Spark Internals

Performance Tuning

Agenda

- Spark working directory
- Spark Event Logging
- Spark User Logging
- Submit spark application
- Customizing the logging behaviour
- Sizing the spark driver and executors
- Java process names for various spark daemons
- Spark Executor Memory Allocation
- Serialization
- File Format and Compression

Spark Scratch Directory

spark.local.dir - default /tmp/

- Stores temporary map output files
- RDDs that get stored on disk
- This should be on a fast, local disk in your system.
- Can also be a comma-separated list of multiple directories on different disks.
- Overridden by SPARK_LOCAL_DIRS (Standalone, Mesos) or LOCAL_DIRS (YARN) environment variables set by the cluster manager.

Web Interface

- Every SparkContext launches a web UI
- Access at http://<driver-node>:4040 in a web browser
- Web UI shows
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Environmental information.
  - Information about the running executors
- For logs - check the logs directory. More on this on a later slide.

Retain Logs using Spark History Server

- Enable Event Logging
  - `spark.eventLog.enabled true`
  - `spark.eventLog.dir hdfs://namenode/shared/spark-logs`
  - `spark.history.fs.logDirectory` (default: /tmp/spark-events) This can be a local file:// path, an HDFS path hdfs://namenode/shared/spark-logs
- Start spark history server or using service start command:
  - `$ sbin/start-history-server.sh`
  - `$ sudo service spark-history-server start`
- Works for spark on YARN and Standalone mode
- The history server displays both completed and incomplete Spark jobs
- Incomplete applications are only updated intermittently as specified by `spark.history.fs.update.interval`

Rest API for Event Logs

- Event logs are exposed through Rest API service as well.
- Accessible at `http://<server-url>:18088/api/v1`
- Example: view all jobs for a given application
- http://localhost:18088/api/v1/applications/application_1489509347069_0002/jobs

Logs on Yarn

View YARN configuration:
http://<resource manager>:8088/conf

Change YARN configuration
$HADOOP_CONF_DIR/yarn-site.xml

Enable Log: `yarn.log-aggregation-enable`

Container Log Location:

- `yarn.nodemanager.remote-app-log-dir`
- `yarn.nodemanager.remote-app-log-dir-suffix`

Control Logging level using /etc/spark/conf/log4j.properties

Retrieve aggregated logs using

```
$ yarn logs -applicationId <application_id>
```

Submit Option

Standalone Mode

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  lib/spark-examples*.jar 1000
```

YARN cluster

```
$ export HADOOP_CONF_DIR=XXX
```

```
$ ./bin/spark-submit \
    --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue default \
    --num-executors 4
    lib/spark-examples*.jar \
    10
```

Passing log4j.properties for a job

```
spark-submit \
    --class com.einext.WordCount \
    --master yarn-cluster \
    --driver-memory 500m \
    --executor-memory 500m \
    --executor-cores 1 \
    --queue default \
    --files log4j.properties \
    --conf "spark.executor.extraJavaOptions='-Dlog4j.configuration=log4j.properties'" \
    --conf "spark.driver.extraJavaOptions='-Dlog4j.configuration=log4j.properties'" \
    /home/cloudera/workspace/WordCount/target/WordCount-0.0.1-SNAPSHOT.jar stories
```

If you are using file, specify the container location as below. If log aggregation is enabled at Yarn, it will move the logs to HDFS as a single file

log4j.appender.file.File={spark.yarn.app.container.log.dir}/spark-app.log

Java Processes

View Java processes

$ sudo jps -lm

`org.apache.spark.deploy.yarn.ExecutorLauncher` -> Application Master in client mode

`org.apache.spark.deploy.yarn.ApplicationMaster` -> Application Master in cluster mode

`org.apache.spark.executor.CoarseGrainedExecutorBackend` -> Executor

Server (physical RAM, physical cores)

YARN

```
yarn.nodemanager.resource.memory-mb
yarn.nodemanager.resource.cpu-vcores
```

Container

Container Size Limits

