The Glorious Dead: Making New Queries Run Faster on the Backs of Slower, Deceased Queries in the *optd* Service

by Yuchen Liang

Carnegie Mellon
Database Group

Overview: Query Optimization

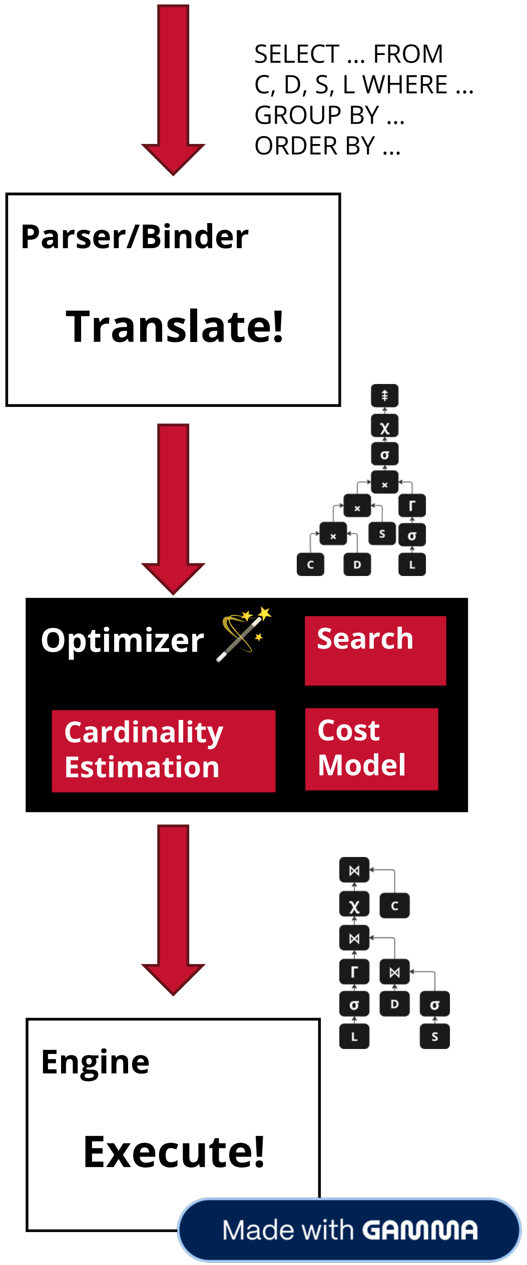
- **User: Write a query in a declarative language (e.g. SQL)**

  The dream: User only needs to specify what result they want: It is the optimizer's job to figure out how to compute the result.

- **Optimizer: Find a correct execution plan with the best cost.**

  Enumerate equivalent plans

  Compare equivalent plans using a cost model



SELECT ... FROM
C, D, S, L WHERE ...
GROUP BY ...
ORDER BY ...

Parser/Binder

Translate!

Optimizer

Search

Cardinality Estimation

Cost Model

Engine

Execute!

