Lab4 Spark Streaming

In this lab we are going to connect a kafka system with spark streaming and perform some basic operations on the stream of data

1)Create Kafka cluster

Following what you learned in previous labs, create a kafka cluster on dataproc using the initialization actions for Kafka. A difference between this cluster and the previous ones is that **you need to use 3 Master nodes**. Use the smallest machine possible for master and worker nodes. Use two worker nodes.

Then you need to log onto the master to generate a topic in Kafka, for example:

kafka-topics.sh --zookeeper localhost:2181 --create --replication-factor 1 --partitions 1 --topic test

And do this to see whether it has been created

kafka-topics.sh --zookeeper locahost:2181 --list

In order to write into the topic you can write something like this:

```
for i in {0..100000}; do echo "mikele2message${i}"; sleep 0.2; done |
/usr/lib/kafka/bin/kafka-console-producer.sh --broker-list
mikele-dev-kafka-w-0:9092 --topic test
```

Or you can write a client on any language to write into Kafka using Kafka libreries. In my case my cluster is called *mikele-dev-kafka* and my topic is called *test*.

Now we are going to see how kafka and spark streaming work together. In this case we will use structured streaming

2)Connecting Kafka and Spark

Now we are going to read the stream from kafka and write the stream coming raw into cloud storage as a parquet file. The file will grow as more data comes into the stream.

This is the code in Python that we need to execute to run the streaming operation:

```
-----START CODING SECTION-------------
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("Structuredkafka") \
    .getOrCreate()

# Create DataFrame from kafka


lines = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "mikele-dev-kafka-w-0:9092") \
  .option("subscribe", "test") \
  .load()
lines = lines.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")


# Start running the query
query = lines \
    .writeStream \
    .outputMode("append").trigger(processingTime='60 seconds') \
    .format("parquet").option("checkpointLocation",
"gs://mikele-sparkstreaming/checkpointing3").option("path",
"gs://mikele-sparkstreaming/streaming/str_pg3").start()

query.awaitTermination()

----------END CODING SECTION------------
```

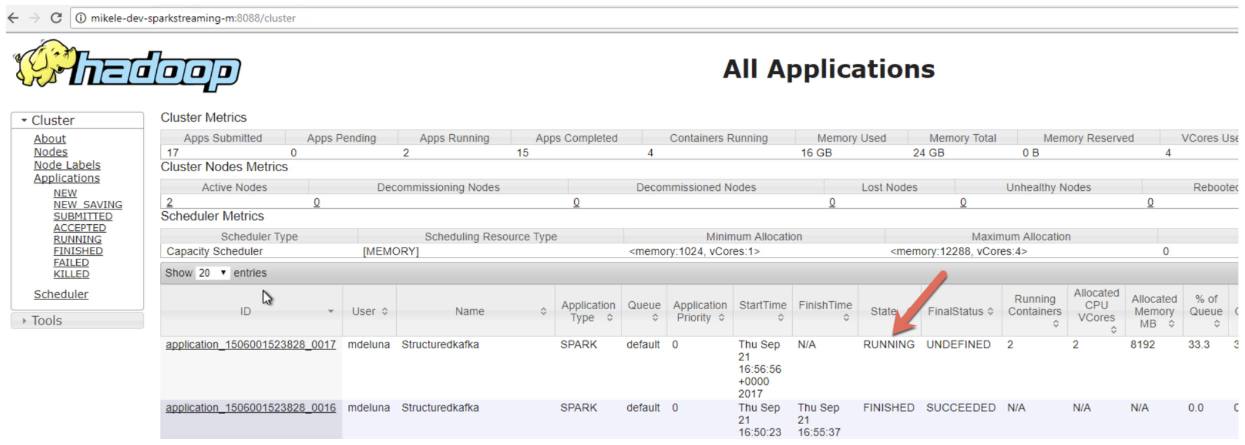
What we are doing here is:

1)Use readStream to read a topic (you could read several or even a pattern) from one or several kafka servers. In my case, I am using my topic named “test”.
2)I create a **query** and I give the command of writing on append mode every 60 seconds. With this we append the information into the parquet file. It is important to have the checkpoint pointing to a folder within a gs bucket, as you can see in the code. The final file is created and appended where the option “Path” is. In my case, in my bucket in the folder streaming.

In order to execute this code, you can do like in previous labs, in this case I execute this from the command line passing Kafka libraries as a parameter. Kafka libraries are already deployed in the cluster.

