Kafka + Spark Streaming + PySpark

Spark Streaming

Connecting the Dots

Student Name: Norina Akhtar – 19643

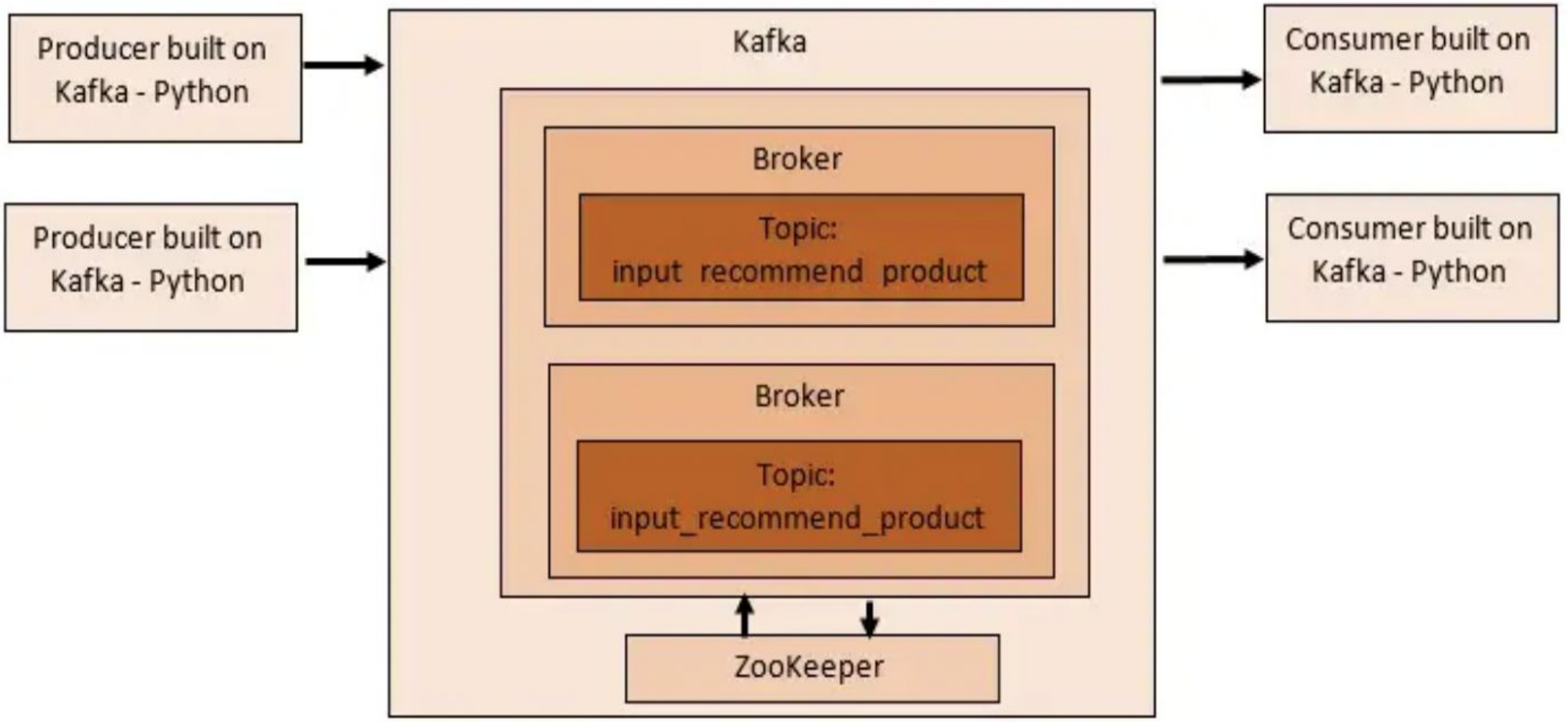
Introduction

Real-time data ingesting is a common problem in real-time analytics, because in a platform such as e-commerce, active users in a given time and the number of events created by each active user are many. Hence, recommendations (i.e., predictions) for each event or groups of events are expected to be near real-time.

