

Spark's Resilient Distributed Datasets (RDDs)

Introduction:

RDD is a fundamental Data Structure of Apache Spark that provide in-memory data processing large for huge amounts of data in parallel across a cluster of computers.

Properties of RDD in Brief:

1. **In-memory**: Spark RDDs are stored in memory, which allows for the fast processing of large datasets by avoiding the need to access data from a disk. This property makes Spark well-suited for use cases where data needs to be processed in real-time or near real-time.
 2. **Lazy evaluation**: Lazy evaluation is a key feature of Spark RDDs. It means that RDD transformations are not immediately executed when called but are instead queued up as a lineage of transformations until an action is called.
 3. **Immutability**: RDDs are immutable, meaning that once created, their contents cannot be modified. This property ensures that RDDs are consistent and predictable, which is important for fault tolerance and parallelism.
 4. **Persistency**: RDDs can be persisted in memory or on disk for fast access and reuse. This property is useful when multiple operations need to be performed on the same RDD or when the same RDD needs to be used across multiple Spark applications.
 5. **Partitioning**: RDDs can be partitioned into smaller chunks to allow for parallel processing across multiple nodes in a cluster. This property improves the overall performance of Spark by allowing for more efficient use of resources.
 6. **Parallelism**: Spark RDDs can be processed in parallel across multiple nodes in a cluster, which allows for fast and scalable processing of large datasets.
 7. **Fault tolerance**: RDDs are designed to be fault-tolerant, meaning that they can recover from node failures by reconstructing lost partitions from the data stored in other nodes in the cluster. This property is essential for ensuring the reliability of Spark applications.
 8. **Location transparency**: RDDs are location transparent, meaning that Spark can manage the distribution of data across multiple nodes in a cluster without requiring the user to specify the location of each data item. This property makes it easy to develop and deploy Spark applications across a wide range of environments.
- RDD is a low-level API and gives a lot of control over the distribution of data to the user and it can be used for both structured and unstructured data, but the best use case is to implement for unstructured data since Data frames and Datasets in Spark provides more optimization for Structured Data.
 - Though Data frames and Datasets are widely used for Structured Data they compile down anyway to RDDs in the spark execution plan

Traverse through all Spark RDD operations:

Let's explore different types of RDDs, transformations, and actions over RDDs with a practical application

We are going to carry through all the spark operations to build an application that gives some fruitful insights over the following datasets.

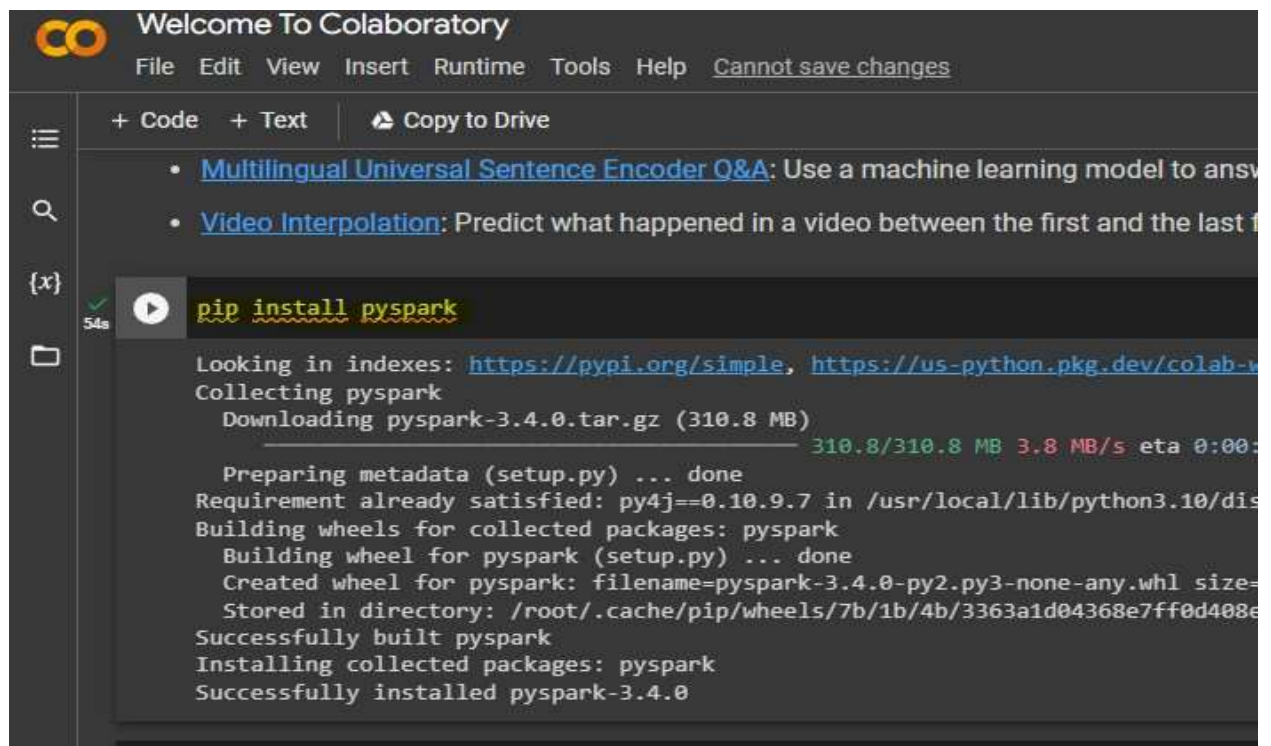
<https://www.kaggle.com/datasets/simhyunsu/imdbextensivedataset>

