**Introduction:**

This assignment uses python as the programming language, Spark RDD, SQL & ML APIs and AWS as the cloud environment to implement and run the project.

AWS services used were EMR Cluster, EMR Notebook & S3.

**EMR Cluster**

* Software configuration emr-5.29.0 was used to include Hadoop 2.8.5, Spark 2.4.4, Livy 0.6.0 & Tenserflow1.14.0 components
* Hardware configuration of 1 Master and 5 Core of m5.xlarge were used.
* Other configuration to set the maximum resource allocations and include the bootstrap files to install required packages (nltk, tensorflow-hub and pandas) were loaded from s3.
* Security groups for master and core were created and enabled port 22 for master, so we can SSH to the master node.

**EMR Notebook**

* EMR Notebooks were created and associated to the EMR cluster.
* Kernel was changed to pyspark.

**S3**

* Copy of MulitiNLI corpus containing dev\_matched, dev\_mismatched, test\_matched test\_mismatched and train .tsv files were uploaded in S3.

**Vocabulary Exploration:**

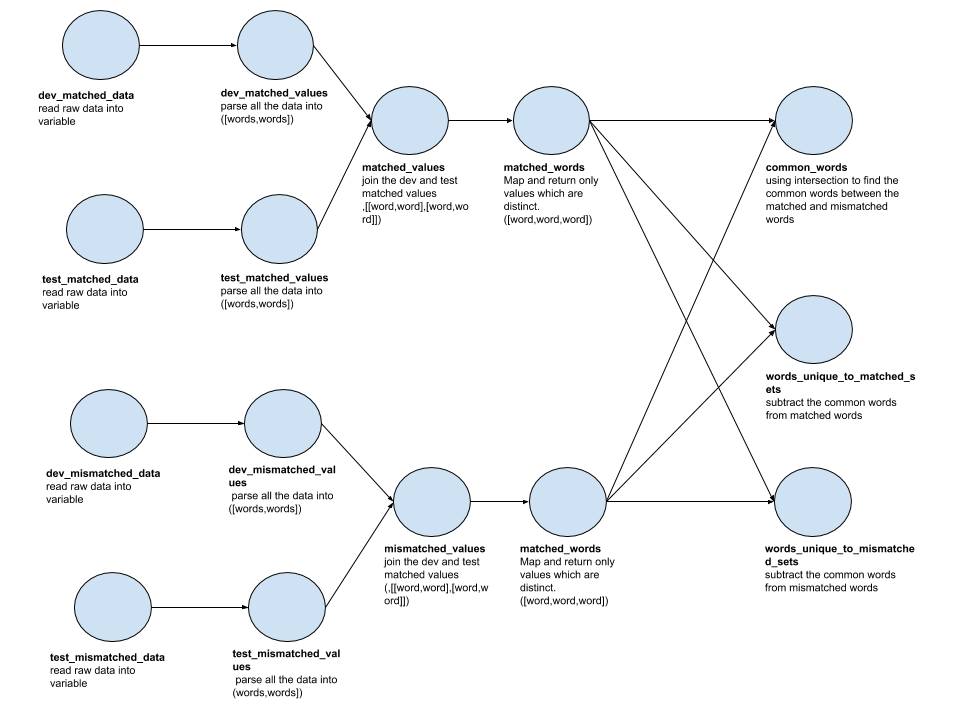
**Task 1**

* Finding the common words between matched and mismatched sets
* the number of words unique to the matched sets
* the number of words unique to the mismatched sets

**Application Design:**

The following DAG diagram shows the data flow and the sequence of executions

\*Please refer to appendix section for more detailed DAG visualizations.



**Implementation Details:**

We start our implementation by reading the tab delimited file and taking only the required columns (sentence 1 and sentence 2) for our analysis.

While we read the data into a new RDD, we combine the required columns into one new column “joined\_sentence” and as well remove header and apply tokenization (nltk) on the joined sentence column to get the words.

For reusability, we put the above logic is one function getParsedDataValues, which takes the textfile as input. We call this function on different data sets to read and convert the file into list of lists format i.e. [[word,word],[word,word]]

We combine dev and test matched RDD’s into single RDD “matched\_values” and mismatched dev and test RDD’s into “mismatched\_values” using union.

The matched and mismatched sets are flattened from list of lists into single list of words for further processing and removed duplicates using distinct.

To get the common words between the matched and mismatched word RDDs, we use intersection of RDDs.