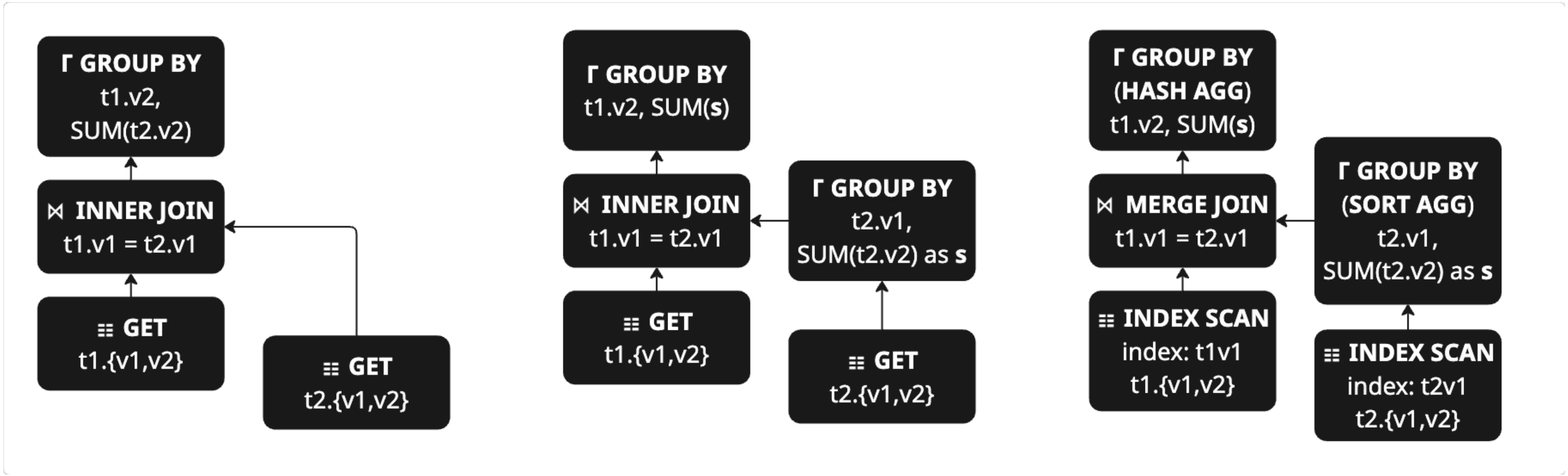
Enumerating Equivalent Plans

1 Logical Transformations

Pushdowns, join ordering, eager aggregation, unnesting, etc.

2 Physical Operator Selection

Take advantage of physical properties, such as tuple ordering, grouping, buffer/stream, etc.



Γ GROUP BY t1.v2, SUM(t2.v2)

⋈ INNER JOIN t1.v1 = t2.v1

GET t1.{v1,v2}

GET t2.{v1,v2}

Γ GROUP BY t1.v2, SUM(**s**)

⋈ INNER JOIN t1.v1 = t2.v1

Γ GROUP BY t2.v1, SUM(t2.v2) as **s**

GET t1.{v1,v2}

GET t2.{v1,v2}

Γ GROUP BY (HASH AGG) t1.v2, SUM(**s**)

