scark-cli Documentation

Release 1.1.0

Lukas Heumos

May 04, 2020

Contents:

Docker Builds Statuses

CHAPTER 1

spark-service

spark-service provides a docker based containerized solution for a quick setup of an Apache Spark infrastructure. It is designed to faciliate benchmarking and interoperability with scark-cli.

1.1 Spark

If you need an introduction to spark head over to Spark introduction.

1.2 Features

- spark-service offers docker containers for master, work and submit nodes
- All docker containers are integrated and connected via docker-compose
- Configurations and allocation parameters of master/worker nodes are easily changed through predefined variables
- Several runnable example scripts are already included
- spark-service is specifically designed to interoperate with scark-cli

1.3 Documentation

Please visit the Documentation.

1.4 Authors

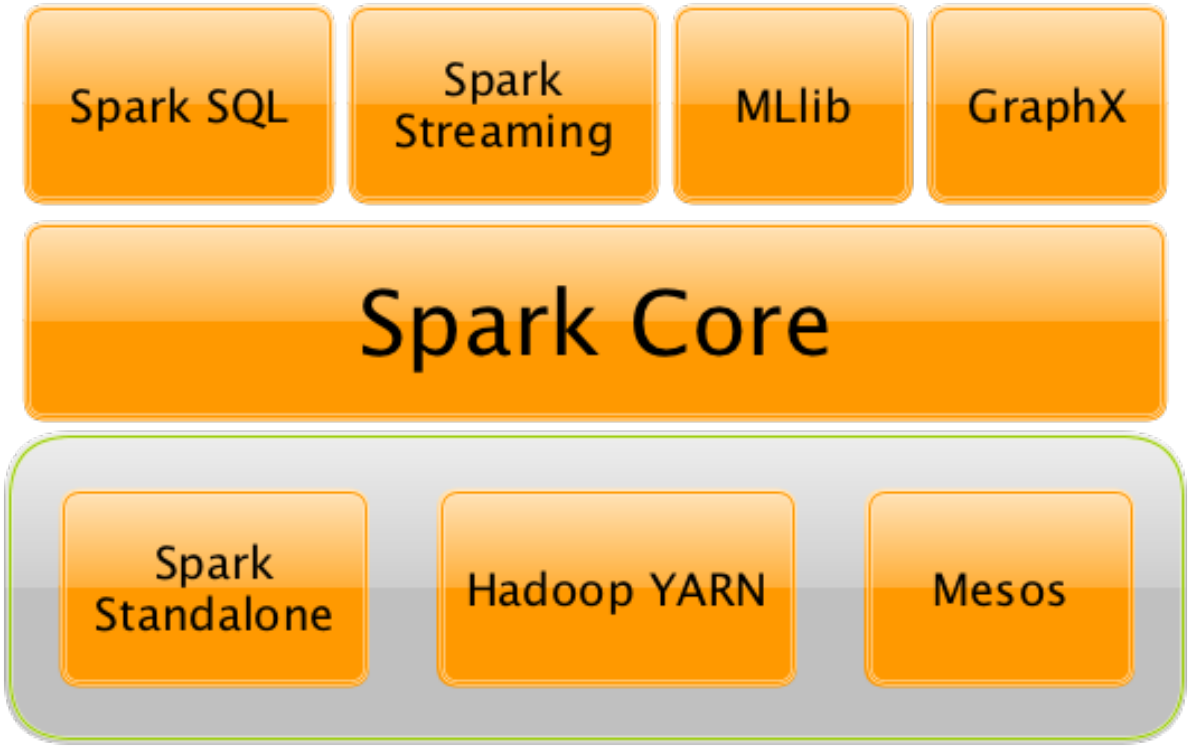
spark-service was designed and implemented by Lukas Heumos.

CHAPTER 2

Spark

2.1 About Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. Basically, it's a cluster computing framework offering a stack of libraries for SQL and data frames (Spark SQL), machine learning (MLIib), graphs (GraphX) and streaming (Spark Streaming).



Although, Spark is written in Scala, applications can also be submitted in Java, Python, R and SQL. Spark can either be run in its standalone cluster mode or on EC2, on Hadoop Yarn, Mesos or on Kubernetes.

2.2 Comparing Apache Spark with Apache Hadoop

Apache Spark differs from Apache Hadoop in their different approaches to processing. Spark is able to do it in-memory, whereas Hadoop's MapReduce has to read from and write to a disk. As a result, the processing speed differs significantly. Commonly, Spark is said to be up to 100 times faster.

However, the volume of data processed also differs: Hadoop's MapReduce is able to work with far larger datasets than Spark. Incase the dataset is larger than available RAM, Hadoop MapReduce may outperform Spark. Due to Hadoop's worse performance, it should only be used if no immediate results are expected e.g. if processing can be done overnight. Spark's iterative processing approach is especially useful, when data is processed multiple times. Spark's Resilient Distributed Datasets (RDDs) enable multiple map operations in memory, while Hadoop MapReduce has to write interim results to a disk.