It is a distributed streaming platform, which helps to build real-time streaming data pipelines

The primary concerns are, *How we will [consume, produce, and process] these events efficiently?*

Design


Producer built on Kafka - Python
Producer built on Kafka - Python
Kafka
Broker
Topic: input recommend product
Broker
Topic: input_recommend_product
ZooKeeper
Consumer built on Kafka - Python
Consumer built on Kafka - Python

Implementing step 1 (Apache Kafka + Kafka-Python)

1.The latest version of Kafka binary distribution is available at

https://kafka.apache.org/downloads.

2. Unzip the folder

Start the zookeeper on first terminal using the following command:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Implementation

3. Starting Kafka Brokers

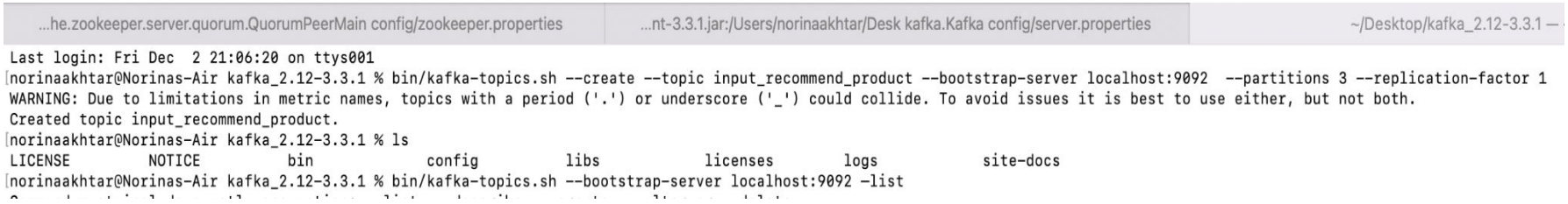
Use another terminal:

```
bin/kafka-server-start.sh config/server.properties
```

