A Journey Into Big Data with Apache Spark: Part 1

The first in a series of posts about getting know Apache Spark for Big Data Processing.

Ashley Broadley Follow
Dec 10, 2018 · 11 min read

. . .

My journey into Big Data began in May 2018. I've been a Software Engineer for over a decade, being both hands on and leading the development of some of Sky Betting & Gaming's biggest products and the services that underpin them. During this time I've learned a lot about how to build and operate applications that scale, at scale—from the need for logs and metrics, to performance optimisation and maintainability. When I was asked to become the Engineering Manager for our Data Tribe, I was both excited and extremely nervous—I'd lead small teams before, but knew nothing about Big Data. So I decided to learn!

"Where do you start with something like this?" is often the question that I would ask myself. The Big Data ecosystem is, well, big. So I asked one of the Principal Engineers within my team what they would recommend and Apache Spark came top of their list.

What is Spark?

From the Apache Spark website:

> ***Apache Spark™*** *is a unified analytics engine for large-scale data processing.*

In Human terms, Spark is a distributed computing framework that has a common interface across multiple languages for things like SQL & MapReduce when querying in-memory datasets. Being distributed, Spark can work with extremely large datasets across a cluster of machines.

Why Spark?

From an Engineering perspective, Spark is available in multiple languages: Scala, Java, Python and R. This seemed like a pretty big win as it doesn't limit what we can hire for to a specific language. Spark has it's own DSL (Domain Specific Language) that's the same across all implementations, meaning there's a common language despite the choice of implementation language. Our Data Science Squad uses a combination of PySpark and SparkR and the wider Tribe uses Scala—we let Squads (mostly) choose their own tooling.

Spark's use of Data Frames lends itself very well to the usual Software Engineering and application design principles such as Unit Testing, Data Modelling, Single Responsibility Principle and so on. Again, another big win. I can continue to build applications in the same way as I've always done (I'm a big fan of the 12factor principles).

Spark also comes with an SQL interface, which is usually familiar to most Programmers/Engineers that have ever needed to store & query data somewhere.

Spark even has a Streaming library available, if that's what you're into. Not that I will be immediately (although I'm huge fan of Apache Kafka), but it does seem quite attractive for future learnings (and I suppose business use cases).

Let's Get Started

Apache Spark seemed quite daunting (as anything new does) and again, I found myself asking *"Where do I start?"* I'm a big believer that understanding things from the ground up makes you a better Engineer. That, plus the need to actually practice using Spark somewhere lead me to build a small local Spark standalone cluster.

Note: I'm a big fan of Docker. It gives you a way to package things and distribute in a very portable way. The only dependency I need to install on my own machine is Docker itself. I'll be using Docker in all my examples, so you might want to get it from here. I'm also running OSX, which has some minor limitations with Docker, but we can work around them.

Our first task is to find a suitable Docker image that contains a version of Java. Being a fan of Open Source Software, I chose to use OpenJDK. I'm not a fan of bloat and decided that Alpine Linux was the way to go. Luckily, there's an image we can use already available. Let's begin creating our *Dockerfile.*

Create a new file in an empty directory called *Dockerfile*. Add the following line at the very top:

```
FROM openjdk:8-alpine
```

Now we can build our image, which uses the OpenJDK image as a base. Build the image by running the following:

```
docker build .
```

This will pull the base image and create our own version of the image. You'll see something akin to the following output:

```
Successfully built 68ddc890acca
```

Instead of noting the output hash each time we build the image, this time we'll *tag* the image, giving it name so it's easier to work with (replacing $MYNAME with your own name, of course). This is a slight tweak to the build command above:

```
docker build -t $MYNAME/spark:latest .
```

Now instead of using the generated hash to refer to the built image, we can refer to it using the tag we gave it. Test by running a container using the tag name like so:

```
docker run -it --rm $MYNAME/spark:latest /bin/sh
```

This image isn't of much use yet, as all it contains is Java. Let's install some utilities that we're going to need to download and install Spark. We're going to need `wget` to download the archive containing Spark and `tar` to extract files from that archive. We'll also need `bash` to run some things. On a new line in the *Dockerfile*, add the following (we need to include `--update` so that Alpine has a fresh list of repositories to download binaries from):

```
RUN apk --update add wget tar bash
```

Now rebuild the image and watch as `wget` , `tar` and `bash` are installed.

