1. Introduction

This report aims to provide a thorough analysis of the Spark application our group has designed and implemented.

During the process of designing and implementing the application, PySpark has been used as the programming language to perform the exploratory data analysis. For the purpose of achieving the most efficient analysis outcomes, the AWS EMR spark cluster has been chosen as the running environment to support our spark application. The program has been processed on a cluster of a single **r5.4xlarge** node.

In order to perform the distributed computation using Pyspark, Spark RDD, Spark dataframe and Spark ML have been used to perform a series of operations. The distributed file system Amazon S3 has been utilised to store the data.

2. Vocabulary Exploration

2.1 Final Results

Results 1:

Requirements	Results
Common words between **matched** & **mismatched** sets	8927
The number of words **unique** to the **matched** sets	11134
The number of words unique to **mismatched** sets	6466

Results 2: Percentage of words appearing

Frequency	Percentage of appearing **(contains stop words)**	Percentage of appearing **(without stop words)**
1	69.5084%	69.6106%
2	11.0784%	11.0901%
3	6.4515%	6.4593%
4	5.2362%	5.2389%
5	7.7255%	7.601%

2.2 Workflow

Part 1 Workflow

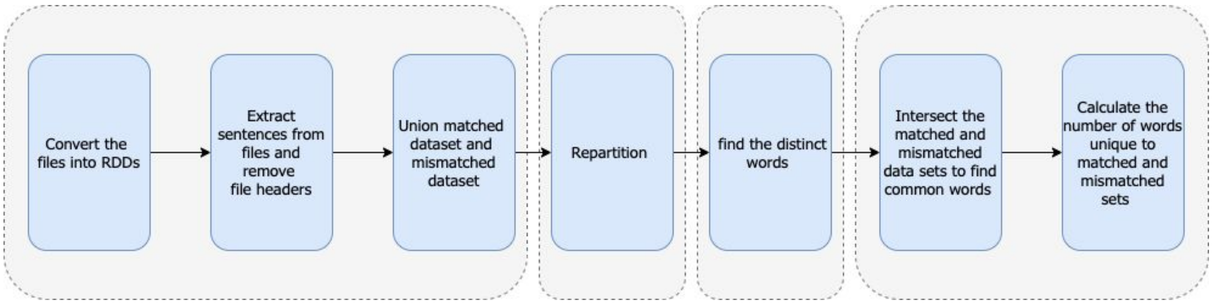

Convert the files into RDDs
Extract sentences from files and remove file headers
Union matched dataset and mismatched dataset
Repartition
find the distinct words
Intersect the matched and mismatched data sets to find common words
Calculate the number of words unique to matched and mismatched sets

Figure 1. Count the matched & mismatched words across four data sets

The workflow of counting the matched and mismatched words across four different datasets: dev_matched.tsv, dev_mismatched.tsv, test_matched.tsv and test_mismatched.tsv has been concluded in figure 1. It can be seen that four shuffle transformations were involved in this process, invoked by the actions of Union(), repartition(), distinct() and intersection() respectively.

Part 2 Workflow

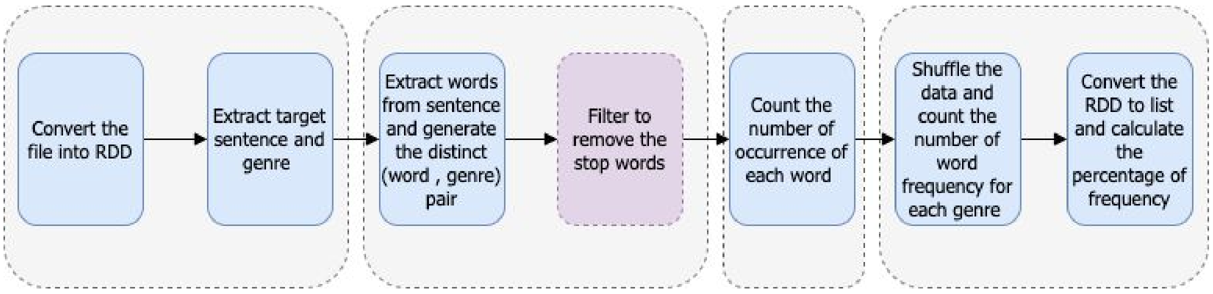

Convert the file into RDD
Extract target sentence and genre
Extract words from sentence and generate the distinct (word , genre) pair
Filter to remove the stop words
Count the number of occurrence of each word
Shuffle the data and count the number of word frequency for each genre
Convert the RDD to list and calculate the percentage of frequency

