**Spark-Kafka Integration Overview**

In this lesson, we’ll be heavily looking at the important components of the Spark-Kafka integration, and also the architecture of offsets, Kafka, and triggers. We’ll also be discussing performance tuning, and looking at some application contexts. There are various Kafka libraries available in the Python Package Index PyPI. This lesson uses pykafka or pykafka.

## Glossary of Key Terms You Will Learn in this Lesson

* **Write Ahead Logs (WAL)**: This is where the operation is logged in a file format. When unfortunate events like driver or node failure happen and the application is restarted, the operation logged in the WAL can be applied to the data.
* **Broadcasting**: Spark allows users to keep a read-only variable cached on each machine rather than sending the data over the network with tasks. This is useful when you have a rather large dataset and don't want to send the dataset over the network.
* **Salting**: Another popular method of redistributing a dataset. This is done by adding a "fake-primary-key" to redistribute the data.

**Exercise - Create a Kafka Producer Server**

In this workspace, you’re going to create a Kafka Producer Server. You’ll be using this code for the next few workspace exercises. You have already learned how to create a Kafka Producer Server in the previous Kafka course, so refer back to those lessons if you need to! We’re just starting you with an empty Python file for this exercise, but you can refer to the solution code if you really need to. (Try not to!)

Requirements:

* Your Kafka Server should ingest the uber.json data file in the workspace correctly. If unsure of the correct path of the file, type pwd in the console to get the absolute path of the file.
* You can name your own topic and feel free to use any free port number for this server.
* This will be your bootstrap server for your streaming application.
* You should be able to check if your server ingests data correctly by using Kafka Consumer Console.
* You can use any Kafka library (pykafka, kafka, kafka-confluent, etc). But if you wish to use a library other than kafka-confluent or kafka-python, you will have to reinstall the library each time you wake up workspace (or anytime after you've refreshed, or woken up, or reset data, or used the "Get New Content" button). The idea here is for you to generate a Python file that has a Kafka producer, and this file should act as your bootstrap server.

Please note: If you encounter this error

ImportError: cannot **import** name 'SourceDistribution' **from** 'pip.\_internal.distributions.source' (/opt/conda/lib/python3.7/site-packages/pip/\_internal/distributions/source/\_\_init\_\_.py)

while installing any packages, please run this command:

conda install <package\_name>

Please refer to the first exercise upon encountering this.

**Kafka Data Source**

KafkaSource is the data source for Apache Kafka in Spark Structured Streaming. It's part of kafka-0-10-sql library in Spark (source code: [**https://github.com/apache/spark/blob/4513f1c0dc450e9249d43fdad618fdcaf8d399b6/external/kafka-0-10-sql/src/main/scala/org/apache/spark/sql/kafka010/KafkaSource.scala)**](https://github.com/apache/spark/blob/4513f1c0dc450e9249d43fdad618fdcaf8d399b6/external/kafka-0-10-sql/src/main/scala/org/apache/spark/sql/kafka010/KafkaSource.scala)). A couple important functions to note in this source are:

* getOffset(), which uses the KafkaOffsetReader to get the latest available offset
* getBatch(), which returns a DataFrame from the start to end of an offset.
* You should define spark-sql-kafka-0-10 module as part of the execution of your project. For Spark environments using shell script, we'll be using the following command to submit Spark jobs:

./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10\_2.12:2.3.4

What we’re actually doing here is including an external package to allow us to use Kafka (broker version 0.10), and the correct version of Spark we have installed (compiled with Scala 2.12 and Spark version 2.3.4). org.apache.spark:spark-sql-kafka-0-10\_<scala\_version>:<spark\_version>.

## KafkaSourceProvider Code Reference

KafkaSourceProvider requires these options

* subscribe, subscribepattern, or assign
* kafka.bootstrap.server

kafka\_df = spark.readStream.\

format("kafka").\ # **set** **data** ingestion **format** **as** Kafka

**option**("subscribe", "<topic\_name>").\ #This **is** **required** although it says **option**.

**option**("kafka.bootstrap.servers", "localhost:9092").\ #You will also need the **url** **and** port **of** the bootstrap **server**

**load**()

Reference: [**https://github.com/apache/spark/blob/master/external/kafka-0-10-sql/src/main/scala/org/apache/spark/sql/kafka010/KafkaSourceProvider.scala**](https://github.com/apache/spark/blob/master/external/kafka-0-10-sql/src/main/scala/org/apache/spark/sql/kafka010/KafkaSourceProvider.scala)

### Exercise: Kafka Source Provider

Using the Kafka Producer Server you created in the previous exercise, we can now ingest data from the Kafka Producer Server to a Structured Streaming application. First you’ll need to set up the entry point for the stream.

