Long-running Spark Streaming Jobs on YARN Cluster

Sep 30th, 2016

A long-running Spark Streaming job, once submitted to the YARN cluster should run forever until it is intentionally stopped. Any interruption introduces substantial processing delays and could lead to data loss or duplicates. Neither YARN nor Apache Spark have been designed for executing long-running services. But they have been successfully adapted to growing needs of near real-time processing implemented as long-running jobs. Successfully does not necessarily mean without technological challenges.

This blog post summarizes my experiences in running mission critical, long-running Spark Streaming jobs on a secured YARN cluster. You will learn how to submit Spark Streaming application to a YARN cluster to avoid sleepless nights during on-call hours.

Fault tolerance

In the YARN cluster mode Spark driver runs in the same container as the Application Master, the first YARN container allocated by the application. This process is responsible for driving the application and requesting resources (Spark executors) from YARN. What is important, Application Master eliminates need for any another process that run during application lifecycle. Even if an edge Hadoop cluster node where the Spark Streaming job was submitted fails, the application stays unaffected.

To run Spark Streaming application in the cluster mode, ensure that the following parameters are given to spark-submit command:

```
1 spark-submit --master yarn --deploy-mode cluster
```

Because Spark driver and Application Master share a single JVM, any error in Spark driver stops our long-running job. Fortunately it is possible to configure maximum number of attempts that will be made to re-run the application. It is reasonable to set higher value than default 2 (derived from YARN cluster property `yarn.resourcemanager.am.max-attempts`). For me 4 works quite well, higher value may cause unnecessary restarts even if the reason of the failure is permanent.

```
1 spark-submit --master yarn --deploy-mode cluster \
2     --conf spark.yarn.maxAppAttempts=4
```

If the application runs for days or weeks without restart or redeployment on highly utilized cluster, 4 attempts could be exhausted in few hours. To avoid this situation, the attempt counter should be reset on every hour of so.

```
1 spark-submit --master yarn --deploy-mode cluster \
2     --conf spark.yarn.maxAppAttempts=4 \
3     --conf spark.yarn.am.attemptFailuresValidityInterval=1h
```

Another important setting is a maximum number of executor failures before the application fails. By default it is `max(2 * num executors, 3)`, well suited for batch jobs but not for long-running jobs. The property comes with corresponding validity interval which also should be set.

```
1 spark-submit --master yarn --deploy-mode cluster \
2     --conf spark.yarn.maxAppAttempts=4 \
3     --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
4     --conf spark.yarn.max.executor.failures={8 * num_executors} \
5     --conf spark.yarn.executor.failuresValidityInterval=1h
```

For long-running jobs you could also consider to boost maximum number of task failures before giving up the job. By default tasks will be retried 4 times and then job fails.

```
spark-submit --master yarn --deploy-mode cluster \
    --conf spark.yarn.maxAppAttempts=4 \
    --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
    --conf spark.yarn.max.executor.failures={8 * num_executors} \
    --conf spark.yarn.executor.failuresValidityInterval=1h \
    --conf spark.task.maxFailures=8
```

Performance

When a Spark Streaming application is submitted to the cluster, YARN queue where the job runs must be defined. I strongly recommend using YARN Capacity Scheduler and submitting long-running jobs to separate queue. Without a separate YARN queue your long-running job will be preempted by a massive Hive query sooner or later.

```
spark-submit --master yarn --deploy-mode cluster \
    --conf spark.yarn.maxAppAttempts=4 \
    --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
    --conf spark.yarn.max.executor.failures={8 * num_executors} \
    --conf spark.yarn.executor.failuresValidityInterval=1h \
    --conf spark.task.maxFailures=8 \
    --queue realtime_queue
```

