Query Optimization

Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

What Can We Optimize?

Operator graph: what operators do we run, and in what order?

Operator implementation: for operators with several impls (e.g. join), which one to use?

Access paths: how to read each table?

» Index scan, table scan, C-store projections, …

Typical Challenge

There is an exponentially large set of possible query plans

Access paths for table 1 × Access paths for table 2 × Algorithms for join 1 × Algorithms for join 2 × …

Result: we'll need techniques to prune the search space and complexity involved

Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

What is a Rule?

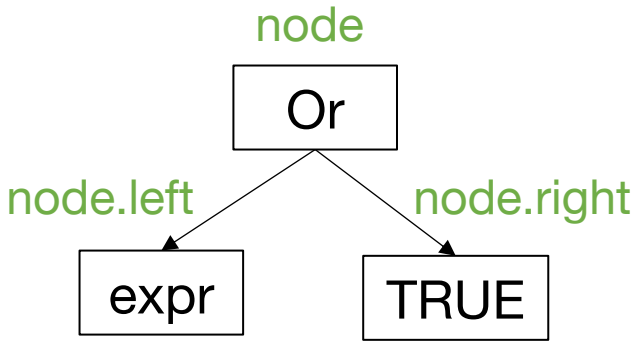
Procedure to replace part of the query plan based on a pattern seen in the plan

Example: When I see expr OR TRUE for an expression expr, replace this with TRUE

Implementing Rules

Each rule is typically a function that walks through query plan to search for its pattern

```
void replaceOrTrue(Plan plan) {
  for (node in plan.nodes) {
    if (node instanceof Or) {
      if (node.right == Literal(true)) {
        plan.replace(node, Literal(true));
        break;
      }
      // Similar code if node.left == Literal(true)
    }
  }
}
```



Implementing Rules

Rules are often grouped into *phases*

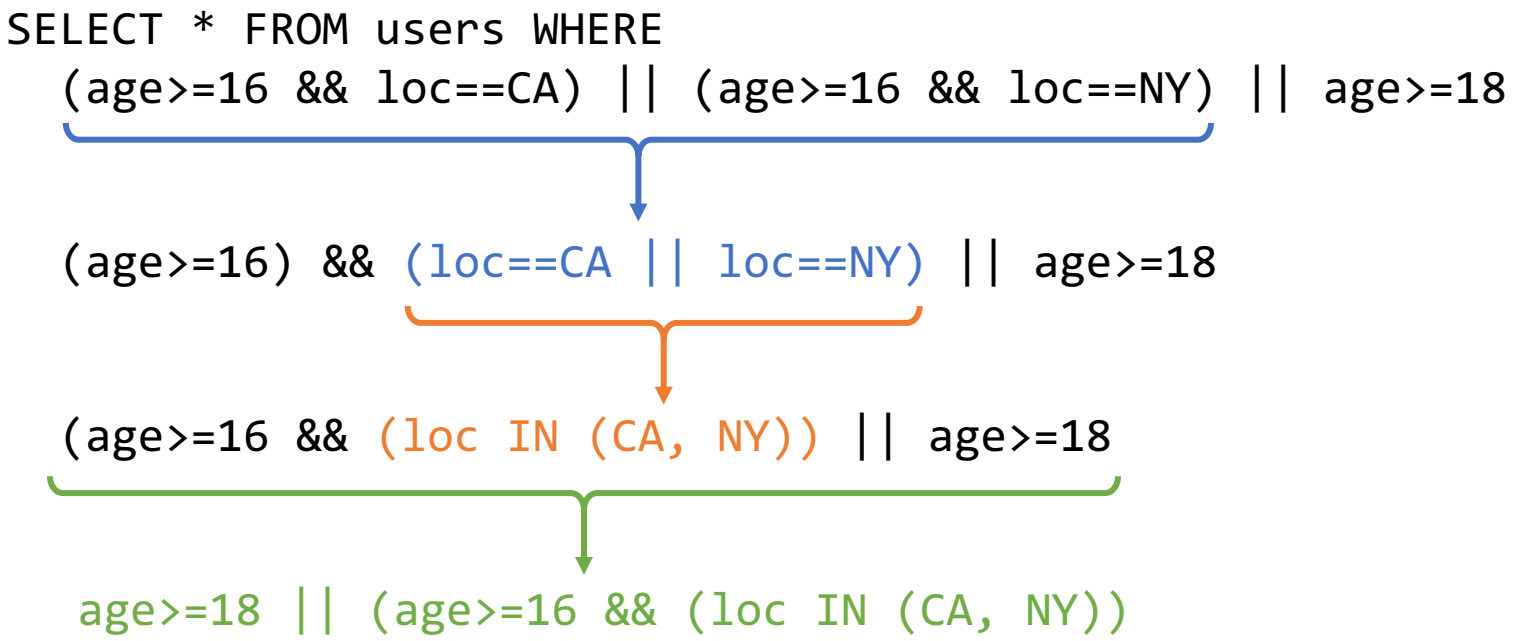
» E.g. simplify Boolean expressions, pushdown selects, choose join algorithms, etc