- 1) IMDB Movies.csv dataset
- 2) IMDB ratings.csv dataset

Setting up the environment

- I am using **PySpark** to explore the RDDs throughout the project.
- I am using Google Colab installed with PySpark for implementing this application

<https://colab.research.google.com/>



The screenshot shows the Google Colaboratory interface. At the top, there's a 'Welcome To Colaboratory' banner with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a status 'Cannot save changes'. Below the menu, there are tabs for '+ Code', '+ Text', and 'Copy to Drive'. The main area displays a list of notebooks, including 'Multilingual Universal Sentence Encoder Q&A' and 'Video Interpolation'. A code cell is selected, showing the command `pip install pyspark` and its output. The output indicates that PySpark 3.4.0 is being downloaded (310.8 MB) and installed successfully. The interface also shows a sidebar with icons for file management and a search bar.

```
Welcome To Colaboratory
File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive

• Multilingual Universal Sentence Encoder Q&A: Use a machine learning model to answer questions
• Video Interpolation: Predict what happened in a video between the first and the last frame

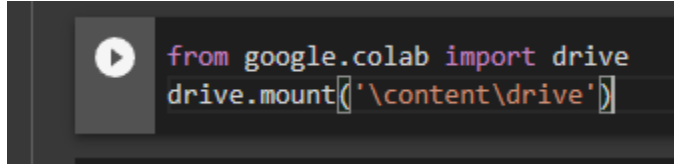
54s pip install pyspark

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels-repository/simple
Collecting pyspark
  Downloading pyspark-3.4.0.tar.gz (310.8 MB)
    310.8/310.8 MB 3.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspark)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.4.0-py2.py3-none-any.whl size=310800000 sha256=7b1b4b3363a1d04368e7ff0d408e
  Stored in directory: /root/.cache/pip/wheels/7b/1b/4b/3363a1d04368e7ff0d408e
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.4.0
```

Load the CSV files into RDDs

To use the external files in Google Colab, we need to download the files and place them in corresponding Google account google drive and mount that drive in the Colab environment

