Using MongoDB and Python for data analysis pipeline

Eoin Brazil, PhD, MSc
Proactive Technical Services, MongoDB

Github repo for this talk: http://github.com/braz/pycon2015_talk/

Hi I'm Eoin, I work with MongoDB in their proactive technical services team and I'm going to talk today about data pipelines with reference to Python and MongoDB. My talk will illustrate how you can leverage these to build operational data pipelines.

Before I start, I should firstly define a few things. Firstly what I mean when I say a data pipeline. It's the stringing together of the tools and systems necessary to move and transform your data to support the visualisation and modelling of your data. This allows you to ask questions of your data or provide dashboard or other data products/services.

Secondly, MongoDB is a document database that provides high performance, high availability, and easy scalability. One of the most popular next-generation database used in a variety of use cases/services from e-commerce product catalogues to real time analytics or single view realtime dashboards.

When designing a data product or service, it requires more than finding useful insights from the data. The product and services must be delivered at a profit to provide competitive advantage. I found a useful HBR blog by Thomas Redman and Bill Sweeney who highlighted the need for two departments when working with data strategies. A lab group and a factory group, the first acts as a research group (akin to Bell Labs, IBM Research or indeed any other corporate R&D group) who question and explore the data whilst the second is focused on more production issues. This second group would be typically staffed by engineers with deep technical skills, a short-term focus and a focus on the consistency, scale and economics of the business (e.g. decreasing the unit cost, predicting the next's month production demands, etc.). This is much more akin to traditional manufacturing. This second group and its concerns are the focus for this talk.

https://hbr.org/2013/04/two-departments-for-data-succe/

Workflow schedulers are pervasive – for instance, any company that has a data warehouse, a specialized database typically used for reporting, uses a workflow scheduler to coordinate nightly data loads into the data warehouse.

Breaking down the concerns for a production data pipeline, I'll look primarily at three topics

Pipelines - I'll briefly cover a number of tools to build a pipeline and focus specifically on Airbnb's tool, Airflow.

Systems - In terms of systems, I'll focus primarily on Sci-kit learn, PyMongo, and I'll introduce Monary. PyMongo and Monary are the two primary ways I'd recommend you interact with your data in MongoDB. These are the building blocks that make up the Python processing elements used to create a pipeline.

Speed - In terms of operational considerations, speed is important and if you've got a large amount of data to transform or move between processing stages you should probably consider Monary for MongoDB. Speed of development is also another important consideration and that's why all of these systems and tools utilise Python to build out your data pipeline.

There are a range of challenges for an operation pipeline. These include taking the data from many sources, formatting it, cleaning it and ensuring all of these tasks occurring without blocking the pipe.

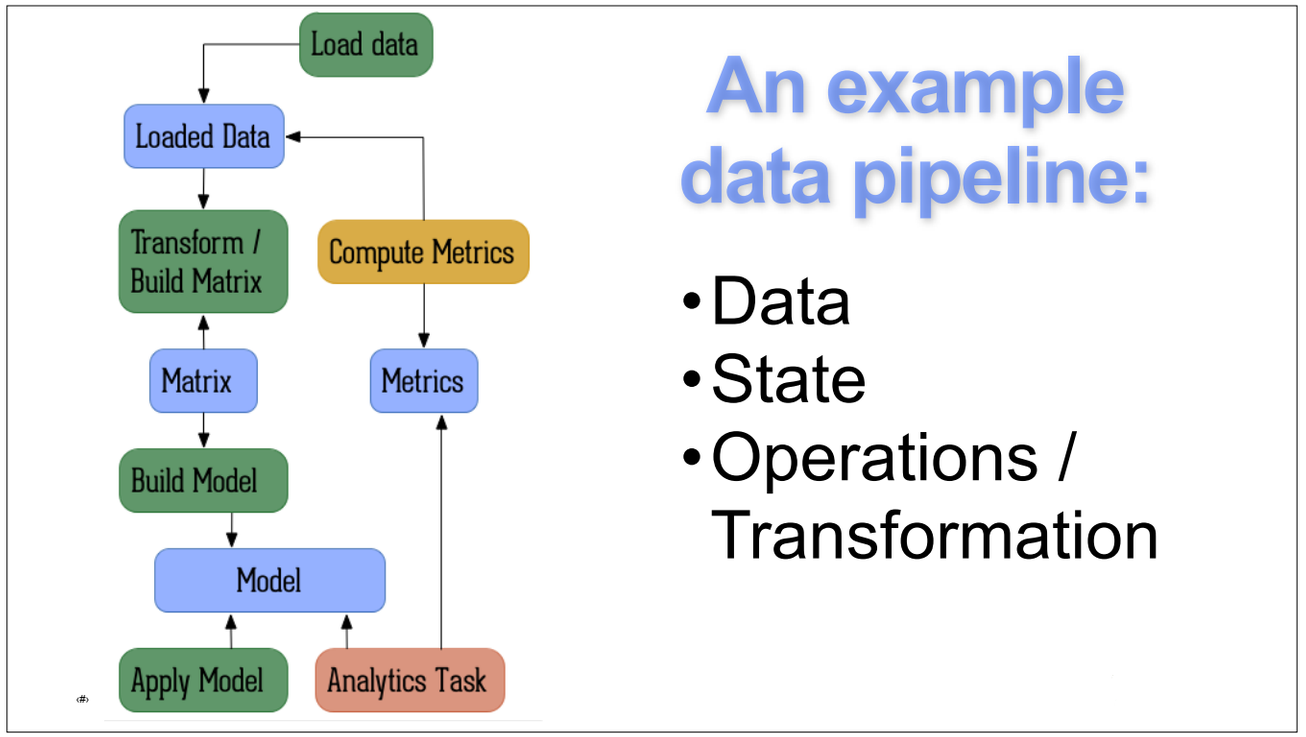
In terms of combining data, you may have a single view application where MongoDB or another DB is aggregating data from multiple sources into a central repository. You will often be dealing with different schemas and will need to design your pipeline to accommodate for this.

These pipelines are often well designed initially with careful tuning but as growth and changes occurring the system becomes stressed. Monitoring and measurement of the pipeline and the 'flow' are important to understand the high and low tide marks for your pipeline

Speedy and free flowing pipelines can be supported by ensuring the provision of adequate CPU, Memory, Disk and Network resources. Beyond ensuring there is sufficient resources, you should also apply any recommendations or guidelines for production. In the case of MongoDB that means you need to implement the production notes to ensure the appropriate configuration of the OS settings. In EC2 you used have to pre-warm the EBS storage volumes, this isn’t the case now but it was a definite speed bump if you where bringing data in cold.

Reproducibility is a key aspect that has gather momentum in the scientific research community, particular those where computation and computational modeling is now playing a much larger role. The ability to transfer and re-run whole swatches of computational models using tools such as notebooks (e.g. Jupyter) have a very safe and bright future. However, in terms of customer success with data pipelines my experience is that the trend goes towards simplification and speed potentially at the cost of some reproducibility. Production systems require many moving parts and whilst there is a simplicity to this type of artefact, it will likely add to the complexity of your eventual problem and your debugging woes.

The experience of the site reliability community and system administrators highlight the need to balance a raft of other concerns when you provide a ‘service’ rather than a prototype.



Let’s look at an example of a data pipeline, I’ve based this example off a talk given by another local tech company here in Dublin, AdRoll. In essence, a data pipeline can be broken into the three categories involving either data, a state change or the operations / transformations conducted on the data for each stage.

In this simplified example, it’s broken into two parts, on the left generating an ML model and on the right generating metrics (e.g. for a dashboard). The key understanding of graphic is that it is data that is transformation or changes state as you move through the stages.

In this presentation I’ll cover some tips on how you can improve the loading of data, on how you can leverage the database to compute metrics, and how you can manage the various stages together such as building or transforming your data.

Averaging a data set

- Python dictionary ~12 million numbers per second
- Python List 110 million numbers per second
- numpy.ndarray 500 million numbers per second

ndarray *or n-dimensional array, provides high-performance c-style arrays uses built-in maths libraries.*

If we look a little deeper at a common task for a data scientist, that of averaging all the data in a set. In terms of Python, you can see several order of improvements depending on how you load this data. It’s clear that whilst the Python List has a good performance, it is still significantly outclassed by the ndarray.

NumPy's arrays are more compact than Python lists, the equivalent of a 3D array in Python takes approximately 20 MB or so, while a NumPy 3D array with single-precision floats in the cells would fit in 4 MB. Access in reading and writing items is also faster with NumPy. Python’s lists are efficient general-purpose containers. They support (fairly) efficient insertion, deletion, appending, and concatenation, and Python’s list comprehensions make them easy to construct and manipulate. However, they have certain limitations: they don’t support “vectorized” operations like elementwise addition and multiplication, and the fact that they can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element. This also means that very few list operations can be carried out by efficient C loops unlike NumPy as they require type checks and other Python book keeping.

A NumPy array is basically described by metadata (number of dimensions, shape, data type, and so on) and the actual data. The data is stored in a homogeneous and contiguous block of memory, at a particular address in system memory (Random Access Memory, or RAM). This block of memory is called the data buffer. This is the main difference with a pure Python structure, like a list, where the items are scattered across the system memory. This aspect is the critical feature that makes NumPy arrays so efficient.

