**DATA PIPELINING**

**By Team 1**

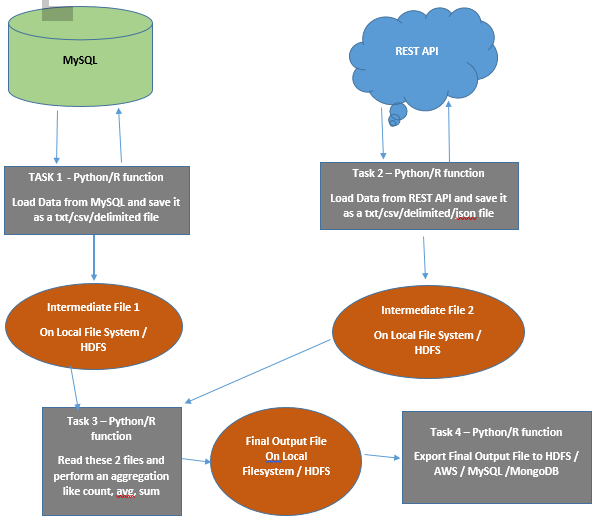
**( Manasi Dalvi & Vishal Satam )**

**Under the guidance of Prof. Srikanth Krishnamurthy**

**at Northeastern University**

**Bigdata pipelines:**

Data pipelines are used to structure the dataflow to modularize the code in Data Science projects. A typical Data Science process commences with data collection from various resources, proceeds with cleaning and filtering of the received data. The clean data is then structured into data repositories and machine learning is applied to explore & analyze the data to provide useful insights to the business.



**Sample pipeline**

*“Components of an Analytics Pipeline: [*[*Reference*](https://dzone.com/articles/how-to-build-an-analytic-pipeline)*]*

* *The major constituents of an Analytic pipeline will be as follows:*
* *The messaging system.*
* *Distribution of messages to various nodes for further processing.*
* *Analytic processing, to derive inferences from data. This will include the application of machine learning of data.*
* *Data storage system for storing results and related information.*
* *Interfaces or consumption of results data, e.g. visualization tools, alerts, etc.”*

There are various tools and workflow management systems encompassing the required components for efficient data flow. Few examples are, Airflow, Luigi, Dask, Celery,Pinball, Azkaban, Oozie, makefiles, etc

In this report we will focus on

* Makefiles
* Luigi,
* DASK,
* Celery
* Airflow

**GNU MAKE**

Make is a build automation tool that is used to compile source code into byte code. This tool is being used to build code since the times when C used to be the mainstream language.

For using Make , we need to create a text file and name it Makefile. This file is used to build the project. Make tool is generic, Since we write the shell commands that need to be executed as recipes in the Makefile, this tool can be used to create data processing pipelines that can call programs written in any language (Python, R, Java, etc).

Advantage of using Makefiles is that the source code of the individual python or R files doesn’t have to be modified.

These files can be created individually and then built in a sequential dependent manner using make files.

We have to specify 3 main things in the make files.

1. Target – This is the target file that the makefile builds eventually.
2. Dependencies - Every target has it’s dependencies and they need to be specified in the line where the target is defined by separated with spaces
3. Recipe – This is the code that needs to be executed for the target. This is usually a set of shell commands.

The default way to invoke make is calling the make command which attempts to build the first target by default. Otherwise we can also specify the target explicitly like

make 🡪 This command executes the Makefile from the present working directory and attempts to build the first target as specified in the Makefile.

make <target name>

We have to keep in mind that make is primarily used as a build tool but we can also use it to create python pipelines for processing data.

GNU Make is the default build utility that comes pre-installed by default in Linux and Mac.

You can type the following to get a list of command line options that can be provided along with the make command.

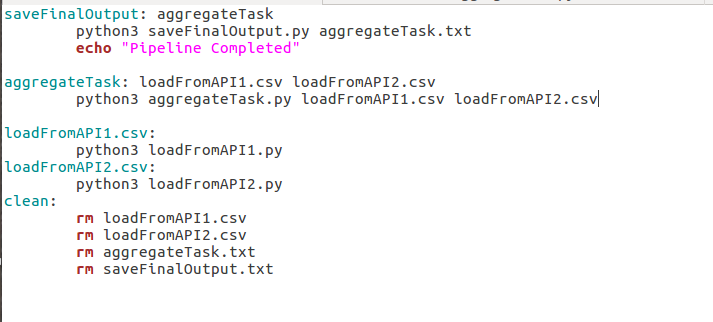
make -help

The following command prints the version of make that is currently installed in your OS

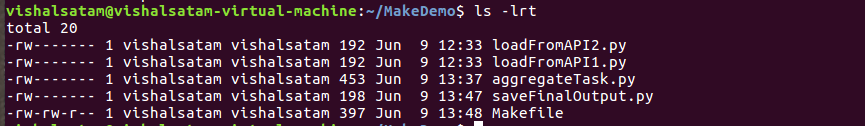
make -v

**Makefile**

This is how the Makefile is written. The main thing to note here is that the file must be named as Makefile.



**Folder Structure**



If we execute the make command, in this directory, make would try to build the first target available in the file. In our example, we have it as saveFinalOutput.

We can also specify targets that we want make to build by writing “*make* saveFinalOutput”

Or “*make clean*”

On executing the make command, the first target would be checked for the dependencies specified. In our case, we have specified aggregateTask as the dependency.

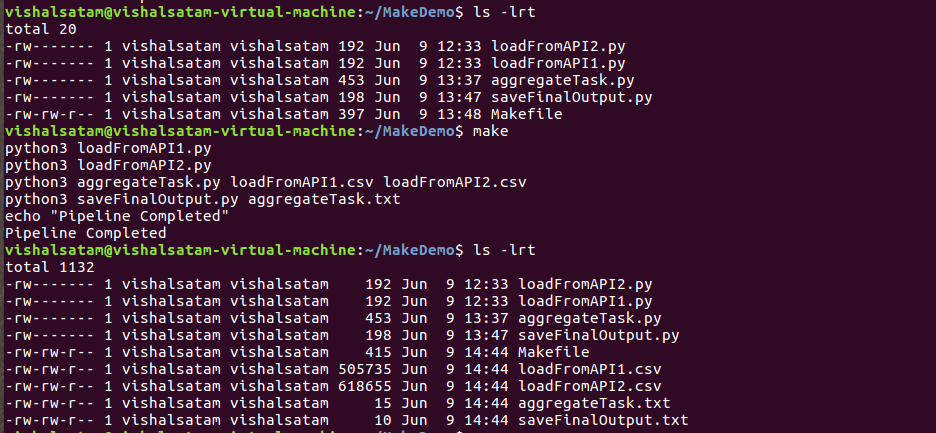
Make would then scan the file t see if the target with the name aggregateTask exists. (In case it doesn’t, then make would check if a file with that name exists in the current working directory.)

Since aggregateTask exists, make then checks for the dependencies specified for aggregateTask. Since we have specified loadFromAPI1 and loadFromAPI2 which exists as targets in the Makefile, these targets will be built first by executing the recipes(bash shell commands) mentioned for these targets.

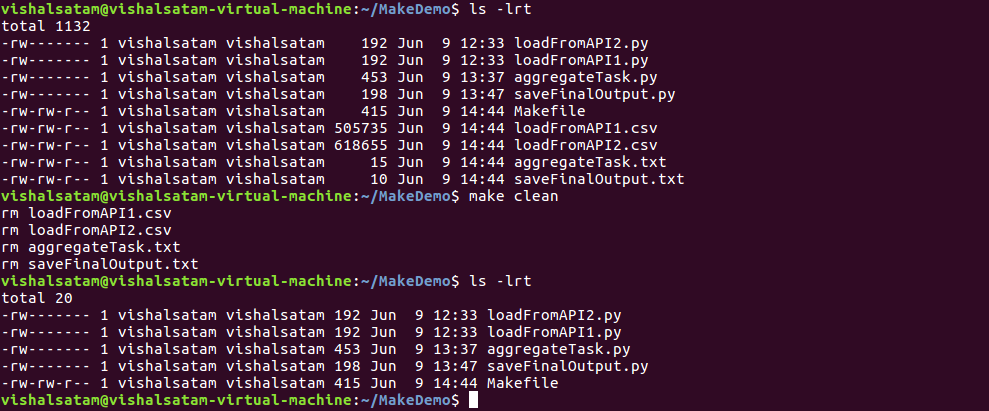
After building these 2 targets, make also checks to see if the files loadFromAPI1.csv and loadFromAPI2.csv exists as targets or as files in the folder. After the 2 earlier tasks run, these files will be available in the folder. Hence, make will continue to build aggregateTask now and execute it’s recipe (bash shell commands).

After aggregateTask is completed, make would then continue to build the saveFinalOutput target by executing it’s recipe (bash shell commands).

Executing “make”



Executing “make clean”

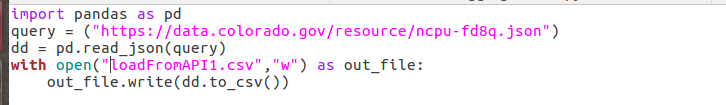


**Dependent modules**

Let ‘s have a look at a makefile and the folder structure in which this file exists.

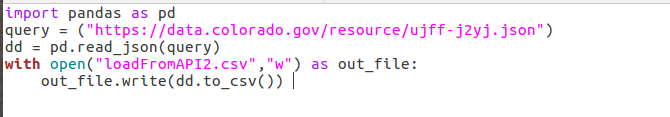
1. loadFromAPI1.py

This module loads data from a public API and saves the data in a csv file.



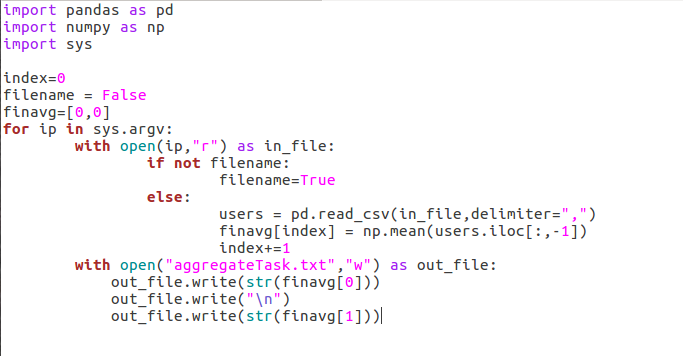
1. loadFromAPI2.py

This module loads data from a public API and saves the data in a csv file.



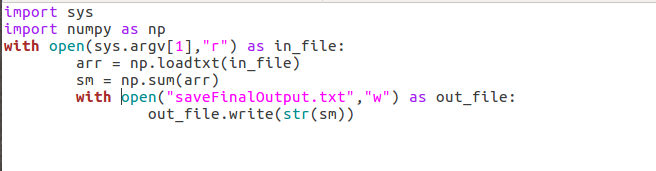
1. aggregateTask.py

This module reads the 2 csv files from the previous 2 steps and calculates the mean and finally saves it in a file aggregateTask.txt



1. saveFinalOutput.py

This module takes the sum of the 2 averages calculated in the previous step and saves it in another file saveFinaOutput.txt



**Additional information and links:**

Example of Python Make: <http://zmjones.com/make/>

Official site : <https://www.gnu.org/software/make/>

Blog for makefiles in python projects: <https://krzysztofzuraw.com/blog/2016/makefiles-in-python-projects.html>

Writing rules:

<ftp://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_4.html>

<https://www.youtube.com/watch?v=tSuEHEScwos>

<https://bsmith89.github.io/make-bml/>

Cleaning Standards:

<https://www.gnu.org/software/automake/manual/html_node/Clean.html>

**NMAKE**

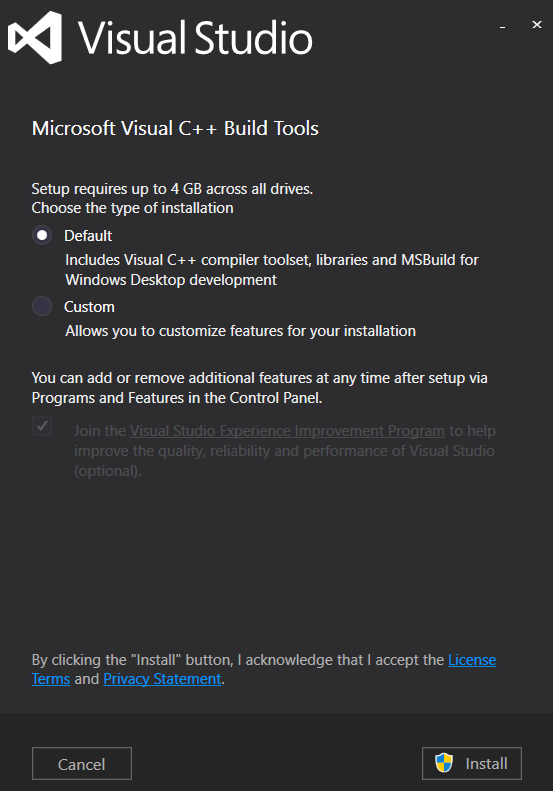
There is a NMake for windows. This is usually shipped as a part of Visual Studio.

The following line is quoted from <https://msdn.microsoft.com/en-us/library/dd9y37ha.aspx>

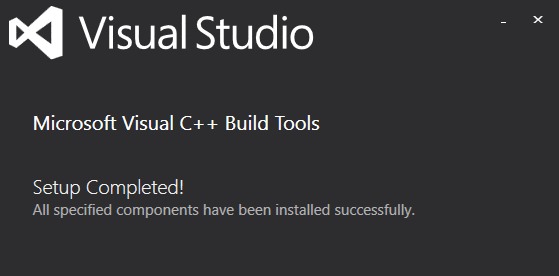
*“NMAKE is included when you install Visual Studio or the Visual C++ command-line build tools. It's not available separately. To download and install Visual Studio, see*[*VisualStudio.com*](https://www.visualstudio.com/)*. To install only the command-line tools, download and install*[*Microsoft Visual C++ Build Tools*](http://go.microsoft.com/fwlink/?LinkId=691126)*. Both installations include 64-bit and 32-bit native command-line tools that target x64, x86, and ARM.”*

**Instructions for installation**

Download the Microsoft Visual C++ Build tools from this link : <http://go.microsoft.com/fwlink/?LinkId=691126>

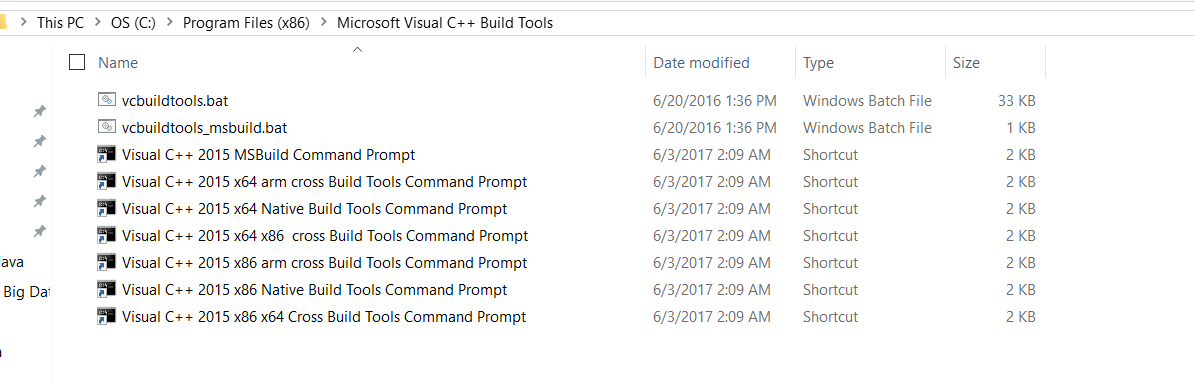


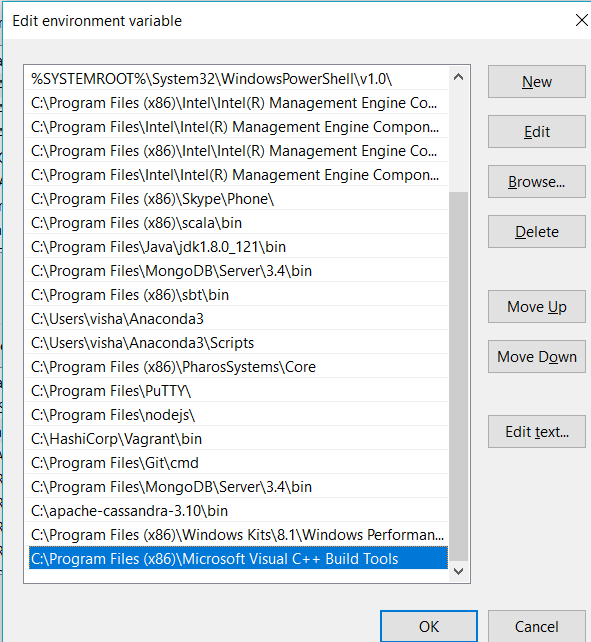
Click Install to install Visual Studio.



Installed Successfully

**Set the environment variable to the installed directory for where the Microsoft Visual Studio Build Tools gets installed in your system. Typically C:\Program Files (x86)\Microsoft Visual C++ Build Tools**





After installation, we can create a Makefile and execute the NMake command in the folder where this file exists and build the pipeline. Please refer to <https://msdn.microsoft.com/en-us/library/dd9y37ha(VS.71).aspx> for details on syntax differences between the GNUMake and the NMake Makefiles.

**LUIGI**

Luigi is a workflow engine that helps build pipelines of batch jobs. We can efficiently chain multiple independent / dependent tasks to create dependency graphs among tasks which can be then scheduled to run on the luigi scheduler. Luigi was created by Spotify to solve their data pipelining needs and then later open sourced. Multiple organizations have adopted Luigi as their workflow engine. Luigi has support for MySQL, Pig, RedShift, Spark, BigQuery and Snakebite is the Hadoop client by Spotify which they open sourced.

**1.Without Luigi -**

We would have to do everything in one function, but then this would defeat the goal of modularity. Our python functions would be cluttered and maintenance will become a real nightmare.

Cron Jobs

Batch jobs can be scheduled as cron jobs. But they have they are not fail safe. If one of the task fails, the next dependent task might fail and so on. One way to mitigate this is by increasing the time interval in between. But this might be a bad idea if one of the tasks is failing due to timeout errors.

**What Luigi is Not**

Luigi is not a replacement for Cron. Spotify claims that if tasks are to be scheduled to run periodically, the easiest way is to trigger a Python script from a cron or a continuously running process.

Luigi is not there to replace mrjob (Hadoop client of Yelp) or scalding (scalding). Infact luigi supports scalding.

**Refer** : <https://www.youtube.com/watch?v=ymF2R_tY1f8>

**2.Creating tasks in luigi**

For creating a new task in Luigi, we have to create a new class for the task and extend the abstract luigi.Task class. We can define the respective functions to be used to perform the activities of this task.