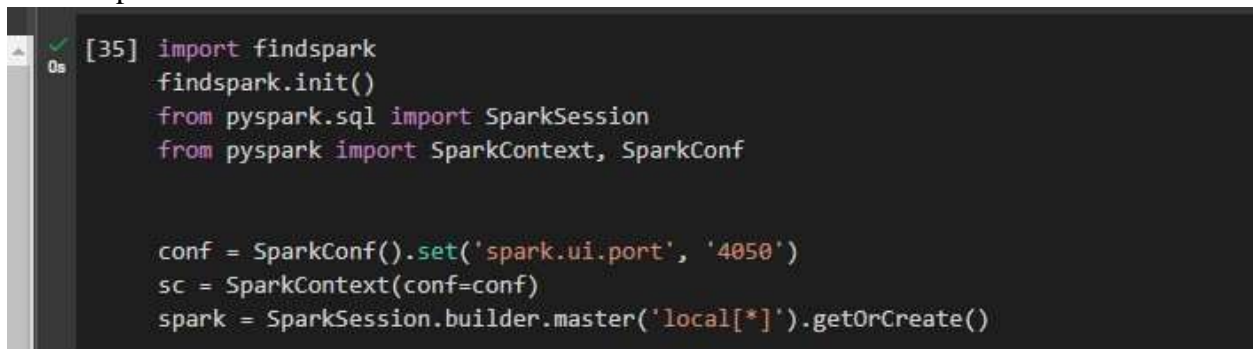
Mounting the drive

A code snippet in a dark-themed editor showing the import of the 'drive' module from 'google.colab' and the execution of 'drive.mount()' to mount the Google Drive to the local file system at the path '\content\drive'.

```
from google.colab import drive
drive.mount('\content\drive')
```

Once the required datasets are available in Files

Create Spark context

A code snippet in a dark-themed editor showing the initialization of a Spark context. It imports 'findspark' and initializes it, then imports 'SparkSession' from 'pyspark.sql' and 'SparkContext', 'SparkConf' from 'pyspark'. It then creates a 'SparkConf' object with 'spark.ui.port' set to '4050', creates a 'SparkContext' with that configuration, and finally creates a 'SparkSession' builder with a local master and gets or creates the session.

```
[35] import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark import SparkContext, SparkConf

conf = SparkConf().set('spark.ui.port', '4050')
sc = SparkContext(conf=conf)
spark = SparkSession.builder.master('local[*]').getOrCreate()
```

- Usually We load the files into RDDs as text files that are separated by Comma since there is no option to load CSV files directly into the RDD since RDDs do have **No Schema**
- There are no records available in RDD and are nothing but Python or Java or Scala objects

As we are using a CSV file where there are certain columns in the file for which the content contains again comma separated values, to avoid misinterpretations while analyzing, we will follow the following steps for creating RDD for the IMDB_Movies dataset

- 1) Create a pyspark data frame from IMDB movies CSV file
- 2) Replace comma contented columns with hyphen-separated values using regex_replace
- 3) Convert the data frame to RDD which will return RDD with Row objects

get the code from the notebook

```
✓ 0s #creating rdds from df
IMDB_movies=IMDB_movies_df_cleansed.rdd

✓ 0s type(IMDB_movies)

pyspark.rdd.RDD
```

Now load the other dataset IMDB_ratings directly using the textFile method of SparkContext

```
IMDB_ratings=sc.textFile('/content/content
type(IMDB_ratings)

pyspark.rdd.RDD
```

