This is the third article of the "Big Data Processing with Apache Spark" series. Please see also: Part 1: Introduction, Part 2: Spark SQL, Part 4: Spark Machine Learning, Part 5: Spark ML Data Pipelines, Part 6: Graph Data Analytics with Spark GraphX.

Introduction

In the first two articles in "Big Data Processing with Apache Spark" series, we looked at what Apache Spark framework is (Part 1) and SQL interface to access data using Spark SQL library (Part 2).

These solutions are based on processing static data in a batch mode, for example as an hourly or daily job. But what about real-time data streams that need to be processed on the fly to perform analytics and create insights for data driven business decision making?

With streaming data processing, computing is done in real-time as data arrives rather than as a batch. Real-time data processing and analytics is becoming a critical component of the big data strategy for most organizations.

In this article, we'll learn about real-time data analytics using one of the libraries from Apache Spark, called Spark Streaming.

We'll look at a web server log analytics use case to show how Spark Streaming can help with running analytics on data streams that are generated in a continuous manner.

Streaming Data Analytics

Streaming data is basically a continuous group of data records generated from sources like sensors, server traffic and online searches. Some of the examples of streaming data are user activity on websites, monitoring data, server logs, and other event data.

Streaming data processing applications help with live dashboards, real-time online recommendations, and instant fraud detection.

If we are building applications to collect, process and analyze streaming data in real time, we need to take different design considerations into account than when we are working on applications used to process the static batch data.

There are different streaming data processing frameworks as listed below:

- Apache Samza
- Storm
- Spark Streaming

We'll focus on Spark Streaming in this article.

Spark Streaming

Spark Streaming is an extension of core Spark API. Spark Streaming makes it easy to build fault-tolerant processing of real-time data streams.

Figure 1 below shows how Spark Streaming fits into overall Apache Spark ecosystem.

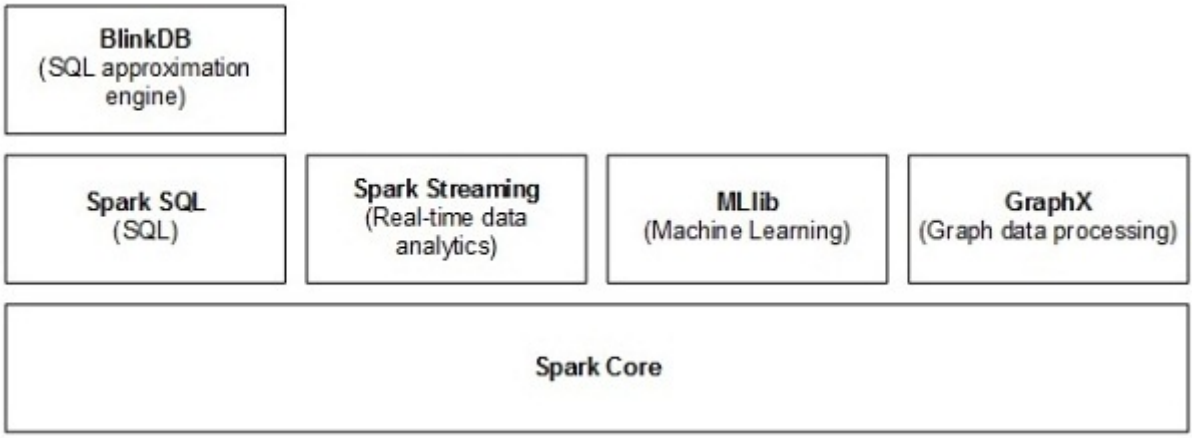

Figure 1. Spark Ecosystem with Spark Streaming Library

The way Spark Streaming works is it divides the live stream of data into batches (called microbatches) of a pre-defined interval (N seconds) and then treats each batch of data as Resilient Distributed Datasets (RDDs). Then we can process these RDDs using the operations like map, reduce, reduceByKey, join and window. The results of these RDD operations are returned in batches. We usually store these results into a data store for further analytics and to generate reports and dashboards or sending event based alerts.

It's important to decide the time interval for Spark Streaming, based on your use case and data processing requirements. It the value of N is too low, then the micro-batches will not have enough data to give meaningful results during the analysis.

