Problem Statement: Explore which federal government agencies have the highest web traffic and which have low traffic and how this varies over time. The federal government provides a vast array of services and information to the American people through its websites. Websites that have little traffic may be poorly designed or not well-publicized. When websites have very high traffic, this may imply that the government must be prepared with resources to meet the high demand for both web traffic and downstream service delivery. In this project, I build a data pipeline, ingesting data in batches from an API weekly, transforming the data with Spark, storing the data in a mySQL database, and visualizing the output with python's altair package. I orchestrate the workflow with Apache Airflow.

High Level Overview of Steps

1. Install required software
2. Use python to extract historical website visits data from the analyicts.usa.gov API (Jan 1, 2020 through May 1, 2023) and store the raw data in a mySQL database.
3. Setup directed acyclic graph in Apache Airflow to do the following steps **once a week:**
    a. Task 1: Extract the latest website visits data
    b. Task 2: Transform and store the data so it is ready for analysis.
    c. Task 3: Load transformed data into mysql into python. Visualize using the python altair package.

Big Dataset Source: analytics.usa.gov API
Hardware: MacOS Monterey Version 12, M1 chip, 16GB
Software: Python 3.9, Spark 3.3.1, Apache Airflow 2.6.0

Benefits: Airflow has a user-friendly visual display of the pipeline.

Challenges: I encountered a few hiccups as I was new to the airflow technology (1) airflow uses the UTC timezone for scheduling jobs (2) if you need to update the configuration file, afterwards, be sure to reset the airflow database, webserver, and scheduler. (3) Your scheduler's first run is sensitive to the start_date setting. For example, if you want to run a weekly job that starts today, be sure your start date is more than a week prior to today.

YouTube URL: Short Video | Long Video. **Git Repo** Link.

DATASET SIZE (Jan 1, 2019 – May 7, 2023): 3,510,428 rows

MOST VISITED AGENCIES IN 2023:

1. Department of Health and Human Services
2. Postal Service
3. Commerce
4. Treasury

LEAST VISITED AGENCIES IN 2023:

1. Nuclear Regulatory Commission
2. U.S. International Development
3. National Science Foundation
4. Housing and Urban Development

NEXT STEPS:

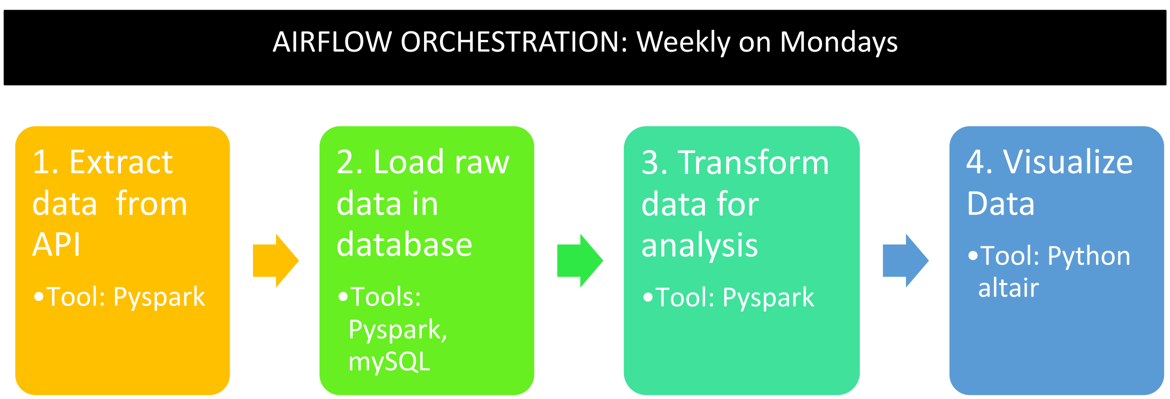
- Additional data from API: device use (mobile, desktop)
- Analyze individual website domain trends
- Integrate weekly/monthly economic data into the pipeline
- Bring in weekly twitter data for each agency to highlight topics of conversation, key influencers
- Serve on website

1. Description of software tools

Apache Airflow is what is considered a data pipeline orchestrator. It allows you to schedule a series of data pipeline tasks in a particular order to occur at a particular time. For example, you can schedule your data ingestion to happen every day at a certain time, and then sequentially run your other data pipeline steps, such as cleaning, transformation, and visualization. A barebones version of this is the bash Crontab utility. Airflow allows you to schedule a more complex set of tasks. Airflow uses what is called a Directed Acyclic Graph or a DAG to schedule task in a sequential order. You run airflow by starting up both (1) the web server and (2) the scheduler. The metadata for apache airflow is stored in a sqlite database.