```
yarn.scheduler.minimum-allocation-mb
yarn.scheduler.maximum-allocation-mb
yarn.scheduler.minimum-allocation-vcores
yarn.scheduler.maximum-allocation-vcores
yarn.nodemanager.vmem-pmem-ratio

yarn.app.mapreduce.am.resource.cpu-vcores
```

Spark on YARN

- Every Spark executor in an application has the same fixed number of cores and same fixed heap size
- There are benefits running multiple tasks in the same executor JVM. The broadcast variables will sent once per executor. Tasks on the executor can share the same broadcast variable.

Resource Allocation on YARN

Configuration	Description
`yarn.nodemanager.resource.cpu-vcores`	maximum sum of cores used by the containers on each node
`yarn.nodemanager.resource.memory-mb`	the maximum sum of memory used by the containers on each node
`spark.executor.cores` (`--executor-cores`)	No of cores per executor. This property controls the number of concurrent tasks an executor can run.
`spark.executor.memory` (`--executor-memory`)	Max amount of **heap size** per executor. VMs can also use some memory off heap, for example for interned Strings and direct byte buffers. Actual requested max(384, .07 * spark.executor.memory)
`spark.executor.instances` (`--num-executors`)	Control the number of executors requested. Not required if spark.dynamicAllocation.enabled = true
`spark.dynamicAllocation.enabled`	enables a Spark application to request executors when there is a backlog of pending tasks and free up executors when idle

Example: 6 x 16 core CPU and 64g RAM

`yarn.nodemanager.resource.memory-mb = 63g` (1 GB left for OS and other Hadoop daemons)
`yarn.nodemanager.resource.cpu-vcores = 15` (1 left for OS and other hadoop daemons
Application Master = 1 CPU, that leaves 14 CPU on one of node

	Parameters	Comment

Option 1	`spark.executor.instances 6`
	`spark.executor.cores 15`
	`spark.executor.memory 63g`
		Try to limit `spark.executor.cores` to 5 to avoid HDFS client performance. Where will AM run? 63GB + the executor memory requires 63 * 1.07 = 67.41 GB container

Option 2	`spark.executor.instances 17`
	`spark.executor.cores 5`
	`spark.executor.memory (63 / 3) * 0.93 = 19g`
		This config results in three executors on all nodes except for the one with the AM, which will have two executors.

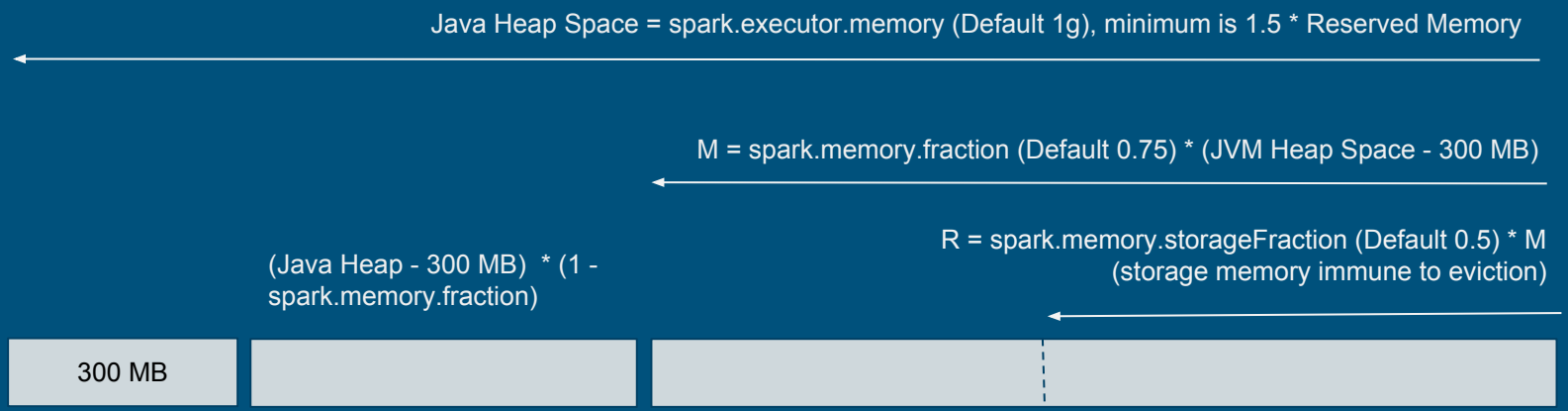
Spark Caching Control

Factors for memory management

- Amount of memory used by objects
- Cost of accessing those objects
- Garbage collection time

- Java objects (deserialized data) take 2-5x more space than raw bytes (serialized data)
- A java object (deserialized object) has a object header 16 bytes that is pointer to its class
- A java string has 40 bytes of overhead (header to the class, length of the string) also each char is stored in 2 bytes for its inter UTF16 encoding

Spark Executor Memory Allocations



Reserved Memory

Dedicated to JVM

User Memory

- Memory used by Intermediate objects
- Spark leaves it to developers to use it.

Spark Execution

Used by spark functions - shuffles, joins, sorts and aggregations. Used to hold output data of s stage. Smaller is this memory, more frequent spill will occur during spark execution resulting in high GC

Storage

- Caching and propagating internal data across the cluster
- Keeps broadcast variables

Out Of Memory Exception

Below are a few possible resolutions

- Increase the amount of memory allocated to executors
- Increase the number of partition … start by doubling the number of partitions
- Decrease the number of core allocated per executor

Serialization Library

- Data is serialized when sent across network or saved in memory/disk in serialized format
- Serialization is specified by this configuration property spark.serializer
- Default serializer org.apache.spark.serializer.JavaSerializer
- Recommended serializer: org.apache.spark.serializer.KryoSerializer
- Datasets or Dataframe use its own serialization library based on tungsten engine

Broadcast Variables

- Keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- Commonly used to distribute data such a lookup table