Compared to Spark Streaming, other stream processing frameworks process the data streams per each event rather than as a micro-batch. With micro-batch approach, we can use other Spark libraries (like Core, Machine Learning etc) with Spark Streaming API in the same application.

Streaming data can come from many different sources. Some of these data sources include the following:

- Kafka
- Flume
- Twitter
- ZeroMQ
- Amazon's Kinesis
- TCP sockets

Another advantage of using a big data processing framework like Apache Spark is that we can combine batch processing and streaming processing in the same system. We can also apply Spark's machine learning and graph processing algorithms on data streams. We'll discuss Machine Learning and Graph Processing libraries, called MLlib and GraphX respectively, in future articles in this series.

Spark Streaming architecture is shown in Figure 2 below.

(Click on the image to enlarge it)

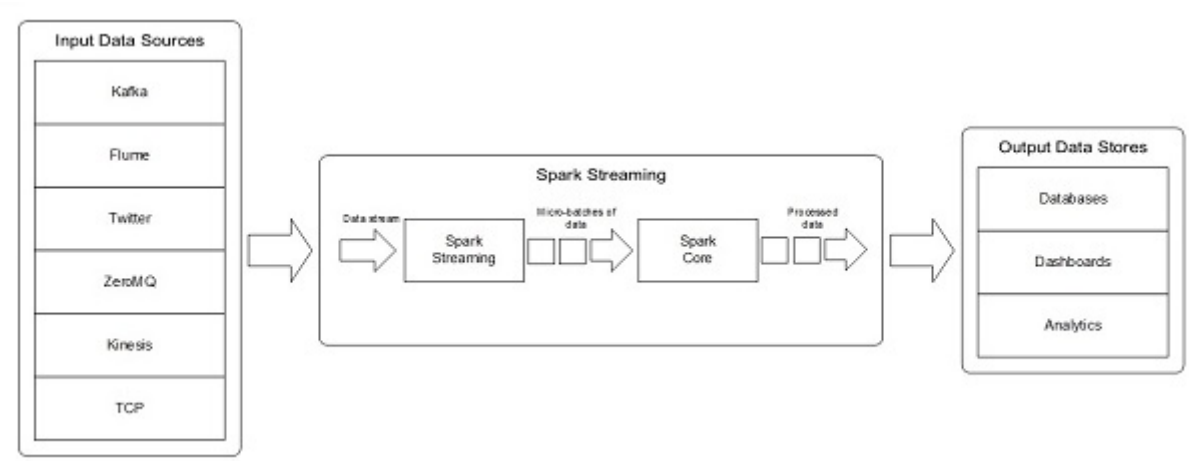

Figure 2. How Spark Streaming works

Spark Streaming Use Cases

Spark Streaming is becoming the platform of choice to implement data processing and analytics solutions for real-time data received from Internet of Things (IoT) and sensors. It is used in a variety of use cases and business applications.

Some of the most interesting use cases of Spark Streaming include the following:

- Uber, the company behind ride sharing service, uses Spark Streaming in their continuous Streaming ETL pipeline to collect terabytes of event data every day from their mobile users for real-time telemetry analytics.
- Pinterest, the company behind the visual bookmarking tool, uses Spark Streaming, MemSQL and Apache Kafka technologies to provide insight into how their users are engaging with Pins across the globe in real-time.
- Netflix uses Kafka and Spark Streaming to build a real-time online movie recommendation and data monitoring solution that processes billions of events received per day from different data sources.

Other real world examples of Spark Streaming include:

- Supply chain analytics
- Real-time security intelligence operations to find threats
- Ad auction platform
- Real-time video analytics to help with personalized, interactive experiences to the viewers

Let's take a look at Spark Streaming architecture and API methods. To write Spark Streaming programs, there are two components we need to know about: DStream and StreamingContext.

DStream

DStream (short for Discretized Stream) is the basic abstraction in Spark Streaming and represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying operations on other DStreams. Internally, a DStream is represented as a sequence of RDD objects.