We have to implement/override a couple of methods from the luigi.task class

1. **Run** – this is where we put the bulk of all the logic for the task goes.
2. **Requires** – This function indicates all the tasks that need to be completed before this task should be scheduled to run prior to running this task.

Handles the dependency chain.

This returns a list of all the dependent tasks that need to run in order for this task to be scheduled for running. If the task is dependent on multiple other tasks, then we can use yield instead of return to yield a list of all dependent tasks.

1. **Output** – this method is invoked on completion and is used to return the target. Usually to write something to a file to indicate that this task has completed successfully or with errors. In luigi, a task is defined as complete if it has output.

There is a concept of targets. luigi.LocalTarget is one which can be the local file on the local file system. We can also provide a luigi.contrib.hdfs.target.HdfsTarget as the target. This is used for running Hadoop jobs as a part of the pipeline. (Refer for more details : ) Another example can be a Hive Target using the luigi.hive.hivePartitionTarget (luigi.HivePartitionTarget expects the data to be inputted as a dictionary.

1. **Complete** -- Tasks that do not have output() can override complete()

For example, if we don’t want to write to a file to indicate completion, maybe we send an email to signal completion, then in these cases, we can override complete() method. Luigi tasks look for this to check if the task has completed.

1. **Mapper and Reducer** – We can override these functions instead of overriding the run function if we want to return a HDFS target to perform a Map Reduce job on Hadoop.

**3.Execution Model**

The execution model in luigi is simple. The luigi tasks can run in worker threads with a central planner in place. During development, we would generally run the Luigi workflow using the command line, but in production, we could use another scheduler. The disadvantage of luigi is that it doesn’t provide native support for distributed computing. It only acts as the workflow engine. Distributed execution has to be taken care of externally. The luigi central scheduler does not help with parallelization.

**Execution Model** : <http://luigi.readthedocs.io/en/stable/execution_model.html>

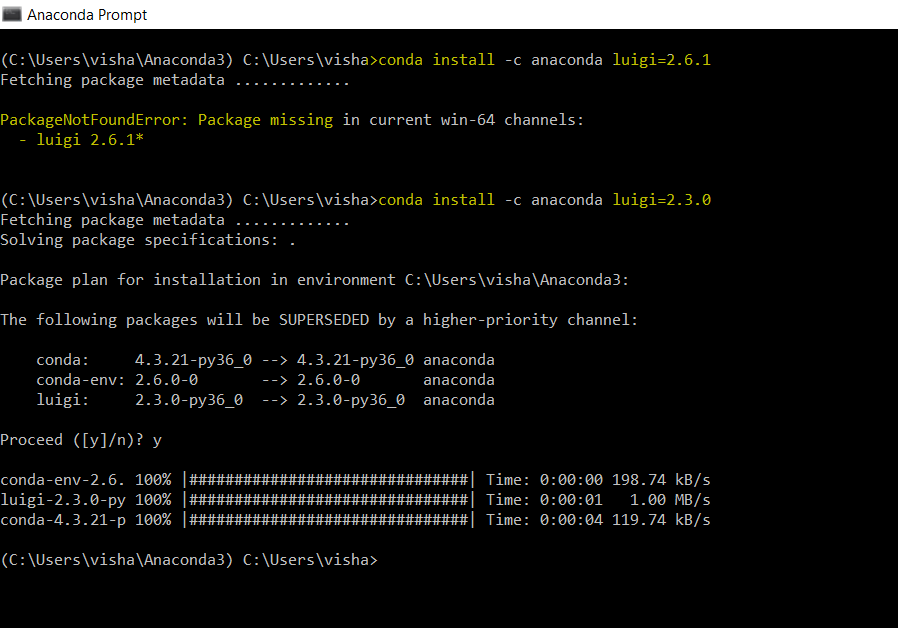
**4. Installation**

Install on Anaconda cloud =:

conda install -c anaconda luigi=2.3.0

Install using pip

pip install luigi



We can see in <https://docs.continuum.io/anaconda/pkg-docs> and <https://anaconda.org/anaconda/luigi> that the latest available version for Anaconda is 2.3.0. This should be fine for the coursework but if you want to explore latest features, then you will have to install it using pip.

**5.Creating LUIGI Tasks**

Luigi pipeline has to be constructed in a backwards manner. Meaning – the requires method should mention all the tasks that need to be completed in order for this task to run. That’s how Luigi schedules it’s tasks. We can add the logic for the task within the run method and return the path of the target file from the output function. Example of tasks are given below.

General steps to perform Task Execution

1. Set PYTHONPATH to the folder where all your task modules are present.
2. Create modules and Tasks within it.
3. Override the functions output/complete, requires, run
4. Execute luigid to start daemon
5. Execute Final Task in your dependency graph

luigi --module <modulename> <Task Classname>

1. visualize status on localhost:8082

According to <http://luigi.readthedocs.io/en/stable/command_line.html> the preferred way of executing luigi pipelined tasks is through the command line tool “luigi” that will be installed by pip or conda.

Luigi is python specific. For any changes to the tasks , the code has to be written in python. For executing R tasks in the pipeline, there is a project called Mortar-Luigi which is explained briefly later.

Luigi requires Python to run. Latest version of Luigi 2.6.1

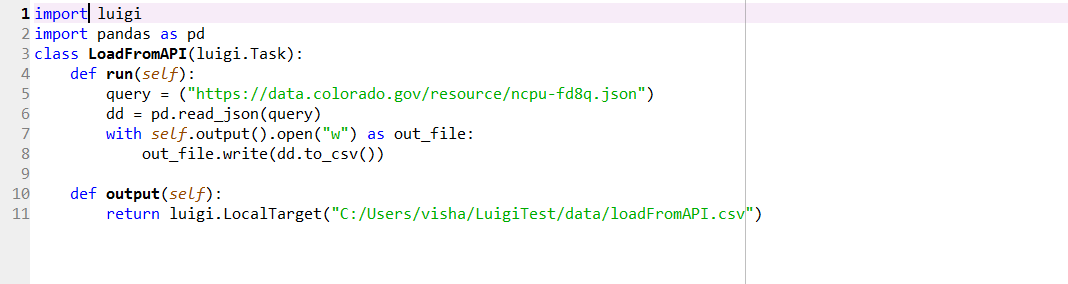
**6.Demo**

We will be creating a pipeline of tasks where we load data from 2 sources, perform some aggregation over them and store the output into a file. This basic structure and idea of the pipeline can be expanded further to build more complex data pipelines to handle large data processing activities. Before we go into the execution of the tasks, we will explain how these tasks are created.

**Task 1**

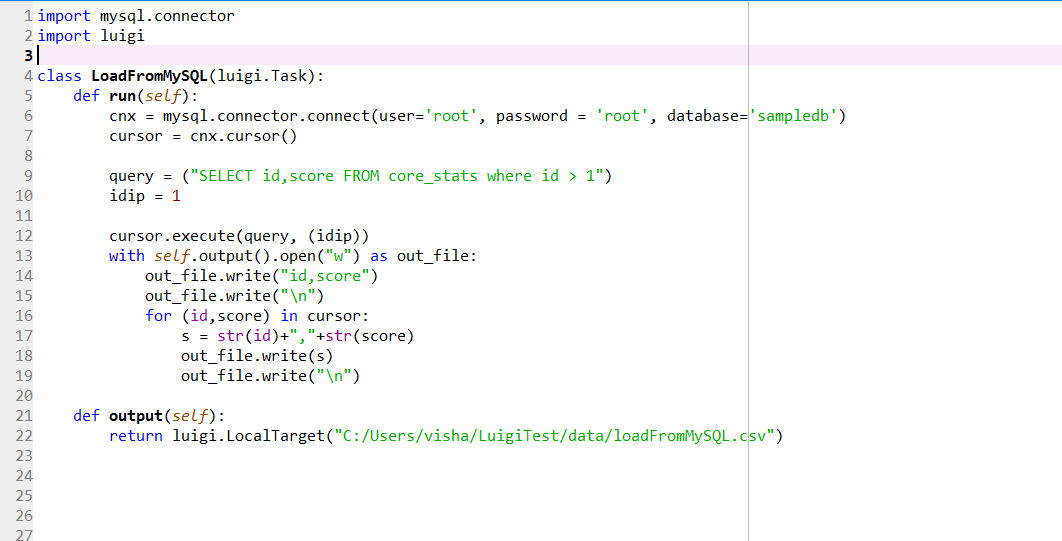
1. **LoadFromAPI task saved in the loadFromAPI module**

In this task, we have not overridden the requires method. This means that this task can run independently of other tasks. This task makes a http call to a public api, gets the data in json format and writes it to a file on the local system. This LocalTarget is returned as the target from the output method.



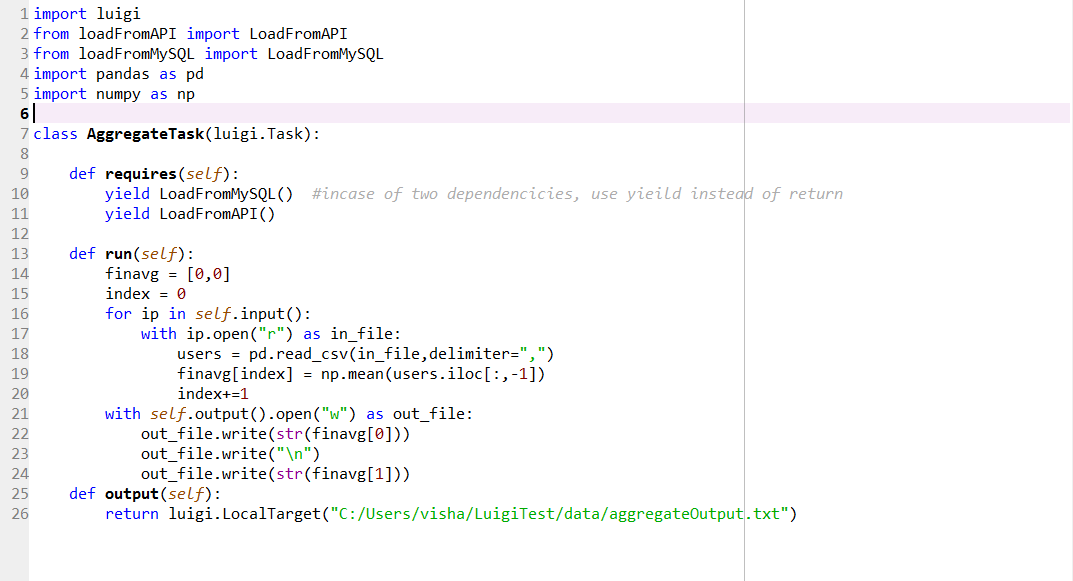
1. **LoadFromMySQL task saved in the loadFromMySQL module**

In this task, we have not overridden the requires method. This means that this task can run independently of other tasks. This task connects to a MySQL sample database, queries for the required data and writes it to a file on the local system. This LocalTarget is returned as the target from the output method.



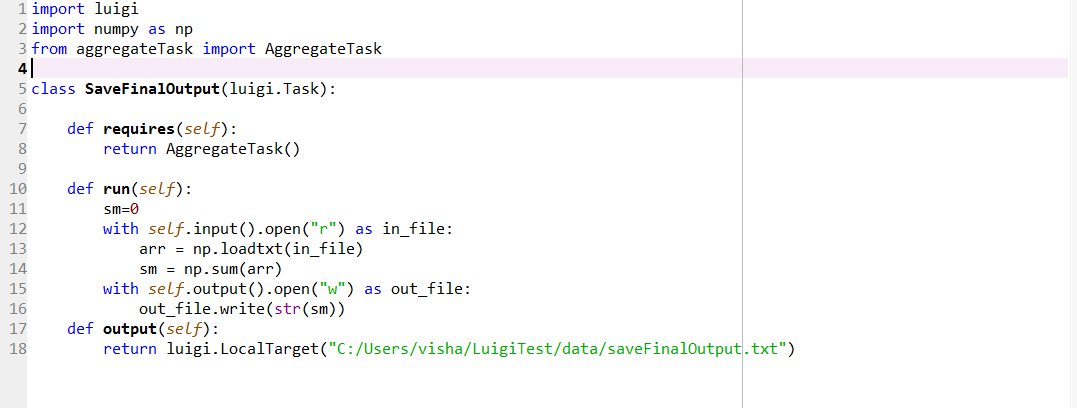
1. **AggregateTask saved in the aggregateTask module**

This task overrides the requires method which indicates to Luigi that the LoadFromMySQL and LoadFromAPI tasks have to run before this task can be scheduler to run. The run method basically just performs aggregation on both the files and outputs the result. The output returns a LocalTarget which will be used by the next task.



1. **SaveFinalOutput task saved as saveFinalOutput module**

This task requires the AggregateTask task to be run before being scheduled. The run method just adds the 2 averages returned by AggregateTask. This is written to another file which is returned as the target by the output method. Please note that we can also overwrite the complete function in this case if we don’t want to return any output to indicate completion.



**7.Running LUIGI Tasks**

There are 2 ways in which we can run luigi tasks. We should invoke the final task in our dependency graph because remember – Luigi builds its dependency graph of tasks backwards using “requires”

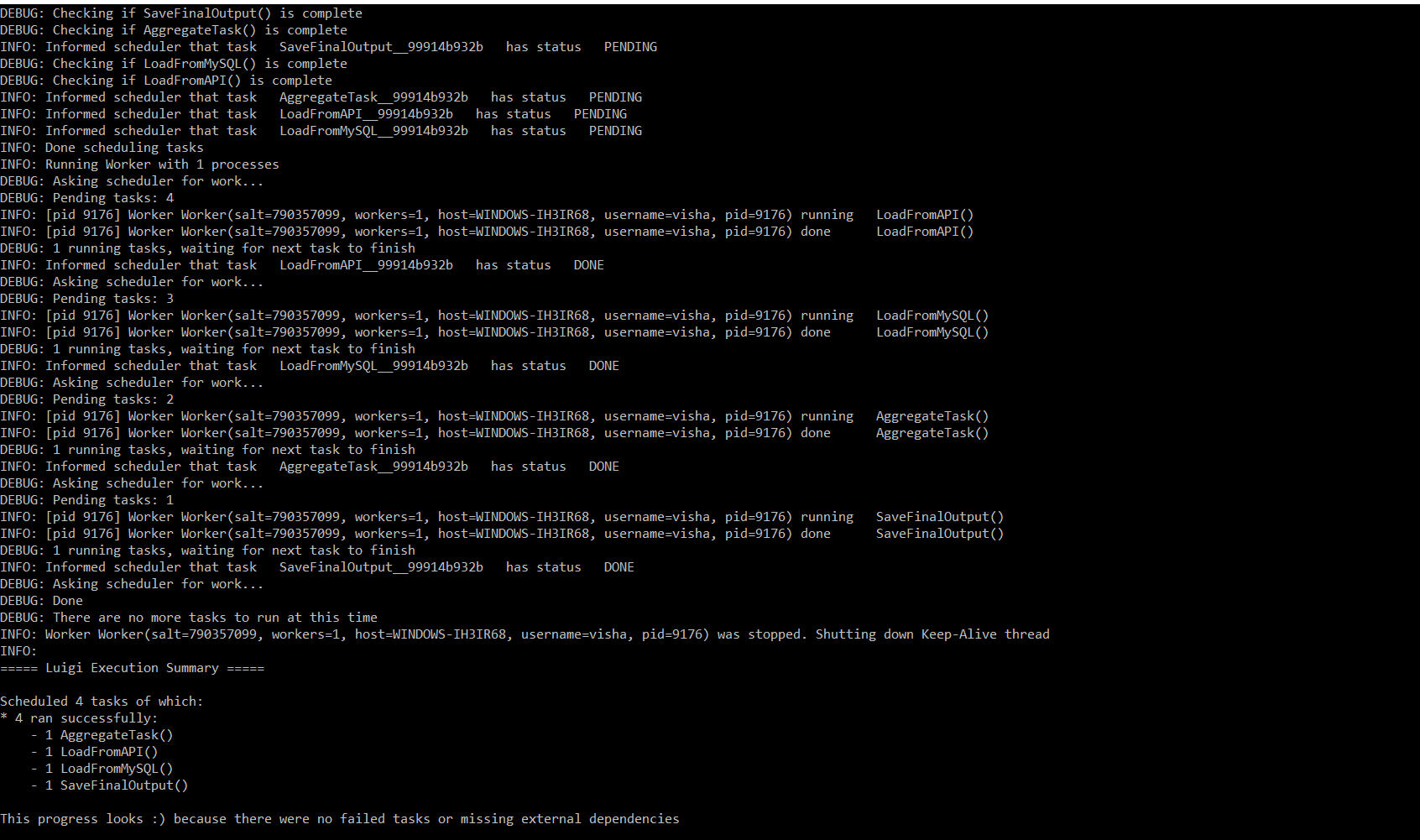
1. Using local scheduler.

This method is used for testing how the tasks will run on Luigi without using the scheduler. This is not recommended any purposes other than to just test the functionality of your code.

For running tasks locally, we can use the following command by specifying the –local-scheduler flag.

luigi --module <module name> <task class name> --local-scheduler



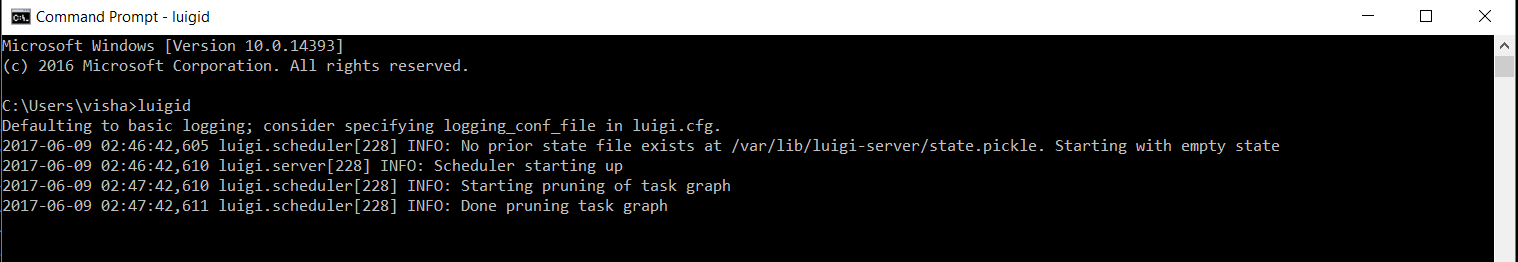




*The “:)” symbol at the end tells you that your job ran successfully.*

1. The other way to run luigi tasks is by invoking these tasks on the scheduler.

For this, we first have to start the luigi scheduler daemon thread by invoking luigid command.



