SPARK+AI SUMMIT 2019

WIFI SSID:SparkAISummit | Password: UnifiedAnalytics

ORGANIZED BY databricks

Spark Listeners

A Crash Course in Fast, Easy Monitoring

Landon Robinson & Ben Storrie

SPOTX

#UnifiedAnalytics #SparkAISummit

Who We Are

Landon Robinson
Data Engineer

Ben Storrie
Data Engineer

Big Data Infrastructure Team @ SpotX

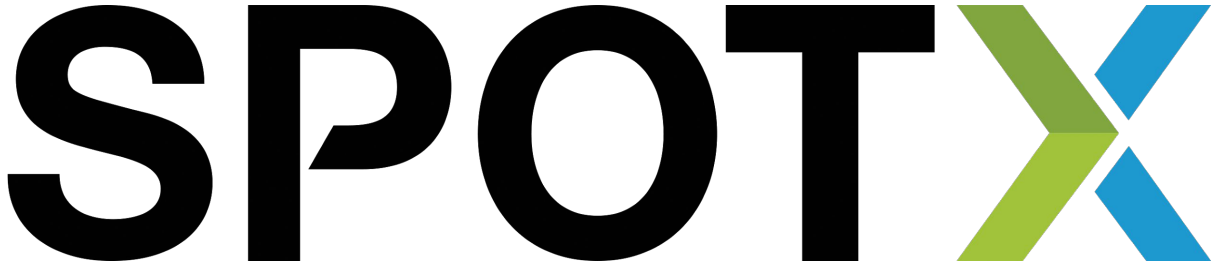
But first… why are we here?

→ Building monitoring for large apps can be **tough**.

→ You want an **effective and reliable solution**.

→ **Spark Listeners are an *awesome* solution!**

Our Company



THE MODERN VIDEO
AD SERVING PLATFORM

(we show you video ads)

(which means we also process a lot of data)

We Process a Lot of Data

Data:

- 220 MM+ Total Files/Blocks
- 8 PB+ HDFS Space
- 20 TB+ new data daily
- 100MM+ records/minute
- 300+ Data Nodes

Apps:

- Thousands of daily Spark apps
- Hundreds of daily user queries
- Several 24/7 Streaming apps

Spark Streaming is Key for Us

Spark Streaming apps are **critical** for us.

They:

- a) **process billions of records** every day
- b) need **real-time monitoring (***visual dashboards + alerts)*
- *c) without sacrificing performance*

So… a clever solution is necessary.

Our Solution

Monitoring and visualization of a Spark Streaming app that is:

- Fast
- Easy to integrate
- Has zero-to-minimal performance impact

Goal & Talking Points

Equip you with the knowledge and code to get fast, easy monitoring *implemented in your app using Spark Listeners.*

Talking Points

- Common Spark Streaming Metrics
- Monitoring is Awesome
- Monitoring can be Difficult
- Less than Ideal Monitoring
- Better Monitoring
  - Spark Listeners!
    - StreamingListeners
    - BatchListeners

Common Metrics of Spark Streaming Apps

Performance:

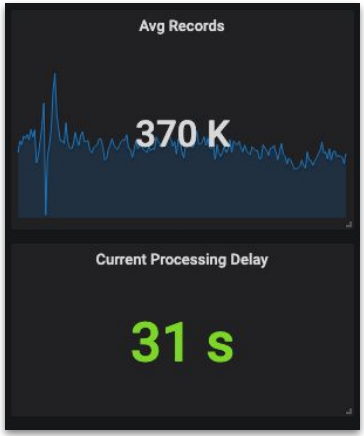
- batch duration
- batch scheduling delays
- # of records processed per batch

Problems:

- failures & exceptions

Recovery:

- Position (offset) within Kafka topic




```
mysql> select * from offsets where app='myApp1' limit 10;
+----------+------------------+--------------+-----------+---------------------+------------+---------------------+
| app      | topic            | partition_id | offset    | offset_ts           | batch_size | modified_date       |
+----------+------------------+--------------+-----------+---------------------+------------+---------------------+
| myApp1   | kafka.some_topic1 |            0 | 877744060 | 2019-03-27 23:30:55 |        836 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |            1 | 877194072 | 2019-03-27 23:30:55 |        836 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |            2 | 877804618 | 2019-03-27 23:30:55 |        810 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |            3 | 877733267 | 2019-03-27 23:30:55 |        785 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |            4 | 878811924 | 2019-03-27 23:30:55 |        859 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |            5 | 877818167 | 2019-03-27 23:30:55 |        813 | 2019-03-27 23:30:55 |
+----------+------------------+--------------+-----------+---------------------+------------+---------------------+
```