Similar to the transformation and action operations on RDDs, DStreams support the following operations:

- map
- flatMap
- filter
- count
- reduce
- countByValue
- reduceByKey
- join
- updateStateByKey

Streaming Context

Similar to SparkContext in Spark, StreamingContext is the main entry point for all streaming functionality.

StreamingContext has built-in methods for receiving streaming data into Spark Streaming program.

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname and port number. For example, if we are using a tool like netcat to test the Spark Streaming program, we would receive data stream from the machine where netcat is running (e.g. localhost) and port number of 9999.

When the code is executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing is done yet. To start the processing after all the transformations have been setup, we finally call start() method to start the computation and awaitTermination() method to wait for the computation to terminate.

Spark Streaming API

Spark Streaming comes with several API methods that are useful for processing data streams. There are RDD-like operations like `map`, `flatMap`, `filter`, `count`, `reduce`, `groupByKey`, `reduceByKey`, `sortByKey`, and join. It also provides additional API to process the streaming data based on window and stateful operations. These include `window`, `countByWindow`, `reduceByWindow`, `countByValueAndWindow`, `reduceByKeyAndWindow` and `updateStateByKey`.

Spark Streaming library is currently supported in Scala, Java, and Python programming languages. Here are the links to Spark Streaming API in each of these languages.

- Spark Streaming Scala API
- Java API
- Python API

Steps in a Spark Streaming program

Before we discuss the sample application, let's take a look at different steps involved in a typical Spark Streaming program.

- Spark Streaming Context is used for processing the real-time data streams. So, the first step is to initialize the StreamingContext object using two parameters, SparkContext and sliding interval time. Sliding interval sets the update window where we process the data coming in as streams. Once the context is initialized, no new computations can be defined or added to the existing context. Also, only one StreamingContext object can be active at the same time.
- After Spark Streaming context is defined, we specify the input data sources by creating input DStreams. In our sample application, the input data source is a log message generator that uses Apache Kafka distributed database and messaging system. Log generator program creates random log messages to

simulate a web server run-time environment where log messages are continuously generated as various web applications serve the user traffic.
- Define the computations using the Sparking Streaming Transformations API like map and `reduce` to DStreams.
- After streaming computation logic is defined, we can start receiving the data and process it using `start` method in StreamingContext object created earlier.
- Finally, we wait for the streaming data processing to be stopped using the `awaitTermination`method of StreamingContext object.

Sample Application

The sample application we discuss in this article is a server log processing and analytics program. It can be used for real-time monitoring of application server logs and performing data analytics on those logs. These log messages are considered time series data, which is defined as a sequence of data points consisting of successive measurements captured over a specified time interval.

Time series data examples include sensor data, weather information, and click stream data. Time series analysis is about processing the time series data to extract insights that can be used for business decision making. This data can also be used for predictive analytics to predict future values based on historical data.

With a solution like this, we don't need an hourly or daily batch job to process the server logs. Spark Streaming receives continuously generated data, processes it, and computes log statistics to provide insights into the data.

To follow a standard example on analyzing the server logs, we'll use Apache Log Analyzer discussed in Data Bricks Spark Streaming Reference Application as a reference to our sample application. This application already has log message parsing code that we'll reuse in our application. The reference application is an excellent resource to learn more about Spark framework in general and Spark Streaming in particular. For more details on Databricks Spark Reference Application, checkout their website.

Use Case

The use case for the sample application is a web server log analysis and statistics generator. In the sample application, we analyze the web server logs to compute the following statistics for further data analysis and create reports and dashboards:

- Response counts by different HTTP response codes
- Response content size
- IP address of the clients to assess where the highest web traffic is coming from
- Top end point URLs to identify which services are accessed more than others

Unlike the previous two articles in this series, we will use Java instead of Scala for creating the Spark program in this article. We'll also run the program as a stand-alone application instead of running the code from the console window. This is how we would deploy Spark programs in Test and Production environments. Shell console interface (using Scala, Python, or R languages) is for local developer testing only.

Technologies

We will use the following technologies in sample application to demonstrate how Spark Streaming library is used for processing the real time data streams.

