IT00CE11-3005

Cloud Computing

Assignment # 4

October 08, 2024

MapReduce | STUDENT NO-2402262

Saad Abdullah

saad.abdullah@abo.fi

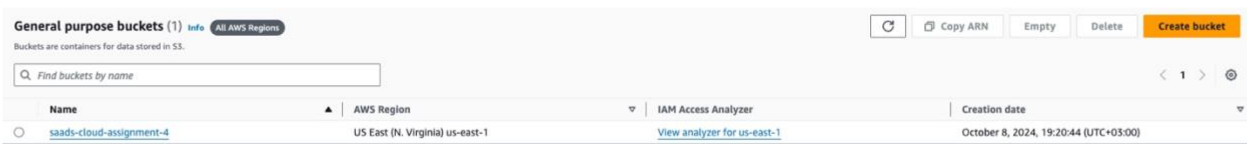
Table of Contents

MapReduce

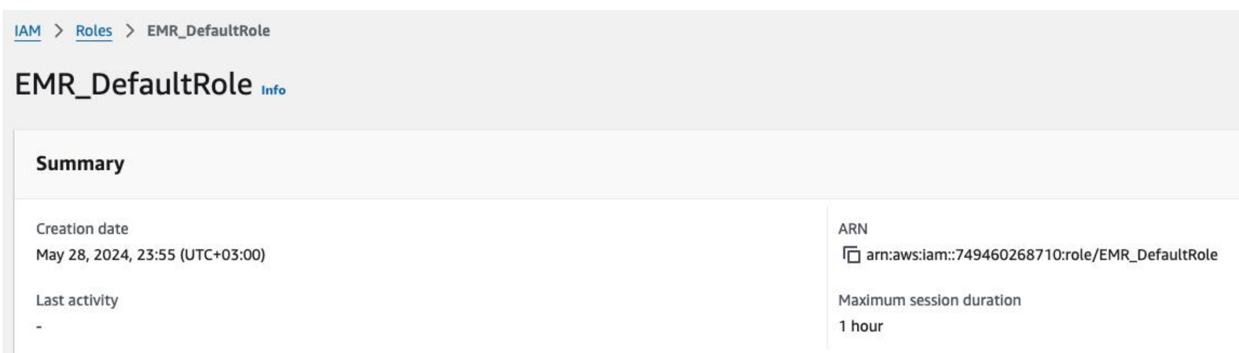
Prerequisite: Choosing instance type and S3 setup:

For the prerequisites of the task, the first step was selecting the appropriate instance type. I opted for m5.xlarge instances in Amazon EMR. This decision was made to ensure faster performance since the m5.xlarge belongs to one of the latest families of m instances, offering balanced compute, memory, and networking resources, which would enhance the speed of our MapReduce operations given the large dataset size of around 700MB.

Next, I set up an S3 bucket to store the input and output files. As part of the bucket setup, I implemented a strict security policy, ensuring that only the EMR default user could access the bucket. Public access was blocked to safeguard the dataset and output from being accessed by unauthorized users. This setup was done to get more hands-on experience with S3. Below is an image of the bucket I used.



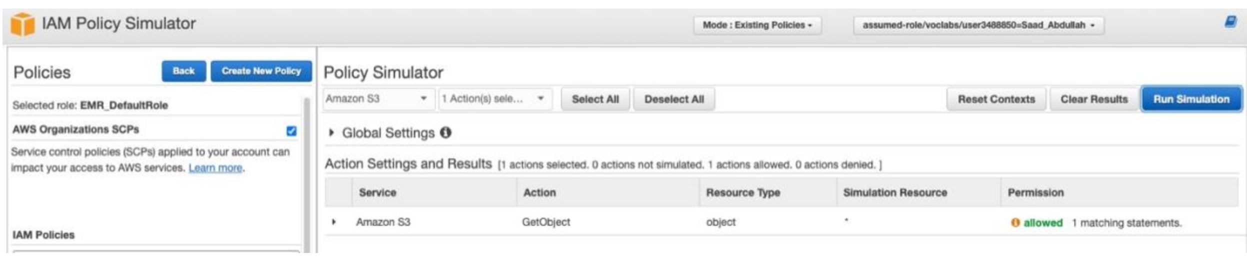
I just wanted to give access to EMR_DefaultRole, which is a premade role in IAM console. I will use this role for cluster creation so that only each node on the cluster has access to S3, and all other access is blocked.