Figure 2. Calculate the percentage of words appearing

The process of calculating the percentage of words appearing frequencies has been included in figure2 & figure3. It can be seen that there were three shuffle transformations was involved in this process:

- ❏ Generate the distinct (word, genre) pair (by the distinct() action)
- ❏ Count the occurrence of each word under a particular genre (by the reduceByKey() action)
- ❏ Count the number of occurrence of each genre (by the reduceByKey() action)

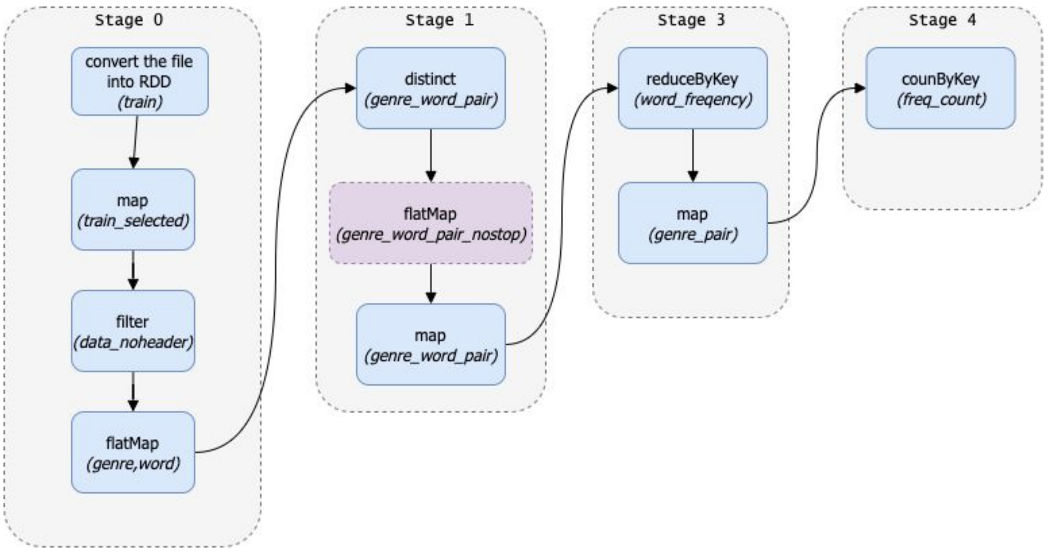
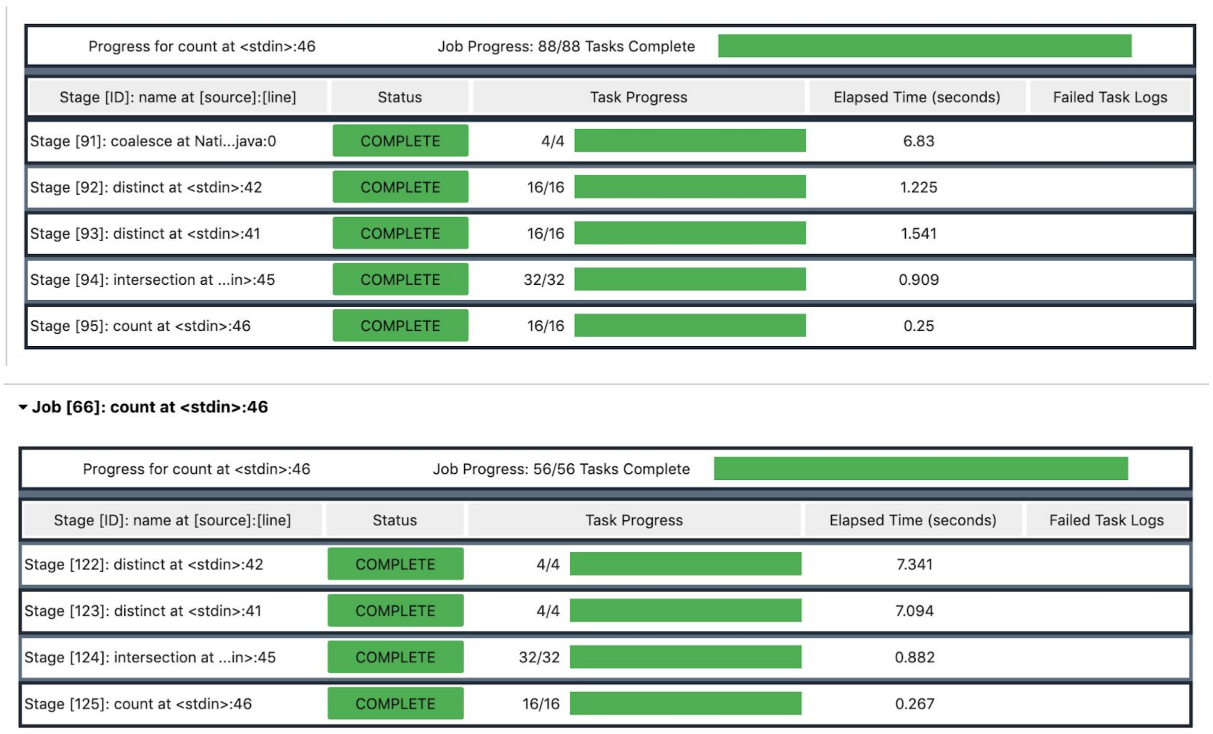


