Lecture 11
10/8/2014

PS 2 Due

Selinger Optimizer (Continued)

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

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

E.g., ABC ===> A B C or A BC ==> Denote as (AB)C or A(BC)
BAC B C
...

==> n! * choose(2(N-1),(N-1))/(N) possible plans

6 * 2 == 12 for 3 relations

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

Ok, how does Selinger do this?

A combination of heuristics plus dynamic programming.

Heuristics:

1) Push down selections and projections to leaves

Now left with a bunch of joins to order.

2) only left deep; e.g., ABCD => (((AB)C)D) show

3) ignore cross products

e.g., if A and B don't have a join predicate, doing consider joining them

However, there are still n! orderings. Can we just enumerate all of them? (No)

10! -- 3million
20! -- 2.4 * 10 ^ 18

so how do we get around this?

Dynamic programming:

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.

Algorithm: compute optimal way to generate every sub-join of size 1, size 2, ... n (in that order).

```
R <--- set of relations to join
for ∂ in {1...|R|}:
    for S in {all length ∂ subsets of R}:
        optjoin(S) = a join (S-a), where a is the single relation that minimizes:
            cost(optjoin(S-a)) +
            min cost to join optjoin(S-a) to a +
            min. access cost for a
```

example: ABCD
only look at NL join for this example

A = best way to access A (e.g., sequential scan, or predicate pushdown into index...)
B = " " " " B
C = " " " " C
D = " " " " D

{A,B} = AB or BA
{A,C} = AC or CA
{B,C} = BC or CB
{A,D}
{B,D}
{C,D}

{A,B,C} = remove A - compare A({B,C}) to ({B,C})A
remove B - compare ({A,C})B to B({A,C})
remove C - compare C({A,B}) to ({A,B})C
{A,C,D}
{A,B,D}
{B,C,D}

{A,B,C,D} = remove A - compare A({B,C,D}) to ({B,C,D})A
.... remove B
remove C
remove D

Complexity:

number of subsets of size 1 * work per subset = W+
number of subsets of size 2 * W +
...
number of subsets of size n * W+

n + n + n ... n
1 2 3 n

number of subsets of set of size n = power set of n = 2^n
(string of length n, 0 if element is in, 1 if it is out; clearly, 2^n such strings)

(reduced an n! problem to a 2^n problem)

what's W? (at most n)

so actual cost is: 2^n * n
n=12 --> 49K vs 479M

So what's the deal with sort orders? Why do we keep interesting sort orders?

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

Self-Tuning Database Systems: A Decade of Progress

Who are the authors of this paper? A team from Microsoft Research who has done outstanding work on automated database design.

This paper is a retrospective of their work. So obviously biased towards what they do, but still a good overview.

What is the problem they are trying to solve?

Two key ones:
1) Automatically create indexes for a set of tables
2) Automatically create materialized views for a set of tables

Why do we want to do this?

Because for non-sophisticated users, figuring out what to create is hard, and it can change over time, so we'd like it to be automatic.

What is a materialized view?

A view that is stored persistently on disk, and kept up to date as tables change. Query optimizer will choose to use a materialized view when it can be used to answer a query.

When are these useful?

If there are common queries that are run over and over, E.g., some kind of sales report

sales : (saleid, date, time, register, product, price, ...)

CREATE MATERIALIZED VIEW sales_by_date as
SELECT date,product,sum(price),count(*) AS quantity
FROM sales
GROUP BY date, product)
Then any query that looks at sales over a particular time period can use this view.

Why are these problems hard?

Consider the problem of index selection. Could create an index on any set of attributes (if I allow multi-attribute indexes.)

Huge number of possible choices. (2^n, where n is number of attributes)

Why are multi-attribute indexes useful? "Covering indexes" -- sort of like materialized views, can answer queries directly from indexes if all attributes I need are in the index. Avoids have to do expensive random I/O in unclustered indexes. They argue this is important in many database queries.

What about materialized view selection?

Obviously a massive space of possible materialized views you could create, since they can be any query.

Since these are all computationally hard problems, fall back on heuristics.

So what's the basic approach they take?

The same for both indexes and materialized views.

1. "*Pruning*" -- Use a trace of historical queries run against the database to decide what materialized views / indexes to create (a "workload"). Need to prune out some infrequent queries from the workload, to consider columns / groups of columns that are frequently queried together.

2. "*Candidate Selection*" -- Derive a set of candidates from the workload -- these candidates have the property that they are optimal, but may use a ton of space. For example, a query like:

SELECT t.a FROM t WHERE t.b = C

We might create an index with t.a and t.x, or a materialized view with t.a and t.x where just those records records with t.x = C are in it.

3. "*Merging*" -- Consider possible mergings -- idea is that these mergings might be suboptimal but not too bad, and each merged candidate should take less space than the candidate it merges.

Example of merging indexes:

For index on (a,b) and index on (a,c), we might created merged index (a,b,c) -- this is suboptimal (especially for a,c), but smaller than the previous two.

Example of merging views.

Many ways to do this, but if my views are:

SELECT t.a FROM t WHERE t.b = C1 and
SELECT t.a FROM t WHERE t.c = C2

I could do SELECT t.a FROM t WHERE t.b = C1 or T.c = C2

At this point there are a big collection of views, and possible mergings, which will take up a ton of space.

4. "*Enumeration*" : Choose which indexes / views to create

Two broad approaches, all based on heuristics.

Bottom up:
- measure the utility of each structure as Δt/space (i.e/ the reduction in runtime for each unit of space it occupies.)

- greedily add highest utility structures until space bound is met, re-evaluating utility after each addition

Top down:
- start with the optimal structures created by the candidate selection step

- consider each possible merging M of two structures A,B

    - replace the A,B pair with M where Δt/space gained is minimized
    e.g.,

    1 second slower and 100 MB space saved (1/100)

        is better than

    10 seconds slower and 10 MB spaced saved (10/10)

How do they propose estimate the cost/benefit of a particular structure?

What-if extensions to the optimizer

What are those?

Basically a way to ask the optimizer what the effect of adding a particular index would be, without actually creating the index (think of this as adding fake entries to the statistics tables in the database.)