To get the unique words from the matched set, we remove the common words from matched words, using subtract on matched words RDD and common words RDD

To get the unique words from the mismatched set, we remove the common words from mismatched words, using subtract on mismatched words RDD and common words RDD

**Results:**

Number of Common Words Between Matched and Mismatched Sets: 9485

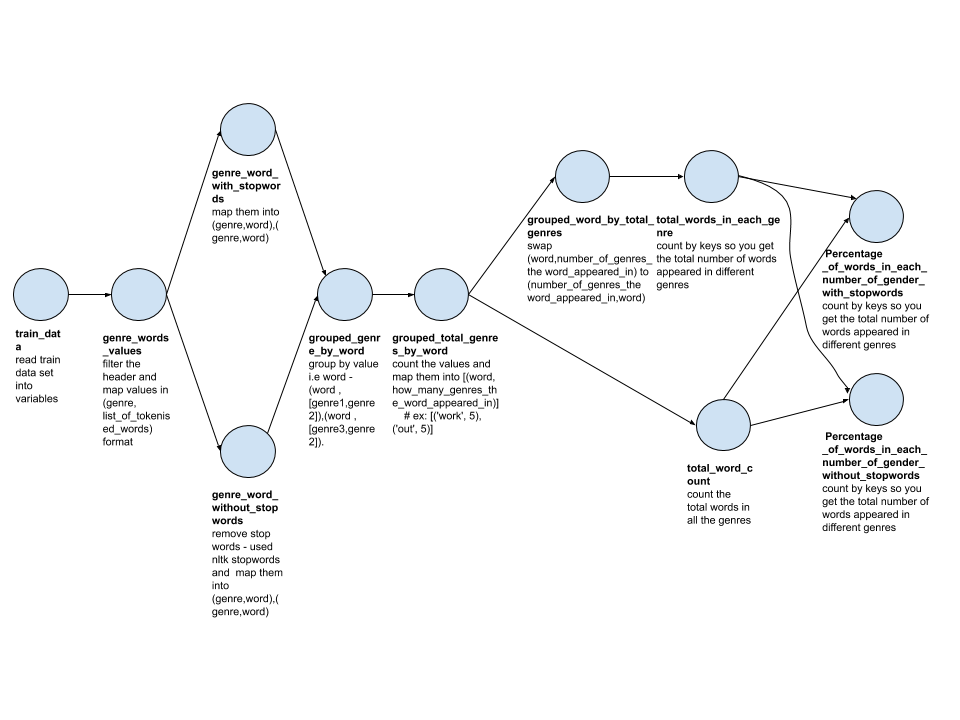
Number of Unique Words in Matched Sets: 10595

Number of Unique Words in Mismatched Sets 8242

**Task 2**

* The percentages of words appearing in five genres, in four genres, in three genres, in two genres and in one genre
* The same percentages after removing a given list of stop words

**Application Design:**

The following DAG diagram shows the data flow and the sequence of executions **Implementation Details:**

For task 2, we read the train data into an RDD which has genre and tokenized words (obtained from combining the sentence1 and 2 and applying nltk tokenizer on it).

We map each word with genre to make a (key,value) tuple and this RDD “genre\_word\_with\_stopwords” still has stop words in them.

We further create new RDD “genre\_word\_without\_stopwords” by filtering the stop words which we obtained from ntlk library (part of nltk.corpus)

We created a generic function (processPercentageOfWords) to calculate the percentage of words in each genre, which takes the (gerne,word) tuple RDD as input.

The function processPercentageOfWords does the following

* It groups all words by its genre i.e. you would have word as key and list of genres as value.
* It now counts all the genres for each word to get the RDD as (word, list of number of genres the word appeared in)
* It gets the total number of words in our RDD.
* It now swaps the key and values and counting by key(genre) would give us number of words that appeared in each genre
* It now divides the number of words appeared in each genre by total words to get the percentage.

The processPercentageOfWords is called first with “genre\_words\_without\_stopwords” to get the percentages and then we call the same function with “genre\_words\_with\_stopwords” to get the result for the task.

**Results:**

processPercentagesOfWords(genre\_word\_with\_stopwords)

total words in all genres = 85621

Percentage of words in 5 genres - 8.988449095432195

Percentage of words in 4 genres - 6.198245757466042

Percentage of words in 3 genres - 7.953656229196109

Percentage of words in 2 genres - 13.52238352740566

Percentage of words in 1 genres - 63.337265390499994

processPercentagesOfWords(genre\_word\_without\_stopwords)

