

Big data Scalable Lyrics Analyzer

Student: Liran Nachman

ID: 311342836

Student: Eden Dupont

ID: 204808596

Student: Daniel Rolnik

ID: 334018009

Student: Eden Sharoni

ID: 315371906

Lecturer: Dr. Morose Boris

Date: 26/04/2020

[Link to video with explanations about project](#)

1. The Problem:

Nowadays, most song recommendation systems rely heavily on tagged songs and genres, and they do not account for their contents. There are 150-300 words per song on average according to the research (1). Every song recommendation system may have millions of songs (for example Spotify has more than 50 million songs (2)). Analyzing lyrics for such a big amount of songs may be a time-consuming task.

Our algorithm will analyze songs based on their lyrics, which may be able to serve song databases for user recommendation.

Scale-out and scale-up configurations are two different representative methods to implement big data analytics infrastructures. In scale-out server clusters (e.g, Spark, Hadoop), server upgrades are performed through adding nodes to the existing cluster system. On the other hand, in a scale-up environment, server upgrades are performed through adding resources (e.g, CPU, memory) to the existing single node-based system. Scale-up servers are mostly used in scientific analytics areas, and big data analytics frameworks are being increasingly used. However, Spark has been reported that it does not scale on the single node scale-up server because of garbage collection(GC) overheads (3).

2. Existing solution (3):

Configuration used in the research to check existing solutions:

Workload	Input data size	Heap size	Configuration	Data type
Word Count	10G	4G	none	text
Naive Basian	10G	4G	none	text
Grep	30G	4G	"the"	text
K-means	4G	4G	k=8	graph
JVM	Spark	Hadoop	OS	Distribution
Openjdk 1.8.0_91	1.3.1	1.2.1	Linux 4.5-rc6	Ubuntu

Research results for existing solutions:

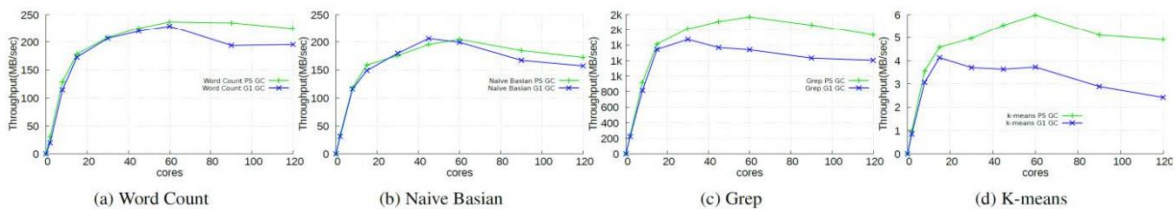


Figure 1 Performance scalability

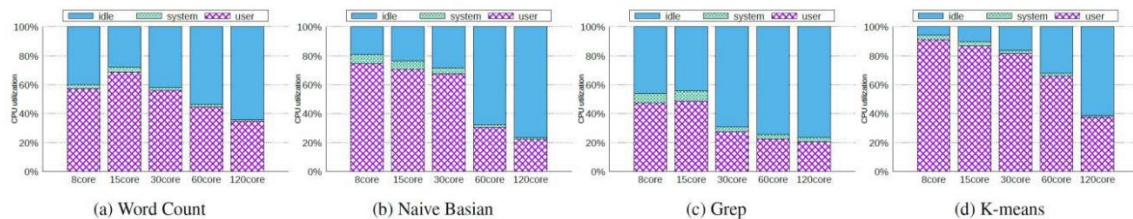
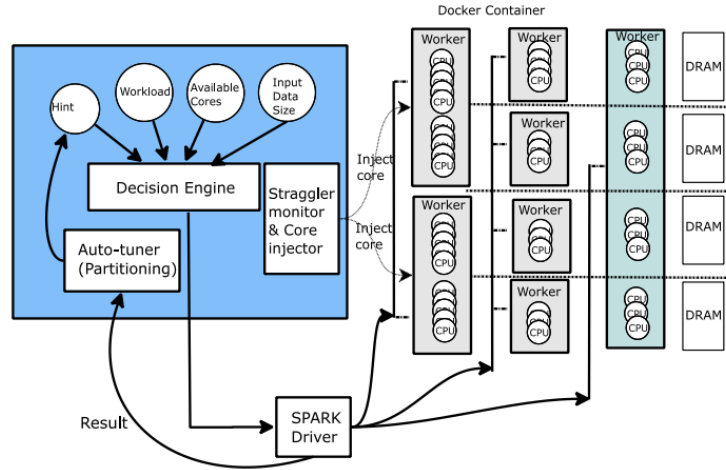


Figure 2 CPU utilization.

