Lecture 9
10/8/09

Query Optimization

Lab 2 due next Thursday.

M pages memory
S and R, with |S| |R| pages respectively; |S| > |R|
M > sqrt(|S|)

External Sort Merge

    split |S| and |R| into memory sized runs
    sort each
    merge all runs simultaneously

total I/O 3 |R| + |S|
(read, write, read)

"Simple" hash

    given hash function h(x), split h(x) values in N ranges
    N = ceiling(|R|/M)

```
for (i = 1…N)
    for r in R
        if h® in range i, put in hash table Hr
        o.w. write out
    for s in S
        if h(s) in range i, lookup in Hr
        o.w. write out
```

total I/O

    N (|R| + |S|)

Grace hash:

```
for each of N partitions, allocate 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 Hr on r tuples in p
    hash s, lookup on Hr
```

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

R1 = 3 6 9
R2 = 1 4 1 7
R3 = 14 11

S1 = 3 12 9 15 6
S2 = 7 4
S3 = 2 8

Now, join R1 with S1, R2 with S2, R3 with S3

Note -- need 1 page of memory per partition. Do we have enough memory?

We have |R| / M partitions

M ≥ sqrt(|R|)

worst case

|R| / sqrt(|R|) = sqrt(|R|) partitions

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

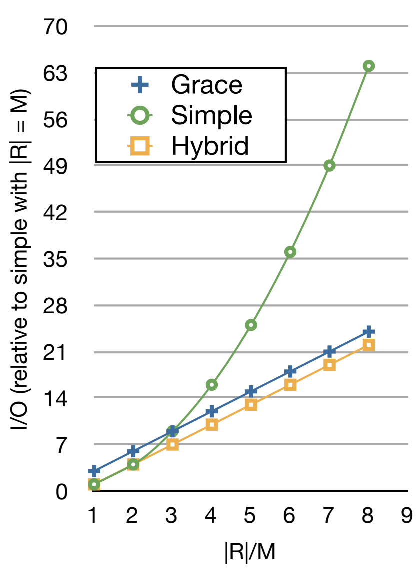
What's hard about this?

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 (as in simple)
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)

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?
- relation cardinalities NCARD
- # pages relation occupies TCARD
- keys in index ICARD
- pages occupied by index NINDX

Estimating selectivity F:

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

col > val
high key - value / high key - low key
1/3 o.w.

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

ex: suppose emp has 10000 records, dept as 1000 records
total records is 10000 * 1000, selectivity is 1/10000, so 1000 tuples expected to pass join
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)
C-outer(A) + NCARD(outer) * C-inner(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)

C-outer is cost of subtree under outer

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

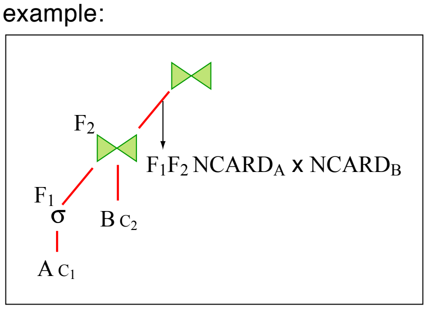


Image by MIT OpenCourseWare.

Merge_Join_x(P,A,B), equality pred

C-outer + C-inner + 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?

```
ABCD  -->  (AB)CD  A(BC)D  AB(CD)  3
             XCD      AXD        ABX        * 2
                                           ===
                                           6 --> (n-1)!
 ==> n! * (n-1)!
```

6 * 2 == 12 for 3 relations

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? (n)

so actual cost is: 2^n * n

<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 join orders.

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

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

```
      A
  D       B
    C       E
```

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

<u>how does selinger estimate the size of a join?</u>

- 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 .1
    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

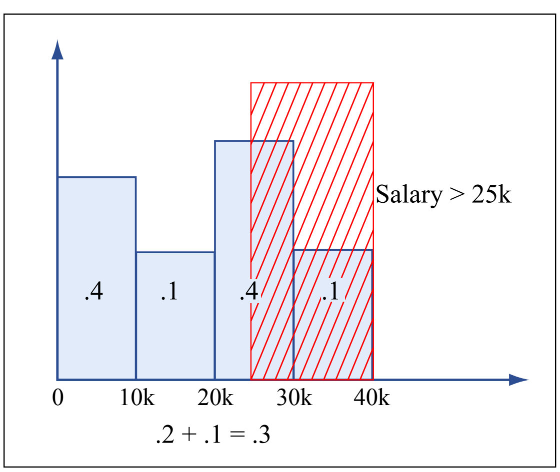


Image by MIT OpenCourseWare.

example: 2d hist

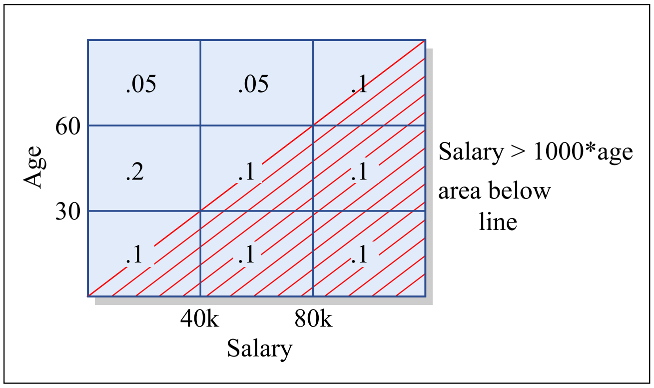


Image by MIT OpenCourseWare.