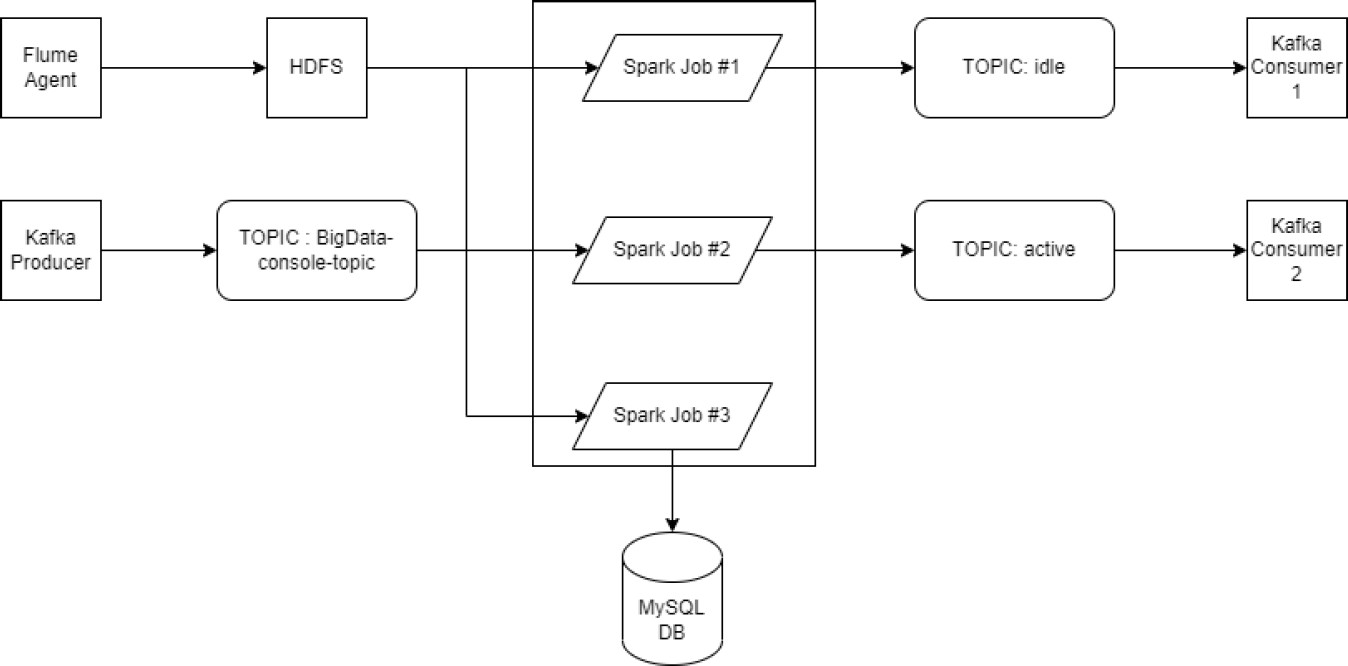
Assignment 2 Report

This report mainly focuses on the setup of the ETL system. The outcome of code will be demonstrated in my demo video.

Large portions of code and wording are borrowed from course exercise files, as the professor did a wonderful job writing them. I followed the code in the exercise precisely and successfully built this.



## Introduction:

My design of the ETL system is exactly the same as the diagram provided, except I only use a Flume agent as my data source. I combined 9-mix, 10-mix, 11-mix, and 12-mix IoT data into a single JSON file, and this file will be used as my data.

A general ETL process is like this:

A script will output this JSON file into a log file in the local file system one line per second. A Flume agent will copy the content of the log file onto an HDFS folder.

Three spark streams are actively monitoring the folder, any new file creation will trigger three spark jobs.

The first two spark jobs will group the data by the activity types, and count them. One spark job will only count the “ACTIVE” data, and send the results to the “ACTIVE” sink. the other will count the “IDLE” data, and send the result to the “IDLE” sink. The output mode for those two spark jobs will be “UPDATE”.

The third spark job will load the data as a spark SQL dataset, and output this dataset to an external POSTERS SQL database system.

A Kafka server will keep running in the backend. There are two topics, “ACTIVE” and “IDLE”, both receive data from the corresponding spark jobs.

