

Final Project for CS9223

Parallel and Real Time Computation of Statistics for Time-series Data Streams Using Stock Quotes as Example

Abstract

This course project is intended to be the beginning of research on the topic of Real-Time Big Data Analytics by using emerging technologies such as Spark/Hadoop, Kafka, Cassandra etc.

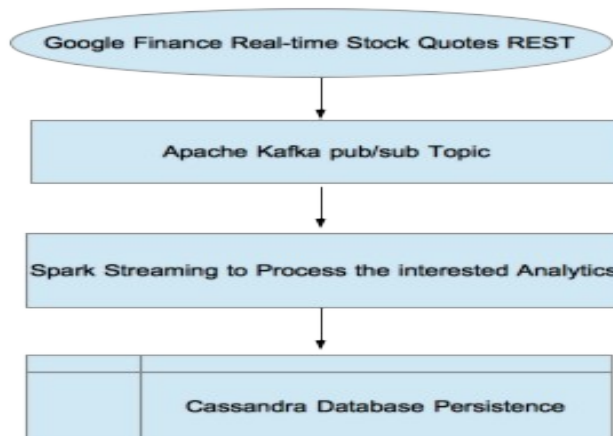
Real-time stream process with Apache Kafka as the backbone of data collection can handle massive, organic data growth via the dynamic addition of streaming sources such as mobile devices, web servers, system logs and of course the stock quotes from varied sources. Kafka can also help capture data in real-time and enable the proactive analysis of data through Spark Streaming.

In addition, Spark Streaming enables the creation of real-time complex event processing architectures with Kafka which is a real-time, fault-tolerant and highly scalable data pipeline for data stream.

Apache Cassandra is a NOSQL database platform particularly suited for these types of Big Data Analytics, its data model is an excellent fit for handling data in sequence regardless of data type or size.

In a short word, the project is going to demonstrate how to integrate Apache Spark, Spark Streaming, Apache Cassandra with the Spark Cassandra Connector and Apache Kafka in general and more specifically for time series data.

Architecture



As illustrated above, the stock quotes events are from google finance, the chosen format of quotes data is json as below:

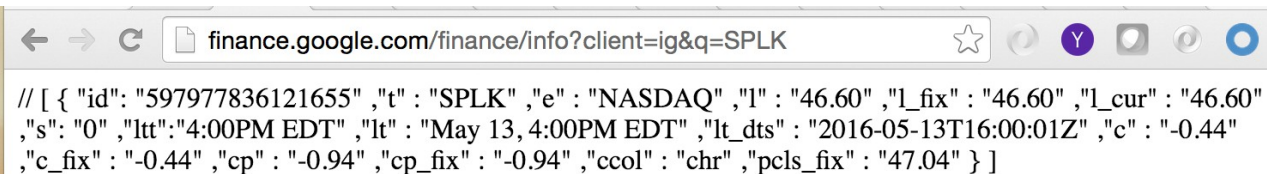
```
{
```

```

    "id": "597977836121655",
    "t": "SPLK",
    "e": "NASDAQ",
    "l": "46.60",
    "l_fix": "46.60",
    "l_cur": "46.60",
    "s": "0",
    "lts": "4:00PM EDT",
    "lt": "May 13, 4:00PM EDT",
    "lt_dts": "2016-05-13T16:00:01Z",
    "c": "-0.44",
    "c_fix": "-0.44",
    "cp": "-0.94",
    "cp_fix": "-0.94",
    "ccol": "chr",
    "pcls_fix": "47.04"
}

```

The stock quotes data is accessible through the RESTful API of google finance:



```

// [ { "id": "597977836121655", "t": "SPLK", "e": "NASDAQ", "l": "46.60", "l_fix": "46.60", "l_cur": "46.60",
"s": "0", "lts": "4:00PM EDT", "lt": "May 13, 4:00PM EDT", "lt_dts": "2016-05-13T16:00:01Z", "c": "-0.44",
"c_fix": "-0.44", "cp": "-0.94", "cp_fix": "-0.94", "ccol": "chr", "pcls_fix": "47.04" } ]

```

Once got the quotes data periodic, I publish them to a topic “stockquote” of Kafka. Spark Streaming now can be configured to consume topics from Kafka and create corresponding Kafka DStreams. Each DStream batches incoming messages into an abstraction called RDD, which is an immutable collection of incoming messages. Each RDD is a micro-batch of the messages and the micro-batching window is configurable to serve the interests of applications.