CHAPTER 3

Installation

The following sections will guide you through building the required docker images and setting up the spark network.

3.1 Requirements

spark-service is based on docker containers and therefore needs docker installed on your machine. All required containers will automatically be downloaded by docker compose. More details at usage.

3.2 Obtaining the code

Clone the code from Github. Please mind the branches. If you want to work with the latest release always use the master branch.

```
git clone https://github.com/qbicsoftware/spark-service
```

Refer to usage for detailed instructions about how to run example scripts and your custom programs on the spark network.

CHAPTER 4

Usage

The following section will guide you through setting up a Spark network and running example scripts, as well as custom scripts.

4.1 Setting up a Spark network

Run docker-compose. All required docker images (base, master, worker, submit) will automatically be pulled from Dockerhub. Subsequently, the Spark network will be set up. A user specified number of workers (see example below) will automatically be added.

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

Verify that the network is up. Visit localhost:8080 and

```
docker network ls
```

Launch a new instance as the driver.

```
docker run --rm -it --network spark-service_spark-network zethson/qbic_spark_
↪submit:latest /bin/sh
```

4.2 Running an example script

A few example scripts are already provided by the spark installation. Let's calculate PI using our (three) worker nodes. While running the job you can observe, that the different workers are taking up different jobs.

```
/spark/bin/spark-submit --master spark://spark-master:7077 \
--class org.apache.spark.examples.SparkPi \
/spark/examples/jars/spark-examples_2.12-2.4.1.jar 1000
```

Visit localhost:8080 using your favorite browser and the job shuold show up as running. *Pi is roughly 3.141609511416095* should be the result. On localhost:8080 you should also see that a job has been completed.

4.3 Running custom programs

To access the local file system a volume has to be mounted for spark to be able to access the program. Place your program which you want to access from the driver into an isolated folder, say: /mnt/spark-apps . Launch your driver instance using the *-v* option to mount the volume /path/to/volume:/path/to/mount/to . Imagine there's a python script pi.py in /mnt/spark-apps:

```
docker run --rm -it --network spark-service_spark-network \
-v /mnt/spark-apps:/opt/spark-apps -v /mnt/spark-data:/opt/spark-data \
zethson/qbic_spark_submit:latest /bin/sh
```

You should now be able to find your script from the driver instance in /opt/spark-apps:

```
ls /opt/spark-apps
```

Now you can submit your script to the cluster to run it:

```
/spark/bin/spark-submit --master spark://spark-master:7077 /opt/spark-apps/pi.py 1000
```

Analogous to above your job should now run and calculate Pi. You can always verify that it ran on all workers using the web UI on localhost:8080 . Moreover, you should find example bash scripts for job submissions here.

4.4 Logs

All log files (stdout and stderr) can be viewed on localhost:8080 by clicking on either the completed job and then the specific worker or by accessing the specific worker directly from the front page.

Head over to configuration for information about how to configure the respective nodes and the expected performance.

CHAPTER 5

Configuration

For the following section we assume that we are working on a machine with 64 cores and 128 gigabytes of RAM. The configuration for the worker nodes can be found here.

Commonly one uses only a single worker node per machine, since each worker spins up its own JVM. Since the change SPARK-1706 Allow multiple executors per worker in Standalone mode in Spark 1.4 it's currently possible to start multiple executors in a single JVM process of a worker.

To launch multiple executors on a machine you start multiple standalone workers, each with its own JVM. It introduces unnecessary overhead due to these JVM processes, provided that there are enough cores on that worker. If you are running Spark in standalone mode on memory-rich nodes it can be beneficial to have multiple worker instances on the same node as a very large heap size has two disadvantages:

- Garbage collector pauses can hurt throughput of Spark jobs.
- Heap size of >32 GB can't use CompressedOops. So 35 GB is actually less than 32 GB.

To explore the effects of providing multiple cores or using multiple workers on your standalone Spark cluster you can run the PI example from above with different parameters settings. You can for example run it with 64 workers using 1 core each or with 1 worker using 64 cores. You will quickly notice that the 64 workers will have an advantage on the standalone cluster, since they're less prone to GC pauses. Moreover, the number of executor cores per worker and the overall executor cores can be limited using the following options:

```
--executor-cores 1
--total-executor-cores 2
```

You can resubmit your job now:

```
/spark/bin/spark-submit --master spark://spark-master:7077 --class org.apache.spark.
↪examples.SparkPi \
--executor-cores 1 --total-executor-cores 2 \
/spark/examples/jars/spark-examples_2.11-2.4.0.jar 5000
```

You could notice severe performance improvements, since executors won't try to use other worker's cores.

CHAPTER 6

Contributing

Please submit open issues and pull requests to spark-service.

CHAPTER 7

Known issues

There are currently no known issues.

CHAPTER 8

History

- 27.08.2019: [1.0.0]

CHAPTER 9

Authors

spark-service was designed and implemented by Lukas Heumos.

CHAPTER 10

Indices and tables

- search