EECS E6893 Big Data Analytics
HW2: Streaming Big Data Analytics & Data Analytics Pipeline
TUTORIAL 1

Ajinkeya Chitrey
ac5166@columbia.edu
Agenda

- Streaming Analytics on Finnhub stock data
- Data pipeline orchestration using Apache Airflow
STREAMING ANALYTICS
Spark Streaming

Streaming Analytics Setup

● Step 1: Create your account using your Columbia email ID on Finnhub
Streaming Analytics Setup

- Step 2: Login and Save your API Key (You will use your own unique API key in the following exercise)
Streaming Analytics Setup

- Step 3: Pull data from Finnhub
Streaming Analytics Setup

- Step 4: Create your GCP Dataproc Cluster as done for previous assignments and setup the Finnhub api

```python
In [33]: !pip install finnhub-python

import finnhub
finnhub_client = finnhub.Client(api_key="YOUR UNIQUE API KEY")
print(finnhub_client.stock_candles('STOCK NAME', 'DATA RESOLUTION', start_unix_timestamp, end_unix_timestamp))
```
TO DO
1. Pull the stock data of “AAPL” at **1-minute level resolution every 5 minutes**. Each such pull made once every 5 minutes should have data from the (current timestamp - one hour) to the current timestamp when you hit the api i.e. **one hour's worth of data in each pull**. Let this entire process run for **30 minutes**. At the end of your program, your code should have generated **~1.5 hours** worth of data for the stock and the api should have been called 7 times (once every 5 minutes, over a period of 30 minutes).
TO DO

2. In each data pull, the data should be **incrementally** loaded into a Spark data frame where the data frame schema will be `['Stock Name', "UTC Timestamp", "c", "l", "h", "o", "v"]`

Sample flow of incremental update operation:
3. Every 5 minutes, after inserting data into the dataframe, compute the **30-minute moving averages** for the stock's "c", "l", "h", "o", and "v" values. Store these moving averages **incrementally** in a separate PySpark dataframe with the schema ["Datetime", "c_MA", "l_MA", "h_MA", "o_MA", "v_MA"], where “Datetime” represents the end timestamp of the moving average window.

**Moving Averages for the data called till 2023-10-02 22:42:00**

**Moving Averages for all the data present after next api call at 2023-10-02 22:46:00**
To Submit:
- PDF with screenshots of your code, brief explanation of the code workflow and the results i.e. dataframe holding the streamed data and the dataframe holding the moving averages.
- Code file for Streaming Analytics section

Important Notes:
- The data should be pulled anytime between 12pm – 4pm on Monday – Friday since that is the only time you will get real-time stock data when the markets are operational so plan your assignment timeline well. You will most probably not be able to pull data outside these timings accurately and on weekends.
- You may not get the data for each minute level timestep, there may be a few data points missing here and there which is fine but ensure that the majority of the expected timestamps are present in each pull.
AIRFLOW DATA PIPELINING
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
● A platform that lets 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
Workflow

- Cook a pizza
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 Firewall rule
- Set service account scope and protocols and ports
Create a VM instance

VM Instances

Compute Engine lets you create virtual machines 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 INSTANCE  TAKE THE QUICKSTART
Create a VM instance
Create a VM instance

**Boot disk**

<table>
<thead>
<tr>
<th>Name</th>
<th>test-airflow-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>New balanced persistent disk</td>
</tr>
<tr>
<td>Size</td>
<td>50 GB</td>
</tr>
<tr>
<td>License type</td>
<td>Free</td>
</tr>
<tr>
<td>Image</td>
<td>Ubuntu 20.04 LTS</td>
</tr>
</tbody>
</table>

**Identity and API access**

- **Service account**
  - Compute Engine default service account

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

**Firewall**

- 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`  
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
CONSTRANIT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt"
# For example: https://raw.githubusercontent.com/apache/airflow/constraints2.2.1/constraints-3.6.txt
pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRANIT_URL}"
# 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 (enter your own details), 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
Helloworld