Zookeeper

Zookeeper is a centralized service providing reliable distributed coordination for distributed applications. Kafka, the messaging system we use in the sample application, depends on Zookeeper for configuration details across the cluster.

Apache Kafka

Apache Kafka is a real time, fault tolerant, scalable messaging system for moving data in real time. It's a good candidate for use cases like capturing user activity on websites, logs, stock ticker data, and instrumentation data.

Kafka works like a distributed database and is based on a partitioned and replicated low latency commit log. When we post a message to Kafka, it's replicated to different servers in the cluster and at the same time it's also committed to disk.

Apache Kakfa includes client API as well as a data transfer framework called Kafka Connect.

Kafka Clients: Kafka includes Java clients (for both message producers and consumers). We will use the Java producer client API in our sample application.

Kafka Connect: Kafka also includes Kafka Connect, which is a framework for streaming data between Apache Kafka and external data systems to support the data pipelines in organizations. It includes import and export connectors to move data sets into and out of Kafka. Kafka Connect program can run as a standalone process or as a distributed service and supports REST interface to submit the connectors to Kafka Connect cluster using a REST API.

Spark Streaming

We'll use Spark Streaming Java API to receive the data streams, calculate the log statistics, and run queries to answer questions like what are the IP addresses where more web requests are coming from, etc.

Table 1 below shows the technologies and tools and their versions used in the sample applications.

Technology	Version	URL
Zookeeper	3.4.6	https://zookeeper.apache.org/doc/r3.4.6/
Kafka	2.10	http://kafka.apache.org/downloads.html
Spark Streaming	1.4.1	https://spark.apache.org/releases/spark-release-1-4-1.html
JDK	1.7	http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
Maven	3.3.3	http://archive.apache.org/dist/maven/maven-3/3.3.3/

Table 1. Spark streaming sample application technologies and tools

Different architecture components of Spark Streaming sample application are illustrated in Figure 3.

(Click on the image to enlarge it)

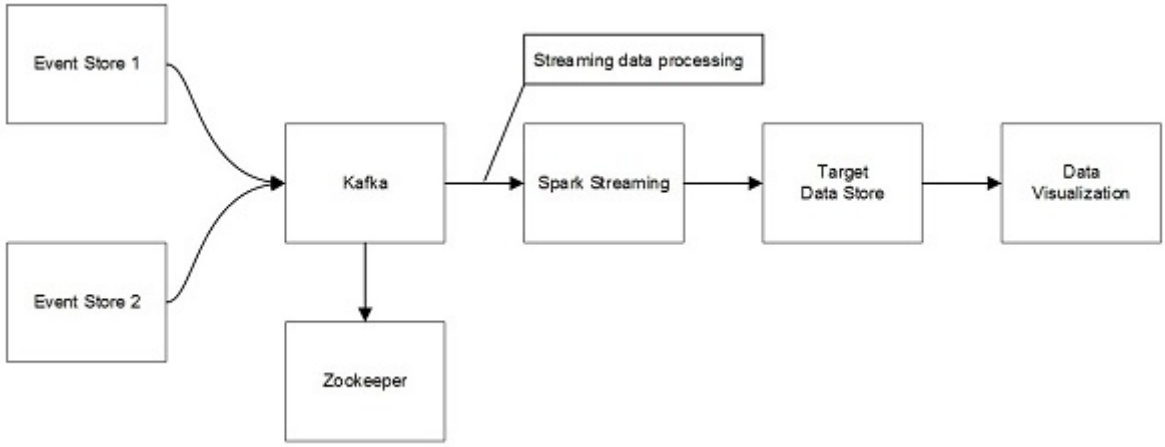

Figure 3. Spark Streaming Sample Application Architecture

Spark Streaming Application Run-time

To setup the Java project locally, you can download Databricks reference application code from Github. Once you get the reference application code, you will need two additional Java classes to run our sample application.

- Log generator (`SparkStreamingKafkaLogGenerator.java`)
- Log analyzer (`SparkStreamingKafkaLogAnalyzer.java`)

These files are provided as a zip file (spark-streaming-kafka-sample-app.zip) on the article website. If you want to run the sample application on your local machine, use the link to download the zip file, extract Java classes and add them to the Java project created in the previous step.