NumPy can also be linked to highly optimized linear algebra libraries like BLAS and LAPACK and can be linked to the Intel Math Kernel Library (MKL).

Data Transformations

C Zero Mean
S One SD
T √ log INV

} 1 - Predictors - M {

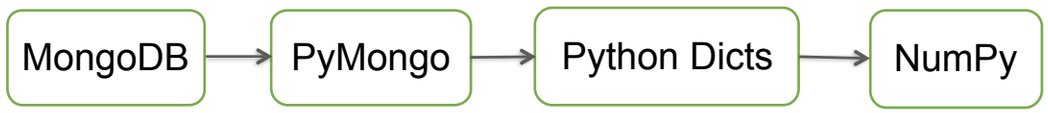
SS M-Dim Sphere
PCA
PLS

Correlation, Dummy Variables, Filtering

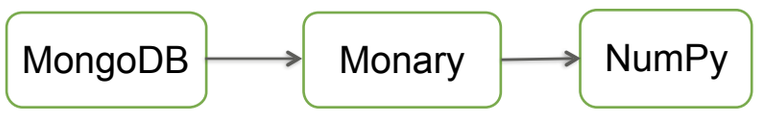
There a range of typical data transformation you might need to implement beyond averaging in a data set.

Workflows to / from MongoDB

PyMongo Workflow: ~150,000 documents per second

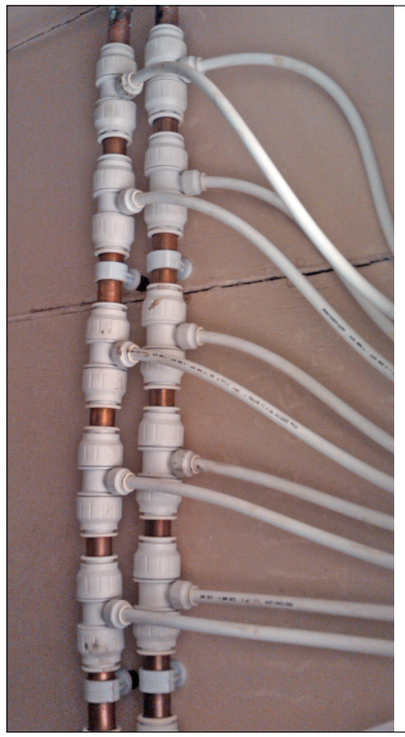


Monary Workflow: 1,700,000 documents per second



The speed and advantages of NumPy for loading / processing documents is clear but the typical approach to moving data from MongoDB to NumPy involved PyMongo and conversion to Python Dictionaries.

There is a new alternative to using PyMongo. There is a library called Monary specifically designed to speed the loading of bulk data reads from MongoDB to NumPy. Monary can only retrieve and store data. It cannot perform any updates or removals. Monary is a simple C library and accompanying Python wrapper which make use of MongoDB C driver. The code is designed to accept a list of desired fields, and to load exactly those fields from the BSON results into some provided array storage.



An example of connecting the pipes

- Monary
- MongoDB
- Python
- Airflow

Firstly dive into MongoDB's Aggregation & Monary

In the next part of my talk, I'll cover Monary and MongoDB in more depth as well as discussing Airflow which a tool to programmatically author, schedule and monitor workflows.

The next section will focus on explore the load aspect and the metric aspect of the example data pipeline I referred to earlier. In terms of MongoDB you can use leverage the Aggregation Framework within the database which itself is modelled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string. The Aggregation framework is the recommended approach for performing data aggregation within MongoDB. It replaces the requirement to use a Map-Reduce style alternative when aggregating data in MongoDB and provides this feature with the database rather than as an external tool such as Hadoop.

Data set and Aggregation

```
zips> db.data.findOne()
{ "_id" : "01001",
   "city" : "AGAWAM",
   "loc" : [
      -72.622739, 42.070206
   ],
   "pop" : 15338,
   "state" : "MA"
}
```

- ID
- City name
- Lat/Long
- Population
- State

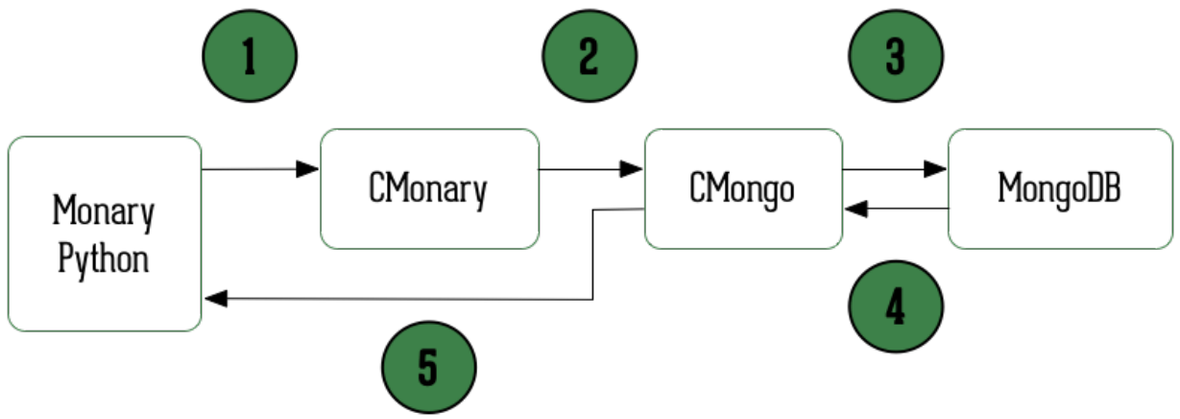
```
pipeline = [{"$group" :
{"_id" : "$state",
"totPop" : {"$sum" :
"$pop"}}}]
```

1. Group documents by state
2. Sum the population value for each invidiual city to give a state population total

In terms of aggregation, MongoDB has a number of example data sets, I'll focus on the 'zips' US city, location and population data set. Each document in this data set has an ID, the city or town's name, the specific latitude and longitude as well as the population and the state it is in.

We'll examine a simple pipeline with Monary. I'll focus on explaining the pipeline firstly. It firstly groups all the documents in this collection by State using the $group operator. The second stage of this pipeline then sums the population value of each of the cities and towns to give a total value of population for that state.

Monary Query



In terms of Monary, as I mentioned it is a simple C library and accompanying Python wrapper which make use of MongoDB C driver. The code is designed to accept a list of desired fields, and to load exactly those fields from the BSON results into some provided array storage.

The mechanics of a Monary query are as follows. It firstly send the data from the Python Monary to the Cmonary the c library which then calls the MongoDB C driver. This driver then queries the database and returns the data which the C driver marshalls and returns to the Python Monary wrapper directly.

If you need to use the aggregation option allowDiskUse, you'll need to recompile the source with this patch. Without this there is a limit of 100 megabytes of RAM in the aggregation for MongoDB.
https://bitbucket.org/djcbeach/monary/issues/14/passing-options-to-aggregate
Feature request to add **kwargs and BSON-encode the kwargs dict to "opts"

Monary Query

```
>>> from monary import Monary
>>> m = Monary()
>>> pipeline = [{"$group" : {"_id" :
"$state", "totPop" : {"$sum" : “$pop"}}}]
>>> states, population =
m.aggregate("zips","data", pipeline,
["_id","totpop"], ["string:2", "int64"])
```


Here’s how you would call this example query in Monary.

You should note a few items about this slide. Firstly, there’s no difference between the pipeline code in Python or if you where to run this pipeline in the Mongo shell / console.

In terms of the call to the aggregation with Monary you should note that you need to pass both the field names and their types.

I’ll walk through the aggregation call to focus on each parameter in more detail.

Monary Query

```
>>> from monary import Monary
>>> m = Monary()
>>> pipeline = [{"$group" : {"_id" :
"$state", "totPop" : {"$sum" : “$pop"}}}]
>>> states, population =
m.aggregate("zips", "data", pipeline,
["_id","totpop"], ["string:2", "int64"])
```

Database

Monary Query

```
>>> from monary import Monary
>>> m = Monary()
>>> pipeline = [{"$group" : {"_id" :
"$state", "totPop" : {"$sum" : “$pop"}}}]
>>> states, population =
m.aggregate("zips","data", pipeline,
["_id","totpop"], ["string:2", "int64"])
```

Field Name

Monary Query

```
>>> from monary import Monary
>>> m = Monary()
>>> pipeline = [{"$group" : {"_id" :
"$state", "totPop" : {"$sum" : “$pop"}}}]
>>> states, population =
m.aggregate("zips","data", pipeline,
["_id","totpop"], ["string:2", "int64"])
```