Then, we start the luigi task by invoking the same command as we executed earlier for local scheduling but this time we remove the local-scheduler flag

luigi --module <modulename> <task class name>





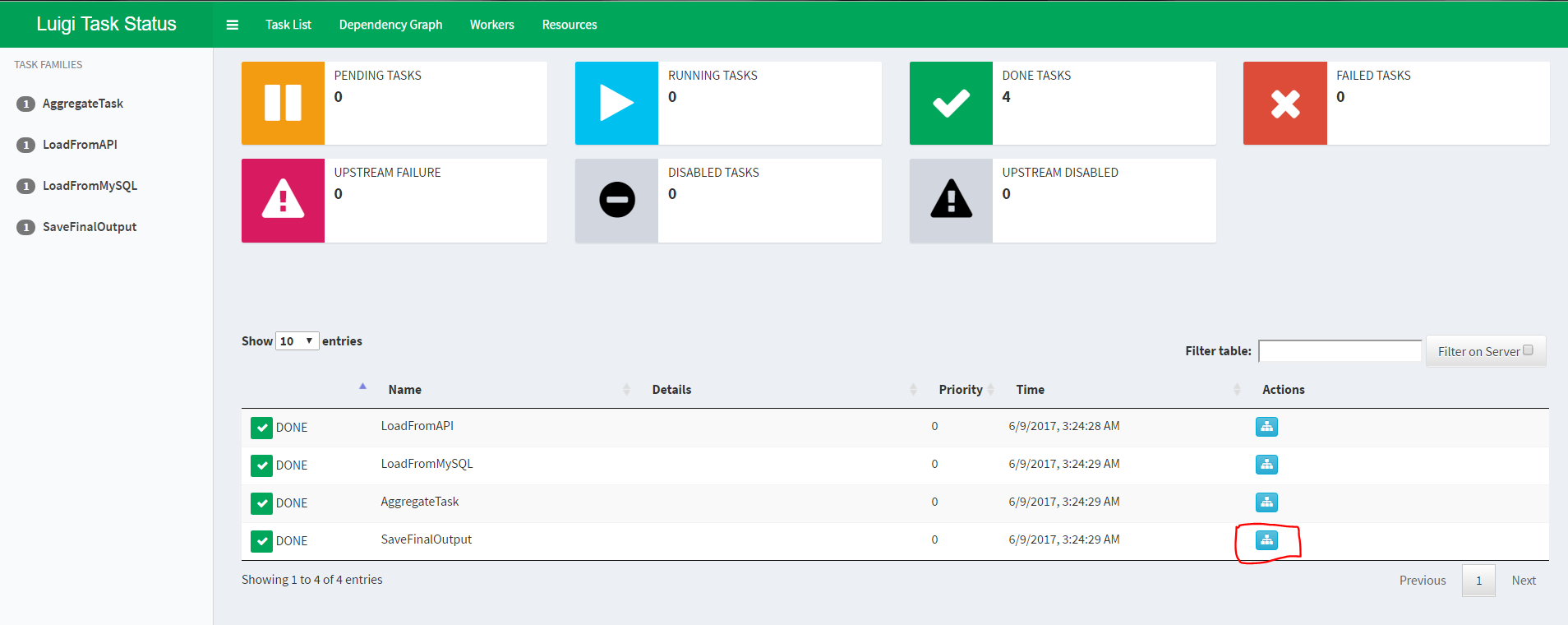
Now, the luigi task has executed on the central scheduler.

**8.Luigi Task Visualizer UI**

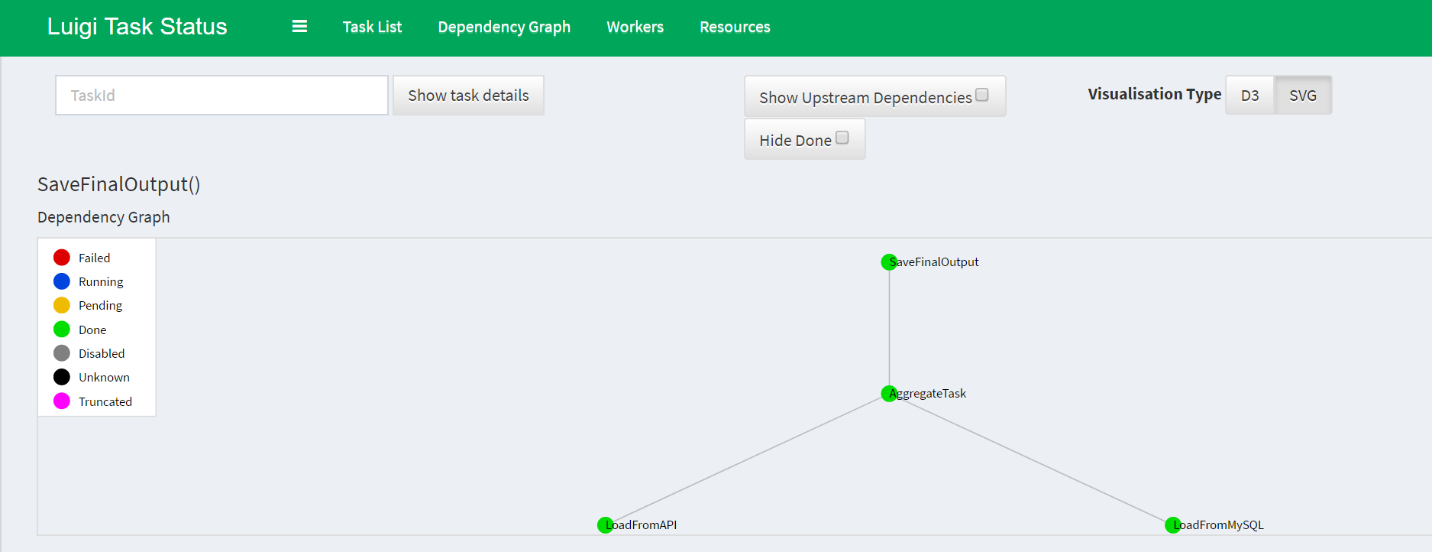
The neat thing about executing tasks on the scheduler is that you get a visual representation of the tasks. Luigi provides a Luigi Task Visualizer web UI which we can open using the default port 8082 on the host address of the scheduler daemon.

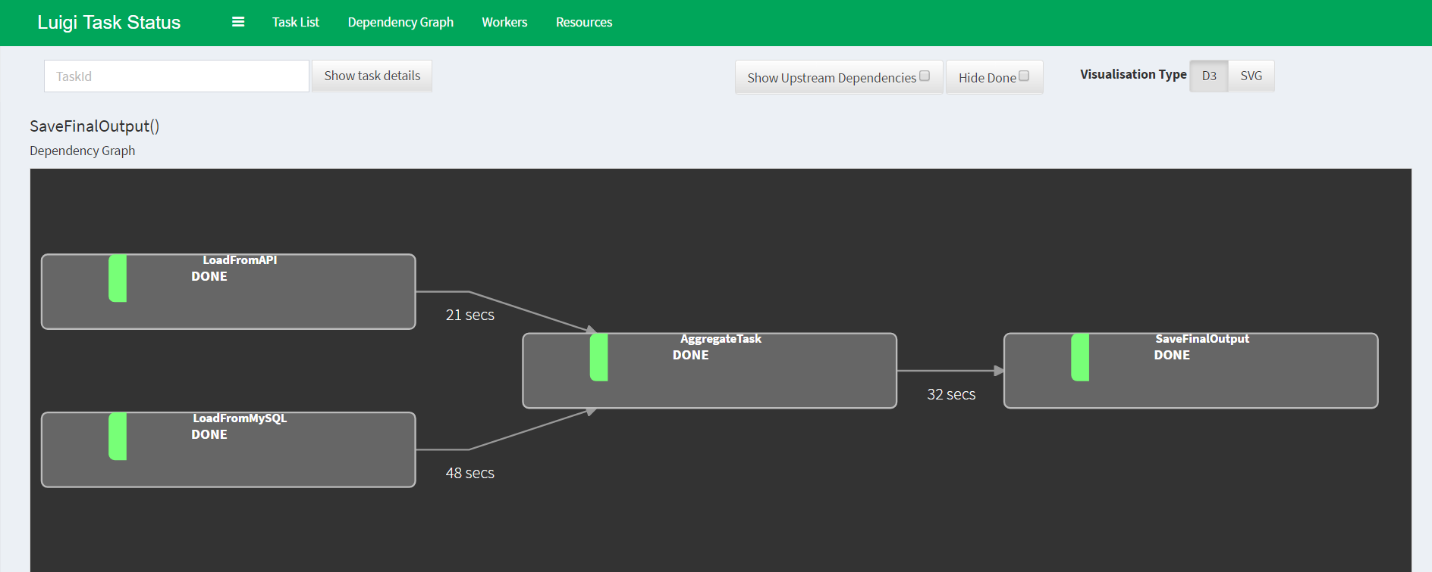
In our case, since we are running the scheduler on our local machine, we can enter

<http://localhost:8082> in the web browser to open the following UI.



As we can see in the above UI, the luigi scheduler has executed our 4 tasks. We can click on the tree icon highlighted under the actions column to view the dependency graph.





In the above visualizations, we can clearly see that our SaveFinalOutput was dependent on the AggregateTask which in turn was dependent on LoadFromMySQL and LoadFromAPI tasks. The scheduler will complete these two tasks before scheduling the Aggregate Task and then scheduler the SaveFinalOutput task to run.

**Explanation for the execution of the dependency graph which we just executed.**

1. Since we invoked SaveFinalOutput task to be run on the scheduler using command mentioned earlier, Luigi scheduler checks the requires function and finds that SaveFinalOutput task depends on the AggregateTask.
2. So, Luigi will try to schedule AggregateTask to be run on the scheduler but first, Luigi scheduler checks the requires function defined in the AggregateTask. Since this function yields 2 tasks with the yield keyword **(Note : For single dependencies, we should use return and not yield)**  Luigi now looks for these 2 tasks, LoadFromAPI and LoadFromMYSQL.
3. Since these 2 tasks don’t have a requires function (meaning that these two tasks are not dependent on the execution of any other tasks) , luigi executes the run method of LoadFromAPI and LoadFromMySQL.
4. Upon successful execution of these 2 tasks, luigi passes the target returned by the output function of these 2 tasks to AggregateTask.
5. AggregateTask can access this using the self.input() method.
6. Upon successful execution of Aggregate task, the output will be written to the final file and this local target is returned to the SaveFinalOutput task.
7. The SaveFinalOutput task saves the final output to the required path or to any other file system or database. After this, the execution ends.
8. In this way, we can easily extend this basic chain example to form more complex pipelines for our projects.

**9.Disadvantage of LUIGI :**

1. **Doesn’t Support distributed execution**
2. **Doesn’t provide a way to trigger flows.**

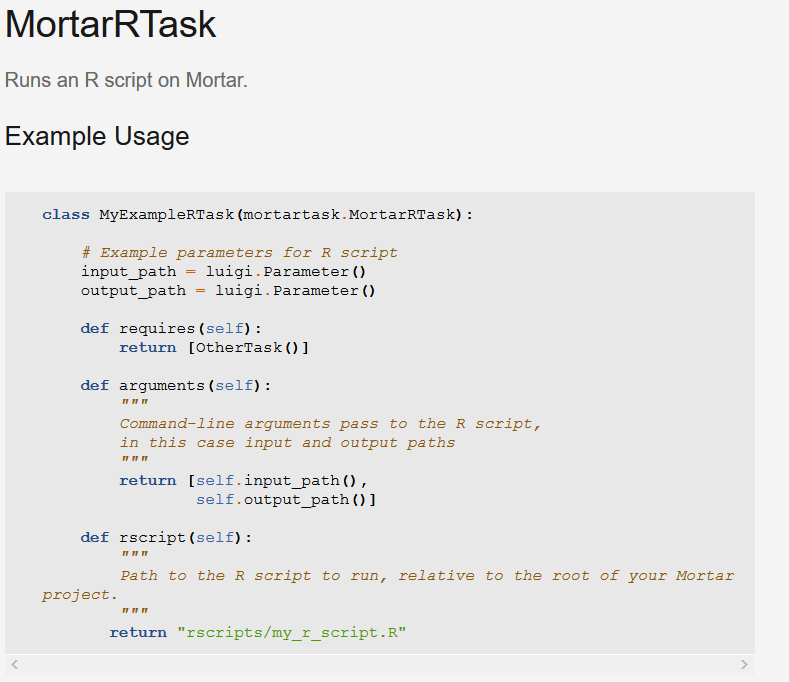
Luigi Website : <https://pypi.python.org/pypi/luigi>

Documentation : <http://luigi.readthedocs.io/en/stable/>

References : <https://marcobonzanini.com/2015/10/24/building-data-pipelines-with-python-and-luigi/>

**Building R tasks in Luigi**

We can easily build a pipeline for R tasks as well using a project called Mortar-Luigi. This project uses the Luigi framework to provide an extension to Luigi so that we can build workflow pipelines for R modules as shown in the below image taken from <http://help.mortardata.com/technologies/luigi/r_tasks>



Mortar provides 2 additional functions that can be overridden for R tasks.

1. arguments : This function specifies the arguments to be passed to R modules. These are similar to the arguments that we can pass to R when executing through the command line.
2. rscript : This function specifies the path relative to the Mortar project where Luigi can find the R script.

The other functions work in the same way as mentioned earlier. output and run, are covered by the general MortarRTask class, so you do not have to include them when writing your own R script Tasks.

Please refer to the below link for more details

<http://help.mortardata.com/technologies/luigi/r_tasks>

<https://github.com/mortardata/mortar-examples>

<http://help.mortardata.com/technologies/r/run_example_r_script#toc_1StarttheExampleProject>

**DASK**

DASK is a framework that is designed for flexible parallel computing. In DASK we can design complex workflows and letting the DASK Scheduler take care of executing them in an efficient multithreaded, parallel processes and/or distributed manner. DASK moves the practice from in memory to on disk.

The DASK scheduler is asynchronous and event driven and can handle multiple workloads from different workers concurrently. Dask-ec2 provides easy integration with a dask-ec2 cluster which can be provisioned easily to run computations. DASK uses native Python libraries.

*“Because dask.distributed is elastic, they can scale up or scale down their cluster resources in response to demand.”*

Official Link : <http://dask.pydata.org/en/latest/>

More Technical Details on Dask : <https://distributed.readthedocs.io/en/latest/>

Check out this youtube channel by Matthew Rocklin on DASK. He is one of the developers for DASK <https://www.youtube.com/channel/UCFYhuCL11p3oO9375_NbLQg>

**Features and modes of execution**

1. It provides familiar data structures (collections) such as bag, array ,dataframe. These collections are analogous to the python lists, numpy array and pandas dataframe respectively. The DASK collections can be leveraged by the user to run computations on large files and can also be used to perform computations on multiple files in parallel which will be treated as one file.
2. The second thing that DASK provides is parallel computation. DASK is optimized for parallel processing and uses the function **compute** and **delayed** to run a complex set of computations in parallel. Delayed is used to specify execution of pre-defined functions in parallel in a distributed manner. Compute is used to perform the actual computation on the specified delayed set of functions.
3. DASK also provides native support for scheduler and workers which can be spawned. A DASK client can be created which can connect to the scheduler and request for our task graph to be executed. The scheduler decides then delegates these tasks further to the workers which perform the tasks in parallel.

This setup can be done on the local machine where the workers will be individual threads or this can also be done in a distributed manner on AWS or on some other network of nodes.

We can use Feather as a bridge between Python and R dataframes <https://news.ycombinator.com/item?id=11384577>

**Referenes :**

<https://github.com/dask/dask-ec2>

<https://distributed.readthedocs.io/en/latest/ec2.html>

**Run in distributed mode on a single laptop**

<https://www.youtube.com/watch?v=uQro_CaP9Fo>

References:

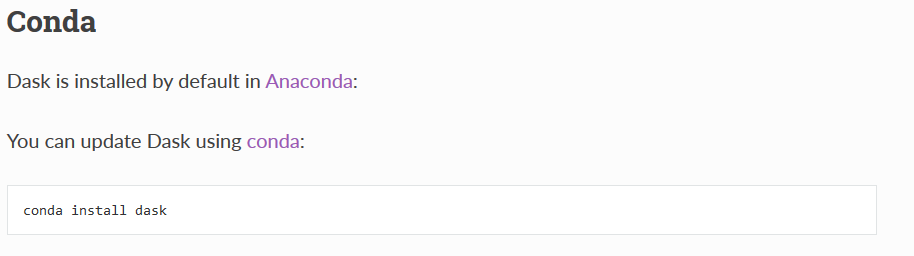
<https://github.com/mrocklin/dask-workshop>

**1.Installation**

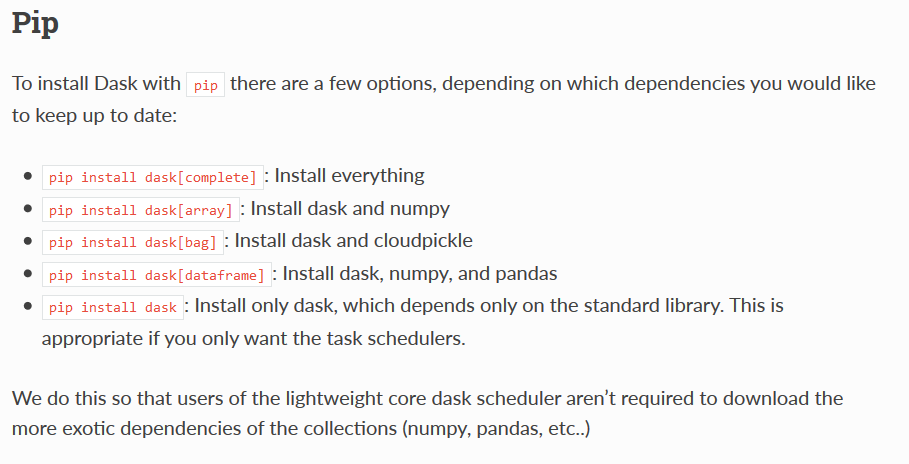
We can install Dask in different ways as illustrated below . Please refer to the official documentation for detailed instructions.

