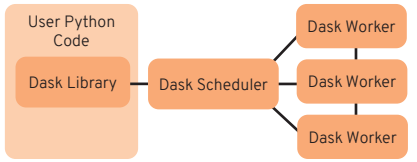
Dask Cheatsheet

Presented by SaturnCloud

What is Dask?

Dask is a Python framework for distributed computing. With Dask, you can easily have code run in parallel over multiple threads or a network of connected machines. Dask works by having the machine that calls it (the client) use the Dask libraries to pass the data and commands to a scheduler, which then in turn distributes the tasks to workers. The workers can be threads on the same machine as the client, or on separate machines entirely.

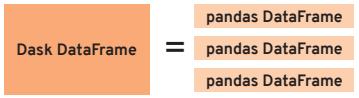


Data Collections

Data collections are Dask data types that can swap in for standard libraries to run on a distributed cluster. They are designed to mimic popular Python libraries.

DASK DATAFRAME

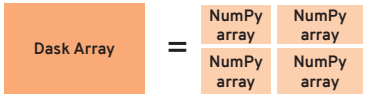
Dask DataFrames have equivalent functions for most pandas operations. This allows you to group, summarize, and more with a parallel backend.



A Dask DataFrame splits data into multiple pandas DataFrames on the backend.

DASK ARRAY

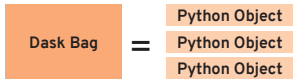
Dask Arrays are meant to mimic the functionality of NumPy arrays using a distributed backend (and common NumPy functions).



A Dask Array is a multi-dimensional collection of NumPy arrays

DASK BAG

Dask Bags are used for taking a collection of Python objects and processing them in parallel. You can apply functions, filter the collection, and group and combine.



A Dask Bag is an unordered sequence of arbitrary Python objects.

pandas

```
import pandas as pd
df = dd.read_csv('pet_data.csv')
df.groupby(df['breed'])['age'].mean()
```

Dask DataFrame

```
import dask.dataframe as dd
df = dd.read_csv('pet_data.csv')
df.groupby(df['breed'])['age'] \
    .mean().compute()
```

NumPy

```
import numpy as np
x = np.array(range(1000))
np.sin(x)
```

Dask Array

```
import numpy as np
import dask.array as da
x = np.array(range(1000))
x = da.from_array(x, chunks = 10)
da.sin(x).compute()
```

Base Python

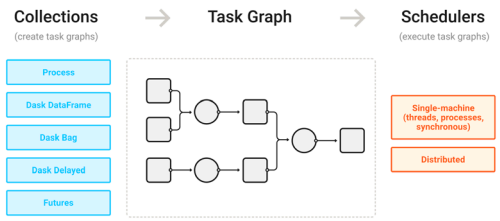
```
pets = choices(["cat","dog","rat"], k=50)
pets = map(lambda x: x.upper(), pets)
counts = Counter(pets).items()
top_2 = sorted(counts, key=lambda x: x[1])[-2:]
```

Dask Bag

```
import dask.bag as db
pets = choices(["cat", "dog"], k=50)
bag = db.from_sequence(pets)
bag.map(lambda x: x.upper()).frequencies() \
    .topk(2).compute()
```

Task Scheduling

Dask works by creating a *task graph*, a directed acyclic graph where each node is a Python function (tasks) and edges are when one task's output is another's input.



Source: Dask Community GitHub

SINGLE MACHINE SCHEDULERS

```
import dask
dask.config.set(scheduler='synchronous')
```

Set the scheduler using `dask.config.set`

`synchronous` - Uses a single thread (no parallelization)

`threads` - Computations done on local threads. Only parallelizes non-Python backend code

`processes` - Computations spread over local processes. Parallelized Python on single machine

MULTI-MACHINE SCHEDULERS

```
from dask.distributed import Client
client = Client()
```

Run over multiple machines using `dask.distributed`

The Dask distributed scheduler is more powerful but also complicated. It is required for using Dask over multiple machines, as well as for some functionality like Dask Futures. The distributed scheduler can be used locally if you want the additional functionality without the cluster.

