Query Processing and Optimization in SQL Database Systems

A Journey into the "Brain" of a Database

Paarth Batra [123cs0022]   Dhruv Singh [523cs0009]

CS302:Database Management Systems

November 5, 2025

Outline

What's the Problem?

- A single SQL query can be run in thousands of different ways.
- We write **Declarative** SQL: "WHAT data I want."
  - `SELECT name FROM students WHERE major = 'CS';`
- The database must create a **Procedural** Plan: "HOW to get the data."

Plan A (Smart):

- Use an index to find 'CS' majors.
- Get names for only those students.
- **Result: 0.5 seconds**

Plan B (Dumb):

- Get names of ALL students.
- Check major for ALL students.
- **Result: 50 minutes**

Key Takeaway

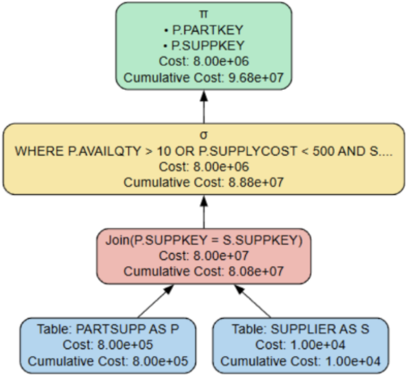
The "brain" that chooses Plan A is the **Query Optimizer**. Our project, is in practice, a simple version of this "brain".

The "Language" of the Optimizer: Relational Algebra

- To optimize a query, the database first translates it into a mathematical language.
- This language is called **Relational Algebra**.
- It's made of simple "verbs" or operators:
  - **Selection (σ):** Filters rows (the `WHERE` clause)
  - **Projection (π):** Selects columns (the `SELECT` clause)
  - **Join ($\bowtie$):** Combines tables (the `JOIN` clause)
  - **Relation (R):** A table (the `FROM` clause)
- These are the building blocks for *every* query plan.

The Relational Algebra Tree

- These operators are arranged into a "Query Tree".
- This tree is the "blueprint" for the query.
- Data flows from the leaves (Tables) up to the root (the final result).
- **The entire goal of query optimization is to find the cheapest tree shape.**



Relational Algebra Tree

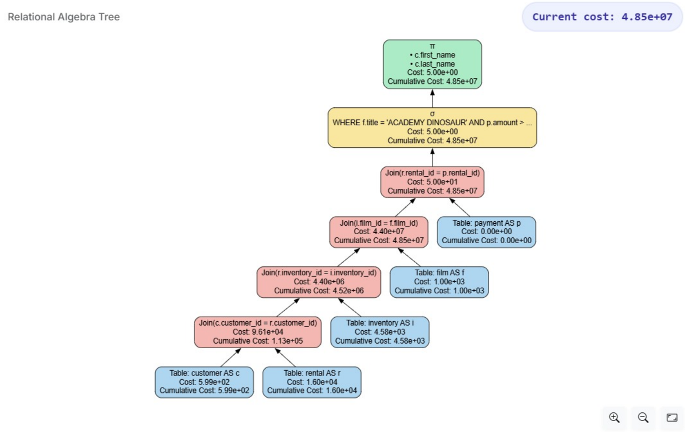
What We Built: QueryTune

- We built a visual, hands-on optimizer to show how these "tree transformations" work.
- It's a web application built in Python using **Flask**.
- **You enter an SQL query...**
- **It draws the "Dumb" Plan:** Parses your SQL into the *unoptimized* tree using a tool called `sqlglot`.
- **It runs "Smart" Rules:** It applies optimization rules to change the tree's shape.
- **It Shows the "Smart" Plan:** It draws the *new, optimized* tree, complete with a new, lower "cost".

Phase 1: The "Dumb" Plan

1. User enters an SQL query.
2. `parse.py` builds the first, unoptimized tree.
3. `fetch_table_statistics` connects to PostgreSQL to get real row counts.
4. `cost_estimator.py` looks at this tree and estimates a *cost*.

Initial Unoptimized Tree

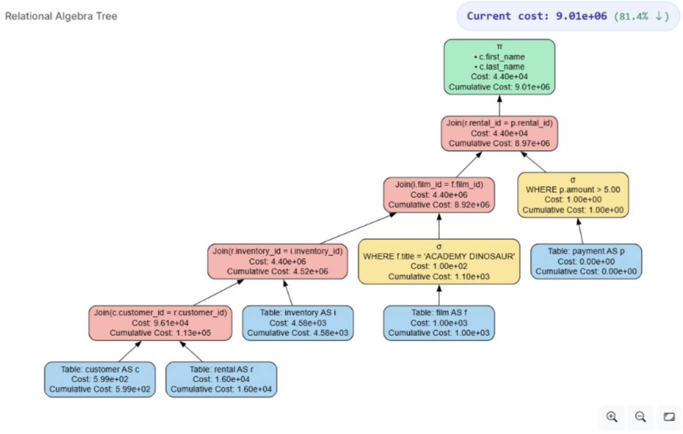


From our project report

Optimization 1: Predicate Pushdown (The "Filter Early" Rule)

- This is the most important heuristic (rule of thumb).
- **Problem:** The "dumb" plan joins two massive tables *first*, and *then* filters the result.
- **Analogy:** "Why buy 1,000 shirts and *then* check the size? Check the size *before* you buy!"
- **Solution:** "Push" the Selection (σ) *down* the tree, so it happens *before* the expensive Join ($\bowtie$).