Requirements:

* Set up the entry point
* Use appropriate configurations in the options to ingest the Kafka stream
* Do a df.printSchema() to explore the schema of the default Kafka ingestion
* You can use any Kafka library (pykafka, kafka, kafka-confluent, etc). But if you wish to use a library other than kafka-confluent or kafka-python, you will have to reinstall the library each time you wake up workspace (or anytime after you've refreshed, or woken up, or reset data, or used the "Get New Content" button). The idea here is for you to generate a Python file that has a Kafka producer, and this file should act as your bootstrap server.
* Please note: If you encounter this error

ImportError: cannot **import** name 'SourceDistribution' **from** 'pip.\_internal.distributions.source' (/opt/conda/lib/python3.7/site-packages/pip/\_internal/distributions/source/\_\_init\_\_.py)

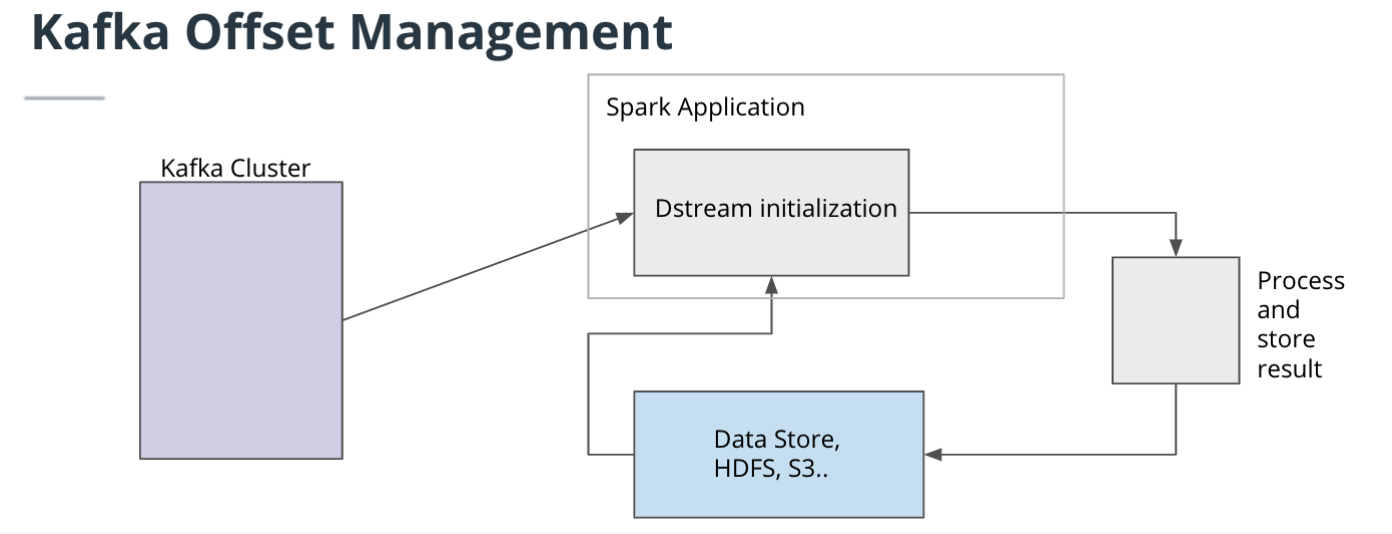
while installing any packages, please run this command:

conda install <package\_name>

## Kafka Offset Reader

KafkaOffsetReader is a class within Spark that uses Kafka's own KafkaConsumer API to read Kafka offsets. This class uses the ConsumerStrategy class that defines which Kafka topic and partitions should be read. ConsumerStrategy is another useful class to specify a fixed collection of partitions and to subscribe to a fixed collection of topics.

You will be using KafkaOffsetReader closely with KafkaSourceProvider.



## Triggers in Spark Streaming - Key Points

Write Ahead Logs enforce fault-tolerance by saving logs to a certain checkpoint directory. Enabling WAL can be done by using the Spark property, spark.streaming.receiver.writeAheadLog.enable.

Spark Triggers determine how often a streaming query needs to be executed. The trigger can be set using a few options in the query builder.

## Exercise: Exploring Triggers

Using the Kafka Server and Structured Streaming application, we will now play with the triggers.

Again, you can use any Kafka library (pykafka, kafka, kafka-confluent, etc). But if you wish to use a library other than kafka-confluent or kafka-python, you will have to reinstall the library each time you wake up workspace (or anytime after you've refreshed, or woken up, or reset data, or used the "Get New Content" button). The idea here is for you to generate a Python file that has a Kafka producer, and this file should act as your bootstrap server.

