EECS E6893 Big Data Analytics
HW4: Data Analytics Pipeline

Yunhang Lin, yl4860@columbia.edu
Workflow

- A sequence of tasks involved in moving from the beginning to the end of a working process
- Started on a schedule or triggered by an event
Workflow

- Cook a pizza
Workflow

- Data analytics

Ingest → Transform → Analyze → Utilize
● A platform let you create, schedule, monitor and manage workflows

Principles:
● Scalable
● Dynamic
● Extensible
● Elegant

Features:
● Pure Python
● Useful UI
● Robust Integrations
● Easy to Use
● Open Source
DAG (Directed Acyclic Graph)

- In Airflow, workflows are created using DAGs
- A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies
- The tasks describe what to do, e.g., fetching data, running analysis, triggering other systems, or more
- A DAG ensures that each task is executed at the right time, in the right order, or with the right issue handling
- A DAG is written in Python
Airflow architecture

- **Scheduler**: handles both triggering scheduled workflows, and submitting tasks to the executor to run.
- **Executor**: handles running tasks.
- **Webserver**: a handy user interface to inspect, trigger and debug the behavior of DAGs and tasks.
- **A folder of DAG files**: read by the scheduler and executor
- **A metadata database**: used by the scheduler, executor and webserver to store state
Airflow installation
Three choices

1. Install and use Airflow in the VM of GCP
2. Install and use airflow in your local machines
3. Google composer
Set up the firewall

- VPC network → Firewall → Create Filewall rule
- Set service account scope and protocols and ports
Create a VM instance

VM Instances

Compute Engine lets you VM instances that run on Google's infrastructure. Create micro-VMs or larger instances running Debian, Windows, or other standard images. Create your first VM instance, import it using a migration service, or try the quickstart to build a sample app.
Create a VM instance

**Boot disk**

<table>
<thead>
<tr>
<th>Type</th>
<th>New balanced persistent disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>50 GB</td>
</tr>
<tr>
<td>Image</td>
<td>Ubuntu 18.04 LTS</td>
</tr>
</tbody>
</table>

**Identity and API access**

- Service accounts
  - Compute Engine default service account

**Access scopes**

- Allow default access
- Allow full access to all Cloud APIs
- Set access for each API

**Firewall**

- Add tags and firewall rules to allow specific network traffic from the Internet
  - Allow HTTP traffic
  - Allow HTTPS traffic
Connect to your VM using SSH
Connect to your VM using SSH

The programs included with the Ubuntu system are free software; the exact distribution terms for each program are described in the individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.

ch3212@hw4:~$
Install and update packages

1. sudo apt update
2. sudo apt -y upgrade
3. sudo apt-get install wget
4. sudo apt install -y python3-pip
Download miniconda and create a virtual environment

1. mkdir -p ~/miniconda3
2. wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh
3. bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
4. rm -rf ~/miniconda3/miniconda.sh
5. ~/miniconda3/bin/conda init bash
6. ~/miniconda3/bin/conda init zsh

# reopen (or we say reconnect) your terminal and create a new environment
7. conda create --name airflow python=3.8

# activate the environment (everytime you open a new terminal, you should run this)
8. conda activate airflow

# optional but in case you don’t like warnings
9. (optional) pip install virtualenv
10. (optional) pip install kubernetes
Install Airflow

# Airflow needs a home. `~/airflow` is the default, but you can put it
# somewhere else if you prefer (optional)
export AIRFLOW_HOME=~/.airflow

# Install Airflow using the constraints file
AIRFLOW_VERSION=2.2.1
PYTHON_VERSION=3.8

# For example: 3.8
CONTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-2.2.1/constraints-3.8.txt"

# run airflow version to check if you install it successfully
airflow version

# The Standalone command will initialise the database, make a user,
# and start all components for you.
# airflow standalone

# Visit localhost:8080 in the browser and use the admin account details
# shown on the terminal to login.
# Enable the example_bash_operator dag in the home page
Initialize the database, make a user, and start webserver

# Initialize the database, after this you will see a new folder airflow in your
# $AIRFLOW_HOME which contains configuration file airflow.cfg
1. airflow db init
2. airflow users create \
   --username yunhang \
   --password 123456 \
   --firstname yunhang \
   --lastname lin \
   --role Admin \ 
   --email yl4860@columbia.edu
3. airflow webserver --port 8080
Airflow UI on your web browser

- Open your browser
- login
Start scheduler

# Open a new terminal (you can use screen if you prefer to open only one
# terminal)

1. conda activate airflow
2. airflow db init
3. airflow scheduler
Airflow examples
# Download helloworld.py from Coursework/Files
# Open a new terminal
conda activate airflow
# Create dags folders
cd airflow
mkdir dags
cd dags
# Upload helloworld.py here
# Check if the script is correct, no errors if it’s correct
python helloworld.py
# Initialize db again and you will see “helloworld” on the website
airflow db init
Helloworld

Tree

Graph
Two ways to trigger a DAG

1. Trigger manually
2. Trigger on a schedule
Scheduler

- The scheduler won't trigger your tasks until the period it covers has ended.
- The scheduler runs your job one schedule_interval after the start date, at the end of the interval.

