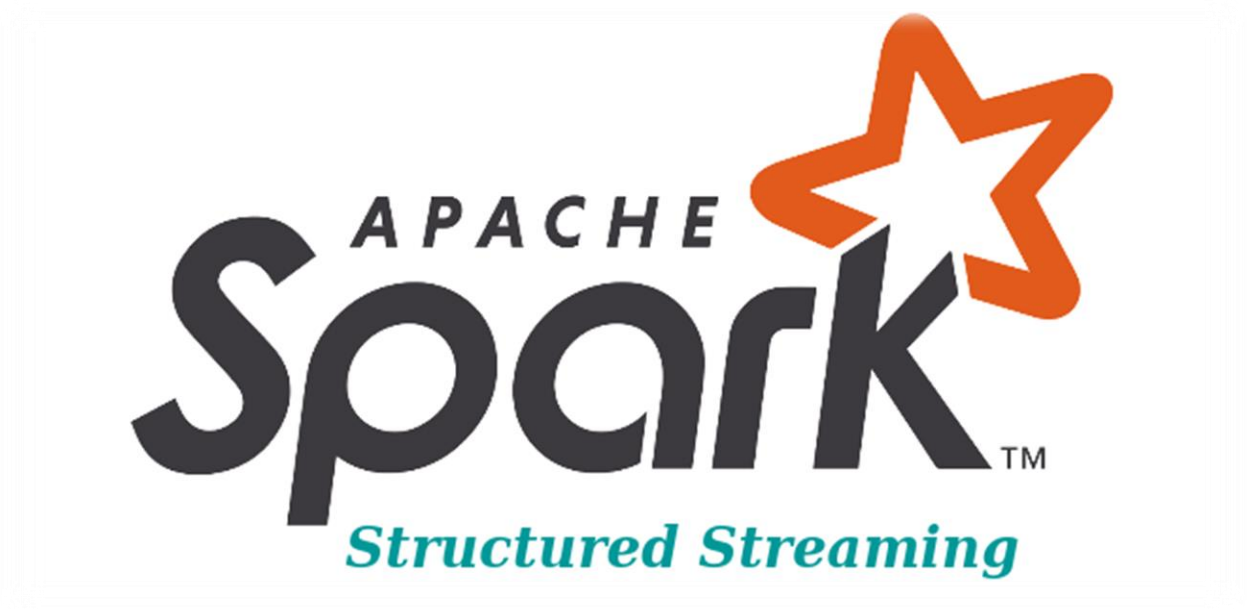


Two examples Hands-on Structure streaming

By Abdelrahman Omar

Contents

Introduction to Spark structure streaming

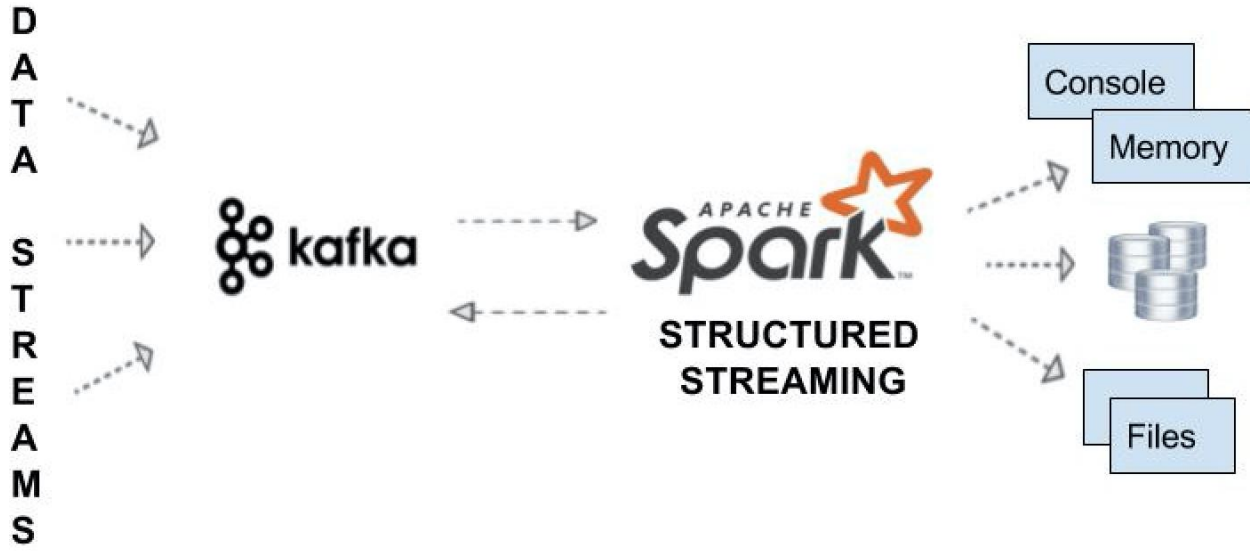
Apache Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on top of the Spark SQL engine. It allows you to process real-time data streams in a scalable, fault-tolerant, and efficient manner.