total words in all genres = 85475

Percentage of words in 5 genres - 8.850541093887102

Percentage of words in 4 genres - 6.204153261187482

Percentage of words in 3 genres - 7.964902018133957

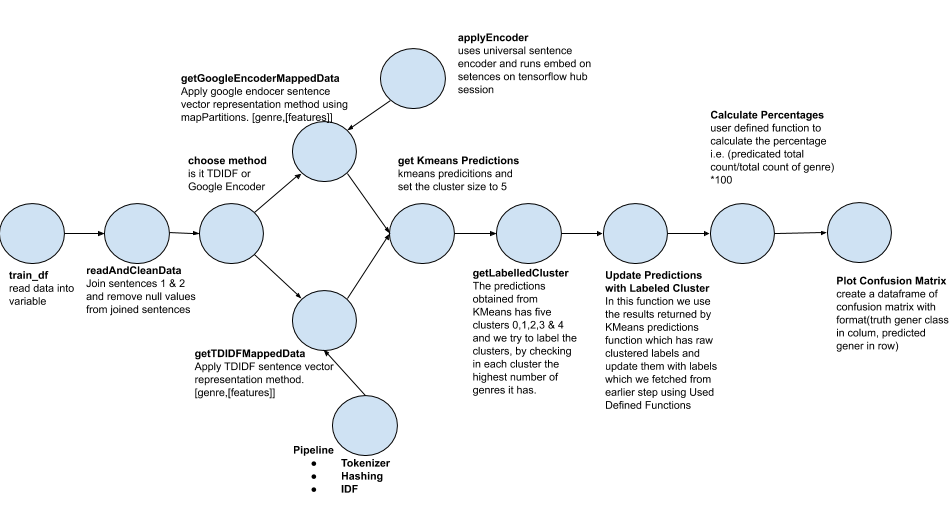
Percentage of words in 2 genres - 13.540801403919275

Percentage of words in 1 genres - 63.43960222287218

**Sentence Vector Exploration:**

**Application Design:**

The following DAG diagram shows the data flow and the sequence of executions



**Implementation Details:**

In this task we opted to create multiple functions to reuse the code.

processSentenceVectorExploration is the main method which is called with different input strings to apply sentence vector representation methods i.e.TDIDF or Google Encoder.

**Cleaning Data:**

* Function to read and clean the data.
* We read the train data set from our s3 location and format them into sql data frame with genre and joined sentence (combining sentence 1&2)
* We clean the data, by removing null entries in the joined sentence column to avoid invalid processing.

**Applying Vector Representation Method:**

* Function to apply a sentence vector representation method based on the input type passed to get the genre feature of sentences.
* If the input type is TDIDF we call getTDIDFFeaturesData method which does the following
  + Tokenize the sentences into words
  + Hashing the words into raw features (number of features is set to 800)
  + And IDF to convert the raw features to features
  + We use pipeline to combine the above steps 1,2 & 3 to fit and transform sentences to features.
* If the input type is GoogleEncoder we call getGoogleEncodedFeaturesData method which does the following
  + It runs the google encoder on the partitioned data set rather on one big single data set.
  + It starts with loading the tensor flow hub i.e. embed.
  + Using tensor flow session, its run embed on list of sentences to get the features results
  + It maps the genre and features after, as google encoder does the features only on list of sentences

**Applying KMeans:**

* Function to apply the Kmeans predication on the genre feature set and divided them into five clusters (set k =5)

**Cluster Labelling:**

* The predictions obtained from KMeans has five clusters 0,1,2,3 & 4 and we try to label the clusters, by checking in each cluster the highest number of genres it has.
* Genre, Predications data frame is grouped by genre and the count is added as new column “genreCount”.
* Now using Window partitionBy on predicition column, we get the max of genreCount for each prediction and add them to a new column maxGenreCount
* We filter the rows which has same maxGenreCount and genreCount and then drop both these columns to get the labels for each cluster.

**Update clusters with labels:**

* In this function we use the results returned by KMeans predicitons function which has raw clustered labels and update them with labels which we fetched from earlier step using Used Defined Functions
* Now we group by prediction and genre to get the count column to see the number of times the genre appeared for that prediction.

**Calculate Percentage:**

* Function to calculate the final percentages of true label against the predicated label.
* Used user defined function to calculate the percentage i.e. (predicated total count/total count of genre) \*100

**Plot Confusion Matrix**:

* Using pandas plot the confusion matrix.

**Results:**

**TDIDF**

travel slate fiction government telephone

travel 57.57% 36.71% 10.61% 54.47% 3.71%

slate 0.00% 0.00% 0.00% 0.00% 0.00%

fiction 39.37% 56.37% 82.01% 37.37% 39.24%

government 2.22% 2.63% 0.79% 7.18% 0.48%

telephone 0.83% 4.29% 6.58% 0.97% 56.58%

**Universal Sentence Encoder -** \* on subset of data

travel slate fiction government telephone

travel 82.80% 5.65% 5.47% 1.42% 3.54%

slate 5.91% 48.02% 25.37% 4.74% 11.06%

fiction 2.69% 7.91% 59.20% 0.95% 10.18%

government 1.08% 25.42% 1.99% 90.05% 4.42%

telephone 7.53% 12.99% 7.96% 2.84% 70.80%

**Performance Analysis:**

For working on performance evaluation between executor number and memory, an experiment will be designed for working on:

1. small number of executors with large memory
2. large number of executors with small memory

Following config values are the ones we experimented with

**spark.executor.memory** - Size of memory to use for each executor that runs the task, by choosing the master and core instance which have more memory.

**spark.executor.cores** - Number of virtual cores are increased by increasing the number of core instances

executor-cores and executor-memory params play an especially important role in spark performance as they control the amount of CPU & memory your spark application gets.