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10 2.11:2.2.0
./structuredkafka.py
```

Connect to your sparkUI as you did in previous labs and you will see the job you just submitted:



mikele-dev-sparkstreaming-m:8088/cluster

All Applications

Cluster: About, Nodes, Node Labels, Applications (NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED), Scheduler

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Use
17	0	2	15	4	16 GB	24 GB	0 B	4

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebootec
2	0	0	0	0	0

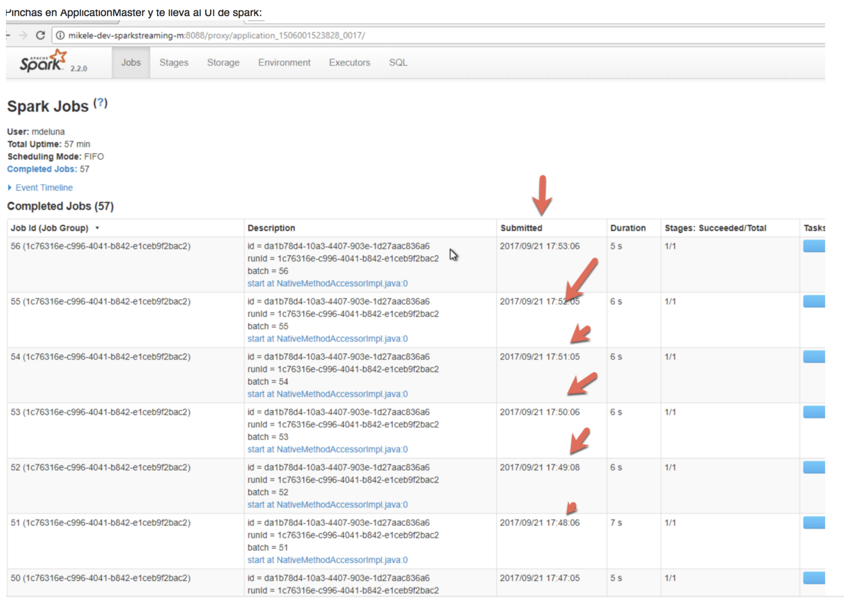
Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:12288, vCores:4>	0

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	% of Queue
application_1506001523828_0017	mdeluna	Structuredkafka	SPARK	default	0	Thu Sep 21 16:56:56 +0000 2017	N/A	RUNNING	UNDEFINED	2	2	8192	33.3
application_1506001523828_0016	mdeluna	Structuredkafka	SPARK	default	0	Thu Sep 21 16:50:23	Thu Sep 21 16:55:37	FINISHED	SUCCEEDED	N/A	N/A	N/A	0.0

Click on the Application Master and this would lead you to the spark streaming. As you can see every 60s something happens, in our case save the information into disk



Pinchas en ApplicationMaster y te lleva al UI de spark:

mikele-dev-sparkstreaming-m:8088/proxy/application_1506001523828_0017/

Spark 2.2.0 — Jobs, Stages, Storage, Environment, Executors, SQL

Spark Jobs (?)

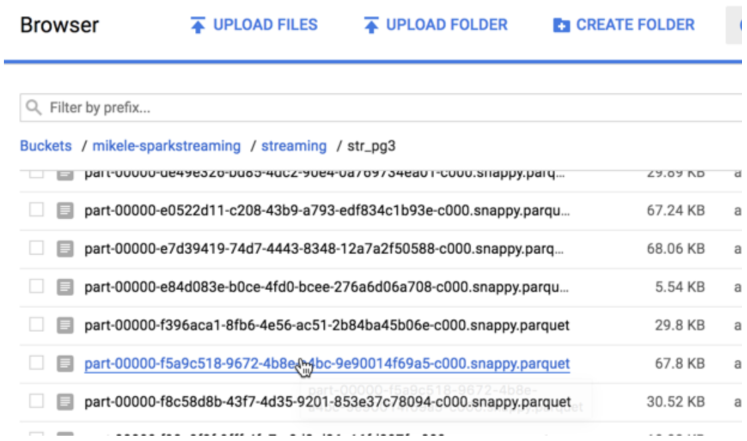
User: mdeluna
Total Uptime: 57 min
Scheduling Mode: FIFO
Completed Jobs: 57

Event Timeline

Completed Jobs (57)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks
56 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 56 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:53:06	5 s	1/1	
55 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 55 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:52:05	6 s	1/1	
54 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 54 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:51:05	6 s	1/1	
53 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 53 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:50:06	6 s	1/1	
52 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 52 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:49:08	6 s	1/1	
51 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2 batch = 51 start at NativeMethodAccessorImpl.java:0	2017/09/21 17:48:06	7 s	1/1	
50 (1c76316e-c996-4041-b842-e1ceb9f2bac2)	id = da1b78d4-10a3-4407-903e-1d27aac836a6 runId = 1c76316e-c996-4041-b842-e1ceb9f2bac2	2017/09/21 17:47:05	5 s	1/1	

In order to see what is happening in the parquet file, go to your bucket and you will see how the different parts of the parquet file are created with a default codec:



Buckets / mikele-sparkstreaming / streaming / str_pg3

part-00000-de49e326-bd85-4dc2-90e4-0a769734ea01-c000.snappy.parq...	29.89 KB	a
part-00000-e0522d11-c208-43b9-a793-edf834c1b93e-c000.snappy.parqu...	67.24 KB	a
part-00000-e7d39419-74d7-4443-8348-12a7a2f50588-c000.snappy.parq...	68.06 KB	a
part-00000-e84d083e-b0ce-4fd0-bcee-276a6d06a708-c000.snappy.parqu...	5.54 KB	a
part-00000-f396aca1-8fb6-4e56-ac51-2b84ba45b06e-c000.snappy.parquet	29.8 KB	a
part-00000-f5a9c518-9672-4b8e-a4bc-9e90014f69a5-c000.snappy.parquet	67.8 KB	a
part-00000-f8c58d8b-43f7-4d35-9201-853e37c78094-c000.snappy.parquet	30.52 KB	a

If you are interested in knowing how big is the parquet file you can run this kind of command in cloud shell, for my bucket and the my parquet file is:

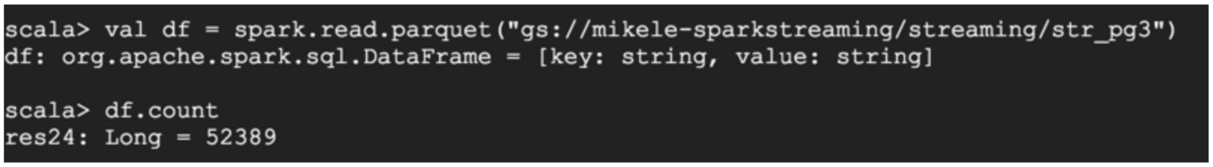
```
gsutil ls -la gs://mikele-sparkstreaming/streaming/str_pg3
```

And it should show you something like this:

```
TOTAL: 61 objects, 2658107 bytes (2.53 MiB)
```

If you execute that several times you will see that the “directory” making the parquet file is growing.

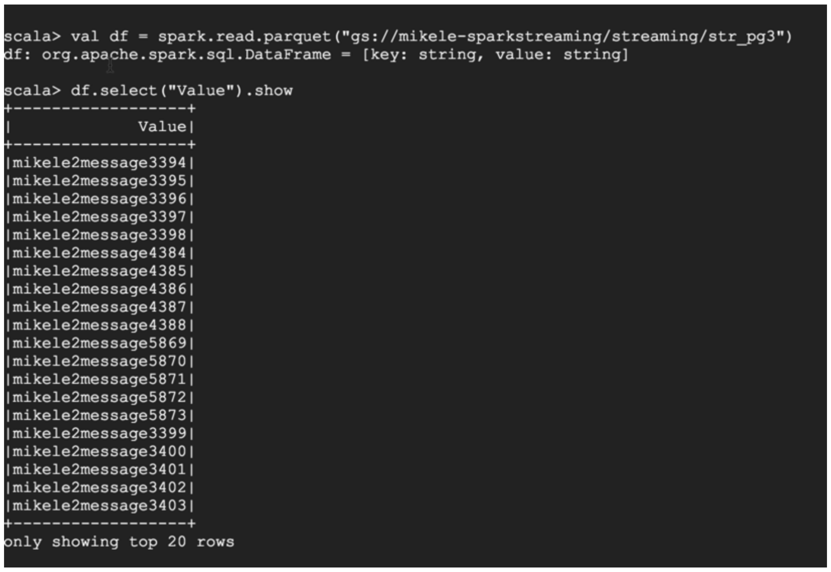
If you want to play with the contents of that parquet file you can create a spark-shell or pySpark and read directly from there by creating a dataframe:



```
scala> val df = spark.read.parquet("gs://mikele-sparkstreaming/streaming/str_pg3")
df: org.apache.spark.sql.DataFrame = [key: string, value: string]

scala> df.count
res24: Long = 52389
```

And then you can see the contents of the file by executing a dataframe operation:



```
scala> val df = spark.read.parquet("gs://mikele-sparkstreaming/streaming/str_pg3")
df: org.apache.spark.sql.DataFrame = [key: string, value: string]

scala> df.select("Value").show
+------------------+
|             Value|
+------------------+
|mikele2message3394|
|mikele2message3395|
|mikele2message3396|
|mikele2message3397|
|mikele2message3398|
|mikele2message4384|
|mikele2message4385|
|mikele2message4386|
|mikele2message4387|
|mikele2message4388|
|mikele2message5869|
|mikele2message5870|
|mikele2message5871|
|mikele2message5872|
|mikele2message5873|
|mikele2message3399|
|mikele2message3400|
|mikele2message3401|
|mikele2message3402|
|mikele2message3403|
+------------------+
only showing top 20 rows
```

Remember **that RDDs and DFs are immutable**, that means that you would have to create the dataframe, in this case df, every time you want to see the current content.

3)What to do next?

Create a different stream, this time with numbers and try to calculate average of that stream every 10 seconds. Apply different operations on the data and experiment different times of saving modes.