CSE 344 Section 7

1. For each of the following statements, identify whether it is true or false and explain why.

   a. The main reason NoSQL database was introduced was because relational databases did not scale up to a very large number of servers.

True. Relational databases are effective at complex, analytical queries (OLAP), but they are not as effective as NoSQL for running many simple queries at the same time. NoSQL was introduced to deal with computing heavy database loads in parallel.

   b. A Resilient Distributed Dataset (RDD) is another term for a distributed file on disks.

False. An RDD is represented as a series of computations. RDDs are "lazily" evaluated, which means that the queries are not computed until the user calls persist() or collect(). Therefore, RDD's are not files; instead, they are abstract representations of a query result.

   c. Hadoop MapReduce is generally faster than Spark at evaluating queries.

False. MapReduce is primarily different from Spark because it writes the result of intermediate computations to disk. This gives MapReduce an advantage over Spark in handling errors, since it does not need to restart the entire computation if an error occurs. However, it makes MapReduce much slower due since it uses more disk writes.

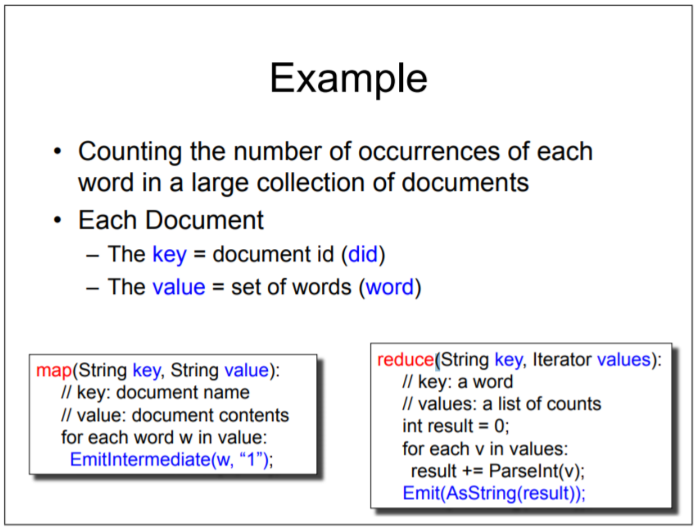
   d. If you can compute a query in parallel on 1000 servers in 15 minutes, then you can always compute it in parallel on 500 servers in at most 30 minutes.

True. Decreasing the number of servers decreases the amount of overhead and network I/O. As a result, halving the number of servers more than halves the execution time.

   e. If you can compute a query in parallel on 500 servers in 30 minutes, then you can always compute it in parallel on 1000 servers in 15 minutes.

False. As the number of servers increases, the gains in efficiency have diminishing returns due to overhead costs and network I/O. As a result, efficiency is sub-linear with the number of servers, so doubling the number of servers does not quite halve the execution time.

The following slide gives an example of the map and reduce functions to group by and aggregate on count:



Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
     EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
     result += ParseInt(v);
   Emit(AsString(result));
```

Use pseudo-code to write the map and reduce functions.

2. Convert the following SQL queries to MapReduce:

   a.

```
Students(name, major, credits)

select s.name, max(s.credits)
from Students s
where s.major == "CSE"
group by s.name;

map(Tuple s):
      // write implementation here
      if s.major == "CSE":
            EmitIntermediate(s.name, s.credits)


// key is name, credits is list of values of credit from tuples which
// have the given name
reduce(String key, List<int> credits):
      // write implementation here
      int current_max = 0
      for each c in credits:
            current_max = max(c, current_max)
      Emit(key, current_max)
```

b.

```
Devices(model, company, height, width)

select model, avg(height * width)
from Devices
where company == “Microsoft”
and height <> width;

map(Tuple p):
   // write implementation here
   if company == “Microsoft” and height != width:
      EmitIntermediate(name, {height * width, 1})




// key is the model, values is tuples containing (height * width, 1)
reduce(String key, List<double, int> values):
   // write implementation here
   total_sum = 0, total_count = 0

   for sum, count in values:
      total_sum += sum
      total_count += count

   Emit(key, total_sum / total_count)
```

c. (final, autumn-17)

```
Product(category, price, quantity)

select p.category, count(*)
from Product p
where p.price > 100
group by p.category
having sum(p.quantity) > 200;


map(Tuple p):
      // write implementation here
      if p.price > 100:
            EmitIntermediate(p.category, p.quantity)



// key is the category, values is the list of quantities having that
// category
reduce(String key, List<int> quantities):
      // write implementation here
      total_quantity = 0, r = 0

      for quantity in quantities:
            total_quantity += quantity
            r ++

      if total_quantity > 200:
            Emit(key, r)
```