I also use python, pyspark, and mysql software, which we cover in-depth in class. The infographic below describes the steps in my data pipeline. I use the **Airflow** as an orchestrator to ingest data in batches once a week. The exhibit below highlights my Extract-Load-Transformation ETL data pipeline that is initialized with Airflow . I use pySpark to initially hold the raw data I pull down from the API. I write the data from pySpark to a mySQL database. I then load the raw data back into pySpark to transform it so it is ready for analysis, aggregating the raw data into monthly visit counts for each agency. Finally, I load the much smaller, aggregated dataset into python and visualize it with the altair package.

Exhibit 1: Extract-Load-Transform (ELT) Data Pipeline Steps and Tools



constants.py batch_api.py transform.py visualize.py
main_api_pull.py

2. Description of data

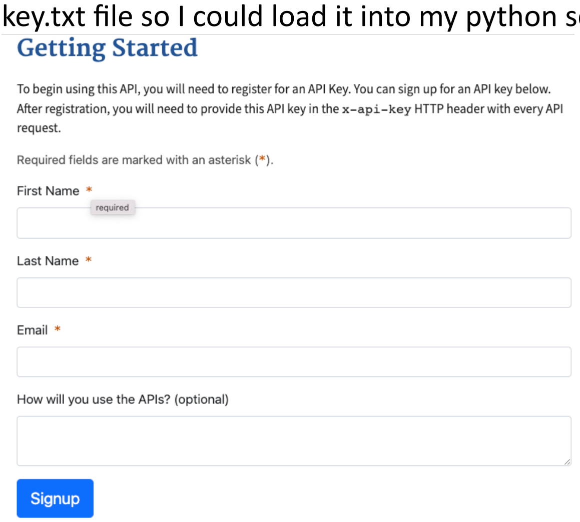
The data is from an API hosted by analytics.usa.gov as part of the openGSA initiative. This API has daily updates on the number of visits to federal government websites, including the specific domain visited and the device used (e.g., desktop, tablet, mobile). You can pull a variety of different reports from the reports endpoint. I use the “site” report for this case study, which allows me to pull the data for each individual website domain for each federal agency. The API limits each query to 10,000 data points. The parameters in the API request include limit, page, after, before, and api_key. I use the python request package to retrieve the data from the API.

An example of the daily website visits data is in the image below. The returned fields are id, date, report_name, report_agency, domain, and visits. The data is standardized and so I did not have to perform any cleaning on my end.

id	date	report_name	report_agency	domain	visits
124444150	2023-05-01	site	justice	justice.gov	306567
124444174	2023-05-01	site	justice	vehiclehistory.bja.ojp.gov	3886
124444180	2023-05-01	site	justice	getsmartaboutdrugs.gov	2791
124444192	2023-05-01	site	justice	bjatta.bja.ojp.gov	696
124444162	2023-05-01	site	justice	ic3.gov	8890
124444204	2023-05-01	site	justice	ncjrs.gov	149
124444168	2023-05-01	site	justice	search.justice.gov	5422
124444198	2023-05-01	site	justice	forfeiture.gov	330
124444186	2023-05-01	site	justice	vcf.gov	1578
124444156	2023-05-01	site	justice	dea.gov	44716
124444175	2023-05-01	site	justice	justthinktwice.gov	3852
124444151	2023-05-01	site	justice	fbi.gov	143137
124444181	2023-05-01	site	justice	leb.fbi.gov	2641
124444163	2023-05-01	site	justice	amberalert.ojp.gov	7623

3. Details of Install and Configuration

Step 0: Request api key from open.gsa.gov/api/dap/#getting-started
I filled out the form below and instantly received an api key in my email. I saved the api key in a key.txt file so I could load it into my python script.


