COMP5349 Cloud Computing

Report of Assignment 2

Jie Lin
jlin9993@uni.sydney.edu.au
The University of Sydney

1. Introduction

Spark is a powerful tool that provides various API for machine learning in a distributed environment. Dataframe gives us a flexible and abstract data structure for representing the data without worrying about the environment and underlying mechanism. Also, Spark with Dataframe will handle the optimization automatically for us according to the running settings.

In this report, we will conduct to workload using PySpark (primarily using Dataframe and Spark.ml package) on a tweet dataset. Specifically, recommend users for tweet users based on their history tweet records. In the first task, we use two feature extractors to extract the contextual information about users' interests and compute the cosine similarity based on the feature vectors. In the second task, we use the Collaborative Filtering algorithm (ALS package) to compute the Association Matrix and the scores to recommend users for each tweet user.

Working Environment

	Hardware	**Software**
Local Machine	MacBook Air (Retina, 13-inch, 2018); 8 GB 2133 MHz LPDDR3; 1.6 GHz Dual-Core Intel Core i5;	Docker Desktop 3.3.1; Spark-notebook @latest; Spark3.1.1
EMR Cluster	Master: 1 node of m5xlarge; Core: 4 nodes of m5xlarge;	emr-6.2.0; Spark3.0.1;

table 1 - experiment environment

2. Workload one

Design

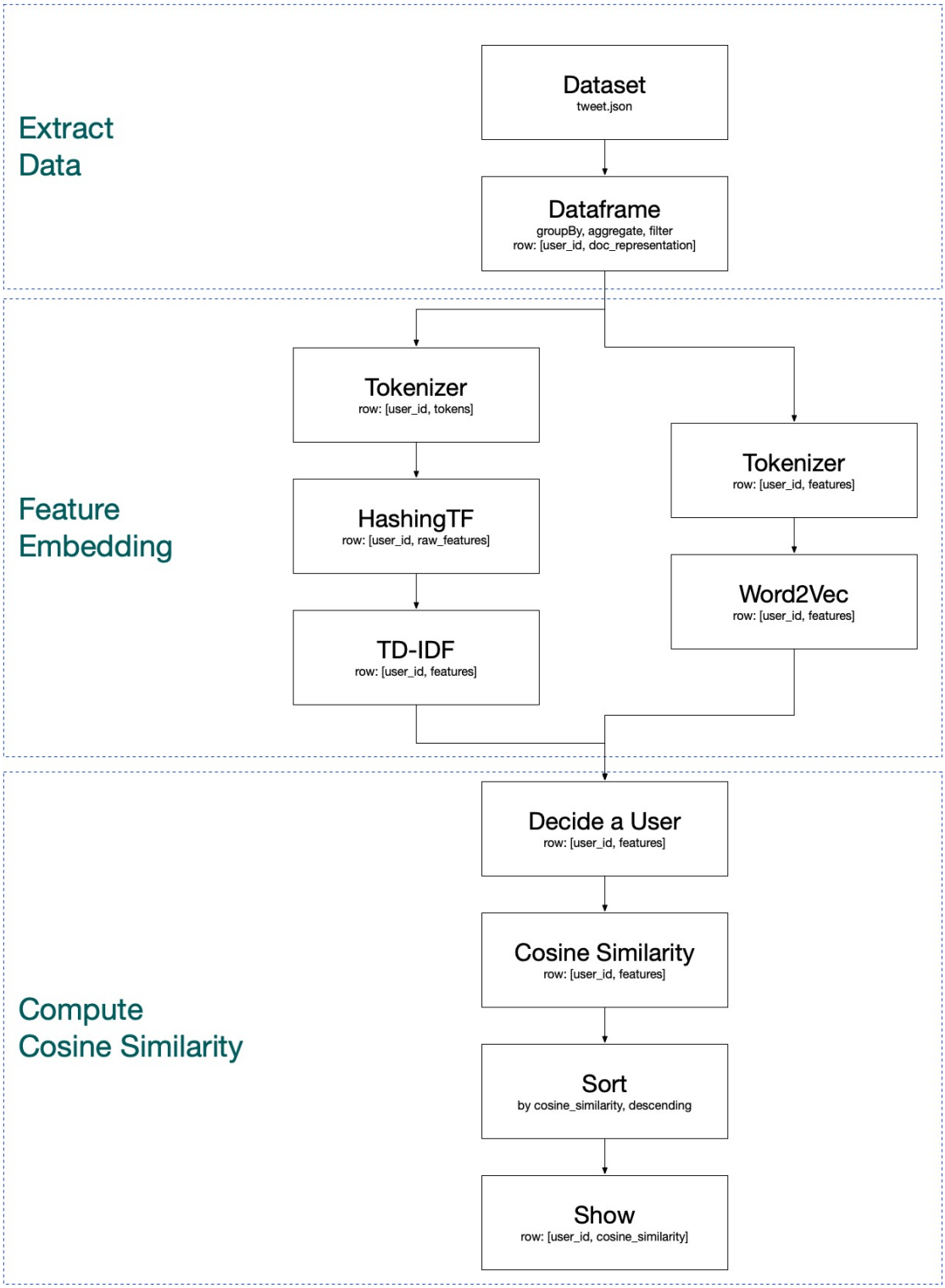


figure 1 - the dataflow breakdown for question 1

The overall workflow is shown above in a graph. There are three main phases, including Extracting

Data, Embedding Features and Computing Cosine Similarity.

