Query Planing & Optimization

Holger Pirk

Query Optimization

Motivation

A step back

- Secondary goal: Performance
  - some kind of numeric value
- Primary goal: Correctness
  - a boolean

Query Optimization

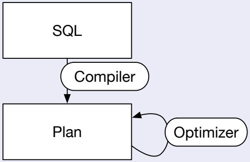
- Start with a correct plan
- Create a better plan
  - maintain correctness
  - a better plan is often much more complicated

Plan Correctness

Expectation Management

- Correctness is hard to prove when semantics are fuzzy
- Query optimizers settle for equivalence
- This puts the burden of correctness on the initial compiler

Visualisation



Plan equivalence

Relational Algebra

- is closed, i.e., every operator takes relations as input and produces relations
- Operators are easily composable
- Syntactically correct plans: easy
- Semantically correct plans: harder
- Semantically equivalent plans: very tricky

Semantic equivalence

- Plans are (semantically) equivalent if they (provably) produce the same output on any database
- Plan equivalence is quite hard to prove

Idea

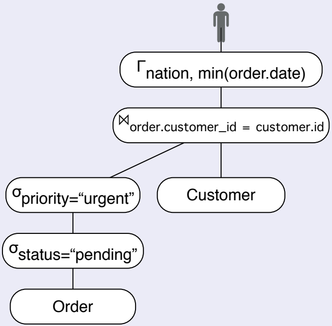
- Divide and Conquer

Transformations (a.k.a. Plan Rewriting)

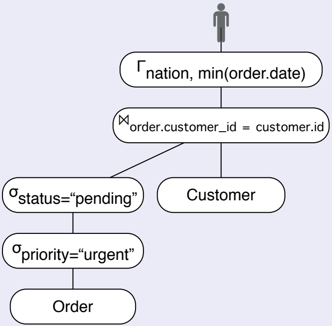
Rationale

- An equivalent transformation of a subplan is an equivalent transformation of the entire plan
- For example: adjacent selections can be reordered

Before



After



Operator Reordering

Many operators are commutative

- Two-way Joins
- Selections
- Unions
- Differences

This allows us to swap their order

- The question is: why would we want that?

Cost Metric

What constitutes a "better" plan

- Not an easy question to answer
- We define some numeric cost metric

Examples for Numeric Cost Metrics

- Sum of all produced Tuples (intermediate and final)
- Number if Page Faults (I/O)
- Number of volcano function calls
- CPU costs
- max(I/O, CPU)
- Total Intermediate Size

Rule-Based Query Optimization

Idea

- Create localized transformation rules in the form
  `Pattern => Rewrite`
  - For example
    ```
    Select(Select(input, condition1), condition2) =>
      Select(Select(input, condition2), condition1)
    ```

Application

- Traverse the plan tree from the root on (in any order)
- For every traversed node, see if the pattern matches
- If so, replace it with the rewrite and start again from the root
- If the pattern never matched, you are done

The problem

- How do you decide when to reorder

Rule-Based Query Optimization

The solution: guards

- Make sure that a produced rewrite does not match the same rule again
- Place guard conditions on the rules
  - An easy example
  - `Select(Select(input, condition1), condition2)`

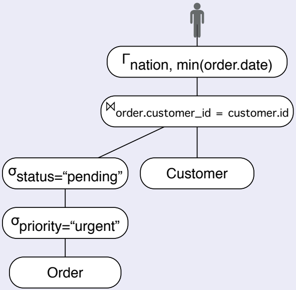
```
 if condition1.cmp = '>' and condition2.cmp = '=='
    => Select(Select(input, condition2), condition1)
```

Context

- Rule-based Query Optimization is the standard in "simple" DBMSs
  - MonetDB, Spark
- Often wrong
  - Does not take data into account

Rule-Based Query Optimization in Action

Before Optimization



Customer

id	nation
1	UK
2	USA
3	China
4	Uk

Order

status	priority	c_{id}	date
x	x	1	17
x	x	2	12
x	x	1	5
x	x	3	93
x	x	3	21
x	x	3	42
x	x	1	31
x	x	2	8
x	x	3	74
x	x	2	44
x	x	1	94
x	x	2	88

. . . and four million more

Cost-Based Query Optimization

Idea

- Cost are data dependent...
- ...but we don't know the data (before running the query)...
- ...so, let's estimate it!

A simple approach

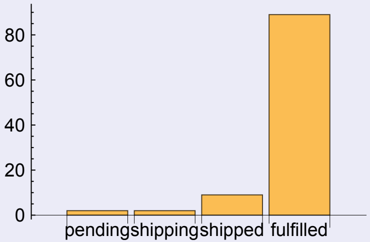
- Let's Estimate the number of tuples produced by an equality select
  - (remember, joins are cross products with selects on top)
- We are selecting one value out of all values in the database
  - Assuming uniform distribution: $\frac{1}{distinctvaluesincolumn}$
- Let's keep the number of distinct values as a "statistic"
- Selectivity of `priority=="urgent"` predicate: 50%
- In practice: only very few orders are urgent, say 2%

Statistics

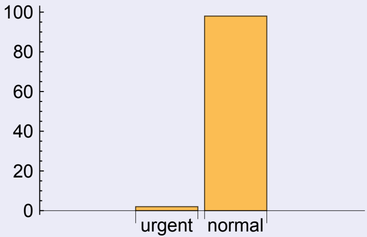
Histograms

- keep a tuple count for every unique value for every column
- Equality predicate selectivity is simply $\frac{\text{occurences of a value}}{\text{total tuple count}}$
- General predicate estimates basically evaluate the query on the histogram first

Status Histogram



Priority Histogram



Complicating Factors

Attribute Correllation

The question

- What is the selectivity of the second selection (`status=="pending"`)
- Assume we have a histogram
- well, it is 2%
- (assuming attribute independence)

Now, assume the following

- The median time to fulfill an order is a week
- Some orders take more than two weeks
- Someone gets upset about this and tries to fix it
- The person sets all order that are pending for more than a week, to priority urgent
- That means, that 50% of the pending orders are now urgent

Physical Plans

Counting tuples is easy, counting costs is hard

- Physical plans are more complicated: they don't only contain the relational operator but the algorithm
- Different algorithms have different costs
  - In terms of intermediate sizes
  - In terms of CPU (i.e., function calls)
- For example: Nested Loop Joins
  - require less space (no need for overallocation)
  - and don't require hash calculation
  - But induce more comparisons
- State of the art: physical plan optimization is rule based
  - Remember the rules for join algorithms? Yeah, that!
- Cost based optimization of physical plans is a research topic (incidentally, one of my topics)

Access Path Selection

Data can be read from multiple sources

- The base table
- A column-store index
- A tree index
- Bitmaps

Indices usually don't contain all the necessary data

- They are mainly used for tuple selection
  - not attribute projection
- They may need to be combined with base table data

Access Path Selection

Example

- A customer has six attributes: `id, name, address, nation, phone, accountNumber`
- Suppose you have a column-index on `nation`
- The query is `select * from customer where nation = "UK"`

The End