Practical Work

Using AmazonEMR

Submitted by:
Priyanka KUMAR

Course:
Big Data Infrastructure & Cloud Computing

Table of Contents

Introduction

Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS to process and analyze vast amounts of data. By using these frameworks and related open-source projects, such as Apache Hive and Apache Pig, you can process data for analytics purposes and business intelligence workloads. Additionally, you can use Amazon EMR to transform and move large amounts of data into and out of other AWS data stores and databases, such as Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.

(Source: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html)

This practical aims to perform a word count example on Amazon EMR cluster in which "The complete works of Shakespeare" will be used as data from which further processing to remove unnecessary texts will be performed and creating a data file which contains all the words. This data file will be further used for performing various operations which included in detail in this report.

Technical Details:

IDE Used: PyCharm Community Edition 2020.3.2

Documentation for Spark: https://spark.apache.org/docs/latest/

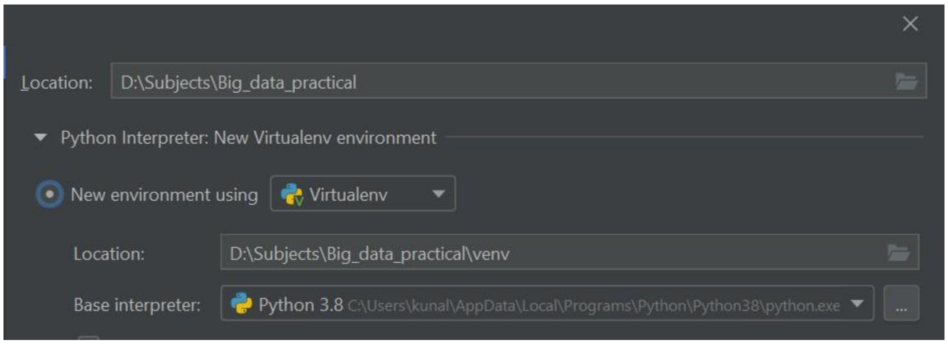
Browser proxy used: foxyproxy

Project Code available at:

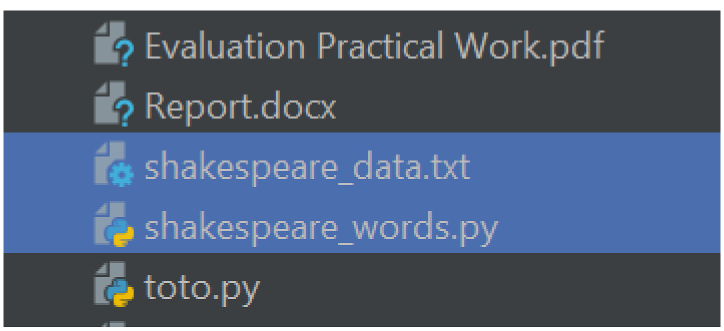
https://github.com/kunalpatz/Big_data_Final_Evaluation.git

Steps

- Create a virtual environment in PyCharm and set it up to use Python 3.8.



- Write a script in Python shakespeare_words.py which generates a list of the words contained in the complete works of Shakespeare and output it to a file. Name this file shakespeare_data.txt.



(The script named "shakespeare_words.py" and "shakespeare_data.txt" are created.)

While creating data file below processing is done:

1. Remove non alphanumeric characters
2. Remove non related paragraphs and words such as License and citations
3. Remove the words such as ACT, SCENE to get only precise content for the book.
4. Remove stop words to avoid to get only these words as highest occurrence.

- Install PySpark using the command line.

```
Terminal:   Local ×  +
Microsoft Windows [Version 10.0.18363.1316]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\Subjects\BigDATA_Practicals>pip install pyspark
```

(Installation was already done on the local)

- Test the installation by running the PySpark shell

```
To adjust logging level use sc.setLogLevel(newLevel). For S
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Python version 3.8.7 (tags/v3.8.7:6503f05, Dec 21 202
SparkSession available as 'spark'.
>>> 21/02/07 15:07:01 WARN ProcfsMetricsGetter: Exception w

>>>
```

- Load shakespeare_data.txt file in the shell and count the number of words in the file.