Extract Data phase

In the Extract Data phase, the original data JSON file was read into a Dataframe format before using it. Then, relevant attributes (such as user_id, replyto_id, retweet_id) were selected. After that, we group 'user_id' use 'F.collect_list' function to aggregate the 'replyto_id' and 'retweet_id' into two columns before they are concatenated into a string of document representation (a sequence of 'reply_id' or ' retweet_id').

```
df = df_origin\
    .select('user_id', 'replyto_id', 'retweet_id')\
    .groupby("user_id")\
    .agg(
        F.collect_list("replyto_id").alias("rp_list"),
        F.collect_list("retweet_id").alias("rt_list"))\
    .withColumn('document_representation', F.concat_ws(' ', F.col('rp_list'), F.col('rt_list')))\
    .filter('document_representation != ""')\
    .select('user_id', 'document_representation')\
    .cache()

# df.show(5, truncate=False)
# df.printSchema()
```

figure 2 - code for extracting data into the document representation

Feature Embedding

In the second phase, we adopt two feature extractors provided by Spark ml library and process the string of document representation into machine-readable feature vectors. The tokenizer is used to separate the string into a list of words; HashingTF convert the list of words into fixed-size vectors; IDF and Word2Vec work differently inside but Spark provide the same APIs for developers to use (ie. model.fit() and model.transform()). To assemble the entire workflow, we use the Pipeline to integrate them together, so that the incoming data will go through the data-preprocessing pipeline before being feed into the model.

```
# Define Transformers
tokenizer = Tokenizer(inputCol="document_representation", outputCol="tokens")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="raw_features")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")
normalizer = Normalizer(inputCol=idf.getOutputCol(), outputCol="norm")

# Assemble Pipeline
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, normalizer])

# Fit the model
idfModel = pipeline.fit(df)

# Transform the data
data = idfModel.transform(df)
data = data.select('user_id', 'features', 'norm').cache()
```

figure 3 - extracting feature using IDF

```
# Define Transformers
tokenizer = Tokenizer(inputCol="document_representation", outputCol="tokens")
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol=tokenizer.getOutputCol(), outputCol="features")
normalizer = Normalizer(inputCol=word2Vec.getOutputCol(), outputCol="norm")

# Assemble Pipeline
pipeline = Pipeline(stages=[tokenizer, word2Vec, normalizer])

# Fit the model
w2vModel = pipeline.fit(df)

# Transform the data
data = w2vModel.transform(df)
data = data.select('user_id', 'features', 'norm')
```

figure 4 - extracting feature using Word2Vec

Cosine Similarity

In the third phase, we pick a user from to compute the cosine similarity with other users. Because Spark does not provide any native function for computing the cosine similarity, we use UDF (User Defined Function) to implement the formula. Mathematically, the Cosine Similarity can be represented by the dot product of the L2 norm of a vector [1]. Finally, we sort the results in descending order and print out only the top 5 records on both feature extractors.

```
# Compute Cosine Similarity
func_dot = F.udf(lambda x,y: float(x.dot(y)), DoubleType() )
results = data.alias("data")\
    .join(user.alias("user"), F.col("data.user_id") < F.col("user.user_id"))\
    .select(
        F.col("data.user_id").alias("user_id"),
        func_dot("data.norm", "user.norm").alias("cos_similarity"))\
    .sort('cos_similarity', ascending=False)

# Show the results
results.show(5)
```

figure 5 - compute cosine similarity

Performance Analysis

Cache the loaded dataset for both workload

Since the original dataset is the same, as well as the original Dataframe after extraction before feeding into two feature extractors. We use '.cache()' to persist the results locally in either the memory. It can avoid unnecessary IO or network transition in the cluster.

Set a proper partition number

In addition, we set the default number of shuffle partition 'spark.sql.shuffle.partitions=5' since the default number 200 is relatively larger for our dataset. We also find that a large shuffle partition number might cause an Out-Of-Memory problem when running on a local machine with limited memory size.

