Name: Vikas Pandey

Neptun ID: I11Z6U

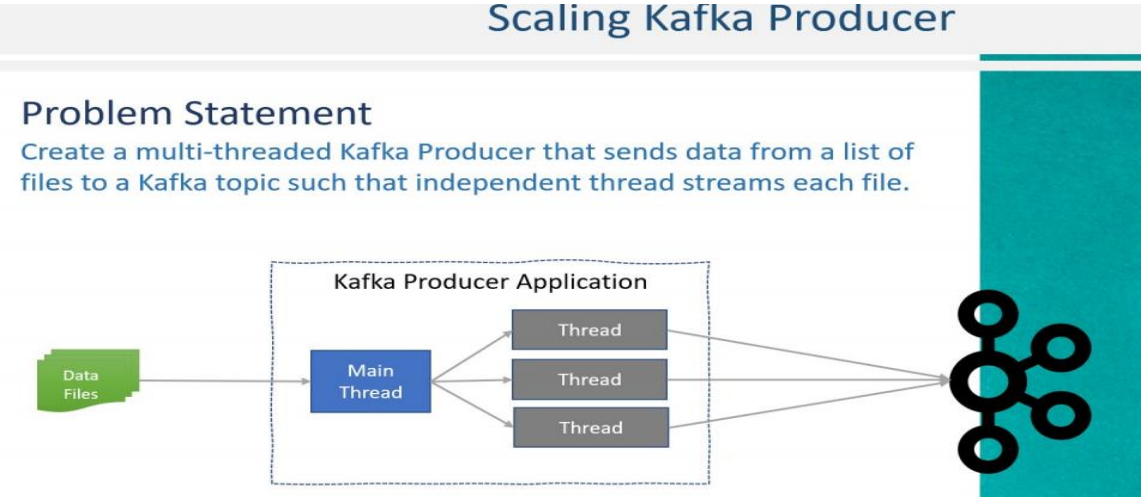
In data acquisition there were three steps:

1. Collecting the data from different sources
2. Using Kafka to read the data file into Kafka cluster.
3. Integrated Kafka cluster to Spark.

Here step 2 and 3 are described:

The task is to read Data from CSV file, collected from different sources, one of them is Example: Histdata.com.

We will be using Kafka to take this data and create a producer to read the data from the csv file.



The ouput will be printed in the Kafka consumer console, We wanted to integrate it with spark, but were unsuccessfull in doing so. The documentation we followed given here.

Step 1:

Installing and configuring Kafka and spark. My focus will be on Kafka.

1. Download Kafka scala latest version from here, you will be directed here, where you cab get tgz file.
2. Now unzip it and set KAFKA_HOME as your Enivronment Variable, also specify the path for Kafka bin directory.
3. You will be needing the Java JDK and Maven as we will be using that as well. For the IDE we will be using IntelliJ.
4. After installing all of these we are good to go. Any problem? Refer here.

Step 2:

Now let us go ahead and work on our Kafka project. We learn about scaling the application (Kafka Producer).

1. We can send data from multiple CSV file, here event and it is the Kafka broker who recieves the messages and acknowedge the successful recipt of the message.

   So, if you are send 10000 of messages you can increase the number of Kafka brokers in the Kafka cluster.

   As single Kafka broker can handle 100 or maybe 1000 of message per second. Meanwhile you can send more messages to the Kafka producer. This provides Linear scalibility.