Below is the snapshot of the policy I used.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::749460268710:role/EMR_DefaultRole"
      },
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::saads-cloud-assignment-4",
        "arn:aws:s3:::saads-cloud-assignment-4/*"
      ]
    }
  ]
}
```

Now it was time to test this policy before creating an EMR cluster, so I did it using the IAM policy simulator provided by AWS.



Now the bucket is ready for usage, I uploaded the first dataset.

S3 URI

s3://saads-cloud-assignment-4/datasets/fiwiki-latest-pages-articles_preprocessed.txt

Amazon Resource Name (ARN)

arn:aws:s3:::saads-cloud-assignment-4/datasets/fiwiki-latest-pages-articles_preprocessed.txt

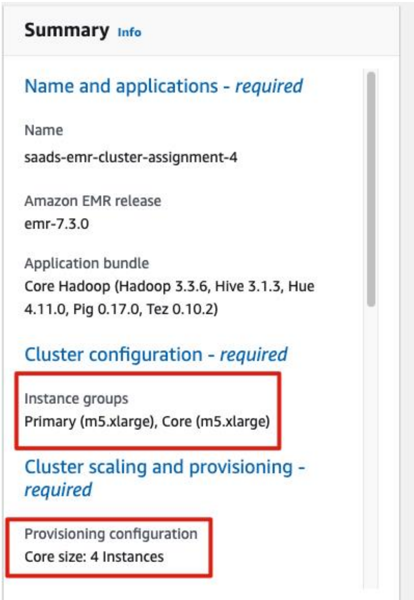
Entity tag (Etag)

745245a4ecada8610e3c3af1a37dbd25-44

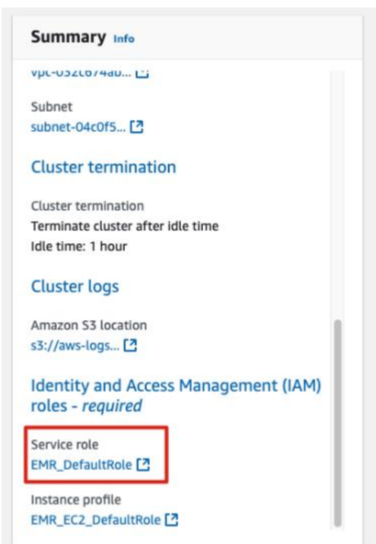
Object URL

https://saads-cloud-assignment-4.s3.amazonaws.com/datasets/fiwiki-latest-pages-articles_preprocessed.tx
t

The next step was to create an EMR cluster, so I went to the EMR console created a cluster there using m5.xlarge instances and I chose Core Hadoop as an Application bundle instead of Spark Interactive because our main goal was MapReduce. The rest of the configurations were set to default.



And as can be seen below I have chosen a role so that the instances can access the s3 bucket.



Since our cluster is ready now, the next thing is to add jobs to the cluster for execution

Just a note that for each of the tasks below, I first wrote a program in Python and ran it locally then I converted it into MapReduce architecture. This helped me to cross verify my results.

Problem 1: Word Counting.

Once the infrastructure was set up, I moved on to implementing the MapReduce programs as described in the assignment.

Task 1.1 A MapReduce program counting from an input file the total number of words and providing as output the 100 most frequent words in decreasing order.

Solution Explanation: The mapper reads lines of text, splits them into words, and outputs each word with a count of 1 in the format word\t1. The reducer aggregates these counts using a defaultdict, handling any parsing errors. It then sorts the word counts in descending order and outputs the top 100 most frequent words with their counts.

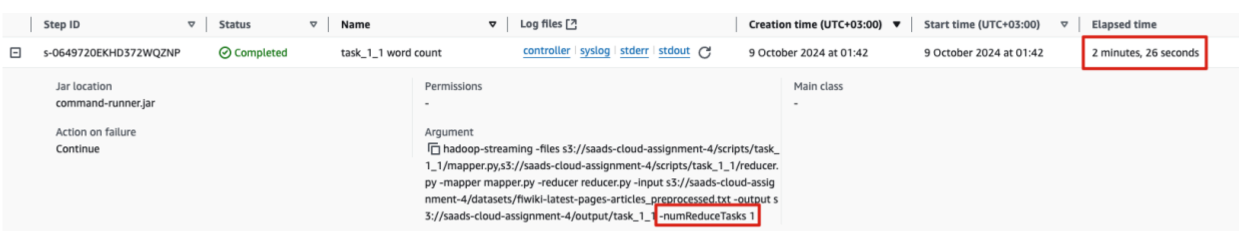
I tried to solve this problem using various architectures which are stated as follows:

1) Using Multiple Mappers & Reducers on an EMR with 1 master and 4 slaves.

The task was done in 1 minute and 24 seconds but it created multiple reducer files (this is default behavior) and for this assignment, I wanted to create a solution in which reducers can only yield 1 output file.

2) Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 4 slaves.

This task took more time (2 minutes and 26 seconds) as compared to the previous one but there was no hassle required to aggregate the results since there was only 1 output file.



3) Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 6 slaves.

It was astonishing for me that even though I added more slaves, still this approach took more time than with 4 slaves. Maybe the bottleneck here was the number of reducers.

Step ID	Status	Name	Log files	Creation time (UTC+03:00)	Start time (UTC+03:00)	Elapsed time
s-09289133C23GY23GFRRQ	Completed	task_1_1 word count with 6 instances	controller | syslog | stderr | stdout	12 October 2024 at 01:19	12 October 2024 at 01:25	2 minutes, 32 seconds

Jar location
command-runner.jar

Action on failure
Continue

Permissions
-

Argument
hadoop-streaming -files s3://saads-cloud-assignment-4/scripts/task_1_1/mapper.py,s3://saads-cloud-assignment-4/scripts/task_1_1/reducer.py -mapper mapper.py -reducer reducer.py -input s3://saads-cloud-assignment-4/datasets/fiwiki-latest-pages-articles_preprocessed.txt -output s3://saads-cloud-assignment-4/output/task_1_1_6_instances -numReduceTasks 1

Main class
-

Below are the snapshots for this setup

Public IPv4 address	Elastic IP	IPv6 IPs	Monitoring	Security group name
100.28.219.63	–	–	disabled	ElasticMapReduce-slave
44.200.165.227	–	–	disabled	ElasticMapReduce-slave
3.236.47.111	–	–	disabled	ElasticMapReduce-slave
44.221.44.9	–	–	disabled	ElasticMapReduce-slave
18.207.243.175	–	–	disabled	ElasticMapReduce-slave
100.27.233.5	–	–	disabled	ElasticMapReduce-slave
44.200.101.3	–	–	disabled	ElasticMapReduce-master

saads-emr-cluster-assignment-4

Summary

Cluster info

Cluster ID
j-31PZXG92QPP8F

Cluster configuration
Instance groups

Capacity
1 Primary | 6 Core | 0 Task

Applications

Amazon EMR version
emr-7.3.0

Installed applications
Hadoop 3.3.6, Hive 3.1.3, Hue 4.11.0, Pig 0.17.0, Tez 0.10.2

In conclusion, different tests took different times but the output was consistent which was:

```
part-00000
ja      3006907
on      1842718
oli     869419
vuonna  457522
myös    413566
hän     408331
joka    354372
Hän     320241
ovat    275556
sekä    264154
ei      255011
mutta   249336
että    219925
tai     200884
se      190972
sen     189589
Vuonna  153198
jälkeen 150047
jonka   145904
vuoden  139018
jossa   131692
hänen   130517
sai     129079
mukaan  121614
Se      118498
olivat  118179
noin    118099
Suomen  117067
muun    112411
muassa  112063
```

I cross-verified the output as:

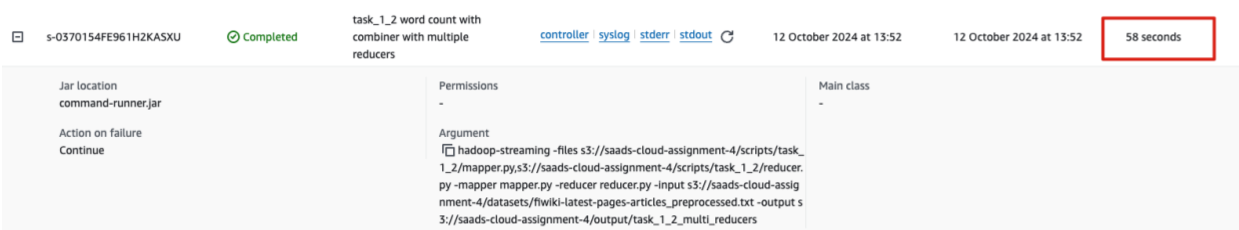
```
[1]: from collections import Counter

     with open('fiwiki-latest-pages-articles_preprocessed.txt', 'r', encoding='utf-8') as f:
         text = f.read()

     words = text.split()
     word_count = Counter(words)
     top_100 = word_count.most_common(100)