- Before performing Queries on the RDDs, cached both the RDDs in memory using the cache() method to perform further operations on top of these RDDs in the most optimized way

```
[154] #caching in memory using cache method for future use
IMDB_movies.cache()

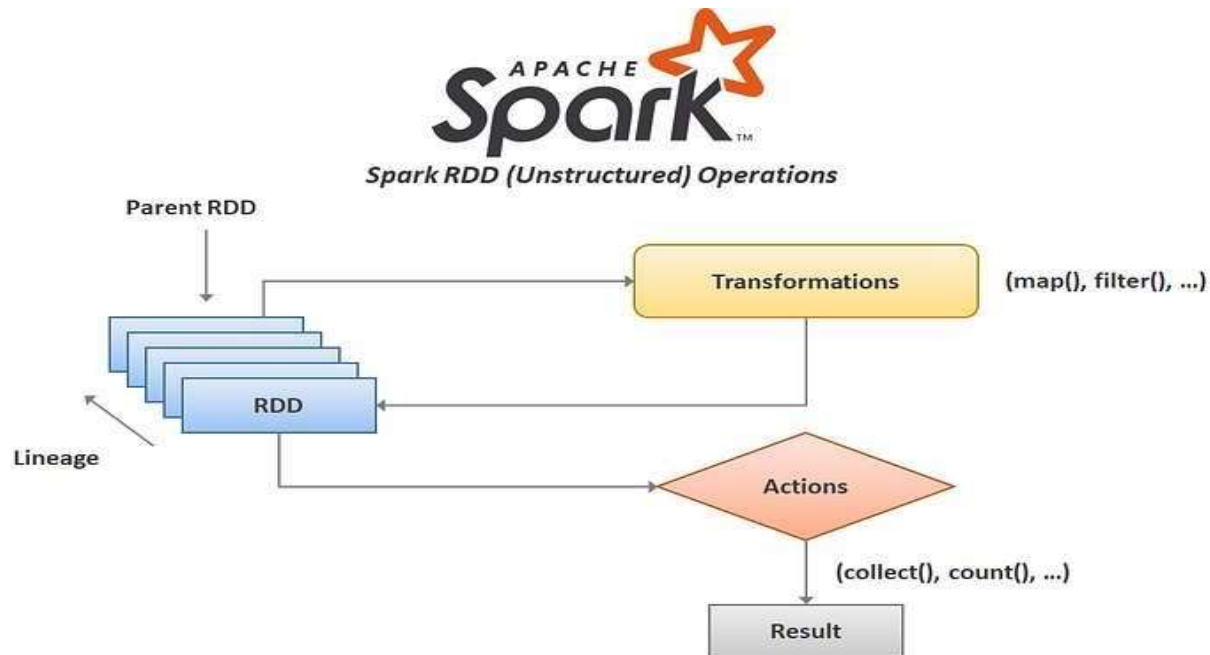
MapPartitionsRDD[54] at javaToPython at NativeMethodAccessorImpl.java:0

✓ 0s [155] #caching in memory using cache method for future use
IMDB_ratings.cache()

/content/contentdrive/MyDrive/data_for_assignment/IMDb ratings.csv MapPartitionsRDD[5] at textFile at NativeMethodAccessorImpl.java:0
```

Before starting, let's understand the Transformations and Actions in Spark

Transformations are manipulations that can be performed on any RDDs that result in and RDD while the Actions are functions that can be applied on RDDs that return the results but not the RDD



- Above image is sourced from the Internet

Let's explore different spark operations with some user queries on top of the above datasets

1) Brief the column information provided in each dataset

For the first RDD, we can get the column names by converting a row into a dictionary and getting the keys

here the **take(x)** will return sample x no of records from RDD

```
▶ for i in IMDB_movies.take(1):  
    for j in i.asDict().keys():  
        print(j)  
    | break
```



```
imdb_title_id  
title  
original_title  
year  
date_published  
genre  
duration  
country  
language  
director  
writer  
production_company  
actors  
description  
avg_vote  
votes  
budget  
usa_gross_income  
worldwide_gross_income  
metascore  
reviews_from_users  
reviews_from_critics
```

for the other data frame, we can get the header using **first()** which returns the first object in RDD

as a string and later split the values using delimiter(',')

```
ratings_header_list=IMDB_ratings.first().split(',')
for i in ratings_header_list:
    print(i)

imdb_title_id
weighted_average_vote
total_votes
mean_vote
median_vote
votes_10
votes_9
votes_8
votes_7
votes_6
votes_5
votes_4
votes_3
votes_2
votes_1
allgenders_0age_avg_vote
allgenders_0age_votes
allgenders_18age_avg_vote
allgenders_18age_votes
allgenders_30age_avg_vote
allgenders_30age_votes
allgenders_45age_avg_vote
allgenders_45age_votes
males_allages_avg_vote
males_allages_votes
males_0age_avg_vote
males_0age_votes
```

2) Give the distinct years from which the movies were listed in IMDB

- Create an RDD with only year values
- Use a distinct method over that RDD to get an rdd that contains only distinct values of years
- Print all of them using the collect() method

```
IMDB_movies.map(lambda rec:rec.year).distinct().collect()

['1894',
 '1906',
 '1911',
 '1912',
 '1919',
```