4. Creating Kafka Topics

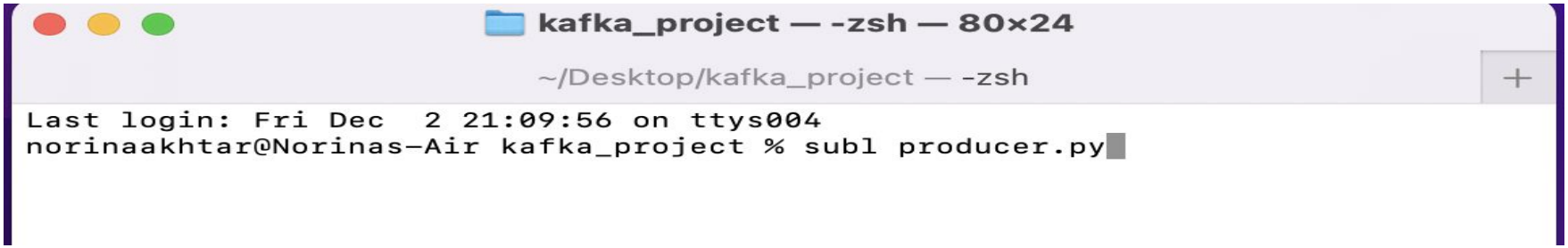
Use another command

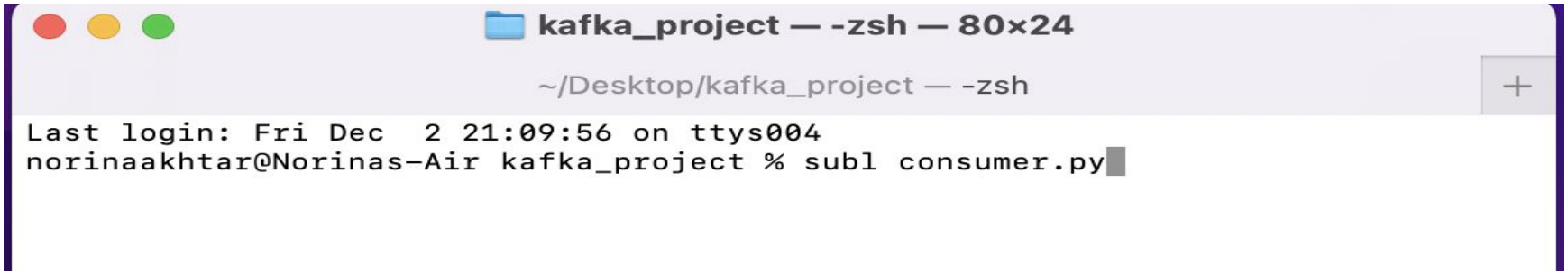
bin/kafka-topics.sh --create --topic input_recommend_product --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1



Implementation

5. Creating Producer and Consumer using Kafka-python





consumer.py UNREGISTERED

consumer.py

```python
from kafka import KafkaConsumer

consumer = KafkaConsumer('input_recommend_product', bootstrap_servers=['localhost:9092'])
for msg in consumer:
    print(msg.value)
```

producer.py UNREGISTERED

producer.py

```python
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')

producer.send('input_recommend_product', b'(1, Main Menu), (2, Phone) , (3, Smart Phone), (4,
producer.close()

```

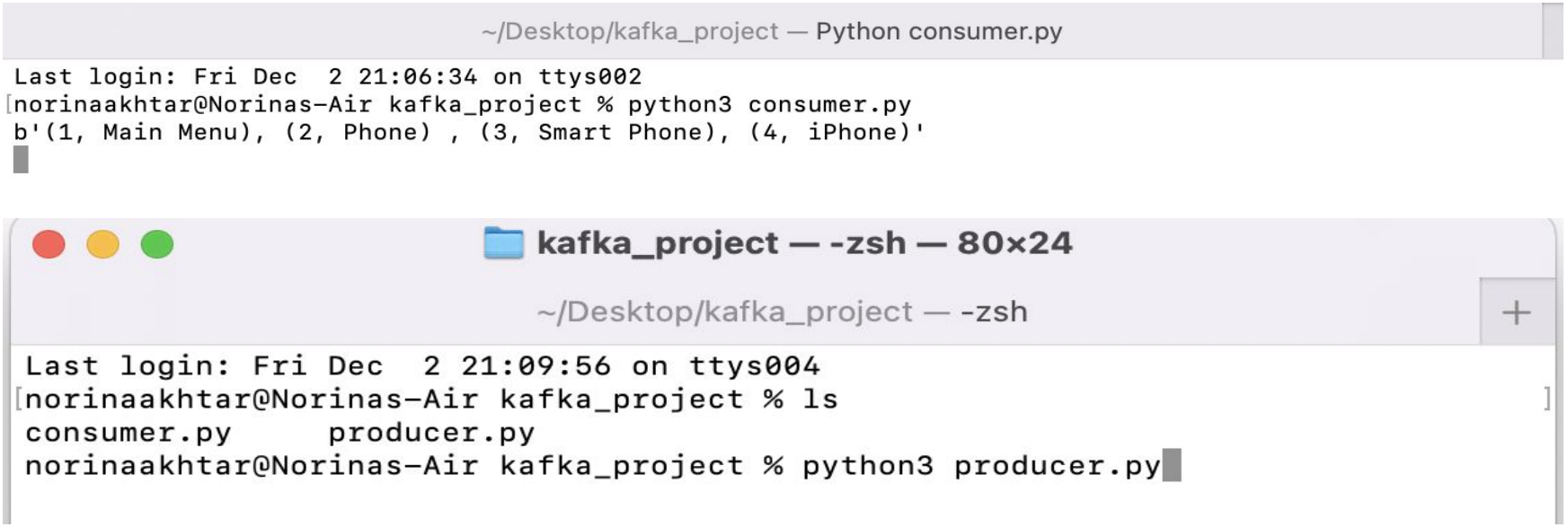
Run Files

Steps:

i. First run consumer.py

ii. Run producer.py on another terminal

iii. You can see **output** on first terminal



Spark streaming

Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including (but not limited to) Kafka, Flume, and Amazon Kinesis. This processed data can be pushed out to file systems, databases, and live dashboards.