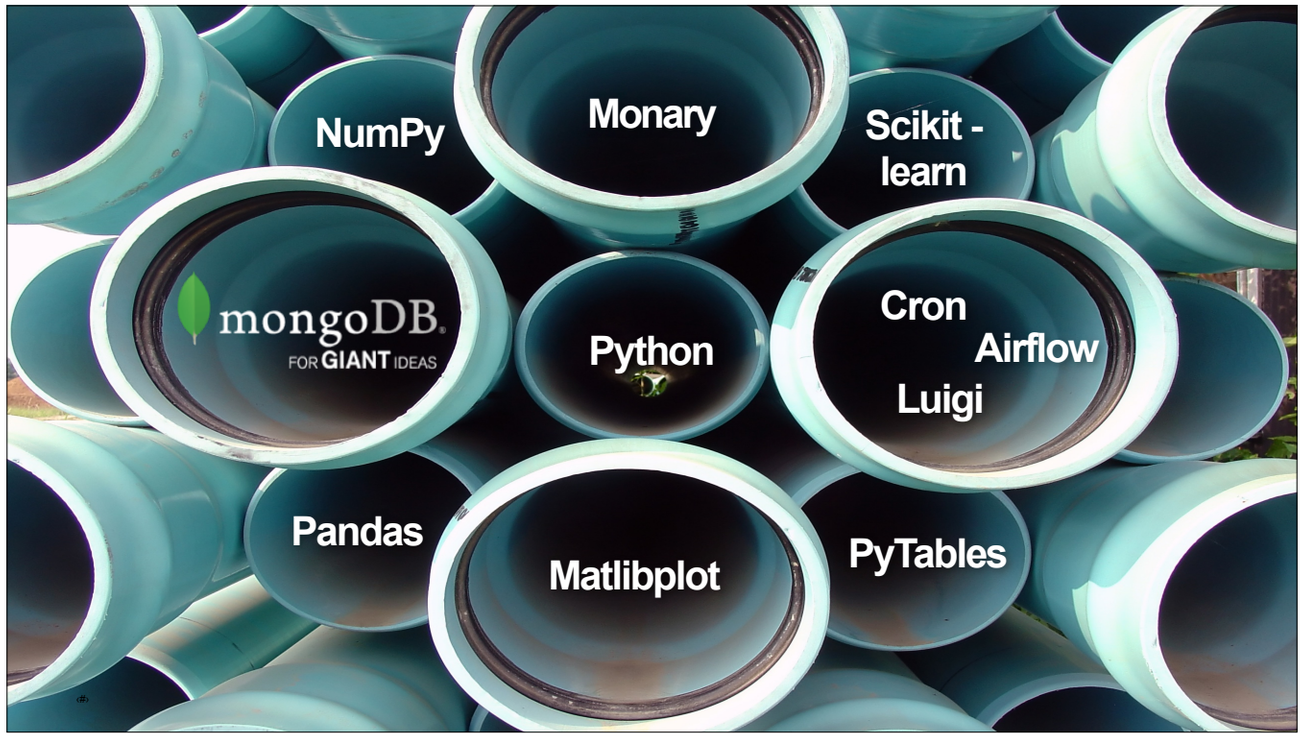
Return type

Aggregation Result

[u'WA: 4866692', u'HI: 1108229', u'CA: 29754890', u'OR: 2842321', u'NM: 1515069', u'UT: 1722850', u'OK: 3145585', u'LA: 4217595', u'NE: 1578139', u'TX: 16984601', u'MO: 5110648', u'MT: 798948', u'ND: 638272', u'AK: 544698', u'SD: 695397', u'DC: 606900', u'MN: 4372982', u'ID: 1006749', u'KY: 3675484', u'WI: 4891769', u'TN: 4876457', u'AZ: 3665228', u'CO: 3293755', u'KS: 2475285', u'MS: 2573216', u'FL: 12686644', u'IA: 2776420', u'NC: 6628637', u'VA: 6181479', u'IN: 5544136', u'ME: 1226648', u'WV: 1793146', u'MD: 4781379', u'GA: 6478216', u'NH: 1109252', u'NV: 1201833', u'DE: 666168', u'AL: 4040587', u'CT: 3287116', u'SC: 3486703', u'RI: 1003218', u'PA: 11881643', u'VT: 562758', u'MA: 6016425', u'WY: 453528', u'MI: 9295297', u'OH: 10846517', u'AR: 2350725', u'IL: 11427576', u'NJ: 7730188', u'NY: 17990402']

Aggregation Result

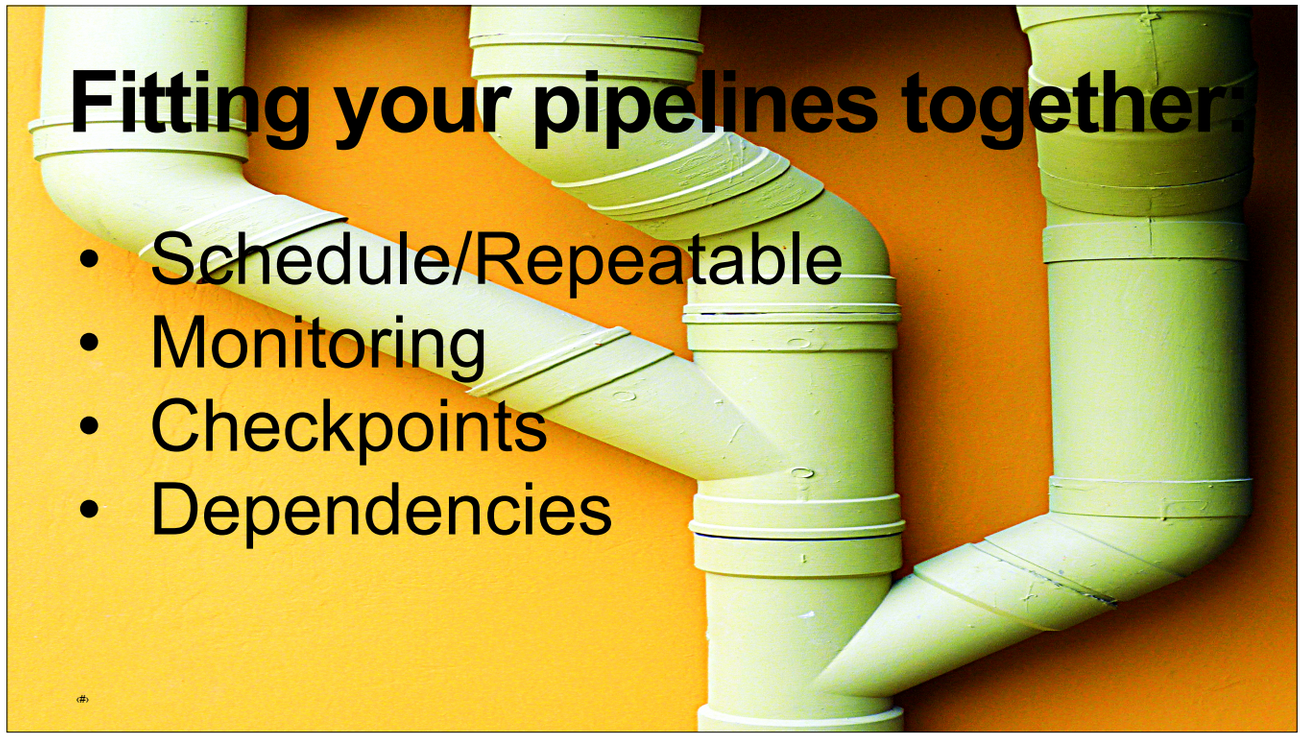
```
[u'WA: 4866692', u'HI: 1108229', u'CA: 29754890', u'OR: 2842321', u'NM:
1515069', u'UT: 1722850', u'OK: 3145585', u'LA: 4217595', u'NE: 1578139', u'TX:
16984601', u'MO: 5110648', u'MT: 798948', u'ND: 638272', u'AK: 544698', u'SD:
695397', u'DC: 606900', u'MN: 4372982', u'ID: 1006749', u'KY: 3675484', u'WI:
4891769', u'TN: 4876457', u'AZ: 3665228', u'CO: 3293755', u'KS: 2475285', u'MS:
2573216', u'FL: 12686644', u'IA: 2776420', u'NC: 6628637', u'VA: 6181479', u'IN:
5544136', u'ME: 1226648', u'WV: 1793146', u'MD: 4781379', u'GA: 6478216', u'NH:
1109252', u'NV: 1201833', u'DE: 666168', u'AL: 4040587', u'CT: 3287116', u'SC:
3486703', u'RI: 1003218', u'PA: 11881643', u'VT: 562758', u'MA: 6016425', u'WY:
453528', u'MI: 9295297', u'OH: 10846517', u'AR: 2350725', u'IL: 11427576', u'NJ:
7730188', u'NY: 17990402']
```



Operationalising a data pipeline in Python involves a number of tools, I've covered MongoDB, Monary, and NumPy briefly. The examples earlier in the talk are a good example of exploratory work but when you change from the 'lab' like environment to the 'factory' like environment you also need to address other concerns. In order to focus on these concerns, I'll introduce some other tools/systems these include Airflow for workflows, Matlibplot for visualisation, Pandas for data manipulation, and Scikit-learn for the machine learning, model generation and analysis.

Airflow - Job scheduling with dependencies that can be triggered based on conditions such as repeating intervals or on completion of other jobs is an important requirement to building a dynamic pipeline. These tools all support this type of scheduling and are an important evolution of old favourites such as cron.

There are a range of other tools and systems that I won't cover in this talk but for every tool I do cover there is probably another two to three OSS projects working on a similar system.

