Dask Yarn

Apr 06, 2021

Contents

Dask-Yarn deploys Dask on YARN clusters, such as are found in traditional Hadoop installations. Dask-Yarn provides an easy interface to quickly start, scale, and stop Dask clusters natively from Python.

```python
from dask_yarn import YarnCluster
from dask.distributed import Client

# Create a cluster where each worker has two cores and eight GiB of memory
cluster = YarnCluster(environment='environment.tar.gz',
                      worker_vcores=2,
                      worker_memory="8GiB")
# Scale out to ten such workers
cluster.scale(10)

# Connect to the cluster
client = Client(cluster)
```

Dask-Yarn uses Skein, a Pythonic library to create and deploy YARN applications.

CHAPTER 1

Install

Dask-Yarn is designed to only require installation on an edge node. To install, use one of the following methods:

Install with Conda:

```
conda install -c conda-forge dask-yarn
```

Install with Pip:

```
pip install dask-yarn
```

Install from Source:

Dask-Yarn is available on github and can always be installed from source.

```
pip install git+https://github.com/dask/dask-yarn.git
```

1.1 Quickstart

Dask-Yarn is designed to be used like any other python library - install it locally and use it in your code (either interactively, or as part of an application). As long as the computer you're deploying on has access to the YARN cluster (usually an edge node), everything should work fine.

1.1.1 Install Dask-Yarn on an Edge Node

Dask-Yarn is designed to be used from an edge node. To install, use either conda or pip to create a new environment and install `dask-yarn` on the edge node.

Conda Environments:

Create a new conda environment with `dask-yarn` installed. You may also want to add any other packages you rely on for your work.

```
$ conda create -n my_env dask-yarn      # Create an environment

$ conda activate my_env                 # Activate the environment
```

Virtual Environments:

Create a new virtual environment with `dask-yarn` installed. You may also want to add any other packages you rely on for your work.

```
$ python -m venv my_env                 # Create an environment using venv

$ source my_env/bin/activate            # Activate the environment

$ pip install dask-yarn                 # Install some packages
```

1.1.2 Package your environment for Distribution

We need to ensure that the libraries used on the Yarn cluster are the same as what you are using locally. By default, `dask-yarn` handles this by distributing a packaged python environment to the Yarn cluster as part of the applications. This is typically handled using

- conda-pack for Conda environments
- venv-pack for virtual environments

See *Managing Python Environments* for more information.

Conda Environments:

If you haven't already installed conda-pack, you'll need to do so now. You can either install it in the environment to be packaged, or your root environment (where it will be available to use in all environments).

```
$ conda install -c conda-forge conda-pack   # Install conda-pack

$ conda-pack                                # Package environment
Collecting packages...
Packing environment at '/home/username/miniconda/envs/my_env' to 'my_env.tar.gz'
[########################################] | 100% Completed |  12.2s
```

Virtual Environments:

If you haven't already installed venv-pack, you'll need to do so now.

```
$ pip install venv-pack                     # Install venv-pack

$ venv-pack                                 # Package environment
Collecting packages...
Packing environment at '/home/username/my-env' to 'my-env.tar.gz'
[########################################] | 100% Completed |  8.3s
```

1.1.3 Kinit (Optional)

If your cluster is configured to use Kerberos for authentication, you need to make sure you have an active ticket-granting-ticket before continuing:

```
$ kinit
```

1.1.4 Usage

To start a YARN cluster, create an instance of `YarnCluster`. This constructor takes several parameters, leave them empty to use the defaults defined in the *Configuration*.

```
from dask_yarn import YarnCluster
from dask.distributed import Client

# Create a cluster where each worker has two cores and eight GiB of memory
cluster = YarnCluster(environment='environment.tar.gz',
                      worker_vcores=2,
                      worker_memory="8GiB")

# Connect to the cluster
client = Client(cluster)

# Do some work here

# Shutdown client and cluster (alternatively use context-manager as shown below):
client.shutdown()
cluster.shutdown()
```

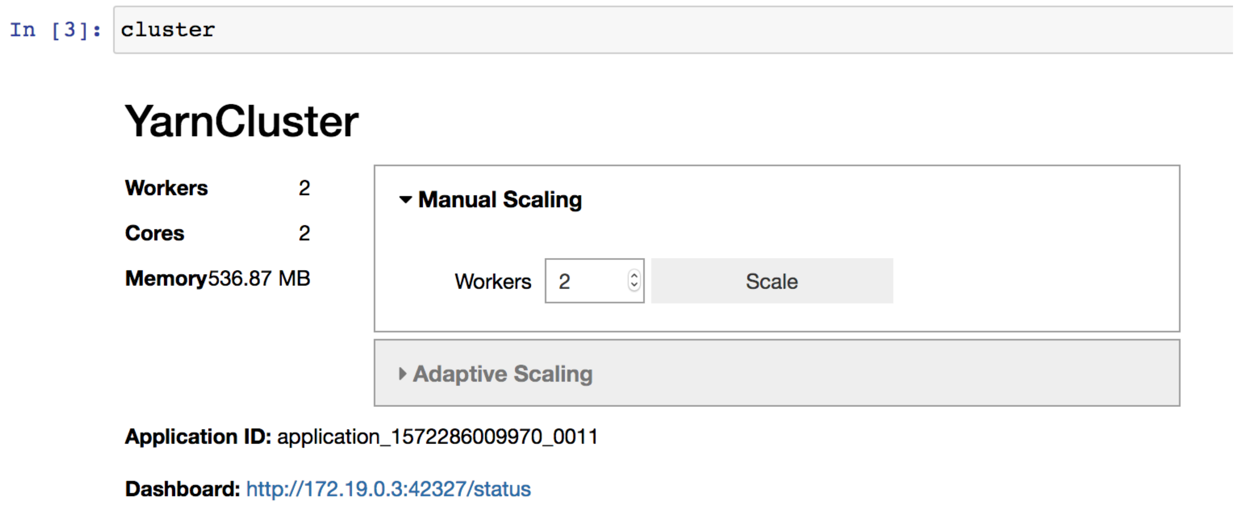
By default no workers are started on cluster creation. To change the number of workers, use the `YarnCluster.scale()` method. When scaling up, new workers will be requested from YARN. When scaling down, workers will be intelligently selected and scaled down gracefully, freeing up resources.

```
# Scale up to 10 workers
cluster.scale(10)

# ...

# Scale back down to 2 workers
cluster.scale(2)
```

