NIKE DTC RI

Basics of Airflow

The WHAT, WHY and HOW to use Airflows

~Pratik Saurav

The Problem

Companies are growing to have a complex network of processes having intricate dependencies

- Analytics ETL processes and batch processing are the critical items to be catered to
- Tons of time was spent writing jobs, monitoring and troubleshooting issues

The Classical Approach of writing batch jobs on your cluster and schedule them using a cron may not suffice.

WHY?

- Data lineage (the complete data cycle) might be opaque in nature
- Duplication of codes/logic
- Maintenance time grows exponentially as the ecosystem grows
- Signal to noise ratio may get out of control

The Genesis

Mu Sigma

Airbnb was outgrowing its workflows methodology

200k+Hive tables

> 10,000	> 6,000	> 6,000	> 1,500
Superset charts and dashboards	Experiments and metrics	Tableau workbooks and charts	Knowledge posts

The Genesis

Airbnb was outgrowing its workflows methodology

- They wanted to create a platform with dynamic pipeline generation, programmatic environment and a platform that people love working in.
- Hence they took it as an opportunity to build and share Airflow

Brethren of airflow?

- Facebook uses SWARM
- Yahoo uses (Used to) OOZIE
- Cron we are well aware of
- Ubisoft uses informatica based platforms

The Genesis

Airflow is an open source platform to programmatically author, schedule and monitor workflows.

Maxime Beauchem (Airbnb) created the whole thing. Wanting to create a platform which much wider scopes unlike the traditional job schedulers, Airbnb tried to identify the pros, cons and features from the best job schedulers in business.

Basic Features:

- Basic dependency management
- Status clarity
- Scheduling
- Easy access to logs and Notification features
- Job prioritization features and Parametrized retries
- Has great capabilities to scale out
- Supports complex dependency rules like branching, joining, subworkflows
- You can define your own constructs, callbacks
- Easy state altering like : DAG surgeries, ownership and versioning, data profiles, centralized connection management, Plugins (variety of plugins with connectivity to oracle, Hadoop, query engines etc.)

Mu Sigma

The Genesis

Airflow is an open source platform to programmatically author, schedule and monitor workflows.

Maxime Beauchem (Airbnb) created the whole thing. Wanting to create a platform which much wide r scopes unlike the traditional job schedulers, Airbnb tried to identify the pros, cons and features fro m the best job schedulers in business.

Basic Features:

- Basic dependency management
- Status clarity and scheduling
- Easy access to logs and Notification features
- Job prioritization features and Parametrized retries
- Has great capabilities to scale out and supports complex dependency rules like branching, joining , subworkflows
- You can define your own constructs, callbacks
- Easy state altering like : DAG surgeries, ownership and versioning, data profiles, centralized conne ction management

The ‘how to_’ guide

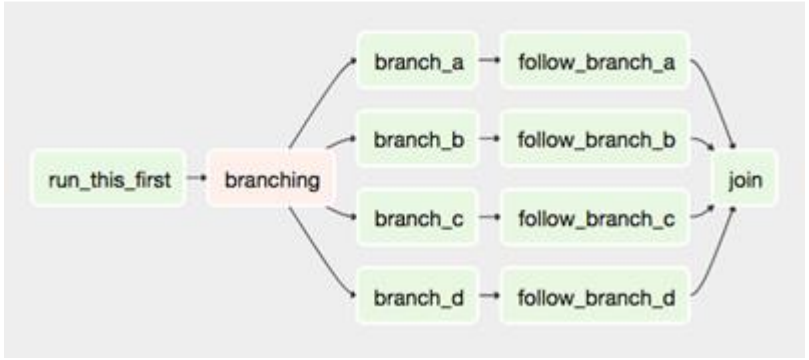
An airflow has four essential blocks for execution

- **DAG**: a description of the order in which work should take place
- **Operator**: a class that acts as a template for carrying out some work
- **Task**: a parameterized instance of an operator
- **Task Instance**: a task that
    - Has been assigned to a DAG and
    - Has a state associated with a specific run of the DAG

The ‘how to_’ guide

An airflow is designed as a Directed Acyclic Graph

- When authoring a workflow, you should think how it could be divided into tasks which can be executed independently.
- You can then merge these tasks into a logical whole by combining them into a graph.



- The shape of the graph decides the overall logic of our workflow.
- Directed Acyclic Graphs (DAGs) are trees of nodes that Airflow's workers will traverse. Each node in the graph can be thought of as a steps and the group of steps make up the overall job.

The ‘how to_’ guide

An airflow is designed as a Directed Acyclic Graph

Why are they Acyclic?