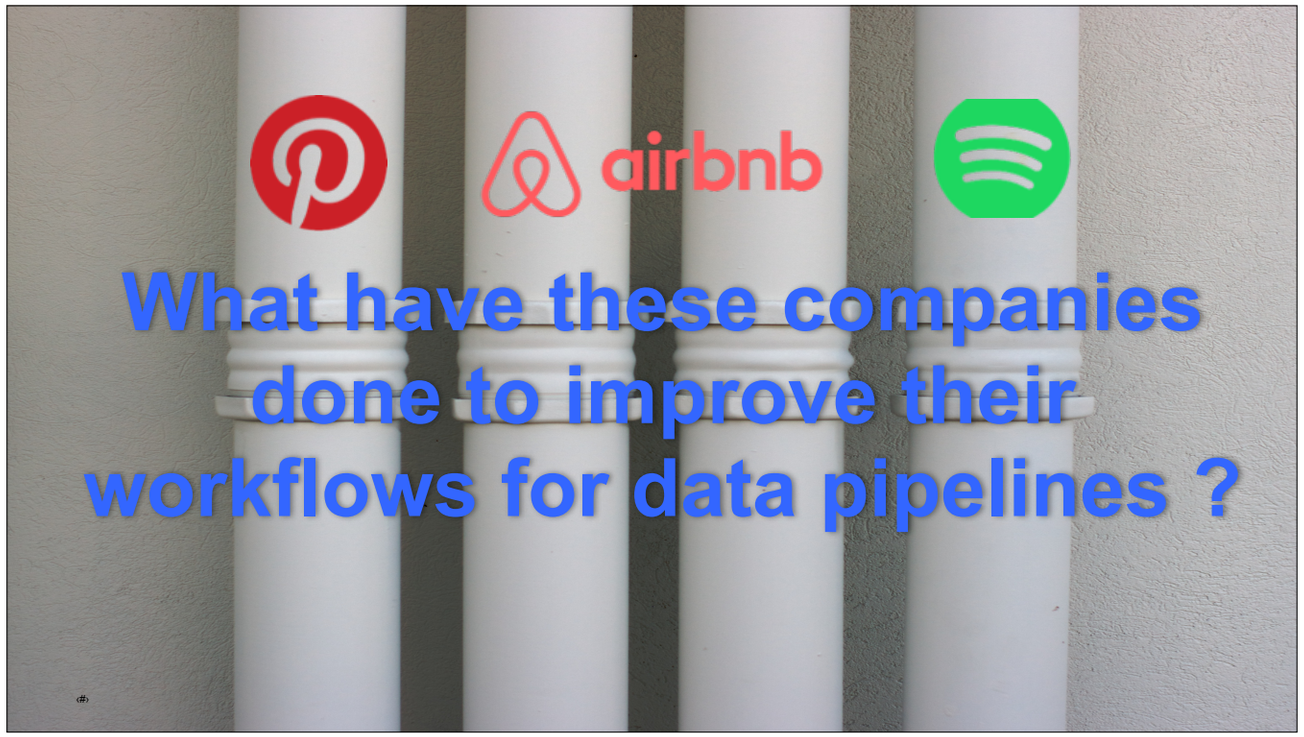


In terms of actual concerns or needs for connecting these components together there are a few important considerations that you need to consider.
You need to ensure that the both the individual stages and the entire pipeline is repeatable and that it can be scheduled. In some cases, you might want to rebuilt a demand forecasting model every day or every hour but you may want to update your dashboard / metrics every minute or more frequently, depending on the business requirement.

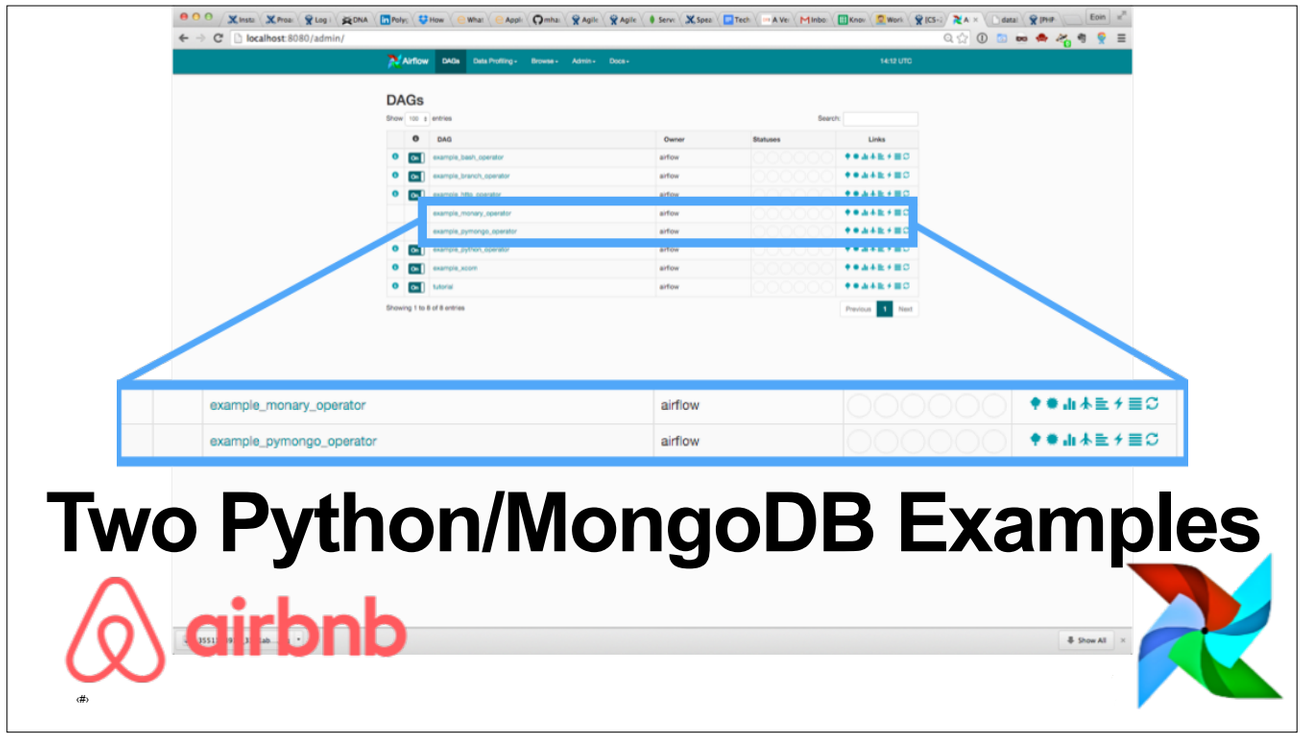
In terms of tracking and monitoring, you need to ensure that your pipeline and the stages within the pipeline can be monitored. This also you to see any issues as they occur, flag warnings or alerts if thresholds are breached and performance the normal maintenance tasks as indicated. Prometheus is a monitoring tool that supports both push and pull monitoring, there are a range of these tools and systems. In terms of MongoDB itself, we recommend using Cloud Manager or Ops Manager for an on premise solution. These tools can allow you to track your service and the components to ensure it is functioning reliably.

Checkpoints are an important feature that have not really been used outside of scientific modelling or high performance computing. A checkpoint takes a snapshot of the calculation and working set for data for that calculation so that if your system crashes you can resume from that point in the process. In many cases, this can help ensure you don't have retry computational intensive calculations or suffer from a cold-start situation for your model.

Dependencies are also important to manage in a pipeline as you often want to ensure a specific ordering of calculations within your pipeline. For instance, you want to ensure that a certain model is created or re-created at a point before being used by another component. If this was not ordered correctly, you could end up using an older model or information.



they all built workflow tools, in some case multiple tools as they learnt more. For example, Airbnb started with Cron then wrote Chronos and have now moved to Airflow.
Spotify's Luigi, OpenStack's Mistral, Pinterest's Pinball, and recently AirBnb's Airflow

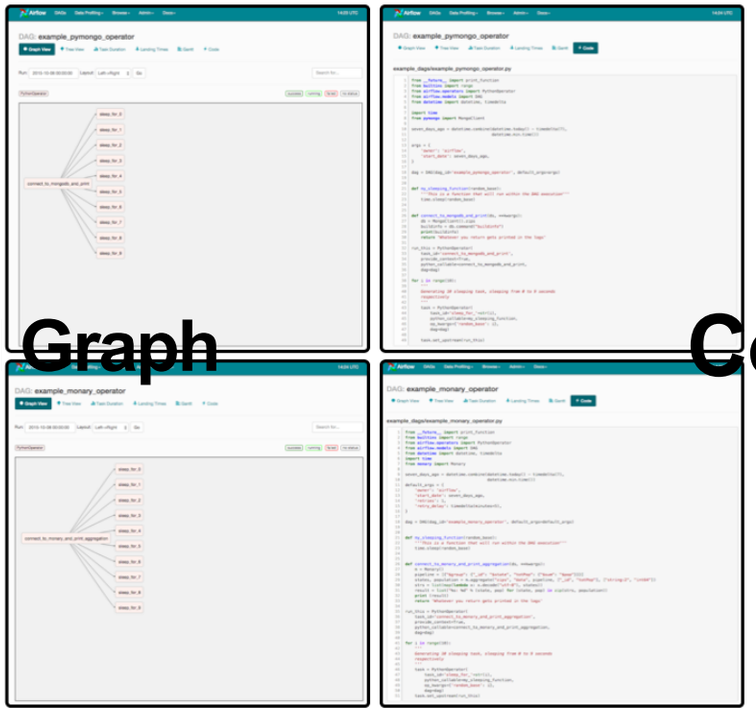


A developer defines his DAG in a Python script. The DAG is then automatically loaded into the DAG engine and scheduled for its first run. Modifying a DAG is as easy as modifying the Python script.

