

Lab 2 -- Due Weds.

Project meeting sign ups

Recap column stores & paper

### ***Join Processing:***

$S = \{S\}$  tuples,  $|S|$  pages

$R = \{R\}$  tuples,  $|R|$  pages

$|R| < |S|$

$M$  pages of memory

### Types of joins

#### **Nested Loops**

```

for s in S
  for r in R
    if pred(s,r) output s join r
  
```

Does it matter which is inner, outer? (yes)

$\{S\} * \{R\}$  compares in either case

suppose  $|S| = 4$ ,  $|R| = 2$ ,  $M=3$ , LRU

$S$  inner =  $2 + 4 + 4 = 10$  pages read

$R$  inner =  $4 + 2 = 6$  pages read

w/out cache:  $|S| + \{S\} * |R|$  i/o s

#### **Index Nested Loops (Index on R)**

```

for s in S
  find matches in R
  
```

$|S| + \{S\}$  i/o

## Block Nested Loops

```
B = block size (< M)
while (not at end of S)
    S' = read B records from S
    for r in R
        for s in S'
            if pred(s,r) output s join r
```

<b>(Sort)Merge (IRI + ISI &lt; M -- in memory)</b>	1<	2<
Sort R, Sort S	2	3
(merge)	5	7
	7	7
IRI + ISI i/os	7	9
	9	

## Hash (simple, IRI < M -- in memory)

Build hash on R, probe with S  
IRI + ISI i/os

## Blocked hash (IRI > M)

Repeat until read all of R:  
Read in M pages of R, probe with S  
IRI + ISI \* ceil (IRI / M) i/os

## Pipelined Hash

As tuples of R arrive, add to hash on R, probe into hash on S  
As tuples of S arrive, add to hash on S, probe into hash on R

Study break

## Shapiro: (Gossip about paper)

What's this paper about?

(Join algorithms for two relations when size of either relation exceeds available RAM )

Equality joins only

What's the big takeaway?

(hash join outperforms sort-merge join )

Always?

(at least, if you have to sort the relations )

Why is this only for "large" memories?

(Requires memory equal to  $\sqrt{S}$ , where S is the larger relation )

$$M \geq \sqrt{S} \geq \sqrt{R}$$

How do these external algorithms work?

2 phases

Phase 1: partition the relation into (sorted/hashed) runs

Phase 2: join the partitions

Sort-merge:

*phase 1*

repeat until done:

read a run of S

sort

write out

repeat until done:

read a run of R

sort

write out

*phase 2*

begin reading from each run of R and S (requires one block of memory for each run)  
join R and S as they appear

Example:

R = 1, 4, 3, 6, 9, 14, 1, 7, 11

S = 2, 3, 7, 12, 9, 8, 4, 15, 6

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

run size = 3

R1	R2	R3	S1	S2	S3
1<	6<	1<	2<	8<	4<
3	9	7	3	9	6
4	14	11	7	12	15

1	6<	1	2<	8<	4<
3<	9	7<	3	9	6
4	14	11	7	12	15

1	6<	1	2	8<	4<
3<	9	7<	3<	9	6
4	14	11	7	12	15

output 3

1	6<	1	2	8<	4<
3	9	7<	3	9	6
4<	14	11	7<	12	15

output 4

1	6<	1	2	8<	4
3	9	7<	3	9	6<
4	14	11	7<	12	15

output 6

1	6	1	2	8<	4
3	9<	7<	3	9	6
4	14	11	7<	12	15<

output 7 ...

How do I pick the run size?

(Make it as big as possible, to minimize the number of runs. Runs can be at most  $M \geq \sqrt{N}$  to sort in memory. )

Claim: Can't be more than  $M$  runs. Why?

Suppose we set the length of a run to  $\sqrt{N}$

Then, there will be  $N/\sqrt{N} = \sqrt{N}$  runs , which is good b/c  $M > \sqrt{N}$ , and need one one page of memory for each run to do merge concurrently

Paper confusingly claims if you have  $M=\sqrt{N}$  memory, runs will be size  $2 * \sqrt{N}$

(Where does 2 come from. Using "selection replacement tree" -- idea is that you store values in a heap, read in a new value whenever you output an old value. Average run length will be  $2 \times$  size of memory). See Knuth.