2. Here we will be using Multi-threaded Kafka producer.

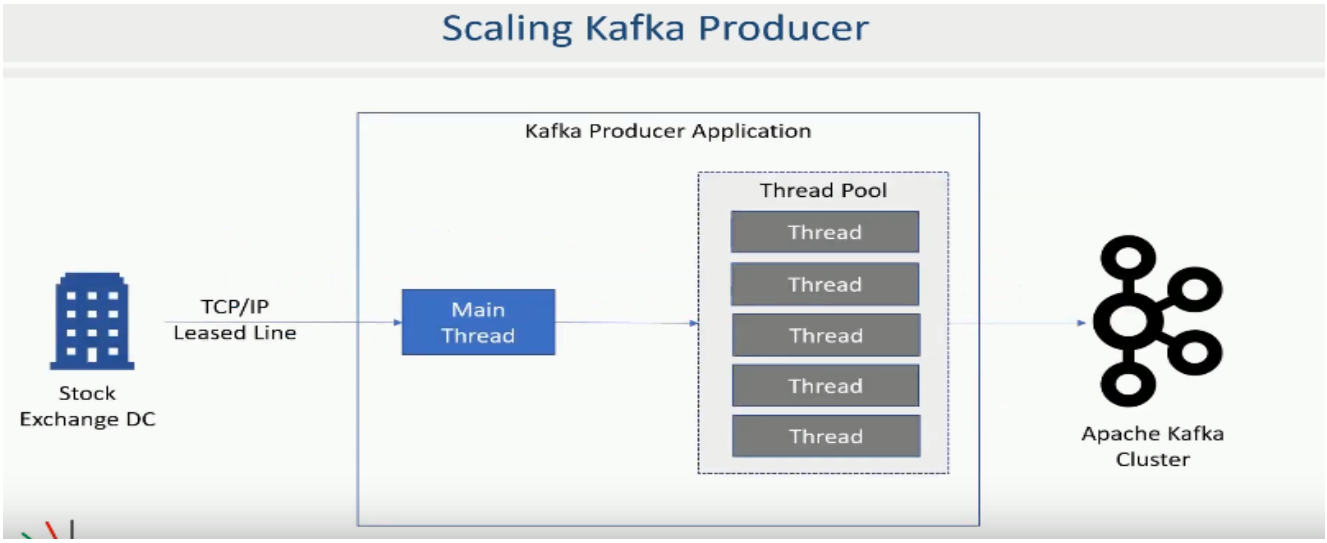
   a. The application recieves the tick by tick data packets from stock exchange over a TCP/IP socket. The packets arrive at high frequency we create a multi threaded data handler.

   b. The main thread listens to the socket and read the data packet as they arrives and immidiately hands over the data packet to the tread pool to be passed on to Kafka cluster.

   c. The amin thread then goes back to reading the data again.

   d. The threads in the thread pool are responsible for uncompressing the data packets and reading the content in the packets, validating the message and sending it to kafka broker.

   e. So if we are passing three data files it must create 3 threads for each data file. Each thread is responsible for reading the records from each file and sending it to Kafka broker. Here we don't want to create multiple producer instances.



Step 3:

1. In this step we will set up the pom.xml file with all the dependencies.

```
    <build>
        <plugins>
            <!-- Maven Compiler Plugin-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- Apache Kafka Clients-->
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>${kafka.version}</version>
        </dependency>

        <!-- Apache Log4J2 binding for SLF4J -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>2.11.0</version>
        </dependency>

    </dependencies>
```

2. The Log4J for logging results.

```
<Configuration status="ERROR">
    <Appenders>
        <Console name="stdout" target="SYSTEM_OUT">
            <PatternLayout pattern="[%d] (%c) %p - %m %n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="stdout"/>
        </Root>
        <Logger name="org.apache.kafka.clients" level="warn" additivity="false">
            <AppenderRef ref="stdout"/>
        </Logger>
        <Logger name="guru.learningjournal.kafka.examples" level="trace"
```

```
additivity="false">
            <AppenderRef ref="stdout"/>
        </Logger>
    </Loggers>
</Configuration>
```

3. Now let us set up the Kafka scripts so the we don't have to run the cmd prompt again and again and we can use the IDE for all of the steps.

   Here are the codes:

```
Start zookeeper: zookeeper-server-start.bat %KAFKA_HOME%\config\zookeeper.properties
Kafka Server 0: kafka-server-start.bat %KAFKA_HOME%\config\server-0.properties
```

```
Kafka Server 1: kafka-server-start.bat %KAFKA_HOME%\config\server-1.properties
```

```
Kafka Server 2: kafka-server-start.bat %KAFKA_HOME%\config\server-2.properties
```

```
The Kafka Topic: kafka-topics.bat --create --zookeeper localhost:2181 --topic
the-finance-data-topic --partitions 5 --replication-factor 3
```

And finally Kafka consumer to print the output in console;

```
Kafka Consumer: kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
the-finance-data-topic --from-beginning
```