Using Cache to avoid re-computation

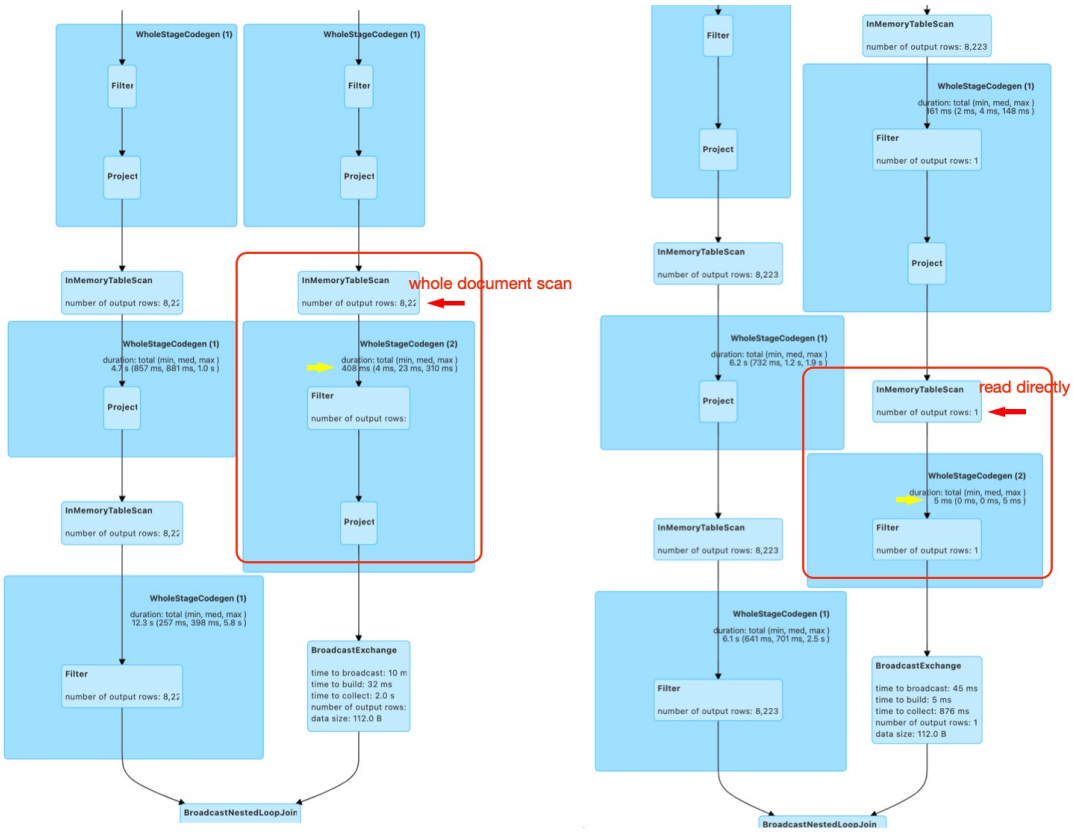


figure 6 - comparison of the DAG with or without caching

```
In [17]: USER_ID = 867477000658841600
         user = df.filter(df['user_id'] == USER_ID)
         user = idfModel.transform(user)
         user = user.select('user_id', 'features', 'norm').cache()
         # user.show()
```

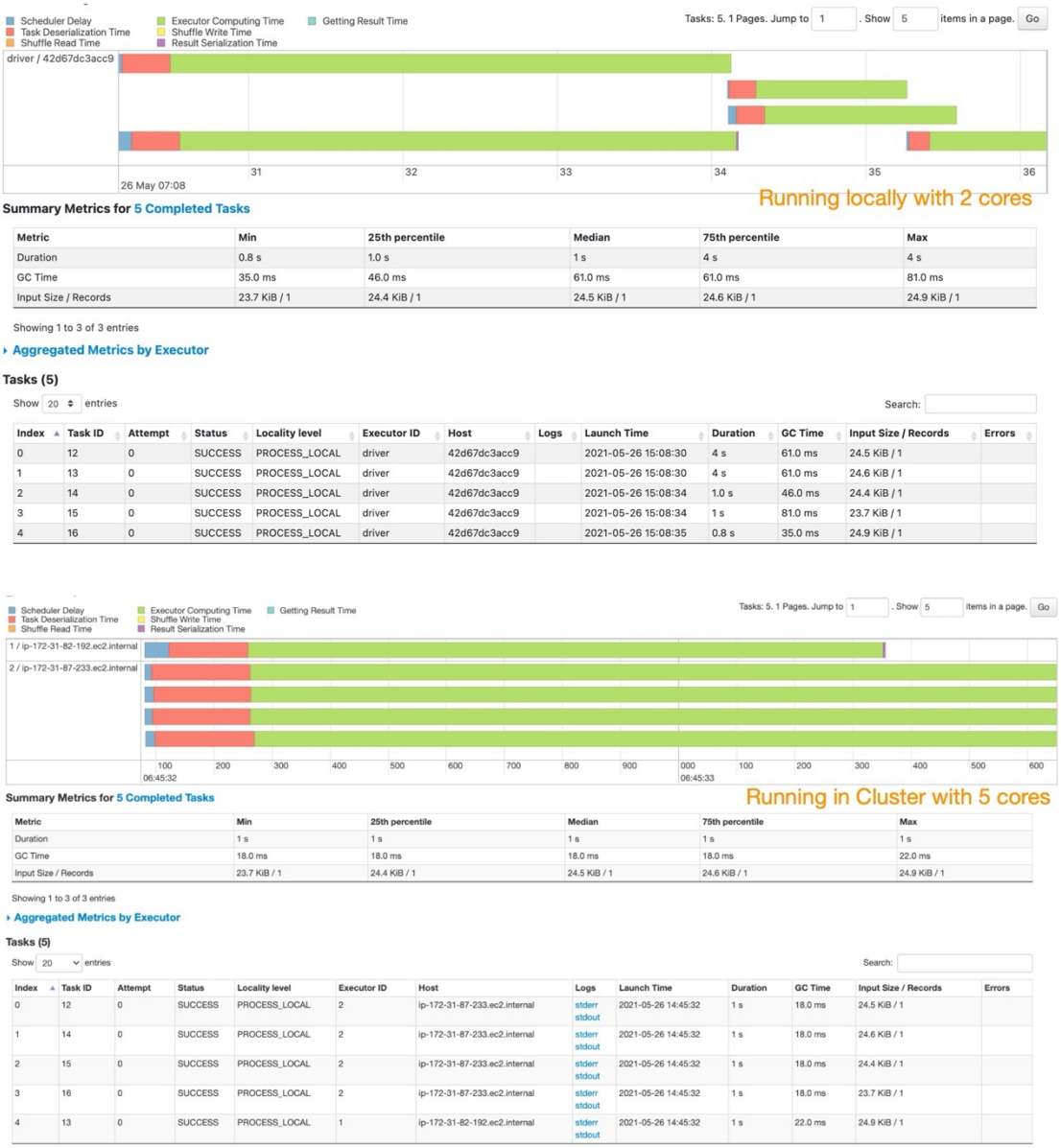
Calculate the cosine similarity.

```
In [18]: func_dot = F.udf(lambda x,y: float(x.dot(y)), DoubleType() )

         results = data.alias("data")\
             .join(user.alias("user"), F.col("data.user_id") < F.col("user.user_id"))\
             .select(
                 F.col("data.user_id").alias("user_id"),
                 func_dot("data.norm", "user.norm").alias("cos_similarity"))\
             .sort('cos_similarity', ascending=False)

         results.show(5)
```

