CptS 415 Big Data
Assignment 3
Jinyang Ruan
011696096

1. Parallel Data Models
   a. What is speedup and scaleup? Give three reasons why we cannot do better than linear speedup.
      Speedup and Scaleup are the degrees of parallelism.
      Specifically, speedup is to use more resources to a fixed number of tasks to proportional reduce the time. Scaleup is a factor that expresses how much more work can be done in the same period by a larger system.
      Reasons:
      1. It has startup costs: initializing each process.
      2. It has some interferences from others: competing for shared resources (network, disk, memory, or locks).
      3. Skew: it is difficult to divide a task into exactly equal-sized parts; the response time (latency) is determined by the largest part.
   b. Describe and compare the pros and cons of the three architectures for parallel systems.
      1. Shared Memory
         Pros: easy to program; efficient communication via data in memory, accessible by all.
         Cons: shared memory and network become bottleneck, and not scalable beyond 32/64 processors.
      2. Shared Disk
         Pros: good fault tolerance, if a processor fails, the other can take over since the database is resident on disk.
         Scalability of shared disk is better than which of shared memory, that is, memory is no longer a bottleneck.
         Cons: disk subsystem is a bottleneck; all I/O to go through a single network; not scalable beyond a couple of hundred processors.
      3. Shared Nothing (network)
         Pros: cheap to build; easy to scaleup.
         Cons: hard to program.

2. MapReduce
   a. Facebook updates the “common friends” of you and response to hundreds of millions of requests every day. The friendship information is stored as a pair (Person, [List of Friends]) for every user in the social network. Write a MapReduce program to return a dictionary of common friends of the form ((User i, User j), [List of Common Friends of User i and User j]) for all pairs of i and j who are friends. The order of i and j you returned should be the same as the lexicographical order of their names. You need to give the pseudo-code of 1 main function, and 1 Map () and 1 Reduce () function. Specify the key/value pair and their semantics (what are they referring to?).

Set a list <key1, value1> of key-value pairs where key1 is a person and value1 is a list of associate friends of that person.

```
void main () {
Input <key1, value1> //e.g. (P1, P2 P3 P4 P5) (P2, P1 P3 P6 P7)
Map (key1, value1);
Reduce (k2, list(v2));
Combine all the data by implementing sort algorithm based on the ASCII code of the key in order to return in the lexicographical order;
}

void Map () {
    // For all person in the list, do
            Compute key-value pairs <k2, v2>,
            The intermediate key-value pairs are hash-partitioned based on k2,
            Each partition (k2, list(v2)) is sent to a reducer
    }

void Reduce () {
    take a partition (k2, list(v2)) as input and compute key-value pairs <k3, v3>
    }
```

b. Most frequent keyword. Search engine companies like Google maintains hot webpages in a set R for keyword search. Each record $r \in R$ is an article, stored as a sequence of keywords. Write a MapReduce program to report the most frequent keyword appeared in the webpages in R. Give the pseudo-code of your MR program.

First round:
Splitting set r on spaces,
For all words in r, do count words,
The output is a list <key1, value1> of where key1 is the word and value1 is the number of the word.
Second round:

```
Map () {
    Applied to each pair (key1, value1), computes key-value pairs <k2, v2>,
    The intermediate key-value pairs are hash -partitioned based on k2,
    Each partition (k2, list(v2)) is sent to a reducer
}
Reduce () {
    takes a partition as input and computes key-value pairs <k3, v3>
}
```

Sort the MR output list <k3, v3> by the value (number) of the words in the decreasing order.
Find the most frequent keyword in R.

3. Apache Spark
    a. Explain the definition of RDD and how the lineage retrieval work.
    RDD is Resilient Distributed Datasets, which is a read-only, in-memory partitioned collection of records. A dataset in RDD format is divided into logical partitions to be computed on different nodes on the cluster. RDDs can contain R, Python, Java, or Scala objects as well as user-defined classes. It can be converted into a spark data frame and vice versa.

    b. List the reasons why Spark can be faster than MapReduce.
    Spark can be faster than MapReduce because:
        1. Fast data processing. In-memory processing makes Spark faster than Hadoop MapReduce – up to 100 times for data in RAM and up to 10 times for data in storage.
        2. Iterative processing. If the task is to process data again and again – Spark defeats Hadoop MapReduce. Spark's Resilient Distributed Datasets (RDDs) enable multiple map operations in memory, while Hadoop MapReduce has to write interim results to a disk.
        3. Near real-time processing. If a business needs immediate insights, then they should opt for Spark and its in-memory processing.
        4. Graph processing. Spark's computational model is good for iterative computations that are typical in graph processing. And Apache Spark has GraphX – an API for graph computation.
        5. Machine learning. Spark has MLlib – a built-in machine learning library, while Hadoop needs a third-party to provide it. MLlib has out-of-the-box algorithms that also run in memory. But if required, our Spark specialists will tune and adjust them to tailor to your needs.
        6. Joining datasets. Due to its speed, Spark can create all combinations faster, though Hadoop may be better if joining of very large data sets that requires a lot of shuffling and sorting is needed.

    c. Explain the definitions of narrow dependencies and wide dependencies. In addition, explain how Spark determines the boundary of each stage in a DAG and why put operators into stages will improve the performance.
        - Narrow dependency is that the RDD partition is only used by one sub-partition. Computations of transformations with this kind of dependency are rather fast as they do not require any data shuffling over the cluster network.
        - Wide dependency is that the RDD partition is used by multiple sub-partitions. the computation speed might be significantly affected as we might need to shuffle data around different nodes when creating new partitions.
        - DAG is defined by Spark as a lineage graph of RDDs which represents the data distributed across different nodes. Computations in Spark are represented by DAG and the DAGScheduler in the Spark scheduling layer incorporates a stage-oriented scheduling. It converts a logical execution plan into a physical execution plan based on several stages. A stage is, in simple words, a step in the physical execution plan. In other words, each job which gets divided into

smaller sets of tasks is a stage and that is how the boundary for each stage is determined.

- Putting the operators into stages improves the overall performance because plans are executed in stages rather than performing them all at once. The data doesn't have to be copied along the way, neither all the operators. Taking the help of lineage retrieval and the DAG, Spark can reinitiate a task automatically from a specific point or stage of its whole execution plan rather than starting from the very beginning. This not only helps save disk space but also saves some computation which eventually leads to more available memory for computation. That is why staging and keeping the operators in stages improves the performance.