<http://dask.pydata.org/en/latest/install.html>

1. **conda** – If you use Anaconda bundle for Python



1. **pip**



1. To install from Source



pip3 install dask[complete]

**Installed in Windows using Anaconda**

conda install dask

conda install dask distributed

**For displaying graphs that we can schedule using visualize function of dask**

conda install graphviz

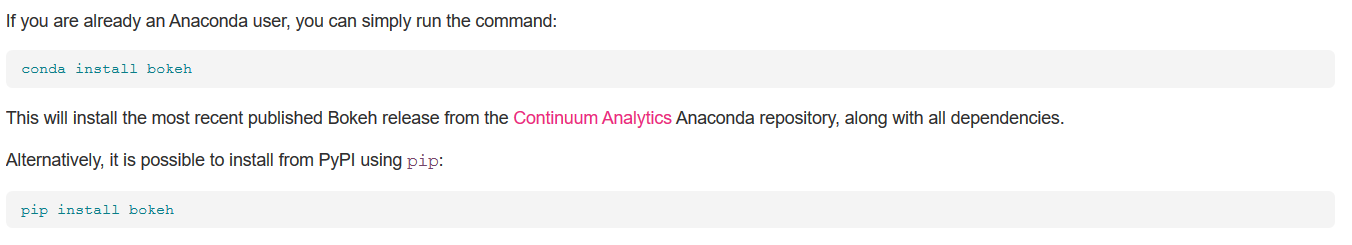
**2.DASK Web framework**

**BOKEH**

Bokeh is a WEB interface provided by DASK and is included along with the dask.distributed framework and helps data scientists track performance, check status, debug failures on their data science pipelines. Bokeh is mainly used to provide a real time analytics tool for the Distributed DASK environment that has to be setup. The following link is the official website for bokeh.

The Bokeh UI is usually available at the port 8787 of the scheduler’s IP address.

<http://bokeh.pydata.org/en/latest/docs/installation.html>



We can use the following command to install BOKEH libraries

**PIP**

pip3 install bokeh

pip install bokeh

**Anaconda**

conda install bokeh

For running Bokeh Web UI, we need to run DASK in a distributed environment. (This has been explained below). The Scheduler and the worker threads must be started. We should check the port number which the scheduler has exposed to display the UI. The demo for this is shown below.

**Official Reference:**

[http://distributed.readthedocs.io/en/latest/web.html#](http://distributed.readthedocs.io/en/latest/web.html)

**DASK can also be run using Python’s IPyParallel**

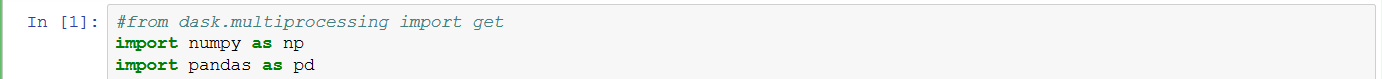
Please refer to this document to understand how to scheduler tasks using IPyParallel

<http://dask.pydata.org/en/doc-test-build/distributed.html#ipython-parallel>

**4.DASK Scheduler**

**1. Get METHOD**

The entry point for all schedulers is the get function. We can create a graph of our tasks and schedule it to run in a distributed manner using the get function. We have created one task graph which we will use to demo for different types of schedulers. This task uses 2 public API to load data into a pandas dataframe and then take the mean of one of the columns and add the mean.

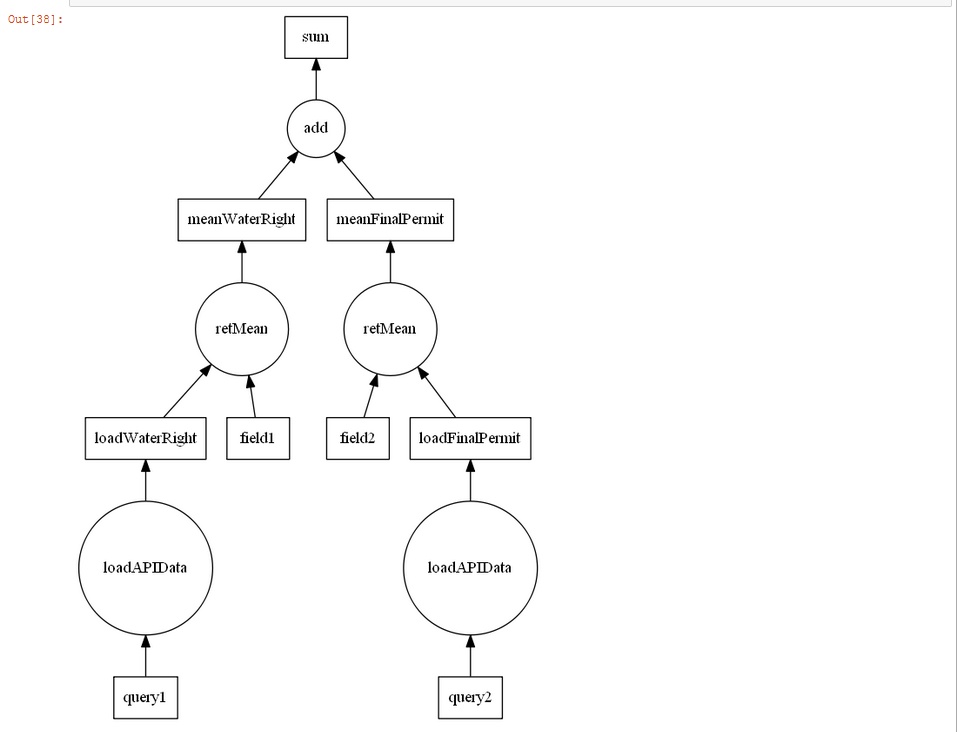






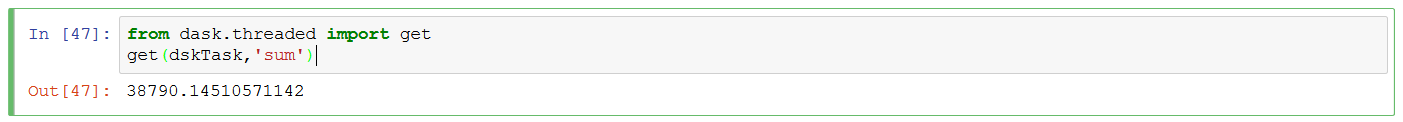
We can visualize the graphs that we generate using the following command. This requires that you have the **graphviz** library installed using conda or pip





**Scheduling the task using the DASK threaded approach**

If a task is run using the get method from the dask.threaded package, then the task that we have created is scheduled to run on Multiple lightweight threads.



**Scheduling the task using the DASK Multiprocessing approach**

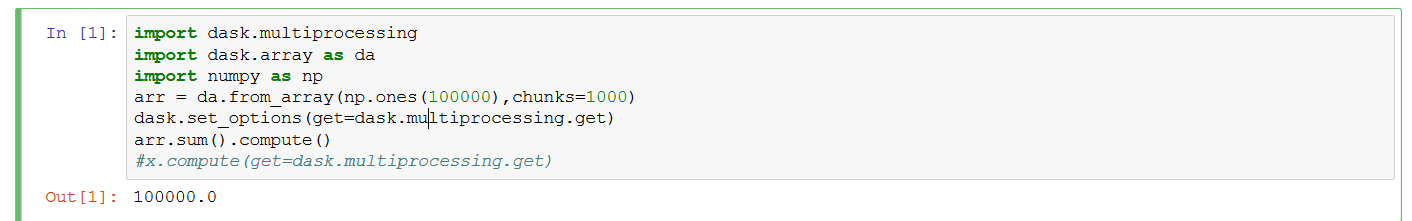
If a task is run using the get method from the dask.multiprocessing package, then the task that we have created is scheduled to run on Multiple Processes.

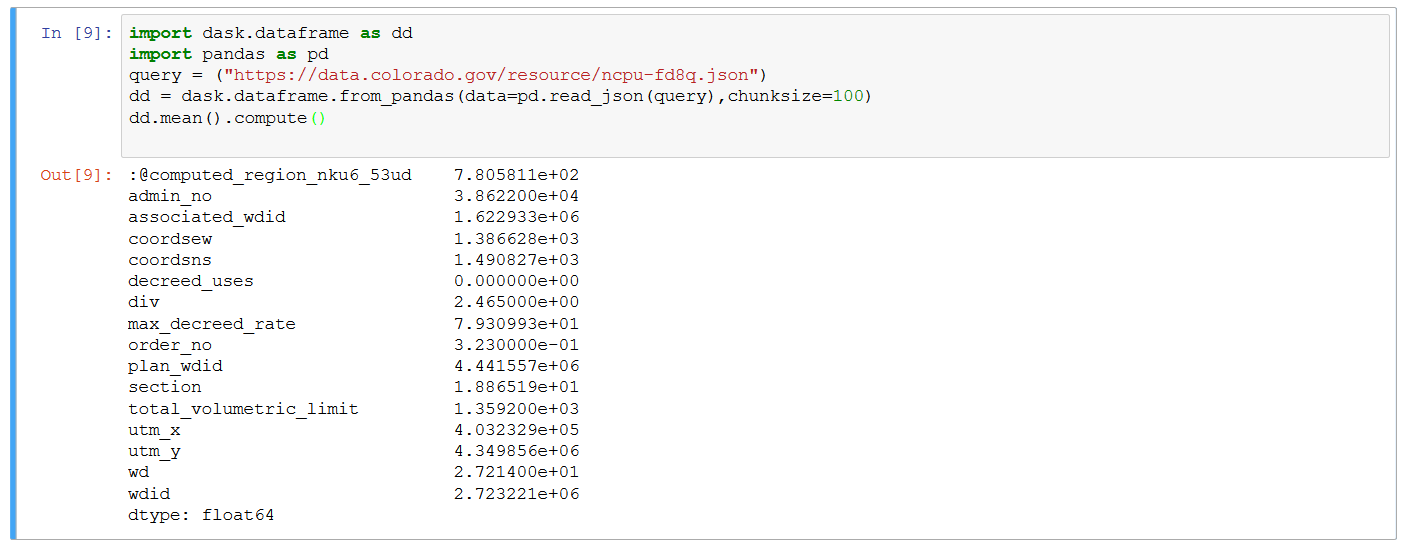


**2.Compute METHOD**

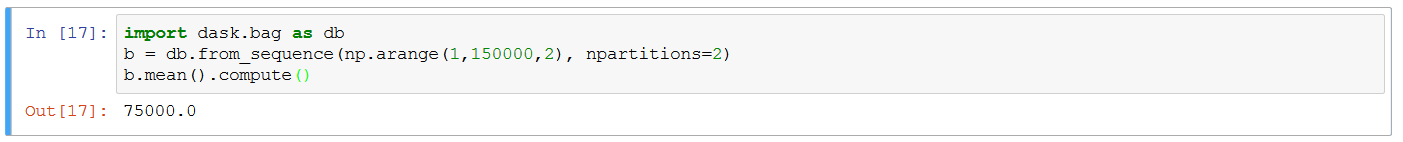
Most users call compute operation on DASK collections which run in a distributed manner. The compute method wraps the get call for DASK collections. Arrays and Dataframes use the threaded scheduler by default. Bags use multiple processes for performing computations by default.

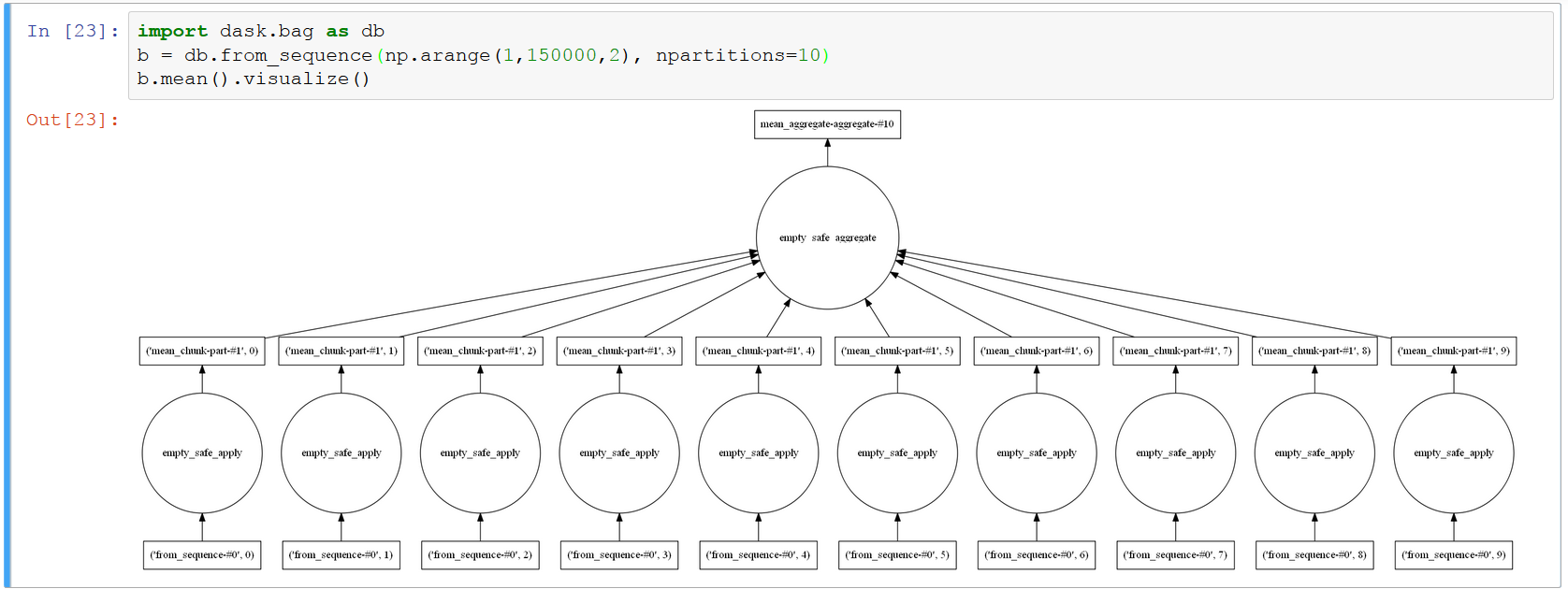
We can also change the default behavior by setting options using dask.set\_options(get=dask.multiprocessing.get) as shown below.





Visualizing Bag. We can see that the npartitions sets the number of partitions for the data which can be worked upon by individual processes. (Get method configurable as mentioned above)

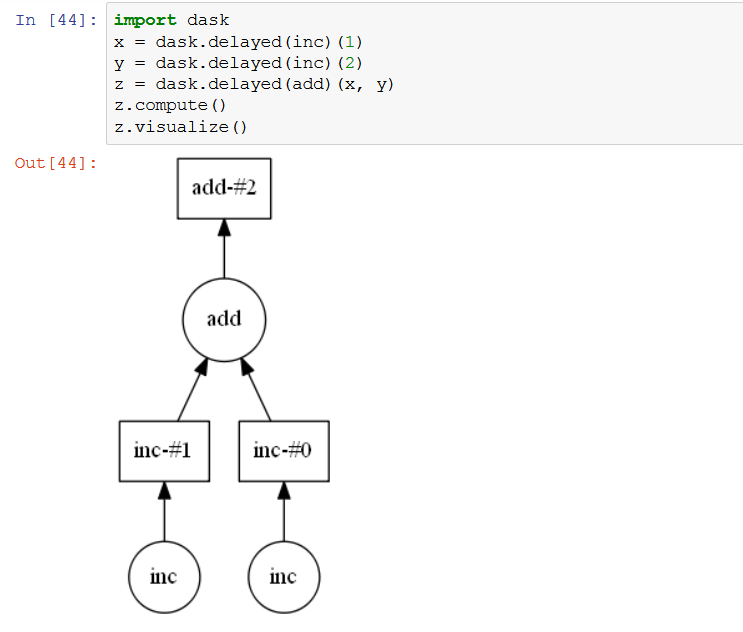




For additional information on arrays and bags, please see the official documentation.

**5.DASK Delayed**

Dask offers a delayed interface which can be used to schedule the tasks to run on the distributed cluster but not execute them immediately. We can use delayed to build the dependency graph as shown below.

In this example, we are scheduling x and y as independent tasks and these are being provided as inputs to the add function. We can use the compute() function to execute the tasks in the pipeline.

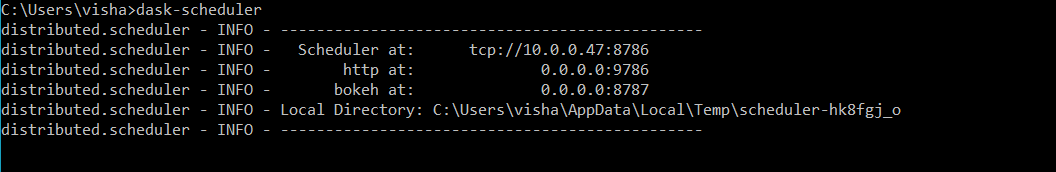
**6.Scheduler**

We can create tasks using the DASK framework There are multiple ways to create the Scheduler and the Worker.

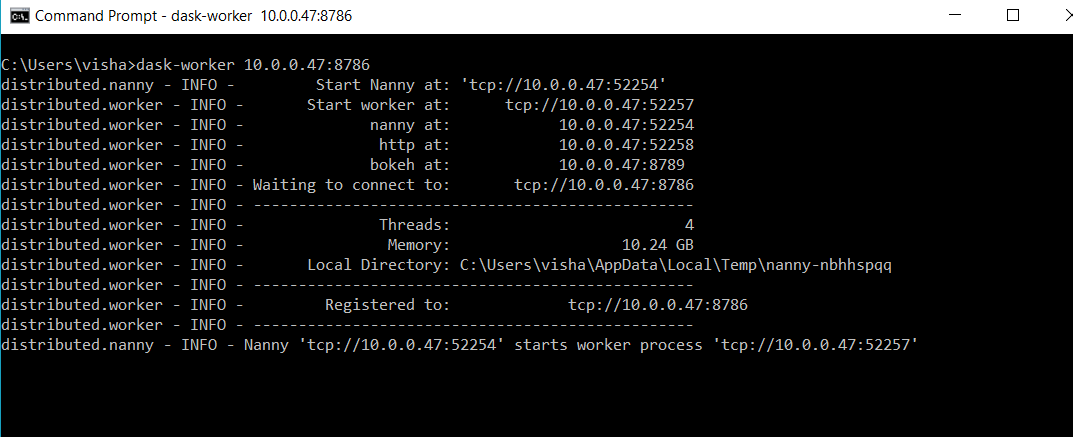
Here we will demonstrate how to create a distributed environment on a single machine. But we can easily scale this to a distributed network of computers.

We can create and run Scheduler daemon to run on one system as follows:

1. Open one terminal and execute the command dask-scheduler. This command starts up a dask scheduler at the ip address of the system. The dask-scheduler usually runs on 8786 port number. We can also change the port numbers if required.