References


https://cloud.google.com/composer/docs/trIGGERing-dags
Tasks

```python
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:

    # 4 examples of tasks created by instantiating operators
t1 = PythonOperator(
    task_id='t1',
    python_callable=correct_sleeping_function,
)
t2_1 = PythonOperator(
    task_id='t2_1',
    python_callable=correct_sleeping_function,
)
t2_2 = PythonOperator(
    task_id='t2_2',
    python_callable=correct_sleeping_function,
    retries=3,
)
t2_3 = PythonOperator(
    task_id='t2_3',
    python_callable=correct_sleeping_function,
)
t3_1 = PythonOperator(
    task_id='t3_1',
    python_callable=correct_sleeping_function,
)
t3_2 = PythonOperator(
    task_id='t3_2',
    python_callable=correct_sleeping_function,
)
t4_1 = BashOperator(
    task_id='t4_1',
    bash_command='sleep 2',
    retries=3,
)
```

def correct_sleeping_function():
    """This is a function that will run within the DAG execution""
    time.sleep(2)
Operators

1. **PythonOperator**
2. **BashOperator**
3. branch_operator
4. email_operator
5. mysql_operator
6. DataprocOperator

... 

... 

**PythonOperator:**

```python
def function():
    print(123)

task = PythonOperator(
    task_id='task_id',
    python_callable=function,
)
```

**BashOperator:**

```bash
task = BashOperator(
    task_id='task_id',
    bash_command='sleep 2',
)
```

# other examples
bash_command='python python_code.py'
bash_command='bash bash_code.sh '

(must have a space to satisfy Jinja template !!!!)
Dependencies

# task dependencies

\[
\begin{align*}
& t1 >> [t2_1, t2_2, t2_3] \\
& t2_1 >> t3_1 \\
& t2_2 >> t3_2 \\
& [t2_3, t3_1, t3_2] >> t4_1
\end{align*}
\]

# t2_1 will depend on t1 running successfully to run. The following ways # are equivalent:

\[
\begin{align*}
& t1 >> t2_1 \\
& t1 << t2_1 \\
& t1.set_downstream(t2_1) \\
& t2_1.set_upstream(t1)
\end{align*}
\]

# you can write in a chain

\[
\begin{align*}
& t1 >> t2_1 >> t3_1 >> t4_1
\end{align*}
\]

28
Trigger the dag

```
schedule_interval=timedelta(days=1),
start_date=datetime(2021, 1, 1),
```

Start scheduling the DAG
Example 2

t1 = PythonOperator(
    task_id='t1',
    python_callable=count_function,
)

t2_1 = PythonOperator(
    task_id='t2_1',
    python_callable=wrong_sleeping_function,
    retries=3,
)

count = 0
def count_function():
    global count
    count += 1
    print('count increase output: {}\n'.format(count))
    time.sleep(2)

def wrong_sleeping_function():
    global count
    print('wrong sleeping function output: {}\n'.format(count))
    assert count == 1
    time.sleep(2)
Example 2

[Diagram showing a Directed Acyclic Graph (DAG) with tasks t1, t2_1, t2_2, t3_1, t3_2, t2_3, and t4_1, illustrating the workflow with arrows indicating dependencies and task statuses such as queued, running, success, failed, up_for_retry, up_for_reschedule, and upstream_failed.]

[Time series graph showing the timeline from 11:42:15 to 11:43:45 with tasks t1, t2_1, t2_2, t3_1, t3_2, t2_3, and t4_1, with vertical bars indicating task execution times.]
Example 2

```python
count = 0
def count_function():
    # this task is t1
    global count
    count += 1
    print('count_increase output: {}'.format(count))
time.sleep(2)

def wrong_sleeping_function():
    # this task is t2_1, t1 >> t2_1
    global count
    print('wrong sleeping function output: {}'.format(count))
    assert count == 1
time.sleep(2)
```

```
{logging_mixin.py:109} INFO - count_function output: 1
...

{logging_mixin.py:109} INFO - wrong_sleeping_function output: 0
...

assert count == 1

AssertionError
```
Why?

