Handout 8:

1. (344 17SU Final)

a) You are using Twitter to study the patterns of migratory birds. A large number of volunteer citizen scientist from all over the world assist you by tweeting bird sightings using the hashtag #BirdSurvey. The response has been overwhelming and you now have a collection of over 5 million tweets in JSON format each geotagged and with specific information about the number and type of birds seen. The tweets are in a large NoSQL document store where the key is the tweetID and the value is the JSON representation of the tweet. For each of the following MapReduce programs give an explanation of what it calculates. Assume that the function getSightings(tweet) returns a list of (bird,count) pairs within that tweets text.

map(id, tweet):
    total = 0
    for bird, count in getSightings(tweet):
        total += count
    EmitIntermediate(tweet.country, total)
	reduce(key,values):
	    Emit(key, sum(values))

Description of output:

The map function emits key value pairs that are the country of the tweet and total number of bird sightings for that tweet. The reduce function sums up all the sightings for each country.
Result: a mapping of all countries and number of bird sightings.

map(id,tweet):
    EmitIntermediate(tweet.country, tweet.username)
	reduce(key,values):
	    unique_values = set(values)
	    Emit(key, unique_values.length)

Description of output:

The map function emits key value pairs that are the country and username of each tweet. The reduce function counts how many unique users sent in a tweet from each country.
Result: a mapping of all countries and the number of unique users who reported from that country.

b) Write pseudo-code for the map and reduce function below. The output of the reduce function should answer each questions (roughly equivalent SQL is also given).

Find the total number of tweets each username sent.

```
SELECT T.username, count(*)
FROM tweets T
GROUP BY T.username;
```

```
map(id,tweet):
        EmitIntermediate(tweet.username , 1)
# This will emit one tuple for each tweet (K, V) → (username, 1)
```

```
reduce(key,values):
        Emit(key, sum(values))
# Because each tuple has a value of 1 and reduce automatically groups by the key (username) we can
sum the 1's to count the number of tweets
```

2. (344 17WI Final)
Given the following query and statistics:

```
SELECT *
FROM R, S
WHERE R.a = 10 AND R.a = S.b
```

T(R) = 100,000	V(R, a) = 300	min(R.a) = 0	max(R.a) = 1000
T(S) = 40,000	V(S, b) = 20	min(S.b) = 0	max(S.b) = 1000

Assume there are no indexes on R or S. There are 4 nodes (N1, N2, N3, N4), and each node has enough main memory to hold its partition of R and S tuples. We partition R and S using one of the following schemes:

i) block partitioned, with each node holding 25,000 tuples of R and 10,000 tuples of S.
ii) range partitioned in the following way:

N1: 0 <= R.a < 250 (same for S.b)	N2: 250 <= R.a < 500 (same for S.b)
N3: 500 <= R.a < 750 (same for S.b)	N4: 750 <= R.a <= 1000 (same for S.b)

a) Under partition scheme i), how many tuples do you expect each partition to generate if the selection predicate was applied first?
For each node we get:
R: 25,000/300= ~83.3 tuples (from predicate R.a = 10)
S: 10,000 tuples (no selection)

b) Instead of selection, suppose we evaluate the join predicate first. How many tuples do you expect each node to send to other nodes if we use broadcast join on R under partition i)? "broadcast join on R" means R is broadcast to other nodes.
For each node we get:
R: 25,000 tuples sent to each node = 75,000 tuples sent in total (don't send to itself)