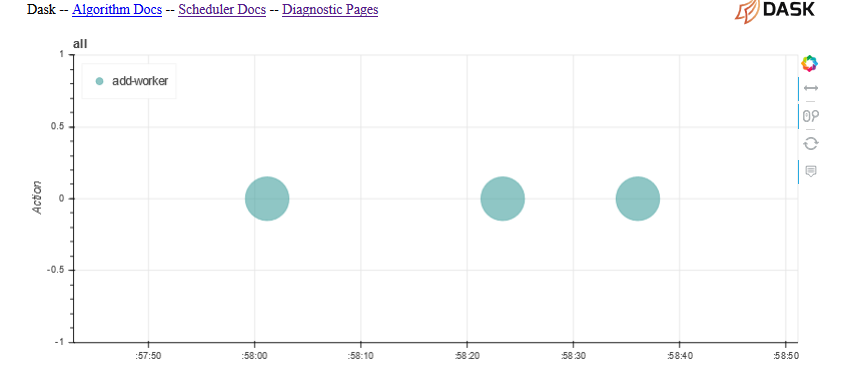
1. Open another 2-3 terminals and execute the command dask-worker <scheduler ip-address : port number>.



When we register workers with the schedulers, these events can be checked on the bokeh Web UI under events as follows:

We can see this by invoking http://<scheduler-ip address>:<port no>/events

In our example, this is available at http://10.0.0.47:8787/events



1. We have the distributed environment ready on our local system. Now, we can create a client which can connect to this scheduler and call the get function to execute the task graph that we created earlier (dskTask created above)

Execute the python code as shown below. The Client class can be used for this.

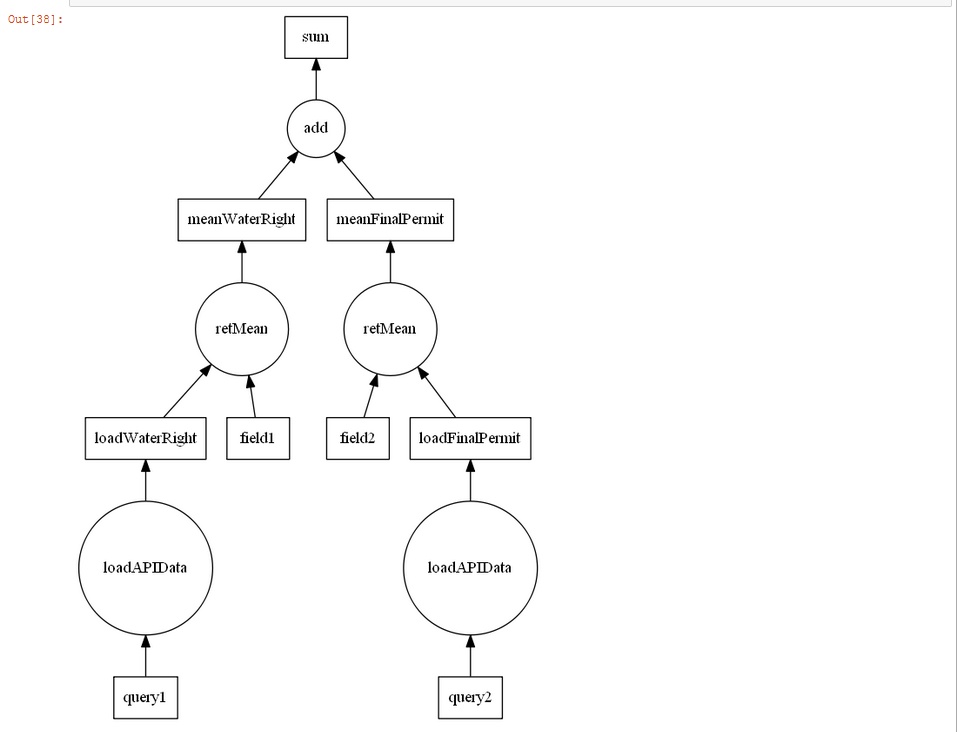
from dask.ditributed import Client

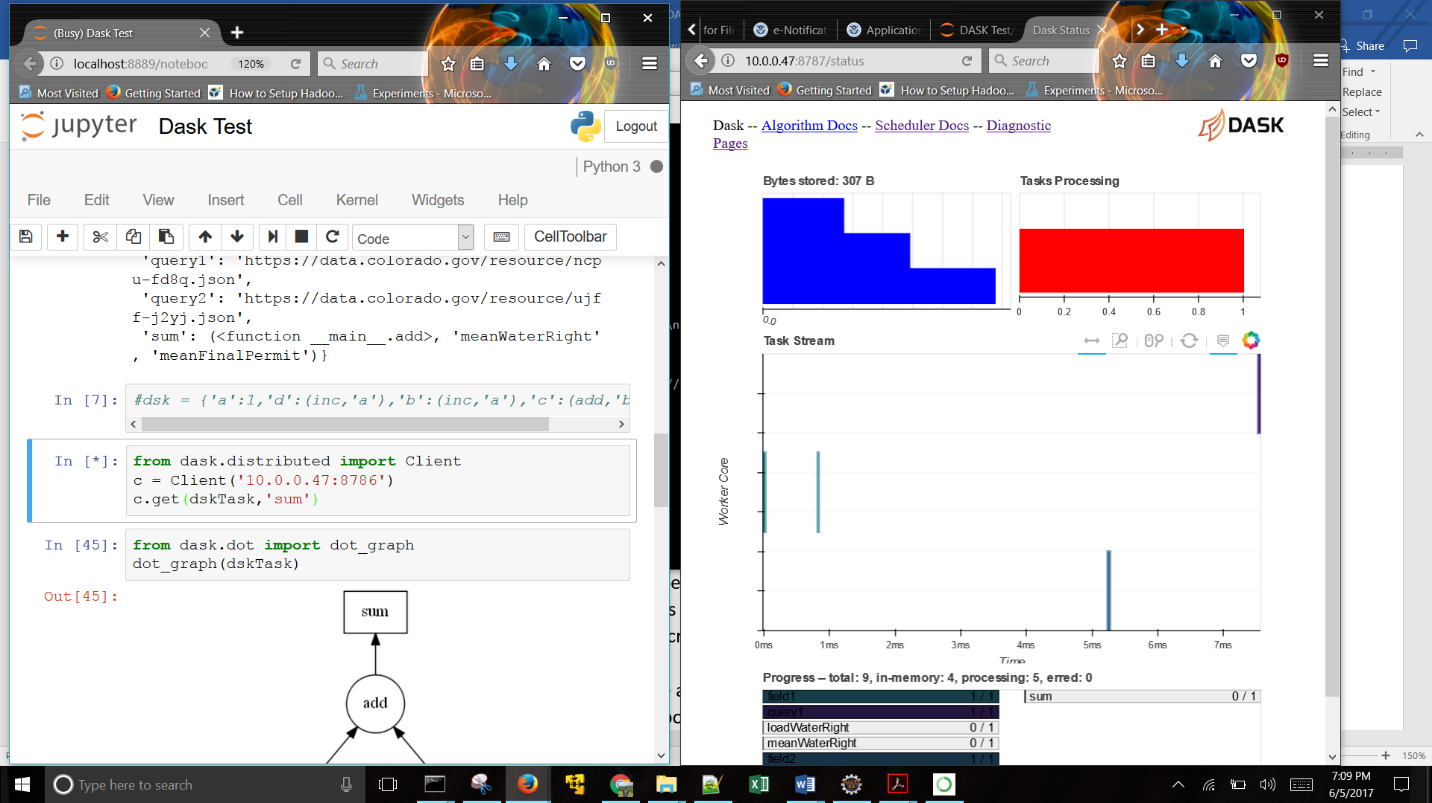
c=Client(<scheduler ip-address : port>)

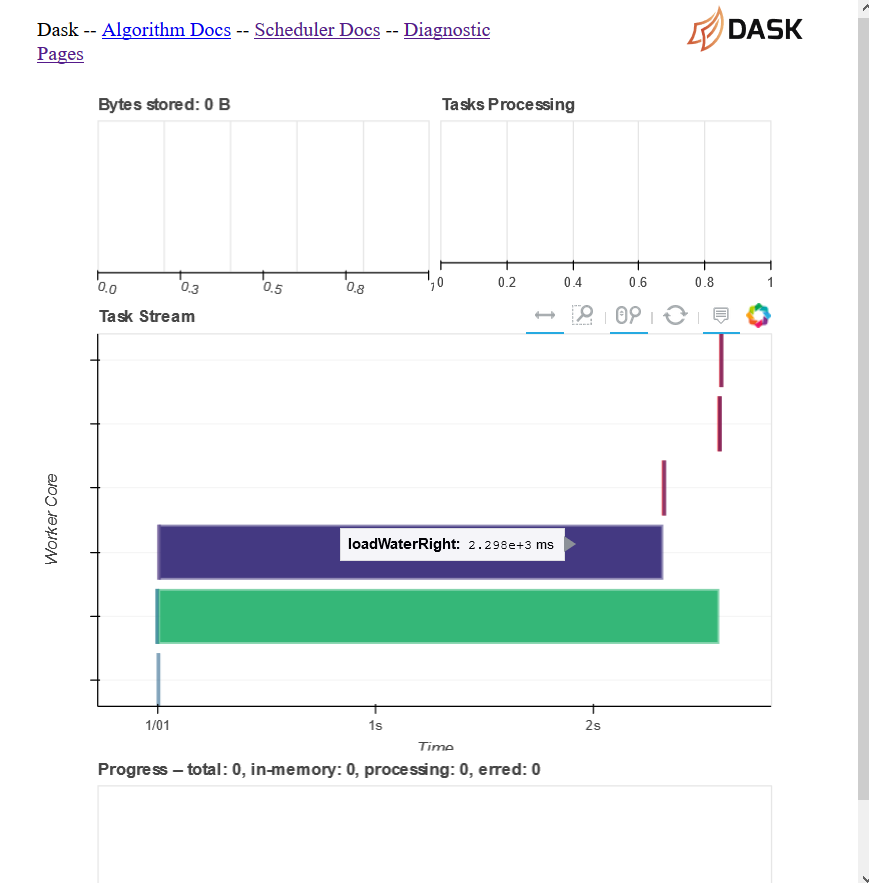
c.get(<taskGraph dictionary>,<task>)

We are executing the task graph that we created earlier.









The above UI is available at http://<scheduler ip-address>:<port>/status

In our example, it was available at http://10.0.0.47:8787:/status

We can see that our pipeline has been executed in a step by step manner and multiple workers running the loading activities in parallel.

**7.DASK-EC2 Framework -AWS EC2 Cluster**

Dask provides a built in framework called the Dask-EC2 framework also called as EC2 Start Script on the official documentation website indicated below that lets you easily create a cluster on Amazon Web Services AWS.

In the below example, we will attempt to create the EC2 cluster on AWS and run our task graph on AWS.

The following steps need to be followed to create and start your EC2 cluster.

1. If running on Windows, make sure you have SSH in your terminal. We are using the Cygwin terminal
2. Install dec2 library using pip install dec2
3. Create and fire up a new EC2 cluster with the following command



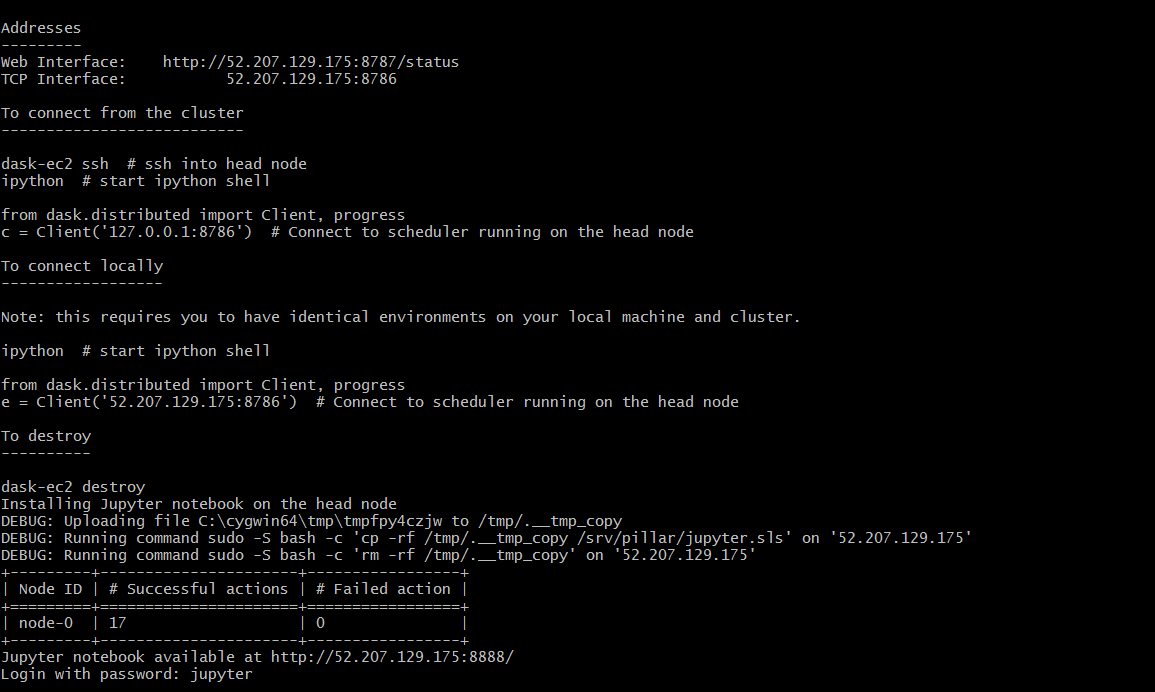
Count = number of instances, nprocs = number of processs that will execute.

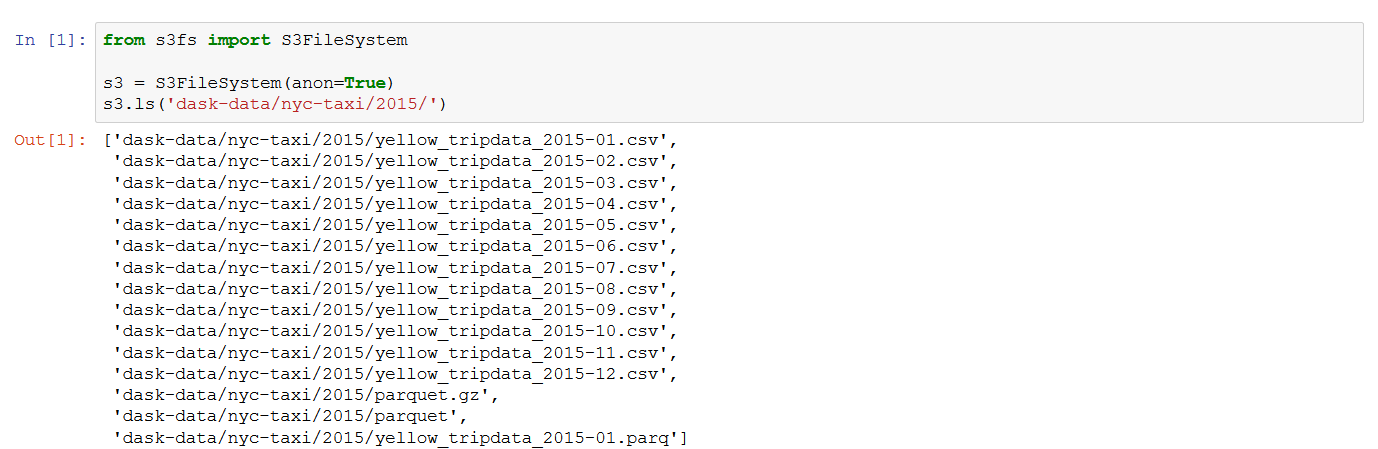
Keyname and keypair are your AWS credentials. (**Note** : since this command connects to the us-east-1 region, please make sure you create and specify the keypair from this region)

1. Using the connection instructions, connect to this server and fire up the jupyter notebook and also connect to the bokeh web ui using your browser.

Create and execute your task graphs similar to the way you created the task graph on the local cluster as described before.