figure 7 - code for caching results before Joining

Caching a small reusable Dataframe before Join can be useful and more efficient to avoid scanning through the documents again. As it is shown on the code snippet, we cache the user selected for computing the Cosine Similarity with other users, before joining the entire Dataframe. The DAG on the right used to cache, which directly read the results from memory. Comparing to the left one without cache, it read the whole Dataframe and do the whole document scan again. The performance time is also different (without cache: 408ms, with cache: 5ms).

The execution schedule in different environment settings



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.8 s	1.0 s	1 s	4 s	4 s
GC Time	35.0 ms	46.0 ms	61.0 ms	61.0 ms	81.0 ms
Input Size / Records	23.7 KiB / 1	24.4 KiB / 1	24.5 KiB / 1	24.6 KiB / 1	24.9 KiB / 1

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	12	0	SUCCESS	PROCESS_LOCAL	driver	42d67dc3acc9		2021-05-26 15:08:30	4 s	61.0 ms	24.5 KiB / 1	
1	13	0	SUCCESS	PROCESS_LOCAL	driver	42d67dc3acc9		2021-05-26 15:08:30	4 s	61.0 ms	24.6 KiB / 1	
2	14	0	SUCCESS	PROCESS_LOCAL	driver	42d67dc3acc9		2021-05-26 15:08:34	1.0 s	46.0 ms	24.4 KiB / 1	
3	15	0	SUCCESS	PROCESS_LOCAL	driver	42d67dc3acc9		2021-05-26 15:08:34	1 s	81.0 ms	23.7 KiB / 1	
4	16	0	SUCCESS	PROCESS_LOCAL	driver	42d67dc3acc9		2021-05-26 15:08:35	0.8 s	35.0 ms	24.9 KiB / 1	

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 s	1 s	1 s	1 s	1 s
GC Time	18.0 ms	18.0 ms	18.0 ms	18.0 ms	22.0 ms
Input Size / Records	23.7 KiB / 1	24.4 KiB / 1	24.5 KiB / 1	24.6 KiB / 1	24.9 KiB / 1

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	12	0	SUCCESS	PROCESS_LOCAL	2	ip-172-31-87-233.ec2.internal	stderr stdout	2021-05-26 14:45:32	1 s	18.0 ms	24.5 KiB / 1	
1	14	0	SUCCESS	PROCESS_LOCAL	2	ip-172-31-87-233.ec2.internal	stderr stdout	2021-05-26 14:45:32	1 s	18.0 ms	24.6 KiB / 1	
2	15	0	SUCCESS	PROCESS_LOCAL	2	ip-172-31-87-233.ec2.internal	stderr stdout	2021-05-26 14:45:32	1 s	18.0 ms	24.4 KiB / 1	
3	16	0	SUCCESS	PROCESS_LOCAL	2	ip-172-31-87-233.ec2.internal	stderr stdout	2021-05-26 14:45:32	1 s	18.0 ms	23.7 KiB / 1	
4	13	0	SUCCESS	PROCESS_LOCAL	1	ip-172-31-82-192.ec2.internal	stderr stdout	2021-05-26 14:45:32	1 s	22.0 ms	24.9 KiB / 1	

figure 8 - comparison of the same aggregate stage running locally and in clusters

The execution timelines show that the number capacity of running task in parallel depends on the number of core or node they have. The first graph indicates that the tasks are assigned into one executor running on two cores. The second graph shows that the tasks are evenly distributed into 5 cores (4 tasks in an executor while 1 task in another node).