Finally, two Kafka console consumers are listening to those two topics. Once new data is sent to a sink, the console consumers will print the data in the console line terminal.

The system is deployed on the GCP.

## Setup of Zookeeper

ZooKeeper is the resource coordinator used for most of the Apache Projects. It coordinates resources between each Apache program.

Once it is up and running in the GCP, there is no need to run it again in the future (if it can successfully run).

The code here is the same as class exercise. Since I configured Zookeeper a long time ago, no detailed screenshot will be provided.

#First Download ZooKeepr and Unzip

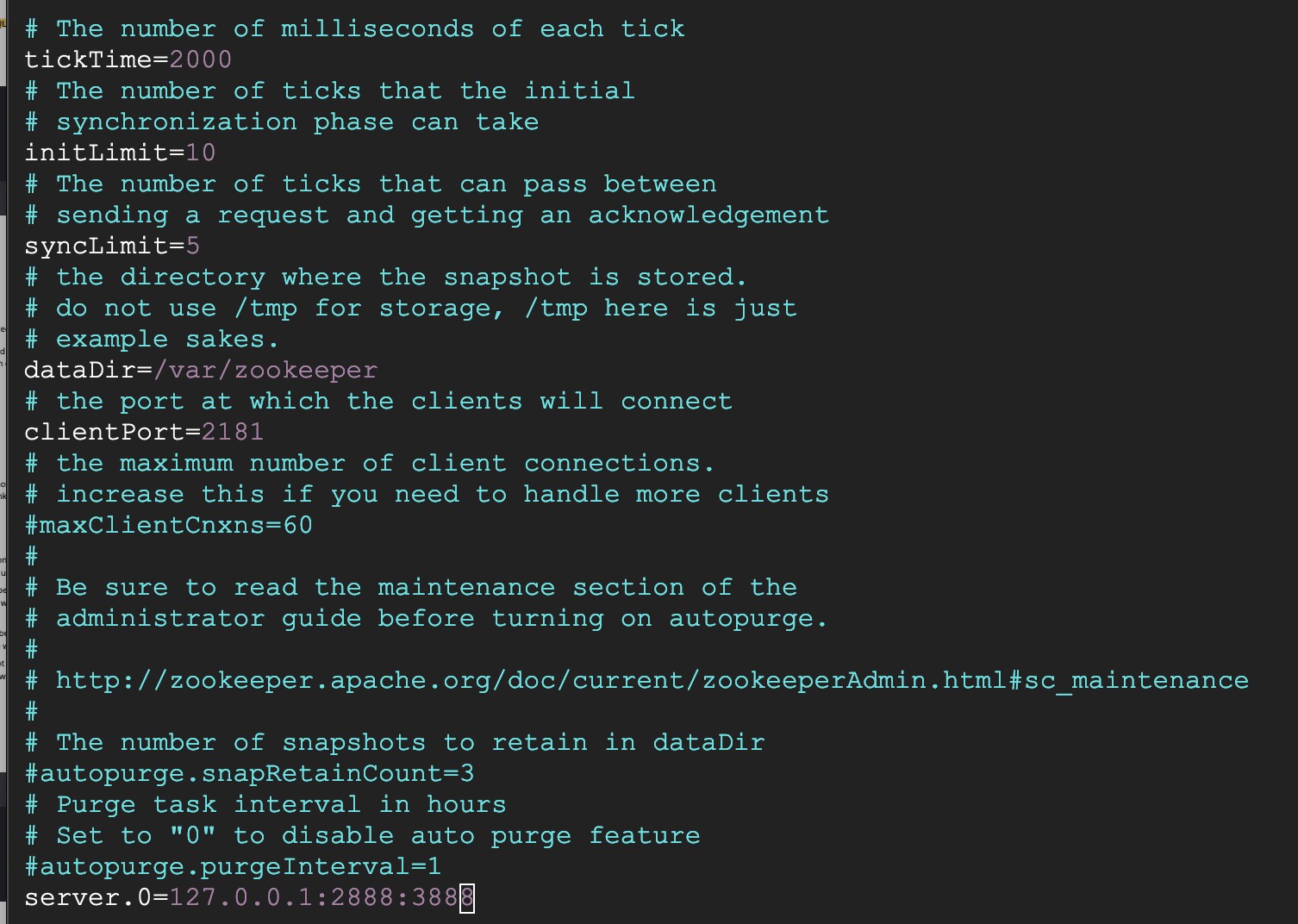
# wget

**https://archive.apache.org/dist/zookeeper/zookeeper-3.4.14/z ookeeper-3.4.14.tar.gz**

# tar -xzf zookeeper-3.4.14.tar.gz

#Make the config file

### cd zookeeper-3.4.14/conf cp zoo\_sample.cfg zoo.cfg vi zoo.cfg



#make sure you have at the end of the file: server.0=127.0.0.1:2888:3888