Ways to edit the above config

* SSH to master node
* EDIT spark-defaults.conf file to set the above values ex: sudo vi /etc/spark/conf/spark-defualts.conf

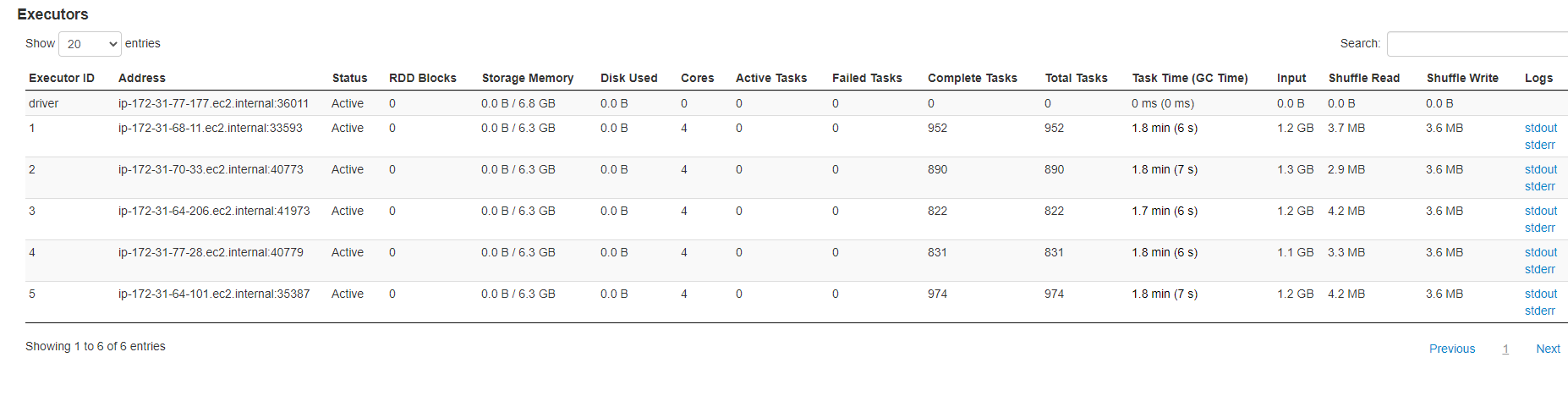
By referring to AWS M5 instance(<https://aws.amazon.com/cn/ec2/instance-types/m5/>), m5.xlarge have 4 Vcpu with 16G memory, m4.large have 8 Vcpu with 16G memory, by exploring on the two designed case, the following experiment group will be created:

* 1 master node with 7 core node cluster based on m5.xlarge in each node(base)
* 1 master node with 7 core node cluster based on m4.large in each node
* 1 master node with 5 core node cluster based on m5.xlarge in each node

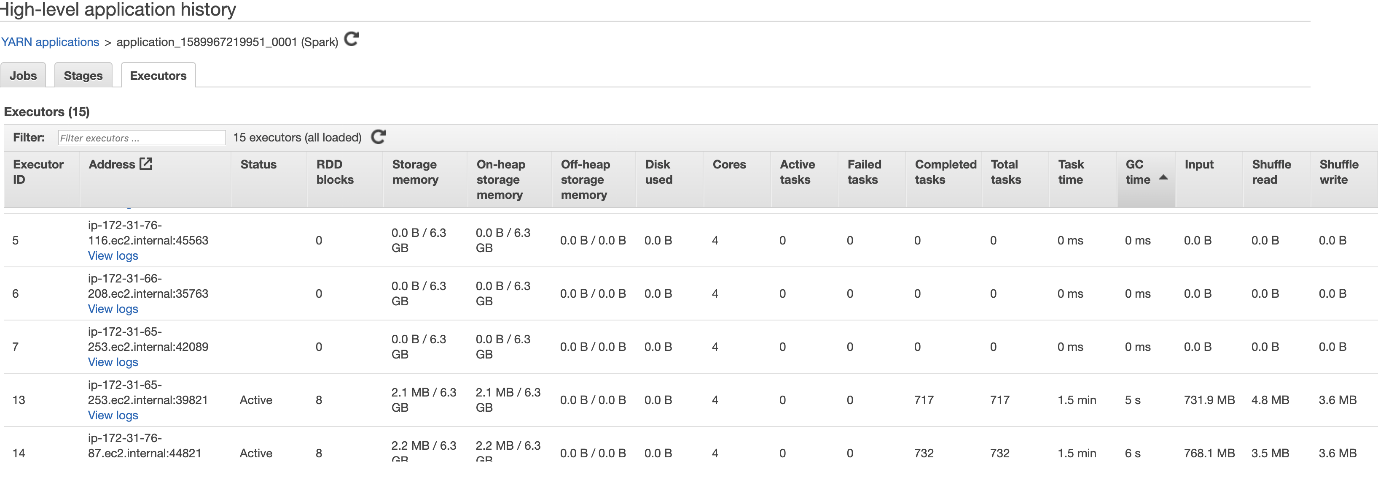
By running the above three experiment group with same notebook, following are the observations