Figure 3. Data flow DAG

2.3 Implemented features

Various programming methodologies have been implemented to improve the permanence of the overall data analytics process.

Repartition



Progress for count at <stdin>:46 Job Progress: 88/88 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [91]: coalesce at Nati...java:0	COMPLETE	4/4	6.83	
Stage [92]: distinct at <stdin>:42	COMPLETE	16/16	1.225	
Stage [93]: distinct at <stdin>:41	COMPLETE	16/16	1.541	
Stage [94]: intersection at ...in>:45	COMPLETE	32/32	0.909	
Stage [95]: count at <stdin>:46	COMPLETE	16/16	0.25	

▾ Job [66]: count at <stdin>:46

Progress for count at <stdin>:46 Job Progress: 56/56 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [122]: distinct at <stdin>:42	COMPLETE	4/4	7.341	
Stage [123]: distinct at <stdin>:41	COMPLETE	4/4	7.094	
Stage [124]: intersection at ...in>:45	COMPLETE	32/32	0.882	
Stage [125]: count at <stdin>:46	COMPLETE	16/16	0.267	

Figure 4. Execution statistics with(first) / without(second) the action of repartition()

It can be seen from figure 4, the action of repartition() significantly improved the program performance by reducing the execution time from 15.584 seconds to 4.006 seconds.

Closure

The stop words list variable has been broadcasted to every node using efficient broadcast algorithms to reduce communication costs. In our case, the stop words list has been broadcasted to every node. The session below shows this process in details:

```
stop_words = stopwords.words('english')
stop_words = sorted(stop_words)
broardcast_stopwords = sc.broadcast(stop_words)
```

Figure 5. Create the broadcast variable

```
stopwords_value = broardcast_stopwords.value
```

Figure 6. Receive the broadcast variable values

Binary search on stop words

The binary search technique has been utilised to remove the stop words from the given list of words. Before performing the binary search, the stop words list has been sorted in ascending order. The flow chart of the searching process is shown in figure 7.

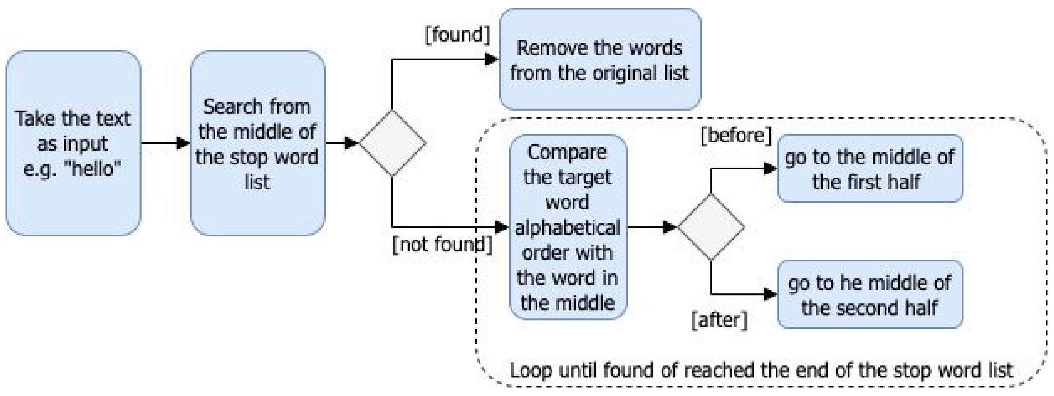

Take the text as input e.g. "hello"
Search from the middle of the stop word list
[found]
Remove the words from the original list
[not found]
Compare the target word alphabetical order with the word in the middle
[before]
go to the middle of the first half
[after]
go to he middle of the second half
Loop until found of reached the end of the stop word list

Figure 7. Binary search on stop words workflow

4. Sentence Vector Exploration

4.1 Method 1: TF-IDF Based Vector Representation

Term frequency-inverse document frequency (TF-IDF) feature vectorization method has been implemented to generate the sentence features.