- Airflow Python script is just a configuration file specifying the DAG's structure as code.
- Different tasks run on different workers at different points in time
- **Script cannot be used to cross communicate between tasks (Xcoms can)**
Why sequential?
Executors

- SequentialExecutor
- LocalExecutor
- CeleryExecutor
- KubernetesExecutor
Change SequentialExecutor to LocalExecutor

- Postgresql
- Modify the configuration in ~/airflow/airflow.cfg

```python
# Default timezone in case supplied date times are naive
# can be utc (default), system, or any IANA timezone string (e.g. Europe/Amsterdam)
default_timezone = utc

# The executor class that airflow should use. Choices include
# ```SequentialExecutor```, ```LocalExecutor```, ```CeleryExecutor```, ```DaskExecutor```,
# ```KubernetesExecutor```, ```CeleryKubernetesExecutor``` or the
# full import path to the class when using a custom executor.
executor = SequentialExecutor

# The SqlAlchemy connection string to the metadata database.
# SqlAlchemy supports many different database engines.
# More information here:
sql_alchemy_conn = sqlite:///home/ch3212/airflow/airflow.db
```
1. Install Postgres and Configure your Postgres user:

# In your terminal
sudo apt-get install postgresql postgresql-contrib
sudo -u postgres psql

# Next enter:
ALTER USER postgres PASSWORD 'postgres'; q

# In your terminal
sudo apt-get install libpq-dev
pip install 'apache-airflow[postgres]'
LocalExecutor

2. Change your SQL Alchemy Conn inside airflow.cfg:

# In your terminal
vim ~/airflow/airflow.cfg
# you will see this line
sql_alchemy_conn = sqlite:///.../airflow.db
# change it to:
sql_alchemy_conn = postgresql+psycopg2://postgres:postgres@localhost/postgres
# Also change
Executor = SequentialExecutor
# To
Executor = LocalExecutor
LocalExecutor

3. Check for DB connection:
# In your terminal
airflow db check
# If Airflow could successfully connect to yours Postgres DB, you will see an INFO
# containing a “Connection Successful” message in it, so now we are good to go.

4. Close the webserver and scheduler; Init your Airflow DB:
# In your terminal
airflow db init

5. Create new user and restart the webserver and scheduler (the original
username is reset since we change the database)
as shown in page 18 and 19; and login on the web browser
LocalExecutor

DAG: helloworld - A simple toy DAG

- t1 → t2_1 → t3_1 → t4_1
- t2_2 → t3_2

Schedule: 1 day, 0:00:00
Next Run: 2021-11-15, 21:28:04
Take home

- DAG
- Scheduler
- Executor
- Database
- Operator

- Cross communication between tasks
- Schedule a job !! start data and schedule interval
Some potential error

1. airflow.exceptions.AirflowException: The webserver is already running under PID 20243.

Using the Command line

```
sudo lsof -i tcp:8080
```

```
使用权    PID   USER      FD  TYPE DEVICE SIZE/OFF NODE NAME
```

```
gunicorn: 28944 yl4860 5u   IPv4 106090 0t0  TCP *:http-alt (LISTEN)
[ready]   28946 yl4860 5u   IPv4 106090 0t0  TCP *:http-alt (LISTEN)
[ready]   28947 yl4860 5u   IPv4 106090 0t0  TCP *:http-alt (LISTEN)
[ready]   28948 yl4860 5u   IPv4 106090 0t0  TCP *:http-alt (LISTEN)
[ready]   28949 yl4860 5u   IPv4 106090 0t0  TCP *:http-alt (LISTEN)
```

Then, Kill all related PID

Sudo kill -9 xxxx

Or,

using the Command line

```
killall -9 airflow
```
Homework
Three tasks

● Helloworld
● Build workflows
● Written parts
Task 1 Helloworld

Q1.1 Install Airflow (20 pts)

Q1.2 Run helloworld (15 pts)

  ○ SequentialExecutor (5 pts)
  ○ LocalExecutor (5 pts)
  ○ Explore other features and visualizations you can find in the Airflow UI. Choose two features/visualizations to explain their functions and how they help monitor and troubleshoot the pipeline, use helloworld as an example. (5 pts)
Task 2  Build workflows

Q2.1 Implement this DAG (25 pts)

- Tasks and dependencies (10 pts)
- Manually trigger it (10 pts)
- Schedule the first run immediately and running the program every 30 minutes (5 pts)
Task 2  Build workflows

Q2.2 Stock price fetching, prediction, and storage every day (25 pts)

- Schedule fetching the stock price of [AAPL, GOOGL, FB, MSFT, AMZN] at 7:00 AM everyday.
- Preprocess data if you think necessary
- Train/update 5 linear regression models for stock price prediction for these 5 corporates. Each linear model takes the “open price”, “high price”, “low price”, “close price”, “volume” of the corporate in the past ten days as the features and predicts the “high price” for the next day.
- Everyday if you get new data, calculate the relative errors, i.e., (prediction yesterday - actual price today) / actual price today, and update the date today and 5 errors into a table, e.g., a csv file.

https://pypi.org/project/yfinance/
Task 3  Written parts

Q3.1 Answer the question (5 pts)
   ○ What are the pros and cons of SequentialExecutor, LocalExecutor, CeleryExecutor, KubernetesExecutor?

Q3.2 Draw the DAG of your group project (10 pts)
   ○ Formulate it into at least 5 tasks
   ○ Task names (functions) and their dependencies
   ○ How do you schedule your tasks?
References


https://cloud.google.com/composer/docs