⋈ MERGE JOIN t1.v1 = t2.v1

Γ GROUP BY (SORT AGG) t2.v1, SUM(t2.v2) as **s**

INDEX SCAN index: t1v1 t1.{v1,v2}

INDEX SCAN index: t2v1 t2.{v1,v2}

After Eager Aggregation

After Operator Selection

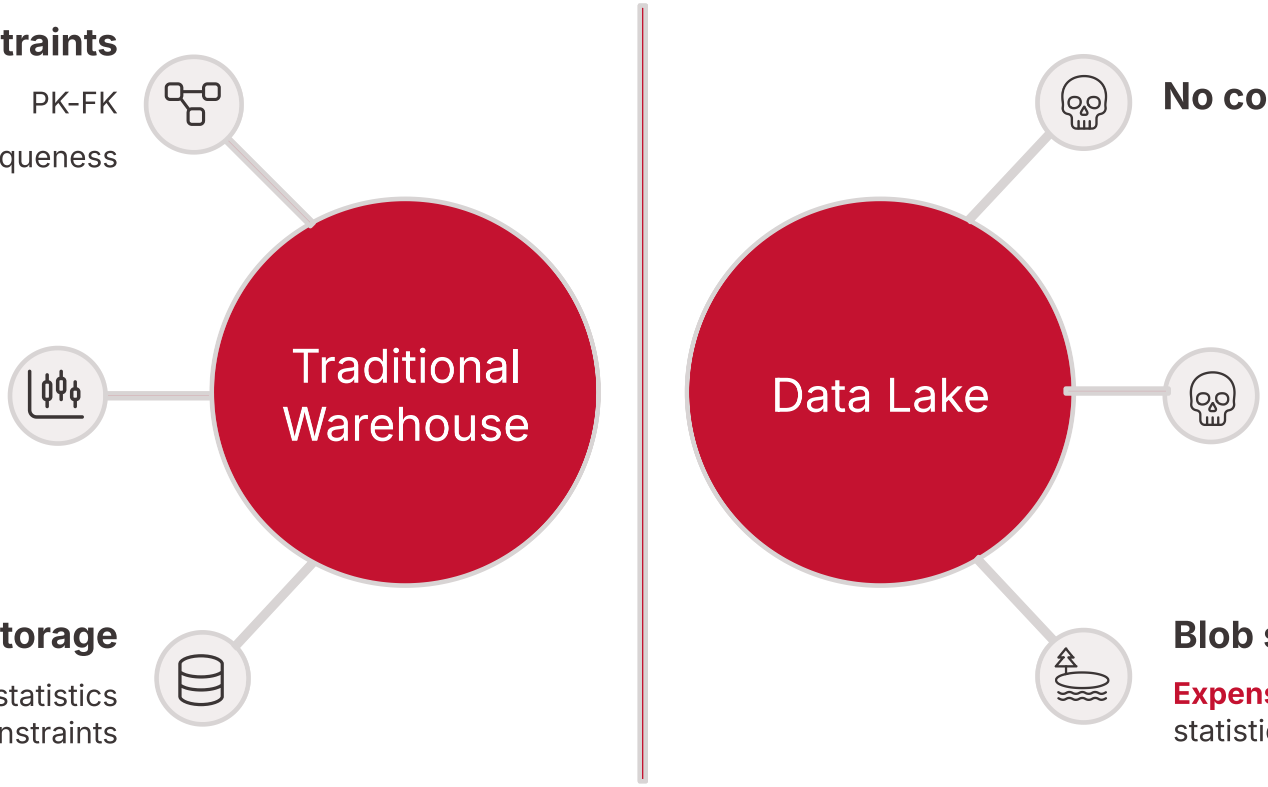
Comparing Equivalent Plans

- **Accurate cost depends on accurate cardinality estimates.**
- **Accurate cardinality estimates depends on rich statistics.**

  Using histograms, sketches, samples to estimate filter selectivity, join size, and number of distinct values.
- **Still an approximation based on many assumptions.**

  Uniformity, independence, containment, and many magic numbers.

Data Lake Challenges



Traditional Warehouse

Rich constraints

PK-FK

Uniqueness

Rich statistics

Histogram

Sketches

Samples

Managed storage

Cheap to maintain statistics and constraints

Data Lake

No constraints

Almost no statistics

Zone maps only

Blob storage

Expensive to maintain statistics and constraints

With Extensibility comes Responsibility

- **High development cost 😢**

  Existing rule may not work with new operators.

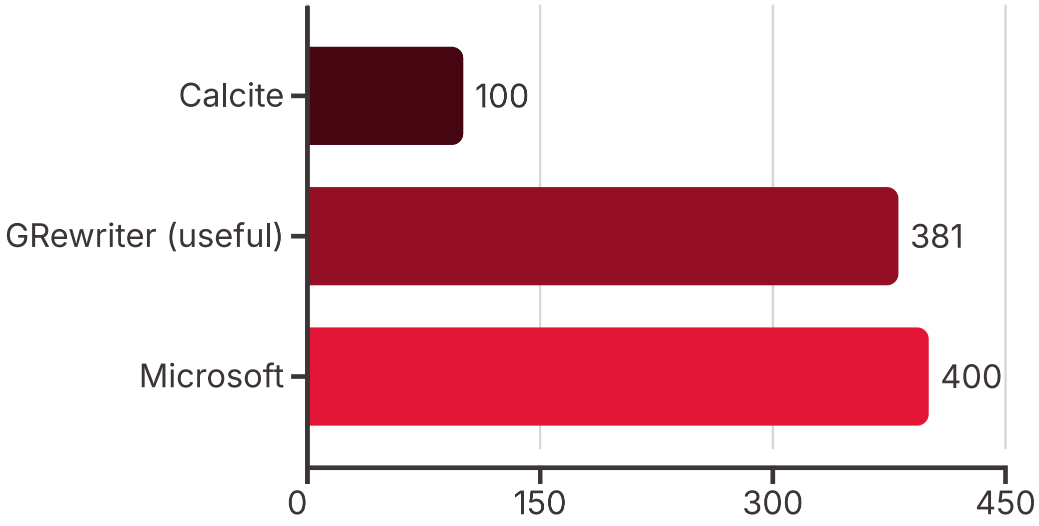
  Roll your own statistics and cost model (default is bad)