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Python version 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020 17:59:51)
SparkSession available as 'spark'.
>>> df = spark.read.text('shakespeare_data.txt')
21/02/10 17:23:31 WARN ProcfsMetricsGetter: Exception when trying to comp
>>> df.count()
390293
>>>
```

- Exit the shell

```
>>> exit()

D:\Subjects\BigDATA_Practicals\Final>SUCCESS: The process with PID 128064 (child process of PID 130432) has been terminated.
SUCCESS: The process with PID 130432 (child process of PID 83624) has been terminated.
SUCCESS: The process with PID 83624 (child process of PID 91724) has been terminated.
```

- Create a word_count.py file and do the same thing like with the shell (reading the text file and displaying the number of words). For that, you need to create an instance of spark in your script using a SparkSession.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('WordCountApp').getOrCreate()

df = spark.read.text('shakespeare_data.txt')

df.show()
```

- Then add spark code to your script in order to show

1. The first three values in the text file

```
# First three values in text file
print(df.take(3))
```

2. The 10 longest words, showing their length

```
# 10 longest words and their length
long = spark.sql('SELECT word, length(word) AS len FROM data ORDER BY len DESC LIMIT 10')
long.show()
```

3. The 10 words having the highest number of occurrences, with their number of occurrences

```
# 10 highest occurred words and count of occurrence
occurrence = spark.sql(
    "SELECT word, count(word) AS word_count FROM data GROUP BY word ORDER BY word_count DESC LIMIT 10")
occurrence.show()
```

- Run your script locally and observe the results.

Results for each question above are answered below:

1. The first three values in the text file

```
[Row(value='desire'), Row(value='fairest'), Row(value='creatures')]
```

2. The 10 longest words, showing their length

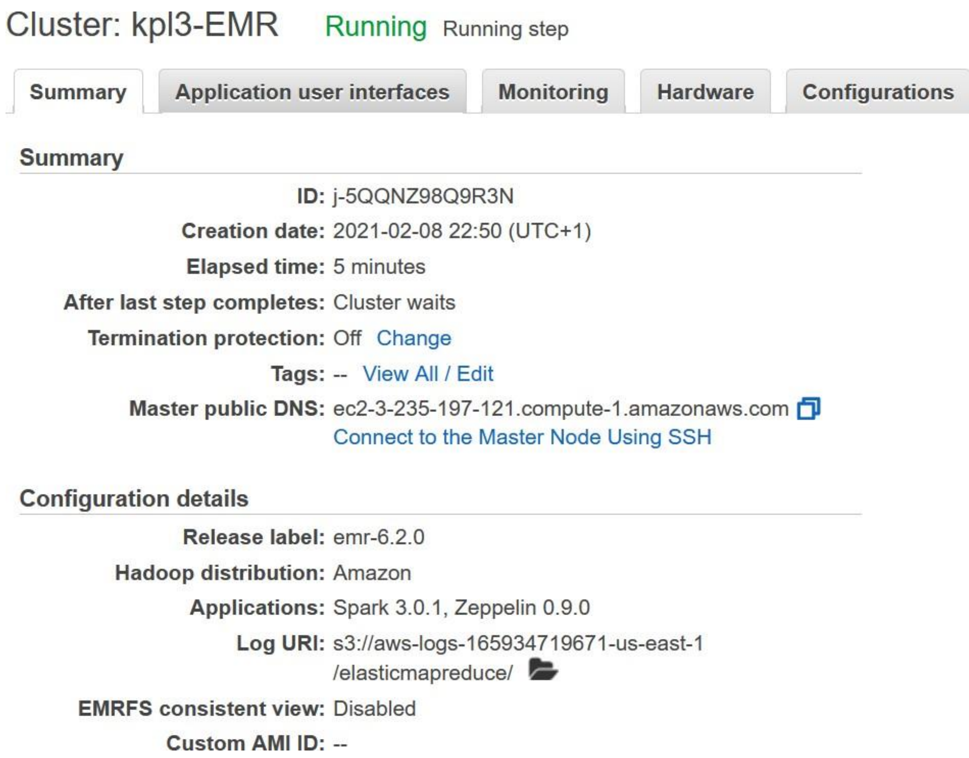
```
+--------------------+---+
|                word|len|
+--------------------+---+
|honorificabilitud...| 27|
|   undistinguishable| 17|
|   undistinguishable| 17|
|   indistinguishable| 17|
|    incomprehensible| 16|
|     enfranchisement| 15|
|     prognostication| 15|
|     interrogatories| 15|
|     interchangeably| 15|
|     notwithstanding| 15|
+--------------------+---+
```