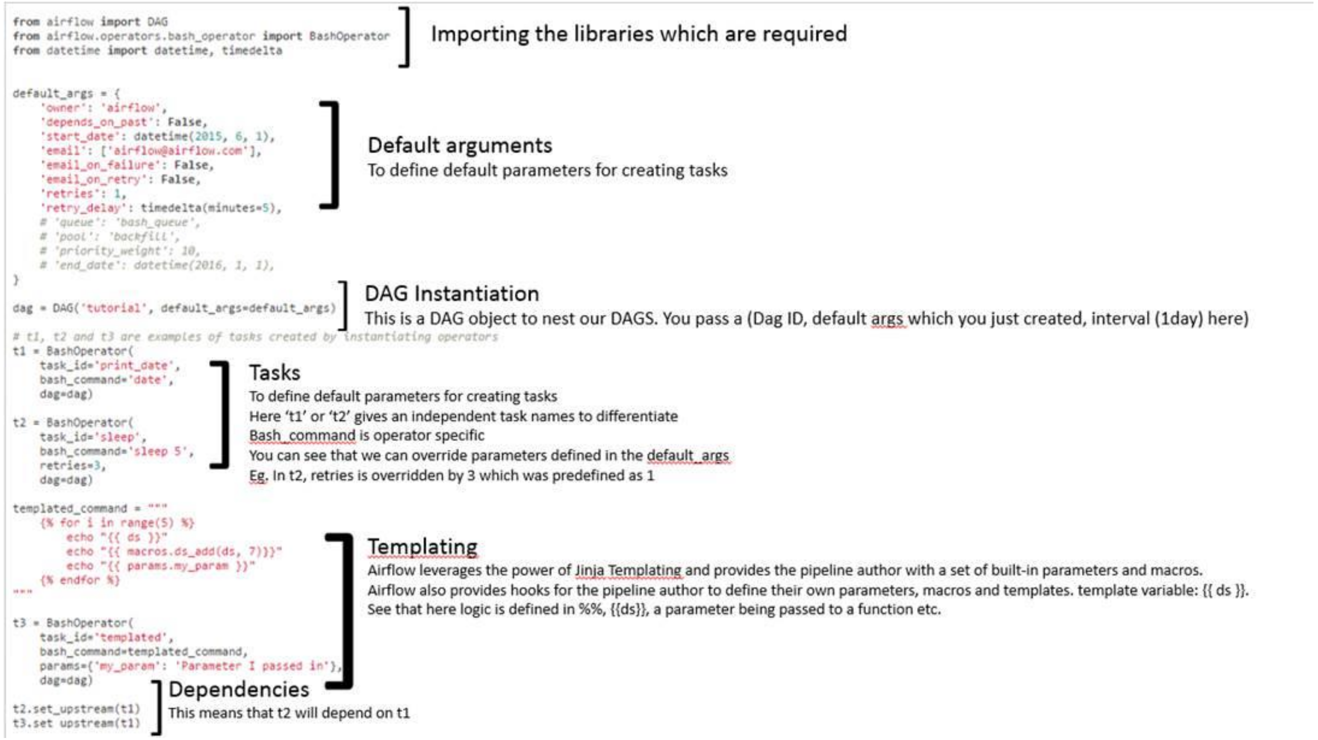
The graphs are acyclic so you cannot jump to a previous task in the graph but you can jump to another task that hasn't yet run in the current job.

This means individual steps can be retried repeatedly if they have a failure without having to restart the whole job.

What is a DAG?
In Airflow, a DAG – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies. DAGs are defined in standard Python files that are placed in Airflow's DAG_FOLDER .

Anatomy of a DAG

A DAG is defined by a DAG definition



```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta
```

Importing the libraries which are required

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@airflow.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}
```

Default arguments
To define default parameters for creating tasks

```
dag = DAG('tutorial', default_args=default_args)
```

DAG Instantiation
This is a DAG object to nest our DAGS. You pass a (Dag ID, default args which you just created, interval (1day) here)

```
# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)
```

Tasks
To define default parameters for creating tasks
Here 't1' or 't2' gives an independent task names to differentiate
Bash_command is operator specific
You can see that we can override parameters defined in the default_args
Eg. In t2, retries is overridden by 3 which was predefined as 1

```
templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)
```

Templating
Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros.
Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates. template variable: {{ ds }}.
See that here logic is defined in %%, {{ds}}, a parameter being passed to a function etc.