We can now download and install Spark. We'll be using the latest version of Spark (2.4.0) based on Scala 2.11 and Hadoop 2.7 (Don't worry about Hadoop, we're not interested in that… yet). Add the following to a new line in your *Dockerfile* and build:

```
RUN wget http://apache.mirror.anlx.net/spark/spark-
2.4.0/spark-2.4.0-bin-hadoop2.7.tgz
```

Docker is pretty clever and will reuse the *layers* that we previously built, so all it has to do at this point is download the Spark archive. Once built, the image will contain the Spark archive ready for us to install. Installation is a matter of simply extracting Spark from the archive and placing it somewhere sensible. Add another line to the *Dockerfile* to do this like so (I break the command on to multiple lines for readability. The `rm` is to simply clean up the downloaded archive) and then build:

```
RUN tar -xzf spark-2.4.0-bin-hadoop2.7.tgz && \
    mv spark-2.4.0-bin-hadoop2.7 /spark && \
    rm spark-2.4.0-bin-hadoop2.7.tgz
```

Congratulations! You've now downloaded and installed Spark on a Docker image. Time to test it! Let's start a container and get a shell. Take note of the ID output at the end of building the image, as we'll be using this now. In your shell run the following command:

```
docker run --rm -it $MYNAME/spark:latest /bin/sh
```

We'll then end up in a shell running inside the container, with which we use to start a Spark *Master*. Spark needs a couple of options when starting to startup successfully. These are the port for the Master to listen on, the port of the WebUI and the hostname of the Master:

```
/spark/bin/spark-class org.apache.spark.deploy.master.Master
--ip `hostname` --port 7077 --webui-port 8080
```

And hopefully you should see a bunch of log output! If so, CONGRATULATIONS! You've successfully started running a Spark Master.

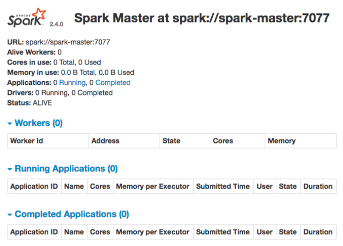
The next step is to get some Workers added to the cluster, but first, we need to set some configuration on the Master so that the Worker can talk to it. To make things easy, we'll give the Master a proper name and expose the Master port (the `--port` option in the last command) and also make the WebUI available to us. Stop the Master and drop out of the container by using CTRL+C and then CTRL+D. You should now be in your local shell. Simply tweak the `docker run` command to add the `--name` , `--hostname` and `-p` options as per below and run:

```
docker run --rm -it --name spark-master --hostname spark-
master \
    -p 7077:7077 -p 8080:8080 $MYNAME/spark:latest /bin/sh
```

Run `docker ps` and you should see the container running with an output similar to this (I've removed some output to make it fit in the code block):

```
CONTAINER ID  PORTS                                   NAMES
3dfc3a95f7f4  ..:7077->7077/tcp, ..:8080->8080/tcp spark-
master
```

In the container, re-run the command to start the Spark Master and once it’s up, you should be able to browse to http://localhost:8080 and see the WebUi for the cluster, as per the screenshot below.



Spark 2.4.0 **Spark Master at spark://spark-master:7077**

URL: spark://spark-master:7077
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (0)

Worker Id	Address	State	Cores	Memory

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Spark Master WebUI

Adding Worker Nodes

As I mentioned, I’m using Docker for Mac, which makes DNS painful and accessing the container by IP nigh on impossible without running a local VPN server or something to work around the issue—that’s beyond the scope of this post. Luckily, Docker has its own networking capability (the specifics of which are out of scope of this post, too) which we’ll use to create a network for the local cluster to sit within. Creating a network is pretty simple and is done by running the following command:

```
docker network create spark_network
```

We don’t need to specify any particular options as the defaults are fine for our use case. Now we need to recreate our Master to attach it to the new network. Run `docker stop spark-master` and `docker rm spark-master` to remove the current *instance* of the running Master. To recreate the Master on the new network we can simply add the `--network` option to `docker run`, as per the below:

```
docker run --rm -it --name spark-master --hostname spark-
master \
    -p 7077:7077 -p 8080:8080 --network spark_network \
    $MYNAME/spark:latest /bin/sh
```