Alternatively, you can enable adaptive scaling using the `YarnCluster.adapt()` method. When enabled, the cluster will scale up and down automatically depending on usage. Here we turn on adaptive scaling, bounded at a minimum of 2 workers and a maximum of 10 workers.

```
# Adaptively scale between 2 and 10 workers
cluster.adapt(minimum=2, maximum=10)
```

If you're working interactively in a Jupyter Notebook or JupyterLab, you can also use the provided graphical interface to change the cluster size, instead of calling `YarnCluster.scale()` or `YarnCluster.adapt()` manually.



```
In [3]: cluster
```

YarnCluster

Workers	2
Cores	2
Memory	536.87 MB

Workers 2

Application ID: application_1572286009970_0011

Dashboard: http://172.19.0.3:42327/status

Normally the cluster will persist until the `YarnCluster` object is deleted. To be more explicit about when the cluster is shutdown, you can either use the cluster as a context manager, or manually call *`YarnCluster.shutdown()`*.

```
# Use ``YarnCluster`` as a context manager
with YarnCluster(...) as cluster:
    # The cluster will remain active inside this block,
    # and will be shutdown when the context exits.

# Or manually call `shutdown`
cluster = YarnCluster(...)
# ...
cluster.shutdown()
```

1.2 Managing Python Environments

We need to ensure that the libraries used on the Yarn cluster are the same as what you are using locally. There are a few ways to specify this:

- The path to an archived environment (either conda or virtual environments)
- The path to a Conda environment (as `conda:///...`)
- The path to a virtual environment (as `venv:///...`)
- The path to a python executable (as `python:///...`)

Note that when not using an archive, the provided path must be valid on all nodes in the cluster.

1.2.1 Using Archived Python Environments

The most common way to use `dask-yarn` is to distribute an archived Python environment throughout the YARN cluster as part of the application. Packaging the environment for distribution is typically handled using

- conda-pack for Conda environments
- venv-pack for virtual environments

These environments can contain any Python packages you might need, but require `dask-yarn` (and its dependencies) at a minimum.

Archiving Conda Environments Using Conda-Pack

You can package a conda environment using conda-pack.

```
$ conda create -n my-env dask-yarn scikit-learn          # Create an environment

$ conda activate my-env                                  # Activate the environment

$ conda-pack                                             # Package environment
Collecting packages...
Packing environment at '/home/username/miniconda/envs/my-env' to 'my-env.tar.gz'
[########################################] | 100% Completed |  12.2s
```

Archiving Virtual Environments Using Venv-Pack

You can package a virtual environment using venv-pack. The virtual environment can be created using either venv or virtualenv. Note that the python linked to in the virtual environment must exist and be accessible on every node in the YARN cluster. If the environment was created with a different Python, you can change the link path using the `--python-prefix` flag. For more information see the venv-pack documentation.

```
$ python -m venv my_env                     # Create an environment using venv
$ python -m virtualenv my_env               # Or create an environment using␣
↪virtualenv

$ source my_env/bin/activate                # Activate the environment

$ pip install dask-yarn scikit-learn        # Install some packages

$ venv-pack                                 # Package environment
Collecting packages...
Packing environment at '/home/username/my-env' to 'my-env.tar.gz'
[########################################] | 100% Completed |  8.3s
```

Specifying the Archived Environment

You can now start a cluster with the packaged environment by passing the path to the constructor, e.g. `YarnCluster(environment='my-env.tar.gz', ...)`.

Note that if the environment is a local file, the archive will be automatically uploaded to a temporary directory on HDFS before starting the application. If you find yourself reusing the same environment multiple times, you may want to upload the environment to HDFS once beforehand to avoid repeating this process for each cluster (the environment is then specified as `hdfs:///path/to/my-env.tar.gz`).