Research shows that using existing solutions after some amount of cores workload flattens because of garbage collector overhead and remote memory access.

That's why *alternative architecture* was suggested in the research:



Configuration used to check solution proposed by the research:

method	executor heap size	number of partitions
non-partition	4G	1
coarse-grained(30 core)	1G	4
fine-grained(15 core)	512M	8

Partitioning values used in to check suggested architecture

Results of running code on the architecture proposed by research:

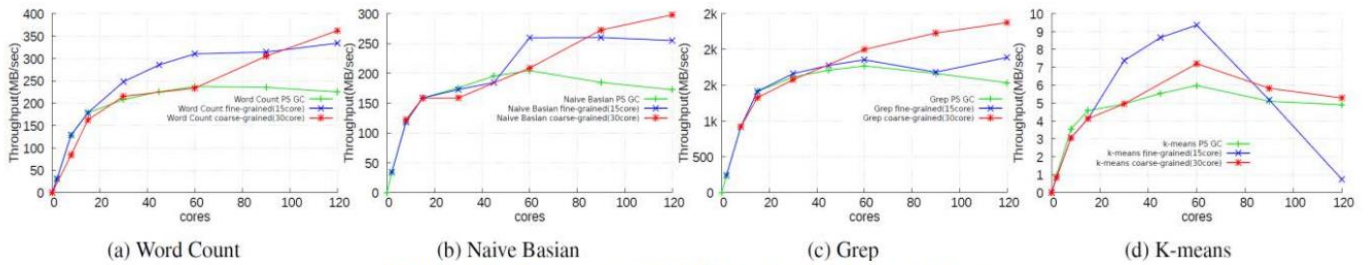


Figure 4 Performance scalability using Docker container

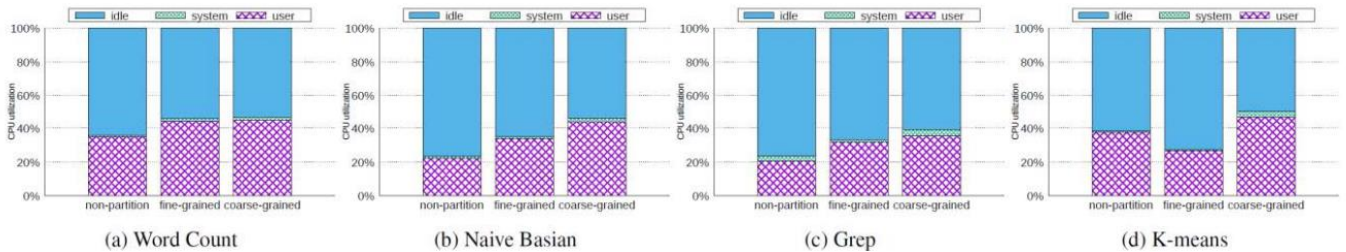


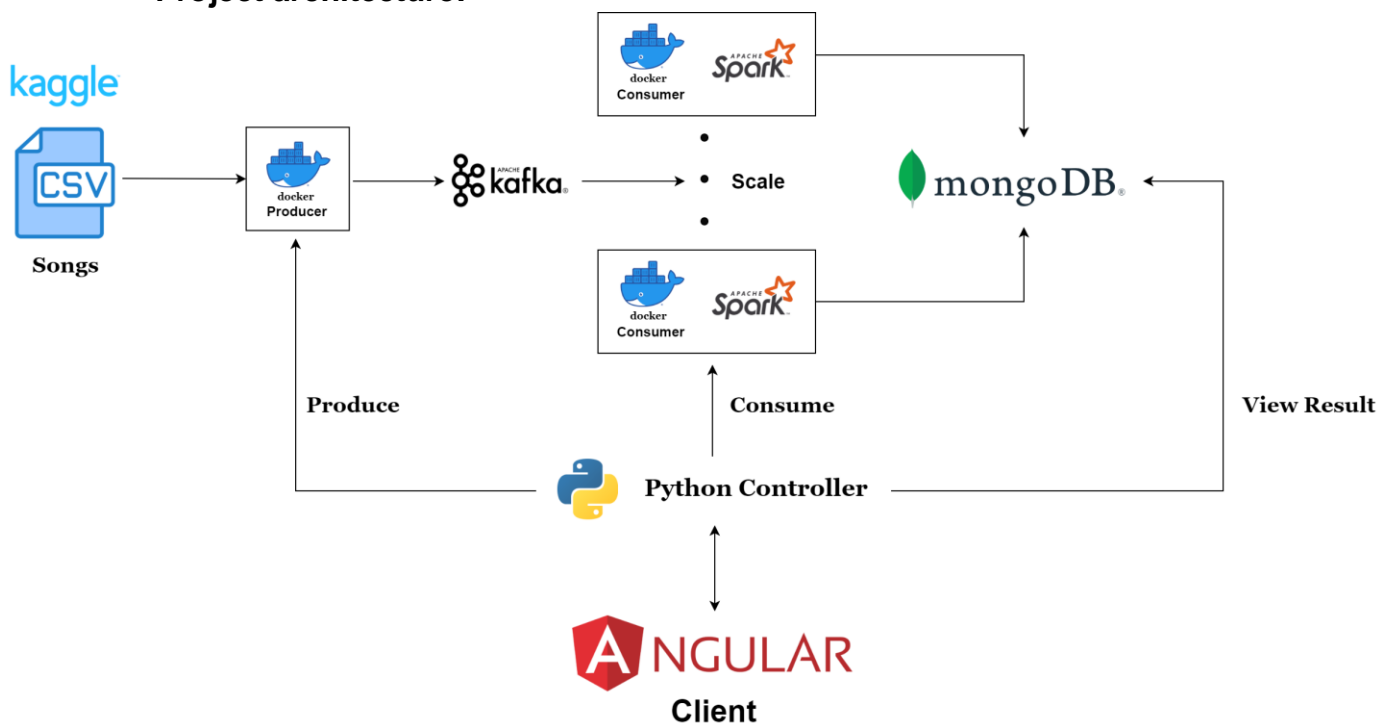
Figure 5 CPU utilization on 120 core

The results show that partitioning approach outperforms non partitioning approach by 1.4x on 120 cores. Therefore, in many core scale-up server, partitioning approach has many advantages over non-partitioning approach in terms of performance scalability. Proposed by research Docker-container method has substantial performance up to 1.7 times compared to existing solutions.

3. Our Solution:

We used data set with 57,650 songs (4) as input for our project.

Project architecture:



Basic flow explanation:

- 1) Using an Angular based web client, the user calls a python based API that she wants to start a Kafka producer (Docker image of producer) to produce data taken from CSV.
- 2) The user can start call via the API for the consumers to work (get batches from Kafka) with different parameters
- 3) Each consumer will use a spark cluster to work on the batch received to analyze the lyrics and send the analysis to a MongoDB database.
- 4) The user may view the analysis in the client by using an API which requests for the data.

Advanced server flow explanation:

In the course we've learned basic algorithms to use Spark with Python. Spark has 2 modes. Spark local and cluster mode.

In the course we studied how to use Spark in local mode, in this mode, the spark workers already have access to all the files we have in the project.