Please note: If you encounter this error

ImportError: cannot **import** name 'SourceDistribution' **from** 'pip.\_internal.distributions.source' (/opt/conda/lib/python3.7/site-packages/pip/\_internal/distributions/source/\_\_init\_\_.py)

while installing any packages, please run this command:

conda install <package\_name>

NEXT

## Recap of Lesson So Far

We have looked at a few key points regarding the integration of Kafka and Spark.

* KafkaSourceProvider provides a consumer for Kafka within Spark, therefore we will not need to create separate consumer modules for consumption.
* Managing offsets becomes crucial to making your Kafka and Spark microservices be best optimized. You’re in full control of your offset management, and you’ll have to make decisions best fitting your business context.

## Integrating Spark and Kafka

Now that we’ve looked over important features on the Kafka API in Spark, we’re ready to integrate Spark and Kafka.

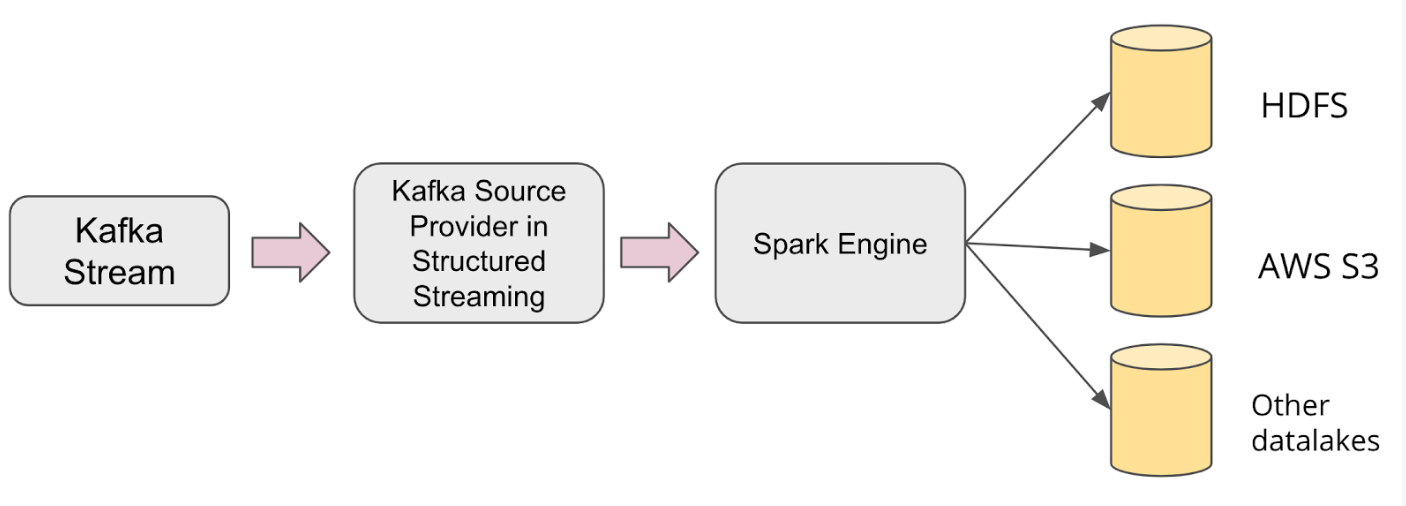
## Kafka Broker with Spark Structured Streaming

To recap from previous lessons, a Kafka broker receives messages from producers. A Kafka broker is a server, and we can think of a Kafka cluster as comprised of one or more Kafka brokers. Producers publish the data (or push messages) into Kafka topics. And then the Kafka consumer pulls messages from a Kafka topic.

Spark Streaming and Structured Streaming already provide a quick way to integrate Kafka to read data and write data back to Kafka.

Once received from Kafka, the source will have the following schema:

* key[binary]
* value[binary]
* topic[string]
* partition[int]
* offset[long]
* timestamp[long]
* timestampType[int]
* The value column should contain the most useful (the content) of the messages.