The sample application can be executed on different operating systems. I ran the application in both Windows and Linux (CentOS VM) environments.

Let's look at each component in the application architecture and the steps to execute Sparking Streaming program.

Zookeeper Commands:

I used Zookeeper version 3.4.6 in the sample application. To start the server, set two environment variables, `JAVA_HOME` and `ZOOKEEPER_HOME` to point to JDK and Zookeeper installation directories respectively. Then navigate to Zookeeper home directory and run the following command to start Zookeeper server.

```
bin\zkServer.cmd
```

If you are using a Linux environment, the command is:

```
bin/zkServer.sh start
```

Kafka Server Commands:

Kafka version 2.10-0.9.0.0 was used in the program, which is based on Scala 2.10 version. Scala version you use with Kakfa is very important because if the correct version is not used, you get run-time errors when executing the spark streaming program. Here are the step to start Kafka server instance:

- Open a new command prompt

- Set JAVA_HOME and KAFKA_HOME variables
- Navigate to Kafka home directory
- Run the following command

```
bin\windows\kafka-server-start.bat config\server.properties
```

For Linux environment, the command is as follows:

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

Log Generator Commands:

Next step in our sample application is to run the message log generator.

Log generate creates test log messages with different HTTP response codes (like 200, 401, and 404) with different end point URLs.

Before we run the log generator, we need to create a Topic that we can write the messages to.

Similar to the previous step, open a new command prompt, set JAVA_HOME and KAFKA_HOMEvariables, and navigate to Kafka home directory. Then run the following command first to view the existing topics available in Kafka server.

```
bin\windows\kafka-run-class.bat kafka.admin.TopicCommand --zookeeper localhost:2181 --
list
```

or in Linux:

```
bin/kafka-run-class.sh kafka.admin.TopicCommand --zookeeper localhost:2181 --list
```

We will create a new topic called "spark-streaming-sample-topic" using the following command:

```
bin\windows\kafka-run-class.bat kafka.admin.TopicCommand --zookeeper localhost:2181 --
replication-factor 1 --partitions 1 --create --topic spark-streaming-sample-topic
```

or in Linux:

```
bin/kafka-run-class.sh kafka.admin.TopicCommand --zookeeper localhost:2181 --
replication-factor 1 --partitions 1 --create --topic spark-streaming-sample-topic
```

You can run the list topics command again to see if the new topic has been created correctly.

After the topic has been created, we can run the log generator program. This is done by executing the Java class called SparkStreamingKafkaLogGenerator. Log generator class takes the following four arguments to specify the configuration parameters.

- Group Id: spark-streaming-sample-group
- Topic: spark-streaming-sample-topic
- Number of iterations: 50
- Interval: 1000

Open a new command prompt to run the log generator. We will set three environment variables (JAVA_HOME, MAVEN_HOME, and KAFKA_HOME) for JDK, Maven, and Kakfa directories respectively. Then navigate to sample project root directory (e.g. c:\dev\projects\spark-streaming-kafka-sample-app) and run the following command.

```
mvn exec:java -
Dexec.mainClass=com.sparkstreaming.kafka.example.SparkStreamingKafkaLogGenerator -
```

```
Dexec.args="spark-streaming-sample-groupid spark-streaming-sample-topic 50 1000"
```

Once the log generator program is running, you should see the test log messages created with the debug messages shown on the console. This is only sample code, so the log messages are randomly generated to simulate the continuous flow of data from an event store like a web server.

Figure 4 below shows the screenshot of log message producer and log messages are being generated.

