Assignment 2: Spark Data Analytics

SID:440043333
Name: Yuqiong GUO

A brief introduction

This assignment requires an analysis of the raw data extracted from the tweeter. The original data contains the tweeter id, related users and corresponding tweeter text as per the column names on the right.

```
['created_at',
 'hash_tags',
 'id',
 'replyto_id',
 'replyto_user_id',
 'retweet_id',
 'retweet_user_id',
 'text',
 'user_id',
 'user_mentions']
```

There are two workloads for this assignment:

1. Find the top 5 users with similar interest as a given user id
2. Recommend top 5 mention users to each tweet user

Both workloads will be implemented in PySpark using SparkRDD, SparkSQL and SparkML API. The performances were compared in both local drive and EMR environment for both workloads.

Workload one

Design

Workload one is to find the top 5 users with similar interest as a given user id. The data required for this task is "user_id","replyto_id","retweet_id". As per the sample in the assignment description, the document representation should consist of both the reply and retweet for the same user. There are few steps to obtain the document presentation:

1. The three rows are selected and “replyto_id","retweet_id" are combined into a new column called rp_rt.
2. Apply groupby function on user_id to combine the data for the same user.
3. Transfer the type for document representation from array<string> to string.

A snip of the code for data preparation shown as below:

```
user_rp_rt = usertweet.withColumn("rp_rt", concat_ws(',', usertweet['replyto_id'],usertweet['retweet_id'])) \
            .groupBy("user_id") \
            .agg((collect_list("rp_rt"))) \
            .withColumnRenamed("collect_list(rp_rt)","document_presentation")
user_rp_rt = user_rp_rt.withColumn("document_presentation", concat_ws(',', user_rp_rt['document_presentation']))
```

Below is the screenshot of the data to be processed in feature extractors:

```
+--------+---------------------+
| user_id|document_presentation|
+--------+---------------------+
|15466159|   1390027514332991489|
|19652471|   1390023742194061312|
|30616018|   1390026843068239874|
|32947971|   1390027514332991489|
|33868781|                     |
+--------+---------------------+
```

Figure 1 document presentation

The second step is to use the feature extractors to obtain the vector for each user. Two feature extractors used for this task are TF-IDF, Word2Vec:

1. TF-IDF: each user document presentation is converted into a sparse vector.

```
+---------+---------------------------------------------
-------------------------------+
|user_id  |idf_vector
|
+---------+---------------------------------------------
-------------------------------+
|15466159 |(262144,[23124],[1.333966003889251])
|
|19652471 |(262144,[204285],[1.4924942556866434])
|
|30616018 |(262144,[158844],[7.3427791893318455])
|
|32947971 |(262144,[23124],[1.333966003889251])
|
|33868781 |(262144,[249180],[2.1269380446080475])
|
```

Figure 2 TFIDF vectors

2. Word2Vec: each user document presentation is converted into a 5 dimensions dense vector.

```
+---------+----------------------------------------------------------------------------------------------------------
+
|user_id  |word2vec
|
+---------+----------------------------------------------------------------------------------------------------------
+
|15466159 |[0.5083292722702026,-0.7850232124328613,0.989631175994873,1.2494803667068481,0.39224156737327576]
|
|19652471 |[-0.10979729145765305,-0.20897576212882996,0.14346691966056824,0.2684172987937927,0.11481891572475433]
|
|30616018 |[-0.054649580270051956,0.06077373027801514,0.07237964868545532,0.08969934284687042,-0.06203801557421684]
|
|32947971 |[0.5083292722702026,-0.7850232124328613,0.989631175994873,1.2494803667068481,0.39224156737327576]
|
|33868781 |[-0.44282031059265137,0.33450374007225037,-0.9823552966117859,-0.3422980010509491,-0.38356536626815796]
|
```

Figure 3 Word2Vec vectors

The third step is to find the vector of the selected user and compare the cosine similarity between the vector of selected users and the rest of the users, this step is the same for both extractors. The top 5 users with the highest similarity will be the result of this workload.

To improve the performance, cache () was used to keep some RDDs in memory to save some time.

Performance analysis

The code was tested on both local drive and EMR cluster.
The result for both feature extractor in both mode (local drive and EMR cluster) is the same as below, however the performances are not quite the same.

```
Top 5 similar interest user with 157101980 is
3338485689
1374238121924165641
263406194
52527662
762090580864278528
```

Figure 4 Result for workload 1

1. Local Drive
   The environment of the local drive is shown below.

Resource Profile Contents

```
Executor Reqs:
        cores: [amount: 1]
        memory: [amount: 1024]
        offHeap: [amount: 0]
Task Reqs:
        cpus: [amount: 1.0]
```

Figure 5 local drive setting

The result for both Word2Vec and TFIDF is shown below. There are more stages and tasks in Word2Vec compared to TFIDF, this might be caused by that the features & vectors in Word2Vec are dense vectors, however in TFIDF is a sparse vector. Word2Vec performance is slightly better compared with TFIDF.