Another important issue for Spark Streaming job is keeping processing time stable and highly predictable. Processing time should stay below batch duration to avoid delays. I've found that Spark speculative execution helps a lot, especially on a busy cluster. Batch processing times are much more stable when speculative execution is enabled. Unfortunately speculative mode can be enabled only if Spark actions are idempotent.

```
spark-submit --master yarn --deploy-mode cluster \
    --conf spark.yarn.maxAppAttempts=4 \
    --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
    --conf spark.yarn.max.executor.failures={8 * num_executors} \
    --conf spark.yarn.executor.failuresValidityInterval=1h \
    --conf spark.task.maxFailures=8 \
    --queue realtime_queue \
    --conf spark.speculation=true
```

Security

On a secured HDFS cluster, long-running Spark Streaming jobs fails due to Kerberos ticket expiration. Without additional settings, Kerberos ticket is issued when Spark Streaming job is submitted to the cluster. When ticket expires Spark Streaming job is not able to write or read data from HDFS anymore.

In theory (based on documentation) it should be enough to pass Kerberos principal and keytab as spark-submit command:

```
 spark-submit --master yarn --deploy-mode cluster \
      --conf spark.yarn.maxAppAttempts=4 \
      --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
      --conf spark.yarn.max.executor.failures={8 * num_executors} \
      --conf spark.yarn.executor.failuresValidityInterval=1h \
      --conf spark.task.maxFailures=8 \
      --queue realtime_queue \
      --conf spark.speculation=true \
```

```
9       --principal user/hostname@domain \
10      --keytab /path/to/foo.keytab
```

In practice, due to several bugs (HDFS-9276, SPARK-11182) HDFS cache must be disabled. If not, Spark will not be able to read updated token from file on HDFS.

```
1  spark-submit --master yarn --deploy-mode cluster \
2       --conf spark.yarn.maxAppAttempts=4 \
3       --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
4       --conf spark.yarn.max.executor.failures={8 * num_executors} \
5       --conf spark.yarn.executor.failuresValidityInterval=1h \
6       --conf spark.task.maxFailures=8 \
7       --queue realtime_queue \
8       --conf spark.speculation=true \
9       --principal user/hostname@domain \
10      --keytab /path/to/foo.keytab \
11      --conf spark.hadoop.fs.hdfs.impl.disable.cache=true
```

Mark Grover pointed out that those bugs only affect HDFS cluster configured with NameNodes in HA mode. Thanks, Mark.

Logging

The easiest way to access Spark application logs is to configure Log4j console appender, wait for application termination and use `yarn logs -applicationId [applicationId]` command. Unfortunately it is not feasible to terminate long-running Spark Streaming jobs to access the logs.

I recommend to install and configure Elastic, Logstash and Kibana (ELK stack). ELK installation and configuration is out of this blog post scope, but remember to log the following context fields:

- YARN application id
- YARN container hostname
- Executor id (Spark driver is always 000001, Spark executors start from 000002)
- YARN attempt (to check how many times Spark driver has been restarted)

Log4j configuration with Logstash specific appender and layout definition should be passed to spark-submit command:

```
1  spark-submit --master yarn --deploy-mode cluster \
2       --conf spark.yarn.maxAppAttempts=4 \
3       --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
4       --conf spark.yarn.max.executor.failures={8 * num_executors} \
5       --conf spark.yarn.executor.failuresValidityInterval=1h \
6       --conf spark.task.maxFailures=8 \
7       --queue realtime_queue \
8       --conf spark.speculation=true \
9       --principal user/hostname@domain \
10      --keytab /path/to/foo.keytab \
11      --conf spark.hadoop.fs.hdfs.impl.disable.cache=true \
12      --conf spark.driver.extraJavaOptions=-Dlog4j.configuration=file:log4j.properties \
13      --conf spark.executor.extraJavaOptions=-Dlog4j.configuration=file:log4j.properties \
14      --files /path/to/log4j.properties
```

Finally Kibana dashboard for Spark Job might look like:

@timestamp	level	message	application-id	source-host	executor-id	attempt-id
2016-09-29T18:53:56.094Z	ERROR	Exception in task 0.1 in stage 499.0 ...	application_1472192936433_1131032	s42055.dc4.local	000009	01
2016-09-29T18:53:56.083Z	ERROR	Aborting task	application_1472192936433_1131032	s42055.dc4.local	000009	01
2016-09-29T18:25:09.314Z	ERROR	Exception in task 1.1 in stage 448.0 ...	application_1472192936433_1131032	s42235.dc4.local	000004	01
2016-09-29T18:25:09.309Z	ERROR	Aborting task	application_1472192936433_1131032	s42235.dc4.local	000004	01
2016-09-29T14:43:51.837Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s41764.dc4.local	000009	01
2016-09-29T14:43:51.462Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s42049.dc4.local	000003	01
2016-09-29T14:43:50.486Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s41290.dc4.local	000007	01
2016-09-29T14:43:50.279Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s42243.dc4.local	000006	01
2016-09-29T14:43:50.246Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s41729.dc4.local	000005	01
2016-09-29T14:43:50.242Z	ERROR	Error in removing shuffle 18	application_1472192936433_1130086	s42243.dc4.local	000006	01
2016-09-29T14:43:50.231Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s42240.dc4.local	000002	01
2016-09-29T14:43:50.207Z	ERROR	Driver 10.68.150.32:41727 disassociat...	application_1472192936433_1130086	s42242.dc4.local	000004	01

Monitoring

Long running job runs 24/7 so it is important to have an insight into historical metrics. Spark UI keeps statistics only for limited number of batches, and after restart all metrics are gone. Again, external tools are needed. I recommend to install Graphite for collecting metrics and Grafana for building dashboards.

First, Spark needs to be configured to report metrics into Graphite, prepare the `metrics.properties` file:

```
1 *.sink.graphite.class=org.apache.spark.metrics.sink.GraphiteSink
2 *.sink.graphite.host=[hostname]
3 *.sink.graphite.port=[port]
4 *.sink.graphite.prefix=some_meaningful_name
5
6 driver.source.jvm.class=org.apache.spark.metrics.source.JvmSource
7 executor.source.jvm.class=org.apache.spark.metrics.source.JvmSource
```

And configure spark-submit command:

```
1  spark-submit --master yarn --deploy-mode cluster \
2       --conf spark.yarn.maxAppAttempts=4 \
3       --conf spark.yarn.am.attemptFailuresValidityInterval=1h \
4       --conf spark.yarn.max.executor.failures={8 * num_executors} \
5       --conf spark.yarn.executor.failuresValidityInterval=1h \
6       --conf spark.task.maxFailures=8 \
7       --queue realtime_queue \
8       --conf spark.speculation=true \
9       --principal user/hostname@domain \
10      --keytab /path/to/foo.keytab \
11      --conf spark.hadoop.fs.hdfs.impl.disable.cache=true \
12      --conf spark.driver.extraJavaOptions=-Dlog4j.configuration=file:log4j.properties \
13      --conf spark.executor.extraJavaOptions=-Dlog4j.configuration=file:log4j.properties \
14      --files /path/to/log4j.properties:/path/to/metrics.properties
```

Metrics

Spark publishes tons of metrics from driver and executors. If I were to choose the most important one, it would be the last received batch records. When `StreamingMetrics.streaming.lastReceivedBatch_records == 0` it probably means that Spark Streaming job has been stopped or failed.

