Data Engineering

Cody Liew
(ca15 ;)

1) Pseudocode:

* I assumed they were sorted in ascending order

```
list1 = R(A)
list2 = S(A)
list1Index = 0
list2Index = 0
list3Length = list1.length + list2.length
list3 = new list[list3Length]
for i = 0 to list3Length - 1
  if list1[list1Index] ≥ list2[list2Index]
    list3.add(list2[list2Index])
    list2Index++
  else
    list3.add(list1[list1Index])
    list1Index++
  end
end
return list3
```

2) This is an extension of Question 1. Have each machine separately sort its blocks first. Then, similarly to Question 1, pop the first value of each block in each machine. Within each machine, compare the top block values and restructure each block so that they are also sorted relative to each other, storing temporary values as necessary. Similarly, on the machine level, read in and remove the first value of each machine comparing then iteratively to build up the sorted list. Repeat this until no more can be read into memory and write this back to the distributed filesystem with a descriptive title specifying its block ID. I would store the final output as a csv or txt file for ease of access, though each file will be fairly large.

Repeat this until no more blocks can be read

3) * Used the pseudocode on the Wikipedia page of Sort-merge join as a reference.

1) Sort R(A,C) and S(A,C) on attribute A

2) Remove all tuples with same lowest A in R and add to left_subset variable
and do the same for S into right-subset variable

3) if left_subset.A == right_subset.A
    add cartessan product of left-subset and right_subset to output
    clear left_subset and right_subset
    repeat step 2 for both R and S
else if left_subset.A < right_subset.A
    clear left_subset
    repeat step 2 for JUST R

else
    clear right_subset
    repeat step 2 for JUST S

4) Repeat step 3 until both left_subset and right_subset
are both empty

5) Return output

Illustrated answer on back

$R = \{(1,a),(3,c),(2,b),(1,d),(2,a),(3,d),(1,d),(4,a),(1,c),(3,a)\}$

$S = \{(1,f),(4,c),(2,b),(1,g),(3,a)\}$

1) Sort R and S, assuming stable sort

$R = \{(1,a),(1,d),(1,d),(1,c),(2,b),(2,a),(3,c),(3,d),(3,a),(4,a)\}$

$S = \{(1,f),(1,g),(2,b),(3,a),(4,c)\}$

2) $R = \{(2,b),(2,a),(3,c),(3,d),(3,a),(4,a)\}$ → left_subset = $\{(1,a),(1,d),(1,d),(1,c)\}$

$S = \{(2,b),(3,a),(4,c)\}$ → right_subset = $\{(1,f),(1,g)\}$

add $\{(1,a,1,f),(1,a,1,g),(1,d,1,f),(1,d,1,g),(1,d,1,f),(1,d,1,g),$
$(1,c,1,f),(1,c,1,g)\}$ to output

3) $R = \{(3,c),(3,d),(3,a),(4,a)\}$ → left_subset = $\{(2,b),(2,a)\}$

$S = \{(3,a),(4,c)\}$ → right_subset = $\{(2,b)\}$

add $\{(2,b,2,b),(2,a,2,b)\}$ to output

4) $R = \{(4,a)\}$ → left_subset = $\{(3,c),(3,d),(3,a)\}$

$S = \{(4,c)\}$ → right_subset = $\{(3,a)\}$

add $\{(3,c,3,a),(3,d,3,a),(3,a,3,a)\}$ to output

5) $R = \{\}$ → left_subset = $\{(4,a)\}$

$S = \{\}$ → right_subset = $\{(4,c)\}$

add $\{(4,a,4,c)\}$ to output

6) return output

4) a) Use a hash function to partition both R and S on the join attribute. This guarantees that all tuples can be joined within the partition after the appropriate computations are calculated. A join can then be computed within the partition using any of the general join algorithms like Sort Merge join or TNLJ. Sorting is used in each partition on the join key.

b) The map stage will partition R and S, perform the relevant computations and then emit the value of the joining attribute for each tuple as the intermediate key and the tuple itself as the intermediate value to the shuffle/reduce step. This step groups by the intermediate key's composite values against the attribute and outputs that. An example of this intermediate tuple for a tuple in R(A,B) joining on A looks like: (1, 2) → (1, R, (1, 2)). This works because each pair of joinable tuples will naturally go to the same reducer. Partition function calculates hash code from just join key. Grouping function groups on just join key.