Connection Instructions for AWS EC2 cluster

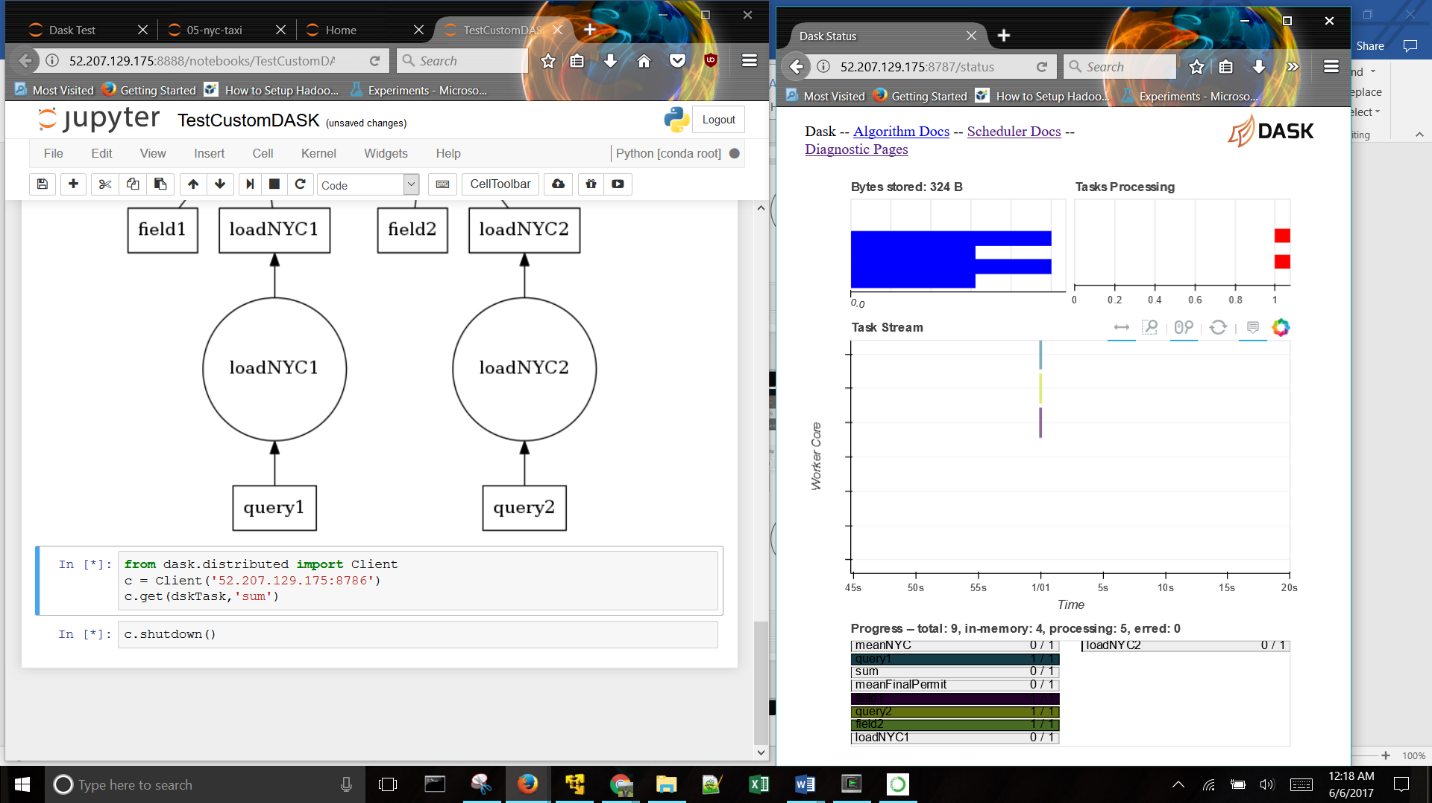


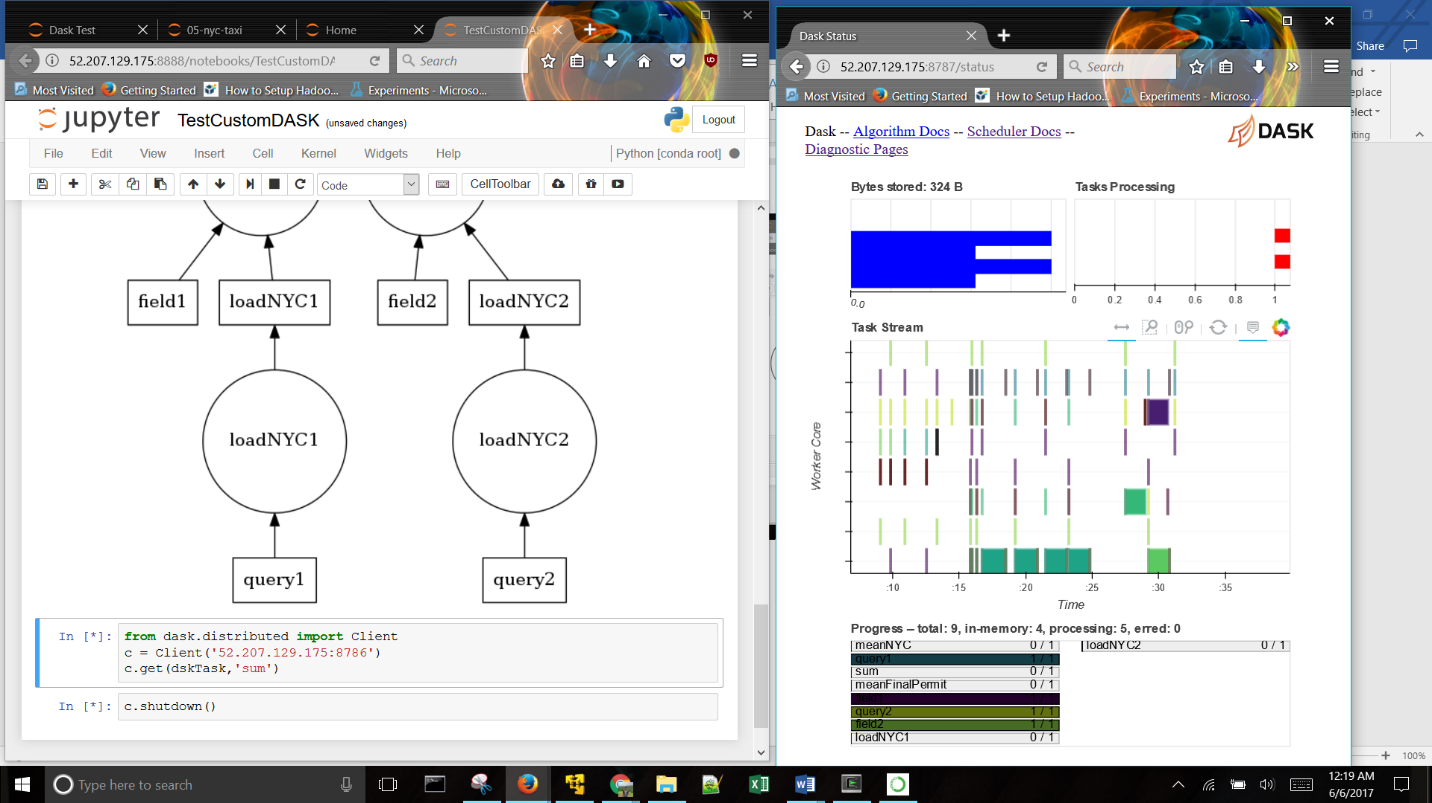


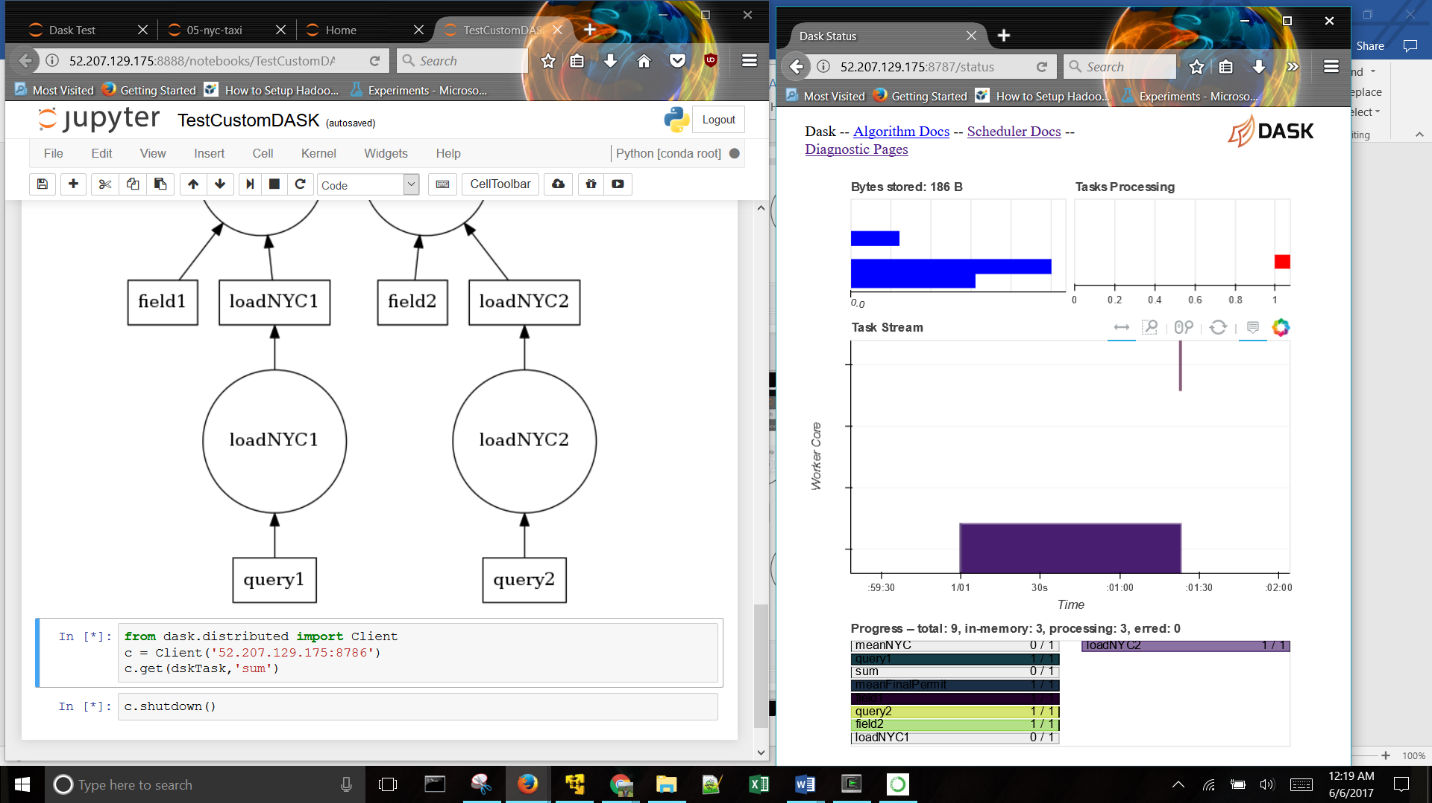


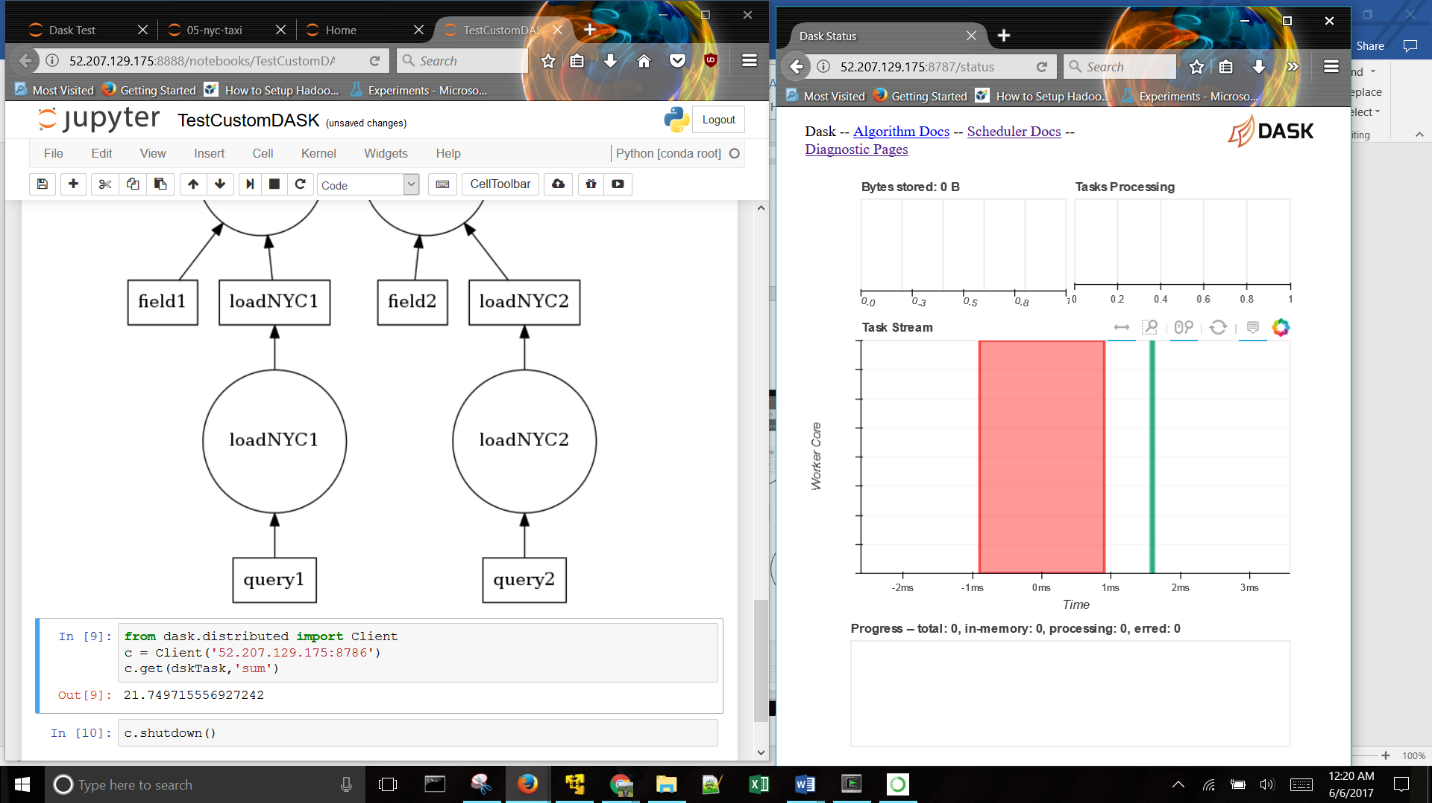












These are some good reference links for understanding more about DASK

<https://www.youtube.com/watch?v=KGlhU9kSfVk>

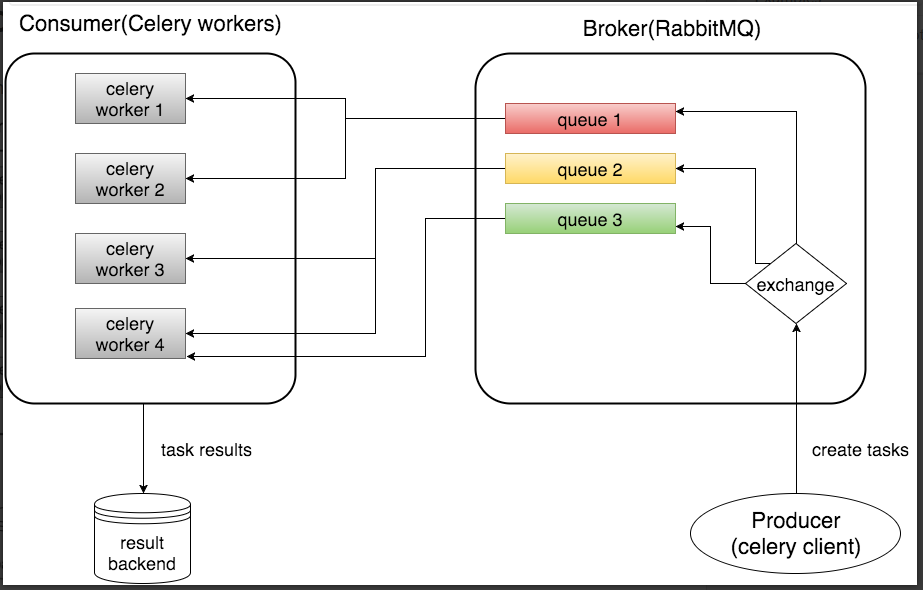
**Celery**

Celery is an asynchronous task queue/job queue based on distributed message passing. It works by posting the execution units, called tasks, to any AMQP – compliant message queue. The tasks added in the queue are then picked up by the available listening celery workers. The queue and the workers are not dedicated and can be hosted locally or on an external host or multiple host. If no error occurs the workers process and execute the tasks and return the results via celery client and back into the application. If a worker goes down in the middle of task processing, the task-message will eventually go unacknowledged, and another worker will pick up and execute the task

**Roles in task queues**:

**Producer**: produces tasks to execute   
**Broker**: dispatches tasks to the task queue, creates the task queue itself, delivers tasks from task queue to consumer   
**Consumer**: consists of workers that execute the tasks

Celery can be used to create fast web responses. For eg: on a new user sing up, the server should send sing up confirmation immediately while the process of sending an email should be assigned to celery. Celery would be running in background, outside of request-response cycle and it can send the actual email. It can also be used to make scripts faster with better cpu utilization. Celery can be used to break large tasks into smaller ones and compute them parallelly thus saving time.

**1.Celery architecture and concepts:**

Producer:

Producer is the user application that creates and sends the message/task.

Exchange:

Exchange is used to send messages to the queue. Exchanges take a message from producer and route it into zero or more queues. The routing algorithm used depends on the exchange type and binding rules.

Broker:

The Broker (RabbitMQ) is responsible for the creation of task queues, dispatching tasks to task queues according to some routing rules, and then delivering tasks from task queues to workers. Besides RabbitMQ other available broker options are Redis, Amazon SQS, or a database such as Django. Celery documentation provides information on the available brokers. ([Celery\_brokers](http://docs.celeryproject.org/en/latest/getting-started/brokers/))

Consumer (Celery Workers):

The Consumer is one or multiple Celery workers executing the tasks. You could start many workers depending on your use case.

Queue: Buffer to store messages.

Routing Keys :Bindings may have an optional routing key attribute. An exchange may use this field to route a message to the bound queue.

**2.Installing Celery and creating first task:**

Please refer to this video for more information on installation and basics: <https://www.youtube.com/watch?v=fg-JfZBetpM&t=307s>

1.Select a broker (RabbitMQ). Visit <https://www.rabbitmq.com/#getstarted> for RabbitMQ documentation.

$ sudo apt-get install rabbitmq-server

Once the command completes, the server is running in the background.

2. Install Celery

$ pip install celery

3. Install Redis as backend

$ sudo apt-get install redis-server

4. Celery application file: tasks.py

from celery import celery

app = Celery('tasks', broker='pyamqp://guest@localhost//' ,backend= 'redis://’)

@app.task

def multiply(x, y):

return x \* y

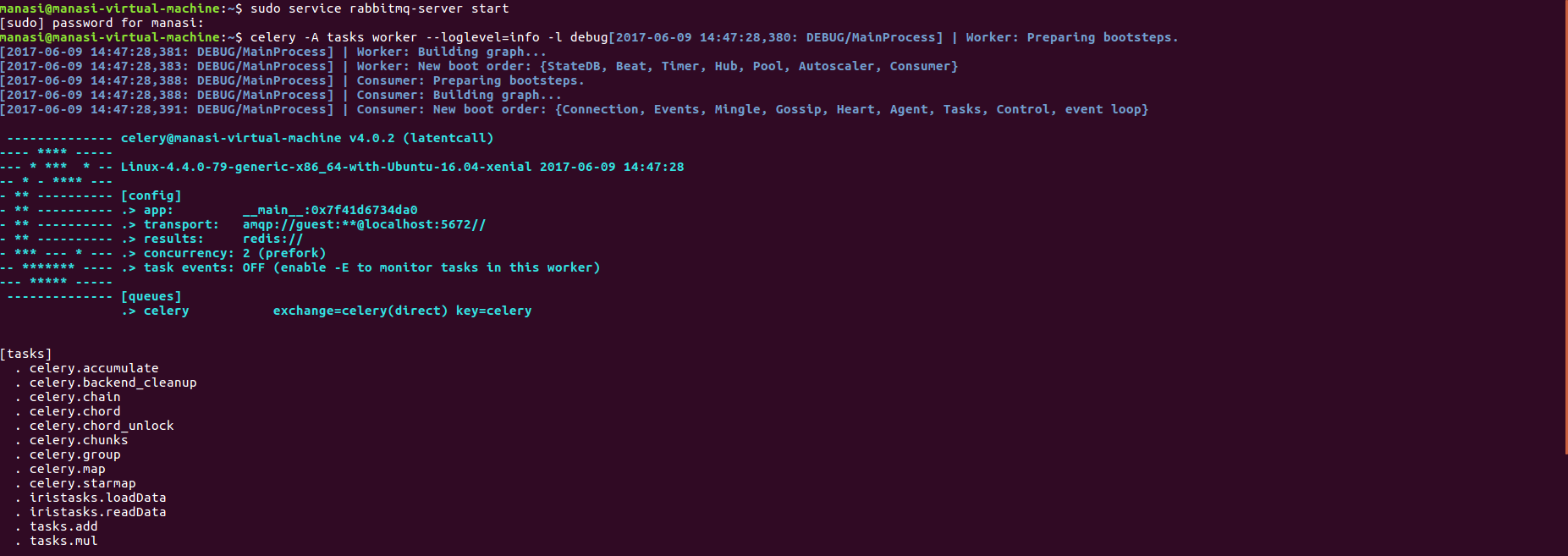
A single task is defined called multiply which returns a multiplication of two numbers.The first argument ‘tasks’ is the name of the current module. Second argument is the broker keyword argument, specifying the broker url, in this case rabbitmq.Third is the backend storage for the results.