Result

```
['1894', '1906', '1911', '1912', '1919', '1913', '1914', '1915', '1916', '1917', '1918',
'1920', '1921', '1924', '1922', '1923', '1925', '1926', '1935', '1927', '1928', '1983',
'1929', '1930', '1932', '1931', '1937', '1938', '1933', '1934', '1936', '1940', '1939',
'1942', '1943', '1941', '1948', '1944', '2001', '1946', '1945', '1947', '1973', '1949',
'1950', '1952', '1951', '1962', '1953', '1954', '1955', '1961', '1956', '1958', '1957',
'1959', '1960', '1963', '1965', '1971', '1964', '1966', '1968', '1967', '1969', '1976',
'1970', '1979', '1972', '1981', '1978', '2000', '1989', '1975', '1974', '1986', '1990',
'2018', '1977', '1982', '1980', '1993', '1984', '1985', '1988', '1987', '2005', '1991',
'2002', '1994', '1992', '1995', '2017', '1997', '1996', '2006', '1999', '1998', '2007',
'2008', '2003', '2004', '2010', '2009', '2011', '2013', '2012', '2016', '2015', '2014',
'2019', '2020', 'TV Movie 2019']
```

3) Give the number of movies that were not released in English

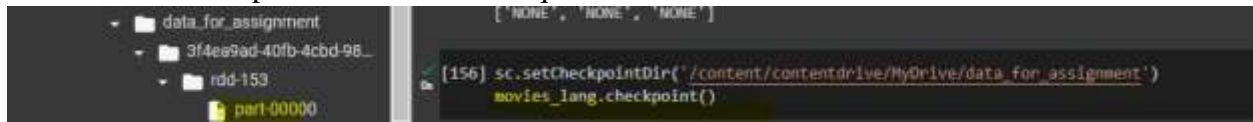
- Create a UDF to check whether 'ENGLISH' present in the language
- Create an RDD only with language column using **map()** method by changing values to upper Case
- Filter out the records to get only films that got released not in English using the **filter()** method
- And get the count of records using the **count()** method

Get the code snippet from notebook here is the result

```
movies_lang.filter(lambda lang: 'ENGLISH' not in lang).count()
```

38409

Let's checkpoint the movies_lang RDD into some external disk space so that it can be useful in the next operations for the next queries



4) Give the unique no of languages in which the films are being released

- * Take the above language RDD that is created in the above question
- * as the language column contains multiple values separated with '-' we need to flatten the RDD using **flatMap()** method and remove extra spaces and provide the distinct count of records



5) Name the top 5 lengthiest and 5 shortest movies along with duration from the list registered in IMDB

- * create 2 rdds which is sorted based on descending order and ascending order using the **sortBy()** method
- * Create an rdd that contains tuples where the first value is the movie name and 2nd value is duration using **map()**
- * Get the top 5 from both the RDDs using **take()**