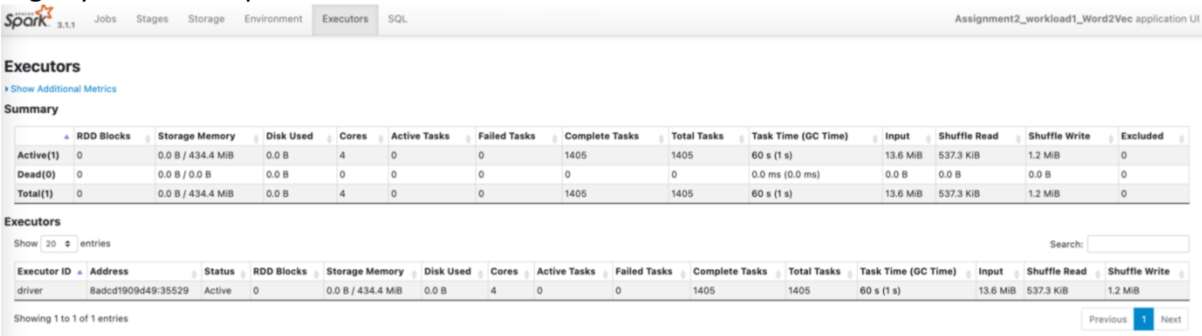


Figure 6 excutors in Word2Vec

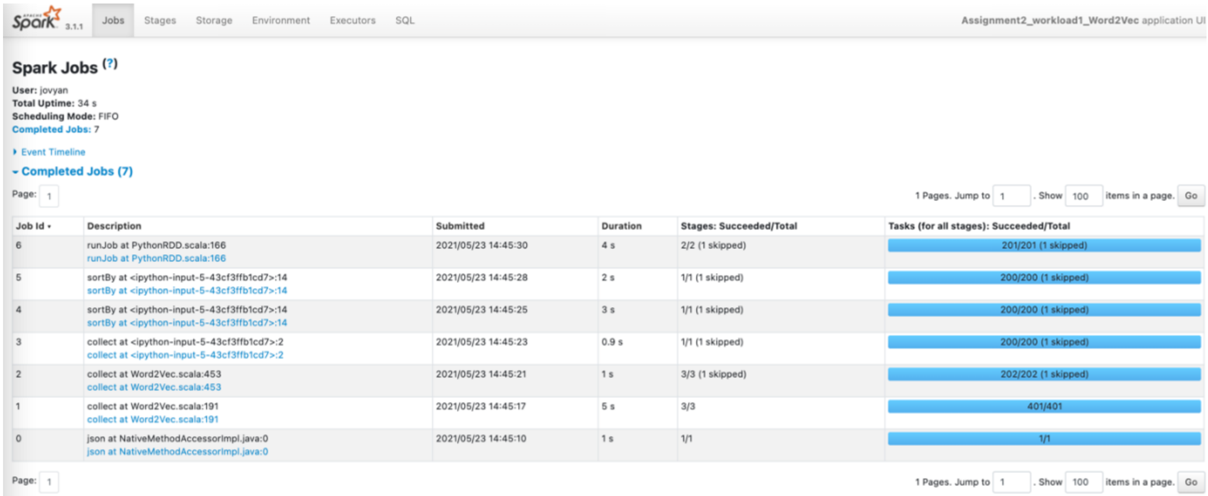


Figure 7 stages in Word2Vec

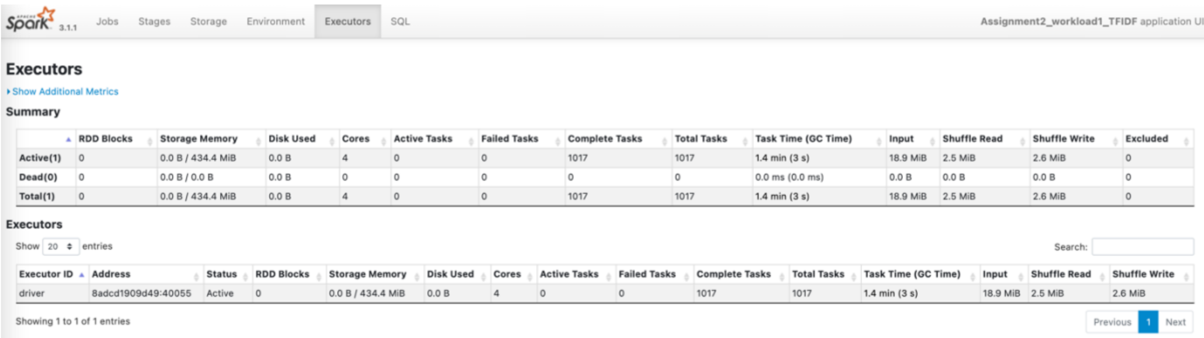
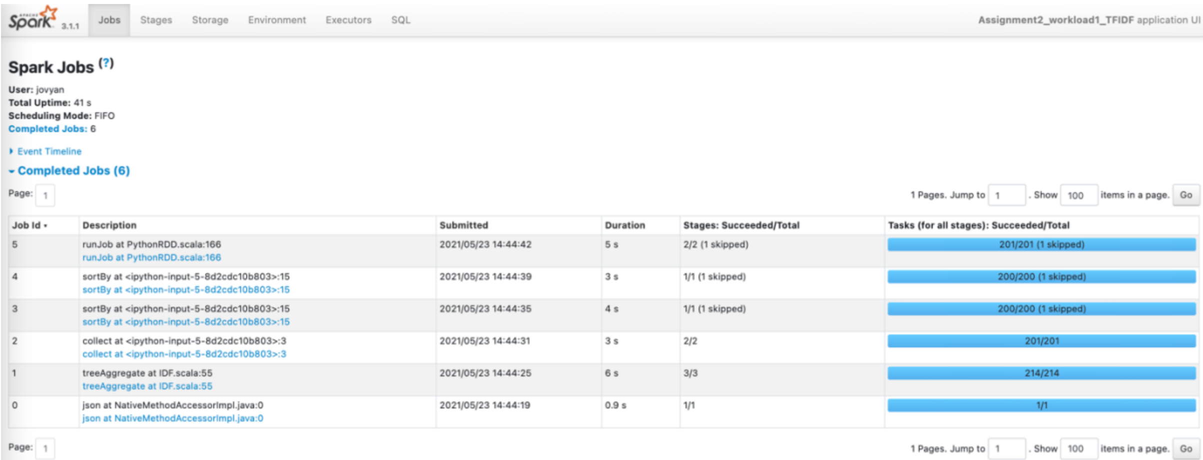


Figure 8 excutors in TFIDF



Spark 3.1.1 | Jobs | Stages | Storage | Environment | Executors | SQL — Assignment2_workload1_TFIDF application UI

Spark Jobs (?)

User: jovyan
Total Uptime: 41 s
Scheduling Mode: FIFO
Completed Jobs: 6

Event Timeline

Completed Jobs (6)

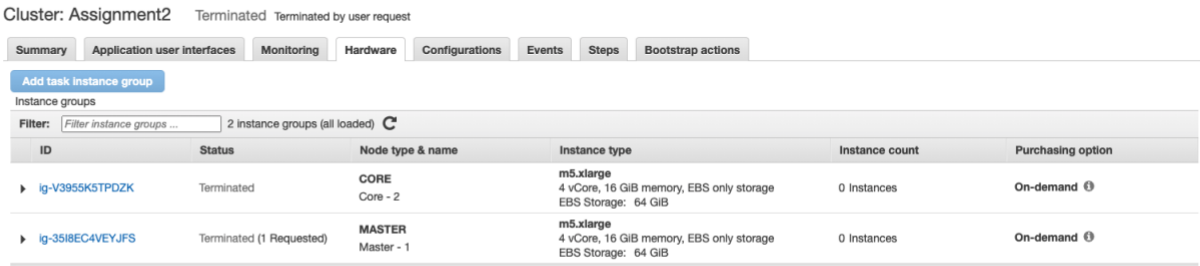
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	runJob at PythonRDD.scala:166	2021/05/23 14:44:42	5 s	2/2 (1 skipped)	201/201 (1 skipped)
4	sortBy at <ipython-input-5-8d2cdc10b803>:15	2021/05/23 14:44:39	3 s	1/1 (1 skipped)	200/200 (1 skipped)
3	sortBy at <ipython-input-5-8d2cdc10b803>:15	2021/05/23 14:44:35	4 s	1/1 (1 skipped)	200/200 (1 skipped)
2	collect at <ipython-input-5-8d2cdc10b803>:3	2021/05/23 14:44:31	3 s	2/2	201/201
1	treeAggregate at IDF.scala:55	2021/05/23 14:44:25	6 s	3/3	214/214
0	json at NativeMethodAccessorImpl.java:0	2021/05/23 14:44:19	0.9 s	1/1	1/1

Figure 9 stages in TFIDF

2. EMR

The EMR cluster was consist of 1 master node and 3 slave nodes, all nodes have 16GB memory as per below. There are 4 executor cores for each node, from the result in Spark-history, the EMR cluster performs slightly better compared with the same extractor in local drive mode.