- **Hard to maintain correctness 😢**

  Too many rules, too many edge cases.

- **Difficult to understand what the system is capable of / not capable of 😢**

- **Optimization overhead ++ 😢**



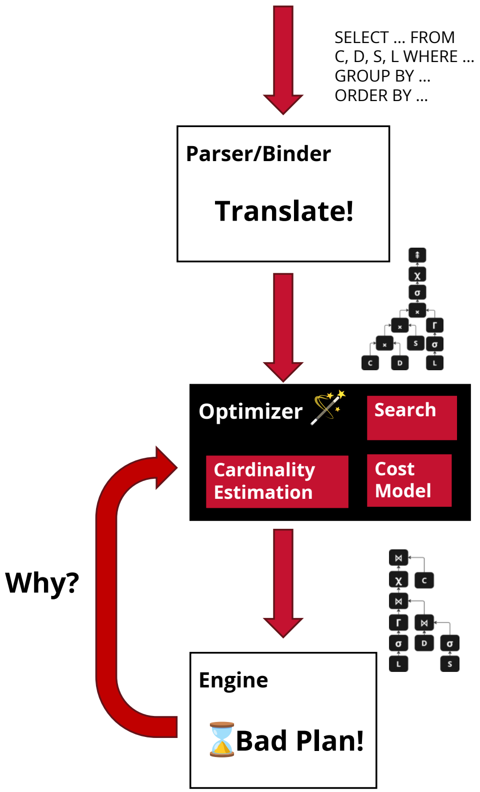
* Calcite rules on average have 200+ LoC.

* GRewriter generates 9.7 · 10^22 rule candidates in 8 days, 13,973 verified rules, 2,034 rules useful in at least one workload, 381 rules useful in all workload evaluated.

Got a bad plan, but why? 🪦

- **We overestimate/underestimate the number of distinct values at a join.**
- **Two predicates are highly dependent of each other.**
- **A key transformation was never triggered.**
- **A better alternative was never considered due to time constraint.**

Observation: So many possible source of errors, do we just put it in a graveyard, or break the abstraction, let end user specify how to query the data?



SELECT ... FROM C, D, S, L WHERE ... GROUP BY ... ORDER BY ...

Parser/Binder

Translate!

Optimizer 🪄

Search

Cardinality Estimation

Cost Model

Why?

Engine

⌛Bad Plan!

Explainability

Why/how did we pick this plan?