Getting Started

To begin using this API, you will need to register for an API Key. You can sign up for an API key below. After registration, you will need to provide this API key in the `x-api-key` HTTP header with every API request.

Required fields are marked with an asterisk (*).

First Name *

required

Last Name *

Email *

How will you use the APIs? (optional)

Signup

Step 1: Create Virtual Environment and Install Python Packages

I created a virtual environment called **airflow_env** with the following command in my terminal. The remainder of all terminal operations in this document occur within this airflow_env virtual environment.

conda create --name airflow_env python=3.9 -y

I installed the needed python packages in this terminal: pyspark, pandas, altair PyMySQL, SQLAlchemy with the following commands in my terminal:

pip install altair
pip install pyspark
pip install pandas
pip install PyMySQL
pip install SQLAlchemy

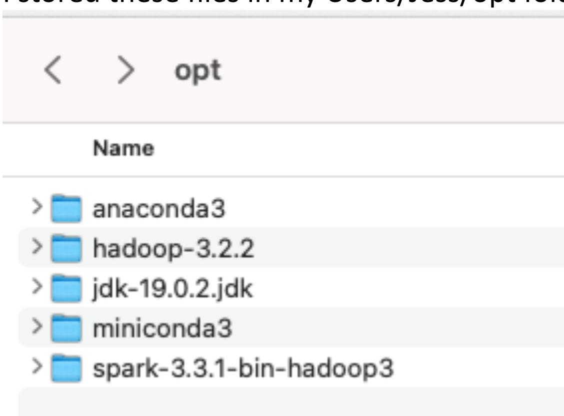
Step 2: Install and Setup Apache Spark

1. **Download and Install:**
   a. **Java SE Development Kit (JDK):** https://www.oracle.com/java/technologies/javase/jdk19-archive-downloads.html

   b. **Hadoop 3.2.2 binaries:** https://github.com/cdarlint/winutils/tree/master/hadoop-3.2.2/bin

   c. **Spark 3.3.1:** Go to spark.apache.org/downloads.html to download a spark-3.3.1-bin-hadoop3.tgz and untar the file.

I stored these files in my Users/Jess/opt folder as pictured below.



Step 3: Download MySQL Community Server and setup mysql database:
https://dev.mysql.com/downloads/mysql/.

- I created a root user and created a password for the root user in the installation process. Then, I logged into the sql database with the following terminal prompt: `sudo mysql –u root –p.` I then entered my password.
- I then created a table called "visits" with the "Create Table" command with the following specifications to hold the data I will pull from my API.

```
mysql> describe visits;
+---------------+------+------+-----+---------+-------+
| Field         | Type | Null | Key | Default | Extra |
+---------------+------+------+-----+---------+-------+
| id            | int  | YES  |     | NULL    |       |
| date          | text | YES  |     | NULL    |       |
| report_name   | text | YES  |     | NULL    |       |
| report_agency | text | YES  |     | NULL    |       |
| domain        | text | YES  |     | NULL    |       |
| visits        | int  | YES  |     | NULL    |       |
+---------------+------+------+-----+---------+-------+
6 rows in set (0.03 sec)
```

2. **Update .zshrc file as follows:**
   a. JAVA_HOME, SPARK_HOME, and HADOOP_HOME environmental variables.
   b. JAVA_HOME and SPARK_HOME to the PATH variable.
   c. /usr/local/mysql location to my PATH variable.

```
export
JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk1.8.0_361.jdk/Contents/Hom
e"
export PATH=$JAVA_HOME/bin:$PATH

export SPARK_HOME=/Users/Jess/opt/spark-3.3.1-bin-hadoop3
export PATH=$SPARK_HOME/bin:$PATH

export PATH=/Users/Jess/opt/scala@2.12/bin:$PATH

export HADOOP_HOME=/Users/Jess/opt/hadoop-3.2.2
```