refer to the code from a notebook, here is the output

```
[139] duration_rdd_desc=IMDB_movies.sortBy(lambda rec: int(rec.duration)*-1)
      duration_rdd_asc=IMDB_movies.sortBy(lambda rec: int(rec.duration))

[140] duration_rdd_desc.map(lambda rec: (rec.title,rec.duration)).take(5)

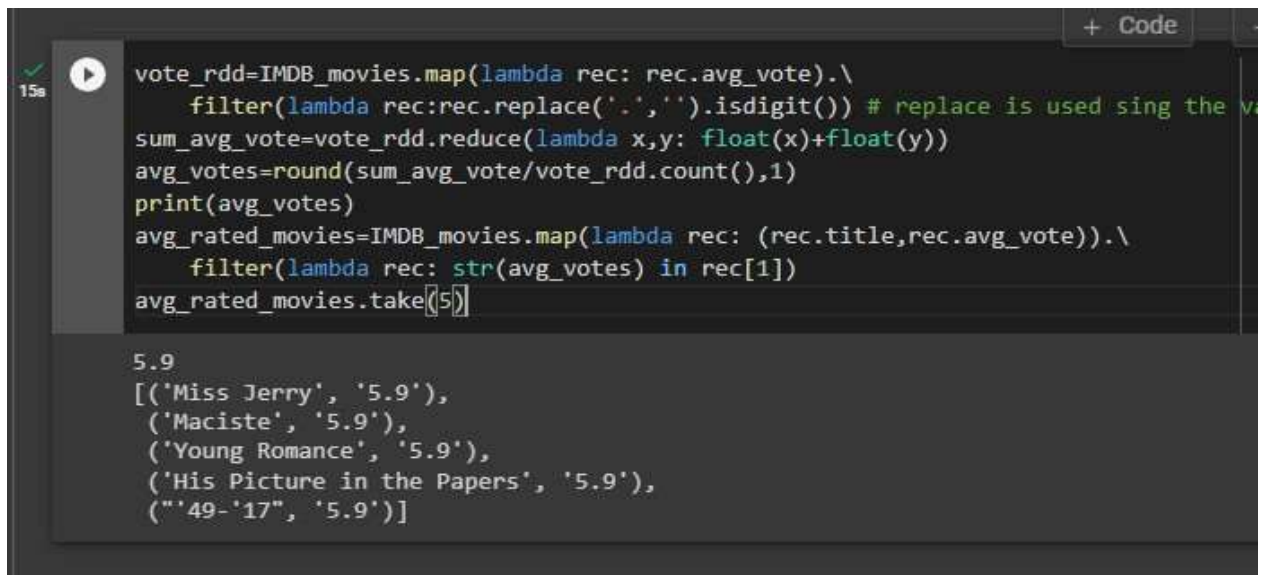
[('La flor', '808'),
 ('Out 1- noli me tangere', '729'),
 ('Khleb - imya sushchestvitelnoe', '580'),
 ('Orbius', '570'),
 ('Ebolusyon ng isang pamilyang Pilipino', '540')]

[140] duration_rdd_asc.map(lambda rec: (rec.title,rec.duration)).take(5)

[('Dragon Ball Z: La vendetta divina', '41'),
 ('Enigma', '42'),
 ('Niagara Falls', '43'),
 ('My Little Pony: Equestria Girls - Holidays Unwrapped', '44'),
 ('Miss Jerry', '45')]
```

6) **Get the top 5 films for which the avg_vote is equal to the average of all available ratings of all movies**

- * Create an RDD that contains only avg_votes and filter out the records which contain values which are not float or integer
- * calculate the sum of all avg_votes using **reduce()** method
- * calculate the average for all avg_votes
- * filter out the records based on new average output and print the top 5 movie names



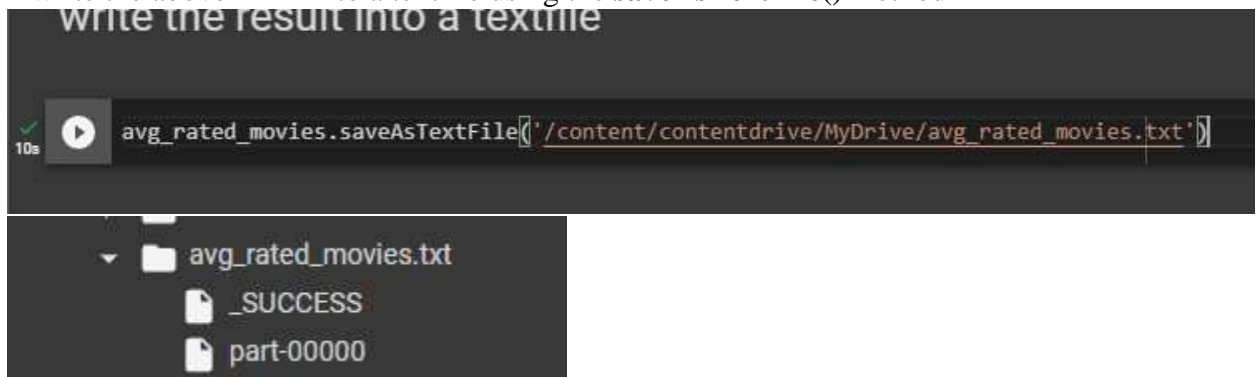
```
vote_rdd=IMDB_movies.map(lambda rec: rec.avg_vote).\
    filter(lambda rec:rec.replace('.','').isdigit()) # replace is used sing the v
sum_avg_vote=vote_rdd.reduce(lambda x,y: float(x)+float(y))
avg_votes=round(sum_avg_vote/vote_rdd.count(),1)
print(avg_votes)
avg_rated_movies=IMDB_movies.map(lambda rec: (rec.title,rec.avg_vote)).\
    filter(lambda rec: str(avg_votes) in rec[1])
avg_rated_movies.take(5)
```