c) A map-side join of R and S will be preferable to a reduce-side join when one of the tables is small enough to fit into memory. In this case, the cost incurred for sorting and merging in the shuffle and reduce stages is mitigated.

d) If both tables are too large to fit into memory, a reduce-side join of R and S will be preferable to a map-side join of R and S.

since sorting is not used, may have to buffer a lot of values to do reduce-side join → inefficient with memory

5) When processing each key in the reducer, can't know where joinable tuple corresponding to currently examined tuple will be encountered, since they are arbitrarily ordered. Using the composite key paradigm described in 4.2, along with first sorting each tuple by join key then table tag and having the partitioner only pay attention to the join key, so all the composite keys with same join key will arrive at the same reducer will yield performance benefits. Due to this organization, each inner loop of the TNLJ (or double for loop) can terminate more quickly, adding condition to terminate whenever it encounters a tuple with a join key that doesn't match the current key. The join will therefore run faster. More data may also be able to be buffered into main memory for performance gains. If possible, buffer ahead of the sets of same join values into memory and stream the values in from the other table and join with the buffered table.

6) *For most of these, I'm assuming performance will be evaluated relative to similarly sized data sets, because, otherwise, I'm not sure what to compare performance to, unless you're asking to compare R and S with Π(R) and Π(S), where Π(R) is the table R with all duplicate tuples eliminated.

a) The "SQL Query Plan Selection" Lecture on slide 32 states that "Cost depends on sorted-ness of inputs," which indicates many duplicates will improve the performance of Sort-Merge Join. This makes sense intuitively because a table of size N with a lot of duplicates compared to a table of size N with no duplicates is likely to be "more ordered" and therefore requires less computation to fully sort it. Also, based off the general algorithm for Sort Merge Join given in Question 3, more duplicates will lead to less "iterations" (the repeats of step 3 in step 4). This potentially yields performance gains depending on the computation cost of the cartesian product implementation. The cost of the computation repeated in step 2 will iterate through the same number of elements regardless and is therefore negligible. The increased cartesian product cost also makes sense intuitively. For instance, given $R=\{1,1,1,1\}$ and $S=\{1,1,1,1\}$, the number of joins done is 16. Given $R=\{1,2,3,4\}$ and $S=\{1,2,3,4\}$, the number of joins done is 4. If the cartesian product is relatively cheap computationally, this cost will be overpowered by the performance gains from less "iterations" (repeats of step 3 in step 4).

b) TNLJ conceptually is just a double for loop where the cartesian product is outputted if the join condition is met. Therefore, the performance of TNLJ with such duplicates will stay the same because either way, MN elements will be iterated over where $M=|R|$ and $N=|S|$. However, if the computation of a cartesian product is considered a non-negligible computation, then TNLJ on two tables with many duplicates of one key will generally be worse because of the added computations.

→ ☆ with a lot of duplicates, key cardinality is low. In this case, sort can be done in $O(k \log k)$ where k = number of unique elements using a hashmap to count frequency. Recovering sorted list will cost $O(n)$