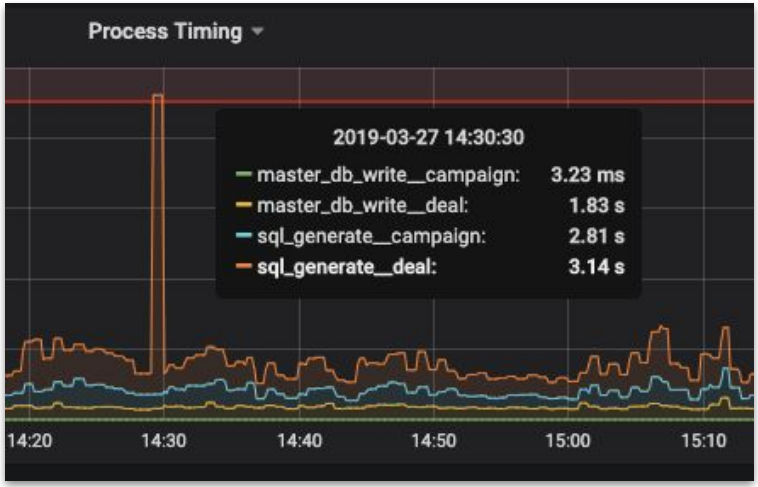
Monitoring is Awesome

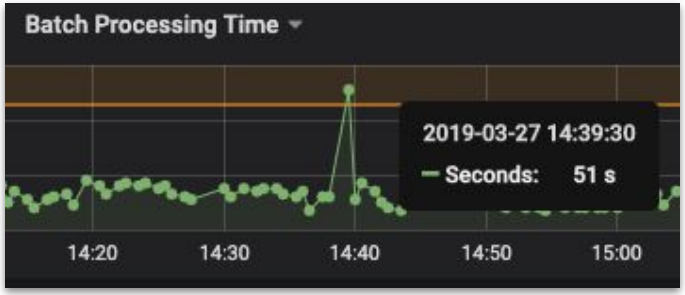
It can reveal:

- **How your app is performing**
- **Problems + Bugs!**

And provide opportunities to:

- **See and address issues**
- **Observe behavior visually**





Monitoring can be Difficult to Implement (*especially when working with very **big** data*)

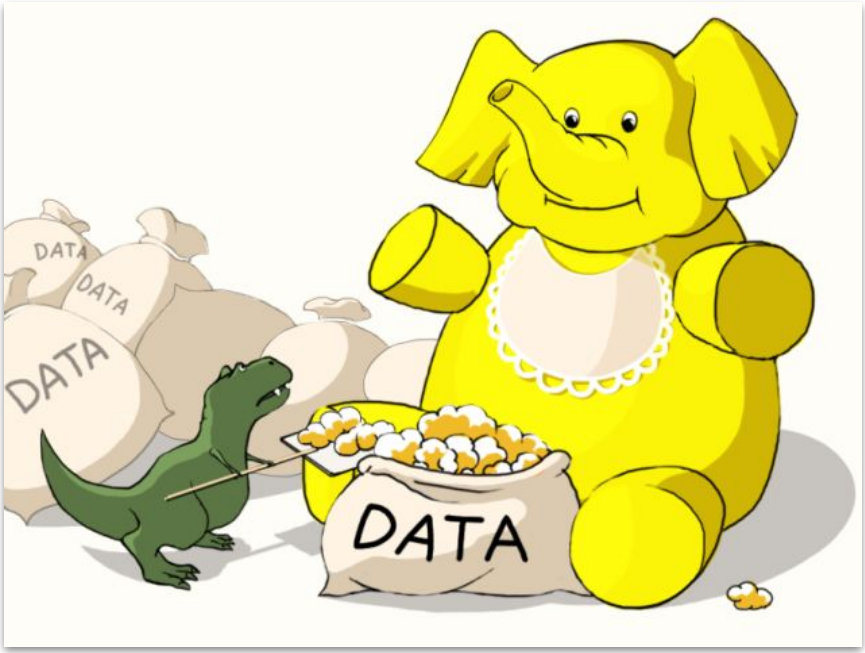
Usually because it requires reading your data!

Example: *.count().print()*

- Shuffling involved, and is often a blocking operation!

Mapping over data to gather metrics can:

- be expensive
- and add processing delays
  - Virtually unreliable with big data



Less than Ideal Monitoring

Make the app compute *all* your metrics!

Example: Looping over RDDs to:

- **Count records**
- **Track Kafka offsets**
- **Processing time / delays**

Why is this bad?

- Calculating performance significantly impacts performance… not great.
- All these metrics are **calculated by Spark!**

```
myStreamingData.count().print()
```

```
myStreamingData.foreachRDD(rdd => {
  rdd.map(x => {
    println(x.topic())
    println(x.offset())
    println(x.partition())
  })
})
```

```
val t0 = System.nanoTime()
// execute a task
val t1 = System.nanoTime()
val processing_time = (t1 - t0) / 1000
println("I took this long: " + processing_time)
```

There has to be a better way...

Streaming Listeners!