Structured Streaming is based on the concept of DataFrames and Datasets, which are high-level abstractions built on top of RDDs (Resilient Distributed Datasets), the core abstraction in Spark. DataFrames and Datasets are similar to tables in a relational database and provide a rich set of operations for data manipulation and transformation.

Structured Streaming supports a variety of input sources, such as Kafka, Flume, and HDFS, and allows you to apply SQL-like operations to the incoming data streams. You can also integrate with various output sinks, such as HDFS, Cassandra, and JDBC.

One of the key advantages of Spark Structured Streaming is its ability to handle late data and out-of-order data, which are common in real-world streaming applications. It also provides a fault-tolerant processing model that ensures that data is processed exactly once, even in the event of failures.

Overall, Spark Structured Streaming is a powerful tool for processing real-time data streams and is widely used in various industries, such as finance, healthcare, and telecommunications, among others.

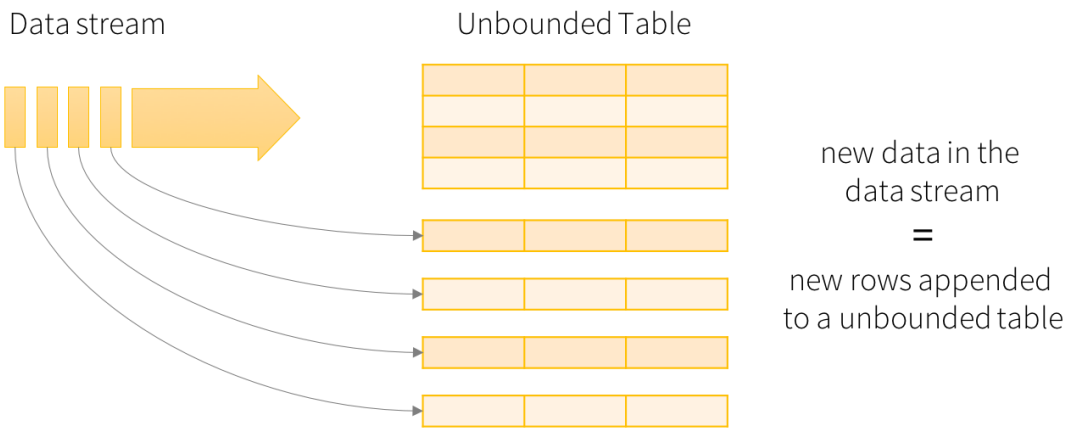


Programming Model

The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended. This leads to a new stream processing model that is very similar to a batch processing model. You will express your streaming computation as standard batch-like query as on a static table, and Spark runs it as an *incremental* query on the *unbounded* input table. Let's understand this model in more detail.

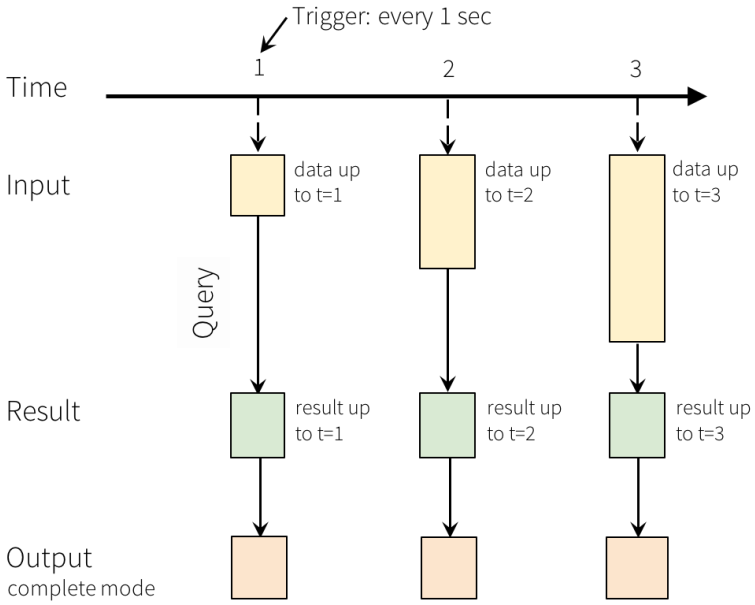
Basic Concepts

Consider the input data stream as the "Input Table". Every data item that is arriving on the stream is like a new row being appended to the Input Table.