After startup you may want to verify that your versions match with the following:

```python
from dask_yarn import YarnCluster
from dask.distributed import Client

cluster = YarnCluster(environment='my-env.tar.gz')
client = Client(cluster)
client.get_versions(check=True)  # check that versions match between all nodes
```

1.2.2 Using Python Environments Local to Each Node

Alternatively, you can specify the path to a conda environment, virtual environment, or Python executable that is already found on each node:

```
from dask_yarn import YarnCluster

# Use a conda environment at /path/to/my/conda/env
cluster = YarnCluster(environment='conda:///path/to/my/conda/env')

# Use a virtual environment at /path/to/my/virtual/env
cluster = YarnCluster(environment='venv:///path/to/my/virtual/env')

# Use a Python executable at /path/to/my/python
cluster = YarnCluster(environment='python:///path/to/my/python')
```

As before, these environments can have any Python packages, but must include `dask-yarn` (and its dependencies) at a minimum. It's also *very important* that these environments are uniform across all nodes; mismatched environments can lead to hard to diagnose issues. To check this, you can use the `Client.get_versions` method:

```
from dask.distributed import Client

client = Client(cluster)
client.get_versions(check=True)  # check that versions match between all nodes
```

1.3 Configuration

Specifying all parameters to the YarnCluster constructor every time may be error prone, especially when sharing this workflow with new users. Alternatively, you can provide defaults in a configuration file, traditionally held in `~/.config/dask/yarn.yaml` or `/etc/dask/yarn.yaml`. Note that this configuration is *optional*, and only changes the defaults when not specified in the constructor. You only need to set the fields you care about, unset fields will fall back to the *default configuration*.

Example:

```
# ~/.config/dask/yarn.yaml
yarn:
  name: dask                   # Application name
  queue: default               # Yarn queue to deploy to

  environment: /path/to/my-env.tar.gz

  scheduler:                   # Specifications of scheduler container
    vcores: 1
    memory: 4GiB

  worker:                     # Specifications of worker containers
    vcores: 2
    memory: 8GiB
```

Users can now create YarnClusters without specifying any additional information.

```
from dask_yarn import YarnCluster

```

(continues on next page)

(continued from previous page)

```
cluster = YarnCluster()
cluster.scale(20)
```

For more information on Dask configuration see the Dask configuration documentation.

1.3.1 Providing a Custom Skein Specification

Sometimes you'll need more control over the deployment than is provided by the above configuration fields. In this case you can provide the path to a custom Skein specification to the `yarn.specification` field. If this field is present in the configuration, it will be used as long as no parameters are passed to the *`YarnCluster`* constructor. Note that this is equivalent to calling *`YarnCluster.from_specification()`* programatically.

```
# /home/username/.config/dask/yarn.yaml
yarn:
  specification: /path/to/spec.yaml
```

The specification requires at least one Service named `dask.worker` which describes how to start a single worker. If an additional service `dask.scheduler` is provided, this will be assumed to start the scheduler. If `dask.scheduler` isn't present, a scheduler will be started locally instead.

In the `script` section for each service, the appropriate `dask-yarn` *CLI Docs* command should be used:

- `dask-yarn services worker` to start the worker
- `dask-yarn services scheduler` to start the worker

Beyond that, you have full flexibility for how to define a specification. See the Skein documentation for more information. A few examples are provided below:

Example: deploy-mode local with node_label restrictions

This specification is similar to that created automatically when `deploy_mode='local'` is specified (scheduler runs locally, only worker service specified), except it adds node_label restrictions for the workers. Here we restrict workers to run only on nodes labeled as GPU.

```
# /path/to/spec.yaml
name: dask
queue: myqueue

services:
  dask.worker:
    # Restrict workers to GPU nodes only
    node_label: GPU
    # Don't start any workers initially
    instances: 0
    # Workers can infinite number of times
    max_restarts: -1
    # Restrict workers to 4 GiB and 2 cores each
    resources:
      memory: 4 GiB
      vcores: 2
    # Distribute this python environment to every worker node
    files:
      environment: /path/to/my/environment.tar.gz
    # The bash script to start the worker
```

(continues on next page)

(continued from previous page)

```
    # Here we activate the environment, then start the worker
    script: |
      source environment/bin/activate
      dask-yarn services worker
```

Example: deploy-mode remote with custom setup

This specification is similar to that created automatically when `deploy_mode='remote'` is specified (both scheduler and worker run inside YARN containers), except it runs an initialization script before starting each service.

```
# /path/to/spec.yaml
name: dask
queue: myqueue

services:
  dask.scheduler:
    # Restrict scheduler to 2 GiB and 1 core
    resources:
      memory: 2 GiB
      vcores: 1
    # The bash script to start the scheduler.
    # Here we have dask-yarn already installed on the node,
    # and also run a custom script before starting the service
    script: |
      some-custom-initialization-script
      dask-yarn services worker

  dask.worker:
    # Don't start any workers initially
    instances: 0
    # Workers can infinite number of times
    max_restarts: -1
    # Workers should only be started after the scheduler starts
    depends:
      - dask.scheduler
    # Restrict workers to 4 GiB and 2 cores each
    resources:
      memory: 4 GiB
      vcores: 2
    # The bash script to start the worker.
    # Here we have dask-yarn already installed on the node,
    # and also run a custom script before starting the service
    script: |
      some-custom-initialization-script
      dask-yarn services worker
```

1.3.2 Default Configuration

The default configuration file is as follows

```
yarn:
  specification: null        # A path to a skein specification yaml file.
                             # Overrides the following configuration if given.
```

(continues on next page)