Other important metrics are listed below:

- When total delay is greater than batch interval, latency of the processing pipeline increases.

```
1 driver.StreamingMetrics.streaming.lastCompletedBatch_totalDelay
```

- When number of active tasks is lower than `number of executors * number of cores`, allocated YARN resources are not fully utilized.

```
1 executor.threadpool.activeTasks
```

- How much RAM is used for RDD cache.

```
1 driver.BlockManager.memory.memUsed_MB
```

- When there is not enough RAM for RDD cache, how much data has been spilled to disk. You should increase executor memory or change `spark.memory.fraction` Spark property to avoid performance degradation.

```
1 driver.BlockManager.disk.diskSpaceUsed_MB
```

- What is JVM memory utilization on Spark driver.

```
1 driver.jvm.heap.used
2 driver.jvm.non-heap.used
3 driver.jvm.pools.G1-Old-Gen.used
4 driver.jvm.pools.G1-Eden-Space.used
5 driver.jvm.pools.G1-Survivor-Space.used
```

- How much time is spent on GC on Spark driver.

```
1 driver.jvm.G1-Old-Generation.time
2 driver.jvm.G1-Young-Generation.time
```

- What is JMV memory utilization on Spark executors.

```
1 [0-9]*.jvm.heap.used
2 [0-9]*.jvm.non-heap.used
3 [0-9]*.jvm.pools.G1-Old-Gen.used
4 [0-9]*.jvm.pools.G1-Survivor-Space.used
5 [0-9]*.jvm.pools.G1-Eden-Space.used
```

- How much time is spent on GC on Spark executors.

```
1 [0-9]*.jvm.G1-Old-Generation.time
2 [0-9]*.jvm.G1-Young-Generation.time
```

Grafana

While you configure first Grafana dashboard for Spark application, the first problem pops up:

> How to configure Graphite query when metrics for every Spark application run are reported under its own application id?

If you are lucky and brave enough to use Spark 2.1, pin the application metric into static application name:

```
1 --conf spark.metrics.namespace=my_application_name
```

For Spark versions older than 2.1, a few tricks with Graphite built-in functions are needed.

Driver metrics use wildcard `.*(application_[0-9]+).*` and `aliasSub` Graphite function to present 'application id' as graph legend:

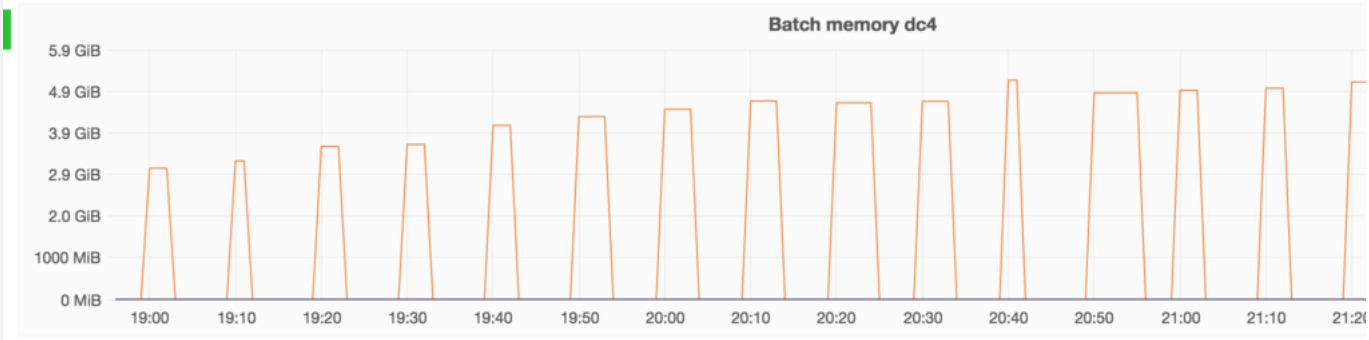
```
1 aliasSub(stats.analytics.$job_name.*.prod.$dc.*.driver.jvm.heap.used, ".*(application_[0-9]+).*
```