Data stream as an unbounded table

A query on the input will generate the “Result Table”. Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we would want to write the changed result rows to an external sink



Programming Model for Structured Streaming

The “Output” is defined as what gets written out to the external storage. The output can be defined in a different mode:

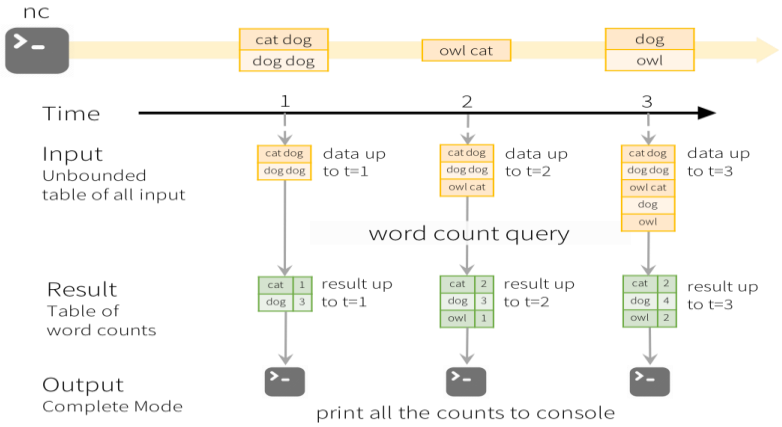
Complete Mode - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.

Append Mode - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.

Update Mode - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1). Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn't contain aggregations, it will be equivalent to Append mode.

Note that each mode is applicable on certain types of queries. This is discussed in detail later.

To illustrate the use of this model, let's understand the model in context of the Quick Example above. The first lines DataFrame is the input table, and the final wordCounts DataFrame is the result table. Note that the query on streaming lines DataFrame to generate wordCounts is exactly the same as it would be a static DataFrame. However, when this query is started, Spark will continuously check for new data from the socket connection. If there is new data, Spark will run an "incremental" query that combines the previous running counts with the new data to compute updated counts, as shown below.



Model of the Quick Example

Example One: Words Count

1. Step write script for the structure stream

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
from pyspark.sql import functions as func

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

# Create DataFrame representing the stream of input lines from connection to localhost:9999
stream = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```

First thing in the script we define the libraries we want to use through the program
Then create spark session.

After creating the spark session we start to create DataFrame that represent the stream input Lines from connection to localhost:9999.

After reading stream input we splits the lines read from stream into words.

```
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)
```

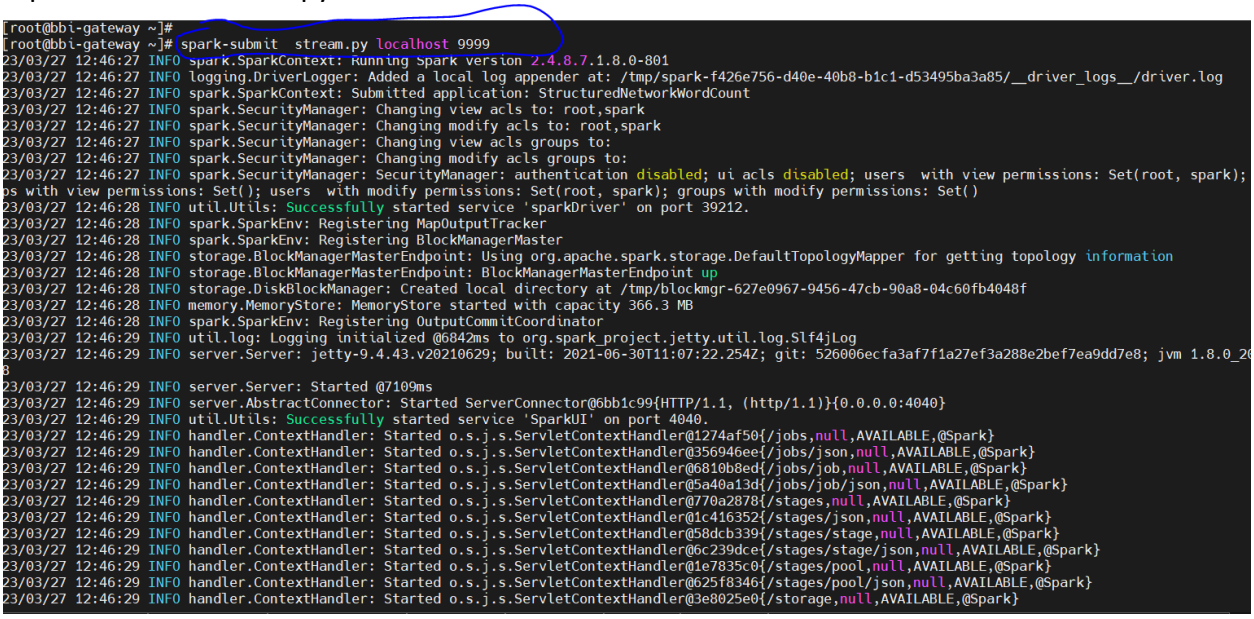
Then we start to implement the logic which is count the words group by each word in that Example.

```
# Generate running word count
wordCounts = words.groupBy("word").count()
```