(continued from previous page)

```
name: dask                 # Application name
queue: default             # Yarn queue to deploy to
deploy-mode: remote        # The deploy mode to use (either remote or local)
environment: null          # The Python environment to use
tags: []                   # List of strings to tag applications
user: ''                   # The user to submit the application on behalf of,
                           # leave as empty string for current user.

host: "0.0.0.0"            # The scheduler host, when in deploy-mode=local
port: 0                    # The scheduler port, when in deploy-mode=local
dashboard-address: ":0"    # The dashboard address, when in deploy-mode=local

scheduler:                 # Specifications of scheduler container
  vcores: 1
  memory: 2GiB
  gpus: 0                 # Number of GPUs requested

worker:                   # Specifications of worker containers
  vcores: 1
  memory: 2GiB
  count: 0                # Number of workers to start on initialization
  restarts: -1            # Allowed number of restarts, -1 for unlimited
  env: {}                 # A map of environment variables to set on the worker
  gpus: 0                 # Number of GPUs requested
  worker_class: "dask.distributed.Nanny" # The kind of worker to launch
  worker_options: {}      # A map of options to pass to the worker

client:                   # Specification of client container
  vcores: 1
  memory: 2GiB
  gpus: 0                 # Number of GPUs requested
  env: {}                 # A map of environment variables ot set on the client
```

1.4 Submitting Applications

Warning: The submission API is experimental and may change between versions

Sometimes you have Dask Application you want to deploy completely on YARN, without having a corresponding process running on an edge node. This may come up with production applications deployed automatically, or long running jobs you don't want to consume edge node resources.

To handle these cases, dask-yarn provides a *CLI Docs* that can be used to submit applications to be run on the YARN cluster asynchronously. There are three commands that may be useful here:

- dask-yarn submit: submit an application to the YARN cluster
- dask-yarn status: check on the status of an application
- dask-yarn kill: kill a running application

1.4.1 Submitting an Application

To prepare an application to be submitted using `dask-yarn submit`, you need to change the creation of your *YarnCluster* from using the constructor to using *YarnCluster.from_current()*.

```
# Replace this
cluster = YarnCluster(...)

# with this
cluster = YarnCluster.from_current()
```

This is because the script won't be run until the cluster is already created - at that point configuration passed to the *YarnCluster* constructor won't be useful. Cluster configuration is instead passed via the `dask-yarn submit` CLI (note that as before, the cluster can be scaled dynamically after creation).

```
# Submit `myscript.py` to run on a dask cluster with 8 workers,
# each with 2 cores and 4 GiB
$ dask-yarn submit \
  --environment my_env.tar.gz \
  --worker-count 8 \
  --worker-vcores 2 \
  --worker-memory 4GiB \
  myscript.py

application_1538148161343_0051
```

This outputs a YARN Application ID, which can be used with other YARN tools.

1.4.2 Checking Application Status

Submitted application status can be checked using the YARN Web UI, or programmatically using `dask-yarn status`. This command takes one parameter - the application id.

```
$ dask-yarn status application_1538148161343_0051
APPLICATION_ID                  NAME    STATE       STATUS      CONTAINERS   ␣
↪VCORES     MEMORY     RUNTIME
application_1538148161343_0051  dask    RUNNING     UNDEFINED   9            17   ␣
↪      33792      6m
```

1.4.3 Killing a Running Application

Submitted applications normally run until completion. If you need to terminate one before then, you can use the `dask-yarn kill` command. This command takes one parameter - the application id.

```
$ dask-yarn kill application_1538148161343_0051
```

1.4.4 Accessing the Application Logs

Application logs can be retrieved a few ways:

- The logs of running applications can be viewed using the Skein Web UI (`dask-yarn` is built using Skein).
- The logs of completed applications can be viewed using the `yarn logs` command.

```
$ yarn logs -applicationId application_1538148161343_0051
```

1.5 Deploying on Amazon EMR

Amazon Elastic MapReduce (EMR) is a web service for creating a cloud-hosted Hadoop cluster.

Dask-Yarn works out-of-the-box on Amazon EMR, following the *Quickstart* as written should get you up and running fine. We recommend doing the installation step as part of a bootstrap action.

For a curated installation, we also provide an example bootstrap action for installing Dask and Jupyter on cluster startup. This script is heavily commented, and can be used as an example if you need a more customized installation.

Here we provide a brief walkthrough of a workflow we've found useful when deploying Dask on Amazon EMR:

1.5.1 Configure the EMR Cluster

The EMR documentation contains an example showing how to configure and start an EMR cluster. We recommend referencing their documentation. A few notes on recommended settings:

- If you plan to use a bootstrap action, you'll need to follow the `Go to advanced options` link on the `Create Cluster` page - this feature is not available under `Quick Options`.
- When chosing which applications to install, `dask-yarn` only requires a Hadoop installation, all other applications are optional depending on your workflow.

While this configuration can take some time and thought, the next time you want to start a Dask cluster on EMR you can clone this cluster to reuse the configuration.

1.5.2 Add a Bootstrap Action

To make installation on a clean cluster easier, we recommend scripting the installation as part of a bootstrap action. For a curated install, we provide an example bootstrap action that you may use. This script does the following:

- Installs miniconda.
- Installs `dask`, `distributed`, `dask-yarn`, pyarrow, and s3fs. This list of packages can be extended using the `--conda-packages` flag.
- Packages the environment with conda-pack for distribution to the workers.
- Optionally installs and starts a Jupyter Notebook server running on port 8888. This can be disabled with the `--no-jupyter` flag. The password for the notebook server can be set with the `--password` option, the default is `dask-user`.

If you require a more customized install than this, you may wish to provide your own script. The example script is heavily commented, and should hopefully provide enough of a reference for your use.