4. Now we need to set up the App Config:

```
class AppConfigs {
    final static String applicationID = "Multi-Threaded-Producer";
    final static String topicName = "the-finance-3-data-topic";
    final static String kafkaConfigFileLocation = "kafka.properties";
    final static String[] eventFiles = {"data/Raw_data.csv", "data/Raw_data.csv"};
}
```

5. Now we should keep some producer level config outside the source code, here broker coordinates. This arrangement allows us to deploy the application and connect to any Kafka cluster by changing the connection details in the properties file.

```
bootstrap.servers=localhost:9092,localhost:9093
```

6. Now let us create the Dispacter class which run the runnable interface, this allows us to use the instance of this class as a separate thread.

   a. This takes Kafka producer instance, Topic Name and the File Location.

```
public class Dispatcher implements Runnable {
    private static final Logger logger = LogManager.getLogger();
    private String fileLocation;
    private String topicName;
    private KafkaProducer<Integer, String> producer;

    Dispatcher(KafkaProducer<Integer, String> producer, String topicName, String
fileLocation) {
        this.producer = producer;
        this.topicName = topicName;
        this.fileLocation = fileLocation;
    }

    @Override
    public void run() {
        logger.info("Start Processing " + fileLocation);
        File file = new File(fileLocation);
        int counter = 0;

        try (Scanner scanner = new Scanner(file)) {
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                producer.send(new ProducerRecord<>(topicName, null, line));
                counter++;
            }
            logger.info("Finished Sending " + counter + " messages from " + fileLocation);
        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }

    }
}
```

7. Another class, DispacterDemo class allows us to create thread and start them.
    a. This is a main thread, which will produce a single instance of Kafka producer.
    b. It then creates two threa using the runnable dispacher and pass the producer instance, along with the Topic name and the File Location to the Dispacher thread.
    c. Each Dispacher thread will send all the line in the file to a given topic.
    d. Main thread waits for all the thread to be processed and close the Producer and Terminate.

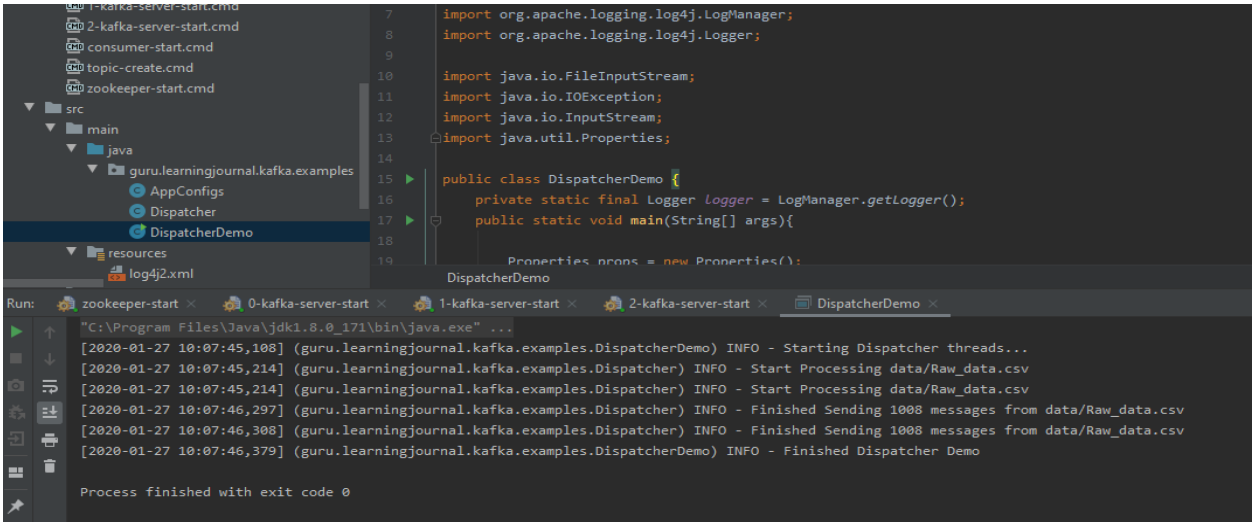
```
public class DispatcherDemo {
    private static final Logger logger = LogManager.getLogger();
    public static void main(String[] args){

        Properties props = new Properties();
        try{
            InputStream inputStream = new
FileInputStream(AppConfigs.kafkaConfigFileLocation);
            props.load(inputStream);
```