Then we run the query and print it to console.

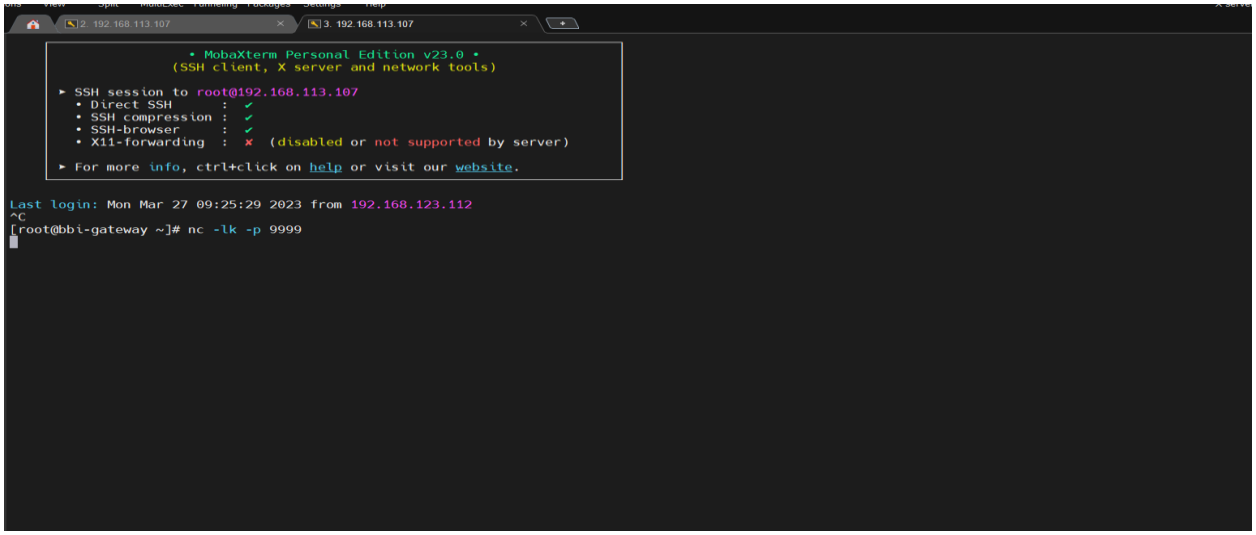
```
# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

2. Run the program using spark-submit command
   “spark-submit stream.py localhost 9999”

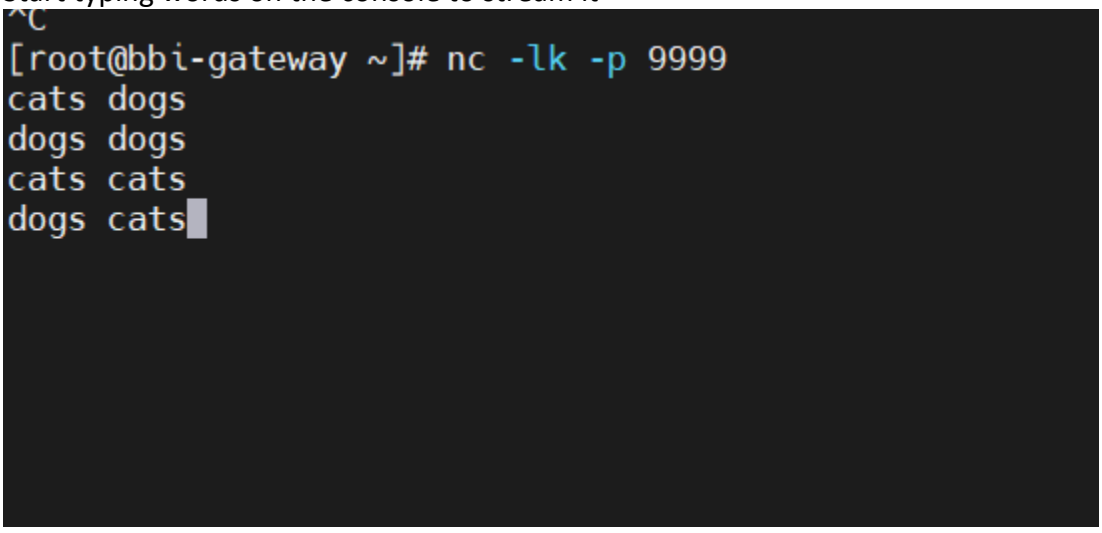


On the other hand we need to configure the terminal for the machine to act as streaming source and network socket.

3. Open another terminal and write this command to make terminal act as stream source
" nc -q 1 localhost 9999"



4. Start typing words on the console to stream it



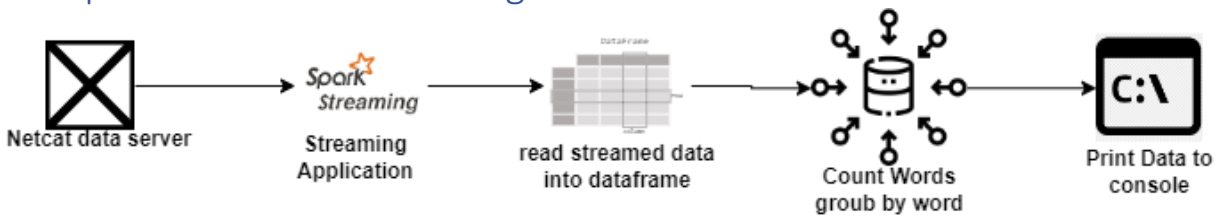
You will find the streaming results written on console.