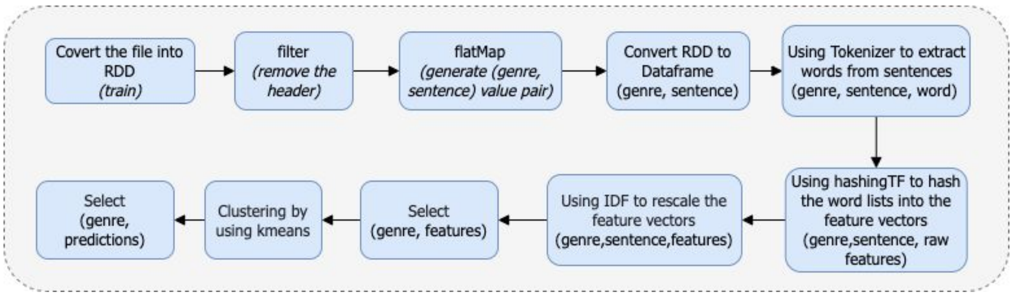

Covert the file into RDD (train)
filter (remove the header)
flatMap (generate (genre, sentence) value pair)
Convert RDD to Dataframe (genre, sentence)
Using Tokenizer to extract words from sentences (genre, sentence, word)
Using hashingTF to hash the word lists into the feature vectors (genre,sentence, raw features)
Using IDF to rescale the feature vectors (genre,sentence,features)
Select (genre, features)
Clustering by using kmeans
Select (genre, predictions)

Figure 8. The workflow of the TF-IDF Based Vector Presentation process

During the sentence vector exploration process, the (sentence, genre) pair has been extracted from the given dataset. Then, Tokenizer was used to split sentences into words. For each sentence, HashingTF has been used to hash the list of words into a feature vector. Finally, IDF has been utilised to rescale the feature vectors, which improved the performance when using text as features.

4.2 Method 2: Pre-trained Sentence Encoder

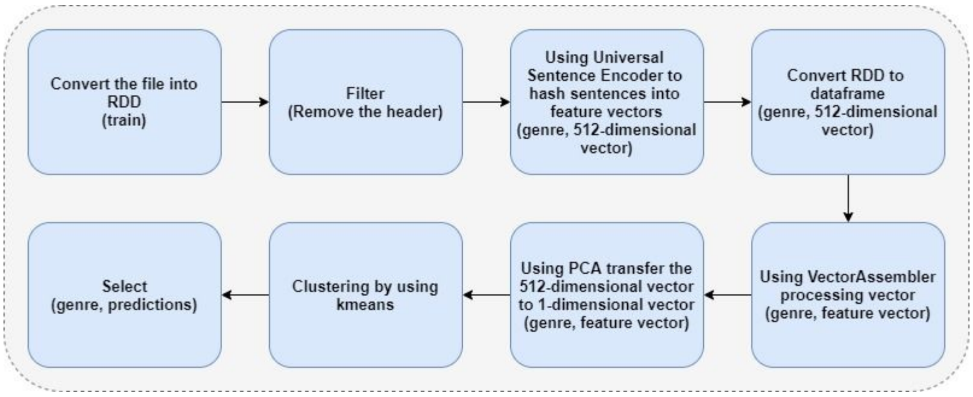

Convert the file into RDD (train)
Filter (Remove the header)
Using Universal Sentence Encoder to hash sentences into feature vectors (genre, 512-dimensional vector)
Convert RDD to dataframe (genre, 512-dimensional vector)
Using VectorAssembler processing vector (genre, feature vector)
Using PCA transfer the 512-dimensional vector to 1-dimensional vector (genre, feature vector)
Clustering by using kmeans
Select (genre, predictions)

Figure 9. The workflow of the Pre-trained Sentence Encoder process

The encoding process by using the pre-trained sentence encoder has been shown in figure 9. Firstly, the encoder hashed the sentences into the raw feature vectors, which consisted of 512-dimensional vectors. Then, the VectorAssembler processing vector and Principal component analysis(PCA) have been used to convert the hashing results into a one-dimensional vector; prior to this process, the original RDD has been converted to a dataframe.