23  
45  
64  
12  
19  
82  
97  
44

M = 3

23  
23 45  
23 45 64

Run = 23

12  
12 45  
12 45 64

Run = 23, 45

12  
12 19  
12 19 64

Run = 23, 45, 64

12  
12 19  
12 19 82

Run = 23, 45, 64, 82

12  
12 19  
12 19 97

Run1 = 23, 45, 64, 82, 97

12  
12 19  
12 19 44

Run2 = 12, 19, 44

#I/Os:

Read R+S once (seq)

Write R+S once (seq)

Read R+S once (random -- can do better if we read long runs instead of 1 page at a time, but that requires memory! Might be better to hierarchically merge sequential runs.)

## Simple hash:

$i = 0$ ;

pass size =  $v$  (e.g.,  $v = 1$ ) // if  $P$  partitions, hash into  $[1 \dots n]$ , e.g.,  $h(x) = x \bmod P$   
for partition  $i$  (on hash values in 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:

$R = 1, 4, 3, 6, 9, 14, 1, 7, 11$

$S = 2, 3, 7, 12, 9, 8, 4, 15, 6$

$h(x) = x \bmod 3$ , 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  $IR+SI$ , Write  $(1/2)(IRI+ISI)$ , Read  $(1/2)(IRI+ISI)$  and are done.  
Total IO is  $2(IRI+ISI)$

For 3 passes, total IO is  $3(IRI+ISI)$

For  $n$  passes, total IO is  $n(IRI+ISI)$

Is this better than blocked hash?

(Depends on relative size of  $IRI$  and  $ISI$  -- if  $ISI$  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:

$R = 1, 4, 3, 6, 9, 14, 1, 7, 11$

$S = 2, 3, 7, 12, 9, 8, 4, 15, 6$

$h(x) = x \bmod 3$

$R_0 = 3\ 6\ 9$

$R_1 = 1\ 4\ 1\ 7$

$R_2 = 14\ 11$

$S_0 = 3\ 12\ 9\ 15\ 6$

$S_1 = 7\ 4$

$S_2 = 2\ 8$

Now, join  $R_0$  with  $S_0$ ,  $R_1$  with  $S_1$ ,  $R_2$  with  $S_2$

*Because we are using the same hash function for  $R$  and  $S$  we can guarantee that the only tuples that will join with partition  $R_i$  are those in  $S_i$*

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.)

$\sqrt{IRI} < M$

partition size =  $M > \sqrt{IRI}$

#parts  $P = IRI/(M) \leq IRI/\sqrt{IRI} = \sqrt{IRI}$

$h(v) \rightarrow [1, k]$

each covers  $k/P$  hash values

Need  $\sqrt{IRI}$  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  $3(IRI+ISI)$  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 )

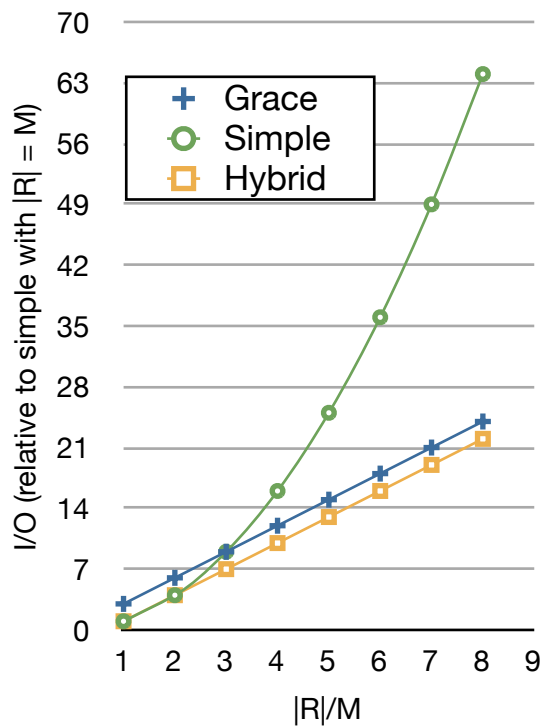
So what does Hybrid do?

$M = \sqrt{IRI} + E$

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)