In this project we worked with Docker containers, so we needed to work with Spark in a cluster mode - for that to work, each worker needs the files.

The docker image wraps the project into a zip file. Which is then extracted by each of the workers, and now they have access to all the files and functions in order to work.

Producer reads CSV and sends job batches to Kafka. On the other side we have Kafka consumers. Each consumer is a thread.

Once Kafka consumer-worker receives a work from the Kafka server, it begins the analysis of the songs within the batch, each song is analyzed using spark, so he can analyze work's input data and return the result (word count per song and emotions).

The Spark: 1 thread is 1 job. 2 threads are 2 jobs that runs in parallel etc. To control the scheduling of the threads, we have a file called “fairscheduler.xml” which is activated when getting spark context. If we look up at the fairscheduler.xml we can see that the pool is defined like a ‘fair_pool’, it is like pool which is full of threads. ‘fair_pool’ works the same way as Round Robin, he divides equal resources for every job and equally divides all existing jobs in parallel, the file defines how the thread will work relative to others in the pool (5)

4. What our solution improve:

Our solution allows easy scale up scalability for word counting and lyric’s emotion recognition. That achieved using:

- a) Kafka
- b) Spark
- c) Threads
- d) Parallel
- e) Cloud
- f) Docker

5. How to run our solution:

- a) Make sure Python 3.7 is installed
- b) Run docker
- c) Enter root directory of the project
- d) Run “start.bat” file

6. What we learned:

It is a complex project consisting of many parts. Some team members were familiar with some technologies, but they never used all of them together. The Biggest challenge was to learn how to connect dockers with spark and how to scale spark’s RDDs using docker containers. Most of the members wasn’t familiar with Angular and typescript, so we needed to study how to use it and how to integrate it to our project.