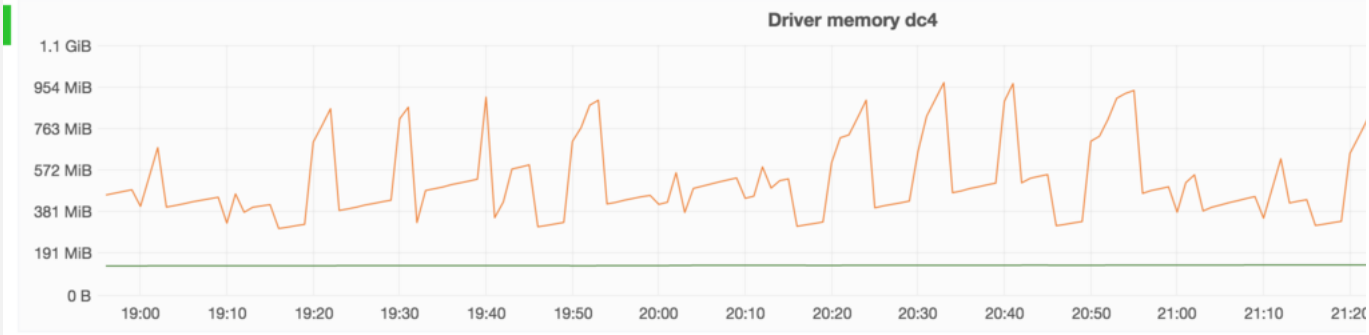
For executor metrics again use wildcard `.*(application_[0-9]+).*`, `groupByNode` Graphite function to sum metrics from all Spark executors and finally `aliasSub` Graphite function to present 'application id' as graph legend:

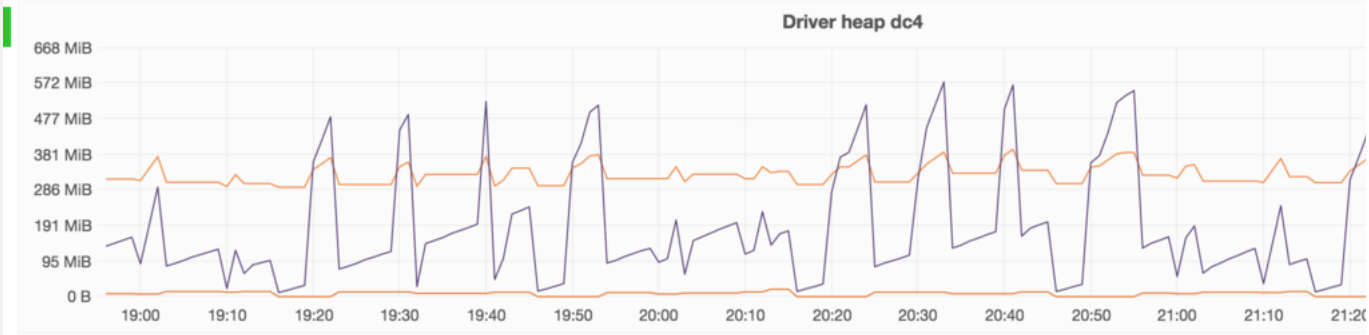
```
1 aliasSub(groupByNode(stats.analytics.$job_name.*.prod.$dc.*.[0-9]*.jvm.heap.used, 6, "sumSeries
```

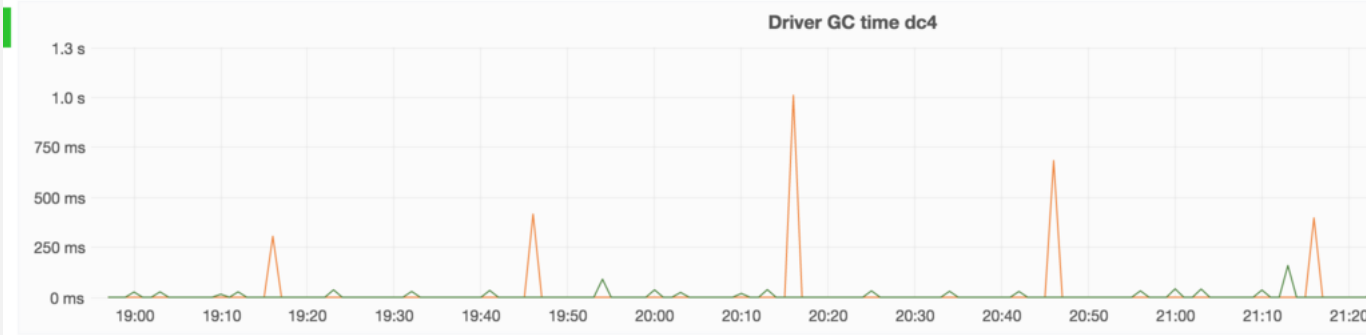
Finally Grafana dashboard for Spark Job might look like:

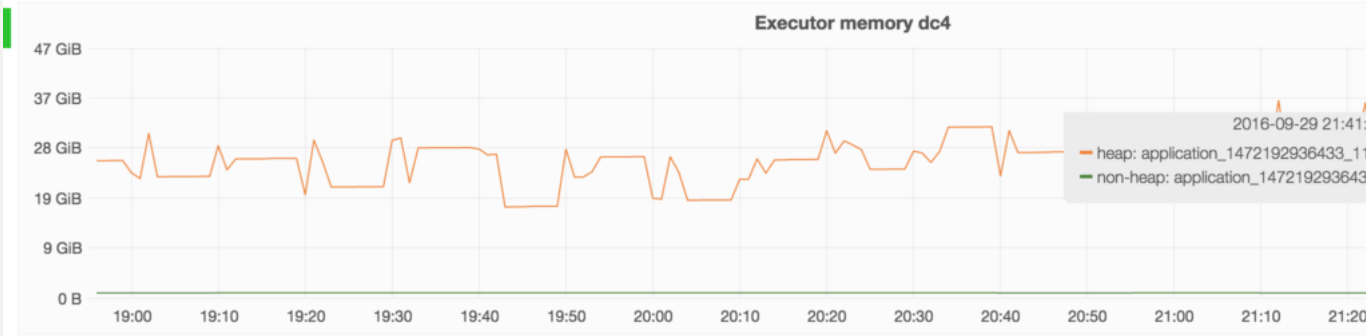

0
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10
21:20


Batch memory dc4
5.9 GiB
4.9 GiB
3.9 GiB
2.9 GiB
2.0 GiB
1000 MiB
0 MiB
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10

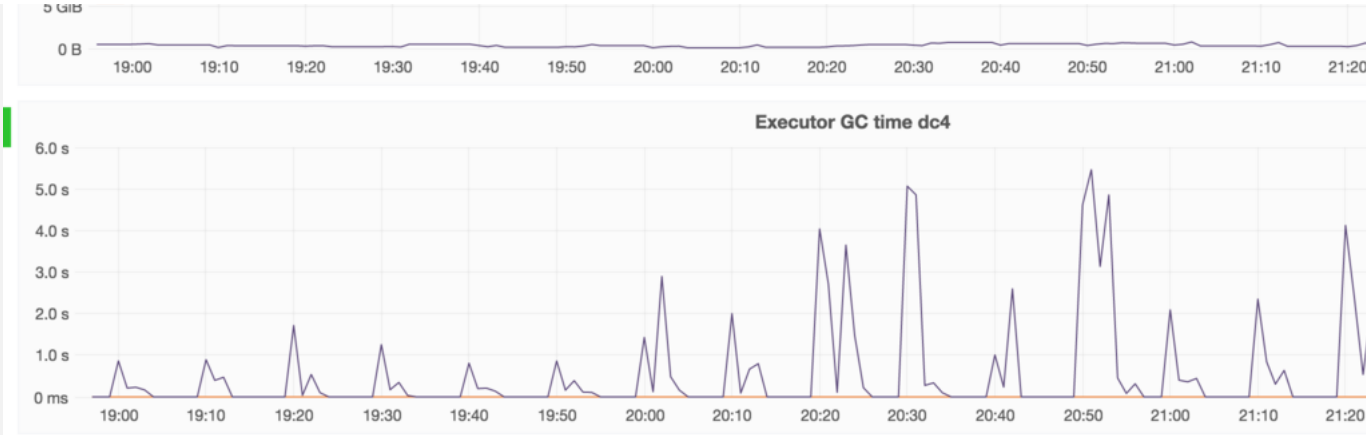

Driver memory dc4
1.1 GiB
954 MiB
763 MiB
572 MiB
381 MiB
191 MiB
0 B
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10