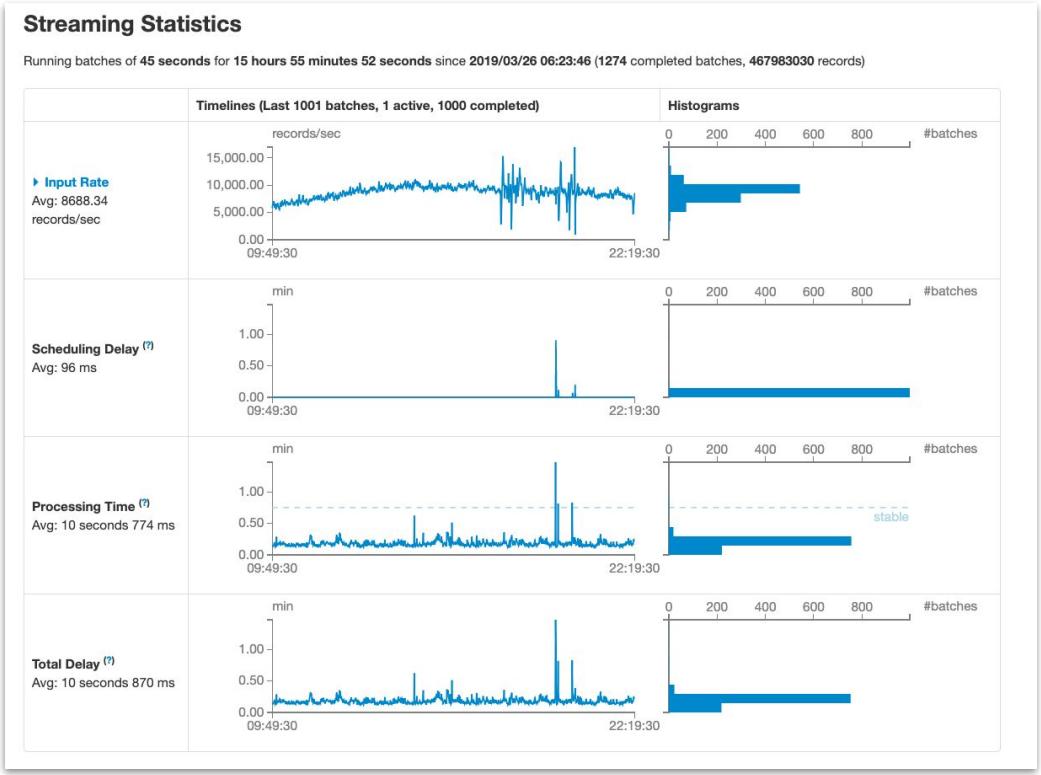
The Better Way: Streaming Listeners

Let a Spark Streaming Listener handle the basics metrics.

Only make your app compute the "special" metrics it has to.

Spark already crunches some numbers for you -- most evident in the Streaming tab.

So let's take advantage of it!



Streaming Listeners: *StreamingListener* Trait

The *StreamingListener Trait* within the Developer API has everything you need.

```
package org.apache.spark.streaming.scheduler
```

It has 8 convenience methods you can override, each with relevant, contextual information from Spark.

***onBatchCompleted()* is the most useful to us. It allows you to execute logic after each batch.**

```
/**
 * :: DeveloperApi ::
 * A listener interface for receiving information about an ongoing streaming
 * computation.
 */
@DeveloperApi
trait StreamingListener {

  /** Called when a receiver has been started */
  def onReceiverStarted(receiverStarted: StreamingListenerReceiverStarted) { }

  /** Called when a receiver has reported an error */
  def onReceiverError(receiverError: StreamingListenerReceiverError) { }

  /** Called when a receiver has been stopped */
  def onReceiverStopped(receiverStopped: StreamingListenerReceiverStopped) { }

  /** Called when a batch of jobs has been submitted for processing. */
  def onBatchSubmitted(batchSubmitted: StreamingListenerBatchSubmitted) { }

  /** Called when processing of a batch of jobs has started.  */
  def onBatchStarted(batchStarted: StreamingListenerBatchStarted) { }

  /** Called when processing of a batch of jobs has completed. */
  def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted) { }

  /** Called when processing of a job of a batch has started. */
  def onOutputOperationStarted(
      outputOperationStarted: StreamingListenerOutputOperationStarted) { }

  /** Called when processing of a job of a batch has completed. */
  def onOutputOperationCompleted(
      outputOperationCompleted: StreamingListenerOutputOperationCompleted) { }
}
```

Streaming Listeners: Extend Trait in New Listener

Create a new listener that extends the *StreamingListener* Trait.

Override the convenience methods.

Within each is contextual monitoring information.

Let's look at onBatchCompleted().

```
/**
  * :: SpotXSparkStreamingListener ::
  * A simple StreamingListener that logs summary statistics across Spark Streaming batches; inherits from DeveloperAPI.
  *
  * @param influxHost        Hostname of the Influx service
  * @param influxDB          Database name in Influx to write to
  * @param influxMeasurement Measurement name in Influx to write to
  * @param mySQLHost         Hostname of the MySQL service
  * @param mySQLDB           Database name in MySQL to write to
  * @param mySQLTable        Table name in MySQL to write to
  * @param mySQLUser         Username for authentication in MySQL
  * @param mySQLPwd          Password for authentication in MySQL
  * @param mySQLConsumer     Unique name for tracking offsets across streaming apps
  */
class SpotXSparkStreamingListener (influxHost: String,
                                   influxDB: String,
                                   influxMeasurement: String,
                                   mySQLHost: String,
                                   mySQLDB: String,
                                   mySQLTable: String,
                                   mySQLConsumer: String,
                                   mySQLUser: String,
                                   mySQLPwd: String) extends StreamingListener {

  // ===========================================================================
  // Variables
  // ===========================================================================

  @transient lazy val influx = InfluxDBWriter.create(influxHost)
  @transient lazy val mysql = MySQLConnection(host = mySQLHost, table = mySQLTable, username = mySQLUser, password = mySQLPwd, database = mySQLDB
  @transient lazy val mySQLConnectionPool = ConnectionPool(mysql.toString).getConnection

  // ===========================================================================
  // Spark Listener Override Methods (for Batches)
  // ===========================================================================

  /**
    * This method executes when a Spark Streaming batch completes.
    *
    * @param batchCompleted Class having information on the completed batch
    */
  override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit = {

    // write performance metrics to influx
    writeBatchSchedulingStatsToInflux(batchCompleted)

    // write offsets (state) to mysql
    writeBatchOffsetsAndCounts(batchCompleted)
```

Streaming Listeners: *onBatchCompleted()*

What data is available to us from Spark?