The data of post processing will be persisted in Cassandra. We may build high performance visualization on top of the data in Cassandra that provides fast and efficient access pattern due to minimal disk seeks if retrieving data by row key and by range since data in Cassandra is sorted and written sequentially to disk.

Design and Program

Run Kafka and topic creation

I used mostly default ports and configuration in this project.

```

$KAFKA_HOME/bin/zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties &
$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties &
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic stockquote
$KAFKA_HOME/bin/kafka-topics.sh --list --zookeeper localhost:2181

```

Quotes publish

I established a Timer to collect data every 10 seconds periodic as **bold code**, and publish them to Kafka.

```
public class StockInput {
    static final String QUOTE_TOPIC = "stockquote";
    static final String ERROR_TOPIC = "errorquote";

    public static void main(String[] args) {
        java.util.Timer t = new java.util.Timer();
        t.schedule(new TimerTask() {
            @Override
            public void run() {
                publish(args);
            }
        }, 5000, 10000);
    }

    // get stock quotes from a list of stock symbols by accessing google
    finance.
    // the quotes data are in json form, which is good for us to publish
    messages to kafka topic.
    public static void publish(String[] quotes) {
        Properties properties = new Properties();
        properties.put("metadata.broker.list", "localhost:9092");
        properties.put("serializer.class", "kafka.serializer.StringEncoder");
        ProducerConfig producerConfig = new ProducerConfig(properties);
        kafka.javaapi.producer.Producer<String, String> producer = new
        kafka.javaapi.producer.Producer<String, String>(
            producerConfig);

        for (String q : quotes) {
            try {
                String msg = getJsonQuote(q);
                KeyedMessage<String, String> message = new
                KeyedMessage<String, String>(QUOTE_TOPIC, msg);
                producer.send(message);
            } catch (IOException e) {
                KeyedMessage<String, String> error = new
                KeyedMessage<String, String>(ERROR_TOPIC,
                q.concat(":".concat(e.getMessage())));
                producer.send(error);
            }
        }

        producer.close();
    }

    public static String getJsonQuote(String stock) throws IOException {
        String googleResponse =
        HttpUtil.get("http://finance.google.com/finance/info?client=ig&q=" + stock);
        String token[] = StringUtils.split(googleResponse, "/");
        String token1[] = StringUtils.split(token[1], "[");
        String token2[] = StringUtils.split(token1[1], "]");

        return token2[0];
    }
}
```

Run Spark

The way to run a spark application is through spark-submit command. Compile the main class as stockquote.jar and “spark-submit” it with proper arguments. Very important part is to set SPARK-CLASSPATH with all additional jar files needed. The script is the attached file “run-spark.sh”.

```
export SPARK_CLASSPATH=$SPARK_HOME/external/kafka/target/spark-streaming-
kafka_2.10-1.6.1.jar:\
$KAFKA_HOME/libs/kafka_2.10-0.9.0.1.jar:\
...
$SPARK_HOME/bin/spark-submit --class $1 \
--master local[8] /home/billtsay/CS9223/bin/stockquote.jar \
$2 $3 $4 $5
```

The script execution is as below:

```
^Cbilltsay@newyork:~/CS9223/bin$
billtsay@newyork:~/CS9223/bin$ ./run-spark.sh nyu.cs9223.StockQuotesProgram localhost:2181 StockQuotes stockquote 3
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/billtsay/CS9223/kafka_2.10-0.9.0.1/libs/slf4j-log4j12-1.7.6.jar!/org/slf4j/impl/Static
LoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/billtsay/CS9223/spark-1.6.1/assembly/target/scala-2.10/spark-assembly-1.6.1-hadoop2.4
0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

Spark Streaming Program

The steps to retrieve quotes data from Kafka, calculate them and then persist the calculated data to Cassandra are as below **bold comments** in the class StockQuotesProgram.

