Cloud Computing Assignment 4

MapReduce

Introduction

This report documents the steps followed to implement several MapReduce programs to process large datasets using Amazon Elastic MapReduce (EMR). The main objectives were to read given .txt files and extract information from them, such as sorting words based on their frequencies, or analyzing domain popularity among clients. The tasks involved working with Hadoop and Java to build scalable solutions for data processing.

Before diving into the specific tasks, several preparatory steps were necessary. I started by creating an Amazon S3 bucket to store input files and output files from the MapReduce jobs. I then created an EMR cluster through the AWS Management Console. The cluster configuration included choosing instance type as m4.large and the number of instances as 3. Initially, I attempted to set up Hadoop on my local machine for testing MapReduce programs. However, this process encountered several issues, including JAVA_HOME configuration problems. As a workaround, I used AWS Cloud9 to compile my Java code into a JAR file. In the Cloud9 environment, I wrote the Java code for the MapReduce tasks and compiled it using the Hadoop libraries available in the cloud environment. After compiling, I created a JAR file from the Java code, which was then uploaded to the S3 bucket. The JAR file was used as input for the steps in the EMR cluster to execute the MapReduce programs. Now, I will explain the architecture of my code for each task.

Task 1.1

In this task, I implemented a program sorting the words by their frequencies given a txt file. Since the professor mentioned that we do not have to write our own codes, I searched and found a java program online. It has two classes: TokenizerMapper and IntSumReduce. The first class uses a mapper function that takes each line of the input text and tokenizes it using StringTokenizer. Then it converts each word to lowercase to ensure case insensitivity. As an output, for every occurrence of a word, it emits the word as the key and 1 as the value (<word, 1>). Then the second class uses the reducer function to receive the output of mapper function and sums up the counts for each word to get the total occurrence, then it finally stores each

word and its total count in the countMap. At the end, cleanup function sorts the map in decreasing order and writes the top 100 most frequent words with their frequencies. As a result, I had 5 output files in my folder, indicating that 5 reducers were used to process the data. The top 10 most frequent words in the file are given in Table 1 below.

ja	3007190	**mutta**	16911
on	33266	**vuoden**	15317
joka	26957	**ei**	15191
sekä	20157	**että**	13309
se	17547	**tai**	12893

Table 1. Top 10 Most Frequent Words

Task 1.2

This task was the same as the first one except there was an additional combiner function. The combiner function in this program basically takes the output from the mapper and performs a local reduction. In this case, instead of sending multiple <word, 1> pairs for the same word to the reducer, the combiner aggregates these counts locally and sends these fewer but larger values to the reducer. Therefore, we can say that using a combiner function reduces the data transfer between mappers and reducers. Considering the fact that task 1.1 took 205 seconds while task 1.2 took 160 seconds, we can say that the overall time for the MapReduce decreases when a combiner is used. As a result, I got the same output as I got from task 1.1.

Task 1.3

In this task, I was asked to count the number of words of length 3 and 5. The mapper function here has two different keys for the words having 3 and 5 letters. While 3-letter words are kept in length3Key, 5-letter words are kept in length5Key. Similar to the previous tasks, mapper function returns these words in the following format: <length_3,1> or <length_5,1>. Reducer function then gets and sums up these values associated with each key. The output of this program is given in Table 2 below.

length_3	4579897
length_5	7428863

Table 2. Number of words having 3 or 5 letters

Task 2.1

In this task, I dealt with a file containing information about 3.4 million requests to an Apache web server. The task was to calculate the total CDN costs based on the number of served requests and the total volume of transferred data. For this task, only two things were important: number of lines in the file (number of requests) and the last numerical value of each line (volume of the data for each request). Therefore, the mapper function here emits two key-value pairs: <request,1> and <dataTransferred, volume>. Then, the reducer aggregates the counts and multiplies it by 0.001. Additionally, it sums up the total volume in bytes and divides it by 1024^3 to convert it to gigabytes. As an output, it writes the total number of requests, total data transferred in GB and the total CDN costs in euros. Table 3 below shows the output of this task.

num_of_requests	3461612.0
transferred_data (GB)	61.0242736665532
cost (EUR)	3466.4939418933245

Table 3. Total CDN costs

Task 2.2

This task was similar to the first task in terms of the requested output. Given the file containing requests, I was asked to return the top 5 most frequent domain names. Mapper function in this program extracts the first field of each file. It then uses a regex pattern to identify IP addresses. Then, if the extracted field is not an IP address, it emits it as <domain, 1>. Reducer function aggregates values for each domain name and stores them in the countMap with their frequencies. The cleanup function then sorts the countMap in decreasing order and finally selects the first 5 elements using the method limit(5). The output of this program is given in Table 4 below.

alyssa.prodigy.com	7780
siltb10.orl.mmc.com	3988
piweba4y.prodigy.com	3636
www-d4.proxy.aol.com	3147
bill.ksc.nasa.gov	3100

Table 4. Top 5 Most Frequent Domain Names

Comment on the Obtained Results and Performance

I implemented Python programs and tested each task locally. As a result, I saw that all the MapReduce programs provided accurate outputs. The use of combiners in Task 1.2 showed a clear improvement in performance by reducing the amount of data transferred between phases as well as the decrease in the time it takes to achieve the task. The EMR cluster allowed for distributed processing, making it feasible to handle large datasets that would be slow or impractical to process locally. For this assignment, my local Python programs and MapReduce programs did not have very different runtimes; however, I believe that as datasets get bigger, MapReduce is more likely to overperform. I would say my programs worked effectively, but further optimizations could be made by adding more instances or trying different instance types in the EMR cluster for better cost-performance balance. Additionally, the number of reducers could help balance between job duration and output file management.

Reflection

Everything I did in this assignment was completely new to me; so I learned about the buckets, clusters and using MapReduce programs. I also discovered how to use AWS Cloud9 as a cloud-based IDE for compiling Java programs and managing Hadoop-related tasks. In general, I believe this assignment was relatively easier. The only problem I faced was that I could not create a .jar file locally. There was a problem with my system environment variables, so I could not use Hadoop in my local computer. Instead, I used Cloud9 platform and uploaded my java files there. The rest of the assignment was satisfying.