```
val addresse: Map[String, Address] = ...
val bpostalcode = sc.broadcast(addresses)

val addresses = rdd.map{info =>
      bpostalcode.value(info.postalcode)}
```

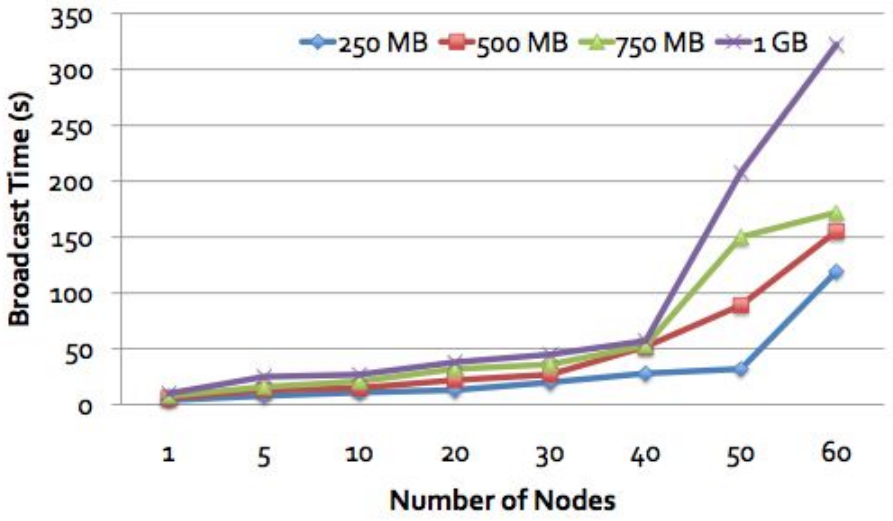


Figure 5: Scalability and performance of CHB (All nodes: m1.large EC2 instances).

References

- Spark On Yarn - inside look https://github.com/jaceklaskowski/mastering-apache-spark-book/blob/master/yarn/spark-yarn-applicationmaster.adoc
- Spark Configuration
- Launching Spark application
- Spark History Server
- Spark memory management
- https://www.slideshare.net/jcmia1/a-beginners-guide-on-troubleshooting-spark-applications