Each phase runs rules till they no longer apply

```
plan = originalPlan;
while (true) {
  for (rule in rules) {
    rule.apply(plan);
  }
  if (plan was not changed by any rule) break;
}
```

Result

Simple rules can work together to optimize complex query plans (if designed well):



Common Rule-Based Optimizations

Simplifying expressions in select, project, etc

- » Boolean algebra, numeric expressions, string expressions, etc
- » Many redundancies because queries are optimized for readability or produced by code

Simplifying relational operator graphs

- » Select, project, join, etc

↖ These relational optimizations have the most impact

Common Rule-Based Optimizations

Selecting access paths and operator implementations in simple cases ← Also very high impact

- » Index column predicate ⇒ use index
- » Small table ⇒ use hash join against it
- » Aggregation on field with few values ⇒ use in-memory hash table

Rules also often used to do type checking and analysis (easy to write recursively)

Common Relational Rules

Push selects as far down the plan as possible

Recall:

$\sigma_p(R \bowtie S) = \sigma_p(R) \bowtie S$ if p only references R

$\sigma_q(R \bowtie S) = R \bowtie \sigma_q(S)$ if q only references S

$\sigma_{p \wedge q}(R \bowtie S) = \sigma_p(R) \bowtie \sigma_q(S)$ if p on R, q on S

Idea: reduce # of records early to minimize work in later ops; enable index access paths

Common Relational Rules

Push projects as far down as possible

Recall:

$\Pi_x(\sigma_p(R)) = \Pi_x(\sigma_p(\Pi_{xUz}(R)))$ z = the fields in p

$\Pi_{xUy}(R \bowtie S) = \Pi_{xUy}((\Pi_{xUz}(R)) \bowtie (\Pi_{yUz}(S)))$

x = fields in R, y = in S, z = in both

Idea: don't process fields you'll just throw away

Project Rules Can Backfire!

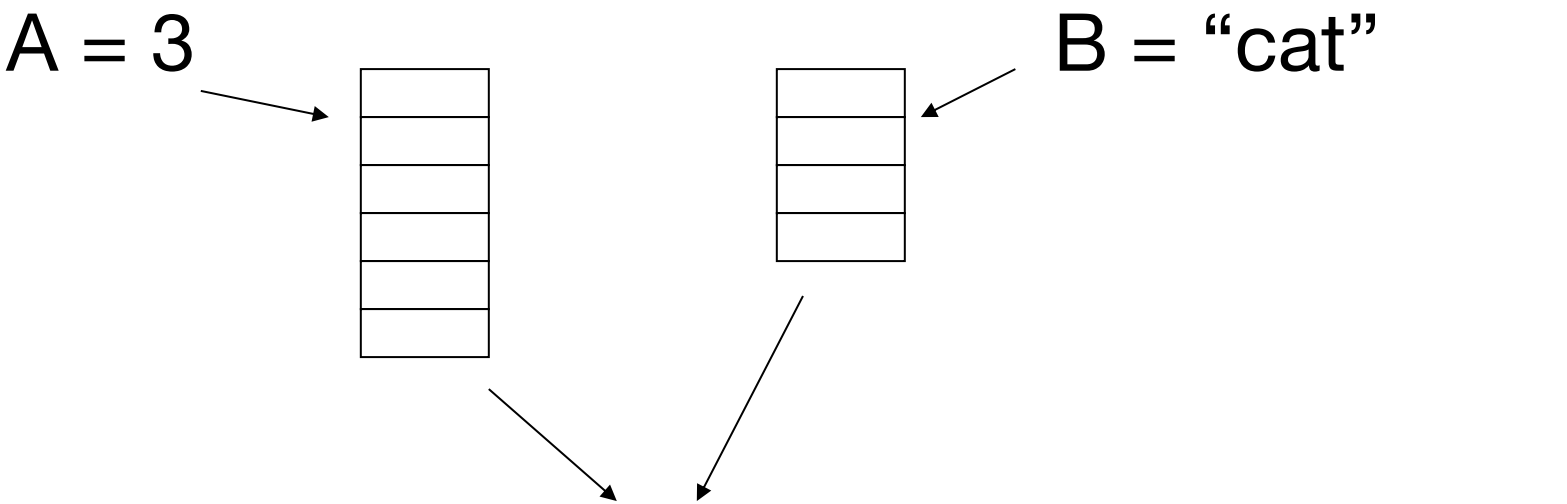
Example: R has fields A, B, C, D, E

p: A=3 ∧ B="cat"

x: {E}

$\Pi_x(\sigma_p(R))$ vs $\Pi_x(\sigma_p(\Pi_{\{A,B,E\}}(R)))$

What if R has Indexes?



In this case, should do $\sigma_p(R)$ first!

Bottom Line

Many valid transformations will not always improve performance

Need more info to make good decisions

- **Data statistics:** properties about our input or intermediate data to be used in planning
- **Cost models:** how much time will an operator take given certain input data statistics?