Lecture 10
10/6/1014

Query Optimization

PS 2 due Wednesday.

Selinger

Famous paper. Pat Selinger was one of the early System R researchers; still active today.

Lays the foundation for modern query optimization. Some things are weak but have since been improved upon.

Idea behind query optimization:
    (Find query plan of minimum cost)

How to do this?
    (Need a way to measure cost of a plan (a cost model))

single table operations

how do i compute the cost of a particular predicate?
    compute it's "selectivity" - fraction F of tuples it passes

how does Selinger define these? -- based on type of predicate and available statistics

what statistics does system R keep?
- NCARD - "relation cardinality" -- number of tuples in relation
- TCARD - # pages relation occupies
- ICARD - keys (distinct values) in index
- NINDX - pages occupied by index
- min and max keys in indexes

(have to realize that the complexity of statistics you could keep in 1978 was pretty simple!)

<u>Estimating selectivity F:</u>

col = val
    F = 1/ICARD() (if index available)
    F = 1/10 (where does this come from?)

col > val
    high key - value / high key - low key (if index available)
    1/3 o.w.

col1 = col2
    1/MAX(ICARD(col1, col2))
    1/10 o.w.

ex: suppose emp has 1000 records, dept has 10 records
total records is 1000 * 10, selectivity is 1/1000, so 10 tuples expected to pass join
(note that this is wrong if doing key/fk join on emp.did = dept.did, which will produce 1000 results!)

Note that selectivity is defined relative to size of cross product for joins!

p1 and p2

    F1 * F2

p1 or p2

1 - (1-F1) * (1-F2)

then, compute access cost for scanning the relation.
how is this defined?
(in terms of number of pages read)

equal predicate with unique index: 1 [btree lookup] + 1 [heapfile lookup] + W

(W is CPU cost per predicate eval in terms of fraction of a time to read a page)

range scan:

clustered index, boolean factors: F(preds) * (NINDX + TCARD) + W*(tuples read)

unclustered index, boolean factors: F(preds) * (NINDX + NCARD) + W*(tuples read)
unless all pages fit in buffer -- why?
...
seq (segment) scan: TCARD + W*(NCARD)

Is an index always better than a segment scan? (no)

multi-table operations

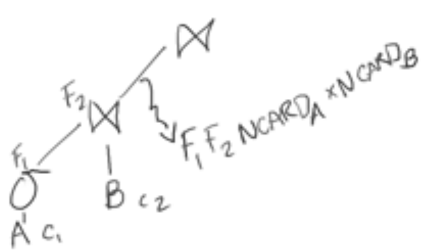
how do i compute the cost of a particular join?

algorithms:
NL(A,B,pred)
Cost(A) + NCARD(A) * Cost(B)

Note that inner is always a relation; cost to access depends on access methods for B; e.g.,
w/ index -- 1 + 1 + W
w/out index -- TCARD(B) + W*NCARD(B)

Cost(A) is cost of subtree under outer

How to estimate # NCARD(outer)? product of F factors of children, cardinalities of children
example:



Merge_Join_x(P,A,B), equality pred

Cost(A) + Cost(B) + sort cost

(Saw cost models for these last time)

At time of paper, didn't believe hashing was a good idea

Overall plan cost is just sum of costs of all access methods and join operators
Then, need a way to enumerate plans

Iterate over plans, pick one of minimum cost

Problem:

Huge number of plans. Example:

suppose I am joining three relations, A, B, C
Can order them as:

(AB)C
A(BC)
(AC)B
A(CB)
(BA)C
B(AC)
(BC)A
B(AC)
(CA)B
C(AB)
(CB)A
C(BA)

Is C(AB) different from (CA)B?
Is (AB)C different from C(AB)?
    yes, inner vs. outer

n! strings * # of parenthetizations

How many parenthetizations are there?

Consider N=1,2,3,4:

  A: (A)
  AB: ((A)(B))
  ABC: ((AB)C), (A(BC))
  ABCD:(((AB)C)D), ((A(BC))D), ((AB)(CD)), (A((BC)D)), (A(B(CD)))

The numbers of plans for N=1,2,3,4 are:

  plans(1) = 1
  plans(2) = 1
  plans(3) = 2
  plans(4) = 5

Generally, plans(N) = choose(2(N-1),(N-1))/(N) *

* The Art of Computer Programming, Volume 4A, page 440-450

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

6 * 2 == 12 for 3 relations

(study break -- postgres)

Ok, so what does Selinger do?

Push down selections and projections to leaves

Now left with a bunch of joins to order.

Selinger simplifies using 2 heuristics? What are they?

- only left deep; e.g., ABCD => (((AB)C)D) show
- ignore cross products

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

still n! orderings. can we just enumerate all of them?

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

so how do we get around this?

Estimate cost by dynamic programming:

idea: if I compute join (ABC)DE -- I can find the best way to combine ABC and then consider all the ways to combine that with DE.

I can remember the best way to compute (ABC), and then I don't have to re-evaluate it.   Best way to do ABC may be ACB, BCA, etc -- doesn't matter for purposes of this decision.

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 (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+

$$\binom{n}{1} + \binom{n}{2} + \binom{n}{3} \ldots \binom{n}{n}$$

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

how are things different in the real world?
- real optimizers consider bushy plans (why?)

```
     A
  D     B
     C     E
```

- selectivity estimation is much more complicated than selinger says
and is very important.

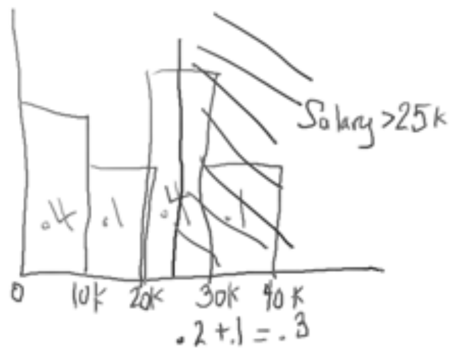
how does selinger estimate the size of a join?

- selinger just uses rough heuristics for equality and range predicates.

- what can go wrong?
consider ABCD
suppose sel (A join B) = .1
everything else is .01
If I don't leave A join B until last, I'm off by a factor of 10

- how can we do a better job?
(multi-d) histograms, sampling, etc.

example: 1d hist



example: 2d hist