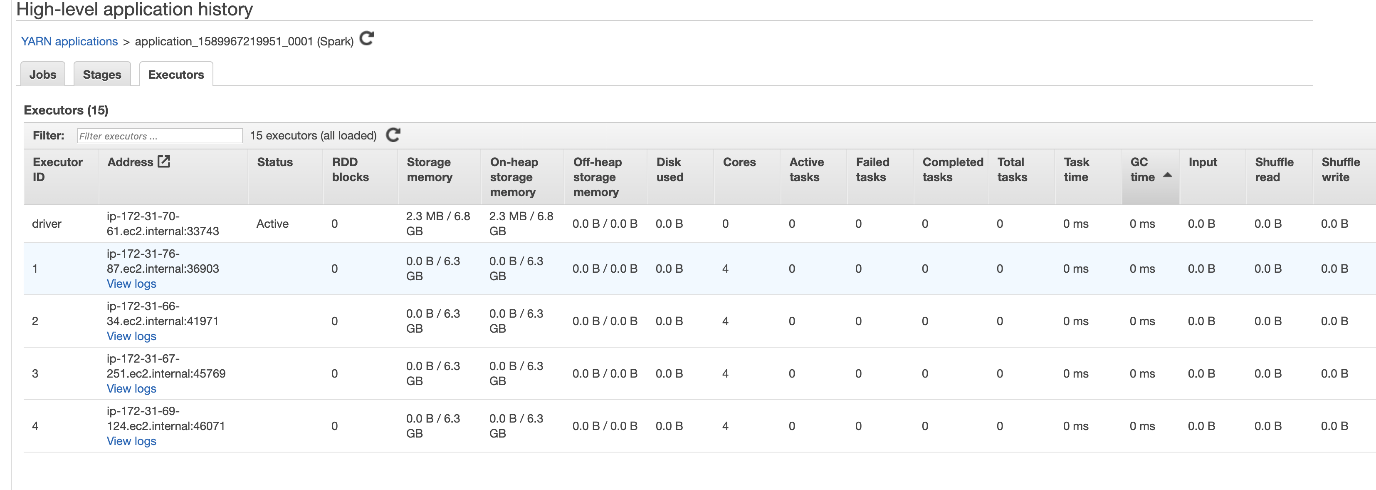
* Total time taken to execute the TDIDF sentence vector exploration with 1 driver and 5 cores was the same with 1 master and 7 cores
* 1 Master 5 Core created 1 driver and 5 executors whereas 1 Master, 7 Core created 1 driver and 16 executors, but had only 4 cores to execute.

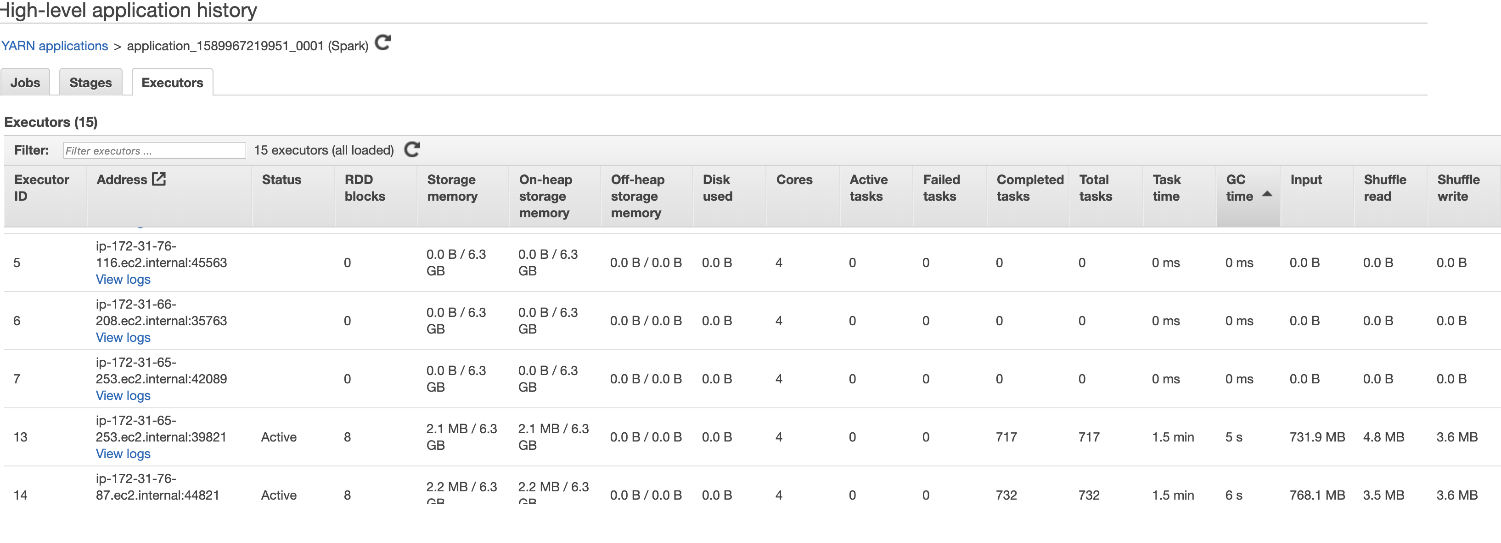
1 master and 5 cores(m5.xlarge)