```
t2.set_upstream(t1)
t3.set upstream(t1)
```

Dependencies
This means that t2 will depend on t1

Anatomy of a DAG

A DAG is defined by a DAG definition

- A DAG definition is a python script wherein you define DAG’s structure as a code.
- Airflow will execute the code in each file to dynamically build the DAG objects.
- The point to be kept in mind that it’s actually just a configuration file you cannot do any data processing here at all.
- The scripts purpose is to define a DAG object. It needs to evaluate quickly (seconds, not minutes) since the scheduler will execute it periodically to reflect the changes if any

The following will come in handy while reading a DAG definition:

- list of operators
- Standard airflow operators
- Airflow tutorial

Creating a DAG

There are three main terminals in airflow

- The Airflow Terminal
- Celery Flower Terminal
- Airflow Scheduler TTY Shell

Creating a DAG

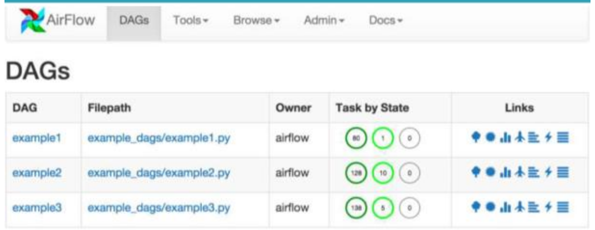
The Airflow terminal is the basic UI terminal

The Airflow Terminal

- It’s basically a GUI which has the list of all DAGs, their statuses, definitions which are very much intuitive
- Your front screen will appear pretty much like this. Where you’ll see the list of DAGs, their owners and statuses