#Create zookeepr variables Folder

### sudo mkdir /var/zookeeper

# Change ownership to local user

### sudo chown [Local User]:[Local User] /var/zookeeper

# Create myid

# Enter server number. First server has number "0" so put "0" in myid file # Server number should match server in zoo.cfg

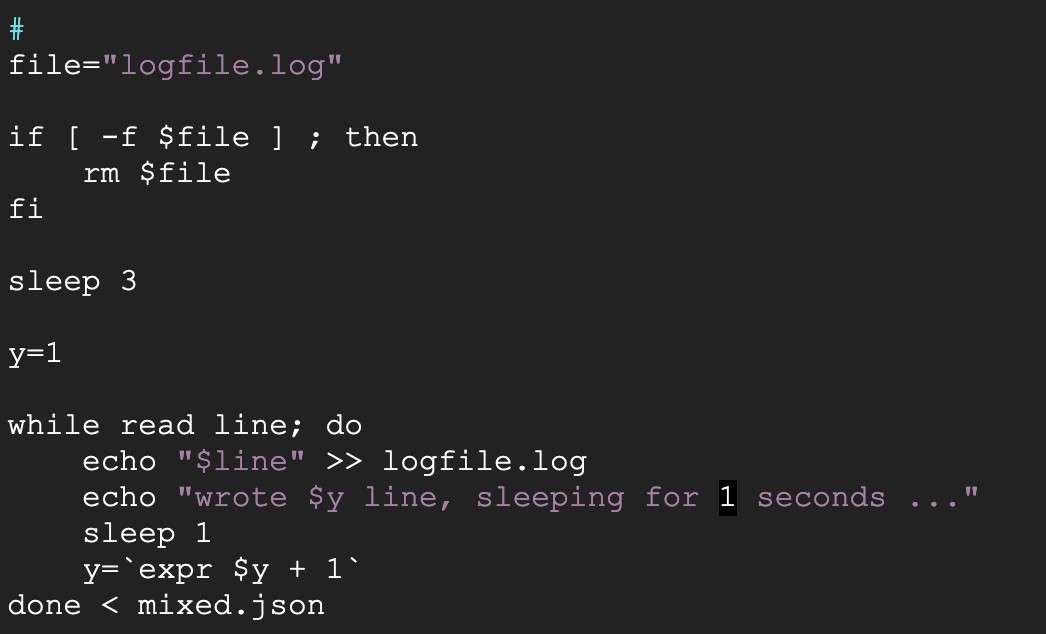
### vi /var/zookeeper/myid

-- Start Zookeeper

### Cd ~/zookeeper bin/zkServer.sh start

Once the ZooKeeper is up and running, there is no need to do any operation unless there is an error.

Setup of Flume

Flume is the tool we use to transfer JSON files to the HDFS system. Normally Flume is used to transfer log files. In our case, we need to accommodate this by recreating our mix.json file into a continuously generating log file. We can achieve this using a script:

This file will first delete logfile if it has already been created. Then, it will create the logfile.log, and output one line of mixed.json file to it every second.