Visual

Graph

Code



example_monary_operator.py

```
from __future__ import print_function
from builtins import range
from airflow.operators import PythonOperator
from airflow.models import DAG
from datetime import datetime, timedelta
import time
from monary import Monary

seven_days_ago = datetime.combine(datetime.today() - timedelta(7),
                                  datetime.min.time())
default_args = {
    'owner': 'airflow',
    'start_date': seven_days_ago,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(dag_id='example_monary_operator', default_args=default_args)

def my_sleeping_function(random_base):
    '''This is a function that will run within the DAG execution'''
    time.sleep(random_base)
```

example_monary_operator.py

```python
from __future__ import print_function
from builtins import range
from airflow.operators import PythonOperator
from airflow.models import DAG
from datetime import datetime, timedelta
import time
from monary import Monary

seven_days_ago = datetime.combine(datetime.today() - timedelta(7),
                                  datetime.min.time())
default_args = {
    'owner': 'airflow',
    'start_date': seven_days_ago,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(dag_id='example_monary_operator', default_args=default_args)

def my_sleeping_function(random_base):
    '''This is a function that will run within the DAG execution'''
    time.sleep(random_base)
```

IMPORTS

example_monary_operator.py

```python
from __future__ import print_function
from builtins import range
from airflow.operators import PythonOperator
from airflow.models import DAG
from datetime import datetime, timedelta
import time
from monary import Monary

seven_days_ago = datetime.combine(datetime.today() - timedelta(7),
                                  datetime.min.time())
default_args = {
    'owner': 'airflow',
    'start_date': seven_days_ago,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(dag_id='example_monary_operator', default_args=default_args)

def my_sleeping_function(random_base):
    '''This is a function that will run within the DAG execution'''
    time.sleep(random_base)
```

SETTINGS

example_monary_operator.py

```python
from __future__ import print_function
from builtins import range
from airflow.operators import PythonOperator
from airflow.models import DAG
from datetime import datetime, timedelta
import time
from monary import Monary

seven_days_ago = datetime.combine(datetime.today() - timedelta(7),
                                  datetime.min.time())
default_args = {
    'owner': 'airflow',
    'start_date': seven_days_ago,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(dag_id='example_monary_operator', default_args=default_args)

def my_sleeping_function(random_base):
    '''This is a function that will run within the DAG execution'''
    time.sleep(random_base)
```

DAG & Functions

example_monary_operator.py

```
from __future__ import print_function
from builtins import range
from airflow.operators import PythonOperator
from airflow.models import DAG
from datetime import datetime, timedelta
import time
from monary import Monary

seven_days_ago = datetime.combine(datetime.today() - timedelta(7),
                                  datetime.min.time())
default_args = {
    'owner': 'airflow',
    'start_date': seven_days_ago,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(dag_id='example_monary_operator', default_args=default_args)

def my_sleeping_function(random_base):
    '''This is a function that will run within the DAG execution'''
    time.sleep(random_base)
```

example_monary_operator.py

```
def connect_to_monary_and_print_aggregation(ds, **kwargs):
    m = Monary()
    pipeline = [{"$group": {"_id": "$state", "totPop": {"$sum":
"$pop"}}}]
    states, population = m.aggregate("zips", "data", pipeline, ["_id",
"totPop"], ["string:2", "int64"])
    strs = list(map(lambda x: x.decode("utf-8"), states))
    result = list("%s: %d" % (state, pop) for (state, pop) in
zip(strs, population))
    print (result)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='connect_to_monary_and_print_aggregation',
    provide_context=True,
    python_callable=connect_to_monary_and_print_aggregation,
    dag=dag)
```

example_monary_operator.py

AGGREGATION

```
def connect_to_monary_and_print_aggregation(ds, **kwargs):
    m = Monary()
    pipeline = [{"$group": {"_id": "$state", "totPop": {"$sum":
"$pop"}}}]
    states, population = m.aggregate("zips", "data", pipeline, ["_id",
"totPop"], ["string:2", "int64"])
    strs = list(map(lambda x: x.decode("utf-8"), states))
    result = list("%s: %d" % (state, pop) for (state, pop) in
zip(strs, population))
    print (result)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='connect_to_monary_and_print_aggregation',
    provide_context=True,
    python_callable=connect_to_monary_and_print_aggregation,
    dag=dag)
```

example_monary_operator.py

```
def connect_to_monary_and_print_aggregation(ds, **kwargs):
    m = Monary()
    pipeline = [{"$group": {"_id": "$state", "totPop": {"$sum":
"$pop"}}}]
    states, population = m.aggregate("zips", "data", pipeline, ["_id",
"totPop"], ["string:2", "int64"])
    strs = list(map(lambda x: x.decode("utf-8"), states))
    result = list("%s: %d" % (state, pop) for (state, pop) in
zip(strs, population))
    print (result)
    return 'Whatever you return gets printed in the logs'
```

DAG SETUP

```
run_this = PythonOperator(
    task_id='connect_to_monary_and_print_aggregation',
    provide_context=True,
    python_callable=connect_to_monary_and_print_aggregation,
    dag=dag)
```

example_monary_operator.py

```
def connect_to_monary_and_print_aggregation(ds, **kwargs):
    m = Monary()
    pipeline = [{"$group": {"_id": "$state", "totPop": {"$sum":
"$pop"}}}]
    states, population = m.aggregate("zips", "data", pipeline, ["_id",
"totPop"], ["string:2", "int64"])
    strs = list(map(lambda x: x.decode("utf-8"), states))
    result = list("%s: %d" % (state, pop) for (state, pop) in
zip(strs, population))
    print (result)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='connect_to_monary_and_print_aggregation',
    provide_context=True,
    python_callable=connect_to_monary_and_print_aggregation,
    dag=dag)
```

example_monary_operator.py

```
for i in range(10):
    '''
    Generating 10 sleeping tasks, sleeping from 0 to 9
seconds
    respectively
    '''
    task = PythonOperator(
        task_id='sleep_for_'+str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': i},
        dag=dag)
    task.set_upstream(run_this)
```

example_monary_operator.py

LOOP

```
for i in range(10):
    '''
    Generating 10 sleeping tasks, sleeping from 0 to 9
seconds
    respectively
    '''
    task = PythonOperator(
        task_id='sleep_for_'+str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': i},
        dag=dag)
    task.set_upstream(run_this)
```

example_monary_operator.py

```
for i in range(10):
    '''
    Generating 10 sleeping tasks, sleeping from 0 to 9
seconds
    respectively
    '''
    task = PythonOperator(
        task_id='sleep_for_'+str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': i},
        dag=dag)
    task.set_upstream(run_this)
```

DAG SETUP

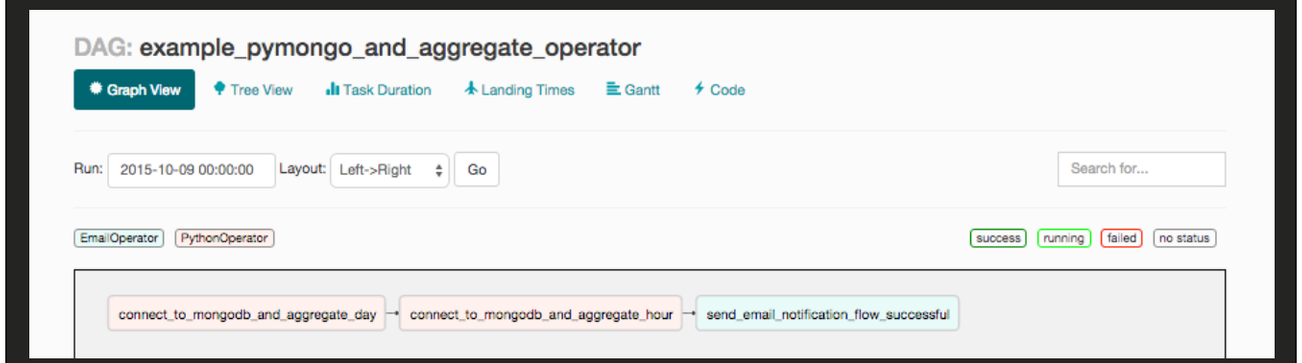
example_monary_operator.py

```
for i in range(10):
    '''
    Generating 10 sleeping tasks, sleeping from 0 to 9
seconds
    respectively
    '''
    task = PythonOperator(
        task_id='sleep_for_'+str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': i},
        dag=dag)
    task.set_upstream(run_this)
```

