work faster.

→ Reduce optimizer time and resource consumption.

→ **Case Study 1:** Tune DPHyp join-ordering performance.

Efficacy: Find better plans within the same planning budget.

→ Compare under a fixed workload and environment.

→ **Case Study 2:** Improve cardinality estimates.

Case Study 1: Improving Pass Performance

DPHyp: Enumerate and compare join orders to choose a plan.

Objective	Reduce join-ordering time without changing the algorithm.
Starting point	A manually validated, agent-written DPHyp pass.
Agent actions	Edit, run, and profile pass code.
Feedback	Planning times and profiles; JOB 15c or all 113 queries.
Evaluation	Median of 15 full-workload planning-time measurements, normalized to the initial implementation.

Stop: Reach 1 ms per query, or three consecutive attempts fail to improve median planning time by more than 3%.

Measure **join-ordering time**, not query execution time.

System: DataFusion v54; GPT-5.6 Sol; 12-core Apple M2 Max; 32 GB unified memory.

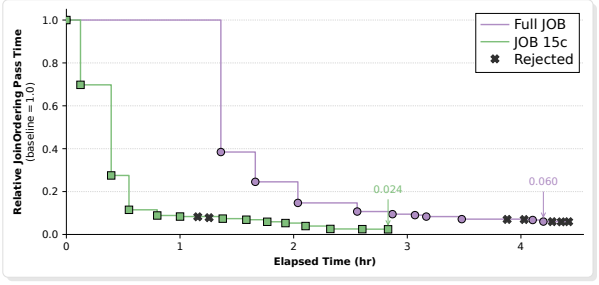
Case Study 1: Improving Pass Performance

The same search algorithm runs up to **41.8× faster** by reducing implementation overhead.

Cache candidate state; reduce allocation, hashing, and graph scans.

→ **15c feedback:**
41.8× after 2.8 hours.

→ **Full-JOB feedback:**
16.6× after 4.2 hours.



Full-JOB planning time, normalized to the baseline.
Lower is better; crosses mark rejected candidates.

System: DataFusion v54; GPT-5.6 Sol; 12-core Apple M2 Max; 32 GB unified memory.

Case Study 2: Improving Cardinality Estimator Efficacy

Objective	Improve cardinality estimates for JOB subplans.
Starting point	A simple estimator with coarse statistics and fixed assumptions.
Agent actions	Edit estimator code and collect richer statistics.
Feedback	Execution cardinalities and catalog information.
Evaluation	Q-error against held-out true cardinalities; compare initial, final, and PostgreSQL after `ANALYZE`.

System: DataFusion v54; GPT-5.6 Sol; 12-core Apple M2 Max; 32 GB unified memory.

Case Study 2: Improving Cardinality Estimator Efficacy

Objective	Improve cardinality estimates for JOB subplans.
Starting point	A simple estimator with coarse statistics and fixed assumptions.
Agent actions	Edit estimator code and collect richer statistics.
Feedback	Execution cardinalities and catalog information.
Evaluation	Q-error against held-out true cardinalities; compare initial, final, and PostgreSQL after `ANALYZE`.

Q-error: $q = \max\left(\frac{\text{estimated}}{\text{actual}}, \frac{\text{actual}}{\text{estimated}}\right)$.

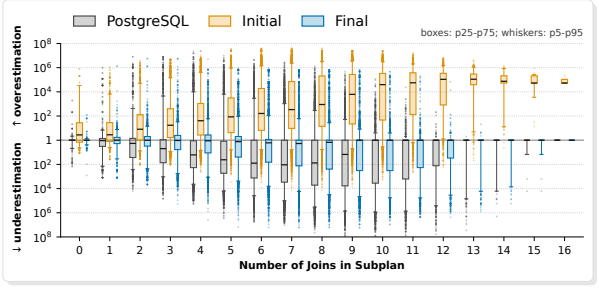
Example: estimated 100 rows, actual 1,000 rows $\Rightarrow q = 10$.
An exact estimate gives $q = 1$; lower is better.

System: DataFusion v54; GPT-5.6 Sol; 12-core Apple M2 Max; 32 GB unified memory.

Case Study 2: Improving Cardinality Estimator Efficacy

Richer statistics and revised estimation logic reduce median q-error by **57**×.