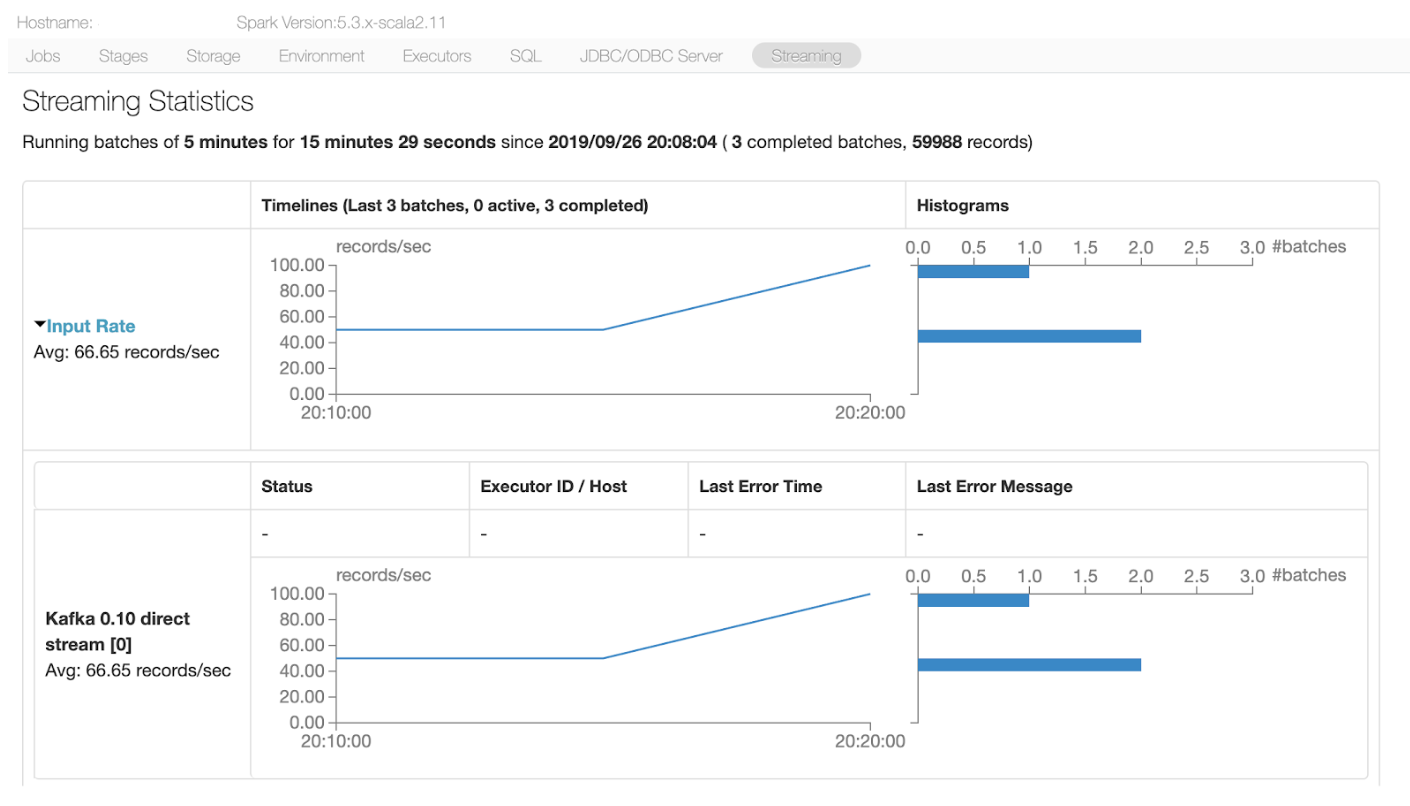
**[Sample Spark Structured Streaming Architecture with Kafka](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/233d13c1-2510-4e7a-9852-26491ae740b0/concepts/b51a4f8b-573d-44d8-8827-7c71c877766a)**