As the spark-history page shows, when the application started, the driver node will start to work and the two executors will join later. Time consumed decreased when compared with the local drive. Each of the executors runs part of the tasks and which will help to reduce the working time. However, not all slaves works as this application is not very complex.

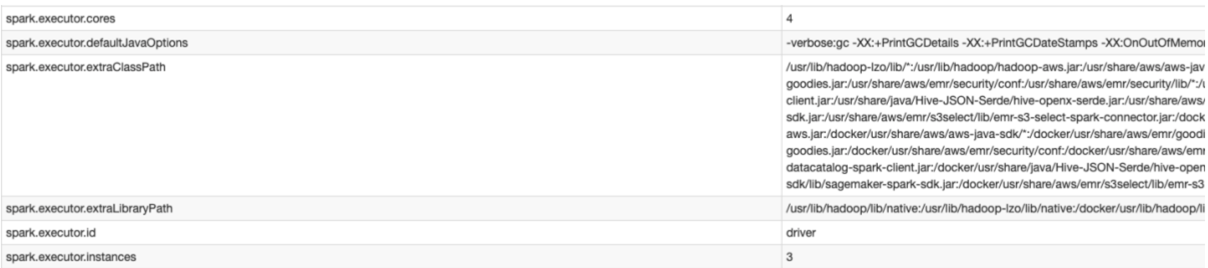


Cluster: Assignment2 Terminated Terminated by user request

Summary | Application user interfaces | Monitoring | Hardware | Configurations | Events | Steps | Bootstrap actions

ID	Status	Node type & name	Instance type	Instance count	Purchasing option
ig-V3955K5TPDZK	Terminated	CORE Core - 2	m5.xlarge 4 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB	0 Instances	On-demand
ig-35I8EC4VEYJFS	Terminated (1 Requested)	MASTER Master - 1	m5.xlarge 4 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB	0 Instances	On-demand

Figure 10 Node setting



spark.executor.cores	4
spark.executor.defaultJavaOptions	-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:OnOutOfMemo
spark.executor.extraClassPath	/usr/lib/hadoop-lzo/lib/*:/usr/lib/hadoop/hadoop-aws.jar:/usr/share/aws/aws-jav goodies.jar:/usr/share/aws/emr/security/conf:/usr/share/aws/emr/security/lib/*:/ client.jar:/usr/share/java/Hive-JSON-Serde/hive-openx-serde.jar:/usr/share/aws/ sdk.jar:/usr/share/aws/emr/s3select/lib/emr-s3-select-spark-connector.jar:/dock aws.jar:/docker/usr/share/aws/aws-java-sdk/*:/docker/usr/share/aws/emr/goodi goodies.jar:/docker/usr/share/aws/emr/security/conf:/docker/usr/share/aws/em datacatalog-spark-client.jar:/docker/usr/share/java/Hive-JSON-Serde/hive-open sdk/lib/sagemaker-spark-sdk.jar:/docker/usr/share/aws/emr/s3select/lib/emr-s3
spark.executor.extraLibraryPath	/usr/lib/hadoop/lib/native:/usr/lib/hadoop-lzo/lib/native:/docker/usr/lib/hadoop/li
spark.executor.id	driver
spark.executor.instances	3

Figure 11 Cluster setting

A sample of the submission in EMR cluster shown as below.

```
[[hadoop@ip-172-31-8-84 Assignment-2]$ bash Ass2_W1_Word2Vec.sh spark-out
```