3. Workload two

Design

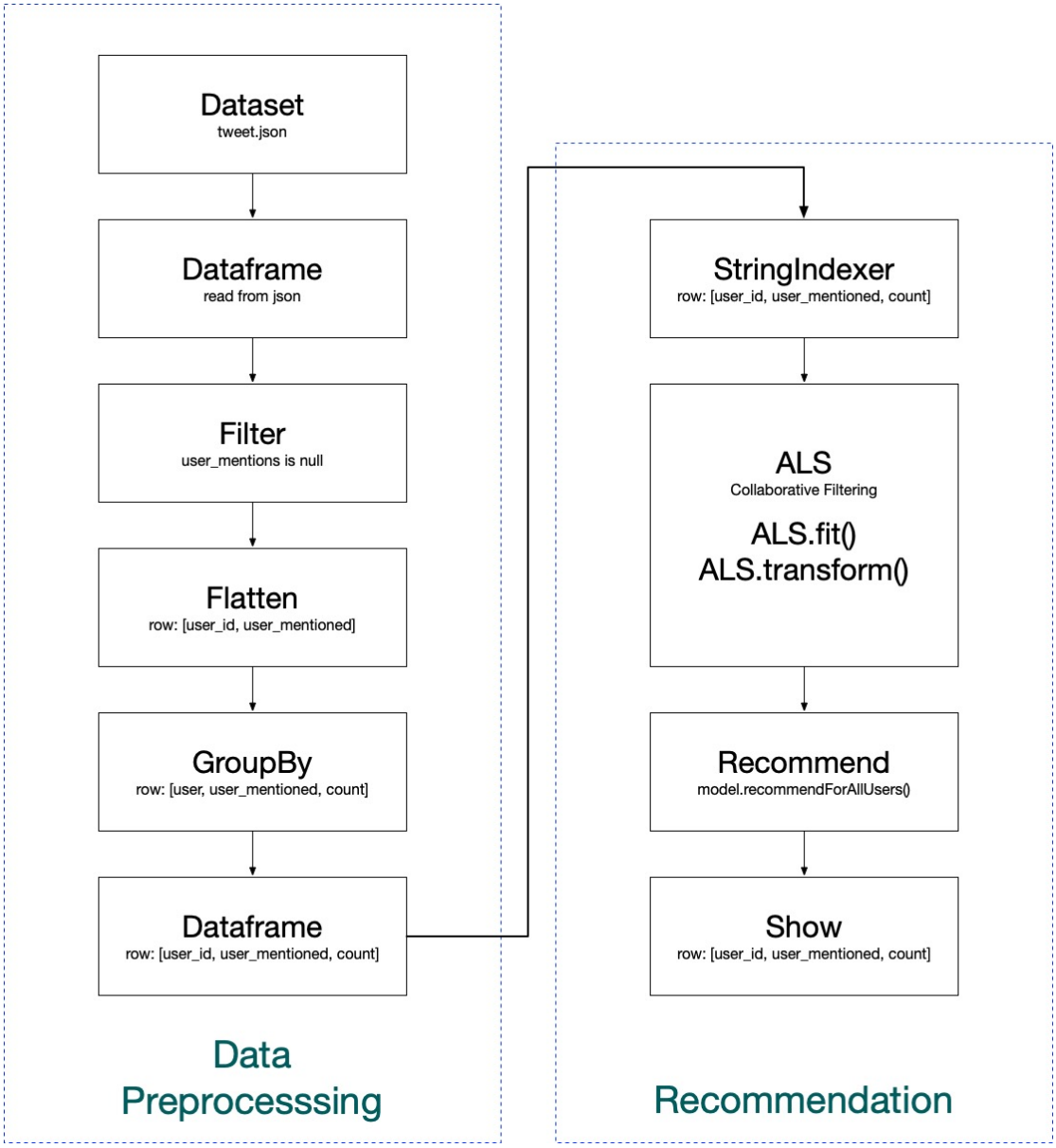


figure 9 - the dataflow breakdown for question 2

The work workflowQ2 is shown above, which consists of two phases.