Tree After Predicate Pushdown

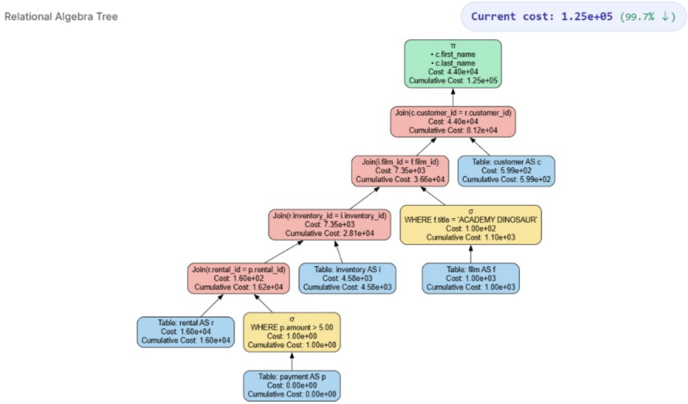


From our project report

Optimization 2: Join Reordering (The "Join Smart" Rule)

- The *order* in which you join tables matters... a lot.
- **Problem:**
  - (Small $\bowtie$ Medium) $\bowtie$ Large = **Fast**
  - (Large $\bowtie$ Large) $\bowtie$ Small = **Very Slow**
- **Solution (Cost-Based):**
  1. Our `join_optimization.py` module finds all tables.
  2. It calculates the cost of *every single possible join order* (using permutations).
  3. It picks the order with the lowest "cost" from our `cost_estimator`.

Result of Join Optimization



From our project report

Code Deep Dive: How We "Push" a Filter

Logic from `pred_pushdown.py`

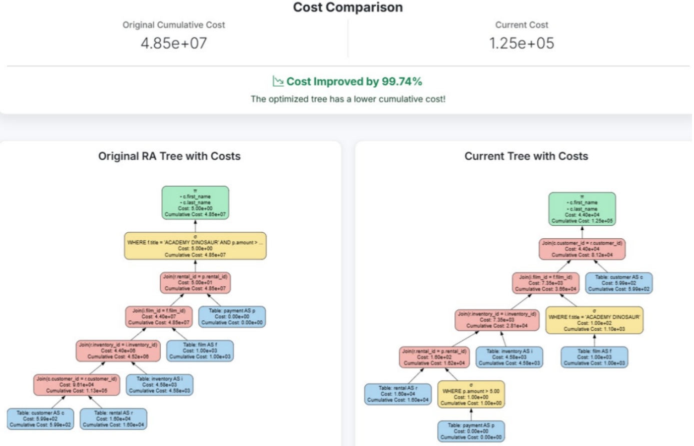
This is the plain-English logic for pushing a filter:

1. Find a Filter (σ) that is above a Join ($\bowtie$).
2. Check the filter's columns (e.g., `S.ACC_BAL > 1000`).
3. Ask: "Do all columns belong to the *left* side of the join?"
   - If YES, move the filter to the left child.
4. Ask: "Do all columns belong to the *right* side of the join?"
   - If YES, move the filter to the right child.
5. If it uses columns from *both*... you can't push it. Leave it alone.

Results: Did It Work?

- **Yes!** We tested our optimizer on 10 queries using a standard Pagila database.
- The Left tree is the "Original Cost" (the dumb plan).
- The Right tree is the "Optimized Cost" (our smart plan).
- For several queries, the optimized plan was over 99% cheaper.

Original vs Optimized Query Costs



From our project report

Limitations (What we couldn't do)

Our optimizer is a great educational tool, but it's not a commercial-grade product.

- **Simplified SQL:** We only support basic `SELECT-FROM-WHERE-JOIN`. We don't support `GROUP BY`, `OUTER JOIN`, etc.
- **Simple Cost Model:** We just count rows. Real optimizers (like in Postgres) model disk I/O, CPU, and memory. Our model can't tell the difference between reading from memory (fast) and disk (slow).
- **Simple Join Strategy:** We only check "left-deep" trees. Real optimizers also check "bushy" trees (e.g., $(A \bowtie B) \bowtie (C \bowtie D)$), which can be faster.

Conclusion & Future Work

Conclusion

We proved that even *basic* optimization rules, when applied systematically, can lead to massive performance gains (**around 85% !**). Further improvements (mentioned below) may lead to even greater performance gains.

Future Work

- **A Smarter Cost Model:** Add I/O and CPU costs.
- **Smarter Join Search:** Use Dynamic Programming (the textbook method) instead of brute-force.
- **The New Frontier:** The hottest research today uses **Machine Learning** to *learn* query costs from experience, rather than *estimating* them (e.g., a project called "Neo").

Questions?

Project Repository:
https://github.com/hydro-7/Query-Optimizer-DBMS