```
public final class StockQuotesProgram {
    private StockQuotesProgram() {
    }
    public static void main(String[] args) {
        if (args.length < 4) {
            System.err.println("Usage: StockQuotesProgram <zkQuorum> <group>
<topics> <numThreads>");
            System.exit(1);
        }
        StreamingExamples.setStreamingLogLevels();
        SparkConf sparkConf = new
SparkConf().setAppName("StockQuotesProgram");

        // 1. collect data for this period Durations.minutes(#)
        JavaStreamingContext jssc = new JavaStreamingContext(sparkConf,
Durations.minutes(2));

        int numThreads = Integer.parseInt(args[3]);
```

```

// 2. connect to topics of kafka and collect data.
Map<String, Integer> topicMap = new HashMap<String, Integer>();
String[] topics = args[2].split(",");
for (String topic : topics) {
    topicMap.put(topic, numThreads);
}

JavaPairReceiverInputDStream<String, String> messages =
KafkaUtils.createStream(jssc, args[0], args[1],
    topicMap);

// 3. parse and convert the json qutoes data into java object
StockQuote for the convenience of calculation.
JavaDStream<StockQuote> quotes = messages.map(new
Function<Tuple2<String, String>, StockQuote>() {
    @Override
    public StockQuote call(Tuple2<String, String> tuple2) {
        Gson gson = new GsonBuilder().create();
        return gson.fromJson(tuple2._2(), StockQuote.class);
    }
});

// 4. sum up the stock price for each stock during this period.
JavaPairDStream<String, Double> price = quotes.mapToPair(new
PairFunction<StockQuote, String, Double>() {
    @Override
    public Tuple2<String, Double> call(StockQuote t) throws Exception
{
        return new Tuple2<String, Double>(t.getSymbol(),
t.getPrice());
    }
}).reduceByKey(new Function2<Double, Double, Double>() {
    @Override
    public Double call(Double i1, Double i2) {
        return i1 + i2;
    }
});

// 5. sum up the count of quotes for each stock during the same
period.

```

```

        JavaPairDStream<String, Integer> count = quotes.mapToPair(new
PairFunction<StockQuote, String, Integer>() {
            @Override
            public Tuple2<String, Integer> call(StockQuote t) throws
Exception {
                return new Tuple2<String, Integer>(t.getSymbol(), 1);
            }
        }).reduceByKey(new Function2<Integer, Integer, Integer>() {
            @Override
            public Integer call(Integer i1, Integer i2) {
                return i1 + i2;
            }
        });

        // 6. join the two streams price and count for each stock.
        JavaPairDStream<String, Tuple2<Double, Integer>> quote =
price.join(count);

        // 7. convert the average price and post current date into java object
AvgQuotes.
        JavaDStream<AvgQuotes> qs = quote.map(new Function<Tuple2<String,
Tuple2<Double, Integer>>, AvgQuotes>() {
            @Override
            public AvgQuotes call(Tuple2<String, Tuple2<Double, Integer>> v)
throws Exception {
                Double p = v._2()._1() / v._2()._2();
                Date date = new Date(); // Right now
                return AvgQuotes.newInstance(v._1(), p, date);
            }
        });

        // 8. persist each java object AvgQuotes into cassandra table
cs9223.stockquotes.
        qs.foreachRDD(new VoidFunction<JavaRDD<AvgQuotes>>() {
            @Override
            public void call(JavaRDD<AvgQuotes> rdd) throws Exception {
                javaFunctions(rdd).writerBuilder("cs9223", "stockquotes",
mapToRow(AvgQuotes.class)).saveToCassandra();
            }
        });
        qs.print();
        jssc.start();

```

```
jssc.awaitTermination();  
  
}  
  
}
```

Creation of stockquotes table in Cassandra

```
billtsay@newyork: ~/CS9223/apache-cassandra-3.5  
cqlsh> source '/home/billtsay/CS9223/bin/stockquotes.cql';  
cqlsh> select * from cs9223.stockquotes;  
  
symbol | eventtime | price  
-----+-----+-----  
APPL | 2016-05-15 06:49:53.604000+0000 | 24.5  
YHOO | 2016-05-15 06:49:53.604000+0000 | 33.4  
SPLK | 2016-05-15 06:49:53.604000+0000 | 24.6  
  