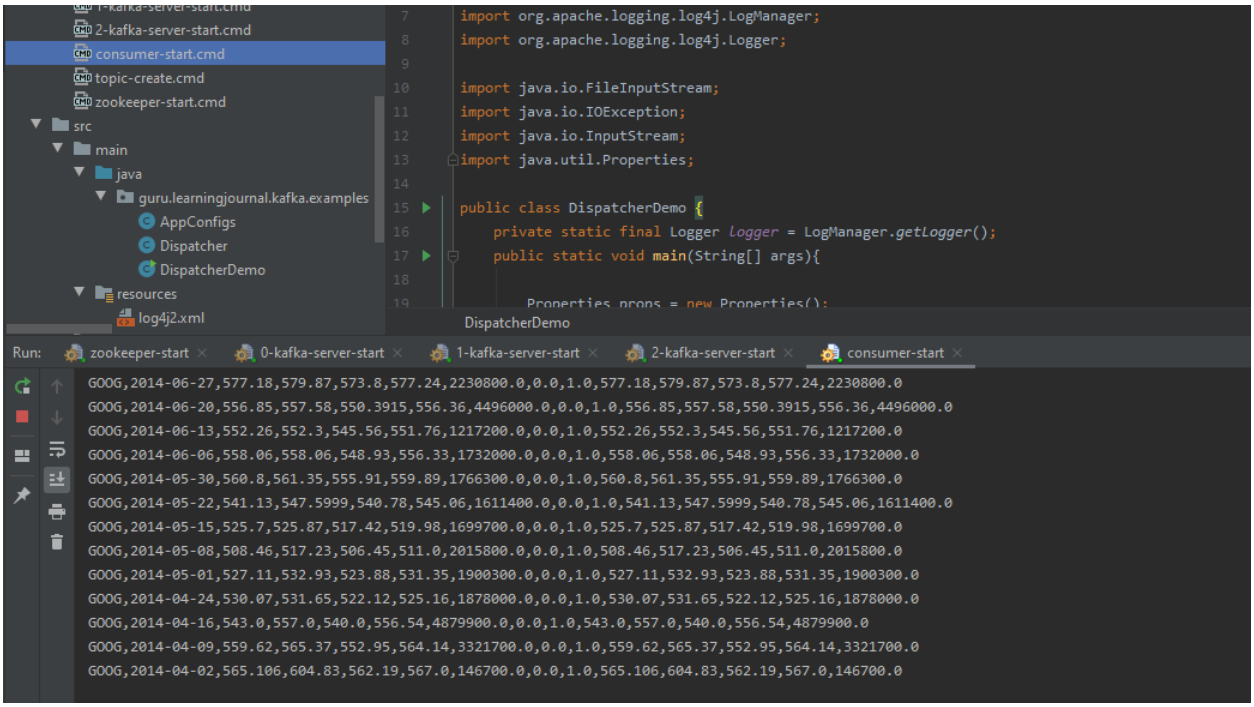
```
            props.put(ProducerConfig.CLIENT_ID_CONFIG,AppConfigs.applicationID);
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        }catch (IOException e){
            throw new RuntimeException(e);
        }

        KafkaProducer<Integer,String> producer = new KafkaProducer<>(props);
        Thread[] dispatchers = new Thread[AppConfigs.eventFiles.length];
        logger.info("Starting Dispatcher threads...");
        for(int i=0;i<AppConfigs.eventFiles.length;i++){
            dispatchers[i]=new Thread(new
Dispatcher(producer,AppConfigs.topicName,AppConfigs.eventFiles[i]));
            dispatchers[i].start();
        }

        try {
            for (Thread t : dispatchers) t.join();
        }catch (InterruptedException e){
            logger.error("Main Thread Interrupted");
        }finally {
            producer.close();
            logger.info("Finished Dispatcher Demo");
        }
    }
}
```

Step 4:

Run the Application:

1. Start the zookeeper script
2. Start all Kafka brokers, server 0,1,2.
3. Create a topic.
4. Execute the producer application.
5. Excute the Kafka consumer script to see the output of the console.





The step 3: Integrated Kafka cluster to Spark, was not achieved in this project, given time till 31st Jan, that can be done as well. 😊