Productionizing a 24/7 Spark Streaming Service on YARN

Issac Buenrostro, Arup Malakar

Spark Summit 2014
July 1, 2014

About Ooyala

Cross-device video analytics and monetization products and services

---

Founded in 2007

300+ employees worldwide

Global footprint of 200M unique users in 130 countries

Ooyala works with the most successful broadcasts and media companies in the world

Reach, measure, monetize video business

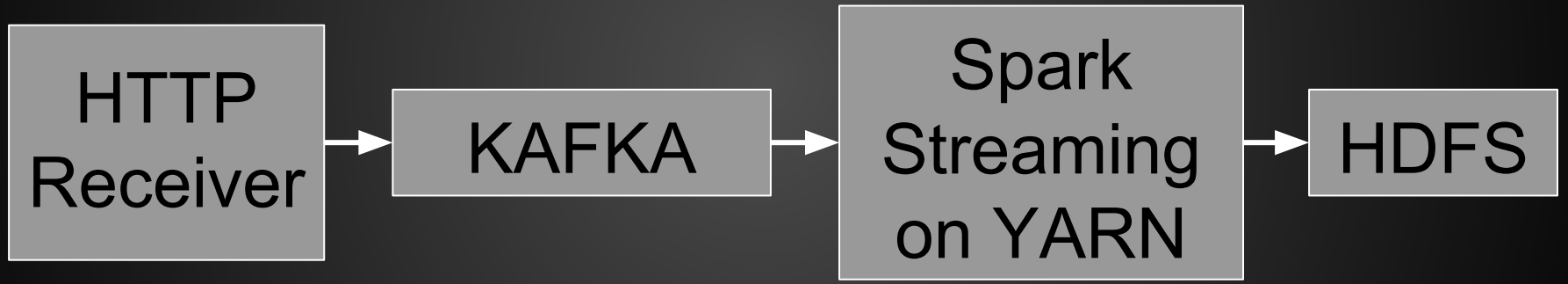
Ooyala Analytics ETL

1. Ingest and process 5B+ events/day
2. Decode -> Enrich -> Persist

Plays/Displays, Conversion Rate, Geo, Device, OS, Unique, Engagement, etc.

Run 24/7, <2 min latency, <1% data loss, reduce COGS, minimal human interaction.

Architecture diagram



Take-Home Message

Yes, there are issues with Spark Streaming

However:

- Allows flexibility not available in other frameworks
- Issues are not huge, working on resolving them
- Awareness / best practices

Hybrid Processing

Batch

- Easy reprocessing
- More historical awareness
- Consistency guarantees

Streaming

- Real time
- Lower scheduling needs

Why Spark?

One codebase to maintain and test

Same framework for Batch and Streaming

Easily maintainable cluster (vs Storm)

Out-of-the-box YARN integration

Why YARN?

- Eliminate operational cost of maintaining a separate spark cluster
- Better H/W utilization
- Easy upgrade
  - 0.9 to 0.9.1 upgrade just a dependency change
  - vs upgrading a dedicated cluster
- Leverage the expertise dealing with YARN/hadoop

YARN Pitfalls

- 384 MB overhead per container
- Disconnect between Spark/YARN UI
- YARN classpath automatically added to Spark

Outline

1. Launching streaming applications
    a. 24/7 running
2. Instrumentation
    a. tracking of workers
3. Consuming streaming data
    a. fixing consumption speed/reliability issues
4. Logging
    a. Reducing amount of logs
5. Persisting streaming data

Launching in YARN

Traditional launching (Spark 0.9.1):

```
export SPARK_JAR=/usr/lib/spark-assembly-1.1.0-SNAPSHOT-hadoop2.2.0.jar
java -cp /etc/hadoop/conf:AppJar.jar:spark-assembly.jar org.apache.spark.
deploy.yarn.Client --jar AppJar.jar --addJars /jars/config.jar --class ooyala.app.
MainClass --arg arg1 --arg arg2 --name MyApp
```

- Launches application in YARN
- Tracks application until done (not great for streaming)

Launching - Streaming way

Streaming launching:

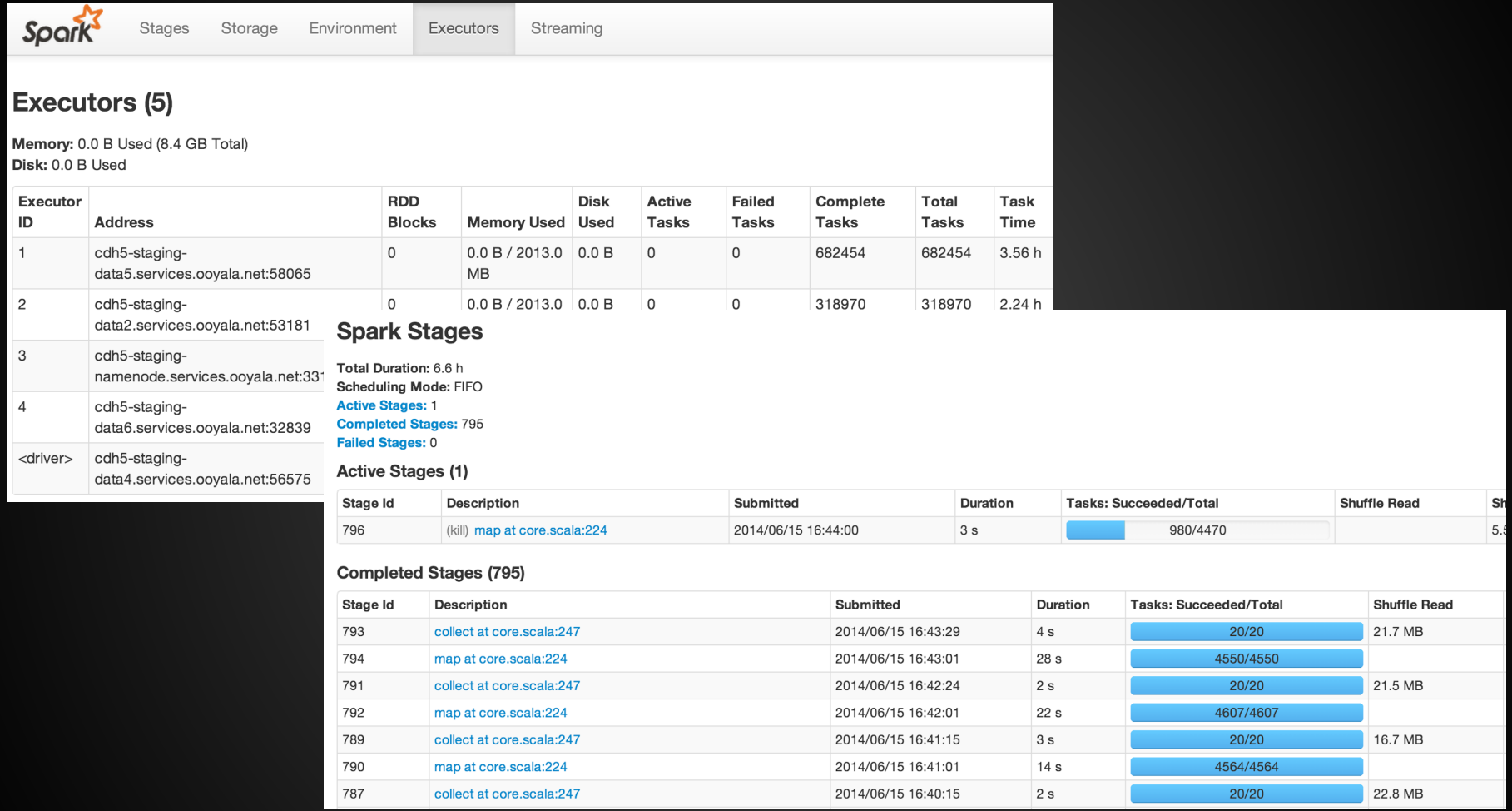
```
java -cp /etc/hadoop/conf:AppJar.jar:spark-assembly.jar org.apache.spark.yarn.StreamingClient --jar AppJar.jar --addJars /jars/config.jar --class ooyala.app.MainClass --arg arg1 --arg arg2 --name MyApp
```

- Launch application IF not running
- Connect to old application if already exists
- Tracks application until done
- Does not kill application if crashed