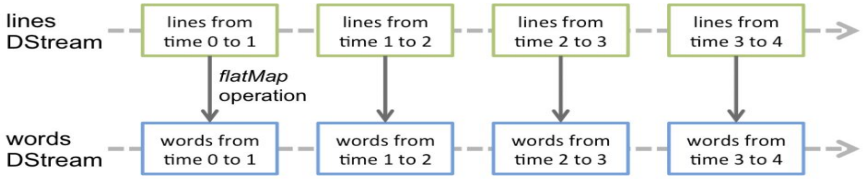
Spark streaming

Spark Streaming provides a high-level abstraction called *discretized stream* (*DStream*), which represents a continuous stream of data.

- DStreams can be created
  - from input data streams from sources such as Kafka, Flume, and Kinesis, or
  - by applying high-level operations on other DStreams.
- Internally, a DStream is represented as a sequence of RDDs.

A DStreams is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable, distributed dataset.

Each RDD in a DStreams contains data from a certain interval,



The flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream.

Implementing spark streaming

Setup

Running scala with sbt:

1. Make sure you have the Java 8 JDK (also known as 1.8)

2. Download SBT:

https://www.scala-sbt.org/download.html

MAC:

brew install sbt

Create the project:

- cd to an empty folder.
- Run the following command sbt new scala/scala3.g8. This pulls the 'scala3' template from GitHub. It will also create a target folder, which you can ignore.
- When prompted, name the application wordcount. This will create a project called "wordcount".

Step 1: Write a standalone application in Spark to count the number of words in a received stream.

Paste the code given code in:

/Users/norinaakhtar/Desktop/new/hello-world/src/main

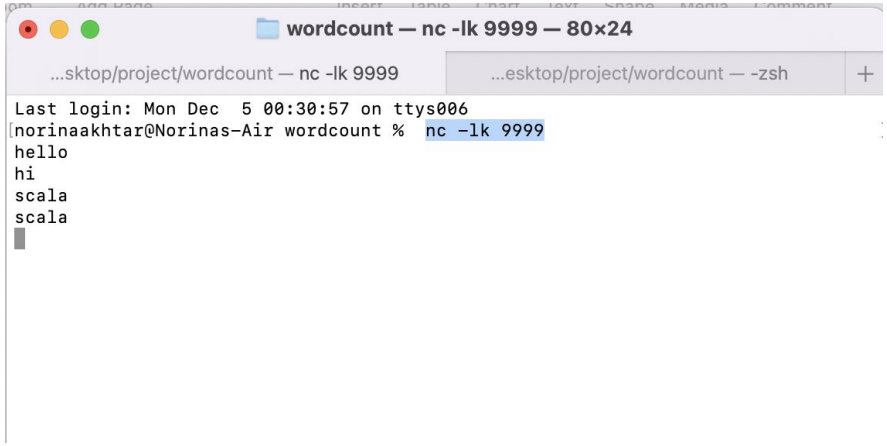
```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.storage.StorageLevel

object NetworkWordCount {
  def main(args: Array[String]) {
    val ssc = new StreamingContext("local[2]",
                  "NetworkWordCount", Seconds(1))
    val lines = ssc.socketTextStream("127.0.0.1", 9999)
    val words = lines.flatMap(_.split(" "))
    val pairs = words.map(x => (x, 1))
    val wordCounts = pairs.reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Build.sbt should look like this