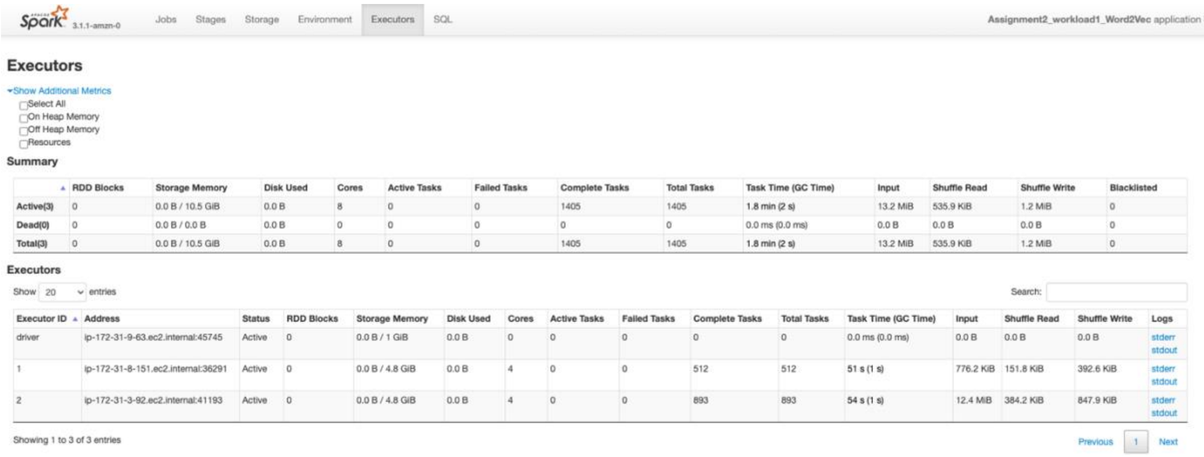


Figure 12 Word2Vec in EMR

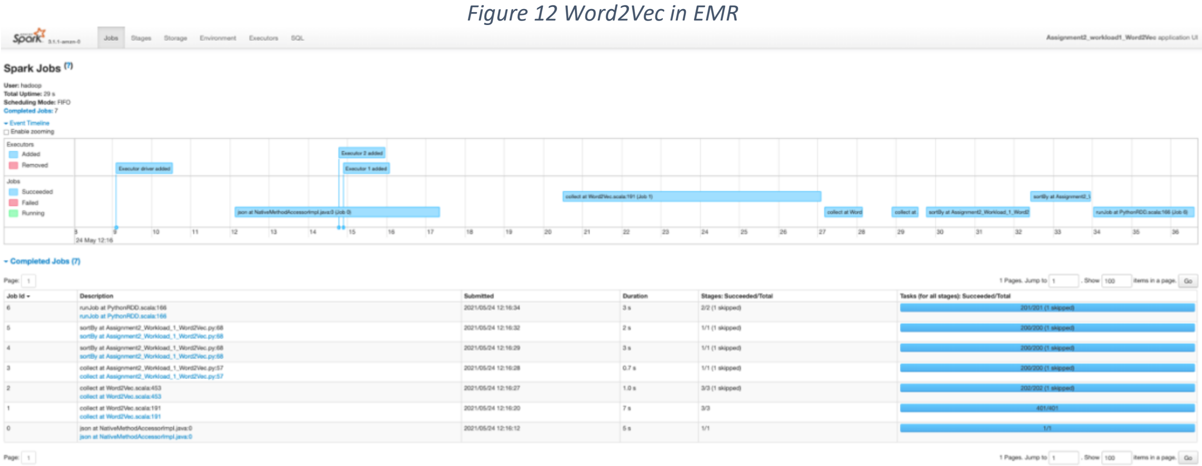


Figure 13 Word2Vec in EMR

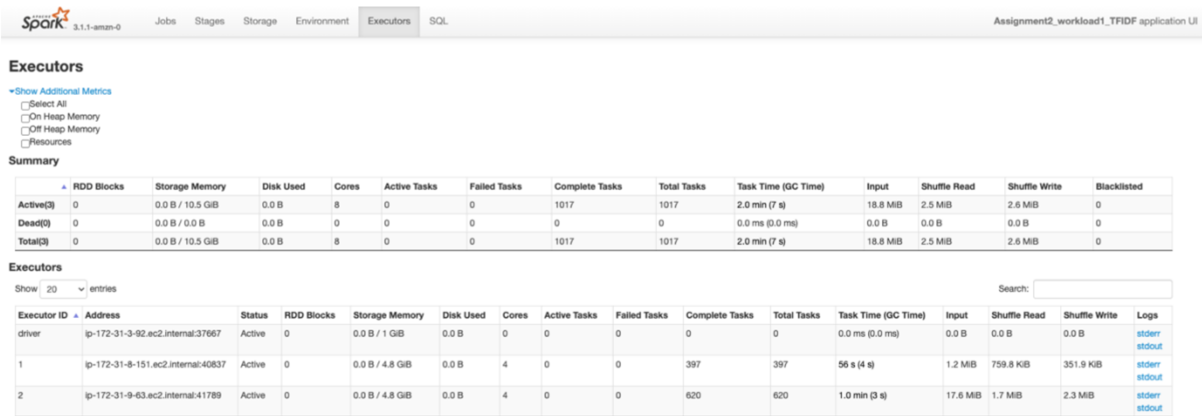


Figure 14 TFIDF in EMR

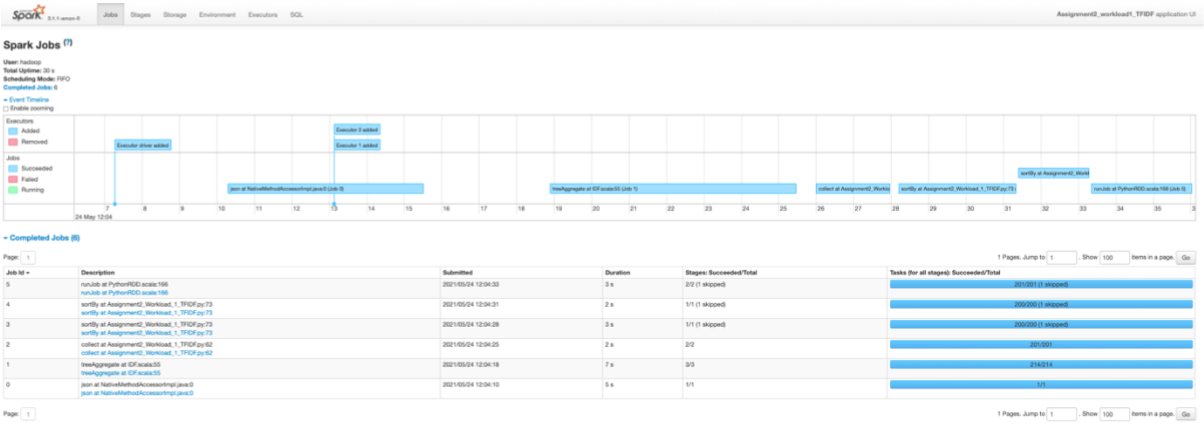


Figure 15 TFIDF in EMR

Workload two

Design

Workload two is to recommend top 5 mention users to each tweet user. The data required for this task is "user_id","user_mentions". By viewing the data extracted, in the user_mentions column, only the first element of each string in the array is required, and the count is 1 for each user_mentions.

```
+-------------------+---------------------------------------------
-------------------+
|user_id            |user_mentions
|
+-------------------+---------------------------------------------
-------------------+
|17799542           |[{807095, [3, 11]}]
|
|1166466828         |[{380648579, [3, 7]}]
|
|1343606436149022723|[{191807697, [3, 16]}]
|
|930226031276982273 |[{15115280, [3, 16]}]
|
|920858307392192513 |[{807095, [3, 11]}]
|
|21458110           |[{20402945, [3, 8]}, {21802625, [130, 138]}]
|
|787062740183552000 |[{26574283, [3, 11]}]
|
```

Figure 16 Raw data

To obtain the data to feed in the recommendation, a few steps required as below:

1. First, extract user_id and user_mentions into an RDD, use flatMap to explode each user_mentions. As the initial user_id and user_mentions type are longint, which will cause a bug when feed into ALS, so new ID has been applied to map the existing user_id and user_mentions. A snip of the code is as below, based on the order of existing user_id, new_user_id will be the sequence of each user_id. The same applies to the user_mentions column.