Launching - Relaunching

Use upstart with Streaming launcher

- Launch application from remote host
- If launcher fails, application unaffected
- On launcher start, connect to existing app
- If application fails, upstream re-launches it


Spark
Stages
Storage
Environment
Executors
Streaming
Executors (5)
Memory: 0.0 B Used (8.4 GB Total)
Disk: 0.0 B Used
Executor ID | Address | RDD Blocks | Memory Used | Disk Used | Active Tasks | Failed Tasks | Complete Tasks | Total Tasks | Task Time
1 | cdh5-staging-data5.services.ooyala.net:58065 | 0 | 0.0 B / 2013.0 MB | 0.0 B | 0 | 0 | 682454 | 682454 | 3.56 h
2 | cdh5-staging-data2.services.ooyala.net:53181 | 0 | 0.0 B / 2013.0 | 0.0 B | 0 | 0 | 318970 | 318970 | 2.24 h
3 | cdh5-staging-namenode.services.ooyala.net:331
4 | cdh5-staging-data6.services.ooyala.net:32839
<driver> | cdh5-staging-data4.services.ooyala.net:56575
Spark Stages
Total Duration: 6.6 h
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 795
Failed Stages: 0
Active Stages (1)
Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Shuffle Read
796 | (kill) map at core.scala:224 | 2014/06/15 16:44:00 | 3 s | 980/4470
Completed Stages (795)
Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Shuffle Read
793 | collect at core.scala:247 | 2014/06/15 16:43:29 | 4 s | 20/20 | 21.7 MB
794 | map at core.scala:224 | 2014/06/15 16:43:01 | 28 s | 4550/4550
791 | collect at core.scala:247 | 2014/06/15 16:42:24 | 2 s | 20/20 | 21.5 MB
792 | map at core.scala:224 | 2014/06/15 16:42:01 | 22 s | 4607/4607
789 | collect at core.scala:247 | 2014/06/15 16:41:15 | 3 s | 20/20 | 16.7 MB
790 | map at core.scala:224 | 2014/06/15 16:41:01 | 14 s | 4564/4564
787 | collect at core.scala:247 | 2014/06/15 16:40:15 | 2 s | 20/20 | 22.8 MB

Streaming

Started at: Sun Jun 15 10:07:31 UTC 2014
Time since start: 6 hours 38 minutes 40 seconds
Network receivers: 50
Batch interval: 1 minute
Processed batches: 398
Waiting batches: 1

Statistics over last 100 processed batches

Receiver Statistics

Receiver	Status	Location	Records in last batch [2014/06/15 16:46:12]	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
KafkaReceiver-0	ACTIVE	cdh5-staging-namenode.services.ooyala.net	1158	15	20	26	-
KafkaReceiver-1	ACTIVE	cdh5-staging-namenode.services.ooyala.net	1304	13	18	25	-
KafkaReceiver-2	ACTIVE	cdh5-staging-namenode.services.ooyala.net	1284	15	21	28	-
KafkaReceiver-3	ACTIVE	cdh5-staging-data5.services.ooyala.net	1035	15	22	28	-
KafkaReceiver-4	ACTIVE	cdh5-staging-data5.services.ooyala.net	1320	16	20	26	-

Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	23 seconds 388 ms	9 seconds 3 ms	11 seconds 389 ms	13 seconds 340 ms	15 seconds 923 ms	36 seconds 302 ms
Scheduling Delay	0 ms	0 ms	0 ms	0 ms	1 ms	4 ms
Total Delay	23 seconds 388 ms	9 seconds 3 ms	11 seconds 389 ms	13 seconds 341 ms	15 seconds 923 ms	36 seconds 303 ms

Failure Cases

Processing stops, consuming continues

Broadcast variables get GCed (SPARK-1697)

Consumers fail silently (SPARK-1975)

Out of Memory if consuming too fast (SPARK-1341)

Instrumentation - Custom Sources

Codahale Metrics

```
val source = new Source {
    override def sourceName = "ooyala.example.metrics"
    override def metricRegistry = new MetricRegistry
    val numReqMeter = metricRegistry.meter("numRequests")
}
SparkEnv.get.metricsSystem.registerSource(source)
source.numReqMeter.mark()
```