4.3 Clustering

Spark Machine learning K-means algorithm has been adopted as a clustering approach in order to divide the feature vectors into distinct groups based on some common characteristics. The K number has been set to 5, which means that the cluster has divided the input features into five prediction groups.

```
kmeans = KMeans().setK(5).setSeed(1)
model = kmeans.fit(feature)
```

Figure 10. K-means clustering

Matrix 1 - TF-IDF Based Vector

Based on the clustering results, two confusion matrices have been presented for better visualisation of the results.

Prediction/True	fiction	government	slate	telephone	travel
0	88.33%	65.03%	76.15%	68.51%	65.01%
1	4.84%	7.16%	7.1%	2.49%	7.33%
2	0%	0%	0%	4.13%	0%
3	4.85%	27.62%	15.89%	2.07%	27.59%
4	0.98%	0.19%	0.02%	22.8%	0.07%

Matrix 2 - Pre-trained Sentence Encoder

Prediction/True	fiction	telephone	travel	government	slate
0	46.17%	4.86%	3.89%	3.58%	13.86%
1	14.37%	56.58%	1.04%	1.79%	8.52%
2	10.66%	2.91%	50.71%	30.47%	27.52%
3	2.85%	7.06%	37.77%	50.89%	23.24%
4	25.95%	28.59%	6.59%	13.26%	26.86%

The confusion matrix above described the performance of the Kmeans cluster on a set of data for which the true values are known. The number of correct and incorrect predictions are summarised with count values and classified into five different prediction groups. The true positive predictions have been highlighted in grey.

By comparing matrix 1 and matrix 2, it can be seen that the prediction in matrix 2 is more accurate than it's in matrix 1. Based on the results from matrix 1, all genre types were classified into the same prediction group, conversely, each prediction group had a unique match with the true value (genre) in matrix 2. In a nutshell, the pre-trained sentence encoder performs better than the TF-IDF based vector in this case.

5. Performance Evaluation

Performance Evaluation on Vocabulary Exploration Part 2

The table below summarises the differences between the two execution environments:

Settings/Environment	Environment 1	Environment 2
Execution Time In total	72.4 s	42s
Number of executor used	2	4
Number of threads	40	136
Total shuffling size	35M	31.8M

Execution Time (Each stage)

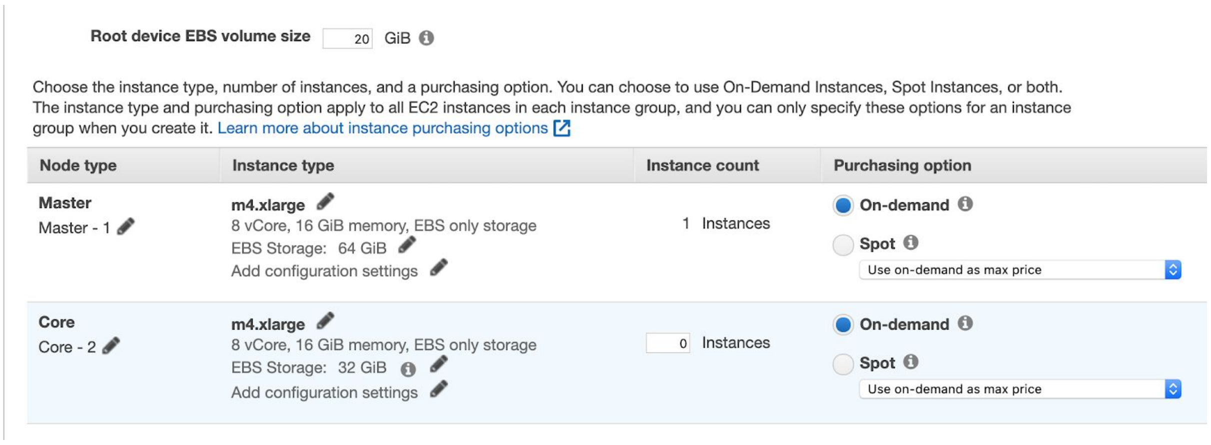
It can be seen that the application's execution time under environment 2 is approximately one time faster than when it's under environment 2. There were four stages involved in the application execution process; In stage 0, a text file has been converted to a resilient distributed dataset(RDD), it can be seen that it has taken approximately ten seconds under both environment 1 and environment 2. (Please see details in Appendix 3 & Appendix 8). Stage 1 of the job involved extracting the distinct key-value pair from the input data. Based on the execution statistics, it can be seen that the application under environment 2 processed two times faster than when it was under environment 1. Stage 2 of the job was invoked by the execution of ReduceByKey(), it can be seen that the application had a similar execution time under both environments, the major reason was that the data was reshuffled/repartitioned in stage two through the distinct() execution, data has the similar structure has been partitioned into one node, hence, this has reduced the overall execution time. Similarly, the execution time in stage 3 which was initiated by the countByKey() action had a similar execution time under both environments.