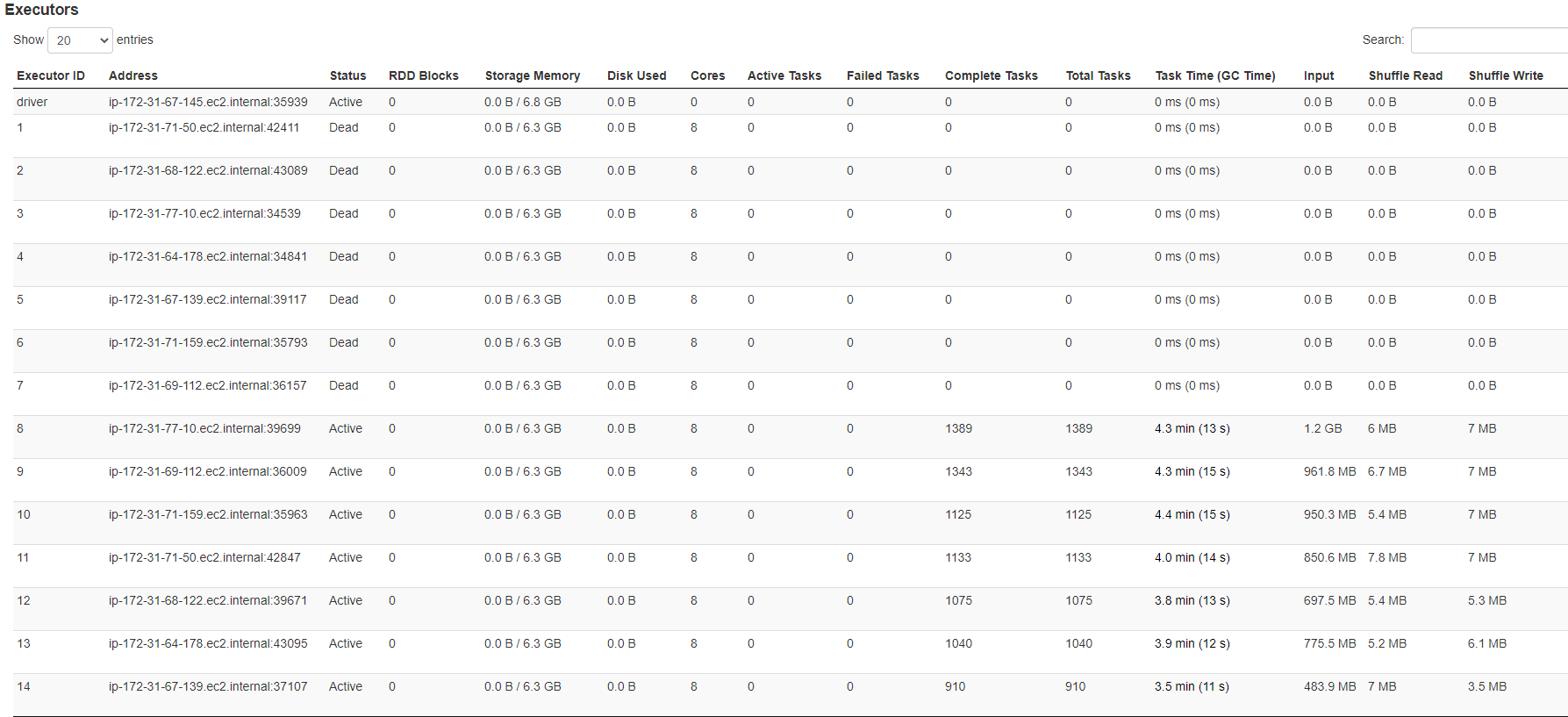
1 master and 7 cores (m5.xlarge)







1 master and 7 cores (m4.xlarge)