Instrumentation

```
// Create an instrumentation
val instrumentation = new SparkInstrumentation("ooyala.examples.metrics")
// Create an accumulator
val numReqs = sc.accumulator(0L)
// Register accumulator in the instrumentation
instrumentation.source.registerDailyAccumulator(numReqs, "numReqs")
// Register the instrumentation with Spark
// Must be done after all accumulators have already been registered
instrumentation.register()
```

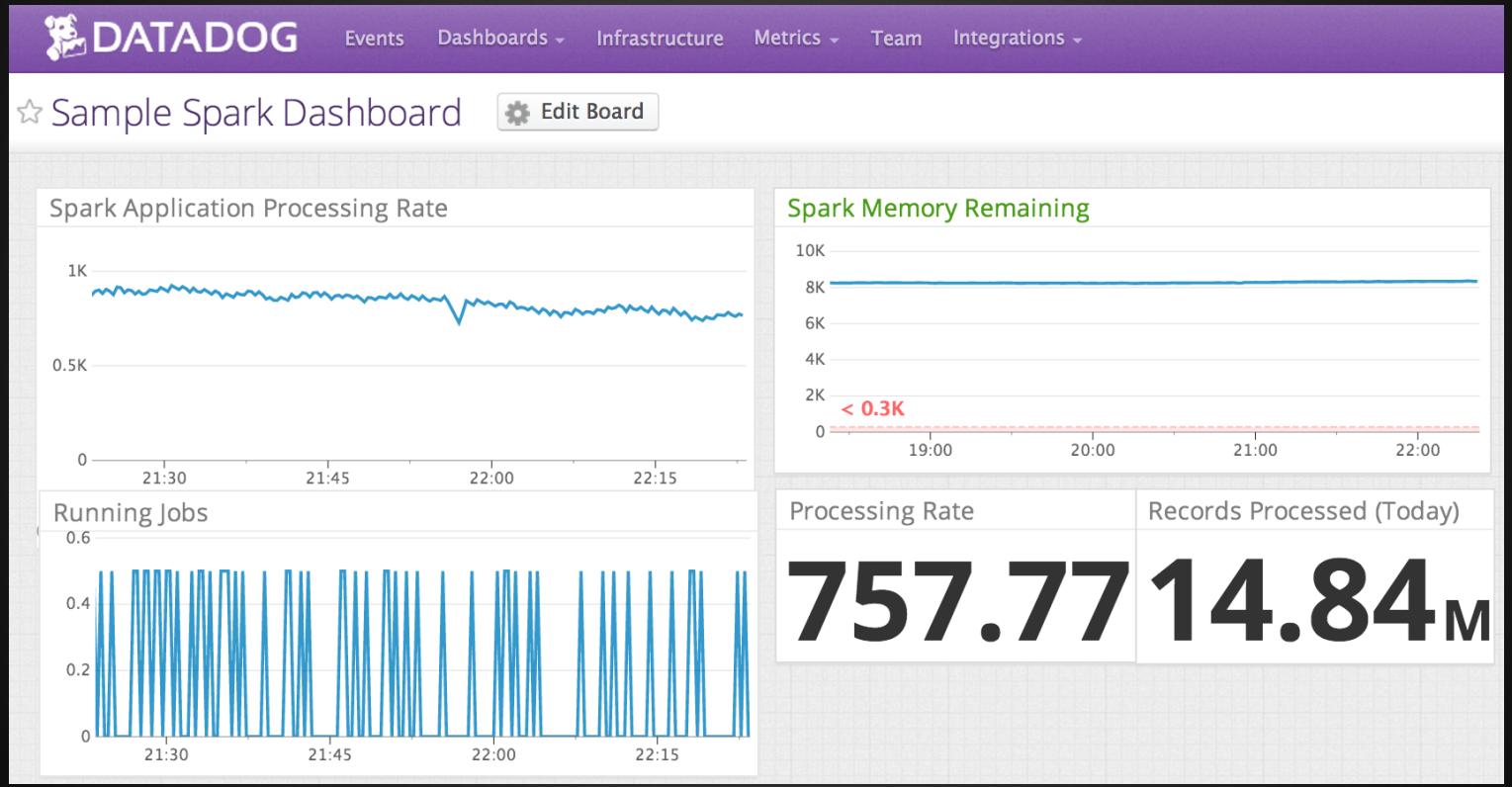
Instrumentation - Custom Sinks

```
class DatadogSink(val property: Properties, val registry: MetricRegistry) extends
Sink {
  val DD_API_KEY = "apikey"
  val apikey = propertyToOption(DD_API_KEY).get

  val reporter = DatadogReporter.forRegistry(registry)
    .convertRatesTo(TimeUnit.SECONDS)
    .convertDurationsTo(TimeUnit.MILLISECONDS)
    .build(transport, host)

  override def start() { reporter.start(pollPeriod, pollUnit) }
  override def stop() { reporter.stop() }
}
```