Number of executors & threads

There were four executors used under environment 2, which was twice of the executors used under environment 1. The same amount of jobs has been divided into a larger number of nodes, thus, a faster application execution time. (please see more details in Appendix 2 & Appendix 6).

6. Appendix

Appendix 1. Cluster setup of environment 1 :



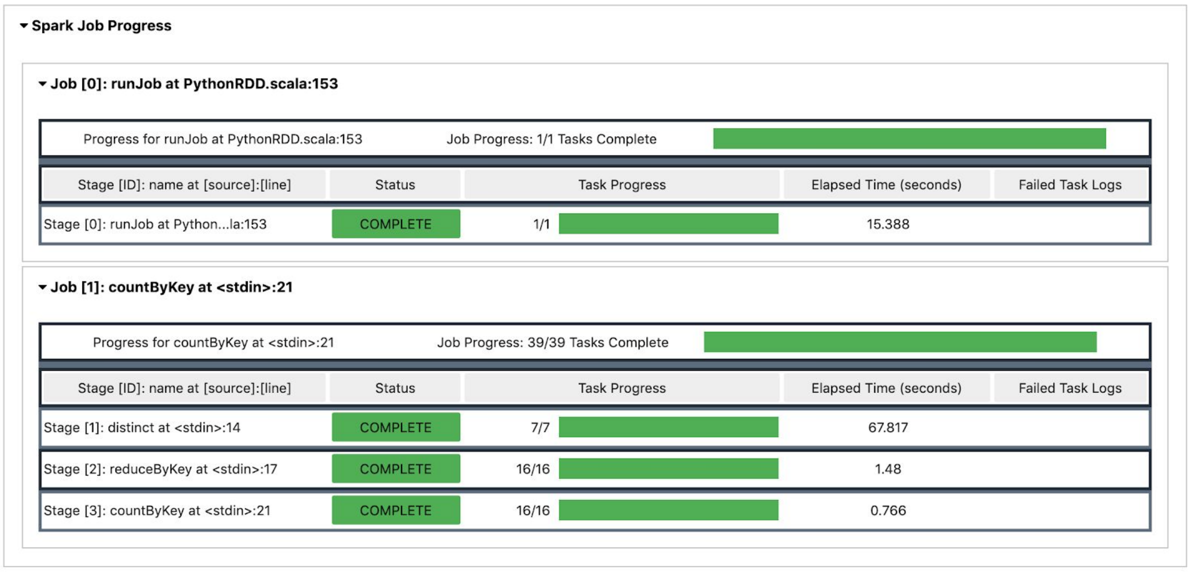
Root device EBS volume size 20 GiB

Choose the instance type, number of instances, and a purchasing option. You can choose to use On-Demand Instances, Spot Instances, or both. The instance type and purchasing option apply to all EC2 instances in each instance group, and you can only specify these options for an instance group when you create it. Learn more about instance purchasing options

Node type	Instance type	Instance count	Purchasing option
Master Master - 1	**m4.xlarge** 8 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB Add configuration settings	1 Instances	[✓ On-demand] Spot Use on-demand as max price
Core Core - 2	**m4.xlarge** 8 vCore, 16 GiB memory, EBS only storage EBS Storage: 32 GiB Add configuration settings	0 Instances	[✓ On-demand] Spot Use on-demand as max price

The program has been processed on a cluster of one m4.xlarge node (one master node) .

Appendix 2. Execution statistics of environment 1 :



Spark Job Progress

Job [0]: runJob at PythonRDD.scala:153

Progress for runJob at PythonRDD.scala:153 Job Progress: 1/1 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [0]: runJob at Python...la:153	COMPLETE	1/1	15.388	