```
name := "wordcount"
version := "1.0"
scalaVersion := "2.10.3"
libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "0.9.0-incubating",
    "org.apache.spark" %% "spark-streaming" % "0.9.0-incubating"
)
```

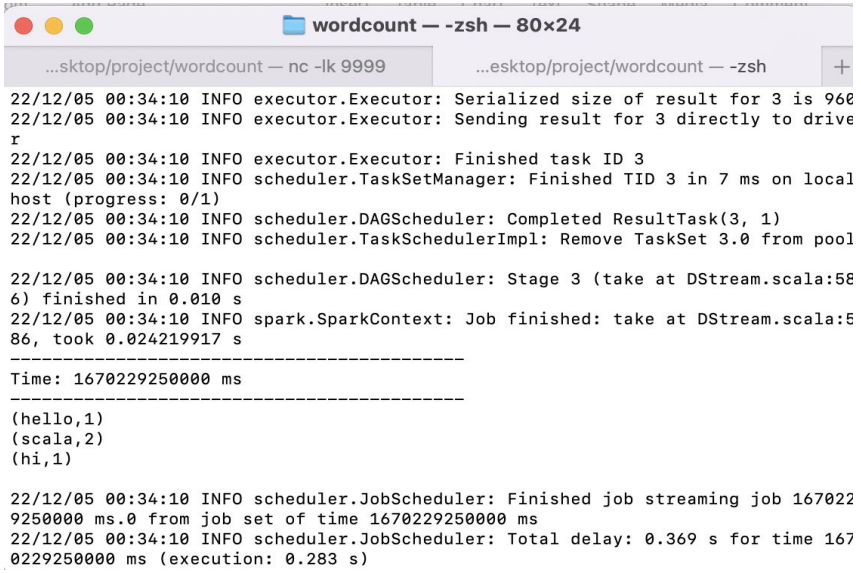
Step1: Result

Run nc -lk 9999 on Terminal 1:



Terminal 2

1. **Cd into wordcount**
2. **2. $ sbt package**
3. **3. $ sbt run**



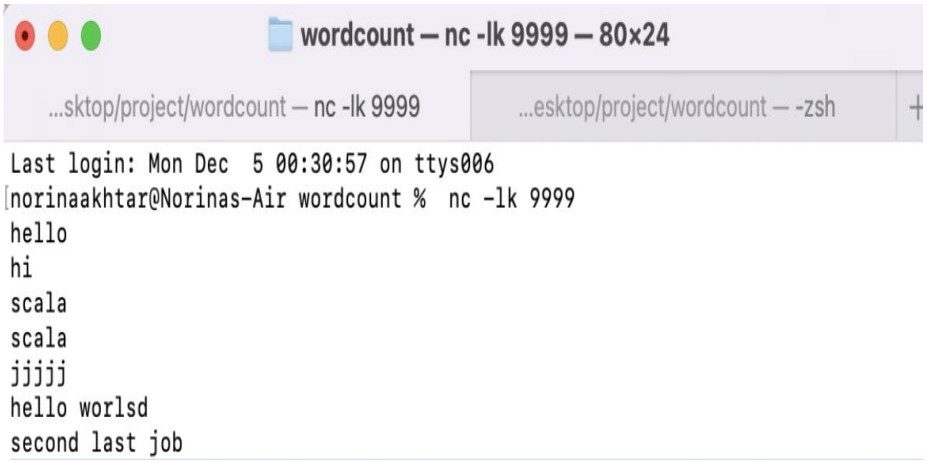
Step 2: Extend the code to generate word count over last 30 seconds of data, and repeat the computation every 10 seconds.

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.storage.StorageLevel
object NetworkWordCount {
  def main(args: Array[String]) {
    val updateFunc = (values: Seq[Int], state: Option[Int]) => {
      val currentCount = values.foldLeft(0)(_ + _)
      val previousCount = state.getOrElse(0)
      Some(currentCount + previousCount)
    }
    val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
    ssc.checkpoint(".")
    val lines = ssc.socketTextStream("127.0.0.1", 9999)
    val words = lines.flatMap(_.split(" "))
    val pairs = words.map(word => (word, 1))
    val stateWordCounts = pairs.updateStateByKey[Int](updateFunc)
    stateWordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Step2 : Result

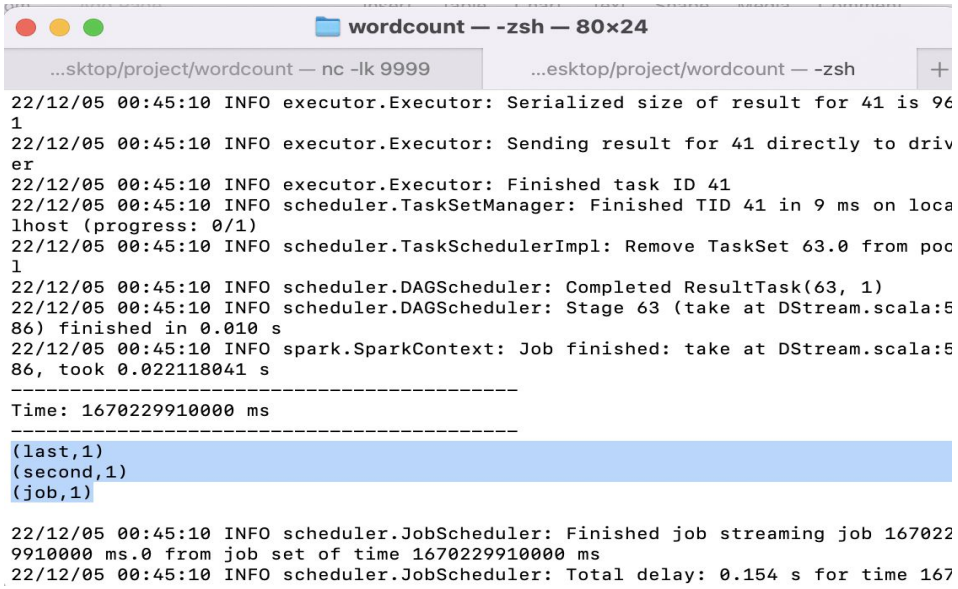
Terminal 1

Run nc -lk 9999 on Terminal 1:



Terminal 2

1. **Cd into wordcount**
2. **2. $ sbt package**
3. **3. $ sbt run**

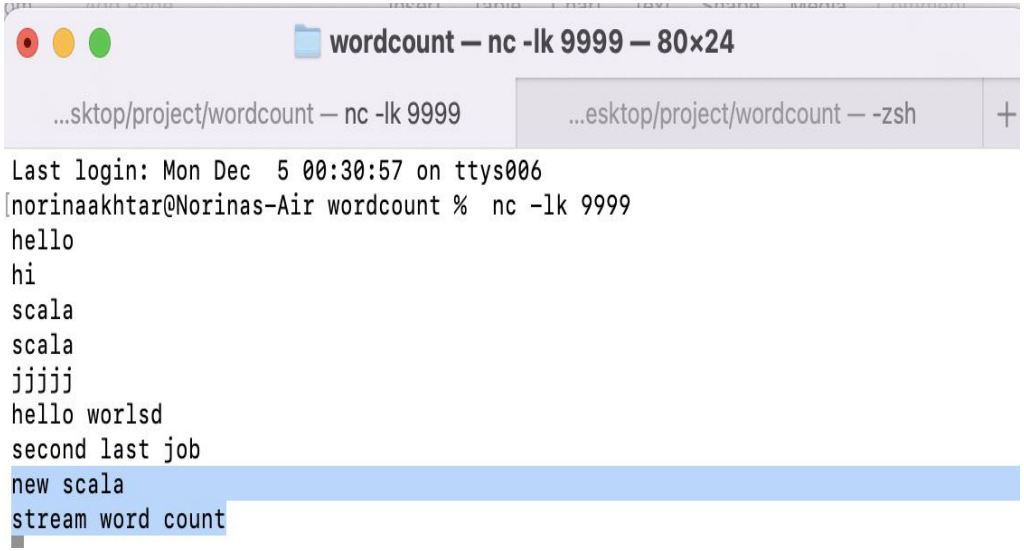


Step 3: Maintain a continuously updated word count for all the words in the stream.

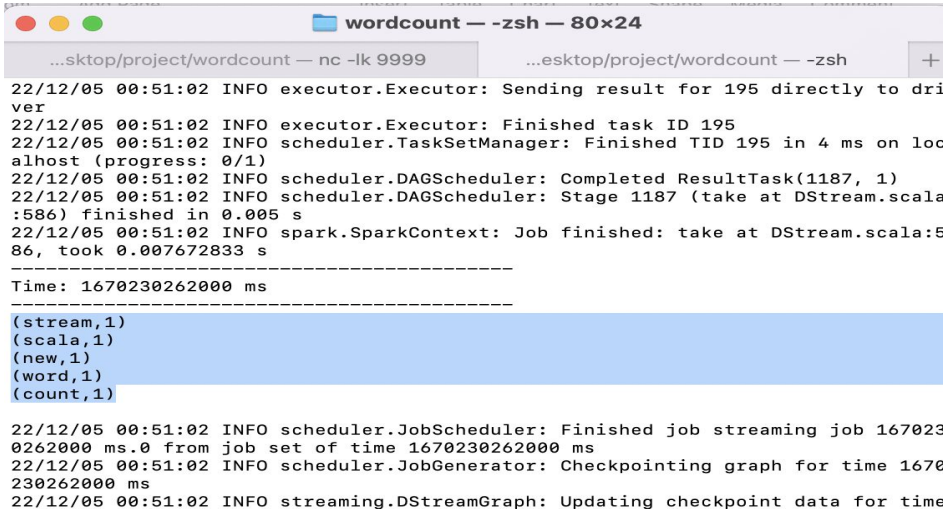
```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.storage.StorageLevel
object NetworkWordCount {
  def main(args: Array[String]) {
    val updateFunc = (values: Seq[Int], state: Option[Int]) => {
      val currentCount = values.foldLeft(0)(_ + _)
      val previousCount = state.getOrElse(0)
      Some(currentCount + previousCount)
    }
    val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
    ssc.checkpoint(".")
    val lines = ssc.socketTextStream("127.0.0.1", 9999)
    val words = lines.flatMap(_.split(" "))
    val pairs = words.map(word => (word, 1))
    val stateWordCounts = pairs.updateStateByKey[Int](updateFunc)
    stateWordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Step 3: Result