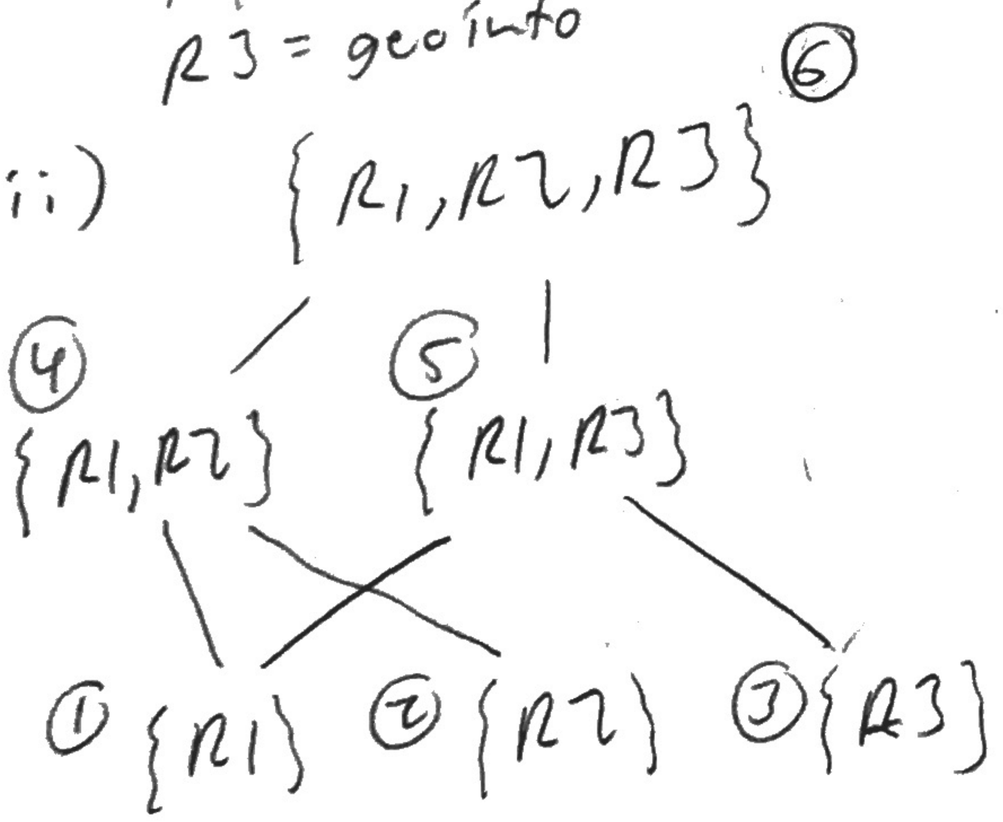
c) Duplicates will worsen the performance of a map-side join. If there are two tables where one key has many duplicates then most partitions will be small with a couple larger partitions. This is suboptimal since these partitions will compute the map-side join in parallel and therefore be bottlenecked by the larger partition of duplicate values. These data sets are another example of a skewed distribution. The performance of map-side joins are generally better when the input tables have a more even distribution.

d) This question is dependent on the size of the data set. As stated before, two tables with many duplicates of one key has a skewed distribution. The performance of the map phase will be the same assuming a partitioning scheme independent of key (ex. random partition). Therefore, we have to look at the shuffle and reduce step. If the data is sufficiently small such that the tuples with the most common key in both tables can be buffered into memory for the cross-product, then performance is about the same. For data that is sufficiently big such that the tuples with the most common key can't be buffered into memory, then this leads to out of memory issues. For data sets of this size, a higher key cardinality and more even data distribution would yield better performance since it's more likely that tuples of all distinct key values could be buffered into memory to perform the cross-product. If a partitioning scheme dependent on the key is used, the answer is the same as for a map-side join.

7) a) i) Interesting Orders: users.name, clicks.user, geoinfo.ipaddr, users.ipaddr

Let R1 = users
R2 = clicks
R3 = geoinfo

ii)



* {R2, R3} is not considered since there's no join condition between clicks and geoinfo

Selinger algorithm:

1) Computes all ways to access each relation in the query. Each relation can be accessed with a full table scan or an index scan if there's an index on a relation that can be used to answer predicate in query. For each relation, cheapest way to scan relation and cheapest way to scan relation that produces records in a particular one recorded by optimizer.

2) For each pair of relations for which a join condition exists, optimizer considers the three available join operators. It records the cheapest way to join each pair of relations and the cheapest way to join each pair of relations that produces its output according to a particular sort order.

3) All 3-relation query plans computed, by joining each two-relation plan produced by the previous phase with the remaining relations in the query