(Click on the image to enlarge it)

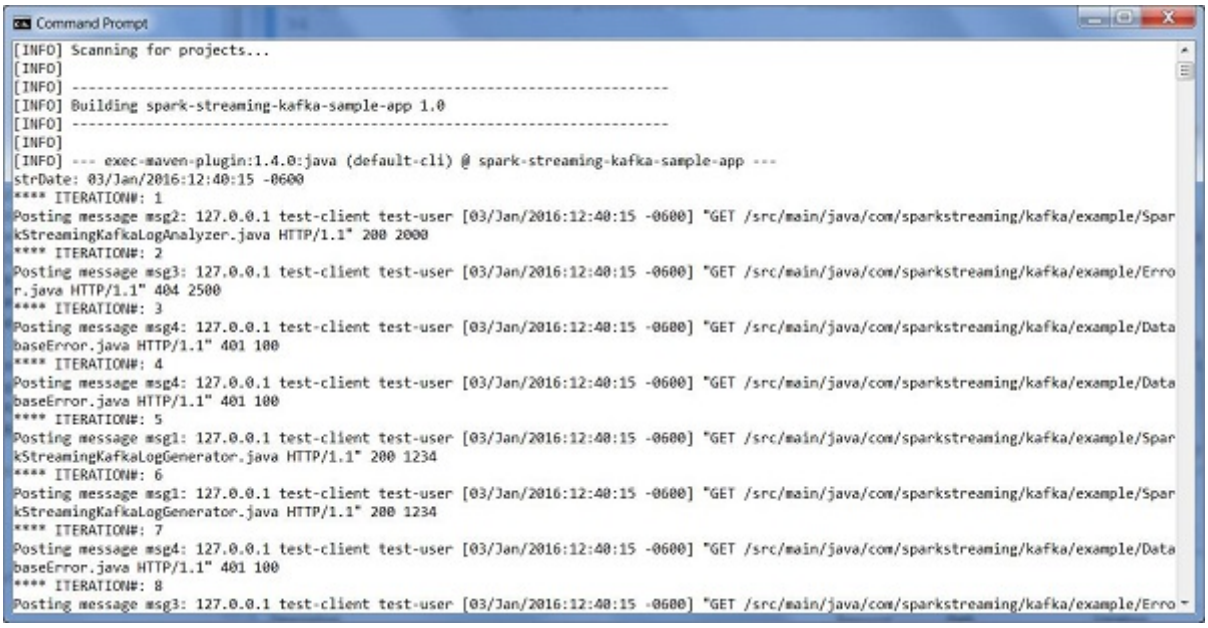

Figure 4. Spark streaming log generator program output

Spark Streaming Commands:

This is the consumer of log messages using Spark Streaming API. We use a Java class called `SparkStreamingKafkaLogAnalyzer` to receive the data streams from Kafka server and process them to create log statistics.

Sparking Streaming processes server log messages and generates cumulative log statistics like web request content size (minimum, maximum, and average), response code counts, IP addresses and the top endpoints.

We create the Spark Context using "local[*]" parameter, which detects the number of cores in the local system and uses them to run the program.

To run the Spark Streaming Java class, you will need the following JAR files in the classpath:

- kafka_2.10-0.9.0.0.jar
- kafka-clients-0.9.0.0.jar
- metrics-core-2.2.0.jar
- spark-streaming-kafka_2.10-1.4.0.jar
- zkclient-0.3.jar

I ran the program from Eclipse IDE after adding the above JAR files to the classpath. Log analysis Spark Streaming program output is shown in Figure 5.

(Click on the image to enlarge it)