Exhibit: My .zshrc file

```
export PATH="/Users/Jess/opt/anaconda3/bin:$PATH"

export JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk1.8.0_361.jdk/Contents/Home"
export PATH=$JAVA_HOME/bin:$PATH

export SPARK_HOME=/Users/Jess/opt/spark-3.3.1-bin-hadoop3
export PATH=$SPARK_HOME/bin:$PATH

export PATH=/Users/Jess/opt/scala@2.12/bin:$PATH

export HADOOP_HOME=/Users/Jess/opt/hadoop-3.2.2

export ANACONDA_HOME=/Users/Jess/opt/anaconda3

#export PYSPARK_DRIVER_PYTHON=jupyter
#export PYSPARK_DRIVER_PYTHON_OPTS='notebook'

export PATH=$PATH:/usr/local/mysql/bin
```

Step 3: Install and Setup Apache Airflow

I do two things to my .zshrc folder. First, I export a filepath for AIRFLOW_HOME. Second, I ensure my PYSPARK_DRIVER lines are commented-out so I can run pyspark as a script through spark-submit rather than opening up a jupyter notebook.

export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'

I installed Apache Airflow following the documentation's QuickStart guide (https://airflow.apache.org/docs/apache-airflow/stable/start.html). I entered the code from the Quickstart guide below in my terminal, which defines the AIRFLOW_VERSION as 2.6.0 and PYTHON_VERSION as 3.9. The CONSTRAINT_URL downloads constraints needed for the install from github. The install also uses pip install.

```
# Install Airflow using the constraints file
AIRFLOW_VERSION=2.6.0
PYTHON_VERSION="$(python --version | cut -d " " -f 2 | cut -d "." -f 1-2)"
# For example: 3.7
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.
# For example: https://raw.githubusercontent.com/apache/airflow/constraints-2.6.0/constraints-3.7.txt
pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL}"
```

Now, with airflow installed, I initialize the airflow database with the terminal command: **airflow db init**. This created a directory inside my AIRFLOW_HOME directory with the files in the image below.

```
airflow
(airflow_env) Jess$ cd airflow/
(airflow_env) Jess$ ls
airflow-scheduler.err        airflow-webserver-monitor.pid    airflow-webserver.pid    logs
airflow-scheduler.log        airflow-webserver.err            airflow.cfg              webserver_config.py
airflow-scheduler.out        airflow-webserver.log            airflow.db
airflow-scheduler.pid        airflow-webserver.out            dags
```

I made a configuration change to the airflow.cfg file to set the "load_examples=False." The default is True. When it is set to True, airflow has a bunch of their own examples in the airflow database and I preferred to clear this out.

```
# Whether to load the DAG examples that ship with Airflow. It's good to
# get started, but you probably want to set this to ``False`` in a production
# environment
load_examples = False
```

I then created an airflow use with login credentials with the "users create" terminal command shown below. I can verify that the user was created with the terminal command also below: airflow users list.

```
(airflow_venv) Jess$ airflow users create \
> --username admin \
> --password admin \
> --firstname Jess \
> --lastname Stockham \
> --role Admin \
> --email jhsmith@berkeley.edu
```

```
User "admin" created with role "Admin"
(airflow_venv) Jess$ airflow users list
id | username | email                | first_name | last_name | roles
===+==========+======================+============+===========+======
1  | admin    | jhsmith@berkeley.edu | Jess       | Stockham  | Admin
```

Next, I started the airflow web server with the following command. This starts the airflow web server on port 8081. The default port is 8080 if you do not specify a port but I have other processes running on 8080, so I changed the port number.

airflow webserver -D -p 8081

I then started the airflow scheduler with the following command:

airflow scheduler -D

4. Code with comments

I describe my series of code files below.

1. constants.py

The constants.py script gets everything setup. It (1) imports python packages I will need (2) defines variables I use as parameters in my api pulls, and (3) prepares Spark to store the data. I import the python packages: requests, json, pandas, pyspark, and datetime. For my api parameters, I import my api key that resides in a text file (key.txt), generate of agencies for which I will request reports, define the api url, and also specify the start and end dates (today's date and the "pull after" date). Notably, the date parameters are updated for each api batch request. I use the datetime python module to calculate today's date and the date 7 days prior. This code also defines my Spark session object, tests that Spark is working, and defines a schema for my Spark dataframe that will hold the data.