5.9

```
[('Miss Jerry', '5.9'),
 ('Maciste', '5.9'),
 ('Young Romance', '5.9'),
 ('His Picture in the Papers', '5.9'),
 ('"49-'17"', '5.9')]
```

- 7) Need the list of all movies which contains the average vote calculated above and write the result into a textfile

* write the above RDD into a textfile using the **saveAsTextFile()** method



```
avg_rated_movies.saveAsTextFile('/content/contentdrive/MyDrive/avg_rated_movies.txt')
```

avg_rated_movies.txt

- _SUCCESS
- part-00000

```
part-00000 x
2521 ('Roxane', '5.9')
2522 ('Victor et Célia', '5.9')
2523 ('Distinction', '5.9')
2524 ('Arkansas', '5.9')
2525 ('Bernie the Dolphin 2', '5.9')
2526 ('Je voudrais que quelqu'un m'attende', '5.9')
2527 ('Dongesaramdeul', '5.9')
2528 ('Little Joe', '5.9')
2529 ('Impractical Jokers: The Movie', '5.9')
2530 ('Una sirena a Parigi', '5.9')
2531 ('Mosul', '5.9')
2532 ('Lyckligare kan ingen vara', '5.9')
2533 ('Und wer nimmt den Hund?', '5.9')
2534 ('Wildlings', '5.9')
2535 ('Saheb', '5.9')
2536 ('Ranarangam', '5.9')
2537 ('Johannes Pääsukese tõeline elu', '5.9')
2538 ('Aku no hana', '5.9')
2539 ('Mitsubachi to enrai', '5.9')
2540 ('Gloria Mundi', '5.9')
2541 ('Abou Leila', '5.9')
2542 ('Aidiyet', '5.9')
2543 ('A Dog Barking at the Moon', '5.9')
2544 ('D/O Parvathamma', '5.9')
2545 ('The Kissing Booth 2', '5.9')
2546 ('Ninu Veedani Needanu Nene', '5.9')
2547 ('Pan T.', '5.9')
```

8) Take a sample partition of approximately 1000 rows and get the average no of votes cast for that sample

- * get the count of records and partition the RDD based to get each partition containing approx. 1000 recs
- * glom the RDD to get the array of rows for each partition using **glom()** method
- * calculate the sum and average, for the taken partition

```
✓ [186] IMDB_movies.count()
15s
85855

✓ [164] IMDB_movies_part=IMDB_movies.repartition(85)
0s

✓ [185] IMDB_movies_part.getNumPartitions()
0s
85

✓ [185] for i in IMDB_movies_part.glom().take(1):
0s
    cnt=0
    for j in i:
        cnt=cnt+int(j.votes) if j.votes.isdigit() else 0
    print(round(cnt/len(i)))
34
```

9) Calculate all the average of men_avg_vote for every distinct year released movies in the list

- * create a pair RDD from IMDB_movies with values (imdb_title_id,year) using **join()**
- * create a pair RDD from IMDB_ratings with values (imdb_title_id,men_avg_vote)
- * join both the rdds and get a pair rdd with values (year, men_avg_vote)
- * using the **aggregateByKey** method calculate both count and sum and create an intermediate RDD with values (year,(sum_of_votes,count_of_votes))
- * calculate the average and collect the RDD

get the whole code snippet from notebook

here is a sample of the output

```
✓ [185] inter.mapValues(lambda v: round(v[0]/v[1],1)).collect()
1s

[('1911', 6.2),
 ('1913', 6.5),
 ('1918', 6.2),
 ('1920', 6.2),
 ('1921', 6.6),
 ('1925', 6.6),
 ('1927', 6.6),
 ('1983', 6.0),
 ('1928', 6.8),
```

Summary

The document provides a comprehensive exploration of various PySpark RDD operations using a practical example. It covers the key features and properties of Spark RDDs, including in-memory storage, lazy evaluation, immutability, persistency, partitioning, parallelism, fault tolerance, and location transparency. Through the practical example, the document demonstrates the use of various RDD operations, such as counts, averages, sums, and joins, with considerable datasets. The document aims to provide a clear understanding of PySpark RDDs and their potential use cases, enabling developers to leverage the power of Spark for efficient and scalable processing of large datasets.