Model filtered distributions, nulls, domain overlap, and uniqueness.



Subplan q-error grouped by number of joins.
Initial, Final, and PostgreSQL; lower is better.

System: DataFusion v54; GPT-5.6 Sol; 12-core Apple M2 Max; 32 GB unified memory.

Future Work

Verify semantics

SMT solvers and proof assistants

Benchmark

Reject regressions; roll back changes

Bound adaptation

Cap agent time and resources

Synthesize

New operators, rules, and methods

Conclusion

- → Traditional optimizers **learn little** from past planning and execution.
- → Adaptive techniques refine estimates, reuse subplans, or steer plan search.
- → But these techniques operate within a **fixed action space**, without changing optimizer implementations.
- → We are building ***optd***, leveraging query feedback and LLM agents to **improve** existing components, **implement** known techniques, and **synthesize** new capabilities.
- → Query optimizers could **evolve with their workloads**, not repeat the same mistakes.

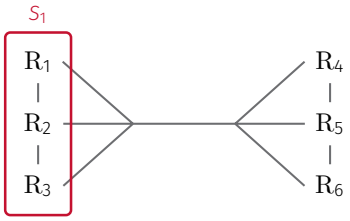
Thanks!

Backup: DPHyp Join Ordering

Use dynamic programming to search a join hypergraph.

- → Grow two disjoint connected subgraphs.
- → A hyperedge may require several relations on each side.
- → Emit a pair only when its join predicate is applicable.

Implementation: Use bitsets to make set operations fast.



Select connected subgraph $S_1 = \{R_1, R_2, R_3\}$.
Grow a complement on the right.

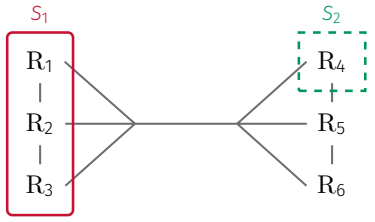
Adapted from Moerkotte and Neumann, SIGMOD 2008, Figures 2–3 (steps 20–23).

Backup: DPHyp Join Ordering

Use dynamic programming to search a join hypergraph.

- → Grow two disjoint connected subgraphs.
- → A hyperedge may require several relations on each side.
- → Emit a pair only when its join predicate is applicable.

Implementation: Use bitsets to make set operations fast.



$S_2 = \{R_4\}$: connected, but not joinable yet.
The hyperedge also requires R_5 and R_6.

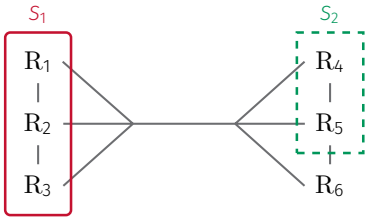
Adapted from Moerkotte and Neumann, SIGMOD 2008, Figures 2–3 (steps 20–23).

Backup: DPHyp Join Ordering

Use dynamic programming to search a join hypergraph.

- → Grow two disjoint connected subgraphs.
- → A hyperedge may require several relations on each side.
- → Emit a pair only when its join predicate is applicable.

Implementation: Use bitsets to make set operations fast.



$S_2 = \{R_4, R_5\}$: still not joinable.
The hyperedge still requires R_6.

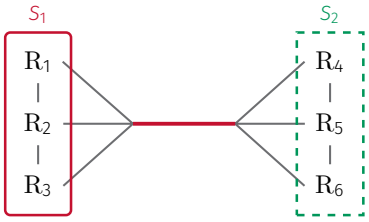
Adapted from Moerkotte and Neumann, SIGMOD 2008, Figures 2–3 (steps 20–23).

Backup: DPHyp Join Ordering

Use dynamic programming to search a join hypergraph.

- → Grow two disjoint connected subgraphs.
- → A hyperedge may require several relations on each side.
- → Emit a pair only when its join predicate is applicable.

Implementation: Use bitsets to make set operations fast.



$S_2 = \{R_4, R_5, R_6\}$: emit the CSG-CMP pair.
Compare joins of the best plans for both sides.

Adapted from Moerkotte and Neumann, SIGMOD 2008, Figures 2–3 (steps 20–23).