# 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 after refreshing it
airflow db init
Helloworld

Tree

Graph
Two ways to trigger a DAG

1. Trigger manually
2. Trigger on a schedule

Trigger manually
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(
    'hello-world',
    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:

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

t1 >> t2_1

t1 << t2_1

t1.set_downstream(t2_1)

t2_1.set_upstream(t1)

# you can write in a chain

t1 >> t2_1 >> t3_1 >> t4_1
Trigger the dag

Start scheduling the DAG

(schedule_interval=timedelta(days=1),
start_date=datetime(2021, 1, 1),)
**Example 2**

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

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

```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)
```
Example 2
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
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
   ```

   ![Command output screenshot](image)

   Then, Kill all related PID
   
   ```
   Sudo kill -9 xxxx
   ```

   Or, using the Command line
   
   ```
   killall -9 airflow
   ```
Homework
Three tasks

- Helloworld
- Build sample workflow
- Design your own workflow for Yahoo Finance data
Task 1  Helloworld

Q1.1 Install Airflow (20 pts)

Q1.2 Run helloworld (15 pts)

- SequentialExecutor
- 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.
Task 2  Build workflows

Q2.1 Implement this DAG (20 pts)

○ Tasks and dependencies (5 pts)
○ Manually trigger it (10 pts)
○ Schedule the first run immediately and then running the program every 30 minutes (5 pts)
Task 3  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].
- 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 current day as the feature and predicts the “high price” for the next day
- Calculate the relative errors for the last 5 days, i.e., (prediction made from yesterday’s data for today - actual price today) / actual price today, and update the prediction date and 5 errors into a table, e.g., a csv file.

```python
import yfinance as yf
ticker = 'AAPL'
aapl = yf.Ticker(ticker)
hist = aapl.history(period='max')
print(type(hist))
print(hist.shape)
print(hist)
```

https://pypi.org/project/yfinance/
Pointers for Q2.2: (You have to think how to create the different tasks and the overall Airflow execution DAG based on the question requirements, below are pointers that may give you some ideas)

- Pull historical data for each corporate till the current date and store data for each in a csv. (Set period in history to “max” of ticker(company.tag).history(period=’max’))

- Use this csv to incrementally train the linear regression models for each corporate’s data. The dataset for each model X and y should be created such that each X(d) = [“open price”, “high price”, “low price”, “close price”, “volume”] of date d and the corresponding y = “High price” for date d+1 as stated in the question.

- Assume current date (the date till which you have pulled the data) is d_current.
  For each corporate,
  o Train/Update a model using data from [X_y] till (d_current – i) days and use this model to predict the y for the (d_current – (i-1)) day. Repeat the training and prediction for i = 5, 4, 3, 2, 1 (make predictions for last 5 days and calculate relative errors for each prediction).
  o For each value of i keep storing the relative errors in a csv for each model for each corporate on each of the 5 testing days.
  o The final errors csv should look something like below:

    | Date | Apple | Google | Meta | Microsoft | Amazon |
    |------|-------|--------|------|-----------|--------|
    | 0    | 11/26/2022 | -0.00501 | -0.0083 | -0.00167 | 0.005777 |
    | 1    | 11/27/2022 | -0.00991 | -0.00755 | -0.00997 | -0.00698 | 0.00609 |
    | 2    | 11/28/2022 | 0.026959 | 0.007073 | 0.005526 | 0.003579 | 0.008291 |
    | 3    | 11/29/2022 | 0.019557 | 0.0088 | 0.008127 | 0.011455 | -0.0204 |
    | 4    | 11/30/2022 | 0.009074 | 0.010989 | -0.00189 | 0.000902 | 0.14896 |

To give an example: the value in cell C2 represents the relative error produced by the linear regression model trained on Apple data till 11/25/2022 and the prediction made for 11/26/2022. Similarly, the value in cell C3 represents the relative error produced by the updated linear regression model trained on Apple data till 11/26/2022 and the prediction made for 11/27/2022. The value in cell F6 represents the relative error produced by the linear regression model trained on Microsoft data till 11/29/2022 and the prediction made for 11/30/2022.
References


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