7. GUI Client Screenshots:

[Song Viewer](#)

[Home](#)
[Settings](#)
[Spark UI](#)

Total records in database: 462750

Delete database

Pick the first letter of the artist to load:

ALL ARTISTS

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

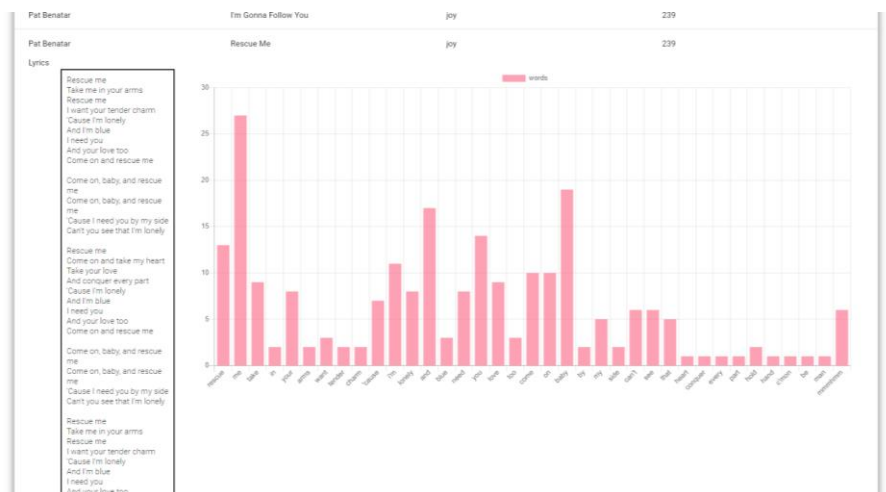
Filter

Artist	Song name	Emotion	Word Count
ABBA	Ahe's My Kind Of Girl	joy	153
ABBA	Andante, Andante	joy	260
ABBA	As Good As New	anticipation	312
ABBA	Bang	joy	200
ABBA	Bang A-Boomerang	joy	198

Items per page: 5

1 - 5 of 106102

Home tab



Output Statistics for individual song

Song Viewer

Home

Settings

Spack UI

Spack worker config

Number of workers *

1

Cores per worker *

2

Memory per worker *


1g

Power setting

☒ Consumer Off/On

Start Producer

Settings Tab



2.6.4

Spark Master at spark://oddb12338ff0:7077

URL: spark://oddb12338ff0:7077

Alive Workers: 1

Cores in use: 0 / Total: 0 Used

Memory in use: 19.0 GB / Total: 0.0 GB Used

Applications: 0 / Running: 7 / Completed: 0

Drivers: 0 / Running: 0 / Completed: 0

Status: ALIVE

Workers (1)

Worker ID	Address	State	Cores	Memory
worker-20200418111813-172.18.0.5-38835	172.18.0.5-38835	ALIVE	6 (0 Used)	19.0 GB (0.0 GB Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	----------------	------	-------	----------

Completed Applications (7)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-202004181147-0002	pyzspark-shell	1	3.0 GB	20200418 11:41:47	root	FINISHED	10 s
app-202004181147-0003	pyzspark-shell	1	3.0 GB	20200418 11:41:47	root	FINISHED	10 s
app-202004181147-0004	pyzspark-shell	1	3.0 GB	20200418 11:41:47	root	FINISHED	9 s
app-202004181147-0005	pyzspark-shell	1	3.0 GB	20200418 11:41:47	root	FINISHED	9 s
app-202004181148-0006	pyzspark-shell	1	3.0 GB	20200418 11:41:48	root	FINISHED	9 s
app-202004181148-0007	pyzspark-shell	1	3.0 GB	20200418 11:41:47	root	FINISHED	10 s
app-202004181149-0008	pyzspark-shell	2	1024 MB	20200418 11:41:49	root	FINISHED	5 s

Spark Tab

Bibliography

1. **Fragapane, Varun Jewalikar and Federica.** musixmatch. *musixmatch*. [Online] Dec 3, 2015. http://lab.musixmatch.com/vocabulary_genres/.
2. **spotify.** Company Info. *spotify*. [Online] Dec 31, 2019. <https://newsroom.spotify.com/company-info/>.
3. **Kyong, Joohyun, Jeon, Jinwoo and Lim, Sung-Soo.** Improving scalability of apache spark-based scale-up server through docker container-based partitioning. *ResearchGate*. [Online] Feb 2017. https://www.researchgate.net/publication/316611536_Improving_scalability_of_apache_spark-based_scale-up_server_through_docker_container-based_partitioning.
4. **Kuznetsov, Sergey.** Song Lyrics. *Kaggle*. [Online] Jan 5, 2017. <https://www.kaggle.com/mousehead/songlyrics>.
5. **Spark.** Job Scheduling. *Spark*. [Online] <https://spark.apache.org/docs/latest/job-scheduling.html#configuring-pool-properties>.