To use the script, follow AWS's documentation on using bootstrap actions. You'll need to upload the script to an S3 bucket accessible to your project first.

1.5.3 Start the EMR Cluster

Once you've finished configuring your cluster, you can start it with the `Create Cluster` button. This may take a while (~10 mins), depending on your settings.

1.5.4 Connect to the EMR Cluster

Once the cluster is running, you'll want to connect to it. Depending on your EC2 security groups settings, you may have direct access to the cluster, or you may need to start an ssh tunnel (default). For either of these you'll need to know the public DNS name of your master node. This address looks like ec2-###-##-##-###.compute-1.amazonaws.com, and can be found by following the AWS documentation.

Direct Access

If you have direct access to the cluster, you should be able to access the resource-manager WebUI at <public-dns-name>:8088. If you used our provided bootstrap action, the Jupyter Notebook should be available at <public-dns-name>:8888.

Using an SSH Tunnel

If you don't have direct access, you'll need to start an SSH tunnel to access the Web UIs or the Jupyter Notebook. For more information, see the AWS documentation.

If you used our provided bootstrap action, the Jupyter Notebook should be available at <public-dns-name>:8888, and can be accessed by starting a SSH tunnel via:

```
$ ssh -i ~/mykeypair.pem -L 8888:<public-dns-name>:8888 hadoop@<public-dns-name>
```

where ~/mykeypair.pem is your .pem file, and <public-dns-name> is the public DNS name of your master node.

1.5.5 Create a Dask Cluster

At this point you should have access to a running EMR cluster, with Dask and its dependencies installed. To create a new Dask cluster running inside the EMR cluster, create an instance of *YarnCluster*. If you didn't use our bootstrap action, you'll also need to provide a path to your packaged environment (see *Managing Python Environments* for more information).

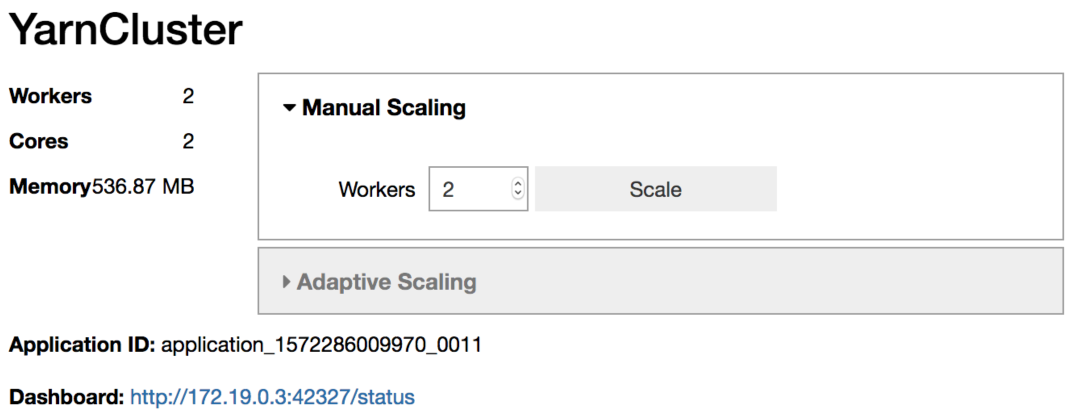
```
from dask_yarn import YarnCluster
from dask.distributed import Client

# Create a cluster
cluster = YarnCluster()

# Connect to the cluster
client = Client(cluster)
```

By default no workers are started on cluster creation. To change the number of workers, use the *YarnCluster.scale()* method. When scaling up, new workers will be requested from YARN. When scaling down, workers will be intelligently selected and scaled down gracefully, freeing up resources.

```
# Scale up to 10 workers
cluster.scale(10)

# ...

# Scale back down to 2 workers
cluster.scale(2)
```

If you're working interactively in a Jupyter Notebook you can also use the provided graphical interface to change the cluster size.

```
In [3]: cluster
```



YarnCluster

Workers 2

Cores 2

Memory536.87 MB

Application ID: application_1572286009970_0011

Dashboard: http://172.19.0.3:42327/status

If you used our bootstrap action, the dask dashboard will also be available, and the link included in the cluster widget above.

1.5.6 Shutdown the EMR Cluster

You can start, scale, and stop many *Dask* clusters within a single EMR cluster. When you're finally done doing your work, you'll want to shutdown the whole EMR cluster to conserve resources. See the AWS documentation for more information.

1.6 Deploying on Google Cloud Dataproc

Dataproc is Google Cloud's hosted service for creating Apache Hadoop and Apache Spark clusters. Dataproc supports a series of open-source initialization actions that allows installation of a wide range of open source tools when creating a cluster. In particular, the following instructions will guide you through creating a Dataproc cluster with Dask and Dask-Yarn installed and configured for you. This tutorial is loosely adapted from the README for the Dask initialization action.

1.6.1 What the Initialization Action is Doing

The initialization action installation script does several things:

- It accepts a metadata parameter for configuring your cluster to use Dask with either its standalone scheduler or with Dask-Yarn to utilize Yarn.
- For the `yarn` configuration, this script installs `dask` and `dask-yarn` on all machines and adds a baseline Skein config file. This file tells each machine where to locate the Dask-Yarn environment, as well as how many workers to use by default: 2. This way, you can get started with `dask-yarn` by simply creating a `YarnCluster` object without providing any parameters. Dask relies on using Yarn to schedule its tasks.
- For the `standalone` configuration, this script installs `dask` and configures the cluster to use the Dask scheduler for managing Dask workloads.
- The Dataproc service itself provides support for web UIs such as Jupyter and the Dask web UIs. This will be explained in more detail below.