example_monary_operator.py

```
$ airflow backfill example_monary_operator -s 2015-01-01 -e 2015-01-02
2015-10-08 15:08:09,532 INFO - Filling up the DagBag from /Users/braz/airflow/dags
2015-10-08 15:08:09,532 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_bash_operator.py
2015-10-08 15:08:09,533 INFO - Loaded DAG <DAG: example_bash_operator>
2015-10-08 15:08:09,533 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_branch_operator.py
2015-10-08 15:08:09,534 INFO - Loaded DAG <DAG: example_branch_operator>
2015-10-08 15:08:09,534 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_http_operator.py
2015-10-08 15:08:09,535 INFO - Loaded DAG <DAG: example_http_operator>
2015-10-08 15:08:09,535 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_monary_operator.py
2015-10-08 15:08:09,719 INFO - Loaded DAG <DAG: example_monary_operator>
2015-10-08 15:08:09,719 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_pymongo_operator.py
2015-10-08 15:08:09,738 INFO - Loaded DAG <DAG: example_pymongo_operator>
2015-10-08 15:08:09,738 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_python_operator.py
2015-10-08 15:08:09,739 INFO - Loaded DAG <DAG: example_python_operator>
2015-10-08 15:08:09,739 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/example_xcom.py
2015-10-08 15:08:09,739 INFO - Loaded DAG <DAG: example_xcom>
2015-10-08 15:08:09,739 INFO - Importing /usr/local/lib/python2.7/site-packages/airflow/example_dags/tutorial.py
2015-10-08 15:08:09,740 INFO - Loaded DAG <DAG: tutorial>
2015-10-08 15:08:09,819 INFO - Adding to queue: airflow run example_monary_operator connect_to_monary_and_print_aggregation 2015-01-02T00:00:00   --local     -sd DAGS_FOLDER/example_dags/
example_monary_operator.py -s 2015-01-01T00:00:00
2015-10-08 15:08:09,865 INFO - Adding to queue: airflow run example_monary_operator connect_to_monary_and_print_aggregation 2015-01-01T00:00:00   --local     -sd DAGS_FOLDER/example_dags/
example_monary_operator.py -s 2015-01-01T00:00:00
2015-10-08 15:08:14,765 INFO - [backfill progress] waiting: 22 | succeeded: 0 | kicked_off: 2 | failed: 0 | skipped: 0
2015-10-08 15:08:19,765 INFO - commandairflow run example_monary_operator connect_to_monary_and_print_aggregation 2015-01-02T00:00:00   --local     -sd DAGS_FOLDER/example_dags/
example_monary_operator.py -s 2015-01-01T00:00:00
Logging into: /Users/braz/airflow/logs/example_monary_operator/connect_to_monary_and_print_aggregation/2015-01-02T00:00:00
u'AK: 544698', u'SD: 695397', u'DC: 606900', u'MN: 4372982', u'ID: 1006749', u'KY: 3675484', u'WI: 4891769', u'TN: 4876457', u'AZ: 3665228', u'CO: 3293755', u'KS: 2475285', u'MS: 2573216', u'FL: 12686644', u'IA:
```

Building your pipeline




```
 pipeline = [{"$project":{'page': '$PAGE', 'time': { 'y':
{'$year':'$DATE'} , 'm':{'$month':'$DATE'}, 'day':
{'$dayOfMonth':'$DATE'}}}}, {'$group':{'_id':
{'p':'$page','y':'$time.y','m':'$time.m','d':'$time.day'}, 'daily':
{'$sum':1}}},{'$out': tmp_created_collection_per_day_name}]
```

Building your pipeline

```
mongoexport -d test -c page_per_day_hits_tmp --type=csv -
f=_id,daily -o page_per_day_hits_tmp.csv

_id.d,_id.m,_id.y,_id.p,daily
3,2,2014,cart.do,115
4,2,2014,cart.do,681
5,2,2014,cart.do,638
6,2,2014,cart.do,610
....
3,2,2014,cart/error.do,2
4,2,2014,cart/error.do,14
5,2,2014,cart/error.do,23
```

Building your pipeline

CONVERSION

```
mongoexport -d test -c page_per_day_hits_tmp --type=csv -
f=_id,daily -o page_per_day_hits_tmp.csv
```

```
_id.d,_id.m,_id.y,_id.p,daily
3,2,2014,cart.do,115
4,2,2014,cart.do,681
5,2,2014,cart.do,638
6,2,2014,cart.do,610
....
3,2,2014,cart/error.do,2
4,2,2014,cart/error.do,14
5,2,2014,cart/error.do,23
```

Building your pipeline

```
mongoexport -d test -c page_per_day_hits_tmp --type=csv -
f=_id,daily -o page_per_day_hits_tmp.csv
```

```
_id.d,_id.m,_id.y,_id.p,daily
3,2,2014,cart.do,115
4,2,2014,cart.do,681
5,2,2014,cart.do,638
6,2,2014,cart.do,610
....
3,2,2014,cart/error.do,2
4,2,2014,cart/error.do,14
5,2,2014,cart/error.do,23
```

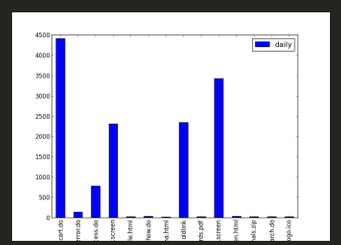
CSV FILE CONTENTS

Building your pipeline

```
mongoexport -d test -c page_per_day_hits_tmp --type=csv -
f=_id,daily -o page_per_day_hits_tmp.csv

_id.d,_id.m,_id.y,_id.p,daily
3,2,2014,cart.do,115
4,2,2014,cart.do,681
5,2,2014,cart.do,638
6,2,2014,cart.do,610
....
3,2,2014,cart/error.do,2
4,2,2014,cart/error.do,14
5,2,2014,cart/error.do,23
```

Visualising the results

```
In [1]: import pandas as pd
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
In [4]: df1 = pd.read_csv('page_per_day_hits_tmp.csv', names=['day', 'month',
'year', 'page', 'daily'], header=0)
Out[4]:
     day  month  year                page  daily
0      3      2  2014             cart.do    115
1      4      2  2014             cart.do    681
..   ...    ...   ...                 ...    ...
103   10      2  2014      stuff/logo.ico      3
[104 rows x 5 columns]

In [5]: grouped = df1.groupby(['page'])
Out[5]: <pandas.core.groupby.DataFrameGroupBy object at 0x10f6b0dd0>

In [6]: grouped.agg({'daily':'sum'}).plot(kind='bar')
Out[6]: <matplotlib.axes.AxesSubplot at 0x10f8f4d10>
```



Scikit-learn churn data

```
['State', 'Account Length', 'Area Code', 'Phone', "Int'l Plan", 'VMail Plan',
'VMail Message', 'Day Mins', 'Day Calls', 'Day Charge', 'Eve Mins', 'Eve
Calls', 'Eve Charge', 'Night Mins', 'Night Calls', 'Night Charge', 'Intl Mins',
'Intl Calls', 'Intl Charge', 'CustServ Calls', 'Churn?']
```

	State	Account Length	Area Code	Phone	Intl Plan	VMail Plan	\
0	KS	128	415	382-4657	no	yes	
1	OH	107	415	371-7191	no	yes	
2	NJ	137	415	358-1921	no	no	
3	OH	84	408	375-9999	yes	no	

	Night Charge	Intl Mins	Intl Calls	Intl Charge	CustServ Calls	Churn?
0	11.01	10.0	3	2.70	1	False.
1	11.45	13.7	3	3.70	1	False.
2	7.32	12.2	5	3.29	0	False.
3	8.86	6.6	7	1.78	2	False.

Scikit-learn churn example

```
from __future__ import division
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json

from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RF
%matplotlib inline
churn_df = pd.read_csv('churn.csv')
col_names = churn_df.columns.tolist()

print "Column names:"
print col_names

to_show = col_names[:6] + col_names[-6:]
```

Scikit-learn churn example

```
from __future__ import division
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json

from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RF
%matplotlib inline
churn_df = pd.read_csv('churn.csv')
col_names = churn_df.columns.tolist()

print "Column names:"
print col_names

to_show = col_names[:6] + col_names[-6:]
```

IMPORTS

Scikit-learn churn example

```
from __future__ import division
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json

from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RF
%matplotlib inline
churn_df = pd.read_csv('churn.csv')
col_names = churn_df.columns.tolist()

print "Column names:"
print col_names

to_show = col_names[:6] + col_names[-6:]
```

LOAD FILE / EXPLORE DATA

Scikit-learn churn example

```
from __future__ import division
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json

from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RF
%matplotlib inline
churn_df = pd.read_csv('churn.csv')
col_names = churn_df.columns.tolist()

print "Column names:"
print col_names

to_show = col_names[:6] + col_names[-6:]
```

Scikit-learn churn example

```
print "\nSample data:"
churn_df[to_show].head(2)
# Isolate target data
churn_result = churn_df['Churn?']
y = np.where(churn_result == 'True.',1,0)
to_drop = ['State','Area Code','Phone','Churn?']
churn_feat_space = churn_df.drop(to_drop,axis=1)
# 'yes'/'no' has to be converted to boolean values
# NumPy converts these from boolean to 1. and 0. later
yes_no_cols = ["Int'l Plan","VMail Plan"]
churn_feat_space[yes_no_cols] = churn_feat_space[yes_no_cols] == 'yes'

# Pull out features for future use
features = churn_feat_space.columns
X = churn_feat_space.as_matrix().astype(np.float)
scaler = StandardScaler()
X = scaler.fit_transform(X)
print "Feature space holds %d observations and %d features" % X.shape
print "Unique target labels:", np.unique(y)
```

