lofn Documentation

Release 0.4.0

Michael T. Neylon

Dec 06, 2017

Contents:

CHAPTER 1

Introduction to lofn

A magical wrapper for serial tools to parallelize them using Spark and Docker.

1.1 How It Works

lofn uses Spark to partition data and schedule tasks. The tasks are carried out by Docker. lofn writes the partitions to temp files for Docker to mount. The Docker container carries out the commands you set up and writes the resultant files to the same temp directory to be read back into Spark.

A program that can be broken down into map and reduce steps can be parallelized using lofn. This can work on both serial and parallelized tools. If a program is built to work in a multi-core environment but not on a cluster, lofn can help deploy it on a cluster and each task can still run with its native parallel code.

Explore some of our examples.

1.2 Supported Features

- Map and Reduce for text files and binary files.
  - `map_binary` takes text as input and produces binary output that can be passed to `reduce_binary`
- User volumes as a global reference that are not partitioned.
- User defined functions can override how to write the temp files from the partition, such as unpacking key, value pairs into separate files.

CHAPTER 2

Getting Started

2.1 Installation

```
pip install lofn
```

2.2 Dependencies

You can use python 2 or 3, 2.7+ and 3.6+ preferably.

Running a script on this framework requires Spark and Docker.

2.2.1 Install Docker

See the Docker Docs on how to install.

2.2.2 Install Spark

See the Downloading Spark instructions to get started. It will require Java be installed and in your `PATH` or set `JAVA_HOME` and downloading the jar files. Then set `SPARK_HOME` as the path to this directory and add its `bin` directory to `PATH` as well.

2.3 Running on Standalone

lofn can be run on Spark standalone on a cluster or a single node. Use `spark-submit` to submit your application to Spark.

2.4 Running on YARN

Some configurations are required for lofn to work on YARN.

2.4.1 Configure the Cluster

Beyond having Spark setup on a YARN cluster ready to submit jobs, follow these steps for lofn to work:

- install lofn on each node
- install Docker on each node
- create a Docker group
- add $USER and `yarn` user to Docker group
- restart yarn daemons and your shell for changes to take effect

See the next page 'Using lofn on AWS' for instructions on how to setup an EMR cluster automatically for `lofn`

2.4.2 Submission

- User volumes must be in HDFS and your `volumes` dictionary should provide the absolute path to the directory on HDFS
- use `spark-submit` to submit the application to Spark

2.5 Examples

Explore some of our examples to get started.

CHAPTER 3

Using lofn on AWS

Setup an Elastic Map Reduce cluster on Amazon with lofn

Setup an EMR cluster with Spark installed. Increase the root volume size from the default since lofn uses this for its temporary files.

3.1 Security Settings

- make sure port 22 is open on each node for SSH
- For docker swarm ensure ports are open: docker swarm

3.2 Manual Setup

The manual steps give a good outline for what is necessary to get a YARN cluster setup for lofn but may vary depending on the OS.

3.2.1 Master Node

Run the following commands on the master node:

```
sudo yum update -y
sudo yum install -y git docker
sudo service docker start
sudo groupadd docker
sudo usermod -a -G docker hadoop
sudo usermod -a -G docker yarn
sudo /sbin/stop hadoop-yarn-resourcemanager
sudo /sbin/start hadoop-yarn-resourcemanager
sudo pip install lofn
```

3.2.2 Worker Node

Run the following commands on the worker node:

```
sudo yum update -y
sudo yum install -y git docker
sudo service docker start
sudo groupadd docker
sudo usermod -a -G docker hadoop
sudo usermod -a -G docker yarn
sudo /sbin/stop hadoop-yarn-nodemanager
sudo /sbin/start hadoop-yarn-nodemanager
sudo pip install lofn
```

exit each shell and log back in for Docker group changes to take effect.

3.2.3 Build Docker Images

If you are using custom images that are not available in a registry, build the images on each node.

3.3 Automatic Setup

If using EMR, the following steps can setup a template for automatically building a cluster that is ready to use lofn.

3.3.1 Elastic Map Reduce (EMR)

You can run a bootstrap script on EMR to automatically install Docker Engine and lofn on each node. After the cluster is up, we then need to configure Docker and join a swarm (to host our Docker images).

Bootstrap

This bootstrap script will install Docker and lofn on each node. Create a shell script in an S3 bucket to run as a bootstrap step using the commands below:

```
#! /bin/bash

sudo yum update -y
sudo yum install -y git docker
sudo service docker start
sudo groupadd docker
sudo usermod -a -G docker hadoop
sudo pip install lofn
```

Step

Run the following as a step, to run after the cluster is running. This only executes on the master, so it will generate some scripts for you to run upon your first login so they can be executed on the workers.