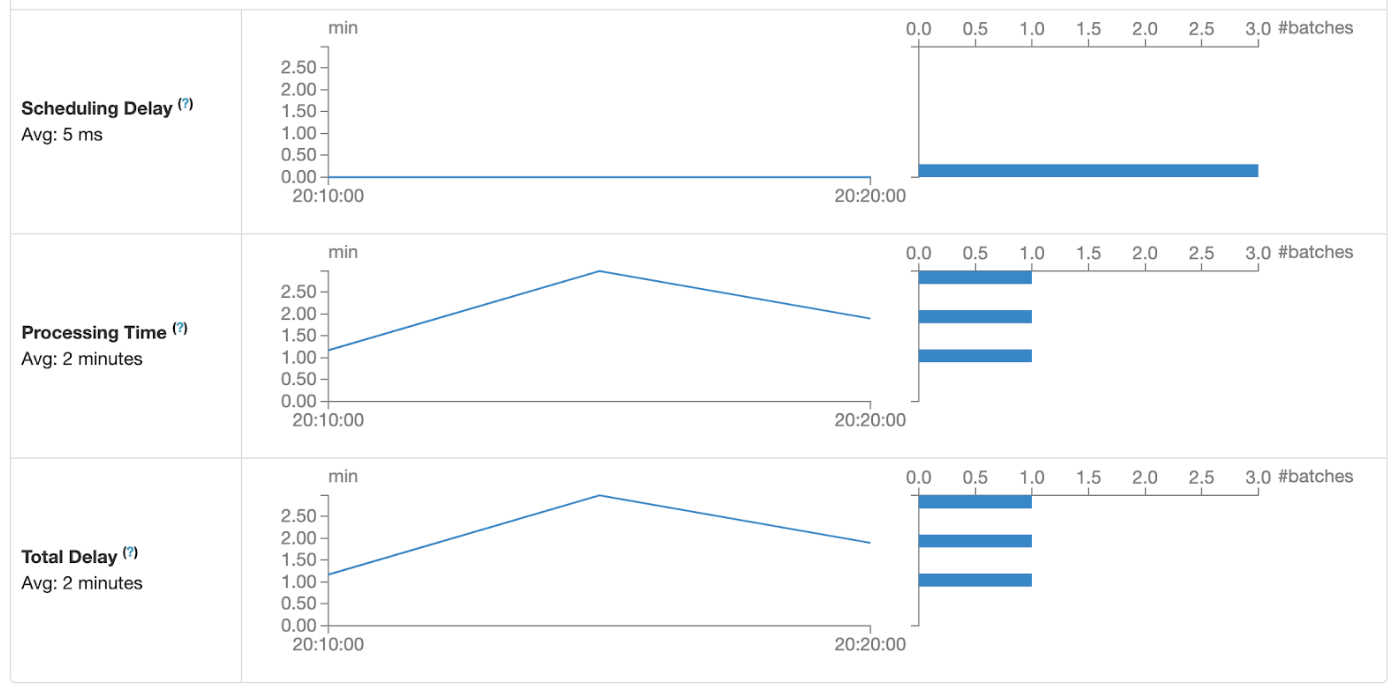
**Structured Streaming on Spark UI**

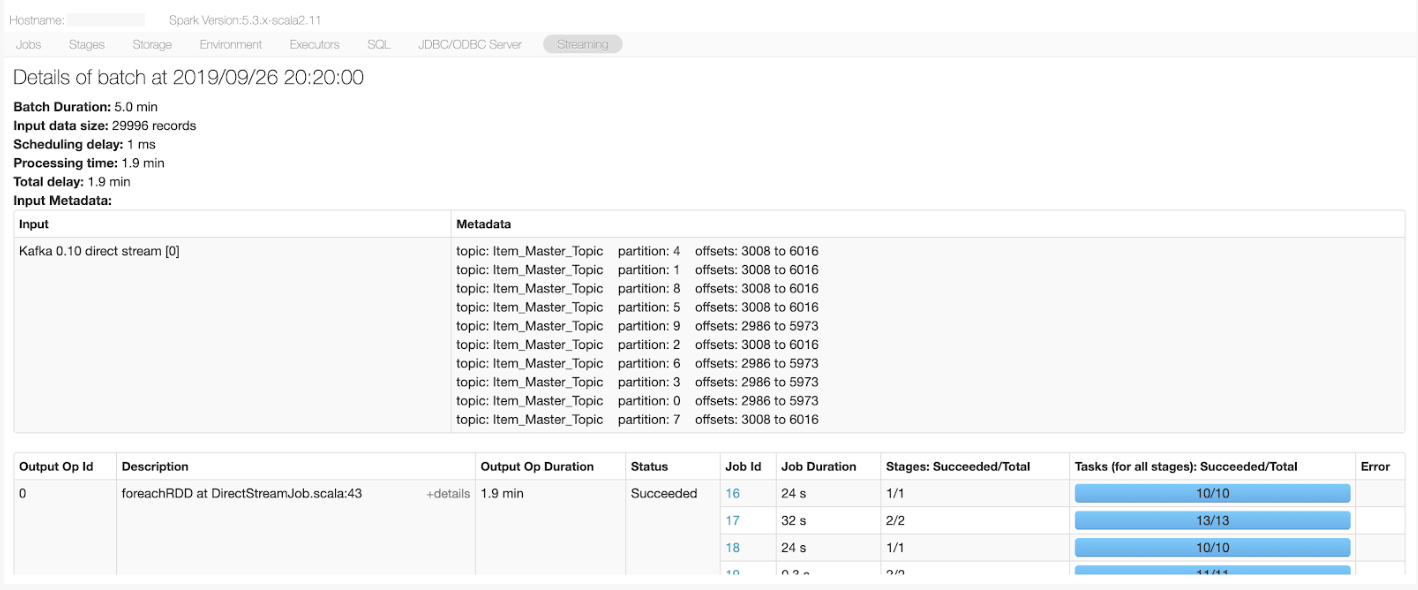
We have looked at various ways of using the Spark UI to monitor and debug Spark applications. How can we monitor and debug Streaming applications?

While the streaming application is running, you’ll see a “Streaming” tab (see image below). This tab in the UI enables you to concurrently monitor your job.