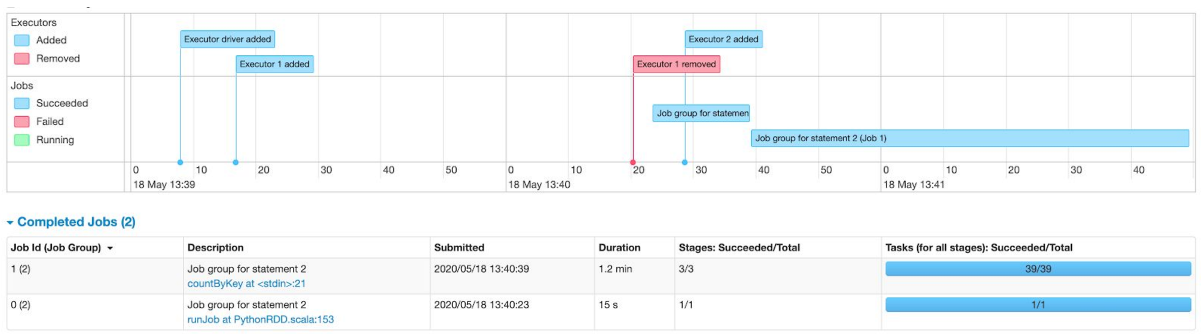
Job [1]: countByKey at <stdin>:21

Progress for countByKey at <stdin>:21 Job Progress: 39/39 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [1]: distinct at <stdin>:14	COMPLETE	7/7	67.817	
Stage [2]: reduceByKey at <stdin>:17	COMPLETE	16/16	1.48	
Stage [3]: countByKey at <stdin>:21	COMPLETE	16/16	0.766	

Stage 1 had the heaviest workload. In this stage, the distinct key-value pair has been found.

Appendix 3. Execution timeline of environment 1 :

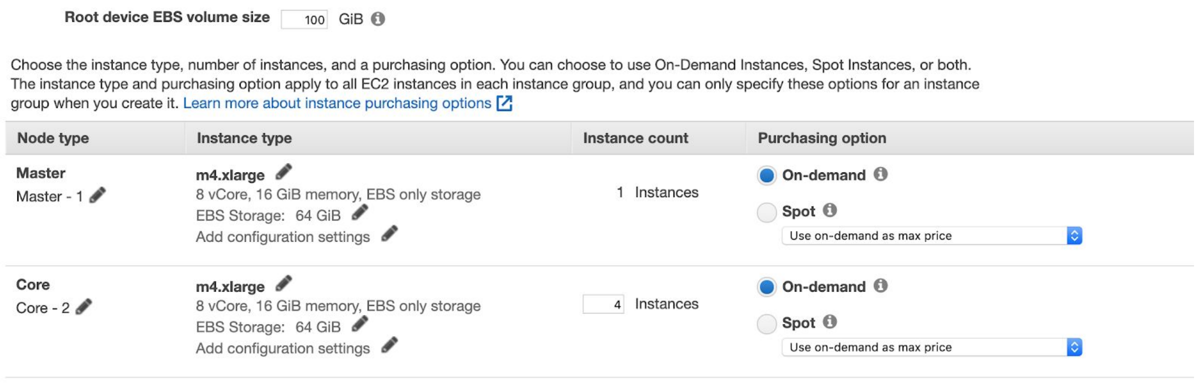


Completed Jobs (2)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1 (2)	Job group for statement 2 countByKey at <stdin>:21	2020/05/18 13:40:39	1.2 min	3/3	39/39
0 (2)	Job group for statement 2 runJob at PythonRDD.scala:153	2020/05/18 13:40:23	15 s	1/1	1/1

The total execution time was 1.35 minutes and two executors have been used to perform the tasks.

Appendix 4. Cluster setup of environment 2 :