Driver heap dc4
668 MiB
572 MiB
477 MiB
381 MiB
286 MiB
191 MiB
95 MiB
0 B
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10


Driver GC time dc4
1.3 s
1.0 s
750 ms
500 ms
250 ms
0 ms
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10


Executor memory dc4
47 GiB
37 GiB
28 GiB
19 GiB
9 GiB
0 B
19:00
19:10
19:20
19:30
19:40
19:50
20:00
20:10
20:20
20:30
20:40
20:50
21:00
21:10


Executor heap dc4
28 GiB
23 GiB
19 GiB
14 GiB
9 GiB



If Spark application is restarted frequently, metrics for old, already finished runs should be deleted from Graphite. Because Graphite does not compact inactive metrics, old metrics slow down Graphite itself and Grafana queries.

Graceful stop

The last puzzle element is how to stop Spark Streaming application deployed on YARN in a graceful way. The standard method for stopping (or rather killing) YARN application is using a command `yarn application -kill [applicationId]`. And this command stops the Spark Streaming application but this could happen in the middle of a batch. So if the job reads data from Kafka, saves processing results on HDFS and finally commits Kafka offsets you should expect duplicated data on HDFS when job was stopped just before committing offsets.

The first attempt to solve graceful shutdown issue was to call Spark streaming context stop method in a shutdown hook.

```
1 sys.addShutdownHook {
2     streamingContext.stop(stopSparkContext = true, stopGracefully = true)
3 }
```

Disappointingly a shutdown hook is called too late to finish started batch and Spark application is killed almost immediately. Moreover there is no guarantee that a shutdown hook will be called by JVM at all.

At the time of writing this blog post the only confirmed way to shutdown gracefully Spark Streaming application on YARN is to notifying somehow the application about planned shutdown, and then stop streaming context programmatically (but not from shutdown hook). Command `yarn application -kill` should be used only as a last resort if notified application did not stop after defined timeout.

The application can be notified about planned shutdown using marker file on HDFS (the easiest way), or using simple Socket/HTTP endpoint exposed on the driver (sophisticated way).

Because I like KISS principle, below you can find shell script pseudo-code for starting / stopping Spark Streaming application using marker file:

```
1  start() {
2      hdfs dfs -touchz /path/to/marker/my_job_unique_name
3      spark-submit ...
4  }
5
6  stop() {
7      hdfs dfs -rm /path/to/marker/my_job_unique_name
8      force_kill=true
9      application_id=$(yarn application -list | grep -oe "application_[0-9]*_[0-9]*"`)
10     for i in `seq 1 10`; do
11         application_status=$(yarn application -status ${application_id} | grep "State : \(RUNN
```

```
12          if [ -n "$application_status" ]; then
13              sleep 60s
14          else
15              force_kill=false
16              break
17          fi
18      done
19      $force_kill && yarn application -kill ${application_id}
20  }
```

In the Spark Streaming application, background thread should monitor marker file, and when the file disappears stop the context calling `streamingContext.stop(stopSparkContext = true, stopGracefully = true)`.

Summary

As you could see, configuration for mission critical Spark Streaming application deployed on YARN is quite complex. It has been long, tedious and iterative learning process of all presented techniques by a few very smart devs. But at the end, long-running Spark Streaming applications deployed on highly utilized YARN cluster are extraordinarily stable.

Posted by Marcin Kuthan Sep 30th, 2016 hdfs, spark, yarn

Tweet

« Spark application assembly for cluster deployments Apache BigData Europe Conference Summary »