Terminal 1



Terminal 2



Connecting the Dots (Python, Spark, and Kafka)

Python, Spark, and Kafka are vital frameworks in data scientists' day to day activities. It is essential to enable them to integrate these framework

Frequently, Data scientists prefer to use Python (in some cases, R) to develop machine learning models. Here, they have a valid justification since data-driven solutions arrive with many experiments. Numerous interactions with the language we use to develop the models are required to perform experiments, and the libraries and platforms available in python to develop machine-learning models are tremendous. This is a valid argument; however, we confront issues when these models are applied to production

To overcome all the problems, we can identify a set of dots that could be appropriately connected. In this process, I attempt to connect these dots, which are Python, Apache Spark, and Apache Kafka

Implementation of Connecting the Dots (Python, Spark, and Kafka

Running out of credit

Enhancement ideas

- We can integrate kafka stream with **Redpanda** , which natively supports the kafka api so it works out of the box with existing tools and integrations. However its, 10x faster and upto 7x more cost effective.

- Different frameworks and formats i-e avro, parquet and persist can be used to improve the performance and tuning spark job.

Conclusion

- Apache — Kafka is a vital platform in building real-time processing solutions. This article gives you an excellent start to set up Apache — Kafka on a distributed environment and provides easy guidance to produce and consume events.

- Python, Spark, and Kafka are important frameworks in a data scientist's daily activities.

- It can help data scientists to perform their experiments in Python while deploying the final model in a scalable production environment.

References

https://spark.apache.org/docs/latest/api/python/index.html

https://spark.apache.org/docs/latest/index.html#running-the-examples-and-shell

https://docs.confluent.io/kafka-clients/python/current/overview.html#ak-python

https://spark.apache.org/docs/latest/submitting-applications.html