Exhibit: constants.py script

```
#!/usr/bin/env python
# coding: utf-8

import requests
import json
import pandas as pd
import pyspark
from datetime import datetime, timedelta
from pyspark.sql import SparkSession

''' The constants.py file defines the parameters that we will use in the api pull.
 This includes the api key, the list of federal agencies for which we will request data,
 the date range (today and 7 days in the past).
 The api documentation is at https://open.gsa.gov/api/dap/'''

# Api key
f = open('/Users/Jess/Documents/e63/apiPull/key.txt', 'r')
key = f.read()

# Agencies to query
agencies = ['agency-international-development',
        'agriculture',
        'commerce',
        'defense',
        'education',
        'energy',
        'environmental-protection-agency',
        'executive-office-president',
        'general-services-administration',
        'health-human-services',
        'homeland-security',
        'housing-urban-development',
        'interior',
        'justice',
        'labor',
        'national-aeronautics-space-administration',
        'national-archives-records-administration',
        'national-science-foundation',
        'nuclear-regulatory-commission',
        'office-personnel-management',
        'postal-service',
        'small-business-administration',
        'social-security-administration',
```

```
        'state',
        'transportation',
        'treasury',
        'veterans-affairs']

# API URL
url = 'https://api.gsa.gov/analytics/dap/v1.1'

# pullAfter: This defines the timeframe of the batch data pull (go back 7 days)
# For example, if today is May 4 when we execute the code, the pullAfter date is April 27.
# We pull down data from the API from April 27 - May 3.
pullAfter = datetime.now() - timedelta(7)
pullAfter = datetime.strftime(pullAfter, '%Y-%m-%d')
print(f'Two Days Ago: {pullAfter}')

# TODAY'S DATE
today = datetime.today().strftime('%Y-%m-%d')
print(f'Today: {today}')

# Schema for Spark Dataframe. Will hold data from API
from pyspark.sql.types import *

schema = StructType([
  StructField("id", IntegerType(), True),
  StructField("date", StringType(), True),
  StructField("report_name", StringType(), True),
  StructField("report_agency", StringType(), True),
  StructField("domain", StringType(), True),
  StructField("visits", IntegerType(), True)
])

print("Loaded Constants")

# Spark Session Object
spark = SparkSession.builder.getOrCreate()

# Show that pyspark is working by printing out "Hello Spark" in a dataframe
df = spark.sql("select 'spark' as hello ")
df.show()

print("Spark is Setup")
```

2. main_api_pull.py

main_api_pull.py is the workhorse that extracts the data from the API. This script imports the constants.py file and defines two functions that pull down the data from the AIP: apiPull() and paginate().

apiPull() requires the parameters page, after, before, and agency to feed into the api request. The parameter "page" determines the page number of the data we are pulling from the API. Since the limit for the api is 10,000 data points per page, we must pull multiple pages when we pull more than 10,000 records for a given agency. The parameter "after" represents the "pullAfter" variable defined in the contants.py file (7 days prior to today). The parameter "before" represents the "today" variable defined in the constants.py file (today's date). The parameter "agency" represents one agency within the list of agencies defined in the constants.py file.

Within the function, I define the parameters variable, which is a dictionary of all parameters I will feed into the api request. I must use a different api endpoint for each agency. Therefore, I define the endpoint with the agencyReport variable. The response variable holds the results of the api request (url + agencyReport endpoint + parameters), implemented using the python requests package. I return a serialized json object.

paginate() calls the apiPull() function and therefore takes in the parameters apiPull requires in its function signature: after, before, agency. Paginate() loops through all pages of the api endpoint.

First, within paginate, I define a blank Spark dataframe, masterDF, that will hold all the data from all pages. The body of the function is a while loop. I use a while loop to continue to execute a loop as long as the last api request returned data (the length of the data variable is not zero). This allows me to extract all pages of the endpoint. The loop calls the apiPull function and stores the result in the variable "data." I then execute an "if, else" set of statmements. **If** the last api request returned no data (the length of the data variable is zero) and I return the Spark dataframe. **Else**, I store the data in a new Spark dataframe; append that dataframe to the masterDF with the Spark union() function; and finally increment the "page" parameter by 1 to prepare for the next loop.