1.6.2 Configuring your Dataproc Cluster

There are several ways to create a Dataproc cluster. This tutorial will focus on using the gcloud SDK to do so.

First, you'll need to create a GCP Project. Please follow the instructions here to do so.

Decide on a name for your Dataproc cluster. Then, pick a geographic region to place your cluster in, ideally one close to you.

The following command will create a cluster for the dask-yarn configuration.

```
CLUSTER_NAME=<cluster-name>
REGION=<region>
gcloud dataproc clusters create ${CLUSTER_NAME} \
  --region ${REGION} \
  --master-machine-type n1-standard-16 \
  --worker-machine-type n1-standard-16 \
  --image-version preview \
  --initialization-actions gs://goog-dataproc-initialization-actions-${REGION}/dask/
↪dask.sh \
  --metadata dask-runtime=yarn \
  --optional-components JUPYTER \
  --enable-component-gateway
```

To break down this command:

- `gcloud dataproc clusters create ${CLUSTER_NAME}` uses the gcloud sdk to to create a Dataproc cluster.
- `--region ${REGION}` specifies the cluster region.
- `--master-machine-type` and `worker-machine-type` allow configuration of CPUs and RAM via different types of machines.
- `image-version preview` specifies the Dataproc image version. You'll use the latest preview image of Dataproc for the most up-to-date features.
- `--initialization-actions gs://goog-dataproc-initialization-actions-${REGION}/dask/dask.sh` specifies the initialization actions to install on the cluster. You can add as many as you'd like via a comma-separated list.
- `--metadata dask-runtime=yarn` specifies to configure your cluster with Dask configured for use with `yarn`.
- `--optional-components JUPYTER` configures the cluster with the Jupyter optional component to access Jupyter notebooks running on the cluster. Like initialization actions, you can add as many optional components as you'd like. These differ from initialization actions in that they come with first-class support from the Dataproc service, but there are less options available.
- `--enable-component-gateway` allows you to bypass needing an SSH tunnel for a certain predetermined list of web UIs on your cluster, such as Jupyter and the Yarn ApplicationMaster, by connecting directly through the Dataproc web console.

1.6.3 Connecting to your cluster

You can access your cluster several different ways. If you configured your cluster with a notebook service such as Jupyter or Zeppelin and enable component gateway (explained above), you can access these by navigating to your clusters page, clicking on the name of your cluster and clicking on the **Web Interfaces** tab to access your web UIs.

You can also ssh into your cluster. You can do this via the Dataproc web console: from the clusters page, click on your cluster name, then **VM Instances** and click **SSH** next to the master node.

Additionally, you can also use the gcloud sdk to SSH onto your cluster. First, locate the zone that your cluster is in. This will be the region you specified earlier but with a letter attached to it, such as `us-central1-b`. To locate your cluster's zone, you can find this on the clusters page next to your cluster. This was determined via Dataproc's Auto Zone feature, but you can choose any zone to place your cluster by adding the `--zone` flag when creating a new cluster.

```
gcloud compute ssh ${CLUSTER_NAME}-m --zone ${ZONE}
```

Once connected, either via a Jupyter notebook or via ssh, try running some code. If your cluster is configured with Dask-Yarn:

```
from dask_yarn import YarnCluster
from dask.distributed import Client
import dask.array as da

import numpy as np

cluster = YarnCluster()
client = Client(cluster)

cluster.adapt()  # Dynamically scale Dask resources

x = da.sum(np.ones(5))
x.compute()
```

If your cluster is configured with the standalone scheduler:

```
from dask.distributed import Client
import dask.array as da

import numpy as np

client = Client("localhost:8786")

x = da.sum(np.ones(5))
x.compute()
```

1.6.4 Monitoring Dask Jobs

You can monitor your Dask applications using Web UIs, depending on the runtime you are using.

For yarn mode, you can access the Skein Web UI via the YARN ResourceManager. To access the YARN ResourceManager, create your cluster with component gateway enabled or create an SSH tunnel. You can then access the Skein Web UI by following these instructions.

For standalone mode, you can access the native Dask UI. Create an SSH tunnel to access the Dask UI on port 8787.

1.6.5 Deleting your Dataproc Cluster

You can delete your cluster when you are done with it by running the following command:

```
gcloud dataproc clusters delete ${CLUSTER_NAME} --region ${REGION}
```

1.6.6 Further Information

Please refer to the Dataproc documentation for more information on using Dataproc.

1.7 API Docs

class dask_yarn.**YarnCluster**(*environment=None*, *n_workers=None*, *worker_vcores=None*, *worker_memory=None*, *worker_restarts=None*, *worker_env=None*, *worker_class=None*, *worker_options=None*, *worker_gpus=None*, *scheduler_vcores=None*, *scheduler_gpus=None*, *scheduler_memory=None*, *deploy_mode=None*, *name=None*, *queue=None*, *tags=None*, *user=None*, *host=None*, *port=None*, *dashboard_address=None*, *skein_client=None*, *asynchronous=False*, *loop=None*)

Start a Dask cluster on YARN.

You can define default values for this in Dask's `yarn.yaml` configuration file. See http://docs.dask.org/en/latest/configuration.html for more information.