Then, we can start our setup of Apache Flume. #First, download and unzip Flume

### wget

[**http://archive.apache.org/dist/flume/1.9.0/apache-flume-1.9.0-bin.tar.gz**](http://archive.apache.org/dist/flume/1.9.0/apache-flume-1.9.0-bin.tar.gz) **tar -xvf apache-flume-1.9.0-bin.tar.gz**

# Add flume path to bash source, so that we can run fulme from any directory without giving full path

### vi .bashrc

-- Go to the end of the file and enter

### export PATH=$PATH:/home/[username]/apache-flume-1.9.0-bin/bin

**source .bashrc**

# Solve the version conflict problem:

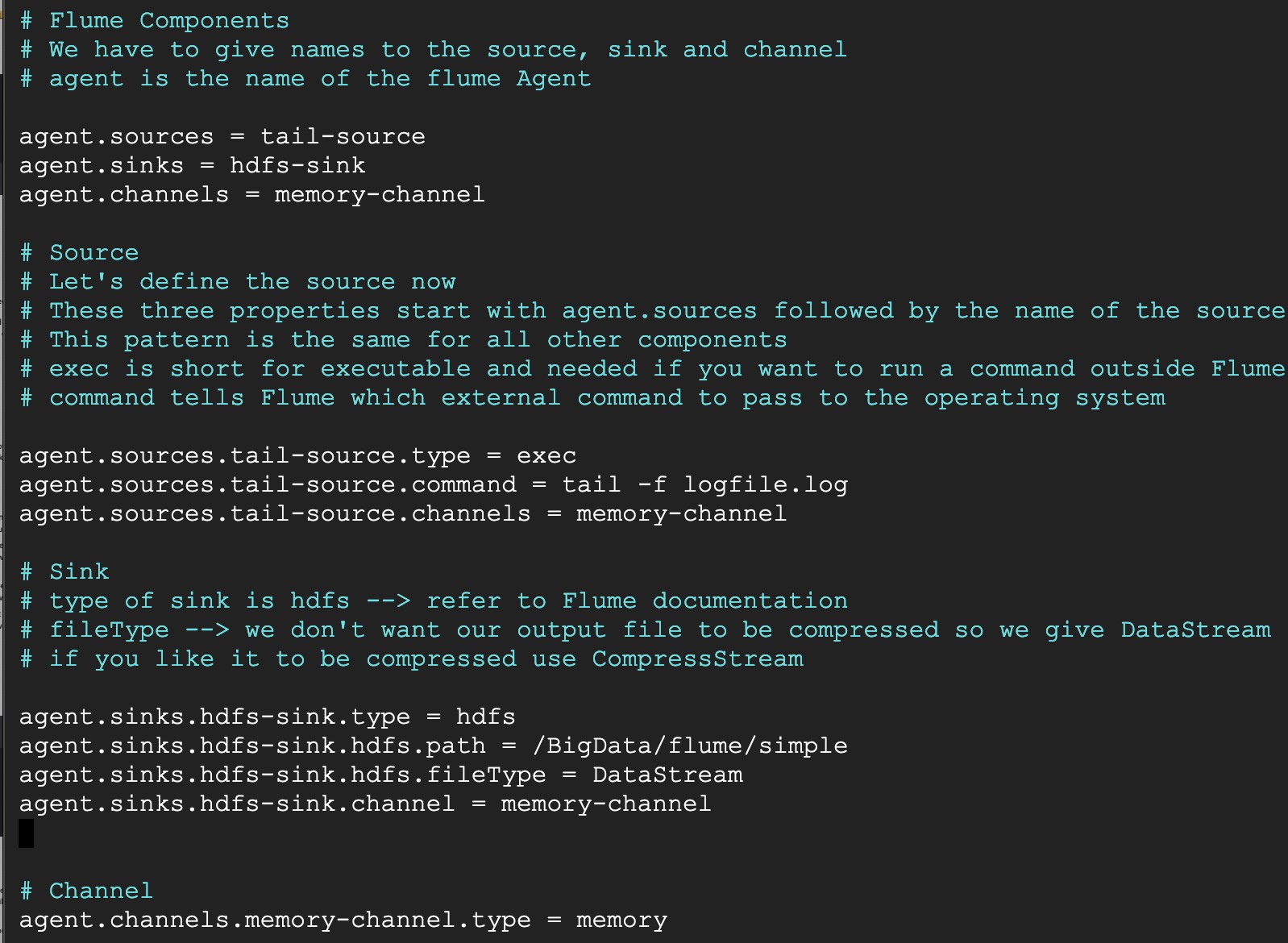
remove older version of Guava from flume lib

rm /home/[username]/apache-flume-1.9.0-bin/lib/guava-11.0.2.jar

-- copy GCP's version of guava into flume lib (Flume will now use this version avoiding conflict) cp /lib/hadoop/lib/guava-28.2-jre.jar /home/username/apache-flume-1.9.0-bin/lib/.