- **Batch Info**
  - numRecords (total)
  - Processing times
  - delays
- **Stream Info**
  - Topic object
    - Offsets
    - numRecords (topic level)

```
batch.batchInfo.
```

```
SpotXSparkStreamingListener.super (tv.spotx.scala.monitoring.listeners)   SpotXSparkStreamingListener
batchTime                                                                 Time
outputOperationInfos                                  Map[Int, OutputOperationInfo]
processingEndTime                                                  Option[Long]
processingStartTime                                                Option[Long]
streamIdToInputInfo                                    Map[Int, StreamInputInfo]
submissionTime                                                             Long
numRecords                                                                 Long
processingDelay                                                    Option[Long]
schedulingDelay                                                    Option[Long]
totalDelay                                                         Option[Long]
```

```
// Store the
val processin
  batch.batch
}
else {
  0
}

// Store the
val schedulin
  batch.batch
```

Streaming Listeners: *onBatchCompleted()*

We want to write performance stats to Influx using our custom method: ***writeBatchSchedulingStatsToInflux()***

We want to write our Kafka Offsets to MySQL using our custom method: ***writeBatchOffsetsAndCounts()***

```
/**
  * This method executes when a Spark Streaming batch completes.
  *
  * @param batchCompleted Class having information on the completed batch
  */
override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit = {

  // write performance metrics to influx
  writeBatchSchedulingStatsToInflux(batchCompleted)

  // write offsets (state) to mysql
  writeBatchOffsetsAndCounts(batchCompleted)
}
```

writeBatchSchedulingStatsToInflux()

This custom method consumes the Batch Info stored in the object...

- **Batch duration**
- **Batch delay**
- **total records**
  - **across all streams/topics**

… and writes them to Influx!

```
/**
  * Pulls, parses, and logs the key performance metrics of the Streaming app and logs them to Influx.
  * Processing Time: How many seconds needed to complete this batch (i.e. duration).
  * Scheduling Delay: How many seconds the start time of this bach was delayed.
  * Num Records: The total number of input records from a live stream consumed this batch.
  *
  * @param batch Class having information on the completed batch
  */
def writeBatchSchedulingStatsToInflux(batch: StreamingListenerBatchCompleted): Unit = {

  // Store the processing time for this batch in seconds
  val processingTime = if (batch.batchInfo.processingDelay.isDefined) {
    batch.batchInfo.processingDelay.get / 1000
  }
  else {
    0
  }

  // Store the scheduling delay for this batch in seconds
  val schedulingDelay = if (batch.batchInfo.schedulingDelay.isDefined && batch.batchInfo.schedulingDelay.get > 0) {
    batch.batchInfo.schedulingDelay.get / 1000
  }
  else {
    0
  }

  // Store the total record count for this batch
  val numRecords = batch.batchInfo.numRecords

  // Log all three (3) metrics to Influx
  influx.write(influxDB, influxMeasurement, Seq(), Seq(("processingTime", processingTime)))
  influx.write(influxDB, influxMeasurement, Seq(), Seq(("schedulingDelay", schedulingDelay)))
  influx.write(influxDB, influxMeasurement, Seq(), Seq(("numRecords", numRecords)))

}
```

writeBatchSchedulingStatsToInflux()

Grafana dashboard reading from an Influx time-series database.

Streaming Listeners: *onBatchCompleted()*

We want to write performance stats to Influx using our custom method:

writeBatchSchedulingStatsToInflux()

We want to write our Kafka Offsets to MySQL using our custom method:

writeBatchOffsetsAndCounts()

```
/**
  * This method executes when a Spark Streaming batch completes.
  *
  * @param batchCompleted Class having information on the completed batch
  */
override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit = {

  // write performance metrics to influx
  writeBatchSchedulingStatsToInflux(batchCompleted)

  // write offsets (state) to mysql
  writeBatchOffsetsAndCounts(batchCompleted)
}
```

writeBatchOffsetsAndCounts()

This custom method consumes the Stream Info stored in the object and has two steps:

- **Write offsets to MySQL**
- **Write # records to Influx**
  - **per stream/topic**

```
/**
  * A combination method that will handle both influx writes and MySQL offsets.
  * This is effectively a convenience method of writeBatchOffsetsToMySQL + writeBatchCountsToInflux.
  *
  * @param batch Class having information on the completed batch
  */
def writeBatchOffsetsAndCounts(batch: StreamingListenerBatchCompleted): Unit = {

  // for each stream topic consumed this batch...
  batch.batchInfo.streamIdToInputInfo.foreach(topic => {

    // write offsets for this topic to mysql
    writeTopicOffsetsToMySQL(topic)

    // write record count for this topic this batch
    writeTopicCountToInflux(topic)
  })
}
```

writeTopicOffsetsToMySQL()

This custom method takes a Topic object…

… and finds the last offset processed for each kafka partition.