This is really no different to the first time we ran the Spark Master, except it uses a newly defined network that we can use to attach Workers to, to make the cluster work 👍. Now the Master is up and running, let’s add a Worker node to it. This is where the magic of Docker really shines through. To create a Worker and add it to the cluster, we can simply *launch a new instance of the same docker image* and run the command to start the Worker. We’ll need to give the Worker a new name, but other than the command remains largely the same:

```
docker run --rm -it --name spark-worker --hostname spark-
worker \
    --network spark_network \
    $MYNAME/spark:latest /bin/sh
```

And to start the Spark Worker on the container, we simply run:

```
/spark/bin/spark-class org.apache.spark.deploy.worker.Worker
\
    --webui-port 8080 spark://spark-master:7077
```

When it’s started and connect to the master, you should see the last line of the output being:

```
INFO  Worker:54 - Successfully registered with master
spark://spark-master:7077
```

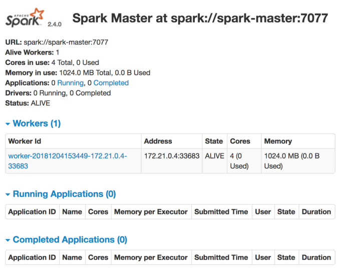
And the Master will output the following line:

```
INFO  Master:54 - Registering worker 172.21.0.2:37013 with 4
cores, 1024.0 MB RAM
```

Congratulations! You’ve setup a Spark **cluster** using Docker!

But Does it Work?

To check it works, we can load the Master WebUI and we should see the Worker node listed under the “Workers” section, but this only really confirms the log output from attaching the Worker to the Master.



Spark 2.4.0 **Spark Master at spark://spark-master:7077**

URL: spark://spark-master:7077
Alive Workers: 1
Cores in use: 4 Total, 0 Used
Memory in use: 1024.0 MB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20181204153449-172.21.0.4-33683	172.21.0.4:33683	ALIVE	4 (0 Used)	1024.0 MB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Spark Master WebUI with Worker

To be a true test, we need to actually run some Spark code across the cluster. Let's run a new instance of the docker image so we can run one of the examples provided when we installed Spark. Again, we can reuse the existing docker image and simply launch a new instance to use as the *driver* (the thing that submits the application to the cluster). This one doesn't need the `--hostname`, `--name` and `-p` options:

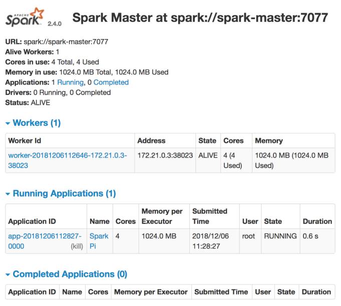
```
docker run --rm -it --network spark_network \
    $MYNAME/spark:latest /bin/sh
```

In the container, we can then submit an application to the cluster by running the following command:

```
/spark/bin/spark-submit --master spark://spark-master:7077 -
-class \
    org.apache.spark.examples.SparkPi \
    /spark/examples/jars/spark-examples_2.11-2.4.0.jar 1000
```

The example provided works out what Pi is utilising the cluster.

While the application is running, you should be able to see that in the Spark Master WebUI:



Spark 2.4.0 **Spark Master at spark://spark-master:7077**

URL: spark://spark-master:7077
Alive Workers: 1
Cores in use: 4 Total, 4 Used
Memory in use: 1024.0 MB Total, 1024.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20181206112646-172.21.0.3-38023	172.21.0.3:38023	ALIVE	4 (4 Used)	1024.0 MB (1024.0 MB Used)

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20181206112827-0000 (kill)	Spark Pi	4	1024.0 MB	2018/12/06 11:28:27	root	RUNNING	0.6 s

Completed Applications (0)

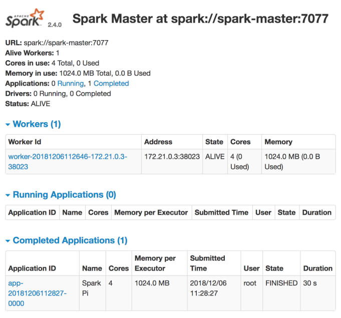
Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Spark Master WebUI — Running Application

