Scalable analytics with Docker, Spark and Python

Description	Architecture with horizontal scaling for offline and online
Period	Fall 2015
Languages & Libs	Bash, Python
Tags	Architecture, Docker, Spark, Nginx, GitHub, JVM
GitHub	nikonyrh/docker-scripts

Traditionally data scientists installed software packages directly to their machines, wrote code, trained models, saved results to local files and applied models to new data in batch processing style. New data-driven products require rapid development of new models, scalable training and easy integration to other aspects of the business. Here I am proposing one (perhaps already well-known) cloud-ready architecture to meet these requirements.

In this example analytical models are written in Python but this pattern could be adapted to other scenarios as well. It is rooted on the idea of having a base docker image with needed Python libraries installed and then "branching" two special-purpose containers from that, as seen in Figure 1. The first one is used for offline model training and other is for online model applying. The main benefit is easy and robust deployment of the platform to new computer instances in cloud or in premises. Additionally it only requires docker support from the host OS and no for example Java or Python needs to be installed there at all.

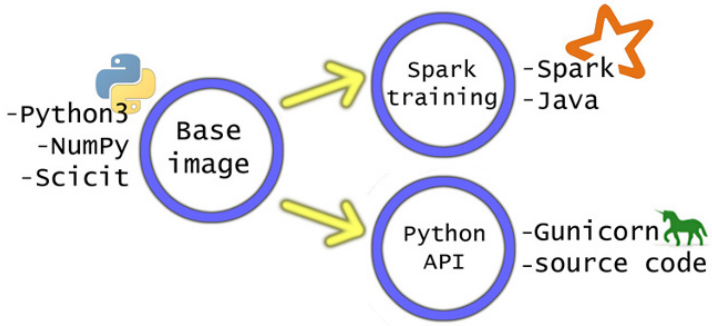

Figure 1: Three docker images' relations and their installed contents.

Every-day development could done within a container as well but for fast prototyping and IDE integration (such as Spyder) it might be simpler to install Python directly to the development machine. Virtualenv can be used to get decent level of isolation between projects.

Model training might be implemented as a high-level function which accepts parameters which restrict the scope to a specific timespan, products, stores, users or other entities. Typically different "scopes" can be trivially trained in parallel, but if it takes tens of minutes to train each chunk then it might be worthwhile to train scopes in sequence and utilize parallelism within the scope. Within a single machine libraries like multiprocessing could be used, but to scale horizontally Apache Spark is a perfect fit. A great benefit is that it supports Scala, Java, Python and R programming languages. It is based on master-slave model, in which job definition (parameters + source code) is submitted to the master and it coordinates sub-jobs execution on slaves. Its architecture is illustrated at Figure 2. It comes with machine learning library MLib or you can write your own.

Naturally Spark could be installed directly on the operating system, but if Python is used then also needed libraries such as NumPy needs to be installed as well. Thus creating new slaves and adding them to the cluster is greatly simplified by creating a Docker image. It is derived from a base image which has relevant Python libraries installed and adds Java + Spark binaries and start-up scripts.

The resulting image is about 1 GB in size but compresses down to 500 MB for transportation. To add a new machine to the cluster the image is simply loaded to the machine and start-up script is called with

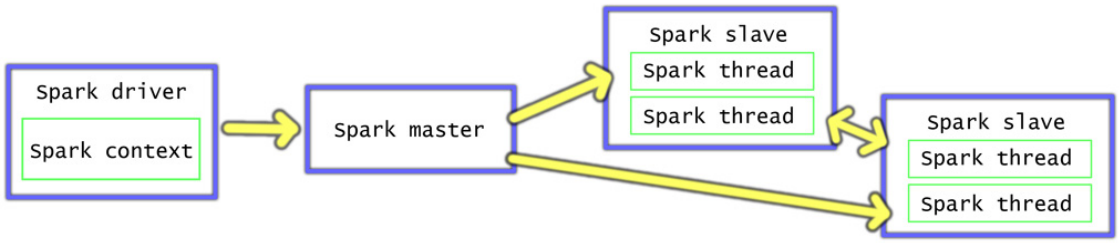

Figure 2: Three main components of Spark: the driver (the client), cluster master and slave nodes.

relevant parameters such as master node's IP and the number of workers / CPU core. Also attention needs to be paid on SPARK_PUBLIC_DNS setting, as a different public ip needs to be advertised to the master than the container's internal ip. Details were discussed at Stackoverflow.

Once models have been trained they are ready to be made available for others to benefit from, be it other teams within the organization or making it available to the public. At the age of service oriented architecture and micro services a HTTP based interface is a natural choice. Naturally scalability and fault tolerance are important aspects as well. There are many ways to achieve this but it is easy to get started with Nginx front-end and Gunicorn back-ends, a seen on Figure 3. This also enables zero-downtime software upgrades by starting a new back-end, confirming it works, adding it to the nginx config and removing + shutting down the old ones. At larger scale all this should be automatically managed but I don't have personal experience of those yet. On this architecture project's source code is "baked in" to the container and models are loaded from for example an external database. Alternatively they could be serialized and compressed into files within the container as well.

It is possible to have clients randomly connect either one of the instances, or have a "primary" Nginx which clients use by default. If the latter option is chosen then other instances' back-ends needs to be added to primary Nginx's config as well, as indicated by grayish arrows on the figure below.

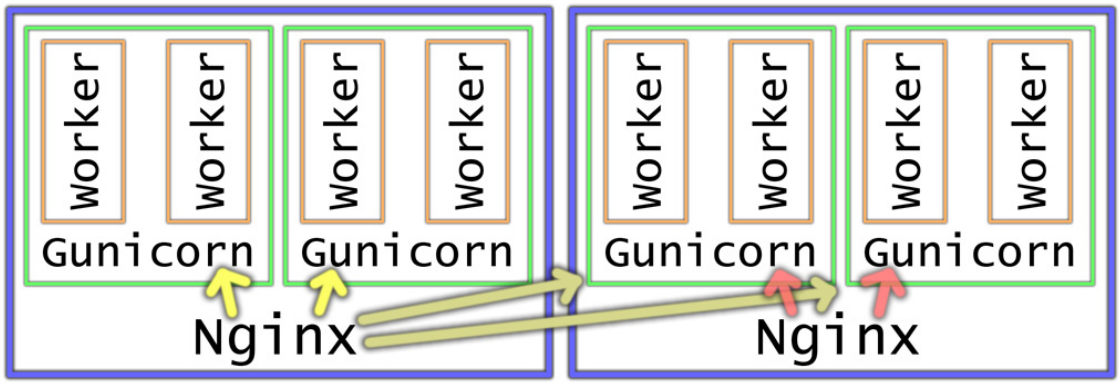

Figure 3: Two example instances with Nginx running on them and two Gunicorn backends on each.

This article didn't cover scalable databases such as Elasticsearch, Cassandra and Amazon Redshift. It is an interesting topic on its own, and a perfect fit for data processing with Spark. It also supports file-based data sources such as HDFS and Amazon S3. An other topic is automatic cluster management systems such as Docker Swarm, Apache Mesos and Amazon EC2 Container Service. Highest cost savings from cloud come from adaptively utilizing resources based on expected load. When new containers are started and others are shut down service discovery becomes a crucial "glue" to stitch everything together in a fault-tolerant manner. This can be solved via traditional DNS or utilize newer solutions like distributed Registrator and Consul.