It then will update each partition in a database with the latest row/offset processed.

```
/**
  * Takes a topic object and writes the max offset for each partition it contains this batch to MySQL.
  *
  * @param topic A topic object within a Batch's StreamIdToInputInfo
  */
def writeTopicOffsetsToMySQL(topic: Tuple2[Int, StreamInputInfo]): Unit = {

  // map offset info to OffsetRange objects
  val partitionOffsets = topic._2.metadata("offsets").asInstanceOf[List[OffsetRange]]

  // for every partition's range of offsets
  partitionOffsets.map(offsetRange => {

    // write the new starting offset for each partition in the topic to the state db
    var maxOffset = offsetRange.untilOffset - 1

    // create a now() timestamp
    val now = new DateTime().toString("YYYY-MM-dd HH:mm:ss")

    // form the sql
    val sql =
      s"""INSERT INTO $mySQLDB.$mySQLTable (consumer, topic, partition_id, offset, offset_ts, batch_size)
         VALUES
         ('$mySQLConsumer', "${offsetRange.topic}", ${offsetRange.partition}, '$maxOffset', '$now', ${offsetRange.count})
         ON DUPLICATE KEY UPDATE offset_ts = VALUES(offset_ts), offset = VALUES(offset),
           batch_size = VALUES(batch_size)
      """

    // execute the sql to offload offsets to the table
    val st = mySQLConnectionPool.createStatement
    st.execute(sql)
    st.close()
  })
}
```

writeTopicOffsetsToMySQL()

```
mysql> select * from offsets where app='myApp1' limit 10;
+----------+------------------+-------------+-----------+---------------------+-----------+---------------------+
| app      | topic            | partition_id | offset    | offset_ts           | batch_size | modified_date       |
+----------+------------------+-------------+-----------+---------------------+-----------+---------------------+
| myApp1   | kafka.some_topic1 |           0 | 877744060 | 2019-03-27 23:30:55 |        836 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |           1 | 877194072 | 2019-03-27 23:30:55 |        836 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |           2 | 877804618 | 2019-03-27 23:30:55 |        810 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |           3 | 877733267 | 2019-03-27 23:30:55 |        785 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |           4 | 878811924 | 2019-03-27 23:30:55 |        859 | 2019-03-27 23:30:55 |
| myApp1   | kafka.some_topic1 |           5 | 877818167 | 2019-03-27 23:30:55 |        813 | 2019-03-27 23:30:55 |
+----------+------------------+-------------+-----------+---------------------+-----------+---------------------+
```

MySQL table maintaining latest offsets for an app.
Data is at an app / topic / partition level.

Reading Offsets from MySQL

Your offsets are now stored in a DB after each batch completes.

Whenever your app restarts, it reads those offsets from the DB...

And starts processing where it last left off!

```
def subscribeToTopicWithState(
                                streamingContext: StreamingContext,
                                topic: String,
                                kafkaParams: Map[String, Object],
                                dbConnection: MySQLConnection,
                                stateConsumer: String,
                                resetOffsets: Boolean = false
                    ): InputDStream[ConsumerRecord[String, Option[GenericRecord]]] = {

  // Store list of topics in an array
  val topics = Array(topic)

  // Either Get Offsets from a State Database or Reset to Latest
  val offsets = getOffsetsFromState(dbConnection, stateConsumer, topic, resetOffsets)

  // Subscribe to the topic based on the offsets provided
  val strategy = getStrategy(offsets, kafkaParams, topics)

  // Create a direct stream and pull data
  val subscribed = KafkaUtils.createDirectStream[String, Option[GenericRecord]](
    streamingContext,
    LocationStrategies.PreferConsistent,
    strategy
  )

  // Filter data coming out of Kafka to remove bad records
  subscribed
}
```

Getting Offsets from the Database

```
def getOffsetsFromState(dbConnection: MySQLConnection,
                        consumer: String,
                        topic: String,
                        resetOffsets: Boolean
                       ): Map[TopicPartition, Long] = {

  // Establish a MySQL Connection given a URL
  val stateDbConn = ConnectionPool(dbConnection.toString).getConnection

  // If You want to reset offsets, just return an empty map
  val offsets =
    if (resetOffsets) {
      new HashMap[TopicPartition, Long]()
    }

    // If you want to use your existing offsets, get them from MySQL
    else {
      val stmt = stateDbConn.createStatement
      val query = "SELECT * FROM " + dbConnection.database + "." + dbConnection.table +
        " WHERE consumer = '" + consumer + "' AND topic = '" + topic + "'"
      logger.info(query)
      val result = stmt.executeQuery(query)
      new RsIterator(result).map(
        x =>
          new TopicPartition(
            x.getString("topic"),
            x.getInt("partition_id")
          ) ->
            (x.getLong("offset") + 1L) // Add one because we record last known offset
      ).toMap
    }

  offsets

}
```

Example: Reading Offsets from MySQL

```scala
def subscribeToTopicWithState(
                              streamingContext: StreamingContext,
                              topic: String,
                              kafkaParams: Map[String, Object],
                              dbConnection: MySQLConnection,
                              stateConsumer: String,
                              resetOffsets: Boolean = false
                    ): InputDStream[ConsumerRecord[String, Option[GenericRecord]]] = {

  // Store list of topics in an array
  val topics = Array(topic)

  // Either Get Offsets from a State Database or Reset to Latest
  val offsets = getOffsetsFromState(dbConnection, stateConsumer, topic, resetOffsets)

  // Subscribe to the topic based on the offsets provided
  val strategy = getStrategy(offsets, kafkaParams, topics)

  // Create a direct stream and pull data
  val subscribed = KafkaUtils.createDirectStream[String, Option[GenericRecord]](
    streamingContext,
    LocationStrategies.PreferConsistent,
    strategy
  )

  // Filter data coming out of Kafka to remove bad records
  subscribed
}
```