The “Streaming” tab contains some information about Spark & Scala versions, but it also shows what kind of streaming data you’re ingesting from (in the example below, Kafka stream), number of offsets in each partition, and also input rate of the data.







**[Sample Spark UI with Structured Streaming](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/233d13c1-2510-4e7a-9852-26491ae740b0/concepts/6aec0c48-b390-47a3-bc4e-ba046e57a6ea)**

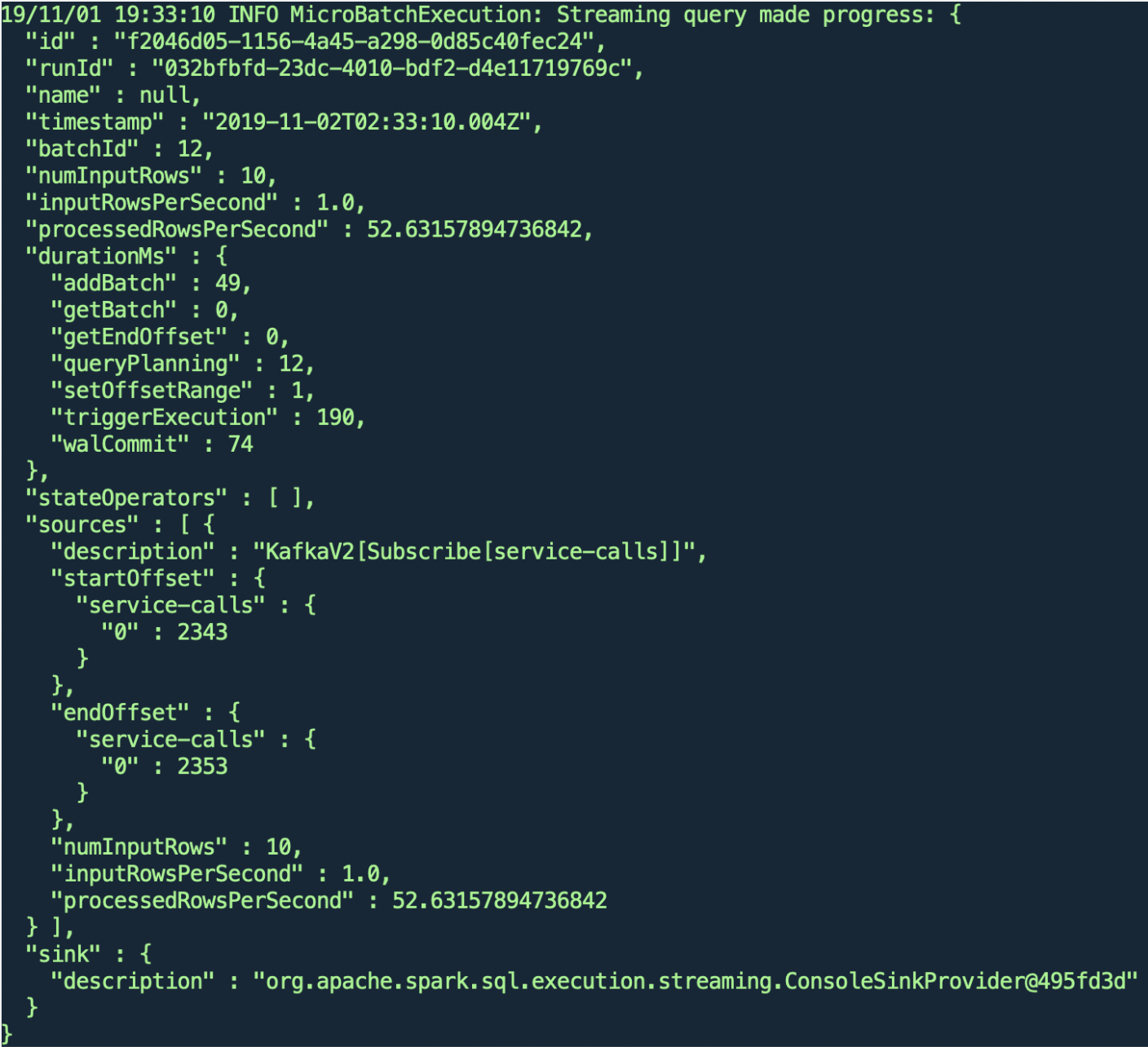
Thanks for your response! Here are some of my thoughts:

In any case you need a streaming application. Some popular domains that use Kafka and Spark integration are advertisement technology, banking, fraud detection, etc. Kafka provides data streams of events and a Spark application will often be integrated to process the stream of data. The sinks should be implemented wherever you want to save your intermediary data, for example, at the end of each micro-batch.