Parameters

environment [str, optional] The Python environment to use. Can be one of the following:

- A path to an archived Python environment
- A path to a conda environment, specified as *conda:///...*
- A path to a virtual environment, specified as *venv:///...*
- A path to a python executable, specifed as *python:///...*

Note that if not an archive, the paths specified must be valid on all nodes in the cluster.

n_workers [int, optional] The number of workers to initially start.

worker_vcores [int, optional] The number of virtual cores to allocate per worker.

worker_memory [str, optional] The amount of memory to allocate per worker. Accepts a unit suffix (e.g. '2 GiB' or '4096 MiB'). Will be rounded up to the nearest MiB.

worker_restarts [int, optional] The maximum number of worker restarts to allow before failing the application. Default is unlimited.

worker_env [dict, optional] A mapping of environment variables to their values. These will be set in the worker containers before starting the dask workers.

worker_gpus [int, options] The number of gpus to allocate per worker

scheduler_vcores [int, optional] The number of virtual cores to allocate per scheduler.

scheduler_gpus [int, options] The number of gpus to allocate per scheduler

scheduler_memory [str, optional] The amount of memory to allocate to the scheduler. Accepts a unit suffix (e.g. '2 GiB' or '4096 MiB'). Will be rounded up to the nearest MiB.

deploy_mode [{'remote', 'local'}, optional] The deploy mode to use. If `'remote'`, the scheduler will be deployed in a YARN container. If `'local'`, the scheduler will run locally, which can be nice for debugging. Default is `'remote'`.

name [str, optional] The application name.

queue [str, optional] The queue to deploy to.

tags [sequence, optional] A set of strings to use as tags for this application.

user [str, optional] The user to submit the application on behalf of. Default is the current user - submitting as a different user requires user permissions, see the YARN documentation for more information.

host [str, optional] Host address on which the scheduler will listen. Only used if `deploy_mode='local'`. Defaults to `'0.0.0.0'`.

port [int, optional] The port on which the scheduler will listen. Only used if `deploy_mode='local'`. Defaults to `0` for a random port.

dashboard_address [str] Address on which to the dashboard server will listen. Only used if `deploy_mode='local'`. Defaults to ':0' for a random port.

skein_client [skein.Client, optional] The `skein.Client` to use. If not provided, one will be started.

asynchronous [bool, optional] If true, starts the cluster in asynchronous mode, where it can be used in other async code.

loop [IOLoop, optional] The IOLoop instance to use. Defaults to the current loop in asynchronous mode, otherwise a background loop is started.

Examples

```
>>> cluster = YarnCluster(environment='my-env.tar.gz', ...)
>>> cluster.scale(10)
```

adapt (*minimum=0*, *maximum=inf*, *interval='1s'*, *wait_count=3*, *target_duration='5s'*, ***kwargs*)

Turn on adaptivity

This scales Dask clusters automatically based on scheduler activity.

Parameters

minimum [int, optional] Minimum number of workers. Defaults to `0`.

maximum [int, optional] Maximum number of workers. Defaults to `inf`.

interval [timedelta or str, optional] Time between worker add/remove recommendations.

wait_count [int, optional] Number of consecutive times that a worker should be suggested for removal before we remove it.

target_duration [timedelta or str, optional] Amount of time we want a computation to take. This affects how aggressively we scale up.

****kwargs :** Additional parameters to pass to `distributed.Scheduler.workers_to_close`.

Examples

```
>>> cluster.adapt(minimum=0, maximum=10)
```

close (***kwargs*)

Close this cluster. An alias for `shutdown`.

See also:

shutdown

dashboard_link
Link to the dask dashboard. None if dashboard isn't running

classmethod from_application_id(*app_id*, *skein_client=None*, *asynchronous=False*, *loop=None*)
Connect to an existing `YarnCluster` with a given application id.

Parameters

app_id [str] The existing cluster's application id.

skein_client [skein.Client] The `skein.Client` to use. If not provided, one will be started.

asynchronous [bool, optional] If true, starts the cluster in asynchronous mode, where it can be used in other async code.

loop [IOLoop, optional] The IOLoop instance to use. Defaults to the current loop in asynchronous mode, otherwise a background loop is started.

Returns

YarnCluster

classmethod from_current(*asynchronous=False*, *loop=None*)
Connect to an existing `YarnCluster` from inside the cluster.

Parameters

asynchronous [bool, optional] If true, starts the cluster in asynchronous mode, where it can be used in other async code.

loop [IOLoop, optional] The IOLoop instance to use. Defaults to the current loop in asynchronous mode, otherwise a background loop is started.

Returns

YarnCluster

classmethod from_specification(*spec*, *skein_client=None*, *asynchronous=False*, *loop=None*)
Start a dask cluster from a skein specification.

Parameters

spec [skein.ApplicationSpec, dict, or filename] The application specification to use. Must define at least one service: `'dask.worker'`. If no `'dask.scheduler'` service is defined, a scheduler will be started locally.

skein_client [skein.Client, optional] The `skein.Client` to use. If not provided, one will be started.

asynchronous [bool, optional] If true, starts the cluster in asynchronous mode, where it can be used in other async code.

loop [IOLoop, optional] The IOLoop instance to use. Defaults to the current loop in asynchronous mode, otherwise a background loop is started.

logs(*scheduler=True*, *workers=True*)
Return logs for the scheduler and/or workers

Parameters

scheduler [boolean, optional] Whether or not to collect logs for the scheduler

workers [boolean or iterable, optional] A list of worker addresses to select. Defaults to all workers if `True` or no workers if `False`

Returns

logs [dict] A dictionary of name -> logs.

scale(*n*)

Scale cluster to n workers.