Once complete, you'll see the value of Pi output in the logs:

```
Pi is roughly 3.1414459514144597
```

And you'll see the application reporting as complete in the WebUI too:



Spark 2.4.0 **Spark Master at spark://spark-master:7077**

URL: spark://spark-master:7077
Alive Workers: 1
Cores in use: 4 Total, 0 Used
Memory in use: 1024.0 MB Total, 0.0 B Used
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20181206112646-172.21.0.3-38023	172.21.0.3:38023	ALIVE	4 (0 Used)	1024.0 MB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20181206112827-0000	Spark Pi	4	1024.0 MB	2018/12/06 11:28:27	root	FINISHED	30 s

Spark Master WebUI — Completed Application

Hooking it Together With Docker Compose

Docker Compose is a neat utility provided with Docker that we can use as an orchestration tool so that we don't have to keep running commands ourselves in a number of terminal windows. We essentially stitch together the various commands and parameterise some things which means we can simply run `docker-compose up` and the cluster starts running.

To enable this, we can create some scripts to copy to the image and run at container start. The first one we'll create will be to setup the Spark Master. Create a new file called `start-master.sh` and add the following command:

```
#!/bin/sh

/spark/bin/spark-class org.apache.spark.deploy.master.Master
\
    --ip $SPARK_LOCAL_IP \
    --port $SPARK_MASTER_PORT \
    --webui-port $SPARK_MASTER_WEBUI_PORT
```

Instead of specifying the IP, Master and WebUI ports directly in the script, we've parameterised them, meaning we can provide them as environment variables. Thus giving greater flexibility around the configuration (ports) the Master listens on. To get the script onto the image, we need to copy it. Before we do that however, we need to ensure the script is executable. Simply run `chmod +x start-worker.sh` to make it so. Now to add the script into the image, in the *Dockerfile*, below the last `RUN` command, add the following:

```
COPY start-master.sh /start-master.sh
```

Before we rebuild the image, we'll create a similar script for the Worker too. Create a new file called `start-worker.sh` and add the following command:

```
#!/bin/sh

/spark/bin/spark-class org.apache.spark.deploy.worker.Worker
\
    --webui-port $SPARK_WORKER_WEBUI_PORT \
    $SPARK_MASTER
```

Again, we've parameterised the configuration to give greater flexibility. Make the new script executable (`chmod +x start-worker.sh`) and add another `COPY` line to the *Dockerfile* to add the script to the image:

```
COPY start-worker.sh /start-worker.sh
```

Rebuild and run. If all is well, you'll land in a shell in the container that has the scripts located in the root of the filesystem:

```
bash-4.4# pwd
/
bash-4.4# ls -l
total 72
...
-rwxr-xr-x    1 root   root   357 Dec 6 11:56 start-
master.sh
-rwxr-xr-x    1 root   root   284 Dec 6 17:10 start-
worker.sh
...
```

Come back out and we'll hook this up with `docker-compose` to orchestrate a cluster.

Create a new file called `docker-compose.yml` and enter the following:

```
version: "3.3"
services:
  spark-master:
    image: $MYNAME/spark:latest
    container_name: spark-master
    hostname: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
    networks:
      - spark-network
    environment:
      - "SPARK_LOCAL_IP=spark-master"
      - "SPARK_MASTER_PORT=7077"
      - "SPARK_MASTER_WEBUI_PORT=8080"
    command: "/start-master.sh"
  spark-worker:
    image: $MYNAME/spark:latest
    depends_on:
      - spark-master
    ports:
      - 8080
    networks:
      - spark-network
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
      - "SPARK_WORKER_WEBUI_PORT=8080"
    command: "/start-worker.sh"
networks:
  spark-network:
    driver: bridge
    ipam:
      driver: default
```

All we've done here is specify the image name to use, set a hostname and container name, expose the right ports and attach to the network we create at the bottom, set some environment configuration and the command to run upon starting. We've also set the Worker to depend on the Master being up and running.

To bring the cluster up, we simply run `docker-compose up` . One of the great things about Docker Compose is that we can scale the Workers by simply adding a `--scale` option to the compose command. Say we want 3 Worker nodes, we run:

```
docker-compose up --scale spark-worker=3
```

And et voila! We're done. Tune in next time when we start to learn the basics by building our own application in Scala.