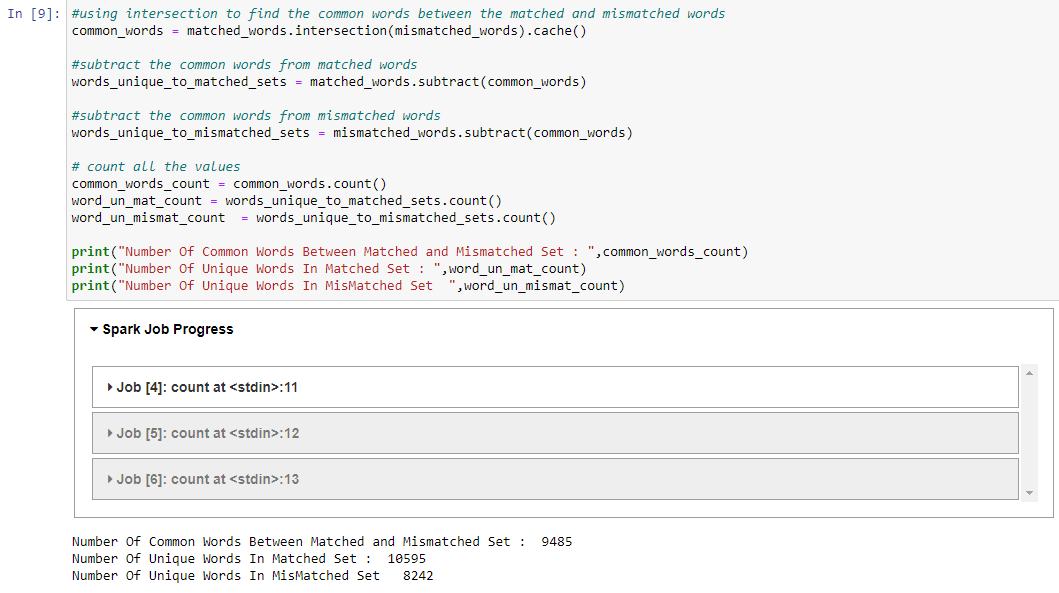
**Conclusion:**

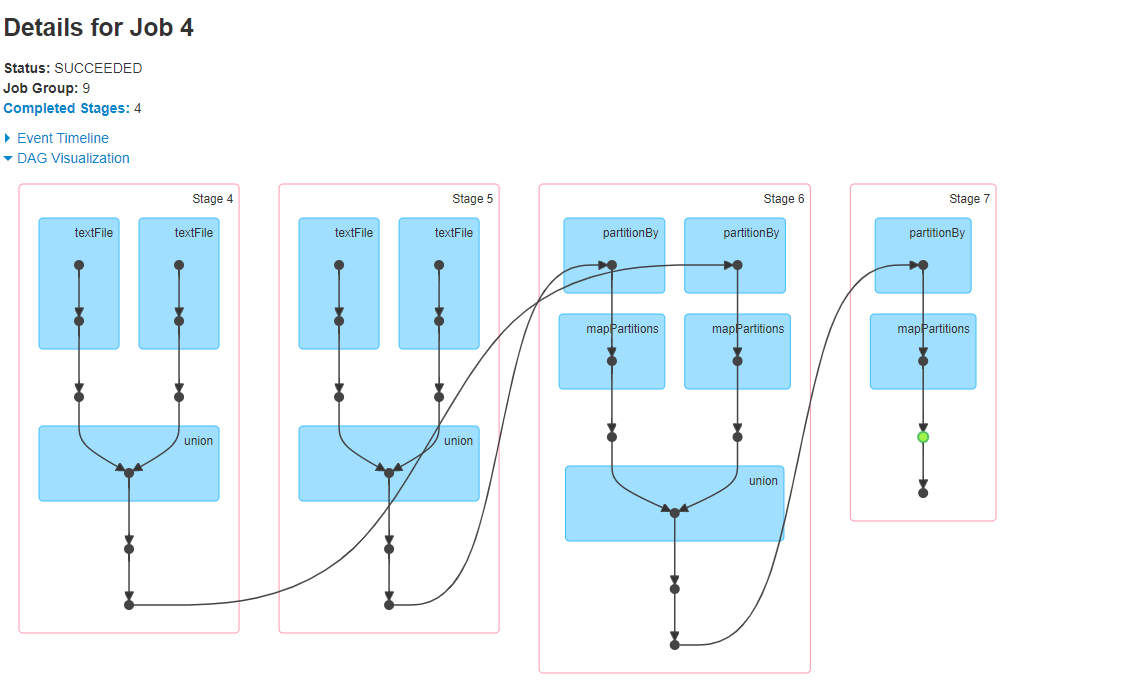
In conclusion, this project is already running well in the AWS EMR cluster in an efficient way, it might be a bit complex in config, but much easier to save time on processing data. Found different types of vector ending method have different set of hardware requirements ex: TDIDF performed well compare to Google Encoder on large data sets.