(3 rows)
```

The script “stockquotes.cql” is as below:

```
/*  
Schema for storing average stock price data.  
*/  
  
DROP KEYSPACE IF EXISTS cs9223;  
CREATE KEYSPACE cs9223 WITH REPLICATION = { 'class' : 'SimpleStrategy',  
'replication_factor' : 1 };  
  
use cs9223;  
  
CREATE TABLE stockquotes (  
    symbol text,           // Stock symbol  
    price double,          // average price  
    eventtime timestamp,   // event time  
    PRIMARY KEY (symbol, eventtime)  
);
```

what it did was to create a keyspace “cs9223” and a table “stockquotes” with interested columns we need. I pre-filled three records to verify if my java code can connect to it and persist in the above figure.

Final Results

The results in Cassandra is as below, we can see the average quotes are populated every 2 minutes.

The average values are “all the same”, that is because I collected data during weekends when I wrote this report so there is no changes due to market close. I did verify the data during weekdays that I can collect real-time data while market is open. **I can provide the updates if requested when I can collect data during weekdays while market is open.**

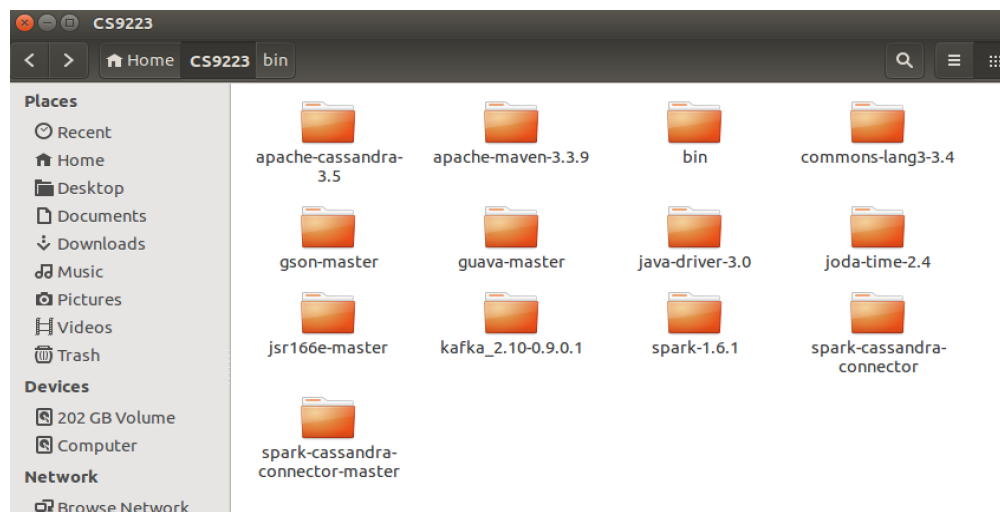
```
billtsay@newyork: ~/CS9223/apache-cassandra-3.5
SPLK | 2016-05-15 07:58:00.133000+0000 | 46.6
SPLK | 2016-05-15 08:00:00.155000+0000 | 46.6
SPLK | 2016-05-15 08:02:00.132000+0000 | 46.6
SPLK | 2016-05-15 08:04:00.128000+0000 | 46.6
SPLK | 2016-05-15 08:06:00.143000+0000 | 46.6
SPLK | 2016-05-15 08:08:00.169000+0000 | 46.6
SPLK | 2016-05-15 08:10:00.114000+0000 | 46.6
SPLK | 2016-05-15 08:12:00.112000+0000 | 46.6
SPLK | 2016-05-15 08:14:00.113000+0000 | 46.6
SPLK | 2016-05-15 08:16:00.114000+0000 | 46.6
SPLK | 2016-05-15 08:18:00.101000+0000 | 46.6
SPLK | 2016-05-15 08:20:00.111000+0000 | 46.6
SPLK | 2016-05-15 08:22:00.110000+0000 | 46.6
SPLK | 2016-05-15 08:24:00.132000+0000 | 46.6
SPLK | 2016-05-15 08:26:00.115000+0000 | 46.6
SPLK | 2016-05-15 08:28:00.129000+0000 | 46.6
SPLK | 2016-05-15 08:30:00.112000+0000 | 46.6
SPLK | 2016-05-15 08:32:00.101000+0000 | 46.6
SPLK | 2016-05-15 08:34:00.118000+0000 | 46.6
SPLK | 2016-05-15 08:36:00.128000+0000 | 46.6
SPLK | 2016-05-15 08:38:00.133000+0000 | 46.6
SPLK | 2016-05-15 08:40:00.112000+0000 | 46.6
SPLK | 2016-05-15 08:42:00.128000+0000 | 46.6