Example: Reading Offsets from MySQL

```
def getStrategy(
                offsets: Map[TopicPartition, Long],
                kafkaParams: Map[String, Object],
                topics: Array[String]
              ): ConsumerStrategy[String, Option[GenericRecord]] ={

  // if offsets were pulled from MySQL, assign the stream to start there
  val strategy = if (offsets.size > 0) {
    ConsumerStrategies.Assign[String, Option[GenericRecord]](offsets.keys.toList, kafkaParams, offsets)
  }

  // otherwise, start at latest data
  else {
    ConsumerStrategies.Subscribe[String, Option[GenericRecord]](topics, kafkaParams)
  }

  strategy
}
```

Reading Offsets from MySQL

```
def subscribeToTopicWithState(
                                streamingContext: StreamingContext,
                                topic: String,
                                kafkaParams: Map[String, Object],
                                dbConnection: MySQLConnection,
                                stateConsumer: String,
                                resetOffsets: Boolean = false
                    ): InputDStream[ConsumerRecord[String, Option[GenericRecord]]] = {

  // Store list of topics in an array
  val topics = Array(topic)

  // Either Get Offsets from a State Database or Reset to Latest
  val offsets = getOffsetsFromState(dbConnection, stateConsumer, topic, resetOffsets)

  // Subscribe to the topic based on the offsets provided
  val strategy = getStrategy(offsets, kafkaParams, topics)

  // Create a direct stream and pull data
  val subscribed = KafkaUtils.createDirectStream[String, Option[GenericRecord]](
    streamingContext,
    LocationStrategies.PreferConsistent,
    strategy
  )

  // Filter data coming out of Kafka to remove bad records
  subscribed
}
```

writeBatchOffsetsAndCounts()

This custom method has two steps:

- **Write offsets to MySQL**
- **Write # records to Influx**
  - **per stream/topic**

```
/**
  * A combination method that will handle both influx writes and MySQL offsets.
  * This is effectively a convenience method of writeBatchOffsetsToMySQL + writeBatchCountsToInflux.
  *
  * @param batch Class having information on the completed batch
  */
def writeBatchOffsetsAndCounts(batch: StreamingListenerBatchCompleted): Unit = {

  // for each stream topic consumed this batch...
  batch.batchInfo.streamIdToInputInfo.foreach(topic => {

    // write offsets for this topic to mysql
    writeTopicOffsetsToMySQL(topic)

    // write record count for this topic this batch
    writeTopicCountToInflux(topic)
  })
}
```

writeTopicCountToInflux()

This method takes a *Topic* object and writes the # of records processed for a single topic (more granular than batch-level record total).

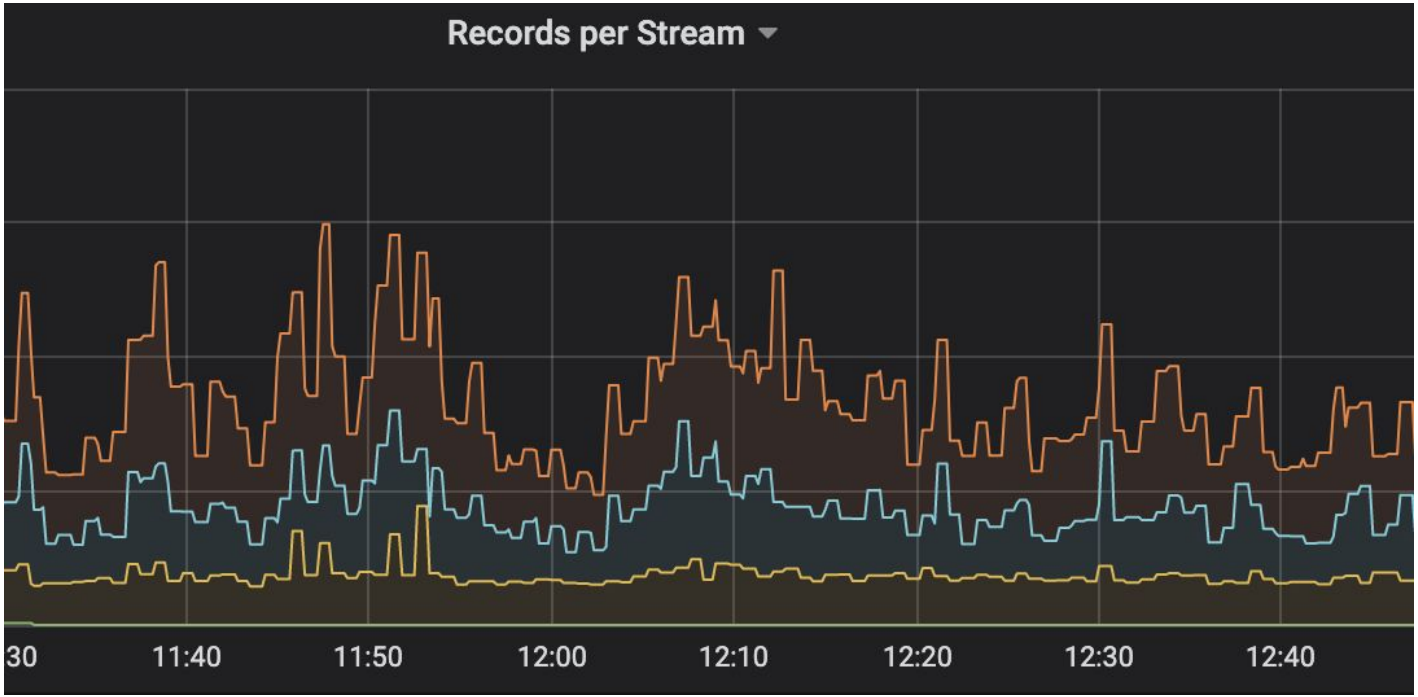
```
/**
  * Takes a topic object and writes the number of records for said topic this batch to Influx.
  *
  * @param topic A topic object within a Batch's StreamIdToInputInfo
  */
def writeTopicCountToInflux(topic: Tuple2[Int, StreamInputInfo]): Unit = {

  // store the individual record count for this topic
  val numRecords = topic._2.numRecords

  // store topicName
  val topicName = topic._2.metadata("offsets").asInstanceOf[List[OffsetRange]].head.topic

  // write record count for this topic this batch
  influx.write(influxDB, influxMeasurement, Seq(), Seq(("numRecords_" + topicName, numRecords)))
}
```