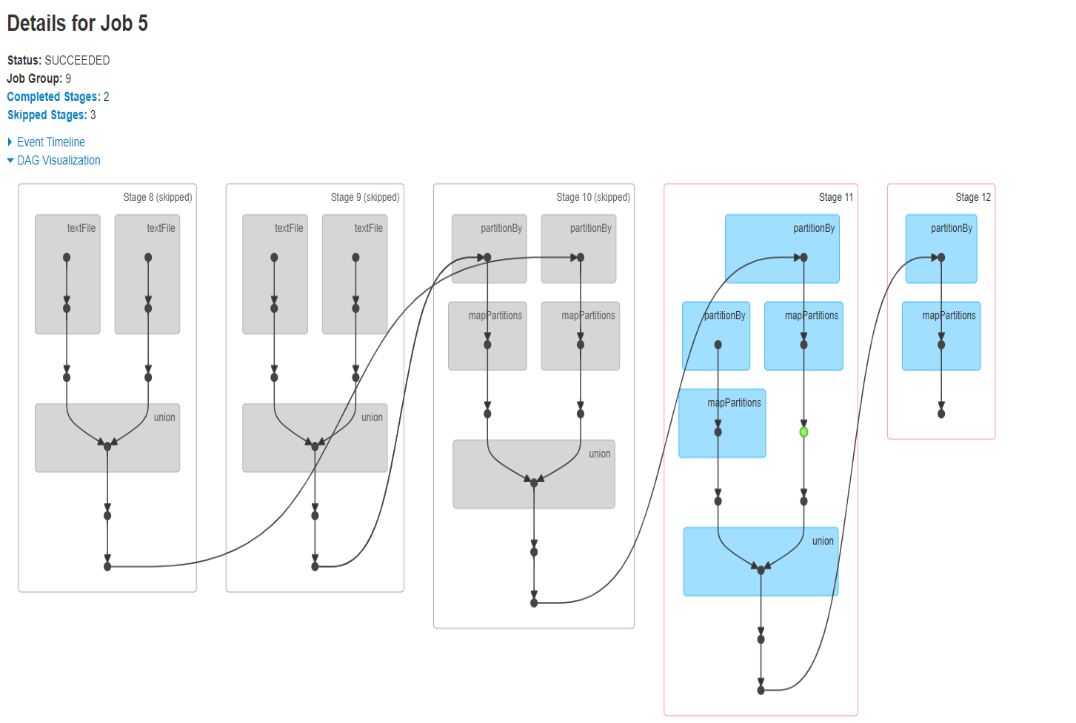
Spark technology have specific performance to resolve some complex data issues, both model currently working well to produce result, but for improving it, more data pre-process technology may be used such as remove specific symbols and organize word format, and also use single test set will working better to evaluate model performance, based on our performance experiment, it is clear to show that there is higher performance in parallel computing such as AWS EMR with spark.

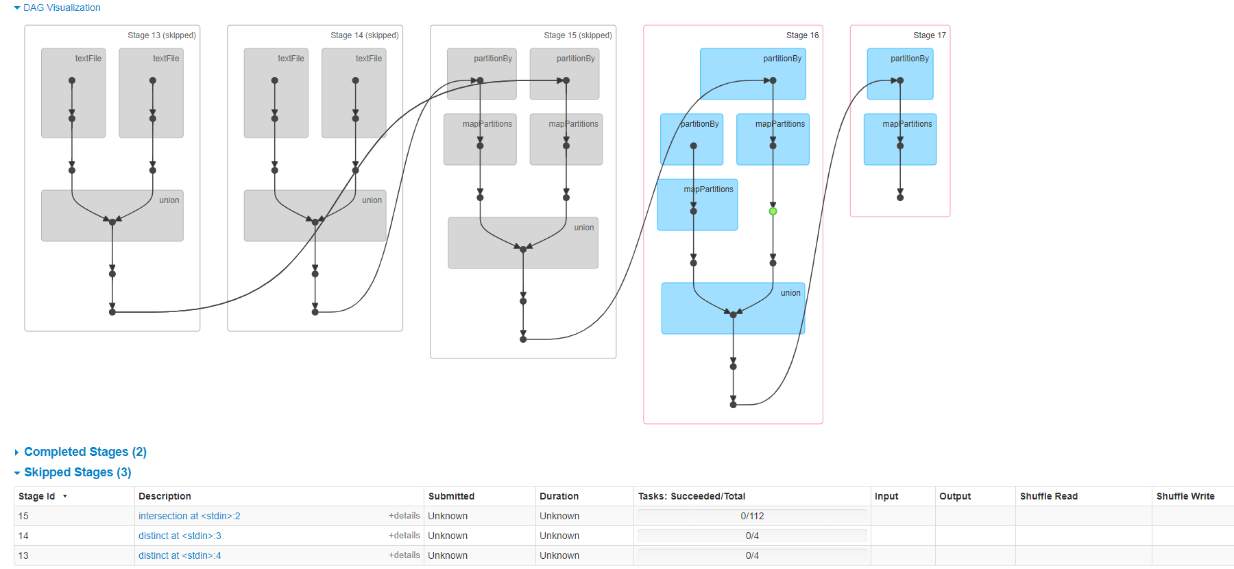
**Appendix:**

**DAG References:** Example DAG visualizations for task1 of vocabulary exploration. The figure shows three jobs 4,5 6 submitted and their captured DAG’s. Skipped stages shows caching of the RDD’s

****

****

****

****