Data Pre-processing

The first phase is Data Pre-processing, in which we focus on transforming the original data file into a rating matrix. The rating matrix should contain three main attributes: users, items, and ratings. In this case, 'users' means the 'user_id' of tweets; 'items' means the 'user_id' appeared in the 'user_mentions' array; 'rantings' means the occurrence number of the same 'user' and 'item' pair.

```
df = df_origin\
    .filter("user_mentions is not null")\
    .select("user_id", "user_mentions.id")\

df = df\
    .select('user_id', F.explode(df.id).alias('user_mentioned'))

df = df.groupBy(df.columns)\
    .count()\
    .where(F.col('count') >= 1)\
    .cache()

# df.show(5)
# df.printSchema()
```

figure 10 - code for data-preprocessing

Recommendation using ALS

In the second phase, we use ALS (Alternating Least Squares) matrix factorization to estimate the scores for users and recommend users to each tweet users. Spark provides an ALS module that we can use to achieve the results. However, the input type that ALS model expects is different from the type that we have. Specifically, the user and item columns that ALS wants should be Integer type whereas the 'user_id' of the original dataset is in Long type or String type. We use StringIndexer to transform the String into Integer before feeding them into the ALS model. Then, we use regular methods provided by ALS (model.fit() and model.transform()) to process our Association Matrix. Finally, the recommendation for each tweet users computed by using 'model.recommendationForAllUsers()'.

```
# Transform long -> int
userIndexer = StringIndexer(inputCol="user_id", outputCol="_user_id")
userIndexerModel = userIndexer.fit(df)
data = userIndexerModel.transform(df)

itemIndexer = StringIndexer(inputCol="user_mentioned", outputCol="_user_mentioned")
itemIndexerModel = itemIndexer.fit(data)
data = itemIndexerModel.transform(data)

# Define the Model
als = ALS(maxIter=5, regParam=0.01,
          userCol="_user_id",
          itemCol="_user_mentioned",
          ratingCol="count",
          coldStartStrategy="drop",
          nonnegative=True)

# Fit
alsModel = als.fit(data)

# Recommendations
userRecs = alsModel.recommendForAllUsers(5)

# Results
userRecs.show(truncate=False)
```

figure 11 - code for implementing ALS

Performance Analysis

Runtime in different environment settings

The execution timelines of both environmental settings are slightly different, even the logical plan of the scripts is the same. Spark might optimize the logical plan to better utilize the setting in the middle. For example, distributing aggregation task to different nodes to run in parallel. Also, the run time on the local machine and cluster are significantly different.