writeTopicCountToInflux()



Grafana dashboard reading from an Influx time-series database.

Streaming Listeners: Add to Context

Finally, instantiate your listener within an application.

You can customize with any args you need. By default it needs none.

Use the *addStreamingListener()* method of your Spark Streaming Context! Done!

```
val sparkConf = new SparkConf()
  .setAppName("My Spark App")
  .setMaster("local[*]")

val ssc = new StreamingContext(sparkConf, Seconds(30))

val listener = new SpotXSparkStreamingListener(
  "influx.yourcompany.com",
  "my_influx_database",
  "my_measurement",
  "myrdbms.yourcompany.com",
  "my_rdbms_database",
  "my_rdbms_table",
  "my_kafka_group_id",
  "my_rdbms_user",
  "my_rdbms_pwd")

ssc.addStreamingListener(listener)
```

The Possibilities are numerous!

The *Developer API* has many other data points you can build great monitoring from, such as:

- StreamingJobProgressListener
  - waitingBatches
  - runningBatches
  - numReceivers
  - lastCompletedBatch
  - **Just to name a few!**

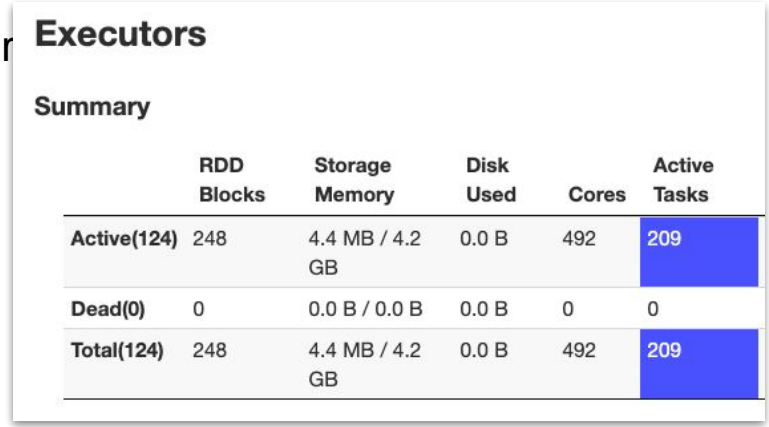
Method and Description
`batchDuration()`
`lastCompletedBatch()`
`lastReceivedBatch()`
`lastReceivedBatchRecords()`
`numReceivers()`
`numTotalCompletedBatches()`
`numTotalProcessedRecords()`
`numTotalReceivedRecords()`
`numUnprocessedBatches()`
`onBatchCompleted(StreamingListenerBatchCompleted batchCompleted)` Called when processing of a batch of jobs has completed.
`onBatchStarted(StreamingListenerBatchStarted batchStarted)` Called when processing of a batch of jobs has started.
`onBatchSubmitted(StreamingListenerBatchSubmitted batchSubmitted)` Called when a batch of jobs has been submitted for processing.
`onReceiverError(StreamingListenerReceiverError receiverError)` Called when a receiver has reported an error
`onReceiverStarted(StreamingListenerReceiverStarted receiverStarted)` Called when a receiver has been started
`onReceiverStopped(StreamingListenerReceiverStopped receiverStopped)` Called when a receiver has been stopped
`processingDelayDistribution()`
`receivedRecordsDistributions()`
`receiverInfo(int receiverId)`
`retainedCompletedBatches()`
`runningBatches()`
`schedulingDelayDistribution()`
`totalDelayDistribution()`
`waitingBatches()`

A Word About Batch Listeners

Allow the extension of non-streaming events and trigger for each application, job, stage, and executor.