Store the code below in an S3 bucket and choose a step as a Custom Jar, the path to which is `s3://<region>.elasticmapreduce/libs/script-runner/script-runner.jar` which allows you to execute a script. Add the s3 path to your script as an argument.

```
#! /bin/bash

sudo usermod -a -G docker yarn
sudo /sbin/stop hadoop-yarn-resourcemanager
sudo /sbin/start hadoop-yarn-resourcemanager

echo '#! /bin/bash' > $HOME/runme.sh
echo 'docker swarm init' >> $HOME/runme.sh
echo 'command=$(docker swarm join-token worker | sed "s/.*command:*//" | tr --delete
↪"\n" | tr --delete "\\\\")' >> $HOME/runme.sh
echo "echo '#! /bin/bash' > $HOME/worker_setup.sh" >> $HOME/runme.sh
echo "echo 'sudo usermod -a -G docker yarn' >> $HOME/worker_setup.sh" >> $HOME/runme.
↪sh
echo "echo 'sudo /sbin/stop hadoop-yarn-nodemanager' >> $HOME/worker_setup.sh" >>
↪$HOME/runme.sh
echo "echo 'sudo /sbin/start hadoop-yarn-nodemanager' >> $HOME/worker_setup.sh" >>
↪$HOME/runme.sh
echo 'echo $command >> $HOME/worker_setup.sh' >> $HOME/runme.sh

workers=$(hdfs dfsadmin -report | grep ^Name | cut -f2 -d: | cut -f2 -d ' ')

for host in $workers;
do
    echo "ssh -A -oStrictHostKeyChecking=no " $host " 'bash -s' < worker_setup.sh" >>
↪$HOME/runme.sh
done
chmod 700 $HOME/runme.sh
```

3.3.2 Login and Finish Setup

When this is finished, log in to the master node with SSH agent forwarding. The agent forwarding will enable SSH into the worker nodes from the master node.

Make sure you add your AWS identity to your local SSH agent:

```
ssh-add <awsid.pem>
```

login to the master:

```
ssh -A hadoop@MASTER_PUBLIC_DNS
```

Execute the script 'runme.sh':

```
./runme.sh
```

3.3.3 Build and Serve Images

At this point, Docker Swarm is running on the cluster and can host a Docker Registry as a service. This enables lofn to use custom images that are not available in a registry without having to manually build the image on each node.

create an overlay network and the registry service on the swarm:

```
docker network create --driver overlay lofn-network
docker service create --name registry --publish 5000:5000 --network lofn-network␣
↪registry:2
```

Build the image, tag it, and push it into the registry. In the example below we are using an image from one of the examples

```
git clone https://github.com/michaeltneylon/lofn.git
cd lofn/example/advanced/gsnap_samtools
docker build -t gsnap_samtools .
docker tag gsnap_samtools localhost:5000/gsnap_samtools
docker push localhost:5000/gsnap_samtools
```

Now in the code, our images will be named as `localhost:5000/gsnap_samtools` so each node in the swarm knows from where to pull the image.

CHAPTER 4

lofn package

4.1 Subpackages

4.1.1 lofn.base package

Submodules

lofn.base.hdfs module

Base classes and functions to interact with HDFS.

lofn.base.hdfs.**get**(*remote*, *local*)
Get from HDFS

lofn.base.hdfs.**mkdir_p**(*path*)
Make directory with parents if they don't exist

lofn.base.hdfs.**put**(*local*, *remote*)
Put into HDFS

lofn.base.hdfs.**rm_r**(*path*)
Remove recursively

lofn.base.tmp module

Base functions and classes for using temp files.

lofn.base.tmp.**create_temp_directory**(*directory=None*)
Create a temporary directory. Supports setting a parent directory that is not temporary and will try to create it if it does not already exist.

Parameters **directory** – specify a parent directory instead of using default (None -> /tmp)

Returns path to directory

lofn.base.tmp.**write_binary_to_temp_file**(*data*, *path*)
Write string directly to file for binary data.

lofn.base.tmp.**write_to_temp_file**(*iterable*, *path*)
Write iterable to temporary file, an item per line.

Parameters

- **iterable** – list or tuples of contents to write to file
- **path** – absolute path to file. We use a tempdir as parent.

Returns path to temporary file

4.2 Submodules

4.3 lofn.api module

Wrapper to execute docker containers as spark tasks.

class lofn.api.**DockerPipe**(*spark_configuration*, ***kwargs*)
Bases: object

Main entry point for lofn.