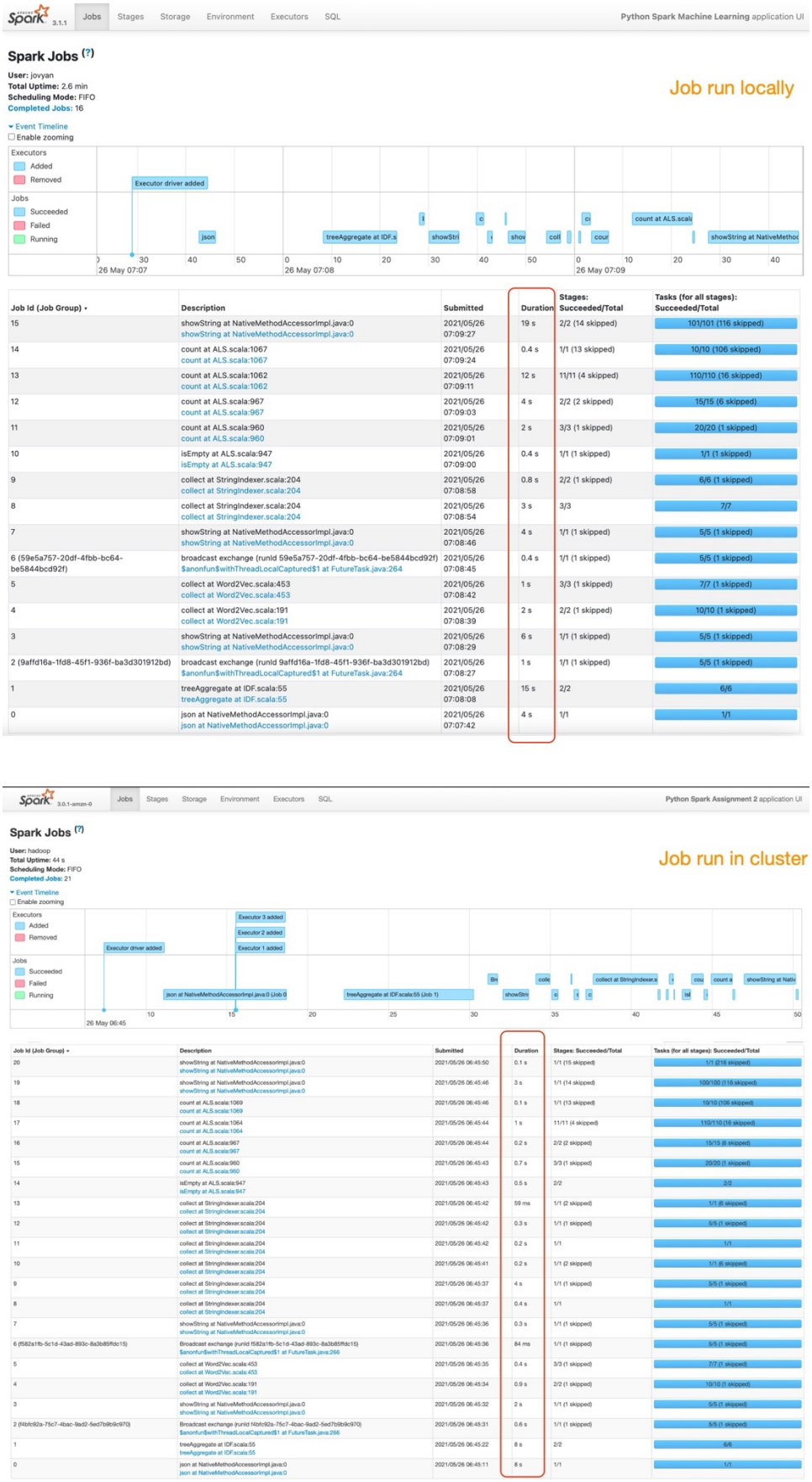


figure 12 - execution timeline in local machine and cluster

The mechanism of caching

Running locally with in memory cache

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
78	showString at NativeMethodAccessorImpl.java:0	+details	2021/05/26 07:09:45	0.3 s	1/1			480.8 KiB	
77	showString at NativeMethodAccessorImpl.java:0	+details	2021/05/26 07:09:27	18 s	100/100	8.4 MiB			2.3 MiB
62	count at ALS.scala:1067	+details	2021/05/26 07:09:24	0.3 s	10/10	168.3 KiB		130.2 KiB	
48	count at ALS.scala:1062	+details	2021/05/26 07:09:23	0.7 s	10/10	286.8 KiB		94.0 KiB	
47	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:22	0.5 s	10/10	94.5 KiB		130.2 KiB	94.0 KiB
46	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:22	0.6 s	10/10	184.0 KiB		93.5 KiB	130.2 KiB
45	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:21	0.5 s	10/10	94.5 KiB		129.9 KiB	93.5 KiB
44	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:20	0.8 s	10/10	184.0 KiB		92.9 KiB	129.9 KiB
43	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:19	1 s	10/10	94.5 KiB		127.9 KiB	92.9 KiB
42	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:17	1 s	10/10	184.0 KiB		90.4 KiB	127.9 KiB
41	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:14	3 s	10/10	94.5 KiB		119.2 KiB	90.4 KiB
40	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:13	1 s	10/10	184.0 KiB		83.0 KiB	119.2 KiB
39	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:12	0.8 s	10/10	94.5 KiB		432.7 KiB	83.0 KiB
37	flatMap at ALS.scala:1688	+details	2021/05/26 07:09:11	0.4 s	10/10	184.0 KiB			432.7 KiB
33	count at ALS.scala:967	+details	2021/05/26 07:09:04	3 s	10/10			58.2 KiB	
32	map at ALS.scala:1599	+details	2021/05/26 07:09:03	0.8 s	5/5	128.3 KiB			58.2 KiB
29	count at ALS.scala:960	+details	2021/05/26 07:09:02	0.6 s	10/10			62.1 KiB	
28	map at ALS.scala:1599	+details	2021/05/26 07:09:01	0.7 s	5/5			74.9 KiB	62.1 KiB
27	mapPartitions at ALS.scala:1356	+details	2021/05/26 07:09:01	0.4 s	5/5	111.5 KiB			74.9 KiB
25	isEmpty at ALS.scala:947	+details	2021/05/26 07:09:00	0.3 s	1/1	22.7 KiB			
23	collect at StringIndexer.scala:204	+details	2021/05/26 07:08:58	0.3 s	1/1			20.9 KiB	
22	collect at StringIndexer.scala:204	+details	2021/05/26 07:08:58	0.4 s	5/5	111.5 KiB			20.9 KiB
20	collect at StringIndexer.scala:204	+details	2021/05/26 07:08:56	0.5 s	1/1			125.8 KiB	
19	collect at StringIndexer.scala:204	+details	2021/05/26 07:08:55	0.9 s	5/5			112.9 KiB	125.8 KiB
18	collect at StringIndexer.scala:204	+details	2021/05/26 07:08:54	1 s	1/1	5.4 MiB			112.9 KiB