```
update_user_id = rawdata.select("user_id") \
                 .rdd.map(lambda x: x[0]) \
                 .distinct().collect()

new_user_id = dict()

for item in update_user_id:
    new_user_id[item] = len(new_user_id)
```

The output of the first 5 rows shown as below, (4,0,1) means the user at location 4 mentioned the user at location 0 once.

```
[(0, 0, 1), (1, 1, 1), (2, 2, 1), (3, 3, 1), (4, 0, 1)]
```

2. Then the new RDD of user_id, user_mentions, mention_count will be processed by reducedByKey to get the sum count of the pairs in which user_id, user_mentions are the same. A snip of the result shown below, which means the new_user_id=80 mentioned user 45 once and user 46&47 twice respectively.

```
Row(user_id=80, user_mentions=45, mention_count=1),
Row(user_id=80, user_mentions=46, mention_count=2),
Row(user_id=80, user_mentions=47, mention_count=2),
```

Then the data will be fed to ALS which is one of the collaborative filter algorithms and found out the top 5 mention user for each user.

Performance analysis

The local drive setting in workload 2 is the same as workload 1.
The result is the same compared with workload 1, the EMR cluster has a slightly better performance compared with the local drive. The extra executors will work in parallel to finish extra tasks. Only 2 out of 3 slaves worked as the executors' page shows, this is also related with the application complexity.