5.Start/restart the rabbitmq server

$ sudo service rabbitmq-server start

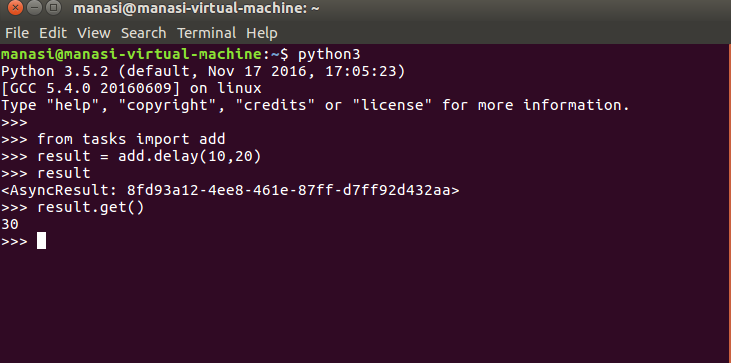
6 .Run the celery worker daemon in debug mode

$ celery -A tasks worker --loglevel=info -l debug



7. Calling the task:

Open a python shell and using the delay() call the task as shown:



Delay() is a shorthand of apply\_async() which generates async calls and results, which can be viewed on the workers output console.

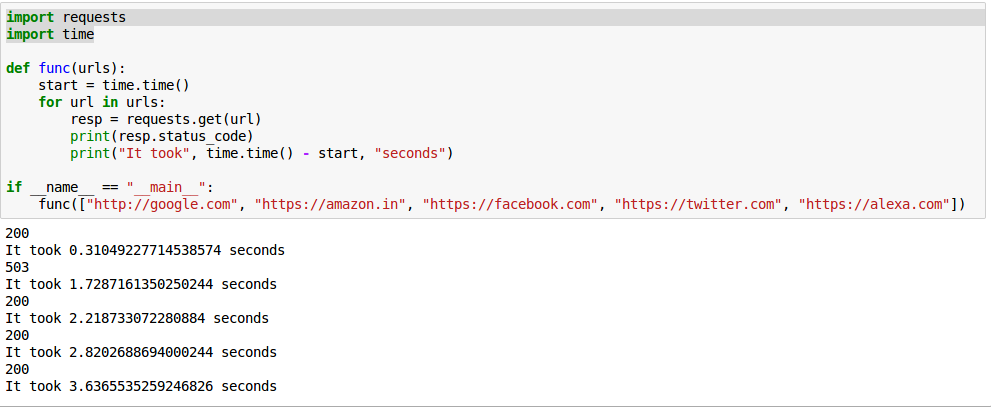
8.Storing Results:

To keep a track of task states various options are available: SQLAlchemy/Django ORM, Memcached, Redis, RPC (RabbitMQ/AMQP). Specify the selected backend using the backend keyword.

app = Celery('tasks', backend='redis://localhost', broker='pyamqp://')

More information on backends: [Backends](http://docs.celeryproject.org/en/latest/userguide/tasks.html#task-result-backends)

**3.A more practical example:**

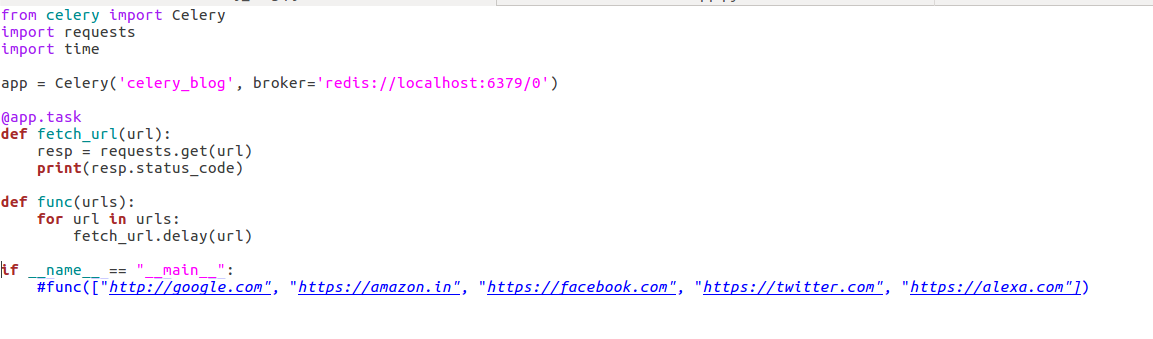


The code gets a list of urls along with the response time without celery support.

The above code has been taken from <http://agiliq.com/blog/2015/07/getting-started-with-celery-and-redis/> . Please refer for more information.

With a celery setup the main component is a celery worker, which would in this case perform the url fetching.

The following image shows the same program with celery enabled. Using Redis as backend.



A celery instance is needed for the celery setup hence the ‘app’ instance is created.

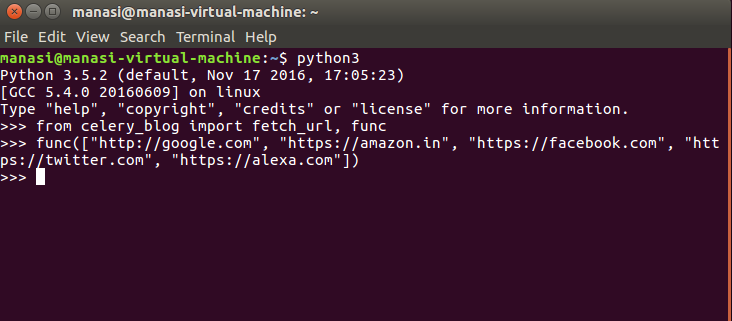
. **Run the file**:

$sudo service rabbitmq-server start

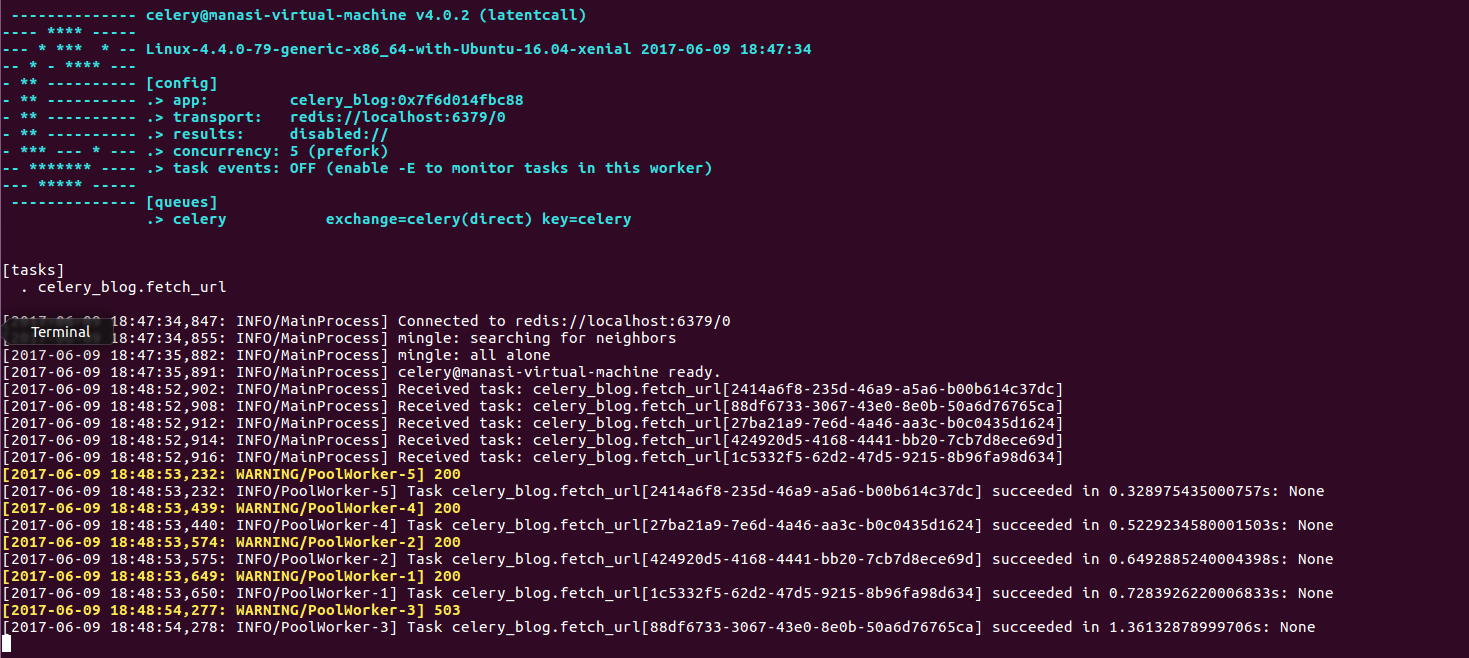
$ sudo redis-server

$ celery worker -A celery\_blog -l info -c 5 (specify concurrency).

In the python shell :



The worker console output for this run:



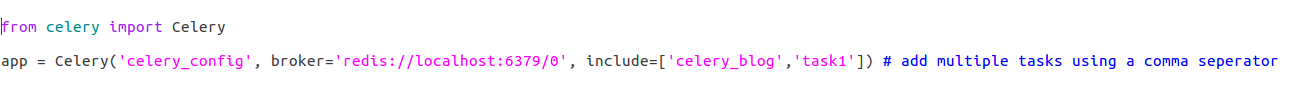
A celery worker can run multiple processes parallelly, as seen in the output console. The command

$ celery worker -A celery\_blog -l info -c 5 specified the number of concurrent processes we desire.

The time taken here is much less than the previous run of the program without celery support.

**4. To store the config file separately for multiple tasks in different modules**

Create a configuration file:



Modify the tasks.py

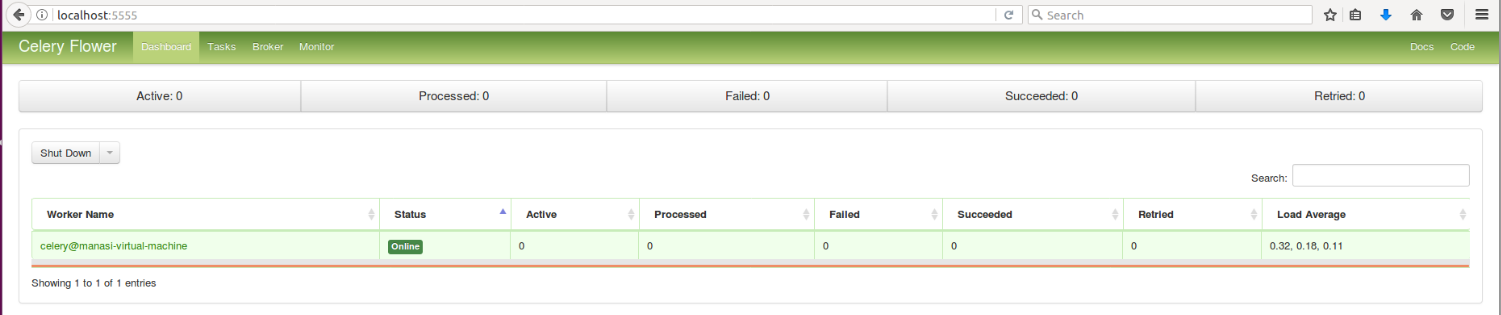


**5.Celery UI**

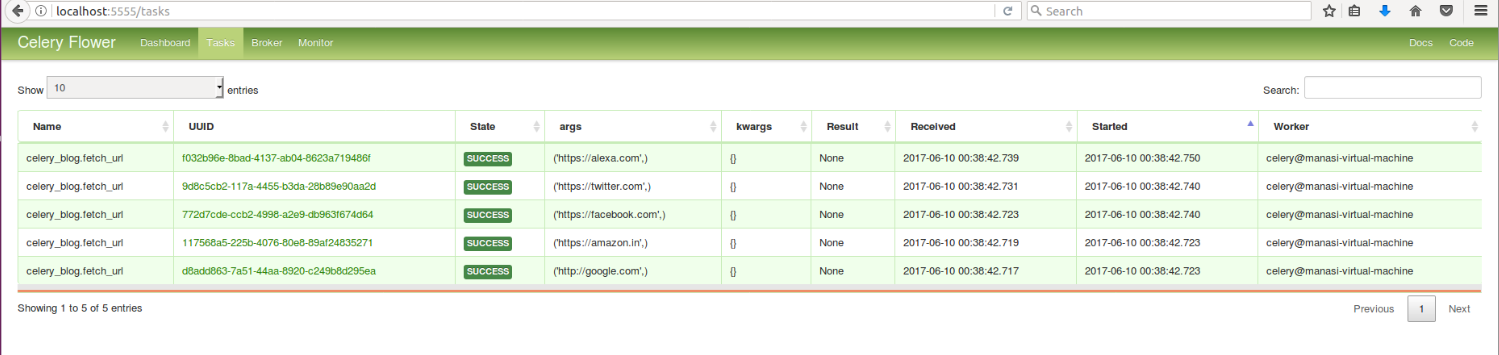
Celery user interface FLOWER can be used to view and monitor the tasks:

Download using $ pip install flower Access using : http://localhost:5555

Dashboard view:



Task view:



Additional Information and links:

Flower docs : <http://flower.readthedocs.io/en/latest/install.html>

Celery docs: <http://docs.celeryproject.org/en/latest/index.html>

Reference blogs:

for canvas work-flows: <http://docs.celeryproject.org/en/latest/userguide/canvas.html#chains>

<http://docs.celeryproject.org/en/latest/userguide/canvas.html#chains>

for celery on Windows:

<https://pratos.github.io/celery-setup-on-windows.html>

for Celery\_Rabbitmq tutorial:

<http://suzannewang.com/celery-rabbitmq-tutorial/>

for Celery hands-on commands:

<https://gist.github.com/amatellanes/a986f6babb9cf8556e36>

For a comparative study between DASK and Celery,

You can visit <http://matthewrocklin.com/blog/work/2016/09/13/dask-and-celery>

**Airflow**

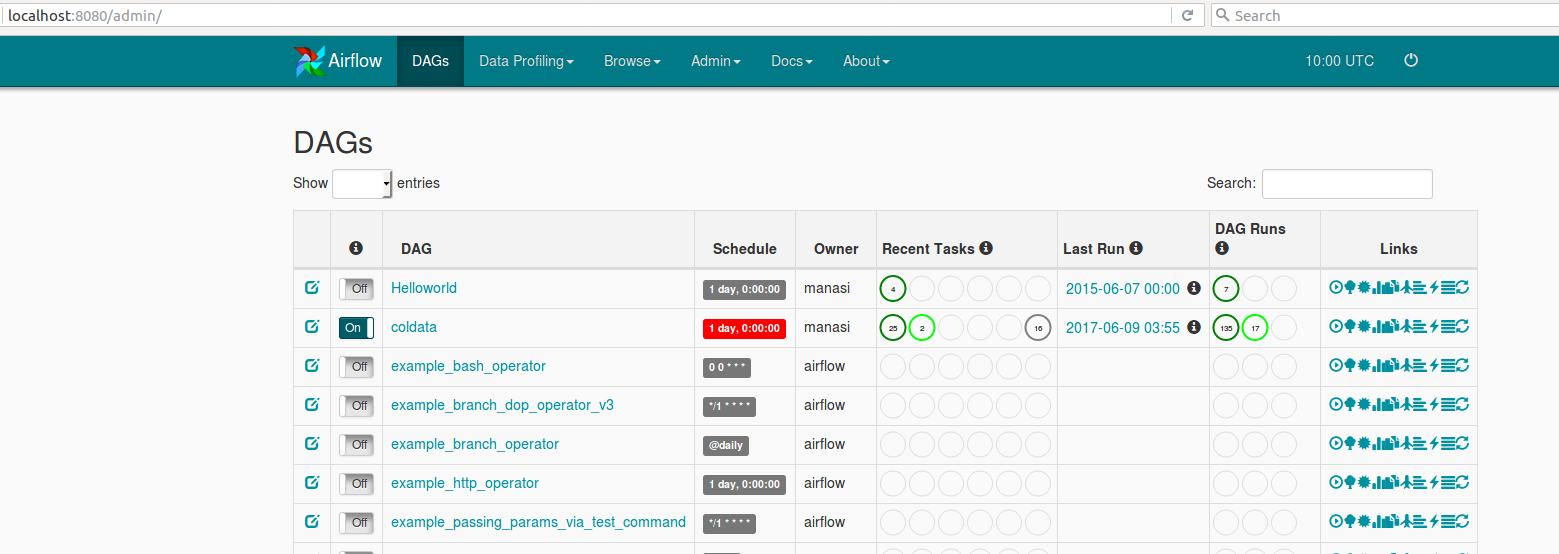
Airflow is a workflow management platform to programmatically author, schedule and monitor workflows.The workflows are directed acyclic graphs(DAGS) of the tasks to be performed and are executed on an array of workers by the Airflow scheduler.It was developed by Airbnb to meet their requirements for managing the growing amount of work. Data warehousing, Growth Analytics, Experimentation, Data infrastructure and maintenance are some of the processes fueled by Airflow at Airbnb.

“Airflow is written in Python from the ground up with a code base which is extensible, documented, consistent, linted and has broad unit test coverage. Airflow comes fully equipped with ways to interact with generally used systems like HIVE, MySQL, HDFS and allows easy triggering of arbitrary scripts and can be easily extended.” Extensibility components,

* **Hooks**, are external system abstractions sharing a homogenous interface. They use centralized vaults that abstracts host/port/login/password information and exposes methods to interact with these system.
* **Operators** leverage hooks to generate a certain type of task that become nodes in workflows when instantiated.
* **Executors** implement an interface that allow Airflow components (CLI, scheduler, web server) to run jobs jobs remotely. Airflow currently ships with a SequentialExecutor (for testing purposes), a threaded LocalExecutor, and a CeleryExecutor that leverages Celery, an excellent asynchronous task queue based on distributed message passing.