## Spark Console Shows Progress Reports

In this concept, we’ll take a look at the Progress Report that shows up in the console. All this information that is coming through the console could be cumbersome, but once we understand what each field is, it’ll be easier to understand.

You’ll see something like this in the Spark Console:



This is called a Progress Report. There are obvious key value pairs, like timestamp and batchId, id for each unique query. But what we really need to take a look at are these three pieces of data, because they provide important information regarding the batch size and processed rows:

* numInputRows : The aggregate (across all sources) number of records processed in a trigger.
* inputRowsPerSecond : The aggregate (across all sources) rate of data arriving.
* processedRowsPerSecond : The aggregate (across all sources) rate at which Spark is processing data.
* Also you can see that we’re using ConsoleSink which is one of the sinks we mentioned before.

## Breakdown of Report Component

Breakdown of ProgressReporter:

* currentBatchId[Long] : ID of current micro-batch
* Id: UUID of the streaming query
* Name: Name of the streaming query. This can be configured in your code
* Sink: Streaming sink of the streaming query

## Exercise: Generate Progress Report

Requirements:

* Add and change the value of maxRatePerPartition in your option
* Change the value of trigger
* Change the value of startingOffsets
* Again, you can use any Kafka library (pykafka, kafka, kafka-confluent, etc). But if you wish to use a library other than kafka-confluent or kafka-python, you will have to reinstall the library each time you wake up workspace (or anytime after you've refreshed, or woken up, or reset data, or used the "Get New Content" button). The idea here is for you to generate a Python file that has a Kafka producer, and this file should act as your bootstrap server.

### Things to think about

Thanks for your response. Here are my answers for what each field means:

* id: ID of the current batch in UUID
* currentDurationMs: cumulative times in milliseconds
* currentTriggerStartOffsets: start offsets per source
* sources.description: the description of source - kafka, rate, etc
* sink: sink types (console, kafka, etc)

# Scenario 1

Given problem: We're given a hypothetical Spark streaming application. This application receives data from Kafka. Every 2 minutes, you can see that Kafka is producing 60000 records. But at its maximum load, the application takes between 2 minutes for each micro-batch of 500 records. How do we improve the speed of this application?

1. We can tweak the application's algorithm to speed up the application.

* Let's say the application's algorithm is tweaked - how can you check if most or all of the CPU cores are working?
* In a Spark Streaming job, Kafka partitions map 1:1 with Spark partitions. So we can increase Parallelism by increasing the number of partitions in Kafka, which then will increase Spark partitions.

1. We can check if the input data was balanced/unbalanced, skewed or not. We can check the throughput of each partition using Spark UI, and how many cores are consistently working. You can also use the htop command to see if your cores are all working (if you have a small cluster).

* Increase driver and executor memory: Out-of-memory issues can be frequently solved by increasing the memory of executor and driver. Always try to give some overhead (usually 10%) to fit your excessive applications.

1. You could also set spark.streaming.kafka.maxRatePerPartition to a higher number and see if there is any increase in data ingestion.

# Scenario 2

Other methods of improving your Spark application are by improving system stability and studying your data.

When the heap gets very big (> 32GB), the cost of GC (garbage collection) gets large as well. We might see this when the join application runs with large shuffles (> 20GB), and the GC time will spike.

Can you use a sample of your data? How about using **salting** or **broadcasting**? Let’s discuss these below.

In the ideal world, when you operate a join on your dataset, join keys will be evenly distributed and each partition will be nicely organized. In the real world, uneven partitioning is unavoidable due to the nature of your dataset.

Using the Spark UI (or even just your logs), you'll commonly see the errors below:

* Frozen stages
* Low utilization of CPU (workers not working)
* Out of memory errors

Carefully studying your data, to minimize the skewed data problem, we can try the following:

* **Broadcasting**: You can increase the autoBroadcastJoinThreshold value in spark.sql property so that the smaller tables get “broadcasted.” This is helpful when you have a large dataset that will not fit into your memory.
* **Salting**: If you have a key with high cardinality, your dataset will be skewed. Now you introduce a “salt,” modifying original keys in a certain way and using hash partitioning for proper redistribution of records.