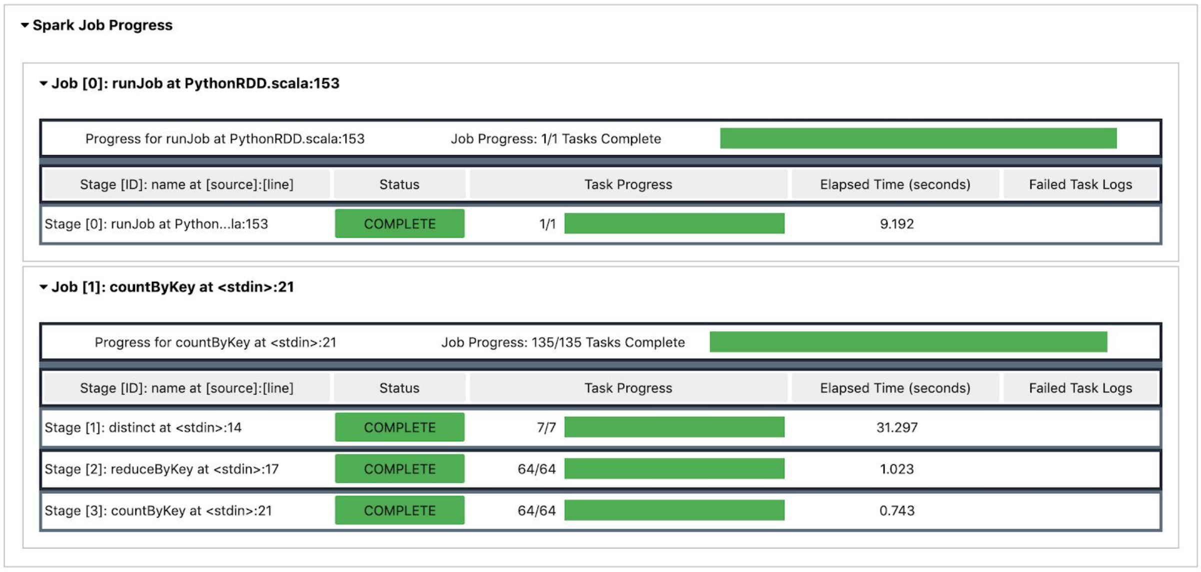
Root device EBS volume size 100 GiB

Choose the instance type, number of instances, and a purchasing option. You can choose to use On-Demand Instances, Spot Instances, or both. The instance type and purchasing option apply to all EC2 instances in each instance group, and you can only specify these options for an instance group when you create it. Learn more about instance purchasing options

Node type	Instance type	Instance count	Purchasing option
Master Master - 1	**m4.xlarge** 8 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB Add configuration settings	1 Instances	[✓ On-demand] Spot Use on-demand as max price
Core Core - 2	**m4.xlarge** 8 vCore, 16 GiB memory, EBS only storage EBS Storage: 64 GiB Add configuration settings	4 Instances	[✓ On-demand] Spot Use on-demand as max price

The program has been processed on a cluster of five m4.xlarge nodes (one master node and four slave nodes).

Appendix 5. Execution statistics of environment 2 :



Spark Job Progress

Job [0]: runJob at PythonRDD.scala:153

Progress for runJob at PythonRDD.scala:153 Job Progress: 1/1 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [0]: runJob at Python...la:153	COMPLETE	1/1	9.192	

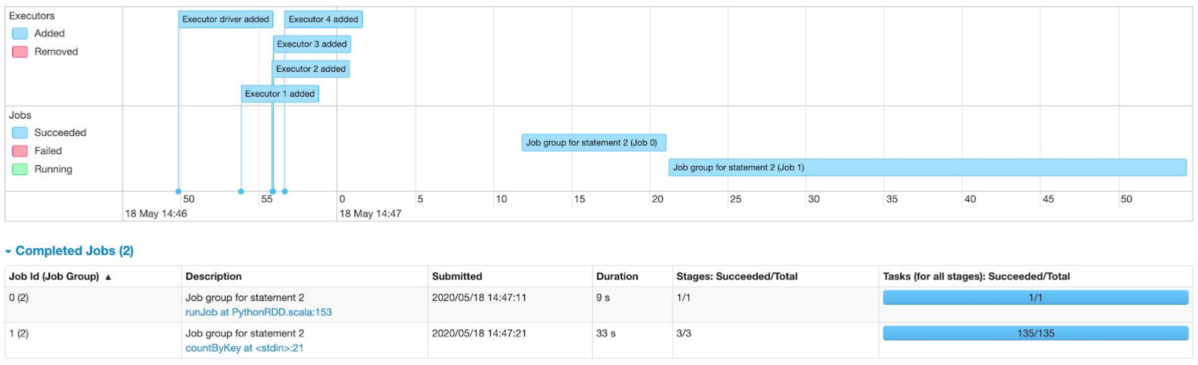
Job [1]: countByKey at <stdin>:21

Progress for countByKey at <stdin>:21 Job Progress: 135/135 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [1]: distinct at <stdin>:14	COMPLETE	7/7	31.297	
Stage [2]: reduceByKey at <stdin>:17	COMPLETE	64/64	1.023	
Stage [3]: countByKey at <stdin>:21	COMPLETE	64/64	0.743	

Stage 1 had the heaviest workload. In this stage, the distinct key-value pair has been found.

Appendix 6. Execution timeline of environment 2 :



Completed Jobs (2)

Job Id (Job Group) ▲	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0 (2)	Job group for statement 2 runJob at PythonRDD.scala:153	2020/05/18 14:47:11	9 s	1/1	1/1
1 (2)	Job group for statement 2 countByKey at <stdin>:21	2020/05/18 14:47:21	33 s	3/3	135/135

The Total execution time was 42 seconds and four executors have been used to perform the tasks.