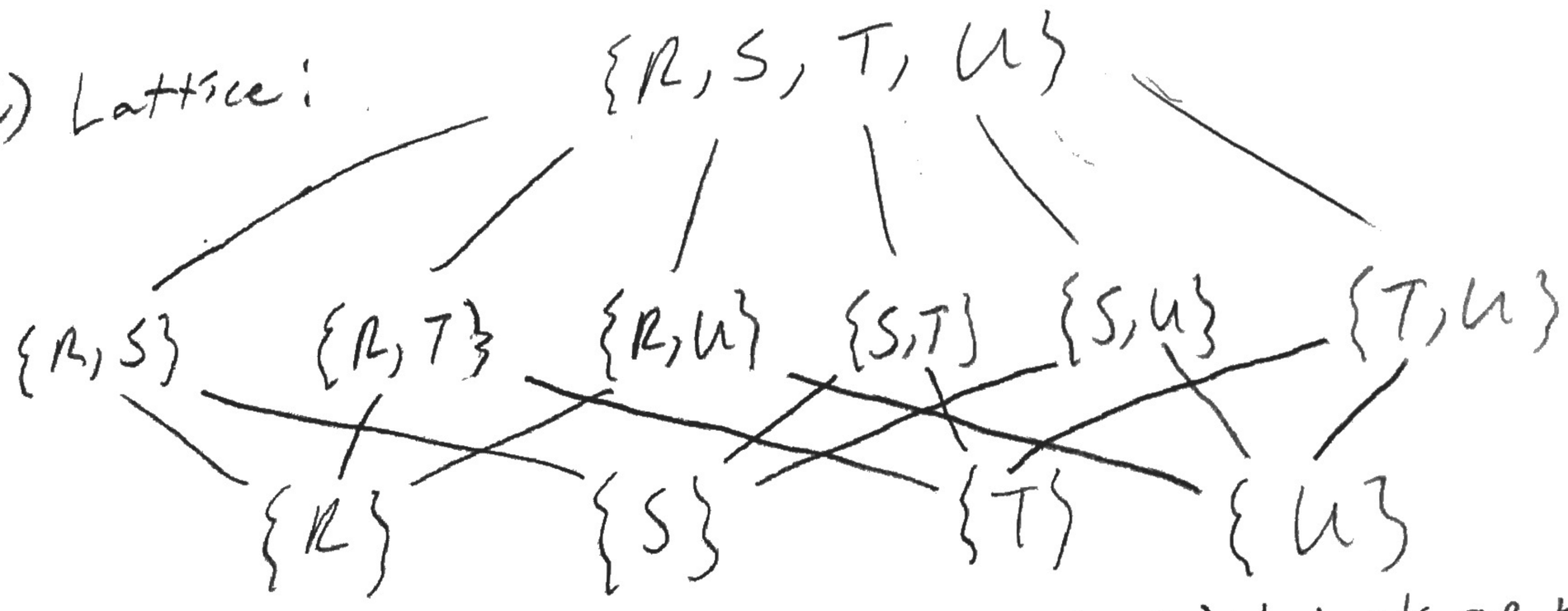
- Nodes 1, 2, and 3 consider which scan operators to use
- Nodes 4 and 5 consider which join operators to use for its node's respective pair
- Node 6 considers which pair to use ({R1, R2}, {R1, R3}) and which of three join operators to use with the corresponding remaining relation.

iii)-Node 1: Full table scan on R1 → B(R1) ~ 10^6

- Node 2: index scan on R2 → B(R) ~ 10^8
- Node 3: Full table scan on R3 → B(R) ~ 1000
- Node 4: tuple nested loop join with index on R2 as right table and R1 as left table
  = Cost(outer) + T(x) × Cost(Inner)
  = $10^6 + 10^6 = 2 \times 10^6$
- Node 5: sort merge join on R1 and R3
  ~ Cost(Right) + Cost(Left) + 2(B(x) + B(y)) = $1000 + 10^6 + 2(1000 + 10^6) \sim 3 \times 10^6$
- Node 6: tuple nested loop join with index on R2 as right table and {R1, R3} as left table
  ~ Cost(outer) + T(x) × Cost(Inner)
  ~ $3 \times 10^6 + 10^9 \times (1) \sim 10^9$

b) A right-deep tree will have the same lattice shape as a left-deep tree. Selinger's algorithm can be extended for only right-deep join trees by reversing the join operator cost calculation at each node considering a join, that is, treating the left expression as the right expression and vice-versa in each join operator cost calculation.

c) Lattice:



For n relations in a bushy join tree, log(n)+1 levels are needed, with each node in level l encapsulating the cost of 2^l relations. In the example above, with n=4, the cost of scans and joins are computed similarly. The difference is that from level 1 to 2, it considers the join at that level with the corresponding pair to get the final relation. Since all the pairs are computed at level 1, the information to compute the cost at level 2 is available.