Parameters

n [int] Target number of workers

Examples

```
>>> cluster.scale(10)  # scale cluster to ten workers
```

shutdown(*status='SUCCEEDED'*, *diagnostics=None*)

Shutdown the application.

Parameters

status [{'SUCCEEDED', 'FAILED', 'KILLED'}, optional] The yarn application exit status.

diagnostics [str, optional] The application exit message, usually used for diagnosing failures. Can be seen in the YARN Web UI for completed applications under "diagnostics". If not provided, a default will be used.

workers()

A list of all currently running worker containers.

1.8 CLI Docs

Warning: The CLI is experimental and may change between versions

1.8.1 dask-yarn

Deploy Dask on Apache YARN

```
usage: dask-yarn [--help] [--version] command ...
```

--help, -h

Show this help message then exit

--version

Show version then exit

dask-yarn kill

Kill a Dask application

```
usage: dask-yarn kill [--help] APP_ID
```

app_id
The application id

--help, -h
Show this help message then exit

dask-yarn services

Manage Dask services

```
usage: dask-yarn services [--help] command ...
```

--help, -h
Show this help message then exit

dask-yarn services client

Start a Dask client process

```
usage: dask-yarn services client [--help] script [args...]
```

script
Path to a Python script to run.

args
Any additional arguments to forward to *script*

--help, -h
Show this help message then exit

dask-yarn services scheduler

Start a Dask scheduler process

```
usage: dask-yarn services scheduler [--help]
```

--help, -h
Show this help message then exit

dask-yarn services worker

Start a Dask worker process

```
usage: dask-yarn services worker [--nthreads NTHREADS]
                                 [--memory_limit MEMORY_LIMIT] [--help]
```

--nthreads `<nthreads>`
Number of threads. Defaults to number of vcores in container

--memory_limit `<memory_limit>`
Maximum memory available to the worker. This can be an integer (in bytes), a string (like '5 GiB' or '500 MiB'), or 0 (no memory management). Defaults to the container memory limit.

--help, -h
Show this help message then exit

dask-yarn status

Check the status of a submitted Dask application

```
usage: dask-yarn status [--help] APP_ID
```

app_id
The application id

--help, -h
Show this help message then exit

dask-yarn submit

Submit a Dask application to a YARN cluster

```
usage: dask-yarn submit [--name NAME] [--queue QUEUE] [--user USER]
                        [--tags TAGS] [--environment ENVIRONMENT]
                        [--deploy-mode DEPLOY_MODE]
                        [--worker-count WORKER_COUNT]
                        [--worker-vcores WORKER_VCORES]
                        [--worker-memory WORKER_MEMORY]
                        [--worker-restarts WORKER_RESTARTS]
                        [--worker-env WORKER_ENV]
                        [--client-vcores CLIENT_VCORES]
                        [--client-memory CLIENT_MEMORY]
                        [--client-env CLIENT_ENV]
                        [--scheduler-vcores SCHEDULER_VCORES]
                        [--scheduler-memory SCHEDULER_MEMORY]
                        [--temporary-security-credentials] [--help]
                        script [args...]
```

script
Path to a python script to run on the client

args
Any additional arguments to forward to *script*

--name <name>
The application name

--queue <queue>
The queue to deploy to

--user <user>
The user to submit the application on behalf of. Default is the current user - submitting as a different user requires proxy-user permissions.

--tags <tags>
A comma-separated list of strings to use as tags for this application.

--environment <environment>
Path to the Python environment to use. See the docs for more information

--deploy-mode <deploy_mode>
Either 'remote' (default) or 'local'. If 'remote', the scheduler and client will be deployed in a YARN container. If 'local', they will be run locally.

--worker-count <worker_count>
The number of workers to initially start.

--worker-vcores <worker_vcores>
The number of virtual cores to allocate per worker.

--worker-memory <worker_memory>
The amount of memory to allocate per worker. Accepts a unit suffix (e.g. '2 GiB' or '4096 MiB'). Will be rounded up to the nearest MiB.

--worker-restarts <worker_restarts>
The maximum number of worker restarts to allow before failing the application. Default is unlimited.

--worker-env <worker_env>
Environment variables to set on the workers. Pass a key-value pair like `--worker-env key=val`. May be used more than once.

--client-vcores <client_vcores>
The number of virtual cores to allocate for the client.

--client-memory <client_memory>
The amount of memory to allocate for the client. Accepts a unit suffix (e.g. '2 GiB' or '4096 MiB'). Will be rounded up to the nearest MiB.

--client-env <client_env>
Environment variables to set on the client. Pass a key-value pair like `--client-env key=val`. May be used more than once.

--scheduler-vcores <scheduler_vcores>
The number of virtual cores to allocate for the scheduler.

--scheduler-memory <scheduler_memory>
The amount of memory to allocate for the scheduler. Accepts a unit suffix (e.g. '2 GiB' or '4096 MiB'). Will be rounded up to the nearest MiB.

--temporary-security-credentials
Instead of using a consistent set of TLS credentials for all clusters, create a fresh set just for this application.

--help, -h
Show this help message then exit

Index