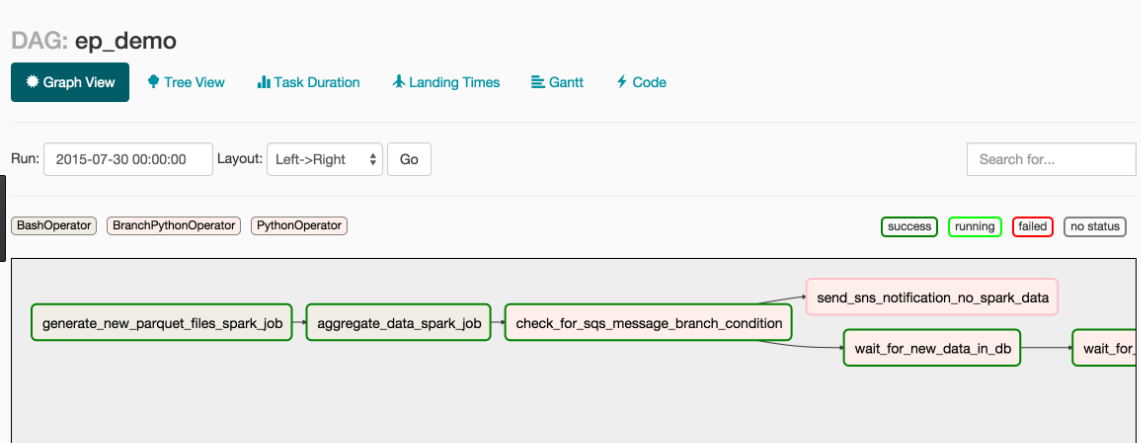
**Airflow UI:**

Airflow UI offers easy scheduling, monitoring and troubleshooting the data pipeline. UI can be accessed via <http://localhost:8080> once the webserver is started.



DAG view displaying the list of DAGs.

The UI also offers a Tree\_view representations of Dag runs spanning across time, a Gnatt chart to identify bottlenecks if any, Graph View to visualize the task dependencies(shown below).

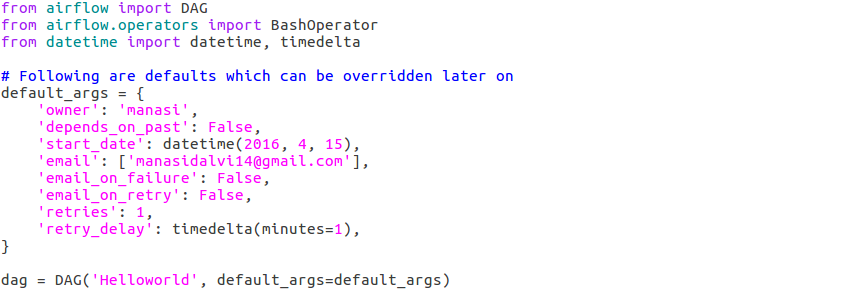


Graph view of a task

Airflow documentation provides additional information on the [UI](http://pythonhosted.org/airflow/ui.html)

**1.Airflow concepts:**

* DAG



* An Airflow’s **DAG** – **directed acyclic graph** – defines a workflow: which tasks should be executed, when and how. It does not do any actual data processing, it’s just a definition of what should be done.
* It is defined as a python script in the folder*$AIRFLOW\_HOME/dags*. The **scheduler** evaluates the script and triggers the dag runs and task executions when necessary. It will also reflect the changes if any.
* DAG must be placed in the globals() as Ariflow loads and DAG object it can import form Dagfile. If a DAG object appears in local scope it won’t be discovered and hence won’t be loaded.
* A dictionary of default\_arguments is passed to a DAG to apply it to its opertors without having to type it again for each operator.
* Operators
* Describe a single task and are used for executing the operations
* Usually stand-alone and don’t share resources. Types,
* Operators that performs an action, or tells another system to perform an action
* Transfer operators move data from a system to another
* Sensors are a certain type of operators that will keep running until a certain criteria is met.
* Tasks

Once an operator is instantiated, it is referred to as a “task”. The instantiation defines specific values when calling the abstract operator, and the parameterized task becomes a node in a DAG. Task instance is a task assigned to a DAG and has a state associated with a specific run of a DAG.

*Complex workflows are built by combining DAGs, Operators and Task Instances.*

**Airflow nomenclature:** summary of the terms used while designing Airflow workflows:

* Airflow DAGs are composed of Tasks.
* Each Task is created by instantiating an Operator class. A configured instance of an Operator becomes a Task, as in: my\_task = MyOperator(...).
* When a DAG is started, Airflow creates a DAG Run entry in its database.
* When a Task is executed in the context of a particular DAG Run, then a Task Instance is created.
* AIRFLOW\_HOME is the directory where you store your DAG definition files and Airflow plugins.

Additional information on Airflow [concepts](https://airflow.apache.org/concepts.html).

**2.Airflow Installation and Basic Commands**:

Please refer to this link for more information on installation and a simple helo world execution : <https://www.youtube.com/watch?v=VKUfHfCkmFY&t=330s>

$ export AIRFLOW\_HOME=~/airflow # set directory for airflow

$pip install apache-airflow # install airflow package

$pip install "apache-airflow[s3, postgres]" #add [subpackages](http://airflow.readthedocs.io/en/latest/installation.html#extra-packages) to be installed.

**Start the airflow meta db and webserver:**

# For the standalone mode, it could be a sqlite database, which applies to sequential executor only

$ airflow initdb

# start the webserver

$ airflow webserver -p 8080

Access using :

<http://localhost:8080/>

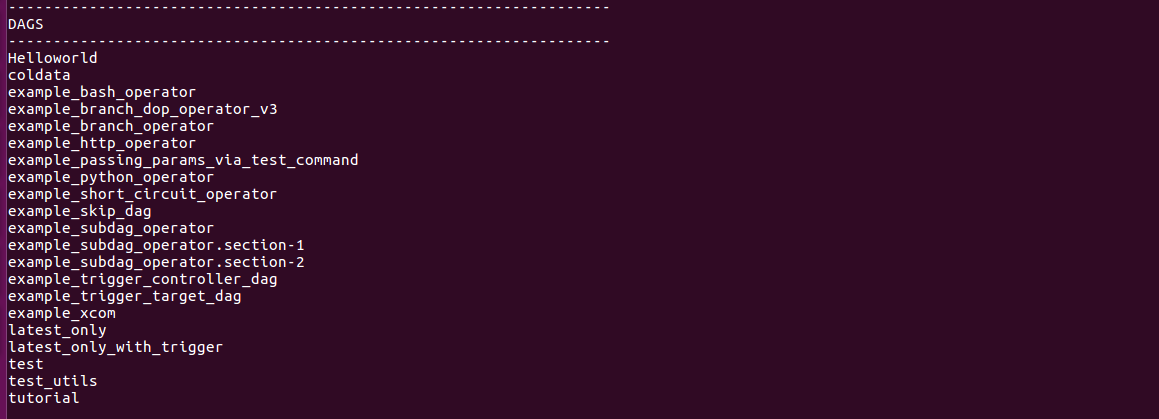
**Airflow command line interface:**

Airflow commands can be run from the command line interface other as it provides a rich CLI.

Save python files under the airflow/dags/hello.py

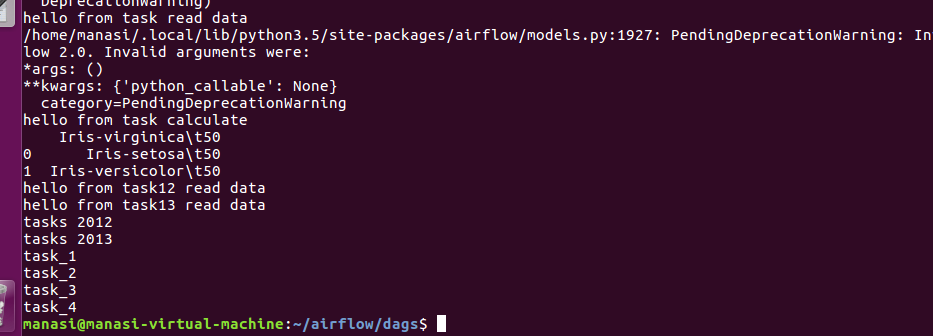
python3 ~/airflow/dags/tutorial.py

# print the list of active DAGs

dags$ airflow list\_dags 

# prints the list of tasks the "Helloworld" dag\_id

airflow list\_tasks tutorial Helloworld



# testing individual tasks in Helloworld

airflow test *Helloworld* 2015-06-01

# once the test run is successful, backfill command is used to consider dependencies, emit log files and communicate with the database.

airflow backfill *Helloworld* -s 2015-06-01 -e 2015-06-07

Visit <https://airflow.incubator.apache.org/tutorial.html> for more information.

**3.Let’s see an example of an entire workflow:**

We are building a workflow pipeline using data from two different sources and performing aggregation tasks and then writing the output to a file.

Step 1: Configure Airflow

1. Stand-alone mode with Sequential Exceuter

With Sequential Excecuter, jobs are picked and run sequentially, ie no parallelism. (Use for a quick start and feel of Airflow). Installing and configuring Airflow as mentioned above will give you the standard package.

For standard implementation visit : [installation](http://airflow.readthedocs.io/en/latest/installation.html)

1. Pseudo-distributed Mode Using Local Executor (Used in the example)

In pseudo-distributed mode, local executor is used. the local workers pick up and run jobs locally via multiprocessing and should be used with moderate amount of scheduled jobs.

Adopt another DB server other than sqlite(sequential exceuter), such as MySQL or PostgreSQL(used in example)

1. Install PostgreSQL :

*$ sudo apt-get install postgresql*

# Connect to the database:

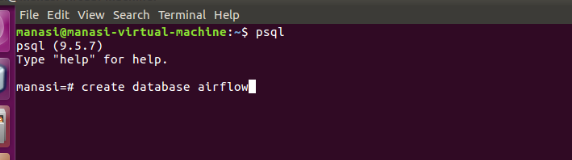
*$ psql*

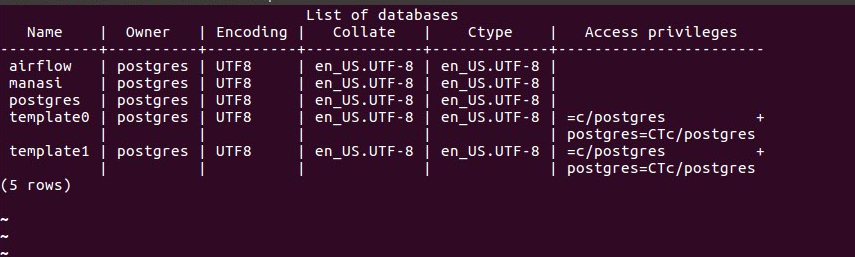
# Create a meta db for airflow:

*manasi=# create database airflow*

# Check if the database is created :

*manasi=# \l*





For more details on PostgreSql installation and database interactions visit [here](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-postgresql-on-ubuntu-14-04).

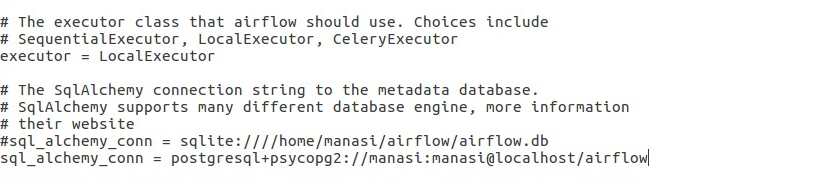
2.Modify the configuration in AIRFLOW\_HOME/airflow.cfg

# Change the executor to Local Executor

executor = LocalExecutor

# Change the meta db configuration. Note: the postgres username and password do not matter for now, since the database server and clients are still on the same host

sql\_alchemy\_conn = postgresql+psycopg2://your\_postgres\_user\_name:your\_postgres\_password@host\_name/database\_name



Configuration file

3.Restart airflow and test your dags:

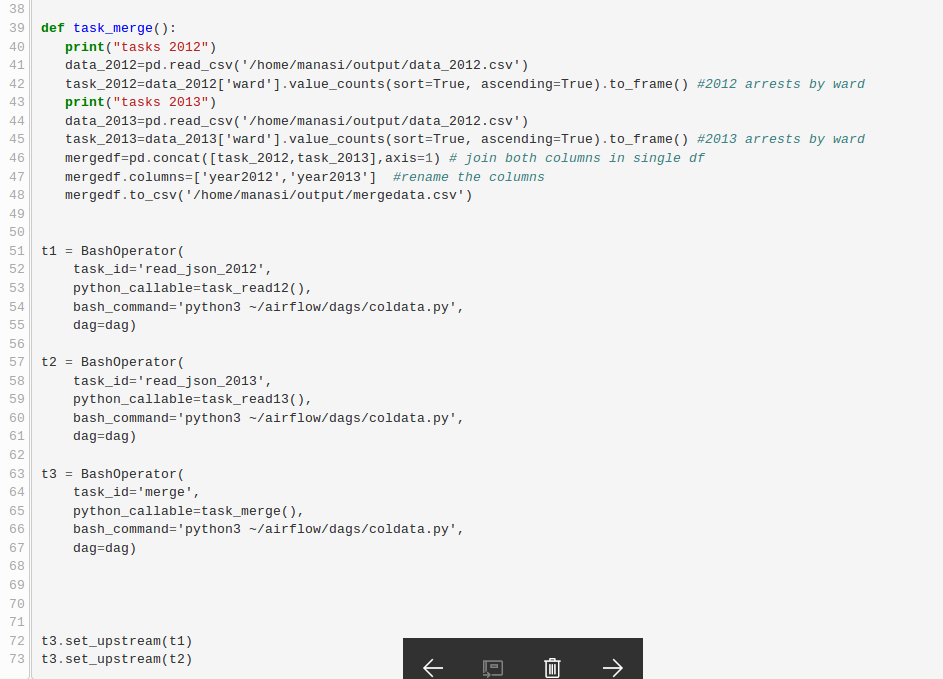
$ airflow initdb

$ airflow scheduler

$ airflow webserver

4.Python script :





The data pipeline consists of three tasks :

def task\_2012() # read Tucson, Az arrests in the year 2012 and stores on the local system.

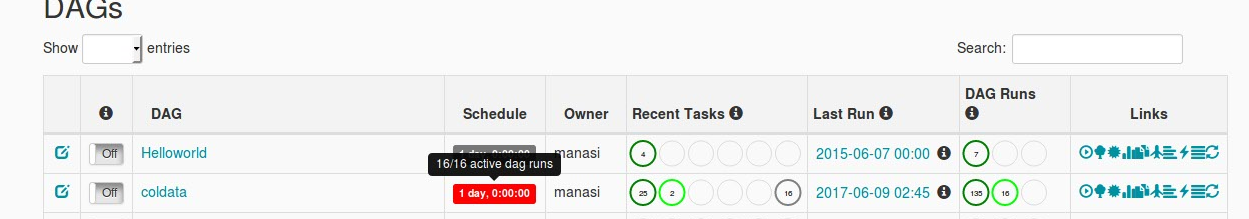
def task\_2013(): # read Tucson, Az arrests in the year 2013 and stores on the local system.

def task \_merge() : # calculates number of arrests made per ward in each year, merges the output and stores in on the local system.

In this case the task t3 is dependent on success of the previous two tasks and hence is up streamed to t1 and t2.

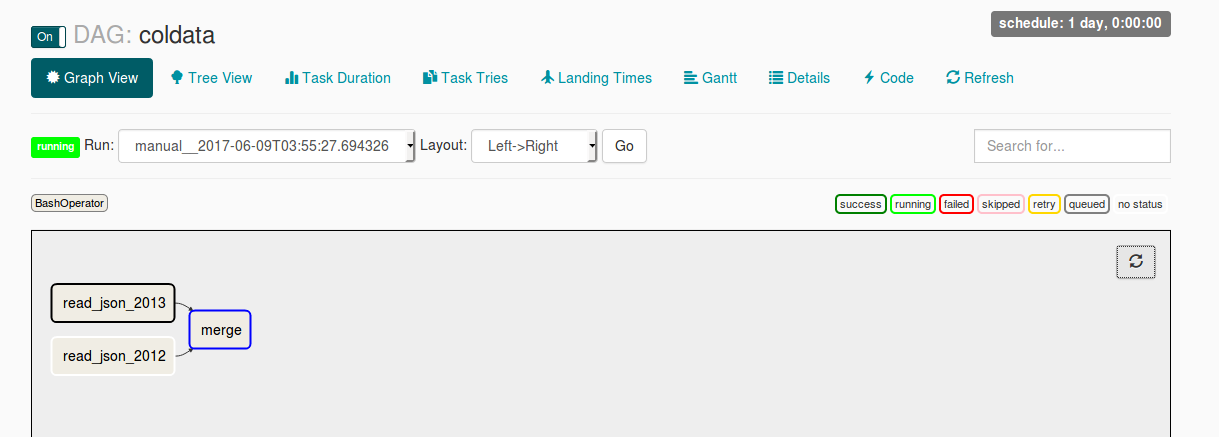
5.Run the Dag from the UI:

5.1 Load the UI: <http://localhost:8080> . The coldata dag will be listed in the dag list. change the off sign to on.

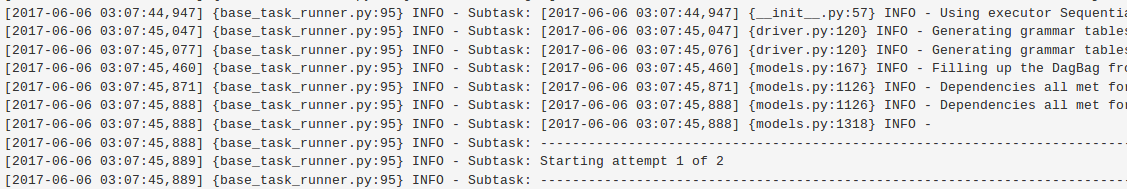


5.2 In order to start a DAG Run, first turn the workflow on (blue arrow), then click the Trigger Dag button (purple arrow) and finally, click on the Graph View (red arrow) to see the progress of the run.

5.3 View the status of the dag by clicking on the dag\_name and selecting graph view. Hovering over the individual task gives the status on the task. Color code displays quick status of the task. You can reload the graph view until both tasks reach the status Success.



5.4 When they are done, you can click on the merge task and then click View Log. If everything worked as expected, the log should show a number of lines and among them something like this:



**Additional information and links:**

Airflow official document **:**  <http://airflow.readthedocs.io/en/latest/index.html>

[Extend to Distributed Mode Using Celery Executor :](http://site.clairvoyantsoft.com/setting-apache-airflow-cluster/)

Under the distributed mode with a celery executor, remote workers pick up and run jobs as scheduled and load-balanced. As being highly scalable, it is the choice when you expect heavy and expanding loads.

[XComs](http://airflow.readthedocs.io/en/latest/concepts.html#xcoms):

Xcoms or “cross-communication” are used to share data/messages between tasks and are specifically designed for intertask communication rather than global settings.

[Airflow docker example](https://github.com/puckel/docker-airflow)

[Postgres installation and basics](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-postgresql-on-ubuntu-14-04)

Blog where you can find a comparison between Airflow, Luigi and Pinball

<https://www.michaelcho.me/article/data-pipelines-airflow-vs-pinball-vs-luigi>