3. The 10 words having the highest number of occurrences, with their number of occurrences

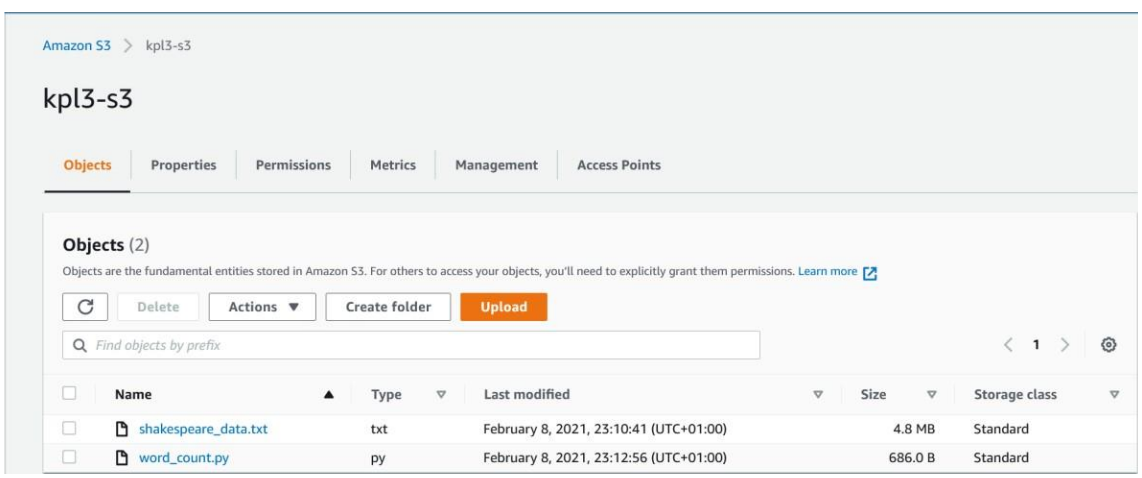
```
+-----+----------+
| word|word_count|
+-----+----------+
| lord|      3094|
| king|      3034|
| good|      2834|
|  sir|      2764|
| come|      2519|
| well|      2240|
| love|      2197|
|  man|      2034|
| hath|      1942|
|enter|      1931|
+-----+----------+
```

Because of removal of stop words, the execution returns the words which are highest occurred in text gives more analysis.

- Create an Amazon EMR cluster (keep default parameters).

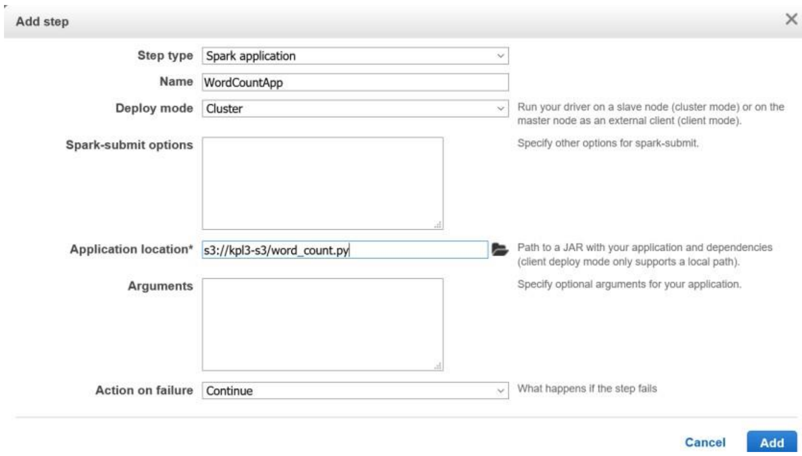


- Create a S3 bucket and upload your data to the bucket.



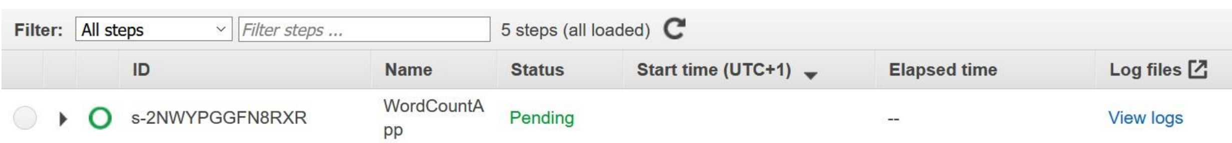
- Open the Spark History Server user interface. Run your script on the cluster using the graphical user interface. Observe the jobs being run in the Spark History Server.

