Hadoop: ReduceMap (Processing Big Data)

Notes by Mannan Ul Haq (22L-7556)

MapReduce is a programming model and processing framework for distributed computing on large datasets. It is a core component of the Apache Hadoop project and is widely used for parallel processing of data across a cluster of computers.

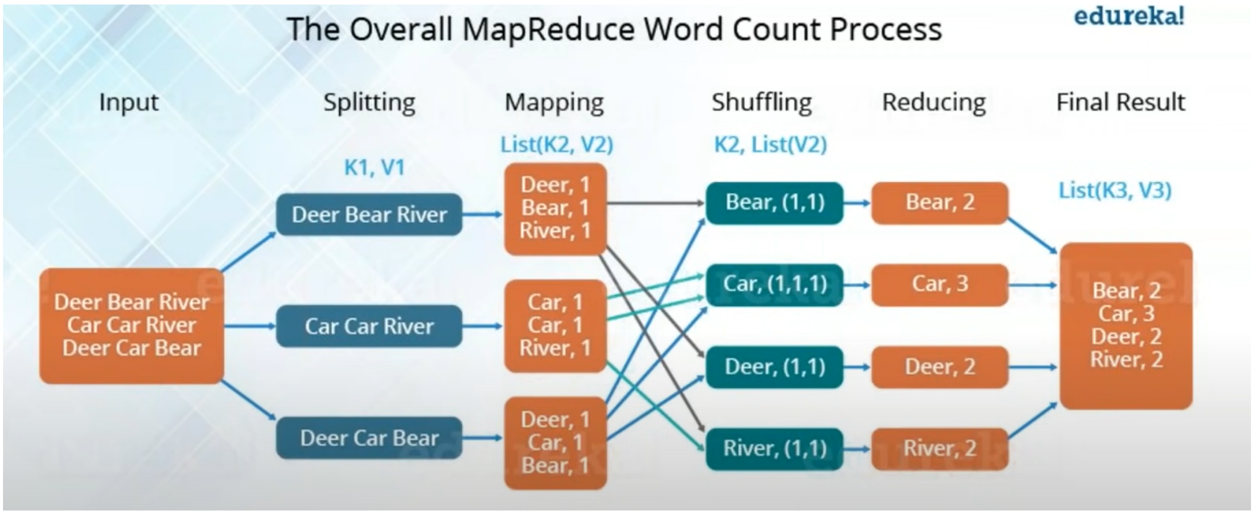
The MapReduce program works in two phases, namely:

- Map
- Reduce

Map tasks deal with the **splitting** and **mapping** of data. Reduce tasks **shuffle** and **reduce** the data (Aggregate, summarize, filter or transform).

Here's a breakdown of what MapReduce entails:

1. **Map Phase:** In the Map phase, the input data is divided into smaller chunks, and each chunk is processed independently by multiple Mapper tasks in parallel. The Mapper tasks produce intermediate key-value pairs as output.
2. **Shuffle and Sort:** After the Map phase, the intermediate key-value pairs are shuffled and sorted based on their keys. This process ensures that all values associated with the same key are grouped together and passed to the same Reducer task.
3. **Reduce Phase:** In the Reduce phase, the shuffled and sorted intermediate data is processed by Reducer tasks. Each Reducer receives a subset of the intermediate data with the same key and performs aggregation or computation on these values to produce the final output.
4. **Output:** The output of the Reducer tasks is typically written to a distributed file system, such as Hadoop Distributed File System (HDFS).



Job Tracker and Task Tracker:

JobTracker and TaskTracker are key components responsible for managing and coordinating the execution of MapReduce jobs across a cluster of machines. Here's an overview of each:

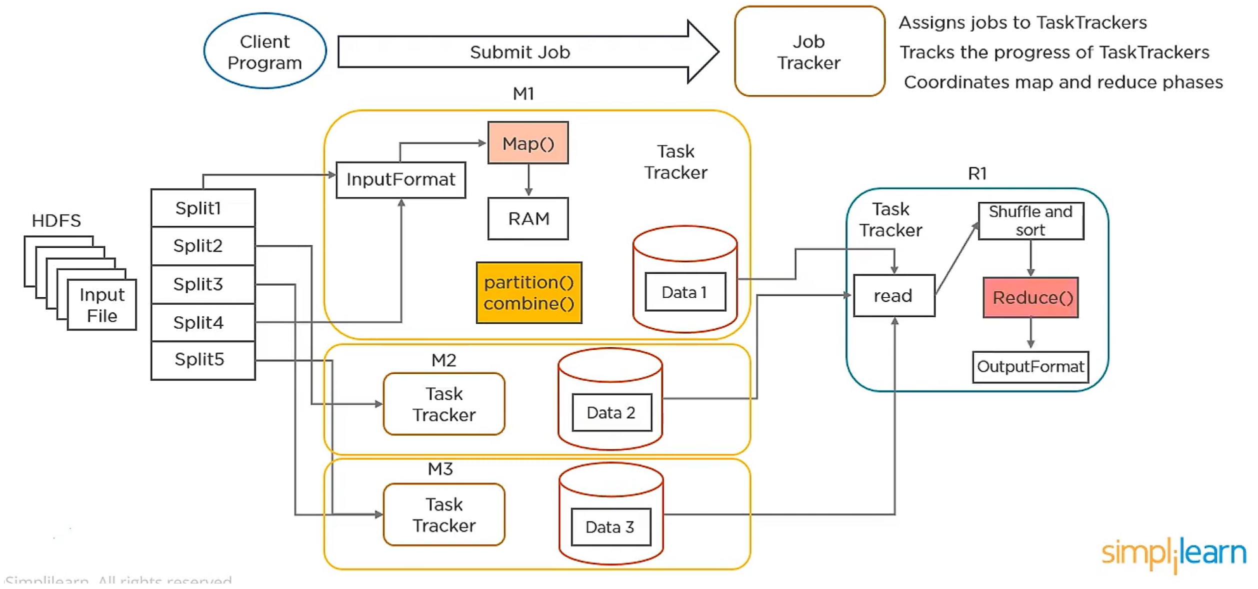
JobTracker:

The JobTracker is the central component of the MapReduce framework. It is responsible for accepting job submissions, scheduling tasks, monitoring their execution, and managing the overall workflow of MapReduce jobs. The JobTracker schedules MapReduce jobs onto available TaskTrackers (worker nodes) in the cluster.

TaskTracker:

The TaskTracker is a worker node component that runs on each machine in the Hadoop cluster. It is responsible for executing MapReduce tasks assigned by the JobTracker and reporting their status back to the JobTracker. The TaskTracker runs Map, Reduce, or other specialized tasks as directed by the JobTracker. It launches and monitors individual task attempts, handling task execution failures and retries.

The TaskTracker periodically sends **heartbeats** to the JobTracker to indicate that it is alive and functioning. These heartbeats also contain status updates on the tasks being executed, allowing the JobTracker to monitor progress and detect failures.



```
# MapReduce Code without MRJob

import sys

# Mapper function to split each line into words and emit (word, 1) pairs.
def Mapper():
    for line in sys.stdin:
        # Split the line into words
        words = line.strip().split()

        # Emit (word, 1) pairs for each word
        for word in words:
            print(f"{word.lower()}, 1")

# It reads mapper outputs and sorts them by key
def Shuffler():
    # Read all lines from stdin and split them into key-value pairs
    mapper_output = [line.strip().split(", ") for line in sys.stdin]
    # Sort the key-value pairs by key
    sorted_mapper_output = sorted(mapper_output, key = lambda x: x[0])

    for key, value in sorted_mapper_output:
        print(f"{key}, {value}")

# The reducer function to aggregate counts for each word
def Reducer():
    current_word = None
    word_count = 0

    for line in sys.stdin:
        # Split the input line into word and count
        word, count = line.strip().split(", ")
        count = int(count)

        # If the word is the same as the current word, add the count to the tota
        if current_word == word:
            word_count += count
        else:
            # If the word is different, emit the total count for the previous wo
            if current_word:
                print(f"{current_word}, {word_count}")
            # Update the current word and reset the count
            current_word = word
            word_count = count

    # Emit the total count for the last word
    if current_word:
        print(f"{current_word}, {word_count}")
```

```
# MapReduce Code with MRJob

from mrjob.job import MRJob
from mrjob.step import MRStep

class WordCount(MRJob):
   def mapper(self, _, line):
      for word in line.split():
         yield(word, 1)

   def combiner(self, word, counts):
      yield(word, sum(counts))

   def reducer(self, word, counts):
      yield(word, sum(counts))

   def steps(self):
      return [
          MRStep(mapper=self.mapper, reducer=self.reducer)
      ]

if __name__ == "__main__":
   WordCount.run()
```

Example of K-Means Clustering:

```
import math
from mrjob.job import MRJob
import numpy as np

class KMeansMRJob(MRJob):

    centroids = []

    def configure_args(self):
        # Add a command-line argument to specify the file containing cluster cen
        super(KMeansMRJob, self).configure_args()
        self.add_file_arg('--Clusters')

    def mapper_init(self):
        # Load cluster centroids from the file specified in the command-line arg
        self.centroids = []
        f = open(self.options.Clusters, 'r')
        for line in f:
          centroid = [float(x) for x in line.strip().split(", ")]
          self.centroids.append(centroid)
        f.close()
    def mapper(self, _, line):
        # Map input data points to their nearest cluster centroid
        data_point = [float(x) for x in line.strip().split(",")]

        # Calculate distances to each centroid
        distances = [math.dist(data_point, centroid) for centroid in self.centro

        # Find the index of the nearest centroid
        nearest_index = distances.index(min(distances)) + 1

        yield nearest_index, data_point

    def reducer(self, cluster_id, data_points):
        # Reduce function to calculate new centroid for the cluster
        cluster_points = list(data_points)
        new_centroid = np.mean(cluster_points, axis = 0)

        yield list(new_centroid), cluster_points

if __name__ == '__main__':
    KMeansMRJob.run()
```