References

- 1) Spark The Definitive Guide: Big Data Processing Made Simple, by Bill Chambers and Matei Zaharia, O'Reilly Media, 2018. ISBN: 978-1491912218
- 2) Apache Spark RDD Programming Guide: <https://spark.apache.org/docs/latest/rdd-programming-guide.html>
- 3) Introduction to Apache Spark RDDs: <https://databricks.com/glossary/what-is-an-rdd>
- 4) Apache Spark RDDs: <https://www.edureka.co/blog/apache-spark-rdds/>
- 5) Working with Resilient Distributed Datasets (RDDs): <https://docs.databricks.com/spark/latest/rdd/index.html>
- 6) Learning Apache Spark - RDDs: <https://data-flair.training/blogs/apache-spark-rdds/>
- 7) Apache Spark RDD: A Beginner's Guide: <https://www.analyticsvidhya.com/blog/2016/10/using-pyspark-to-perform-transformations-and-actions-on-rdd/>
- 8) Spark RDD vs Dataframe vs Dataset: <https://towardsdatascience.com/spark-rdd-vs-dataframe-vs-dataset-5f7c6c7841b9>
- 9) Apache Spark RDD operations with examples: <https://www.janbasktraining.com/blog/apache-spark-rdd-operations-with-examples/>
- 10) Resilient Distributed Datasets (RDDs): <https://sparkbyexamples.com/spark/spark-resilient-distributed-datasets-rdd/>
- 11) RDDs in Spark: Introduction and Operations: <https://www.baeldung.com/scala/spark-rdds>

Code and Steps for running the code:

We have used Google Colab for running the code.

Attaching the code below:



Spark_RDDs.ipynb

Steps:

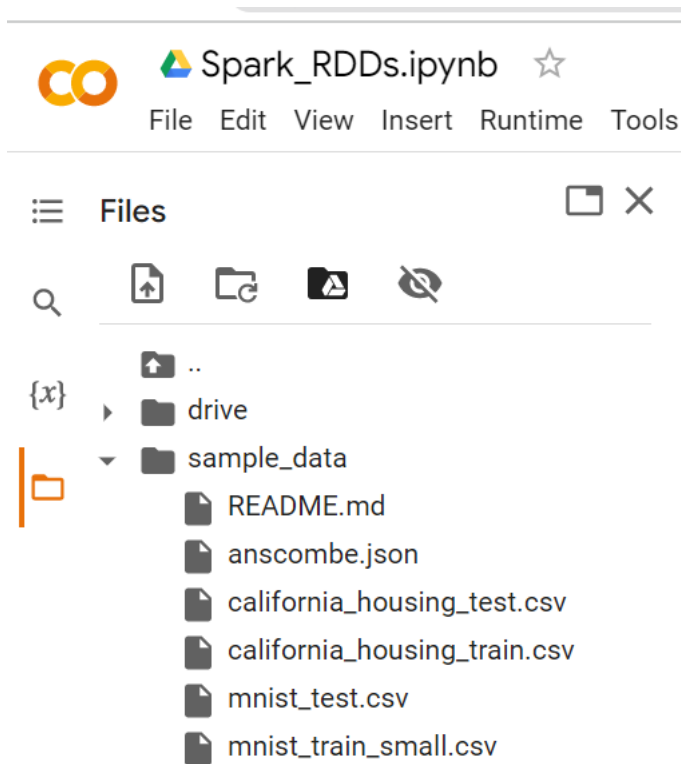
1. Open google colab in chrome → <https://colab.research.google.com/>
2. Place the above attached file in local and upload the file in google colab.
3. After opening the file, please run below command

```
from google.colab import drive
drive.mount('/content/drive')
```

Once it is mounted, it will be displayed as below:



4. Then the google drive will be mounted to the Google Colab.



Drive is displayed on left panel.
Path can be found next to the drive.

5. Place the below CSV's in the personal drive. As the drive is mounted to the colab we are using, the csv's will be displayed in the left side panel. So the location can be changed in the code as per the location in the left panel.



archive.zip

6. Once the location is changed in the code, run the cells from the start.
7. Output can be seen after each cell is being run.