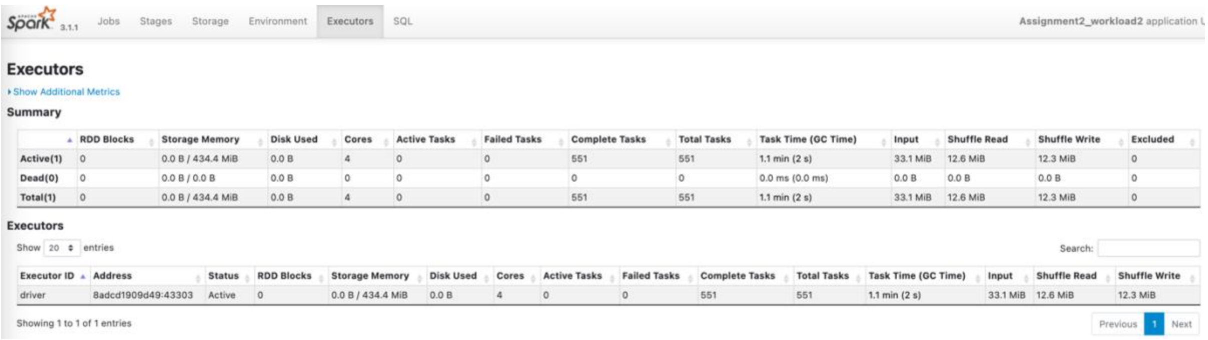


Figure 17 excutors in local drive

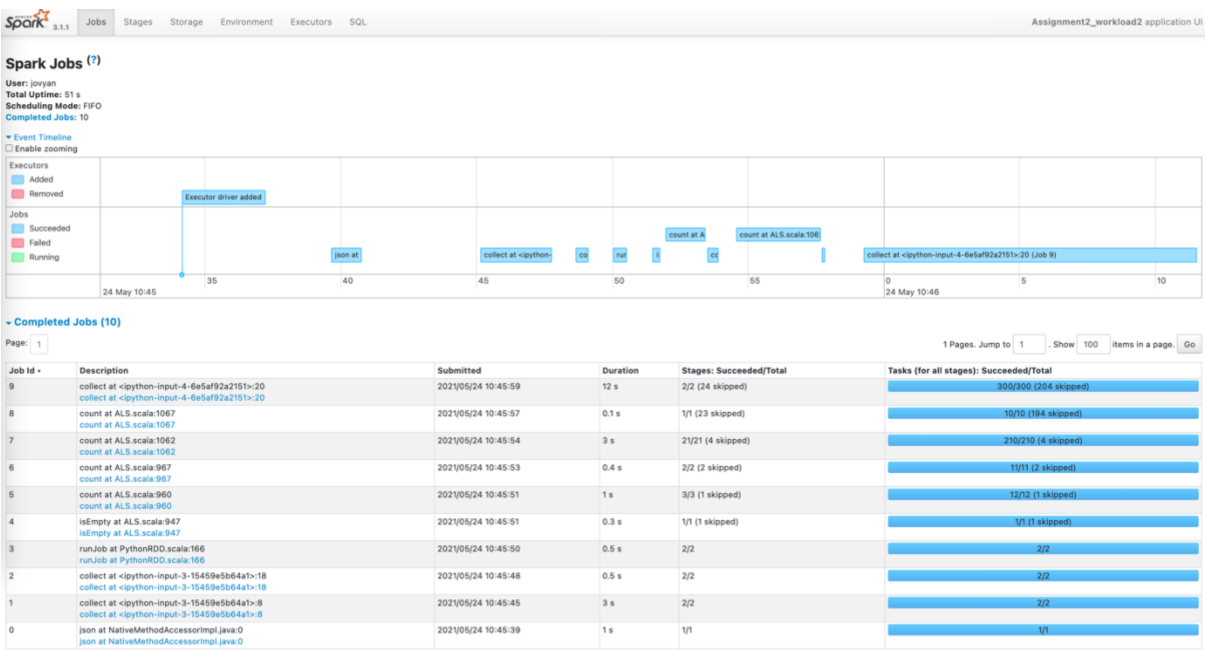


Figure 18 stages in local drive

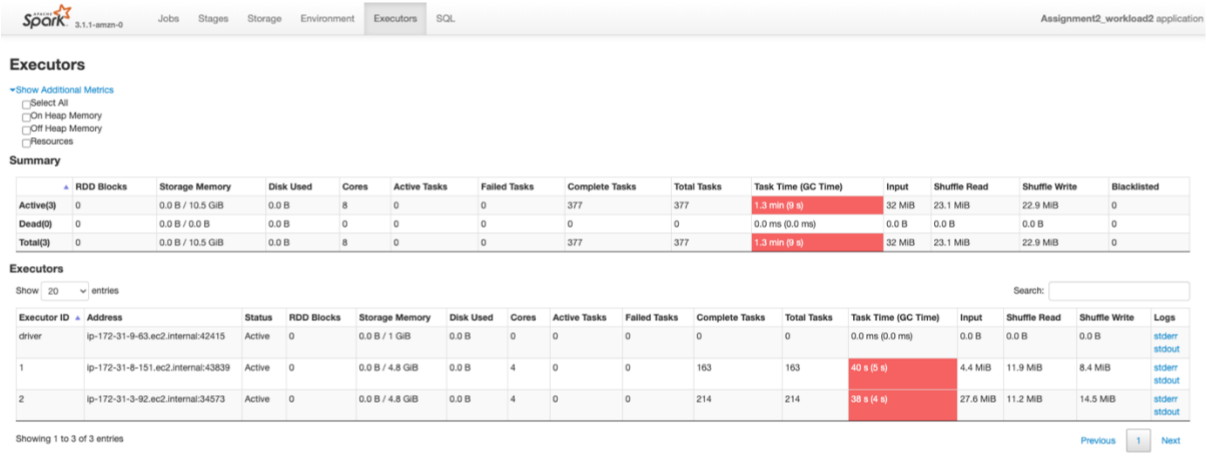


Figure 19 excutors in EMR

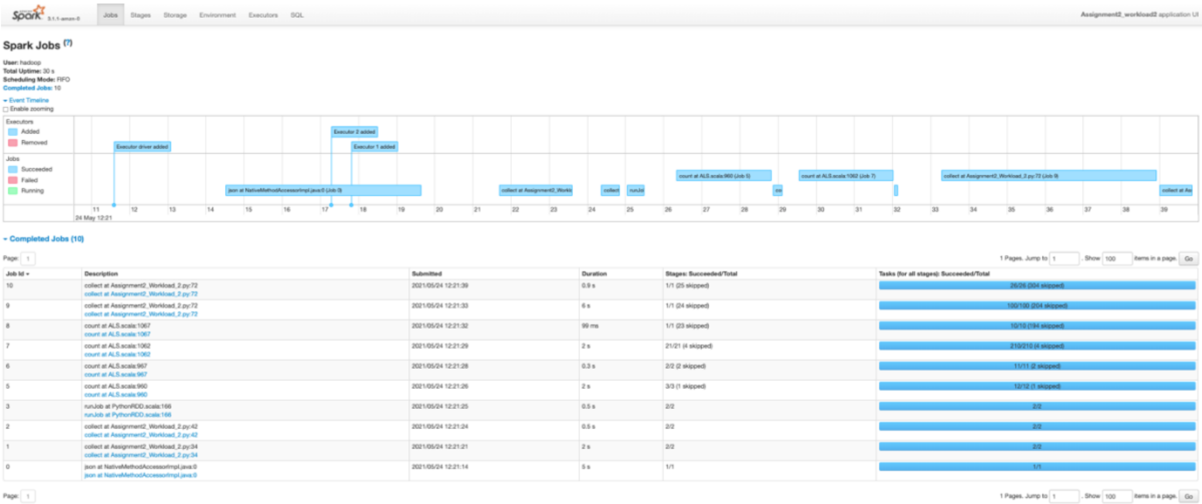


Figure 20 stages in EMR

A brief conclusion

Below is the runtime recorded by Spark-history and also a table of the summary for the time spent on each task.

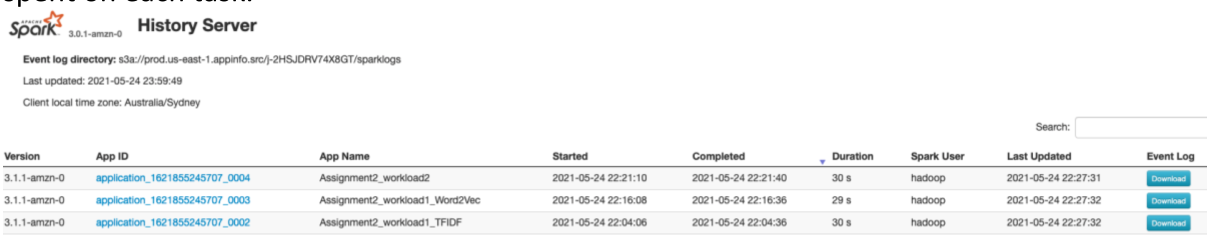


Figure 21 EMR runtime

Figure 22 Local Drive runtime

Runtime	W1 Word2Vec	W1 TFIDF	W2
Local Drive	34s	41s	51s
EMR	29s	30s	30s

Generally, the EMR performance slightly better than the local drive, and Word2Vec performs better compared with TFIDF. The code use cache() to put some RDD in memory for later use which helps the function to be more efficient.
However, the advantage of using EMR cluster will be shown with the sample size increasing. Currently, there is not much difference in the consuming time, as the dataset is small and the application is relatively simple, so most of the time was spent on the data I/O.