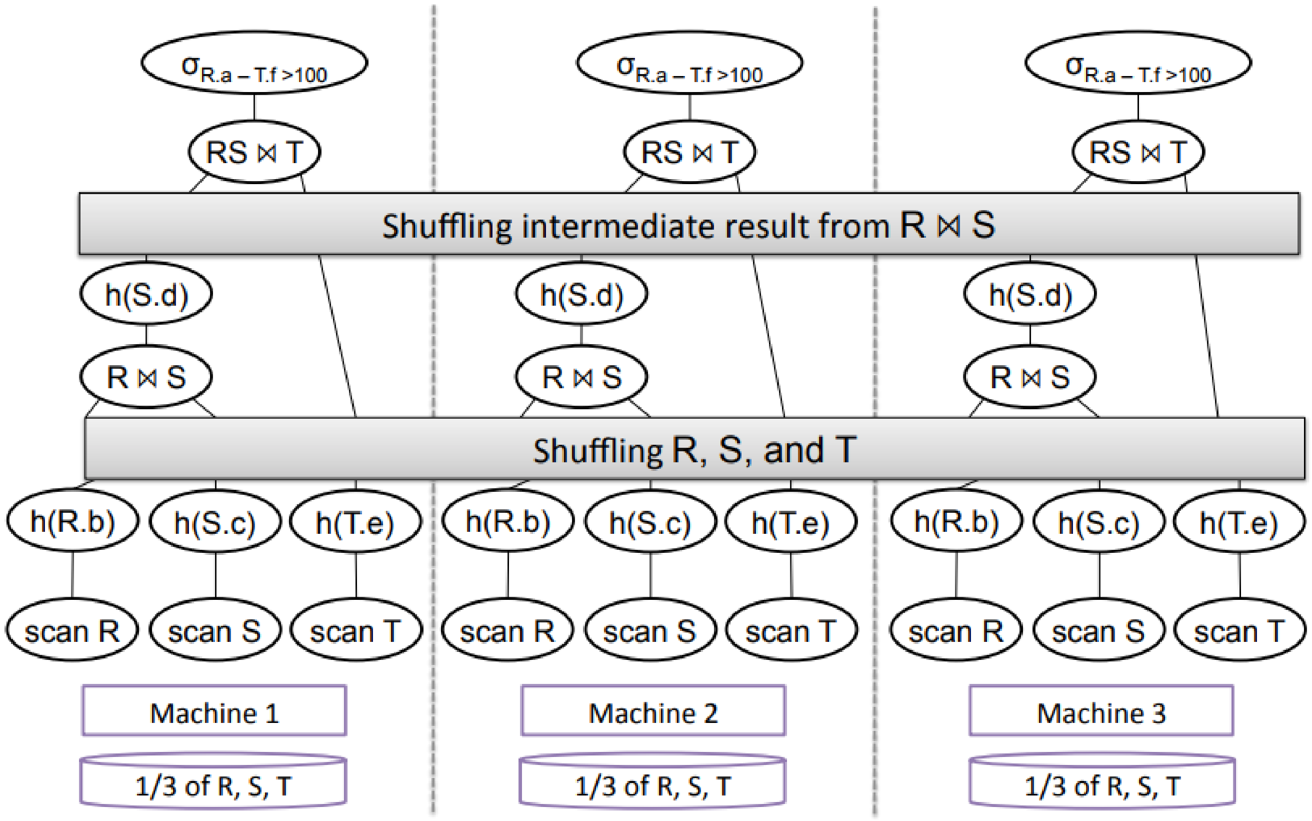
c) Under partition scheme ii), how many tuples do you expect each node to generate if the selection predicate was applied first?
Only node N1 will have the tuples wanted (from predicate R.a = 10).
Therefore only N1 will generate:
R: 100,000/300 = ~333.3 tuples

e) Your roommate comes up with yet another partitioning scheme: block partition R as in i), but replicate S entirely on all nodes. She finds that the query now runs faster even when compared to hash partitioning on R.a and S.b (assume the nodes have enough memory to hold all tuples in either scheme). This is surprising because both schemes need to perform no network I/O for the join, and hash partitioning should read strictly less of S on every node, since S is partitioned instead of replicated. Provide one explanation for how this could occur.
R.a is skewed. Using hash partitioning might end up hashing all R tuples to one node, and the executing local join on that node will make it become the bottleneck. On the other hand S.b being skewed has no effect since under the new scheme S is replicated entirely on all nodes.

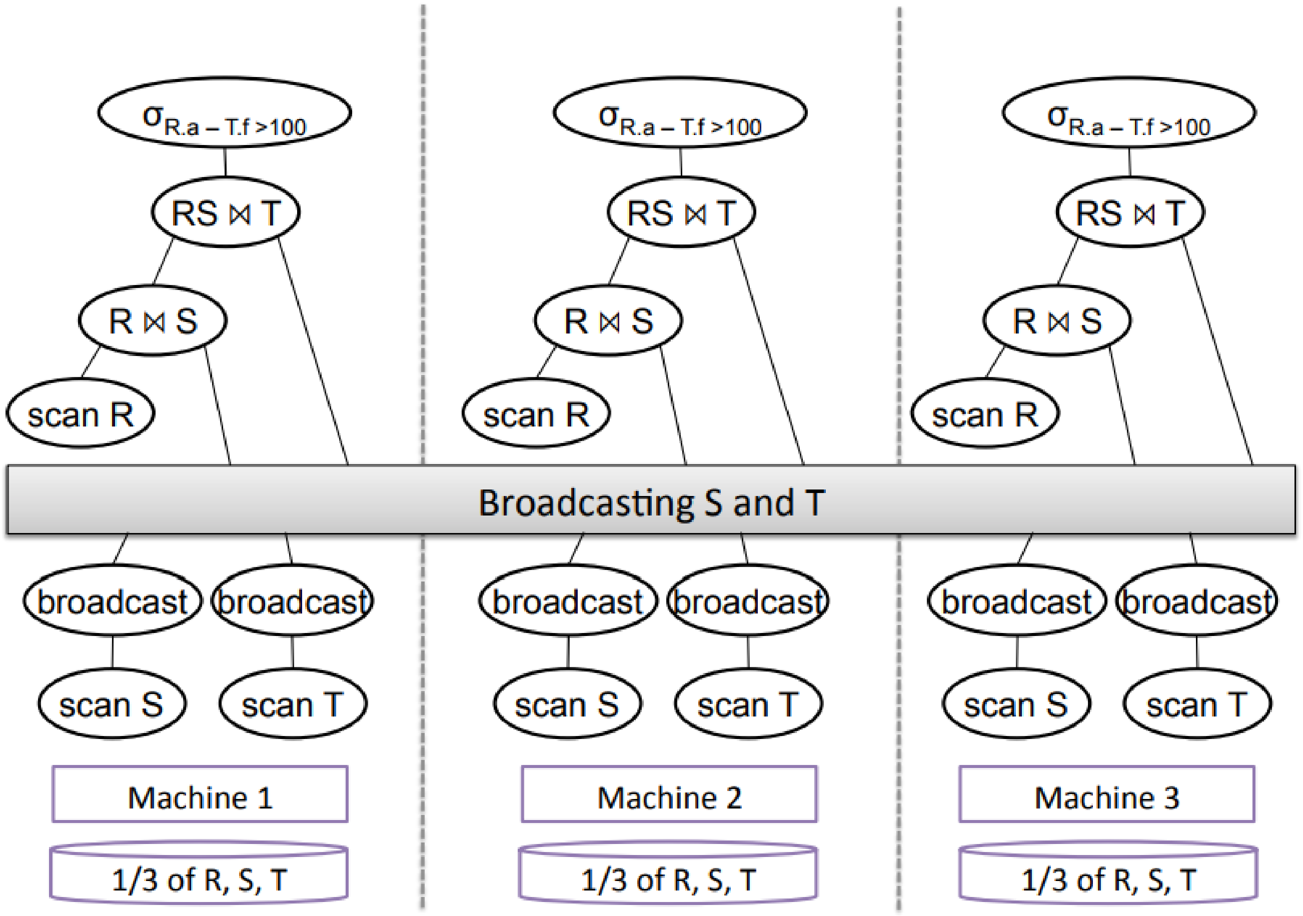
2. (344 15AU Final)