Valuable usage:

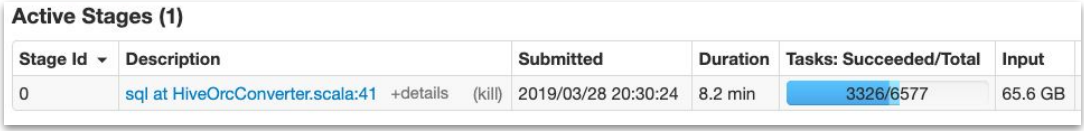
- Write statistics for runtimes of applications (given appId or appName, both available)
- Write info about executor removal
  - A reason is provided
- Write each Job duration, identifying long running jobs
  - Find potential skew
  - Find potential problem nodes



Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks
Active(124)	248	4.4 MB / 4.2 GB	0.0 B	492	209
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0
Total(124)	248	4.4 MB / 4.2 GB	0.0 B	492	209



Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
0	sql at HiveOrcConverter.scala:41 +details (kill)	2019/03/28 20:30:24	8.2 min	3326/6577	65.6 GB

Batch Listeners: *SparkListener* Abstract Class

Similar to streaming, everything you'll need is in the *SparkListener,* but this time in an *Abstract Class* within the Developer API.

However, it has *18* convenience methods for overriding.

We find

***onApplicationStart()*,**

***onApplicationEnd()*, and**

onExecutorRemoved()

to be the most useful.

```scala
/**
 * :: DeveloperApi ::
 * A default implementation for [[SparkListenerInterface]] that has no-op implementations for
 * all callbacks.
 *
 * Note that this is an internal interface which might change in different Spark releases.
 */
@DeveloperApi
abstract class SparkListener extends SparkListenerInterface {
  override def onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit = { }

  override def onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit = { }

  override def onTaskStart(taskStart: SparkListenerTaskStart): Unit = { }

  override def onTaskGettingResult(taskGettingResult: SparkListenerTaskGettingResult): Unit = { }

  override def onTaskEnd(taskEnd: SparkListenerTaskEnd): Unit = { }

  override def onJobStart(jobStart: SparkListenerJobStart): Unit = { }

  override def onJobEnd(jobEnd: SparkListenerJobEnd): Unit = { }

  override def onEnvironmentUpdate(environmentUpdate: SparkListenerEnvironmentUpdate): Unit = { }

  override def onBlockManagerAdded(blockManagerAdded: SparkListenerBlockManagerAdded): Unit = { }

  override def onBlockManagerRemoved(
      blockManagerRemoved: SparkListenerBlockManagerRemoved): Unit = { }

  override def onUnpersistRDD(unpersistRDD: SparkListenerUnpersistRDD): Unit = { }

  override def onApplicationStart(applicationStart: SparkListenerApplicationStart): Unit = { }

  override def onApplicationEnd(applicationEnd: SparkListenerApplicationEnd): Unit = { }

  override def onExecutorMetricsUpdate(
      executorMetricsUpdate: SparkListenerExecutorMetricsUpdate): Unit = { }

  override def onExecutorAdded(executorAdded: SparkListenerExecutorAdded): Unit = { }

  override def onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit = { }

  override def onBlockUpdated(blockUpdated: SparkListenerBlockUpdated): Unit = { }

  override def onOtherEvent(event: SparkListenerEvent): Unit = { }
}
```

Batch Examples

```
class BatchListenerUtils(
                              measurementName: String,
                              database: String) extends SparkListener {

    override def onApplicationStart(applicationStart: SparkListenerApplicationStart): Unit = {
                influxWriter.write(myTimeSeriesTS,
                                   applicationStart.appName,
                                   applicationStart.time)
    }

    override def onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit = {
                influxWriter.write(myTimeSeriesTS,
                                   executorRemoved.reason)
    }
}
```

Review

Monitoring spark apps is tough!

But we found a solution: **Spark Listeners!**

Streaming

Batch

Both provide actionable data:

- **Performance metrics**
- **Data volume**


SPARK+AI
SUMMIT 2019
Q & A
#UnifiedAnalytics #SparkAISummit

Code!

Blog: **hadoopsters.dev**

Github: **gist.github.com/hadoopsters**

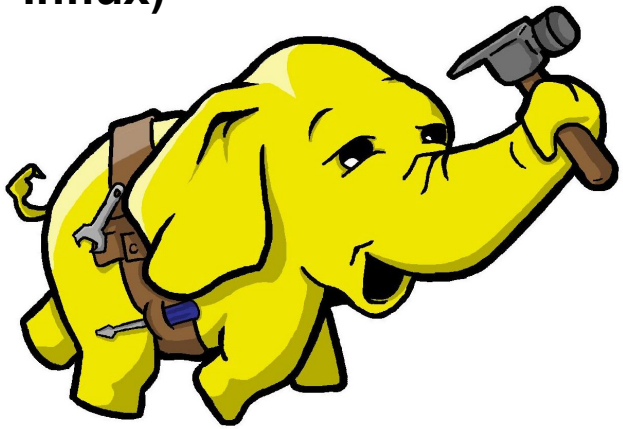
- **ExampleStreamingListener.scala (Generic implementation)**
- **SpotXSparkStreamingListener.scala (MySQL + Influx)**

Landon Robinson

- **lrobinson@spotx.tv**

Ben Storrie

- **bstorrie@spotx.tv**



SPARK+AI SUMMIT 2019

DON'T FORGET TO RATE AND REVIEW THE SESSIONS

SEARCH SPARK + AI SUMMIT

Download on the App Store

GET IT ON Google Play