```
-------------------------------------------
Batch: 1
-------------------------------------------
23/03/27 12:52:14 INFO codegen.CodeGenerator: Code generated in 20.048575 ms
+----+-----+
|word|count|
+----+-----+
|dogs|    3|
|cats|    3|
+----+-----+

23/03/27 12:52:14 INFO v2.WriteToDataSourceV2Exec: Data source writer org.apache.spark.sql.execution.streaming.sources.MicroBatchWriter@4a049899 committed.
23/03/27 12:52:14 INFO spark.SparkContext: Starting job: start at NativeMethodAccessorImpl.java:0
23/03/27 12:52:14 INFO scheduler.DAGScheduler: Job 3 finished: start at NativeMethodAccessorImpl.java:0, took 0.000119 s
23/03/27 12:52:14 INFO streaming.CheckpointFileManager: Writing atomically to hdfs://bbi-master-2.test.local:8020/tmp/temporary-0451d1e1-7749-4041-bfc6-0eac9
66c04d/commits/1 using temp file hdfs://bbi-master-2.test.local:8020/tmp/temporary-0451d1e1-7749-4041-bfc6-0eac9b66c04d/commits/.1.c82f5d23-543d-4269-954c-dc
d7dab3aa7.tmp
23/03/27 12:52:14 INFO streaming.CheckpointFileManager: Renamed temp file hdfs://bbi-master-2.test.local:8020/tmp/temporary-0451d1e1-7749-4041-bfc6-0eac9b66c
4d/commits/.1.c82f5d23-543d-4269-954c-dc1d7dab3aa7.tmp to hdfs://bbi-master-2.test.local:8020/tmp/temporary-0451d1e1-7749-4041-bfc6-0eac9b66c04d/commits/1
23/03/27 12:52:14 INFO streaming.MicroBatchExecution: Streaming query made progress: {
  "id" : "3264dfcd-3f50-4602-9e61-eb2bad691249",
  "runId" : "55b9e1bd-0765-401b-b0f5-bca4947470e4",
  "name" : null,
  "timestamp" : "2023-03-27T10:52:01.003Z",
```

The full code example.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

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

    # Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)



# Generate running word count
wordCounts = words.groupBy("word").count()

 # Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Example One: Word count diagram



Example two: Rooms Average temperature

1. Write the script

First thing in the script we define the libraries we want to use through the program
Then create spark session.

After creating the spark session, we start to create DataFrame that represent the stream input Lines from connection to localhost:9999.

After reading stream input we splits the lines read from stream into words.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
from pyspark.sql import functions as func

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

# Create DataFrame representing the stream of input lines from connection to localhost:9999
stream = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```

Then we start to implement the logic which is count the get the average of temperature group by each word in that
Example.