- When you click on a DAG the airflow will open up it’s entire facebook profile in front of you XP (The name, status, profile picture(Acyclic Graphs), Events, Details, The codes associated) Everything!

Mu Sigma

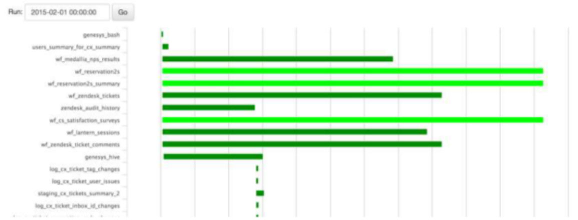
UI – DAG view



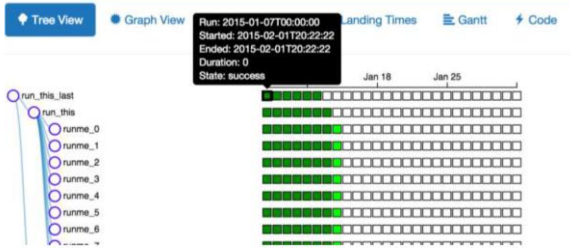
DAGs

DAG	Filepath	Owner	Task by State	Links
example1	example_dags/example1.py	airflow	80 1 0	
example2	example_dags/example2.py	airflow	128 10 0	
example3	example_dags/example3.py	airflow	138 5 0	

UI - Gantt

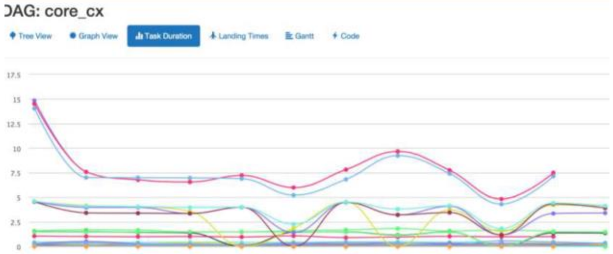

Run: 2015-02-01 00:00:00
Go
genesys_bash
users_summary_for_cx_summary
wf_medallia_nps_results
wf_reservation2s
wf_reservation2s_summary
wf_zendesk_tickets
zendesk_audit_history
wf_cs_satisfaction_surveys
wf_lantern_sessions
wf_zendesk_ticket_comments
genesys_hive
log_cx_ticket_tag_changes
log_cx_ticket_user_issues
staging_cx_tickets_summary_2
log_cx_ticket_inbox_id_changes

UI – Tree View

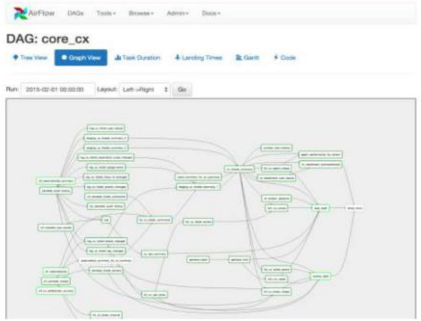


Run: 2015-01-07T00:00:00
Started: 2015-02-01T20:22:22
Ended: 2015-02-01T20:22:22
Duration: 0
State: success

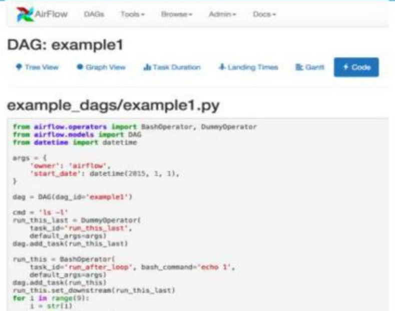
UI – Task duration



UI – Graph View



UI – Code

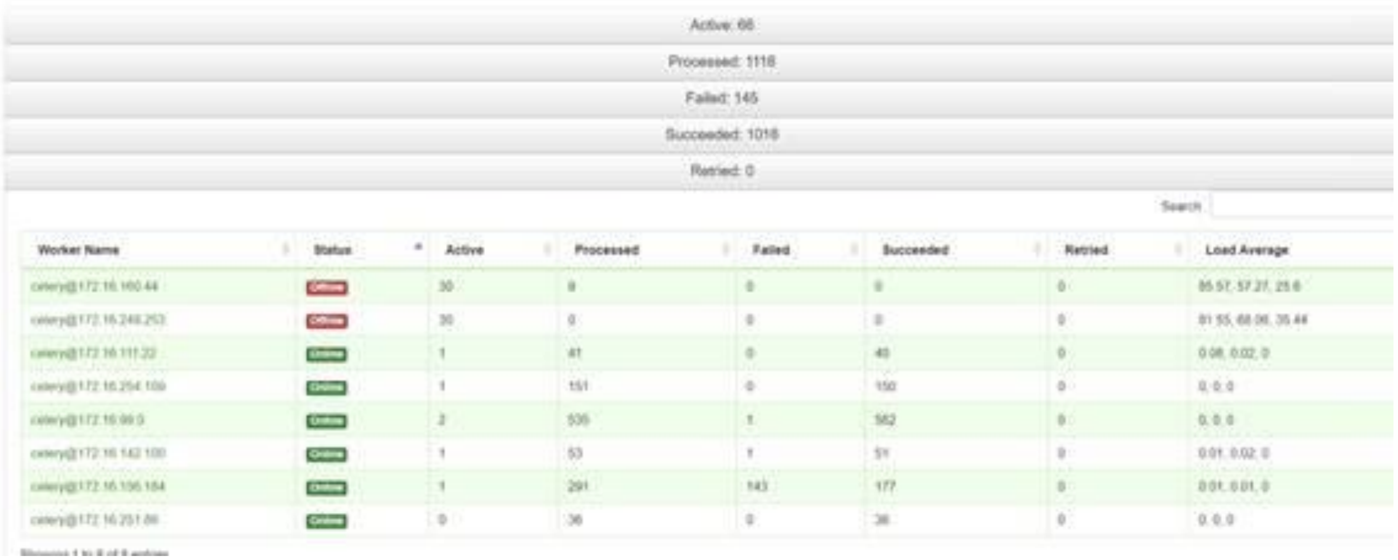


example_dags/example1.py

