Lecture 10
10/11/2017

Project Proposals Due
Quiz Recap

Selinger review

Last time, saw how Selinger optimizer estimates selectivity, and costs plans, and looked at how Postgres estimates selectivity.

Then discussed join ordering problem, where N joins results in N! possible orderings

Selinger reduces this to 2^n possible plans; still big, but a lot smaller

Selinger assumptions:
- Push down selections
- Left deep only (reduces number of plans considered)

Key observation:

Optimal substructure property: The best way to join a subset of relations N* in a set of N relations is the same no matter what other relations are in N. E.g., if the best way to join the set of relations {A,B,C} is the plan (AB)C, then that won't change when I'm considering how to join D to that subset. -- I can do (D(AB))C or ((AB)C)D -- and only the 2nd one is left deep.

The reason this is true is that what happens after I join these three relations is the same no matter how I did the join (mostly, except if I want a sorted output -- more later.)

Therefore: if I want to join, say ABCD -- if I already know best way to join all size 3 subsets (ABC,ABD,BCD), then I can compute the optimal 4-join just by finding the lowest cost way to add the additional relation onto each of these optimal 3-joins, and then picking the overall lowest cost plan.

So the plan is to compute the optimal way to join progressively larger subsets, remembering the best way to compute each smaller subset, so that I won't have to recompute them when figuring out the best way to join with a larger subset.

(Show slides)

<u>So what's the deal with sort orders? Why do we keep interesting sort orders?</u>

Selinger says: although there may be a 'best' way to compute ABC, there may also be ways that produce interesting orderings -- e.g., that make later joins cheaper or that avoid final sorts.

So we need to keep best way to compute ABC for different possible sort orders.

so we multiply by "k" -- the number of interesting orders

In reality query optimizers eliminate the left deep assumption (we'll look at an example problem in a minute where that matters), this increases complexity to 3^n.

Measuring the Complixity of Join Enumeration in Query Optimization, VLDB 1990

So thats query optimization.

Previous years we talked about materialized views (show slide), but skipping that this time.

Instead, let's look at some query plans in Postgres.

Quiz review:

4. [8 points]: In addition to the functional dependencies suggested by the statements in the lower right of the diagram, assume that every attribute is functionally dependent on the key of the entity in which it appears. Write a collection of BCNF relations corresponding to this diagram.

(Write your answer in the space below.)

Answer:
fish: (fid, color, tank refs tank.tid)
fish_eats: (fid, ftid, quantity)
foodtype: (ftid, name)
tank: (tid, size)

Note that fish-tid is a many to one dependency, therefore we don't need an extra table for it.

Answer:

The hobby and dept tables can both fit into RAM. The emp and hobbies do not. By doing the emp-dept join first, with emp as the outer, we are able to scan emp just once. We only scan dept once because it fits into memory. We have to do the join with hobbies next (because the join with hobby would be a cross product). Due to the constraints of left-deep plans, we must put the hobbies table on the inner, requiring us to scan it once for each tuple output by the emp/dept join (10^4 times). Finally, we can do the join with hobby; since hobby fits into memory, we only scan it once as well.

The following table summarizes the pages used by each of the tables:

Table	Formula	No. pages
emp	$10^5/(1000/25)$	2500
dept	$10^3/(1000/12)$	12
hobby	$10^3/(1000/25)$	25
hobbies	$2 \times 10^5/(1000/8)$	1600