Parameters **spark_configuration** – spark configuration (e.g. pyspark.SparkConf())

Keyword Arguments

- **temporary_directory_parent** – specify an absolute path to the parent directory to use for temp dirs and files. The default is None, which then uses a location specified by a platform-dependent list or uses environment variables TMPDIR, TEMP, or TMP. If specifying a path, it either needs to exist on all nodes or you must run it with appropriate permissions so lofn can attempt to create it.
- **shared_mountpoint** – the path to the directory inside each docker container that maps to the temporary directory on the host. (Default '/shared/')
- **volumes** – User defined volumes to mount in the container. This is useful for data that is not being partitioned and needs to be read into each container, such as a global reference. This must be given as a dictionary. The keys for the dictionary are the absolute paths to the directory on the host you want to bind. The value of each of these keys is the information on how to bind that volume. Provide a 'bind' path, which is the absolute path in the container you want that volume to mount on, and optionally provide a 'mode', as ro (read-only, the default) or rw (read-write). The structure of this input is similar to docker-py volumes, and resembles the following structure: {'[host_directory_path]': {'bind': '[container_directory_path]', 'mode': '[ro|rw] } }

map(*rdd*, *image_name*, *command*, ***kwargs*)
Map step by applying Spark's mapPartitions. This writes the partition to temp file(s) and executes a docker container to run the commands, which is read back into a new RDD.

Parameters

- **rdd** – a spark RDD as input
- **image_name** – Docker image name
- **command** – Command to run in the docker container

Keyword Arguments

- **container_input_name** – the name of the file within the shared_mountpoint that is written to before the map from the host, and read from as the first step in the map in the container. (Default 'input.txt')
- **container_output_name** – the name of the file the map step writes to inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'output.txt')
- **docker_options** – additional docker options to provide for 'docker run'. Must be a list.
- **map_udf** – optional keyword argument to pass a function that accepts a partition and transforms the data into a dictionary with a key as filename and value as contents, a list (iterable) of elements to write to that file within the temporary directory.

Returns transformed RDD

map_binary(**args*, ***kwds*)

Map binary output as a context manager. This currently takes rdd as input and will output a directory of the newly written binary files so that they can be read by the user with sc.binaryFiles. After finishing with the context manager, the temp files are destroyed.

Parameters

- **rdd** – spark RDD input
- **image_name** – docker image
- **command** – docker command to run

Keyword Arguments

- **container_input_name** – the name of the file within the shared_mountpoint that is written to before the map steps from the host, and read from as the first step in the map in the container. (Default 'input.txt')
- **container_binary_output_name** – the name of the output file in map_binary step inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'output.bin')
- **docker_options** – additional docker options
- **map_udf** – function that takes one input, the partition, and returns a dictionary of filename: contents (as iterable).
- **hdfs_tempdir** – temporary directory on HDFS to hold binary files. The default attempts to find the home directory for the user, but can be overridden by specifying an absolute path to use.

Returns directory path containing the output binary files to be read

reduce(*rdd*, *image_name*, *command*, ***kwargs*)

Apply a Spark reduce function to the input RDD. This will take rolling pairs and write to temp files, run a docker container, execute a command in the container over the temp files, write to temp files, and return the result.

Parameters

- **rdd** – a spark RDD as input
- **image_name** – Docker image name
- **command** – Command to run in the docker container

Keyword Arguments

- **container_input_name** – the name of the file within the shared_mountpoint that is written to before the reduce from the host, and read from as the first step in reduce in the container. (Default 'input.txt')
- **container_output_name** – the name of the file the reduce step writes to inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'output.txt')
- **docker_options** – additional docker options to provide for 'docker run'. Must be a list.
- **reduce_udf** – The default behavior for handling the pairs of partitions is to append right to left and write to one temp file. This can be overridden by supplying a 'reduce_udf' function that takes two inputs, the pair of partitions, and transforms them to return a dictionary mapping a key of filename to value of contents in a list (iterable) of elements to write to a file within the temp directory.

reduce_binary(*rdd*, *image_name*, *command*, ***kwargs*)

Reduce partitions from map_binary output. The format of these partitions is different so this handles them and also writes the temp files as one string rather than trying to split newlines since these are binary.