Then, we need to configure flume, ie, let the flume agent know how to transfer the file.

We need to tell flume the location of our logfile, where to store the files in HDFS, and how to store them. We achieve this by creating a config file for flume.

Called flume-data.config. Later when invoking flume we need to specify the location of thisfile.

In this file, we told Flume the property of our source(the logfile) and sink. The sink for flume is where to store the data, we specify an HDFS folder called /BigData/flume/simple. Later this folder will be used as a trigger for spark.

Then we can invoke the flume agent by:

**flume-ng agent --conf /home/y279guo/flume/data -f /home/y279guo/flume/data/flume-data.config**

**-Dflume.root.logger=DEBUG,console -n agent**

Notice my user name is y279guo, and I placed the flume configuration file in

**/home/y279guo/flume/data.**

## Setup of Kafka Server and Listener

Kafka is used as sinks for our data. A Kafka server will maintain two topics called “active” and “idle”. Then, two console consumers will listen to those two topics.

Once data is sent to those topics by spark streaming, the two consumers will display the corresponding data.

#First download and unzip kafka

### wget

[**https://packages.confluent.io/archive/4.1/confluent-4.1.4-2.11.tar.gz**](https://packages.confluent.io/archive/4.1/confluent-4.1.4-2.11.tar.gz) **tar -xzf confluent-4.1.4-2.11.tar.gz**