Interrogability

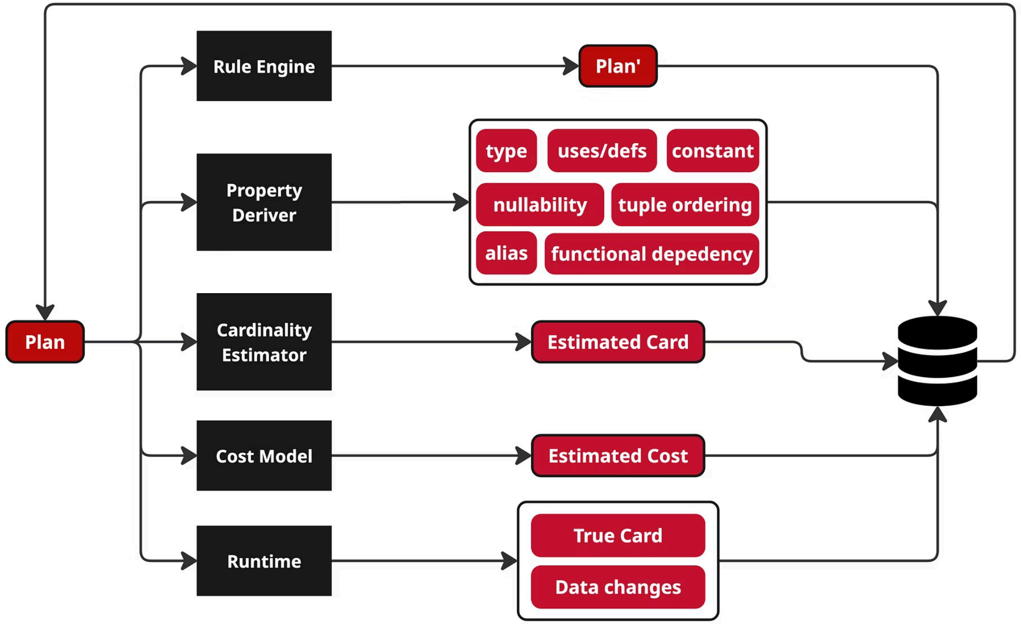
What if we changed X?

Adaptivity

Learning from optimizing and executing every query instance

The Responsible *optd* Service

- Persist optimization states across query instances
- Track decision points during optimization and perform what-if analysis
- Use data-flow based analyses to perform transformations
- Adaptively generate and maintain statistics, as well as learning column relationships.



The Breadcrumb Architecture

Logging all optimizer events in an internal database.

- **Internal**

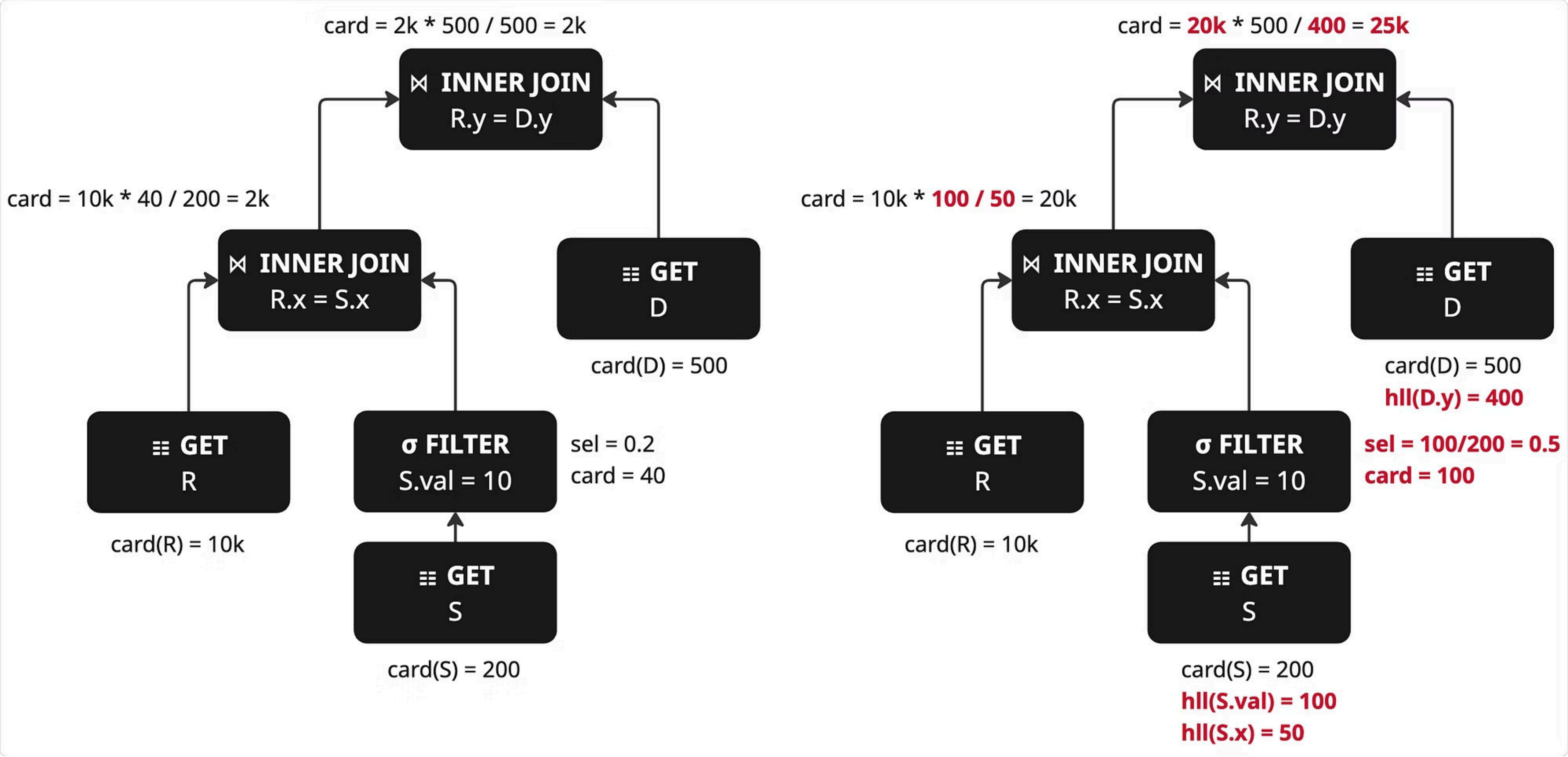
  Enumeration: creation of equivalent sub-plans, rules fired, required properties