Exhibit: main_api_pull.py script

```
#!/usr/bin/env python
# coding: utf-8

# Step 1: Load constant variables
from constants import *

# Step 2: Define functions to pull data from API
def apiPull(page, after, before, agency):
    """Request data from api"""

    """
    Parameters:
    - page: page number of the request
    - after: data after which we are requesting data
    -agency: federal agency for which we are requesting data
    """

    # Parameters used in the api request
    parameters = {
```

```
    "limit": 10000,  # Limit is 10,000
    "page" : page,
    "after": after,
    "before": before,
    "api_key": key}

    # Endpoint
    agencyReport = '/agencies/' + agency + '/reports/site/data'

    # Request data
    response = requests.get(url + agencyReport, params=parameters)

    print(f'response status code: {response.status_code}')

    # Return a list of json objects
    return response.json()


def paginate(after, before, agency):
    """Loop through all pages until we have all requested data for that agency"""
    """
    Parameters:
    - after: data after which we are requesting data
    -agency: federal agency for which we are requesting data
    """

    masterDF = spark.createDataFrame(data = [], schema = schema)    # initialize blank df
    page = 1 # starting place
    data = [1] # placeholder

    # If the request returned data, keep going
    while len(data) != 0:
        data = apiPull(page, after, before, agency)
        print(masterDF.count())

        # If the length of the data returned is 0, stop and return df
        if len(data) == 0:
            print(f'done with {agency}')
            return masterDF

        # If the data has a length > 0, append that data to the masterDF
        # Increment the page count by 1
        else:

            df = spark.createDataFrame(data, schema)
            masterDF = masterDF.union(df)
            page += 1
```

3. historical_api.py

The historical_api.py script pulls down all website visit data for all federal agencies from the last 5 years: from Jan 1, 2019 through April 30, 2023. This large extraction grabs 3,496,620 rows of data. First, historical_api.py imports constants.py and the main_api_pull.py scripts. Then I loop through all agencies in the agency list, calling the paginate() function for each one. I append all the API data for all agencies in a Spark dataframe called allAgencyDF. Then, I write the allAgencyDF to a mysql database, in a table called “visits” that uses the schema I defined in constants.py.

I run historical_api.py as a script through spark-submit and it takes several minutes to run. I navigate to the folder that holds the scripts so I do not need to write a long path to the script. My terminal command is below.

```
$SPARK_HOME/bin/spark-submit historical_api.py
```

Exhibit: historical_api.py script

```
#!/usr/bin/env python
# coding: utf-8

# Step 1: Load constant variables
from constants import *

# Step 2: Load functions we will use to pull down data from API
from main_api_pull import *

# Step 3: Extract historical data from API: Jan 1, 2019 - April 30, 2023
# Instantiate Blank DF to hold all data
allAgencyDF = spark.createDataFrame(data=[], schema=schema)

# Loop through all agencies
for agency in agencies:

  # Pull Data
  agencyDF = paginate(after='2019-01-01', before='2023-04-26', agency=agency)
  print(f'agency count: {agencyDF.count()}')
  allAgencyDF = allAgencyDF.union(agencyDF)

allAgencyDF.count()

# Write data to "visits" table in mysql database
allAgencyDF.write \
  .format("jdbc") \
  .option("url", "jdbc:mysql://localhost:3306") \
  .option("dbtable", "websites.visits") \
  .option("user", "root") \
  .option("password", "password") \
  .mode("overwrite").save()
```

Exhibit: Snips of terminal output for bash run of historical_api.py

```
Jess$ $SPARK_HOME/bin/spark-submit historical_api.py
Two Days Ago: 2023-05-01
Today: 2023-05-08
Loaded Constants
+-----+
|hello|
+-----+
|spark|
+-----+

Spark is Setup
response status code: 200
0
response status code: 200
8296
done with agency-international-development
agency count: 8296
response status code: 200
0
```

```
response status code: 200
80000
response status code: 200
90000
response status code: 200
100000
response status code: 200
110000
response status code: 200
120000
response status code: 200
130000
response status code: 200
140000
response status code: 200
150000
response status code: 200
160000
response status code: 200
170000
response status code: 200
180000
response status code: 200
```