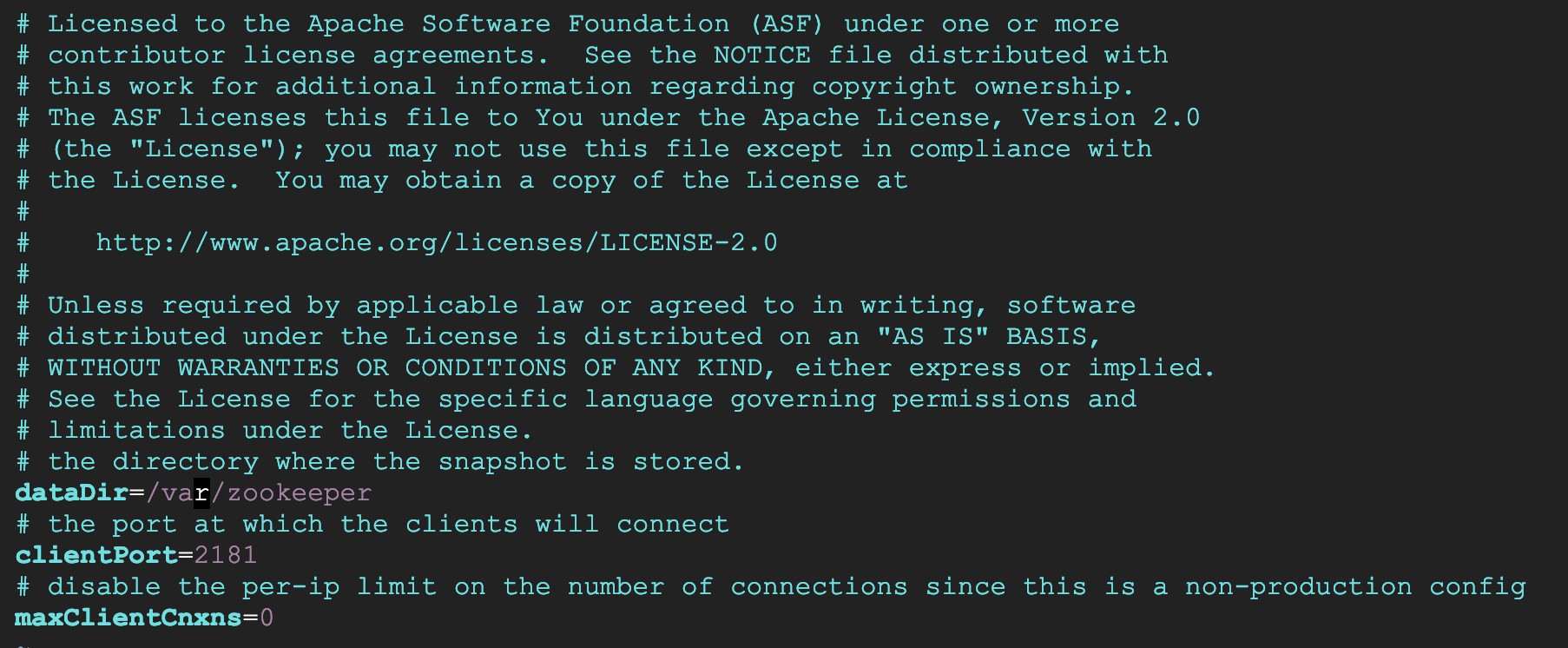
#Double check server ID is 0,

### vi etc/kafka/server.properties

#Make sure zookeeper.connect = localhost:2181 (host name and ports) and dataDir =

/var/zookeeper

### vi etc/kafka/zookeeper.properties



#First start kafka with log to check no errors:

### bin/kafka-server-start etc/kafka/server.properties

#Then stop the server and start it without log to freeup terminal

### bin/kafka-server-stop

**nohup bin/kafka-server-start etc/kafka/server.properties > /dev/null 2>&1 &**

Then, we will create two topics, active and idle

### bin/kafka-topics --create --zookeeper localhost:2181

**--replication-factor 1 --partitions 3 --topic idle**

### bin/kafka-topics --create --zookeeper localhost:2181

**--replication-factor 1 --partitions 3 --topic active**

Then we will create two consumers listening to active and idle, notice we need one terminal for each.

### bin/kafka-console-consumer --bootstrap-server localhost:9092 --topic active

**bin/kafka-console-consumer --bootstrap-server localhost:9092 --topic idle**

## Setup of Spark jobs

Spark streaming is used to do transformation on the loaded data and send them to corresponding sinks. In our case, incoming data needs to be filtered and grouped. We need to send aggregated results to three different sinks.

Notice we are outputting aggregated results, meaning we could not use append output mode, as aggregated results may require change to previous data, but append mode does not allow that.

We will be using “update” output mode, meaning only the data which changed in the result table will be sent to the sinks. And we will perform a group by activity type on the incoming data, and do a count by type. Meaning data sent to each sink will be a count of each activity, and since new data will be continuously fetched to HDFS, the count will keep increasing until no data is coming from the source.

Spark streaming job requires a checkpoint folder to remember the state of each job. We will create chkpt and chkpt1 for idle and active data streams.

### Mkdir chkpt Mkdir chkpt1

Then, we need to open a spark shell:

### spark-shell --master local --packages org.apache.spark:spark-sql-kafka-0-10\_2.12:3.1.3

Here is my spark code for the active and idle sink:

**#Import the corresponding import org.apache.spark.sql.\_**

**import org.apache.spark.sql.types.\_**

**Here I created a schema for the incoming data, only Arrival time, device and activity type will be stored.**

**val userSchema = new StructType()**

**.add("Arrival\_Time", "string")**

**.add("Device", "string")**

**.add("gt", "string")**

**#Load the data from the HDFS where flume agent stores the data. val iot = spark.readStream.format("json")**

**.schema(userSchema)**

**.option("path", "hdfs:///BigData/flume/simple/").load()**

**#First select idle activities, which includes sit and stand val idle = iot.filter(iot("gt") === "sit" || iot("gt") === "stand")**

**#Group the data based on activity types**

**val idle\_group = idle.groupBy(col("gt")).count**

**#Create key value pair for transfer**

**val idle\_key\_val = idle\_group.withColumn("key", lit(100)).select(col("key").cast("string"), concat(col("gt"), lit(" "), col("count")).alias("value"))**