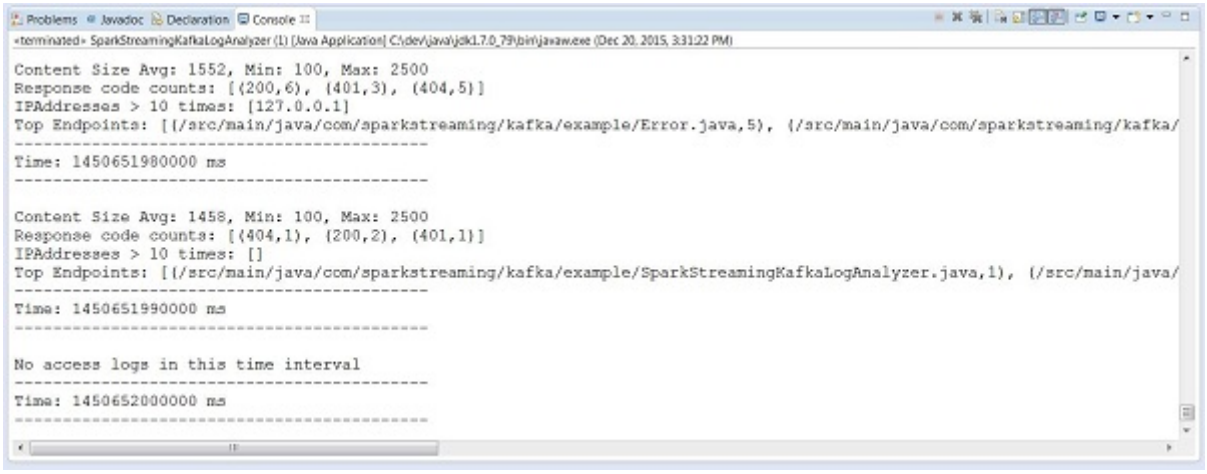

Figure 5. Spark streaming log analytics program output

Visualization of Spark Streaming Applications

When Spark Streaming program is running, we can check the Spark console to view the details of the Spark jobs.

Open a new web browser window and navigate to URL http://localhost:4040 to access the Spark console.

Let's look at some of the graphs showing the Spark Streaming program statistics.

First visualization is the DAG (Direct Acyclic Graph) of a specific job showing the dependency graph of different operations we ran in the program, like `map`, `window`, and `foreachRDD`. Figure 6 below shows the screenshot of this visualization of Spark Streaming job from our sample program.

(Click on the image to enlarge it)

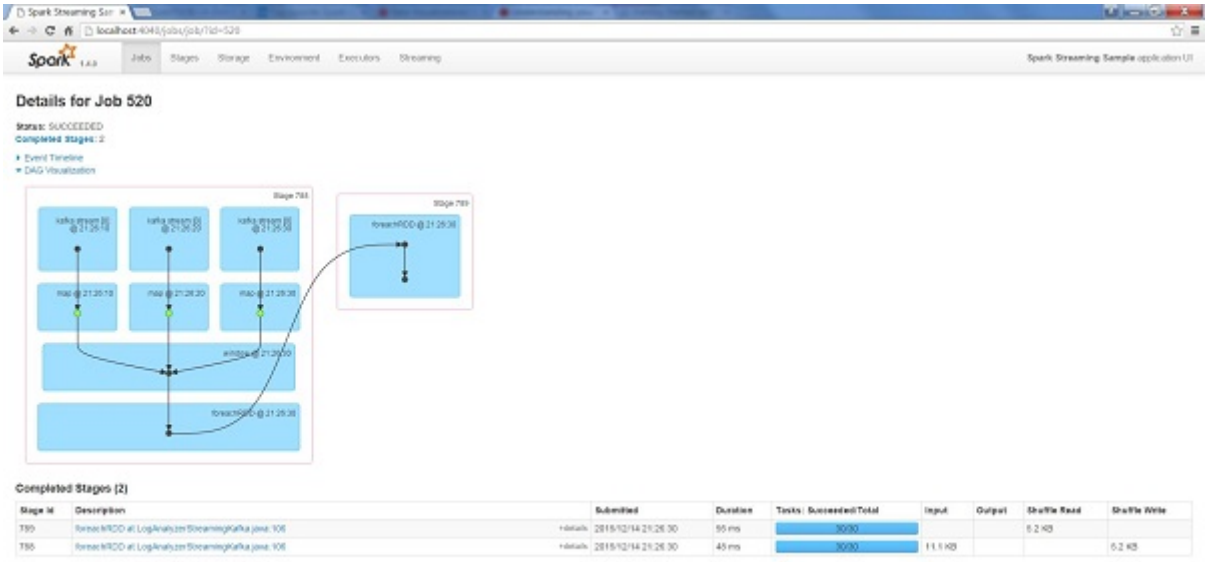

Figure 6. DAG visualization graph of spark streaming job

Next graph we look at is the streaming statistics which include the input rate showing the number of events per second, processing time in milliseconds.

Figure 7 shows these statistics during the execution of Spark Streaming program when the streaming data is not being generated (left section) and when the data stream is being sent to Kafka and processed by Spark Streaming consumer (right section).