Parallelizing Python Code

In addition to using the helpful data collections, users may also want to directly parallelize their code themselves. Dask has multiple methods for taking Python code and running it in a distributed manner:

DASK DELAYED

Dask Delayed allows you to take arbitrary functions and execute them in a non-sequential order. By converting functions to delayed versions, they will instead be executed lazily.

```
def times_2(x):
    return x * 2
a = dask.delayed(times_2)(10)
b = dask.delayed(times_2)(result)
b.compute() #returns 40
```

Use `dask.delayed` as a wrapper around a function to make it the delayed version (which returns a Delayed object). To get the value, use `.compute()`.

```
@dask.delayed
def plus_5(x):
    return x + 5
plus_5(plus_5(10)).compute()
```

For convenience, you can use the `@dask.delayed` decorator above a function.

DASK FUTURES

Dask Futures builds on the concurrent.futures module from Python. This allows you to run multiple tasks concurrently in Python, but unlike Dask Delayed the computations are immediate, rather than lazy.

```
from dask.distributed import Client
client = Client()
def sub_3(x):
    return x - 3
a = client.submit(sub_3, 12
a.result() #blocks until result
b = client.map(sub_3, range(10,20))
client.gather(b)#list to results
```

Submit individual tasks using `.submit()`, or apply a function using `.map()`. These return future objects or a list of future objects, respectively. To get the results, you can use `.result()` or `.gather()` for a list of futures.

Glossary

Client - the machine calling Dask
Cluster - set of connected machines to execute work
Dask Collection - a data type that uses Dask on the backend
Lazy - code that only runs when the result is needed
Local - the code running on the client
Scheduler - program coordinates executing task graph
Task - unit of Python code to execute
Task graph - network of how tasks relate
Worker - machine in the distributed cluster that executes tasks

Machine Learning with Dask

Dask is well-suited for machine learning. However, most common machine learning libraries do not natively support training on data distributed across workers, and so special libraries must be used:

DASK-ML

Dask-ML is a library of helper functions for using Dask for machine learning. It contains functions for common ML algorithms like linear regressions, as well as parallelizable tasks like crossvalidation and hyperparameter tuning.

```
dask_ml.model_selection.GridSearchCV()
dask_ml.model_selection.RandomizedSearchCV()
```

Dask-ML supplies drop-in replacements for Scikit-Learn functions for hyperparameter search. This allows you to test different combinations of parameters much more quickly than with Scikit-Learn since the backed is parallelized.

XGBOOST

The popular machine learning method XGBoost has built-in support for Dask, allowing the model to be trained across multiple workers concurrently.

```
dtrain = xgb.dask.DaskDMatrix(client, X, y)
output = xgb.dask.train(client,params,detrain,…)
```

With Dask and XGBoost, first create a special Dask version of the data (here X and y are Dask Arrays or Dask DataFrames). Also pass the Dask client. Then use the special XGBoost training function for Dask again passing the Dask client to it.

LIGHTGBM

This is a gradient-boosting tree-based ML framework, with a focus on faster training speeds and a lower memory usage. LightGBM has built in support for distributed training with Dask.

```
dX = dask.array.from_array(X, chunks=(100, 50))
dy = dask.array.from_array(y, chunks=(100,))
dask_model = lightgbm.DaskLGBMClassifier(n_estimators=10)
dask_model.fit(dX, dy)
```

Here LightGBM is being used for the DaskLGBMClassifier model, to classify Dask arrays. This allows the model to be trained on data spread over multiple workers.

Dask with GPUs

At its core, Dask doesn't distinguish between CPU and GPU computations. Dask can run anything so long as your workers have the correct libraries and hardware. That said, there are a number of frameworks well suited for using Dask with GPUs.