Parameters

- **rdd** – spark RDD
- **image_name** – docker image
- **command** – docker command

Keyword Arguments

- **container_binary_input_1_name** – the name of the left side file in reduce_binary step inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'input_1.bin')
- **container_binary_input_2_name** – the name of the right side file in reduce_binary step inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'input_2.bin')
- **container_binary_output_name** – the name of the output file in reduce_binary step inside the container. This path will belong inside of the shared_mountpoint which maps to the host temp directory for that partition. (Default 'output.bin')
- **docker_options** – additional docker options
- **reduce_udf** – default behavior for reduce_binary is to write each input as a temp file to give two binary input files to the container. Write a UDF here that takes two inputs and outputs a dictionary mapping filename: contents (a string)

Returns iterable of reduce results

4.4 lofn.docker_handler module

Run Docker containers.

exception lofn.docker_handler.**DockerFailure**
Bases: exceptions.Exception

Custom exception to communicate the container failed.

lofn.docker_handler.**execute**(*command*)
Use the shell to invoke Docker. Catch exceptions and return to master to be caught and reported helpfully.

Parameters **command** – bash command to call docker with requested configuration

Returns if failure, the message about why it failed is returned, otherwise it returns False

lofn.docker_handler.**run**(*image_name*, *command*, *bind_files*, *volume_files*, ***kwargs*)
Make system calls with subprocess to run our containers.

Parameters

- **image_name** – docker image to run a container
- **command** – docker command to run in container
- **bind_files** – container paths to set as mount point
- **volume_files** – host paths to mount

Keyword Arguments

- **docker_options** – additional options to pass to docker run as a list
- **temporary_directory_parent** – specify the parent directory for temporary directories and files. Must exist or have permission to create the directory. The default is None, which uses the a default from a platform-dependent list or the system's environment variable for TMP, TMPDIR, or TEMPDIR.

Returns status of execution, False is no issues otherwise returns failure message.

lofn.docker_handler.**validate_user_input**(*volumes*)
Validate the user input. Checks the type and structure.

4.5 lofn.hdfs_handler module

Interact with HDFS.

lofn.hdfs_handler.**setup_user_volumes_from_hdfs**(**args*, ***kwds*)
GET a local copy of the volume to mount in Docker and update and map the volumes dictionary with the new local temp directory.

Parameters **volumes** – Docker volumes specification as a dictionary, similar to structure seen in DockerPy. User defined volumes to mount in the container. example: {'[host_directory_path]': {'bind': '[container_directory_path]', 'mode': '[ro|rw] }} The host directory path in the dictionary must be the absolute path to the directory on HDFS.

Keyword Arguments **temporary_directory_parent** – manually specify the parent directory or the temp files directories, else default is None which uses the system's TMP/TMPDIR.

Returns a volumes dictionary mapping to the new temporary directories

4.6 lofn.tmp_file_handler module

Create temp files for Docker read/write

class lofn.tmp_file_handler.**UDF**(*temporary_directory*, *user_function*, ***kwargs*)
Bases: object

Define how to handle RDD partitions to temp files.

The return should be a dictionary, with filename as key and list of elements as value. These files are written inside of the shared mount point temporary directory.

These file names override the input container name.

map_udf()
Unpack a partition into multiple files based on a user defined function. The udf should return a dictionary, with filename as key and list of elements as value.These files are written inside of the shared mountpoint temporary directory.

These filenames override the input container name.

reduce_udf()
Define handling of pairs of partitions for the reduce step. Pass a function to handle the pair of partitions input and return a dictionary mapping file name as key and value as an iterable of contents to write to the temp file.

This will override the input container name.

write_temp_files(*inner_partitions*)
Write contents to temp file with defined name. Iterables are written line by line, while binary data is written as a single string.

Parameters **inner_partitions** – the content to write, either as iterable for regular files or string for binary data

lofn.tmp_file_handler.**handle_binary**(*origin_directory*, *destination_directory*, *input_path*, *master_type*)
Move, rename, and keep temp file outputs into one directory to be read back by user with sc.binaryFiles() and then remove the original temp directory used by the containers.

Parameters

- **origin_directory** – temporary directory mounted by container
- **destination_directory** – the shared temporary directory, either locally or on hdfs, for all the output files to be moved
- **input_path** – the full path to the file to be moved to on HDFS
- **master_type** – spark master type, yarn or standalone

Returns path to new file

lofn.tmp_file_handler.**read_back**(*shared_dir*, *output_file*)
Read text files back into an iterable from temp file then destroy the temp file and its parent directory.

Parameters

- **shared_dir** – the temporary directory that the container mounted
- **output_file** – path to output filename

Returns iterable of output file contents

lofn.tmp_file_handler.**read_binary**(*shared_dir*, *output_file*)
Read back binary file output from container into one string, not an iterable. Then remove the temporary parent directory the container mounted.

Parameters

- **shared_dir** – temporary directory container mounted
- **output_file** – path to output file

Returns str of file contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index