```
from airflow.operators import BashOperator, DummyOperator
from airflow.models import DAG
from datetime import datetime

args = {
    'owner': 'airflow',
    'start_date': datetime(2015, 1, 1),
}

dag = DAG(dag_id='example1')

cmd = 'ls -l'
run_this_last = DummyOperator(
    task_id='run_this_last',
    default_args=args)
dag.add_task(run_this_last)

run_this = BashOperator(
    task_id='run_after_loop', bash_command='echo 1',
    default_args=args)
dag.add_task(run_this)
run_this.set_downstream(run_this_last)
for i in range(9):
    i = str(i)
```

Creating a DAG

The flower terminal is used for performance monitoring of workers



Active: 68

Processed: 1118

Failed: 145

Succeeded: 1018

Retried: 0

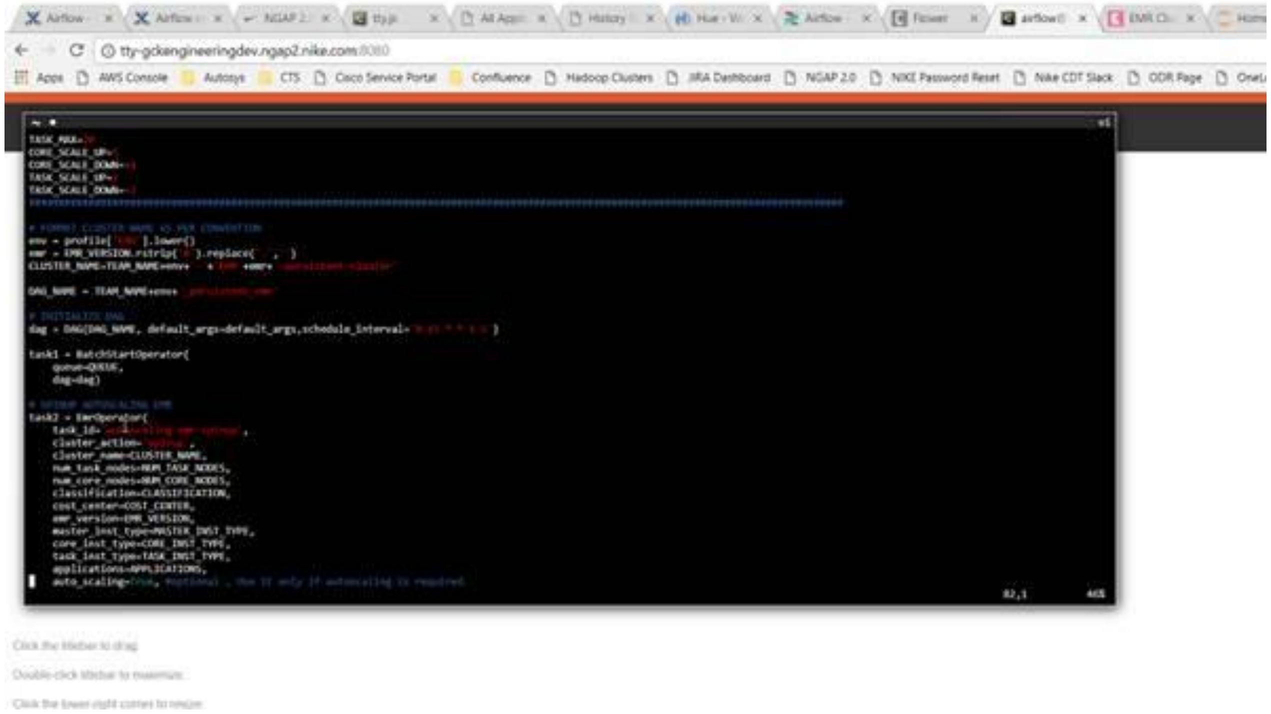
Search

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@172.16.160.44	Offline	30	9	0	0	0	85.57, 57.27, 25.6
celery@172.16.248.253	Offline	30	0	0	0	0	81.55, 68.06, 35.44
celery@172.16.111.22	Online	1	41	0	40	0	0.08, 0.02, 0
celery@172.16.254.109	Online	1	151	0	150	0	0, 0, 0
celery@172.16.99.0	Online	2	535	1	562	0	0, 0, 0
celery@172.16.142.100	Online	1	53	1	51	0	0.01, 0.02, 0
celery@172.16.196.184	Online	1	291	143	177	0	0.01, 0.01, 0
celery@172.16.251.86	Online	0	36	0	36	0	0, 0, 0

Showing 1 to 8 of 8 workers

Creating a DAG

The scheduler tty terminal is used for deploying jobs to the workers



tty-gckengineeringdev.ngap2.nike.com:8080

```
TASK_MAX=[illegible]
CORE_SCALE_UP=[illegible]
CORE_SCALE_DOWN=[illegible]
TASK_SCALE_UP=[illegible]
TASK_SCALE_DOWN=[illegible]
##########################################################################################

# FORMAT CLUSTER NAME AS PER CONVENTION
env = profile[ [illegible] ].lower()
emr = EMR_VERSION.rstrip( [illegible] ).replace( [illegible] , [illegible] )
CLUSTER_NAME=TEAM_NAME+env+ [illegible] + [illegible] +emr+ [illegible]

DAG_NAME = TEAM_NAME+env+ [illegible]

# INITIALIZE DAG
dag = DAG(DAG_NAME, default_args=default_args,schedule_interval= [illegible] )

task1 = BatchStartOperator(
    queue=QUEUE,
    dag=dag)

# SETUP AUTOSCALING EMR
task2 = EmrOperator(
    task_id= [illegible] ,
    cluster_action= [illegible] ,
    cluster_name=CLUSTER_NAME,
    num_task_nodes=NUM_TASK_NODES,
    num_core_nodes=NUM_CORE_NODES,
    classification=CLASSIFICATION,
    cost_center=COST_CENTER,
    emr_version=EMR_VERSION,
    master_inst_type=MASTER_INST_TYPE,
    core_inst_type=CORE_INST_TYPE,
    task_inst_type=TASK_INST_TYPE,
    applications=APPLICATIONS,
    auto_scaling=True, #optional , Use it only if autoscaling is required
```