Running in cluster

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
99	showString at NativeMethodAccessorImpl.java:0	+details	2021/05/26 06:45:50	0.1 s	1/1			482.9 KiB	
83	showString at NativeMethodAccessorImpl.java:0	+details	2021/05/26 06:45:47	3 s	100/100	8.2 MiB			2.4 MiB
68	count at ALS.scala:1069	+details	2021/05/26 06:45:46	0.1 s	10/10	168.3 KiB		129.7 KiB	
54	count at ALS.scala:1064	+details	2021/05/26 06:45:46	94 ms	10/10	286.8 KiB		92.7 KiB	
53	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:46	77 ms	10/10	94.5 KiB		129.7 KiB	92.7 KiB
52	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	85 ms	10/10	184.0 KiB		92.8 KiB	129.7 KiB
51	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	96 ms	10/10	94.5 KiB		129.2 KiB	92.8 KiB
50	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	84 ms	10/10	184.0 KiB		92.2 KiB	129.2 KiB
49	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	0.1 s	10/10	94.5 KiB		127.6 KiB	92.2 KiB
48	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	0.1 s	10/10	184.0 KiB		89.4 KiB	127.6 KiB
47	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	0.1 s	10/10	94.5 KiB		116.7 KiB	89.4 KiB
46	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	0.1 s	10/10	184.0 KiB		81.7 KiB	116.7 KiB
45	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:45	0.2 s	10/10	94.5 KiB		432.8 KiB	81.7 KiB
44	flatMap at ALS.scala:1690	+details	2021/05/26 06:45:44	0.1 s	10/10	184.0 KiB			432.8 KiB
39	count at ALS.scala:967	+details	2021/05/26 06:45:44	0.1 s	10/10			58.2 KiB	
38	map at ALS.scala:1601	+details	2021/05/26 06:45:44	0.1 s	5/5	128.3 KiB			58.2 KiB
35	count at ALS.scala:960	+details	2021/05/26 06:45:44	0.2 s	10/10			62.1 KiB	
34	map at ALS.scala:1601	+details	2021/05/26 06:45:44	0.2 s	5/5			74.9 KiB	62.1 KiB
33	mapPartitions at ALS.scala:1358	+details	2021/05/26 06:45:43	0.4 s	5/5			112.9 KiB	74.9 KiB
31	isEmpty at ALS.scala:947	+details	2021/05/26 06:45:43	0.1 s	1/1			22.9 KiB	
30	rdd at ALS.scala:697	+details	2021/05/26 06:45:43	0.4 s	1/1	5.4 MiB			112.9 KiB
29	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:42	51 ms	1/1			20.9 KiB	
26	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:42	0.2 s	5/5			112.9 KiB	20.9 KiB
24	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:42	0.1 s	1/1	5.4 MiB			112.9 KiB
23	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:41	0.1 s	1/1			125.8 KiB	
20	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:37	4 s	5/5			112.9 KiB	125.8 KiB
18	collect at StringIndexer.scala:204	+details	2021/05/26 06:45:37	0.4 s	1/1	5.4 MiB			112.9 KiB

figure 13 - different caching mechanism on the local machine and on cluster

We also compare the input size between stages on the local machine and cluster. The graphs show the difference in caching mechanism. When running locally, the in-memory cache will be effectively used to avoid re-computation, while in the cluster the file might be distributed across all nodes and being recomputed when needed.

The number of executors on the cluster

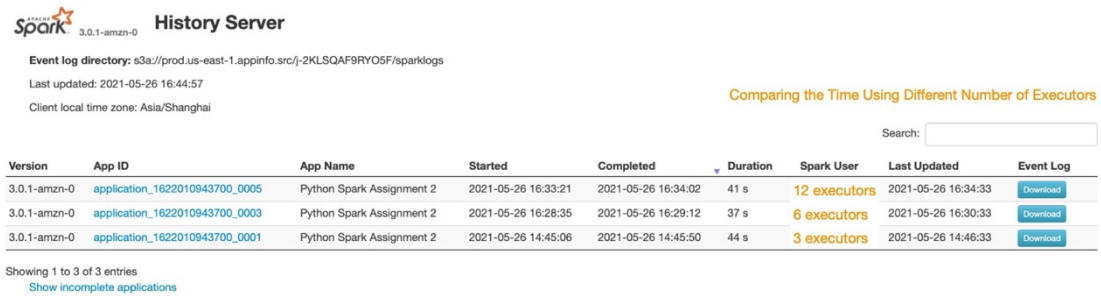


figure 14 - the runtime on the cluster using a different number of executors

Different parameters were set to a 'spark-submit' file and executed on the cluster. In particular, the number of executors was experimented with and find that the difference in runtime is insignificant, given this dataset.

4. Conclusion

Effectively using cache can reduce the re-computation. Caching intermediate results for later use is feasible, but it does not always work when running on a cluster.

Some configuration parameters can decide the performance of a task, such as the number of shuffle partition, the number of executors, etc. However, a larger number of executors not always compromise better performance with regards to time.

5. References

[1] B. (2020, October 19). Euclidean Distance vs Cosine Similarity. Baeldung on Computer Science. https://www.baeldung.com/cs/euclidean-distance-vs-cosine-similarity.

[2] PySpark Documentation — PySpark 3.1.1 documentation. (n.d.). PySpark. Retrieved May 26, 2021, from https://spark.apache.org/docs/latest/api/python/