---MORE---
symbol | eventtime | price
-----|-----|-----
SPLK | 2016-05-15 08:44:00.089000+0000 | 46.6
SPLK | 2016-05-15 08:46:00.136000+0000 | 46.6
SPLK | 2016-05-15 08:48:00.150000+0000 | 46.6

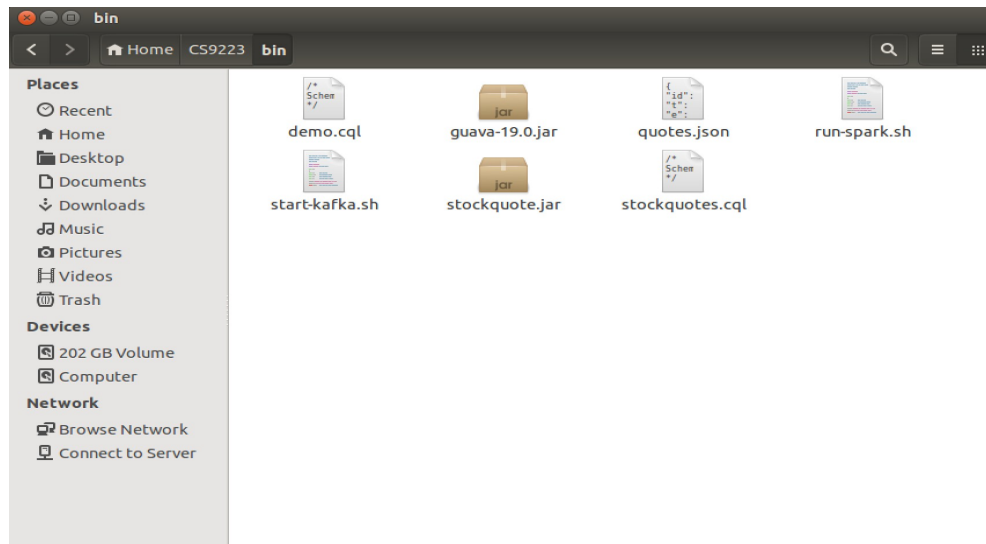
(203 rows)
cqlsh>
```

Project Folder

Main Folder



and Subfolder bin



Summary

This course project is to prove the concept of using Kafka to publish real time data from data sources such as stock quotes. I set up an interested real-time period of 10 seconds and it is adjustable for the application needs. My purpose is to do real time analytics of calculating average price in 2 minutes for each stock through Spark Streaming. Of course, we can do any complicated statistics calculation, here I just did a simple case. Then I persisted the calculated data into Cassandra.

Reference

1. Spark Cassandra Connector I used is from <https://github.com/datastax/spark-cassandra-connector>.
2. Cassandra Java Driver I used is from <https://github.com/datastax/java-driver>.
3. The Cassandra database is downloaded from <http://cassandra.apache.org/download/>.
4. The Apache Kafka is downloaded from <http://kafka.apache.org/downloads.html>.
5. The Apache Spark is downloaded from <http://spark.apache.org/downloads.html>.
6. The Scala version is 2.10 as the latest 2.11 is not popular such that I am unable to integrate all of them together to make all things running.
7. I also downloaded quite a lot of shared jars that are described in the attached “run-spark.sh” file. Those jars are needed to execute the spark streaming program I wrote.