Instrumentation - Example



Consuming Data - Docs Way

```
val conf = new SparkConf().setAppName(name).setMaster(sparkMaster)

val sc = new SparkContext(conf)
val ssc = new StreamingContext(sc, Seconds(60))

val dStream =
  KafkaUtils.createStream(ssc, zkConnect, consumerGroup, Map(topic -> 10),
StorageLevel.MEMORY_ONLY).map( x => x._2)
```

Consuming Data - Issues

All consumers in same node

Consumption could be too fast

Consumption independent of processing

(1.0.0) ReceiverTracker stuck (SPARK-1975)

Consuming Data - Fixes

```
val conf = new SparkConf().setAppName(name).setMaster(sparkMaster)
  .set("spark.streaming.receiver.maxRate", maxRate.toString) // throttling

val sc = new SparkContext(conf)
val ssc = new StreamingContext(sc, Seconds(60))

val kafkaInputs = (1 to numPartitions).map { _ =>
  KafkaUtils.createStream(ssc, zkConnect, consumerGroup, Map(topic -> 1),
    StorageLevel.MEMORY_ONLY).map( x => x._2)
}
val dStream = ssc.union(kafkaInputs)
```

Consuming Data - Data Loss

Streaming data replicated across nodes

- Up to one batch lost on crash
- Data loss if tasks hang
- Kafka autocommit may prevent recovery

Disable autocommit? Keep track of finished offsets?

Logging - Where to Find

- YARN makes log available in HDFS for old jobs
- Logs of running jobs are available in the YARN UI.
- Spark should link to hadoop log UI from SPARK UI
- SPARK_LOG4J_CONF makes it easy to set custom log4j configuration file; maybe conflicts with YARN cp (SPARK-2007)

Logging - Reduced Logs

- Too much logging => slow application, logs hard to interpret
- Set log level to **WARN** for certain spark packages, set additivity to false
  - log4j.logger.org.apache.spark.rdd.NewHadoopRDD=WARN, <appender>
  - log4j.additivity.org.apache.spark.rdd.NewHadoopRDD=false
- Recommended for:
  - log4j.logger.org.apache.spark.{storage, scheduler, CacheTracker, CacheTrackerActor, MapOutputTrackerActor, MapOutputTracker, executor}

Streaming Outputs

Atomic writing ensures consistency (temp + move)

Fine control of output using forEachRdd, mapPartitions

CompletionIterator provides finer control on file closing

e.g. Ooyala ETL uses columnar parquet files in HDFS

Conclusion

- Real-time pipeline required for low latency
- Message drops hard to avoid / reprocessing is hard
- Hybrid model of real time followed by batch processing gives best of both worlds
- Instrumentation is key to a healthy real-time pipeline
- Logging needs major fixing

QUESTIONS

Issac Buenrostro - buenrostro@ooyala.com

Arup Malakar - arup@ooyala.com

Code samples at:
https://gist.github.com/9b94736c2bad2f4b8e23.git

What is Spark?

Open-source data analytics cluster computing framework

- In-memory distributed computing
- MapReduce framework with friendly API
- Integrable with Hadoop/YARN
- Includes Shark(SQL), Streaming, MLib, GraphX

What is YARN?

Resource management platform for Hadoop clusters

- Part of hadoop since v2.0
- Hadoop map/reduce runs on top of YARN
- Allows for running of custom applications
- Integration with Spark, Storm

Who are we?

Analytics ETL development team at Ooyala

- Video Displays, Plays
- Conversion rate
- Viewership
  - Unique viewers
  - Geographic, Devices, Operating System, Browser
  - Engagement (Which segment of the video is more popular etc)
  - Advertising analytics
- Social