**# Send the data to the kafka server, notice to specify the checkpoint folder and ouput mode.**

**val idle\_stream = idle\_key\_val.writeStream**

**.format("kafka")**

**.option("kafka.bootstrap.servers", "localhost:9092")**

**.option("topic", "idle")**

**.option("checkpointLocation", "file:////home/y279guo/chkpt")**

**.outputMode("update")**

**.start()**

**#Exactly the same process for active data.**

**val active = iot.filter(!(iot("gt") === "sit" || iot("gt") === "stand")) val active\_group = active.groupBy(col("gt")).count**

**val active\_key\_val = active\_group.withColumn("key", lit(100)).select(col("key").cast("string"), concat(col("gt"), lit(" "), col("count")).alias("value"))**

**val active\_stream = active\_key\_val.writeStream**

**.format("kafka")**

**.option("kafka.bootstrap.servers", "localhost:9092")**

**.option("topic", "active")**

**.option("checkpointLocation", "file:////home/y279guo/chkpt1")**

**.outputMode("update")**

**.start()**

## Setup of PostgreSQL

I choose PostgreSQL because it is free and open source, and I encountered compatibility issues with JDBC when trying to connect my spark streaming job to an external MySQL database.

Reference([https://blog.knoldus.com/streaming-from-kafka-to-postgresql-through-spark-structure](https://blog.knoldus.com/streaming-from-kafka-to-postgresql-through-spark-structured-streaming/) [d-streaming/](https://blog.knoldus.com/streaming-from-kafka-to-postgresql-through-spark-structured-streaming/))

First we need to instal PostgreSQL:

### # Create the file repository configuration:

**sudo sh -c 'echo "deb** [**http://apt.postgresql.org/pub/repos/apt**](http://apt.postgresql.org/pub/repos/apt)

### $(lsb\_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

**# Import the repository signing key:**

### wget --quiet -O - https://[www.postgresql.org/media/keys/ACCC4CF8.asc](http://www.postgresql.org/media/keys/ACCC4CF8.asc) | sudo apt-key add -

**# Update the package lists:**

### sudo apt-get update

**# Install the latest version of PostgreSQL.**

### # If you want a specific version, use 'postgresql-12' or similar instead of 'postgresql':

**sudo apt-get -y install postgresql**

Then we need to use the PostgreSQL shell to set up database user information.

### sudo -u postgres psql

Then we need to create a credential for the spark job to connect:

### CREATE USER yazheng WITH ENCRYPTED PASSWORD 'password';

Final step will be create the corresponding database:

### CREATE DATABASE activity WITH OWNER yazheng

Then, we need to enter the spark shell

### #First we tell spark how to connect to PostgreSQL and how to store the data.

**def postgresqlSinkOptions: Map[String, String] = Map( "dbtable" -> "public.activity", // table**

### "user" -> "yazheng", // Database username "password" -> "password", // Password "driver" -> "org.postgresql.Driver",

**"url" -> "jdbc:postgresql://localhost:9092/activity"**

### )

**#Then we wrote the spark streaming job which writes to the DB using the above configuration, and notice the savemode is update, meaning everytime a count increases, it will be written to the external DB.**

### def writeToPostgresql(dataset: Dataset, mode: SaveMode = SaveMode.append) = {

**dataset.writeStream**

### .foreachBatch { (batch: Dataset, \_: Long) => batch.write

**.format(dataSource)**

### .options(postgresqlSinkOptions)

**.mode(mode)**

### .save()

**}**

### .start()

**.awaitTermination()**

### }

**#Then we actually invoke the function, remember our dataset is iot. writeToPostgresql(iot).**

## Order to run the whole process:

After the initial setup of zookeeper, kafka server, we can begin the ETL. Before run the whole ETL process, we need to make sure:

1: The logfile.log is deleted, so that flume does not send duplicated data

2: All files under /BigData/flume/simple is deleted, so that spark does not send duplicated data 3: All files under chkpt, chkpt1 and chkpt2 folders are deleted so that all spark jobs can run from beginning.

Then ,we can first create two console line listener to the two kafka topic we created, and begin idle, active and PostgreSQL spark jobs. Since no data has been sent to HDFS, they will not do anything.

Then, we need to run the script to generate some logfile first, as if no logfile.log exists when we run flume, we will have an error.

Last, we can invoke the flume agent. It will keep sending data to the HDFS, which triggers three spark streaming jobs to run, hence completing the whole process.