  Comparison: estimated cost with statistics/assumption used, and the search space pruned

- **External**

  data changes, schema changes, feedback from execution-time statistics

What-if Analysis in Action



card = 2k * 500 / 500 = 2k

⋈ INNER JOIN R.y = D.y

card = 10k * 40 / 200 = 2k

⋈ INNER JOIN R.x = S.x

𝌆 GET D

card(D) = 500

𝌆 GET R

card(R) = 10k

σ FILTER S.val = 10

sel = 0.2
card = 40

𝌆 GET S

card(S) = 200

card = **20k** * 500 / **400** = **25k**

⋈ INNER JOIN R.y = D.y

card = 10k * **100 / 50** = 20k

⋈ INNER JOIN R.x = S.x

𝌆 GET D

card(D) = 500
hll(D.y) = 400

𝌆 GET R

card(R) = 10k

σ FILTER S.val = 10

sel = 100/200 = 0.5
card = 100

𝌆 GET S

card(S) = 200
hll(S.val) = 100
hll(S.x) = 50

$$\mathrm{sel}(A.x = c) = \mathrm{dom}(A.x) / \mathrm{card}(A) \mid \mathrm{sel}(p) = 0.2$$

$$\mathrm{card}(A \bowtie B \mid A.x = B.y) = \mathrm{card}(A) * \mathrm{card}(B) / \max(\mathrm{dom}(x), \mathrm{dom}(y))$$

Learning from the Graveyard

With the breadcrumbs, we can

- **Track which assumptions were wrong**
- **Re-optimize the query based on observation**
- **Build better estimation models over time**
- **Proactive recommendations (partitioning, materialized views)**

Development

Summer 2025 (Finished)

- □ Written in Rust 🦀
- □ Multi-threaded search with the `tokio` async runtime.
- □ Data-flow analysis and transformation framework.
- □ Apache Datafusion Adapter.

Fall 2025

- □ Breadcrumbs
- □ Catalog Service
- □ Statistics

Beyond

- □ Feedback from Execution
- □ Replay optimization trace
- □ ...

Key Differentiators

- **Explainability** and **interrogability** drives innovations in query optimizations and encourages adoption.

  Powered by breadcrumbs persistence

- **Dataflow-based** analysis and transformations

  Not just another Cascades implementation

- **Integrated Optimizer Service for Data Lakes**

Questions?