## - Part 1: Introducing Use Case

We will exploitan Open Source Dataset which contains 100K notes introduced by 1000 Users on 1700 Films in order to program a system that will introduce us to films similar to a given film, The principle that we will follow is: “If the majority of users who have seen the two films have given two similar ratings to the two films: We judge that these two films are similar”

For example: If the user “Metidji Sid Ahmed” rated the series “Breaking Bad” with 5/5 and the “Better Call Saul” series 4.5/5: we can judge that these two series are similar (that is to say if a user appreciates one of these two: He will most likely appreciate the other )

## - Part 2: Presentation of process followed - Algorithm -:

Like any problem in Big Data, the resolution of this process will follow a MapReduce architecture with several Pipelines:

and Here are the steps to follow to have the films similar to a predefined film: “film1”

- We find the different pairs of films ( film1 , filmX) that have been watched by the same user , we name the pair of ratings given to these two films by this user by ( rating1_0 , ratingX_0)

- For each pair of films (film1, filmX) found, we find all the other N users who also watched this pair, so we will have N pairs of ratings (rating1_1, ratingX_1), (rating1_2, rating X_2), ....., (rating1_N, ratingX_N)
- We define a similarity function which will measure the rate of similarity between the ratings given to Film1 with the ratings given to Film 2: i.e. we calculate the similarity between rating1_0 , rating1_1 , rating1_2 , ... , rating1_N with ratingX_0 , ratingX_1 , ratingX_2 , ..... , ratingX_N . This function will give us as an output ( (film1 , filmX) , Similarity )
- We take out the results obtained by the Similarity score and we display the 10 films with the highest similarity score with Film1

## - Part 3: Presentation of the process followed - Programming -:

The algorithm presented above is an algorithm that works perfectly but before to implement it, we must reformulate it following the Spark paradigm

1.We Initialize the Pyspark environment

```
conf = SparkConf().setMaster("local[*]").setAppName("MovieSimilarities")
sc = SparkContext(conf = conf)
```

● we put local[*] in order to use all the processor cores to reduce the execution time

2. We read the table that contains the names of the movies (in order to make the correspondence between MovieId and Movie Name) with a simple Python script, Then we start processing by reading the u.data file which contains our dataset

```
print("\nLoading movie names...")
nameDict = loadMovieNames()

data = sc.textFile("file:///SparkCourse/ml-100k/u.data")
```

3. we map each user rating to an entity ( Key , Value ) with Key = userId and Value = ( movieId , rating )

```
ratings = data.map(lambda l: l.split()).map(lambda l: (int(l[0]), (int(l[1]), float(l[2]))))
```

4. thanks to the Self-Join: We will obtain all the possible combinations (two by two) of films rated by the same user

```
joinedRatings = ratings.join(ratings)
```

- <u>output:</u> userId => ( (movie1 , rating1 ) , (movie2 , rating2) )

5. Since the order of elements of “Value” is not important, we will have duplications on the result obtained from Self-Join ( we will have an entity ( userId , ( (movie1 , rating1 ) , (movie2 , rating2) ) as we will have ( userId , ( (movie2 , rating2 ) , (movie1 , rating1) ) ) . So we have to call a filter function to get rid of this duplication

```
uniqueJoinedRatings = joinedRatings.filter(filterDuplicates)
```

6. We call the 'Map' function to the current result in order to make the Key of our data the couple ( movie1 , movie2 ) instead of UserId , and the Value is ( rating1 , rating2 )

```
moviePairs = uniqueJoinedRatings.map(makePairs)
```

- <u>output:</u> (movie1, movie2) => (rating1, rating2)

7. we call the GroupByKey function in order to group the data that represent the ratings given to the same couple of films

```
moviePairRatings = moviePairs.groupByKey()
```

- output: (movie1, movie2) => ( (rating1, rating2), (rating1, rating2) , .... )

8. Now, for each line we will call our function which calculates the similarity between the films (film1, film2) based on ( ( rating1 , rating2 ), (rating1 , rating2 ) , .... ) , and we store it in the cache in order to be exploited by the different nodes without having to redo the same calculation

```
moviePairSimilarities = moviePairRatings.mapValues(computeCosineSimilarity).cache()
```

output: (movie1, movie2) => similarity

9. Optional step: we can store the output obtained from the 8th result in a file in order to avoid redoing the calculations made in all these stages

10. The main program: The user will introduce the ID of the movie that liked it: movieID in order to recommend it to similar movies (we give it the lines Key,Value where (movie1 = MovieID OR movie2 = movieID) and we output the result obtained according to the similarity score

```python
# Extract similarities for the movie we care about that are "good".

scoreThreshold = 0.97
coOccurenceThreshold = 50

movieID = int(input("Enter movie ID: "))

# Filter for movies with this sim that are "good" as defined by
# our quality thresholds above
filteredResults = moviePairSimilarities.filter(lambda pairSim: \
    (pairSim[0][0] == movieID or pairSim[0][1] == movieID) \
    and pairSim[1][0] > scoreThreshold and pairSim[1][1] > coOccurenceThreshold)

# Sort by quality score.
results = filteredResults.map(lambda pairSim: (pairSim[1], pairSim[0])).sortByKey(ascending = False).take(10)

print("Top 10 similar movies for " + nameDict[movieID])
for result in results:
    (sim, pair) = result
    # Display the similarity result that isn't the movie we're looking at
    similarMovieID = pair[0]
    if (similarMovieID == movieID):
        similarMovieID = pair[1]
    print(nameDict[similarMovieID] + "\tscore: " + str(sim[0]) + "\tstrength: " + str(sim[1]))
print("--- %s seconds ---" % (time.time() - start_time))
```

## - Part 4: A running example

Our user wants to know movies similar to the movie “Star Wars”

```
Loading movie names...
Enter movie ID: 50
22/06/30 18:33:45 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped

[Stage 1:=============================>                            (2 + 2) / 4]

Top 10 similar movies for Star Wars (1977)
Empire Strikes Back, The (1980) score: 0.9895522078385338   strength: 345
Return of the Jedi (1983)   score: 0.9857230861253026   strength: 480
Raiders of the Lost Ark (1981)  score: 0.981760098872619    strength: 380
20,000 Leagues Under the Sea (1954) score: 0.9789385605497993   strength: 68
12 Angry Men (1957) score: 0.9776576120448436   strength: 109
Close Shave, A (1995)   score: 0.9775948291054827   strength: 92
African Queen, The (1951)   score: 0.9764692222674887   strength: 138
Sting, The (1973)   score: 0.9751512937740359   strength: 204
Wrong Trousers, The (1993)  score: 0.9748681355460885   strength: 103
Wallace & Gromit: The Best of Aardman Animation (1996)  score: 0.9741816128302572   strength: 58
--- 88.03608059883118 seconds ---

Process finished with exit code 0
```

- Part 5: Repeat the same execution example on a Spark Cluster deployed on GCP (Google Cloud Platform):