Scikit-learn churn example

```
print "\nSample data:"
churn_df[to_show].head(2)
# Isolate target data
churn_result = churn_df['Churn?']
y = np.where(churn_result == 'True.',1,0)
to_drop = ['State','Area Code','Phone','Churn?']
churn_feat_space = churn_df.drop(to_drop,axis=1)
# 'yes'/'no' has to be converted to boolean values
# NumPy converts these from boolean to 1. and 0. later
yes_no_cols = ["Int'l Plan","VMail Plan"]
churn_feat_space[yes_no_cols] = churn_feat_space[yes_no_cols] == 'yes'
```

FORMAT DATA FOR USAGE

```
# Pull out features for future use
features = churn_feat_space.columns
X = churn_feat_space.as_matrix().astype(np.float)
scaler = StandardScaler()
X = scaler.fit_transform(X)
print "Feature space holds %d observations and %d features" % X.shape
print "Unique target labels:", np.unique(y)
```

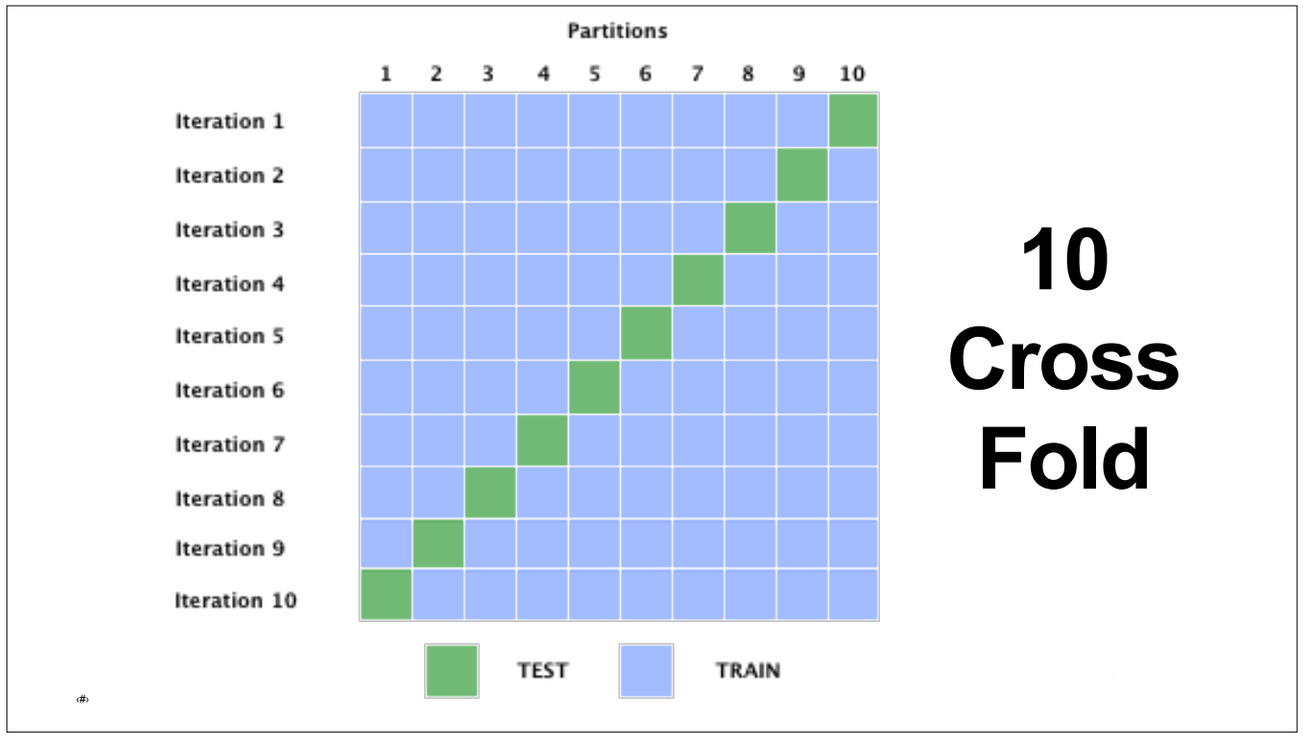
Scikit-learn churn example

```
print "\nSample data:"
churn_df[to_show].head(2)
# Isolate target data
churn_result = churn_df['Churn?']
y = np.where(churn_result == 'True.',1,0)
to_drop = ['State','Area Code','Phone','Churn?']
churn_feat_space = churn_df.drop(to_drop,axis=1)
# 'yes'/'no' has to be converted to boolean values
# NumPy converts these from boolean to 1. and 0. later
yes_no_cols = ["Int'l Plan","VMail Plan"]
churn_feat_space[yes_no_cols] = churn_feat_space[yes_no_cols] == 'yes'

# Pull out features for future use
features = churn_feat_space.columns
X = churn_feat_space.as_matrix().astype(np.float)
scaler = StandardScaler()
X = scaler.fit_transform(X)
print "Feature space holds %d observations and %d features" % X.shape
print "Unique target labels:", np.unique(y)
```

FORMAT DATA FOR USAGE

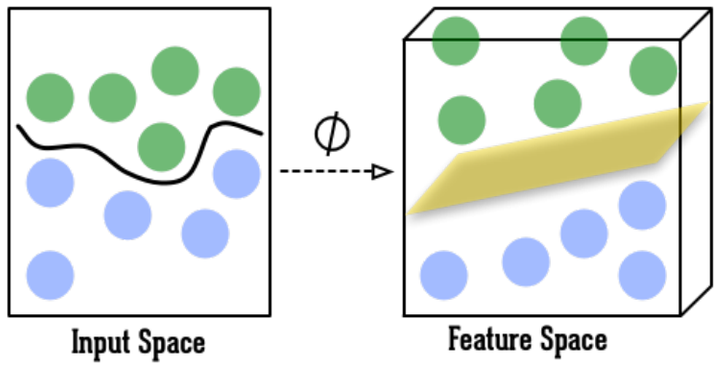
Scikit-learn churn example

```
print "\nSample data:"
churn_df[to_show].head(2)
# Isolate target data
churn_result = churn_df['Churn?']
y = np.where(churn_result == 'True.',1,0)
to_drop = ['State','Area Code','Phone','Churn?']
churn_feat_space = churn_df.drop(to_drop,axis=1)
# 'yes'/'no' has to be converted to boolean values
# NumPy converts these from boolean to 1. and 0. later
yes_no_cols = ["Int'l Plan","VMail Plan"]
churn_feat_space[yes_no_cols] = churn_feat_space[yes_no_cols] == 'yes'

# Pull out features for future use
features = churn_feat_space.columns
X = churn_feat_space.as_matrix().astype(np.float)
scaler = StandardScaler()
X = scaler.fit_transform(X)
print "Feature space holds %d observations and %d features" % X.shape
print "Unique target labels:", np.unique(y)
```



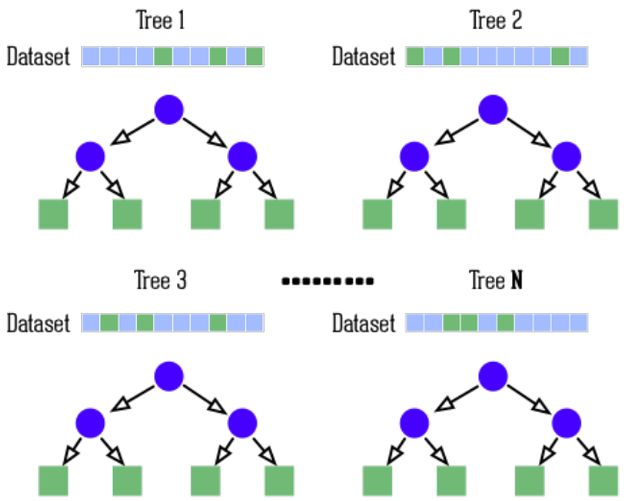
10 Cross Fold

Support Vector Machine



A list of number [a n-dimensional vector] and transform the points into higher dimensions so it is easier to separate them using a [n-1] dimensional hyperplane.