a) Consider relations R(a,b), S(c,d), and T(e,f). All three are horizontally partitioned across N = 3 machines as shown in the diagram below. Each machine locally stores approximately 1 N of the tuples in R, S, and T. The tuples are randomly organized across machines (i.e., R is block partitioned across machines). Show a relational algebra plan for the following query and how it will be executed across the N = 3 machines. Use hash-join (a.k.a shuffle-join) operators. Include operators that need to re-shuffle data and add a note explaining how these operators will re-shuffle that data.

```
SELECT *
FROM R, S, T
WHERE R.b = S.c AND
      S.d = T.e AND
      (R.a - T.f) > 100
```



b) Now consider the case where the R relation is very large and both S and T are very small. Show a plan that uses broadcast joins to compute the result of the query.



c) Explain how the query would be executed in MapReduce. Make sure to specify the computations performed in the map and the reduce functions. Use hash-joins (shuffle joins). No need to give the pseudocode. Just state what each function does and what it outputs in plain text format.

Answer:
Job1:
Map the tuples of R and S into key-value pairs (R.b, (‘R’, R.a, R.b)) and (S.c, (‘S’, S.c, S.d)), respectively
Reduce the tuples by joining the tuples mapped (i.e. take a Cartesian product of all tuples)
Call the output of Job1 RS

Job2:
Map the tuples of RS and T into key-value pairs (S.d, (‘RS’ , R.a, R.b, S.c, S.d)) and (T.e, (‘T’ , T.e, T.f)), respectively
Reduce the tuples by joining the tuples mapped (i.e. take a Cartesian product of all tuples) and also check that the tuple is such that R.a - T.f > 100

Longer explanation:
We will use two MapReduce jobs. The first job will compute the join of R and S.

In the first job, each map task scans either a block of R or a block of S and calls the map function for each tuple. For blocks of R, the map function outputs tuples of the form: key = R.b and value = (‘R’, R.a, R.b). For blocks of S, the map function outputs tuples of the form: key = S.c and value = (‘S’ , S.c, S.d). The MapReduce engine reshuffles the output of the map phase and groups it on the intermediate key, which is the join attribute.

Each reduce task computes the cartesian product of the tuples from the two relations for each group assigned to it (by calling the reduce function) and outputs the final result of the first join.

We now need a second MapReduce job that will perform the second join of RS and T.

In this second job, each map task scans either a block of RS or a block of T and calls the map function for each tuple. For blocks of RS, the map function outputs tuples of the form: key = S.d and value = (‘RS’ , R.a, R.b, S.c, S.d). For blocks of T, the map function outputs tuples of the form: key = T.e and value = (‘T’ , T.e, T.f).

The MapReduce engine reshuffles the output of the map phase and groups it on the intermediate key, which is the join attribute.

Each reduce task computes the cartesian product of the tuples from the two relations for each group assigned to it (by calling the reduce function). The reduce task selects tuples with R.a - T.f > 100 and outputs the final result of the query.