(Click on the image to enlarge it)

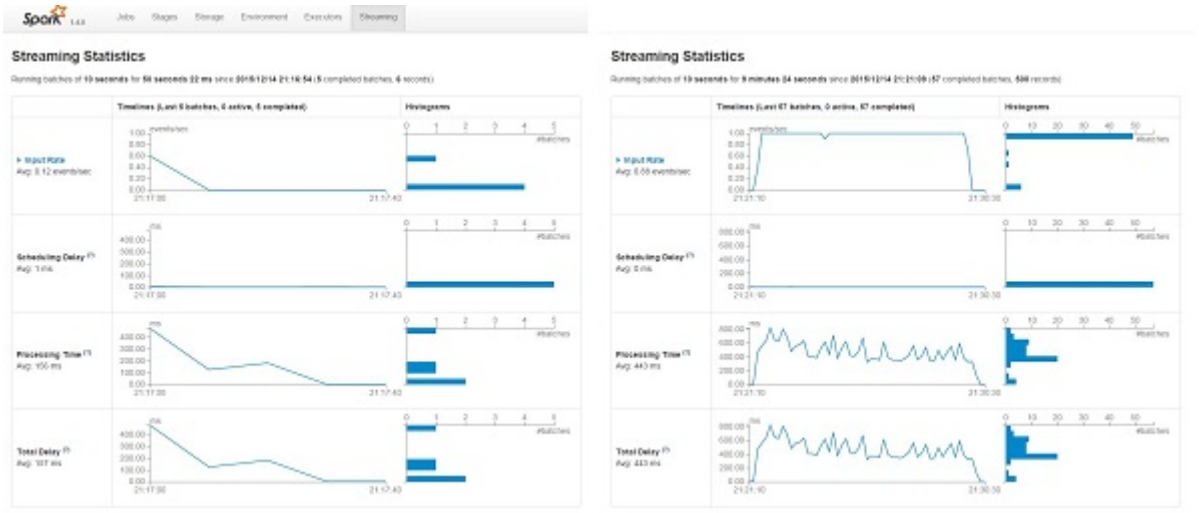

Figure 7. Spark visualization showing streaming statistics for the sample program

Conclusions

Spark Streaming library, part of Apache Spark eco-system, is used for data processing of real-time streaming data. In this article, we learned about how to use Spark Streaming API to process data generated by server logs and perform analytics on the real-time data streams.

What's Next

Machine Learning, Predictive Analytics, and Data Science are recently getting lot of attention for problem solving in different use cases. Spark MLlib, Spark's Machine Learning library, provides several built-in methods to use different machine learning algorithms like Collaborative Filtering, Clustering, and Classification.

In the next article, we will explore Spark MLlib and look at couple of use cases to illustrate how we can leverage data science capabilities of Spark, to make it easy to run Machine Learning algorithms.

In the future articles of this series, we'll look at the upcoming frameworks like BlinkDB and Tachyon.

References

- Big Data Processing using Apache Spark - Part 1: Introduction
- Big Data Processing using Apache Spark - Part 2: Spark SQL
- Apache Spark Main Website
- Spark Streaming Website
- Spark Streaming Programming Guide
- Spark Streaming - Scala Code Examples
- Spark Streaming - Java Code Examples
- Data Bricks' Apache Spark Reference Application
- Tagging and Processing Data in Real-Time Using Spark Streaming - Spark Summit 2015 Conference Presentation
- MapR's Quick Guide to Spark Streaming

About the Author

Srini Penchikala currently works as Senior IT Architect and is based out of Austin, Texas. He has over 20 years of experience in software architecture, design and development. Srini is currently authoring a book on Apache Spark. He is also the co-author of Spring Roo in Action book from Manning Publications. He has presented at conferences like JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. Srini also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, OReilly Network (ONJava), DevX Java, java.net and JavaWorld. He is a Lead Editor for Data Science community at InfoQ.

This is the third article of the "Big Data Processing with Apache Spark" series. Please see also: Part 1: Introduction, Part 2: Spark SQL, Part 3: Spark Streaming, Part 4: Spark Machine Learning, Part 5: Spark ML Data Pipelines, Part 6: Graph Data Analytics with Spark GraphX.