RAPIDS

RAPIDS allows for using modern machine learning methods directly on GPUs, and is compatible with Dask for situations with large datasets

```
import dask_cudf
from dask.distributed import Client
from cuml.dask.ensemble import
RandomForestClassifier
taxi = dask_cudf.read_csv("taxi.csv")
X = taxi[["puloc", "doloc", "count"]]
y = (taxi["tip"] > 1).astype("int32")
X, y = client.persist([X, y])
_ = wait([X, y])
rfc = RandomForestClassifier(n_estimators=100)
_ = rfc.fit(X, y)
```

By loading data using cudf, it will be stored for use by a GPU instead of a CPU, while still being compatible with Dask. From there, the cuml random forest algorithm which uses the GPU can be called. This workflow of using multiple workers with GPUs can be a far faster replacement replacement for ML on a single CPU

PYTORCH

You can use Dask with PyTorch, either to train many different models at once using Dask workers, or by having all the workers coordinate training the same model

```
from torch.nn.parallel \
    import DistributedDataParallel as DDP
def train():
    model = Model()
    model = model.to(torch.device(0))
    model = DDP(model)
    ...
futures = dispatch.run(client, train)
```

Using the DDP library from PyTorch a single model can be trained across multiple workers and GPUs. Dask can be used by having each Dask worker run a training function (here called train) to train the model on each GPU. The DDP library coordinates parameter fitting between the models. For more information watch this talk (scld.io/dasksummitgpus) and check out the Saturn Cloud DDP helper library (github.com/saturncloud/dask-pytorch-ddp).

Use Dask Locally

To quickly get started and develop, Dask can be run with a distributed cluster on the same machine as the client. Dask can parallelize across the different cores of the machine.

```
conda install dask
```

Install the Dask package from conda

By default Dask uses a single local scheduler, so just install the package and start running. Certain tasks benefit from using the newer distributed scheduler locally, which can be called from `dask.distributed`

```
from dask.distributed import Client
client = Client()
```

Use the distributed scheduler by creating a Dask client object with it

Using the local Dask environment can be a great way to write and debug code before shifting over to a multi-machine system.

Use Dask on a Cluster

If you'd like to run Dask on a cluster of workers (and don't want to have a system setup for you with cloud tools like Saturn Cloud), Dask provides a number of methods for creating clusters yourself

Kubernetes

With Kubernetes you can easily manage a Dask cluster, while having other services on the same cluster.

```
helm repo add dask https://helm.dask.org/
helm repo update
helm install my-dask dask/dask
```

YARN

Dask-Yarn can be used to deploy on YARN clusters, which is especially useful if you already have one from Spark or Hadoop

Command line

You can set up a Dask cluster manually by calling dask-scheduler on a machine, and then dask-worker on each machine you'd like to be a worker (and registering it to the scheduler)

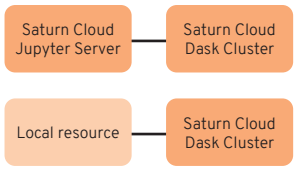
```
dask-scheduler
> Scheduler at:   tcp://192.0.0.100:8786
dask-worker tcp://192.0.0.100:8786
```

Use Dask on Saturn Cloud

Rather than having to set up and manage Dask cluster yourself, you can use Saturn Cloud to get started in seconds. The Saturn Cloud Hosted Free plan gives you 3 hours of free Dask usage a month (with GPUs!), and both paid and enterprise plans are also available. Go to saturncloud.io/signup to get started.

To use Dask, create a new Jupyter server and attach a Dask cluster to it. Once you've done that your Dask commands will execute on the external Dask cluster rather than the Jupyter server.

Methods for using Dask on Saturn Cloud



You can either connect to the Saturn Cloud Dask Cluster through the associated Jupyter Server, or you can connect directly to the Dask Cluster through a local resource like a laptop, or even a different machine in the cloud.

SaturnCloud