```
    .load()

df_room_temp_stream = stream.selectExpr("split(value, ' ')[0] as room","split(value, ' ')[1] as temp")

df_room_temp = df_room_temp_stream.selectExpr("CAST(room AS STRING)", "CAST(temp AS INT)")

roomAVGtemp=df_room_temp.groupBy('room').avg('temp')
```

Then we run the query and print it to console.

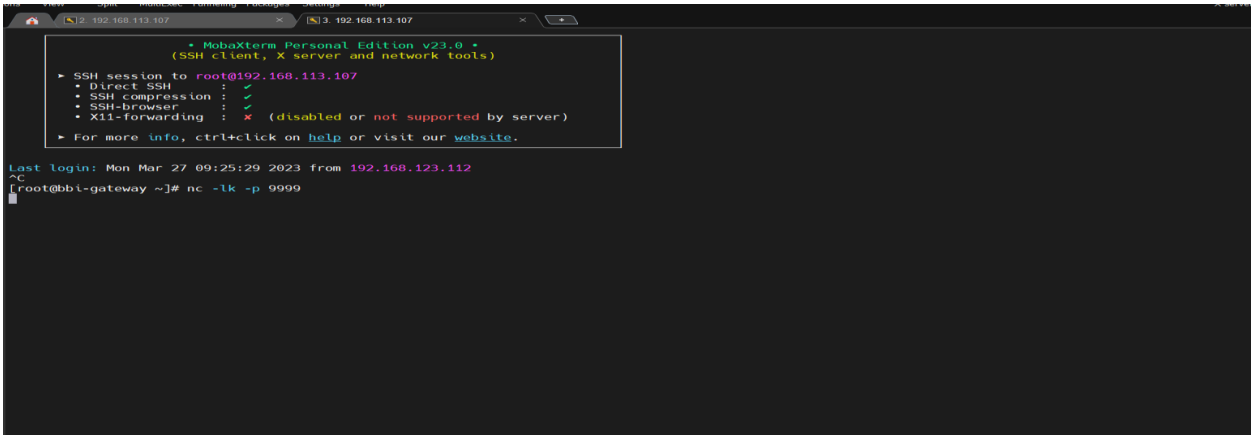
```
# Start running the query that prints the running counts to the console
query = roomAVGtemp \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