Random Forest



Traverse each tree and at each node in a tree:

i. Select **m** random predictor variables from available set
ii. Use the variable with best split [use objective function]
iii. Move to next node in tree

Scikit-learn churn example

```
from sklearn.svm import SVC
from sklearn.ensemble import
RandomForestClassifier as RF
from sklearn.metrics import
average_precision_score
from sklearn.cross_validation import
KFold

def accuracy(y_true,y_pred):
    # NumPy interpretes True and
False as 1. and 0.
    return np.mean(y_true == y_pred)

def run_cv(X,y,clf_class,**kwargs):
    # Construct a kfolds object
    kf =
KFold(len(y),n_folds=3,shuffle=True)
    y_pred = y.copy()

    # Iterate through folds
    for train_index, test_index in
kf:
        X_train, X_test =
X[train_index], X[test_index]
        y_train = y[train_index]
        clf = clf_class(**kwargs)
        clf.fit(X_train,y_train)
        y_pred[test_index] =
clf.predict(X_test)
    return y_pred

print "Support vector machines:"
print "%.3f" % accuracy(y,
run_cv(X,y,SVC))
print "Random forest:"
print "%.3f" % accuracy(y,
run_cv(X,y,RF))
```

Scikit-learn churn example

```
from sklearn.svm import SVC
from sklearn.ensemble import
RandomForestClassifier as RF
from sklearn.metrics import
average_precision_score
from sklearn.cross_validation import
KFold

def accuracy(y_true,y_pred):
    # NumPy interpretes True and
False as 1. and 0.
    return np.mean(y_true == y_pred)
```

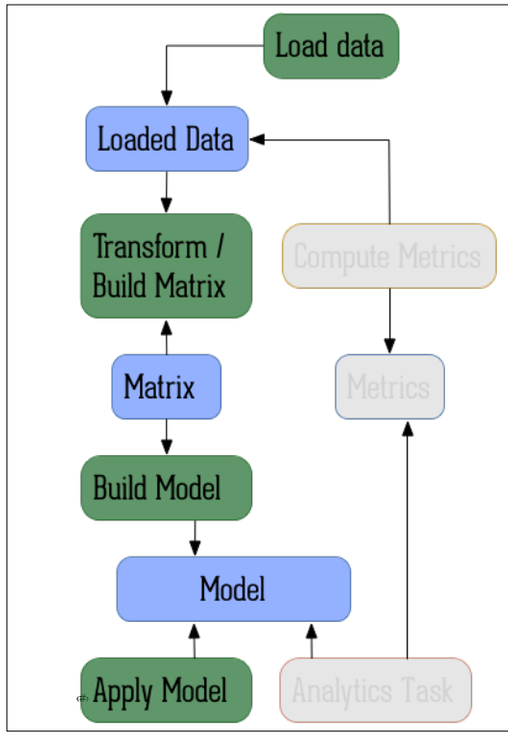
```
def run_cv(X,y,clf_class,**kwargs):
    # Construct a kfolds object
    kf =
KFold(len(y),n_folds=3,shuffle=True)
    y_pred = y.copy()

    # Iterate through folds
    for train_index, test_index in
kf:
        X_train, X_test =
X[train_index], X[test_index]
        y_train = y[train_index]
        clf = clf_class(**kwargs)
        clf.fit(X_train,y_train)
        y_pred[test_index] =
clf.predict(X_test)
    return y_pred
```

```
print "Support vector machines:"
print "%.3f" % accuracy(y,
run_cv(X,y,SVC))
print "Random forest:"
print "%.3f" % accuracy(y,
run_cv(X,y,RF))
```

Cross Fold
K=3

Scikit-learn churn example

```
from sklearn.svm import SVC
from sklearn.ensemble import
RandomForestClassifier as RF
from sklearn.metrics import
average_precision_score
from sklearn.cross_validation import
KFold

def accuracy(y_true,y_pred):
    # NumPy interpretes True and
False as 1. and 0.
    return np.mean(y_true == y_pred)

def run_cv(X,y,clf_class,**kwargs):
    # Construct a kfolds object
    kf =
KFold(len(y),n_folds=3,shuffle=True)
    y_pred = y.copy()

    # Iterate through folds
    for train_index, test_index in
kf:
        X_train, X_test =
X[train_index], X[test_index]
        y_train = y[train_index]
        clf = clf_class(**kwargs)
        clf.fit(X_train,y_train)
        y_pred[test_index] =
clf.predict(X_test)
    return y_pred

print "Support vector machines:"
print "%.3f" % accuracy(y,
run_cv(X,y,SVC))
print "Random forest:"
print "%.3f" % accuracy(y,
run_cv(X,y,RF))
```

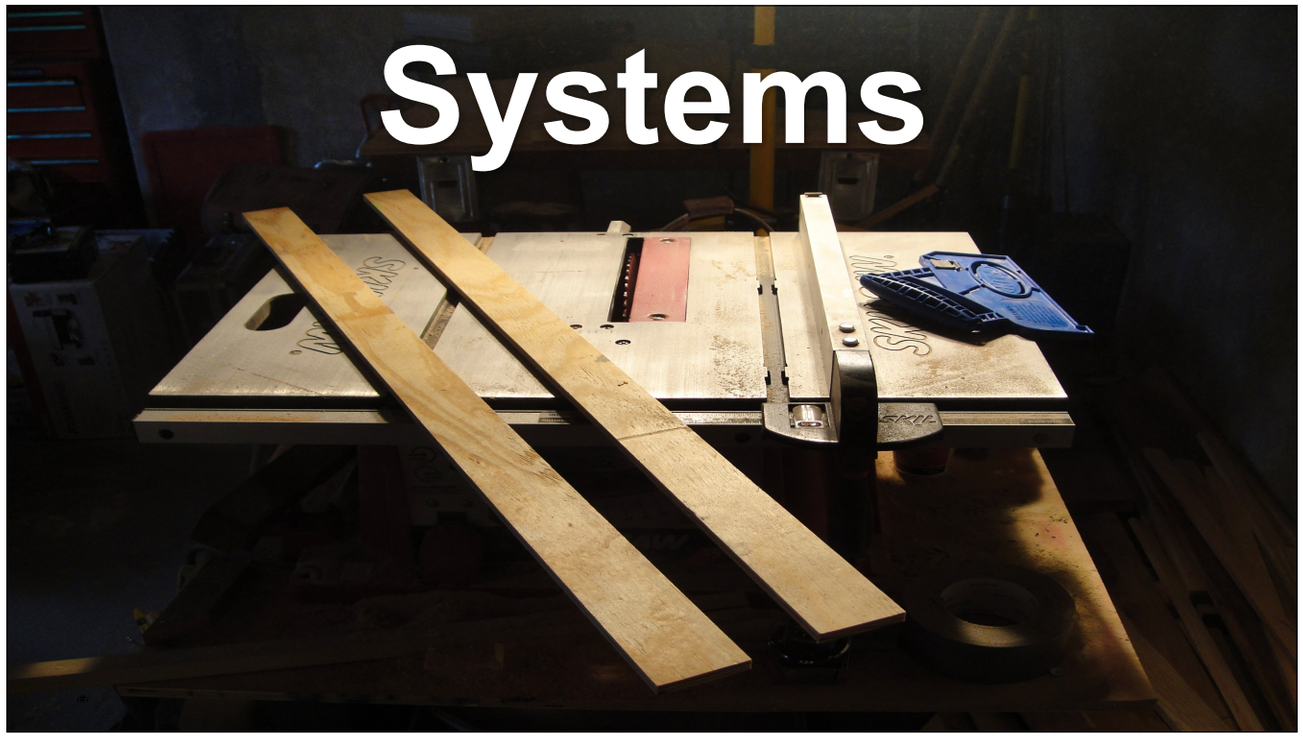


An example data pipeline:

- Data
- State
- Operations / Transformation

Reviewing our data factory concept, which is more akin to manufacturing. We looked a pipelines and the tools you could use to build them, I specifically focused on Airbnb's tool, Airflow. It allows you to author workflows as directed acyclic graphs (DAGs). We looked at using both the command line and at the UI. These allow you to monitor and troubleshoot the pipeline as well as triggering tasks.

A key differentiator is the fact that Airflow pipelines are defined as code (as opposed to a markup language in Oozie or Azkaban), and that tasks are instantiated dynamically (as opposed to creating tasks by deriving classes in Luigi). This makes Airflow the best solution out there for dynamic pipeline generation, which can be used to power concepts as "analytics as a service", "analysis automation" and computation frameworks, where pipelines are generated dynamically from configuration files or metadata of any form. Examples of that at Airbnb include our A/B testing framework, an anomaly detection framework, an aggregation framework and others.



We also looked at the systems which where the building blocks of any pipeline. We looked at Scikit-learn, PyMongo, and Monary. If you need to interact with data in MongoDB you should use PyMongo and Monary. The Scikit-learn is a great library for exploring your data and creating your models. You can use visualisation libraries like Matlibplot to graph and display the results.

Speed - In terms of operational considerations, speed is important and if you've got a large amount of data to transform or move between processing stages you should probably consider Monary for MongoDB. Speed of development is also another important consideration and that's why all of these systems and tools utilise Python to build out your data pipeline.

Photo Credits

flickr

https://www.flickr.com/photos/rcbodden/2725787927/in, Ray Bodden

https://www.flickr.com/photos/iqremix/15390466616/in, iqremix

https://www.flickr.com/photos/storem/129963685/in, storem

https://www.flickr.com/photos/diversey/15742075527/in, Tony Webster

https://www.flickr.com/photos/acwa/8291889208/in, PEO ACWA

https://www.flickr.com/photos/rowfoundation/8938333357/in, Rajita Majumdar

https://www.flickr.com/photos/54268887@N00/5057515604/in, Rob Pearce

https://www.flickr.com/photos/seeweb/6115445165/in, seeweb

https://www.flickr.com/photos/98640399@N08/9290143742/in, Barta IV

https://www.flickr.com/photos/aisforangie/6877291681/in, Angie Harms

https://www.flickr.com/photos/jakerome/3551143912/in, Jakerome

https://www.flickr.com/photos/ifyr/1106390483/, Jack Shainsky

https://www.flickr.com/photos/rioncm/4643792436/in, rioncm

https://www.flickr.com/photos/druidsnectar/4605414895/in, druidsnectar

Thanks!

Questions?

Eoin Brazil
eoin.brazil@mongodb.com