Now that we have cleaned up our data and the skew problem as much as we could, and also assuming that our code is optimized, let’s talk about how we can stabilize the system through a couple of different methods:

* Auto scaling
* Speculative execution

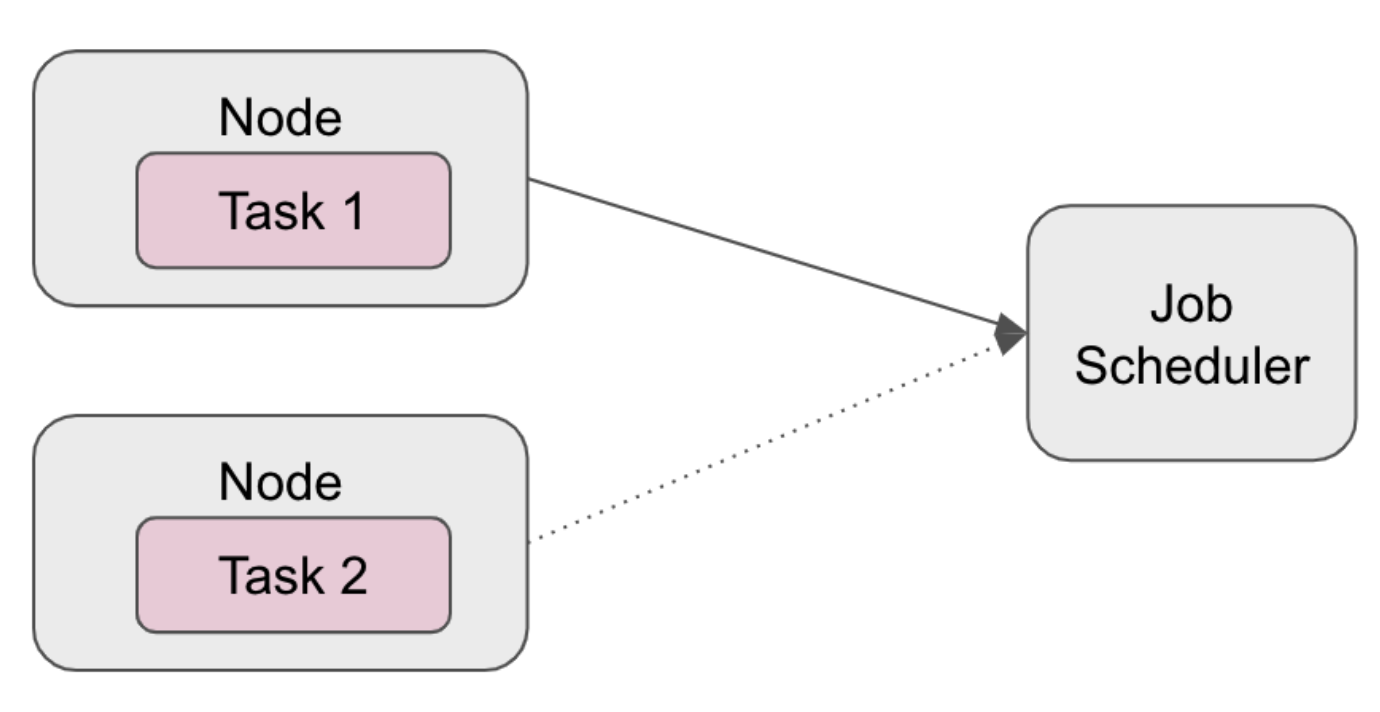
Auto scaling is only doable with cloud clusters as you can always add more nodes freely. Two popular tech stacks that are used are AWS Auto Scaling (if AWS EMR clusters are used) or auto scaling with Kubernetes (a container-orchestration system).

Speculative execution is another popular addition to stabilize and reduce bottleneck-like threshold. Speculative execution in Spark detects the “speculated task” (which means this task is running slower than the median speed of the other tasks), and then submits the “speculated task” to another worker. This enables continuous parallel execution rather than shutting off the slow task.

## Notes on Scenario 2 diagram below

In this sample image below, let’s say task 1 and 2 are identical to each other. In fact, task 2 is a duplicate of task 1. In the node that contains task 1, if the task is seen to be slower than the median speed, then the job scheduler launches the task 2 (duplicate of task 1) in another node.

If the duplicate task is faster, the result of task 2 will be submitted to the job scheduler. If the original task is faster, then the result of task 1 will be used. In Spark UI, whichever node that was killed due to late execution will be noted as killed intentionally.



### Things to think about

Thanks for your response! Here are my thoughts:

For additional improvements, you could always do a set of rigorous stress testing on your system to better understand your system stability and daily throughput and latency of the data, and even go further on what could be your max daily throughput and latency of the data.

### Further Optional Reading on Performance Tuning

Further reading on JVM and GC (for those who are not familiar with Java): [**https://betsol.com/2017/06/java-memory-management-for-java-virtual-machine-jvm/**](https://betsol.com/2017/06/java-memory-management-for-java-virtual-machine-jvm/)

Tuning: [**https://spark.apache.org/docs/latest/tuning.html**](https://spark.apache.org/docs/latest/tuning.html)

Tuning on Cloudera blog: [**https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-1/**](https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-1/)