1. Run the program using spark-submit command
   “spark-submit tempraturestream.py localhost 9999”

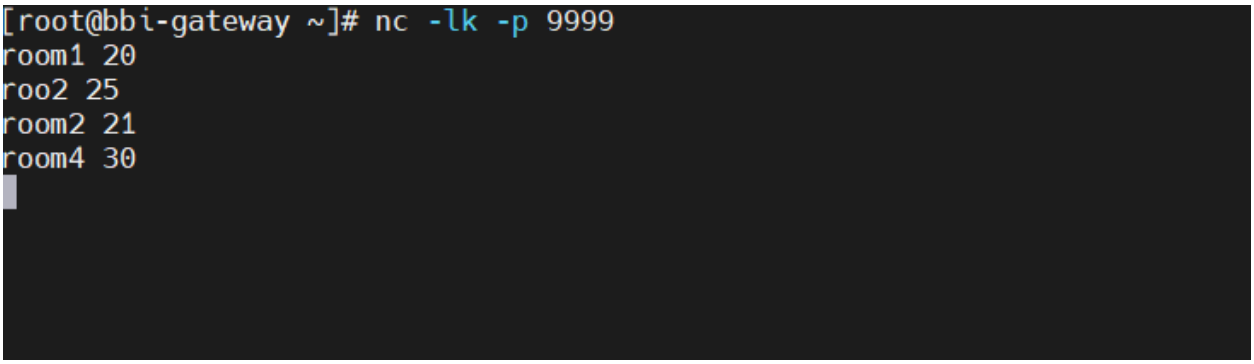
```
[root@bbi-gateway ~]# spark-submit  tempraturestream.py localhost 9999
23/03/27 13:51:59 INFO spark.SparkContext: Running Spark version 2.4.8.7.1.8.0-801
23/03/27 13:51:59 INFO logging.DriverLogger: Added a local log appender at: /tmp/spark-216c351d-249d-4ee4-96b6-0b2b42ec6141/__driver_logs__/driver.log
23/03/27 13:51:59 INFO spark.SparkContext: Submitted application: StructuredNetworkWordCount
23/03/27 13:51:59 INFO spark.SecurityManager: Changing view acls to: root,spark
23/03/27 13:51:59 INFO spark.SecurityManager: Changing modify acls to: root,spark
23/03/27 13:51:59 INFO spark.SecurityManager: Changing view acls groups to:
23/03/27 13:51:59 INFO spark.SecurityManager: Changing modify acls groups to:
23/03/27 13:51:59 INFO spark.SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users  with view permissions: Set(root, spark); grou
ps with view permissions: Set(); users  with modify permissions: Set(root, spark); groups with modify permissions: Set()
```

   On the other hand, we need to configure the terminal for the machine to act as streaming source and network socket.

2. Open another terminal and write this command to make terminal act as stream source " nc -q 1 localhost 9999"



3. Start typing room name and temperature on the console to stream it.



You will find the stream result written on the console.

```
-------------------------------------------
Batch: 2
-------------------------------------------
+-----+---------+
| room|avg(temp)|
+-----+---------+
|room4|     30.0|
|room1|     23.0|
|room2|     23.0|
+-----+---------+

23/03/27 14:16:25 INFO v2.WriteToDataSourceV2Exec: Data source writer org.apache.spark.sql.execution.streaming.sourc
23/03/27 14:16:25 INFO spark.SparkContext: Starting job: start at NativeMethodAccessorImpl.java:0
23/03/27 14:16:25 INFO scheduler.DAGScheduler: Job 5 finished: start at NativeMethodAccessorImpl.java:0, took 0.0001
23/03/27 14:16:25 INFO streaming.CheckpointFileManager: Writing atomically to hdfs://bbi-master-2.test.local:8020/tm
eb12cd/commits/2 using temp file hdfs://bbi-master-2.test.local:8020/tmp/temporary-ca24c6a4-7917-41fc-b223-df2087eb1
858a151cc.tmp
23/03/27 14:16:25 INFO streaming.CheckpointFileManager: Renamed temp file hdfs://bbi-master-2.test.local:8020/tmp/te
cd/commits/.2.d96545fb-fd47-4884-9d4c-ad0858a151cc.tmp to hdfs://bbi-master-2.test.local:8020/tmp/temporary-ca24c6a4
23/03/27 14:16:25 INFO streaming.MicroBatchExecution: Streaming query made progress: {
  "id" : "d1a65384-c256-4ba1-99dc-6cdc249e555f",
  "runId" : "130df05b-fc4f-452f-b176-a99cf71713de",
  "name" : null,
  "timestamp" : "2023-03-27T12:15:38.173Z",
  "batchId" : 2,
  "numInputRows" : 1,
  "inputRowsPerSecond" : 90.90909090909092,
  "processedRowsPerSecond" : 0.02103536044090155,
  "durationMs" : {
    "addBatch" : 47344,
    "getBatch" : 0,
    "getEndOffset" : 0,
    "queryPlanning" : 92,
    "setOffsetRange" : 0,
    "triggerExecution" : 47539,
```

Full code

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
from pyspark.sql import functions as func

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

# Create DataFrame representing the stream of input lines from connection to localhost:9999
stream = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into room , temprature columns
df_room_temp_stream = stream.selectExpr("split(value, ' ')[0] as room","split(value, ' ')[1] as temp")

df_room_temp = df_room_temp_stream.selectExpr("CAST(room AS STRING)", "CAST(temp AS INT)")

#calculate average of temprature groub by temp

roomAVGtemp=df_room_temp.groupBy('room').avg('temp')



# Start running the query that prints the running counts to the console
query = roomAVGtemp \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Example Two: temperature analysis diagram