     for word, count in top_100:
         print(f"{word}\t{count}")

     ja      3006907
     on      1842718
     oli     869419
     vuonna  457522
     myös    413566
     hän     408331
     joka    354372
     Hän     320241
     ovat    275556
     sekä    264154
```

Task 1.2 An extension of the previous program implementing a combiner in the map function.

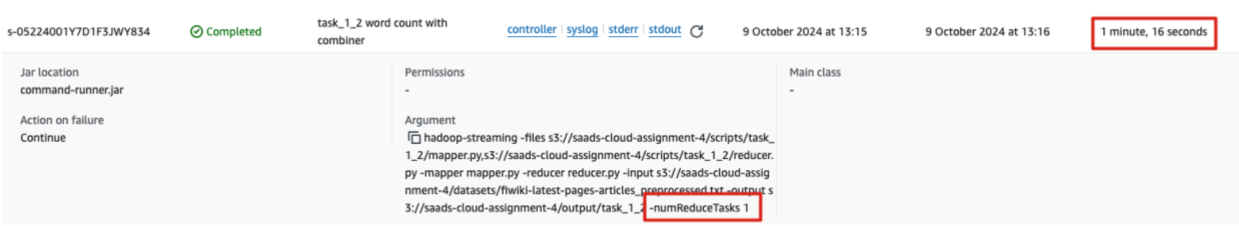
Solution Explanation: The mapper is extended to include a combiner. It processes each line, splitting it into words and incrementing the count for each word locally before emitting the partial aggregated results. This reduces the amount of data shuffled to the reducer. The reducer works similarly to Task 1.1, aggregating word counts from all mappers and sorting the results to output the top 100 most frequent words. The combiner reduces data transfer and improves overall performance by handling some aggregation in the map phase itself.

The results were promising. There was a good difference in performance. Following are the architectures I used:

1) **Using Multiple Mappers & Multiple Reducer on an EMR with 1 master and 4 slaves.**
   Multiple reducers increased the performance here. (58 Seconds)



2) **Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 4 slaves.**
   1 reducer saved the time for aggregation but it was time consuming (1 minute and 16 Seconds)

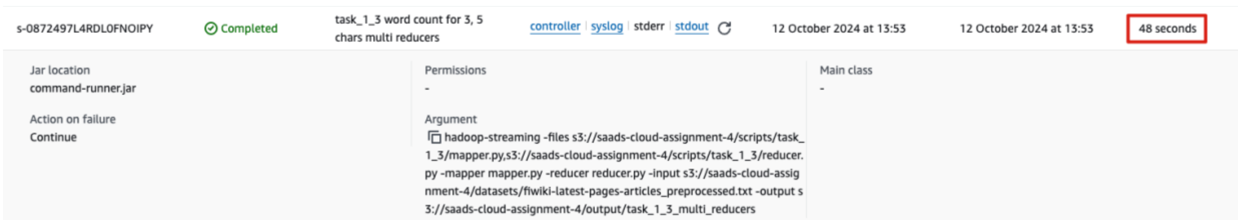


Task 1.3 A MapReduce program counting the number of words of length 3 and 5 (i.e., how many words having 3 and 5 characters does the input file contain).

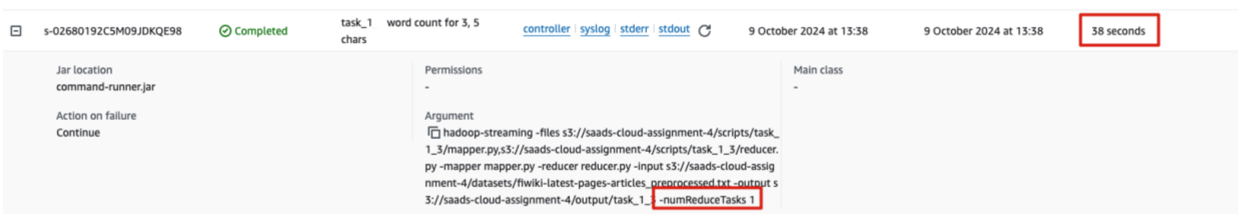
Solution Explanation: The mapper reads input line by line, splitting it into words and counting the occurrences of words that are exactly 3 or 5 characters long. The results are locally aggregated and output as word lengths with their corresponding counts. The reducer then sums the counts for words of these specific lengths from all mappers, outputting the total number of words with lengths 3 and 5.

For this task, an EMR with 4 slaves was used and architectures are as follows:

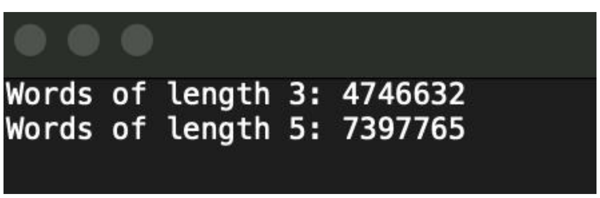
1) **Using Multiple Mappers & Multiple Reducer on an EMR with 1 master and 4 slaves.**
   Results were astonishing here as multiple reducers (48 seconds) even took more time than 1 reducer (38 seconds).



2) Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 4 slaves.



The output was as follows:



Conclusion:

In summary, the MapReduce tasks produced consistent results, with performance varying based on the EMR cluster configuration. Task 1.1 showed a trade-off between speed and simplicity when using multiple vs. single reducers. Task 1.2 benefited from the combiner, improving performance by reducing data shuffling. Task 1.3, which focused on counting words of specific lengths, was executed efficiently. The m5.xlarge instances offered a good balance of speed and resource efficiency. I think choosing multiple reducer or one reducer will vary from problem domain to problem domain and it surely will have an impact on performance.

Problem 2: CDN billing

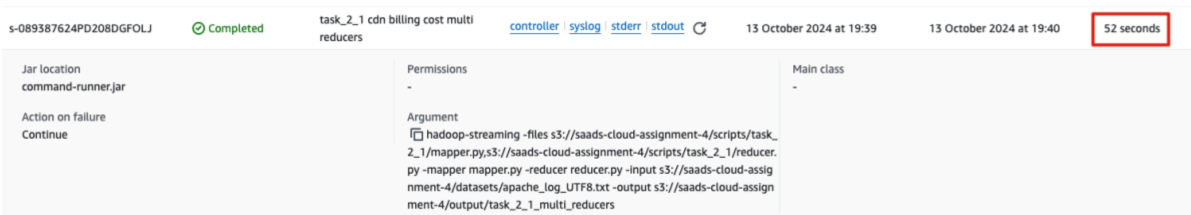
Below is the experimentation I did with this problem.

Task 2.1: Implement a MapReduce program to calculate the resulting CDN costs

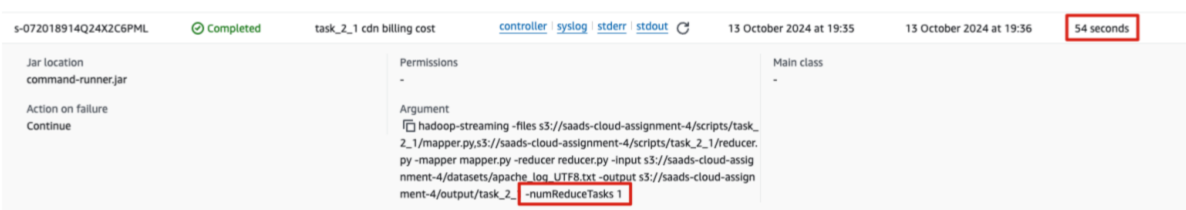
Solution Explanation: The mapper processes each line of the log by counting all requests, regardless of whether they match the expected format (regex), and emits a count of 1 for total requests. It attempts to extract the number of bytes transferred for valid entries, outputting this value when applicable, while ignoring cases where bytes are marked as "-". The reducer accumulates the total counts from the mapper, calculating the CDN costs based on a rate of 0.001 EUR per request and 0.08 EUR per GB of data transferred. It then outputs the total number of requests, total bytes transferred, the individual costs for requests and data transfer, and the overall combined CDN cost.

For this task, the setup was: EMR with four slaves. In the calculations, I have made an assumption that CDN is charging for every request, be it status 200 or 404. The following are the architectures for this solution:

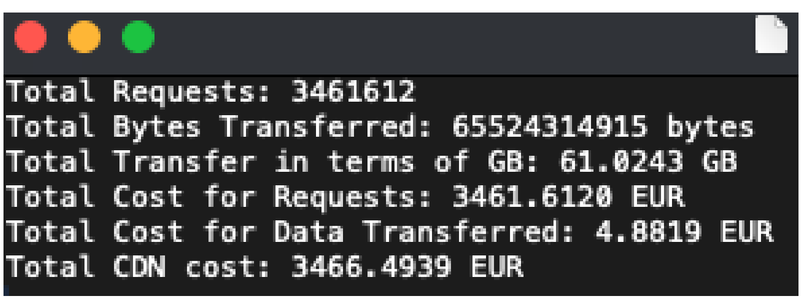
1) **Using Multiple Mappers & Multiple Reducer on an EMR with 1 master and 4 slaves.**
   Multi reducer approach was efficient in this case with 52 second completion time.



2) **Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 4 slaves.**
   It took 54 seconds for this solution to complete.



and the output is as follows:



```
Total Requests: 3461612
Total Bytes Transferred: 65524314915 bytes
Total Transfer in terms of GB: 61.0243 GB
Total Cost for Requests: 3461.6120 EUR
Total Cost for Data Transferred: 4.8819 EUR
Total CDN cost: 3466.4939 EUR
```

Since the dataset had 3.4 million requests and it can be verified as the number of requests calculated by the program are same.

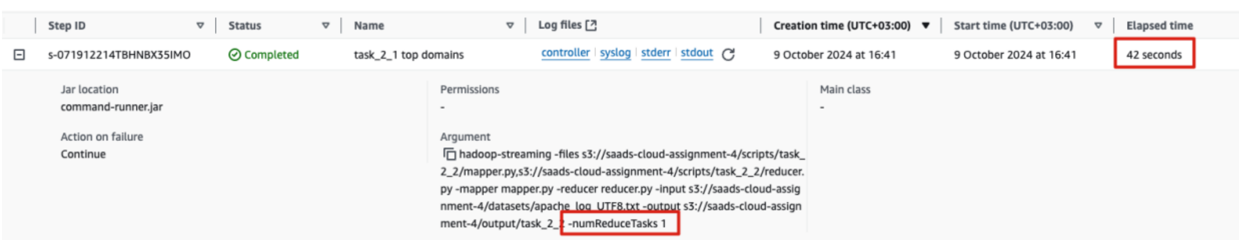
Task 2.2: Implement a MapReduce program to provide the 5 most popular domain names.

Solution Explanation: The mapper reads log entries line by line, extracts the client name (which could be an IP address or a domain name), and checks if it's a domain. It skips IP addresses and processes only domain names by splitting the client name into parts and considering the last two segments as the domain (e.g., "example.com"). For each valid domain, it outputs a count of 1 for that domain. The reducer then aggregates the counts for each domain, stores them in a dictionary, and finally sorts and selects the top 5 most frequent domains, which it prints along with their corresponding counts.

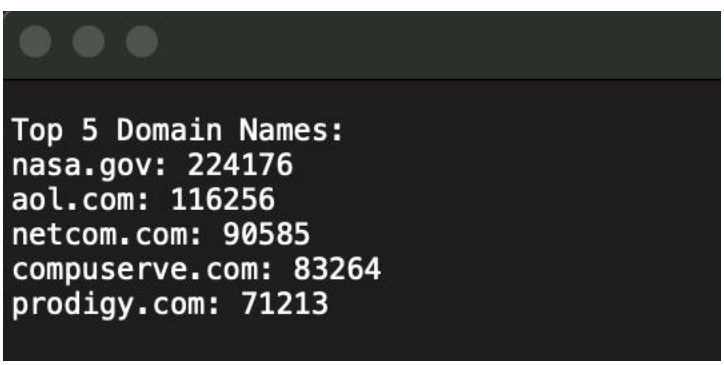
The solution architectures are as following:

1) **Using Multiple Mappers & Multiple Reducer on an EMR with 1 master and 4 slaves.**
   Here again, astonishing results as multiple reducer took more time (46 seconds) as compared to 1 reducer approach (42 seconds)

2) Using Multiple Mappers & 1 Reducer on an EMR with 1 master and 4 slaves.



and the output is as follows:



Conclusion:

The MapReduce programs for CDN billing and domain name extraction performed efficiently, with Task 2.1 completing in 52 seconds and Task 2.2 in 42 seconds (picking best times). Writing the regex for parsing the CDN requests took time, but after local verification, the outputs were accurate and matched the dataset. The EMR cluster with four slaves handled the 3.4 million requests smoothly.

The output for both single and multiple reducer approaches are placed in zip files for tasks.

Reflection:

Have you learned anything completely new?

Yes, a lot of things I explored: the policies of S3, IAM console, IAM policy simulator, and EMR as a whole.

Did anything surprise you?

Yes, the way of adding more slaves for task 1 didn’t result in an improvement to the system.

Did you find anything challenging? Why?

Understanding how Hadoop streaming was challenging yet interesting.

Did you find anything satisfying? Why?

Completing the assignment was satisfying ☺

--- THE END ---