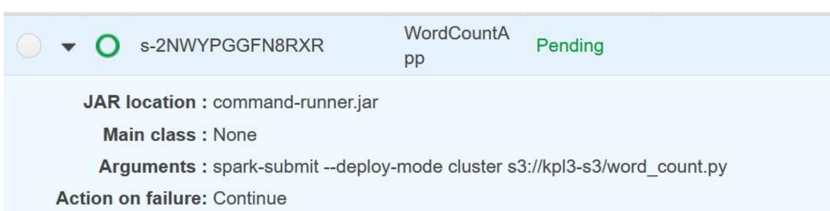
Add Step:



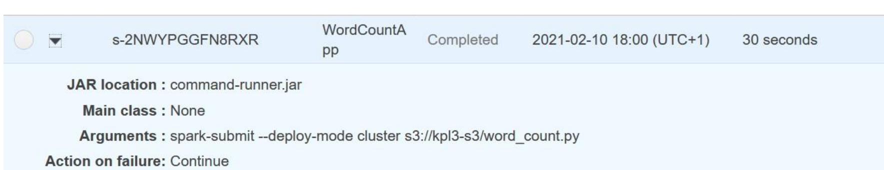
(Step name "WordCountApp" added along with script location in s3 bucket.)

Launching the step:





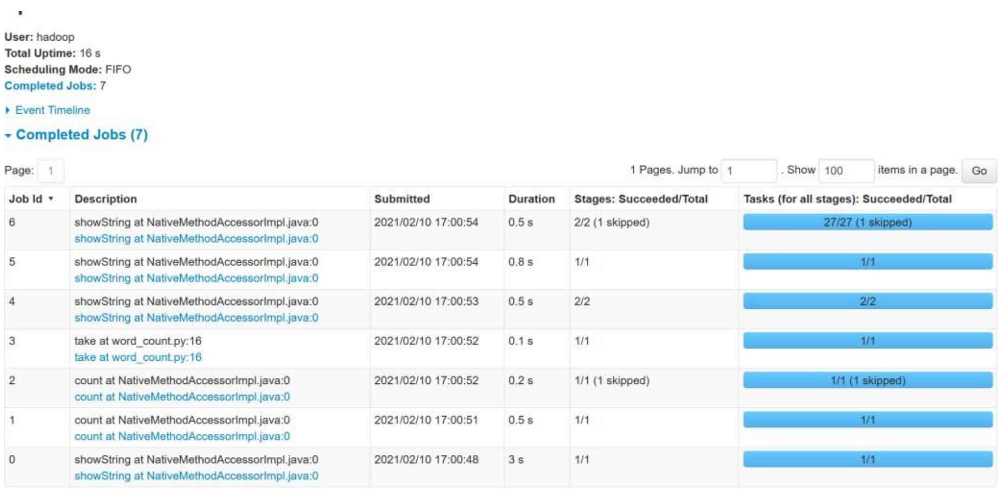
(Screenshot showing details about the step execution status and arguments)



(Screenshot showing step execution completion)

- Observe the final state in Spark History Server.

Observe Spark history Server:


User: hadoop
Total Uptime: 16 s
Scheduling Mode: FIFO
Completed Jobs: 7

Event Timeline

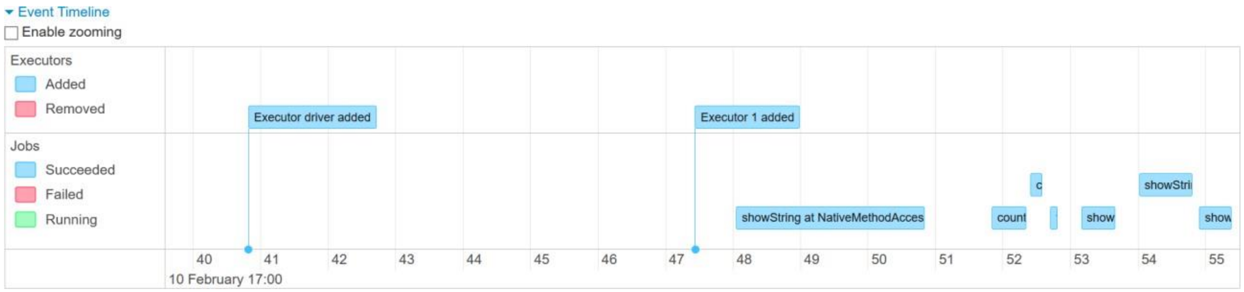
Completed Jobs (7)

