Lecture 9
3.16.21

Query Optimization

Lab 2 due Friday

Signup sheets for final projects

Java tutorial on Monday

Agenda:

Finish discussion of join algos
Query Optimization

Simple hash:

```
i = 0;
pass size = v (e.g., v = 1)  // if P partitions, hash into [1…n], e.g., h(x) = x mod P
for partition i (on hash values in range range [v*i , v*(i+1)) )
        scan S, hash, if in partition, insert into hash table
        o.w., write back out

        scan R, hash, if in partition, lookup in hash table, output matches
        o.w., write back out
```

repeat with reduced R and S, in round i+1 ; example:

R = 1, 4, 3, 6, 9, 14, 1, 7, 11
S = 2, 3, 7, 12, 9, 8, 4, 15, 6
h(x) = x mod 3, pass size = 1
Pass 1: h(x) in range [0..1)
R hash table: 3 6 9
remainder: 1 4 14 1 7 11
S probe with : 3 12 9 15 6 --> 3 6 9 join
remainder: 2 7 8 4

Pass 2: h(x) in range [1..2)
R hash table: 1 4 1 7
remainder: 14 11
S probe with : 7 4 --> 7 4 join
remainder: 2 8

Pass 2: h(x) in range [1..2)
R hash table: 14 11
S probe with : 2 8 --> no join

Somewhat complex to analyze:

Read R, S (seq)

Amount we write depends on number of passes. In pass 1, we write:

((p-1)/p)R, ((p-1)/p)S (seq)

We then read all this data back in (seq), and in pass 2, we write:

((p-2)/p)R, ((p-2)/p)S (seq)

And so on...

So for 2 passes, we get:

Read |R+S|, Write (1/2)(|R|+|S|), Read (1/2)(|R|+|S|) and are done.
Total IO is 2(|R|+|S|)

For 3 passes, total IO is 3(|R|+|S|)
For n passes, total IO is n(|R|+|S|)

Is this better than blocked hash?
(Depends on relative size of |R| and |S| -- if |S| is much smaller than R, blocked has will be better since it doesn't rewrite R)

Grace hash:

```
        choose P partitions, with one page per partition
        hash r into partitions, flushing pages as they fill
        hash s into partitions, flushing pages as they fill
        for each partition p
                build a hash table T on r tuples in p
                lookup each s in T outputting matches
```

example:

R = 1, 4, 3, 6, 9, 14, 1, 7, 11
S = 2, 3, 7, 12, 9, 8, 4, 15, 6

h(x) = x mod 3

R0 = 3 6 9
R1 = 1 4 1 7
R2 = 14 11

S0 = 3 12 9 15 6
S1 = 7 4
S2 = 2 8

Now, join R0 with S0, R1 with S1, R2 with S2

Because we are using the same hash function for R and S we can guarantee that the only tuples that will join with partition Ri are those in Si

How do I pick the partition size?

(Assume uniform distribution of tuples to partitions, make each partition equal to M pages (minus a couple for active pages of S being read.)

sqrt(|R|) < M

partition size = M > sqrt(|R|)

#parts P = |R|/(M) ≤ |R|/ sqrt(|R|) = sqrt(|R|)
h(v) -> [1,k]
each covers k/P hash values

Need sqrt(|R|) pages of memory b/c we need at least one page per partition as we write out (note that simple hash doesn't have this requirement)

I/O:

read R+S (seq)
write R+S (semi-random)
read R+S (seq)

also 3(|R|+|S|) I/OS

What's hard about this?

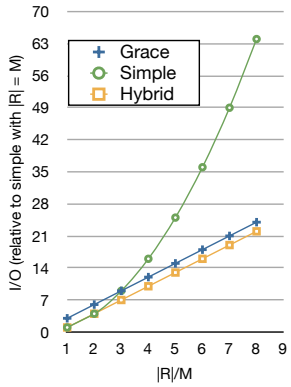
Possible that some partitions will overflow -- e.g., if many duplicate values
What do they say we should do?
- (Leave some more slop (assign fewer values to each partition) by assuming that each record takes a few more bytes to store in hash table.)
- Split partitions that overflow

When does grace outperform simple?

(When there are many partitions, since we avoid the cost of re-reading tuples from disk in building partitions)

When does simple outperform grace?
(When there are few partitions, since grace re-reads hash tables from disk)

So what does Hybrid do?
- M = sqrt(|R|) + E
- Make first partition of size E, do it on the fly.
- Do remaining partitions as in grace.



Why does grace/hybrid outperform sort-merge?
- CPU Costs!
- I/O costs are comparable
- 690 / 1000 seconds in sort merge are due to the costs of sorting
- 17.4 in the case of CPU for grace/hybrid!

Will this still be true today?
- (Yes)

Show example queries in Postgres.

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 the size of its input)

How do I estimate the size of an operators input? (From stats over base tables, or using "selectivity" - fraction F of tuples -- it's children passed)

How does Selinger estimate size of base tables? -- Using some (simple) statistics :

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

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

How does Selinger estimate selectivity F:

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

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

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

Example: 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)

Estimating the cost of single table operations

How is cost defined?
(in terms of number of pages read + a weighted factor of # predicate evals)

Equality 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

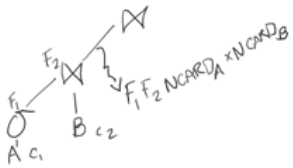
How do i compute the cost of a particular join?

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

(Some of these plans Selinger wouldn't consider because they aren't left deep)

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)

==> 4 choose 2 / 3 == 6 / 3 = 2
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+

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

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.

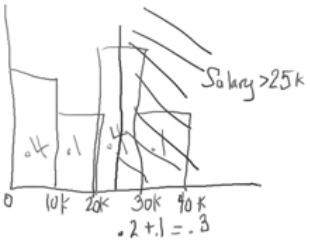
how does selinger estimate the size of a join?

- 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