Page: 1   1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:54	0.5 s	2/2 (1 skipped)	27/27 (1 skipped)
5	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:54	0.8 s	1/1	1/1
4	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:53	0.5 s	2/2	2/2
3	take at word_count.py:16 take at word_count.py:16	2021/02/10 17:00:52	0.1 s	1/1	1/1
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:52	0.2 s	1/1 (1 skipped)	1/1 (1 skipped)
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:51	0.5 s	1/1	1/1
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:00:48	3 s	1/1	1/1

(Total 27 Tasks ran for execution.)

Observe Event timeline:



(The execution took 16s for this step. This screenshot shows Event for added Executers and succession of the jobs.)

- Propose and test different optimizations to improve the performances. Show the performance gains attained with each optimization tested.

Spark Performance tuning is a process to improve the performance of the Spark and PySpark applications by adjusting and optimizing system resources (CPU cores and memory), tuning some configurations.

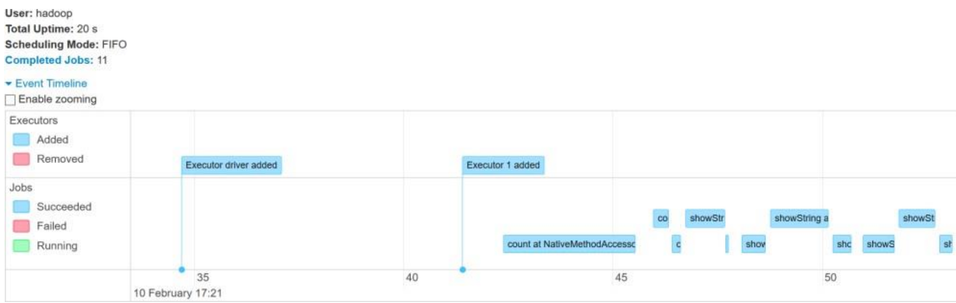
Below are trial runs performed under different implementations wrt the df read function.

Trial 1

Adding repartition:

```
df = spark.read.text('s3://kpl3-s3/shakespeare_data.txt').repartition(10)
```

Added repartition 10.



For the execution it took 20s.

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
10	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:52	0.3 s	1/1 (2 skipped)	26/26 (11 skipped)
9	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:51	0.9 s	1/1 (1 skipped)	10/10 (1 skipped)
8	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:50	0.8 s	1/1	1/1
7	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:50	0.4 s	1/1 (2 skipped)	26/26 (11 skipped)
6	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:48	1 s	1/1 (1 skipped)	10/10 (1 skipped)
5	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:48	0.6 s	1/1	1/1
4	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:47	63 ms	1/1 (1 skipped)	1/1 (1 skipped)
3	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:46	0.9 s	1/1	1/1
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:46	0.2 s	1/1 (2 skipped)	1/1 (11 skipped)
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:45	0.4 s	1/1 (1 skipped)	10/10 (1 skipped)
0	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:21:42	3 s	1/1	1/1

Though 1 task reduced, it created 11 jobs.

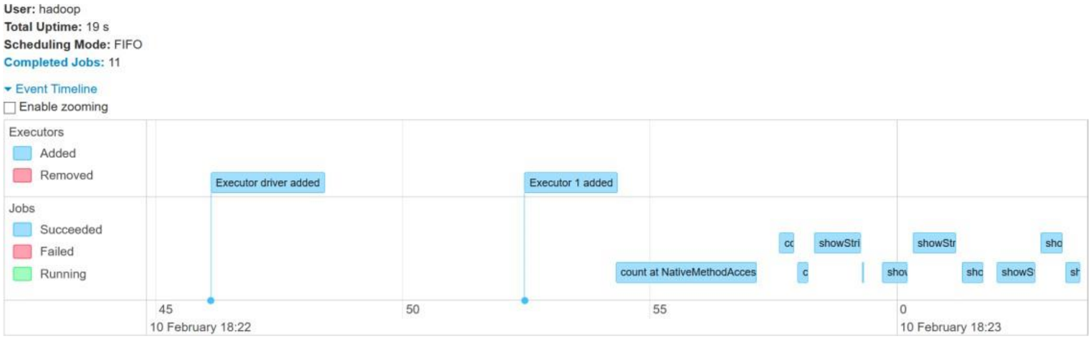
As execution time required is more than previous execution, this can be improved.

Trial 2

Reducing repartition size to 2

```
df = spark.read.text('s3://kpl3-s3/shakespeare_data.txt').repartition(2)
```

This execution took 19s to execute the step



Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
10	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:23:03	0.3 s	1/1 (2 skipped)	26/26 (3 skipped)
9	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:23:02	0.4 s	1/1 (1 skipped)	2/2 (1 skipped)
8	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:23:02	0.8 s	1/1	1/1
7	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:23:01	0.4 s	1/1 (2 skipped)	26/26 (3 skipped)
6	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:23:00	0.9 s	1/1 (1 skipped)	2/2 (1 skipped)
5	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:59	0.5 s	1/1	1/1
4	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:59	36 ms	1/1 (1 skipped)	1/1 (1 skipped)
3	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:58	0.9 s	1/1	1/1
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:57	0.2 s	1/1 (2 skipped)	1/1 (3 skipped)
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:57	0.3 s	1/1 (1 skipped)	2/2 (1 skipped)
0	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 18:22:54	3 s	1/1	1/1

The time of execution was more with repartition. The repartition re-distributes the data from all partitions which usually is full shuffle leading to very expensive operation.

To find an optimal repartition number below runs have been performed:

Number of Repartition	Execution Time	Tasks
3	19s	26
4	19s	26
6	19s	26
8	20s	26

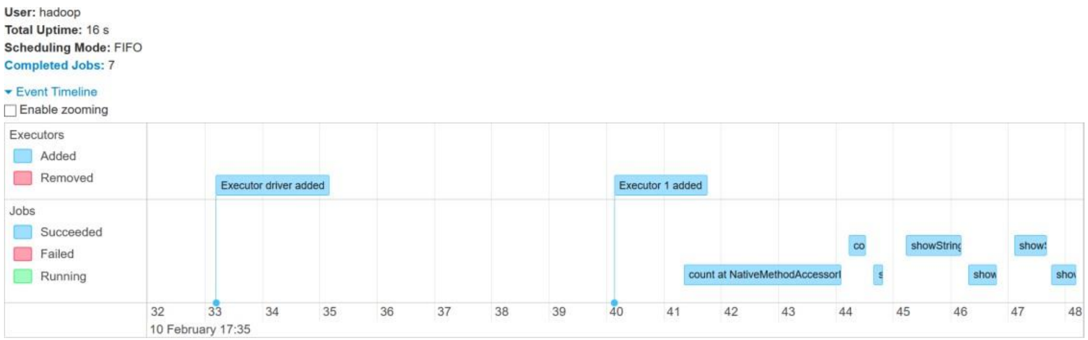
Trial 3

Adding coalesce:

```
df = spark.read.text('s3://kpl3-s3/shakespeare_data.txt').coalesce(4)
```

When you want to reduce the number of partitions prefer using coalesce() as it is an optimized or improved version of repartition() where the movement of the data across the partitions is lower using coalesce which ideally performs better when you dealing with bigger datasets.

(Source: https://sparkbyexamples.com/spark/spark-performance-tuning/)



While execution of this implementation, time required is same as first execution.

Completed Jobs (7)

Page: 1   1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:47	0.4 s	1/1 (1 skipped)	26/26 (1 skipped)
5	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:47	0.6 s	1/1	1/1
4	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:46	0.5 s	1/1 (1 skipped)	26/26 (1 skipped)
3	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:45	1.0 s	1/1	1/1
2	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:44	0.2 s	1/1	1/1
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:44	0.3 s	1/1 (1 skipped)	1/1 (1 skipped)
0	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/02/10 17:35:41	3 s	1/1	1/1

But the number of Task reduced by 1. Was 27 and it is 26 with this trial.

To find an optimal repartition number below runs have been performed:

Number of Repartition	Execution Time	Tasks
2	18s	26
6	16s	26