Exhibit: Partial data from resulting table “visits” after I run historical_api

```
mysql> select * from visits where report_agency="justice" order by date limit 30;
```

id	date	report_name	report_agency	domain	visits
32288662	2019-01-01	site	justice	ovcttac.gov	263
32288658	2019-01-01	site	justice	ovc.gov	323
32288659	2019-01-01	site	justice	vcf.gov	309
32288670	2019-01-01	site	justice	search.usmarshals.gov	106
32288657	2019-01-01	site	justice	search.dea.gov	417
32288669	2019-01-01	site	justice	admin.dea.gov	106
32288656	2019-01-01	site	justice	ojp.gov	423
32288655	2019-01-01	site	justice	crimesolutions.gov	541
32288668	2019-01-01	site	justice	nsi.ncirc.gov	118
32288654	2019-01-01	site	justice	search.atf.gov	558
32288653	2019-01-01	site	justice	ojjdp.gov	569
32288652	2019-01-01	site	justice	amberalert.gov	585
32288667	2019-01-01	site	justice	it.ojp.gov	125
32288660	2019-01-01	site	justice	ovc.ncjrs.gov	268
32288651	2019-01-01	site	justice	oig.justice.gov	586
32288650	2019-01-01	site	justice	nationalgangcenter.gov	607
32288649	2019-01-01	site	justice	foia.gov	670
32288666	2019-01-01	site	justice	bja.gov	198
32288648	2019-01-01	site	justice	usmarshals.gov	1072
32288647	2019-01-01	site	justice	nicsezcheckfbi.gov	1089
32288632	2019-01-01	site	justice	justice.gov	52809
32288633	2019-01-01	site	justice	bop.gov	42884
32288665	2019-01-01	site	justice	unicor.gov	200
32288634	2019-01-01	site	justice	dea.gov	10069
32288661	2019-01-01	site	justice	search.ada.gov	265
32288635	2019-01-01	site	justice	atf.gov	9475
32288646	2019-01-01	site	justice	getsmartaboutdrugs.gov	1186
32288664	2019-01-01	site	justice	einfo.eoir.justice.gov	227
32288636	2019-01-01	site	justice	nsopw.gov	8924
32288645	2019-01-01	site	justice	justthinktwice.gov	1224

```
30 rows in set (1.29 sec)

mysql> select count(*) from visits;
+----------+
| count(*) |
+----------+
|  3493620 |
+----------+
1 row in set (0.49 sec)
```

```
mysql> select distinct report_agency from visits;
+-------------------------------------------+
| report_agency                             |
+-------------------------------------------+
| agency-international-development          |
| agriculture                               |
| commerce                                  |
| defense                                   |
| education                                 |
| energy                                    |
| environmental-protection-agency           |
| executive-office-president                |
| general-services-administration           |
| health-human-services                     |
| homeland-security                         |
| housing-urban-development                 |
| interior                                  |
| justice                                   |
| labor                                     |
| national-aeronautics-space-administration |
| national-archives-records-administration  |
| national-science-foundation               |
| nuclear-regulatory-commission             |
| office-personnel-management               |
| postal-service                            |
| small-business-administration             |
| social-security-administration            |
| state                                     |
| transportation                            |
| treasury                                  |
| veterans-affairs                          |
+-------------------------------------------+
27 rows in set (1.76 sec)
```

4. **batch_api.py (Task 1 Called by Apache Airflow pipeline)**

batch_api.py is almost same as historical_api.py, but instead of calling all historical data, it instead dynamically pulls the latest data from the last seven days. This is a script that I will call inside of **Apache Airflow** to implement the data pipeline weekly . Notably, when I write the new batch data to the mysql database table "visits," I am "appending" the data because we are appending the batch to the historical data already pulled in historical_api.py. Every weekly batch will keep appending on to this table.

```
# Step 1: Load constant variables
from constants import *

# Step 2: Load functions we will use to pull down data from API
from main_api_pull import *

# Step 3: Pull All Historical Data

# Instantiate Blank DF to hold all data
allAgencyDF = spark.createDataFrame(data = [], schema = schema)

agencies = ['justice']
# Loop thorugh all agencies
for agency in agencies:

    # Pull All Data from Jan 1, 2020 through May 2, 2023
```

```
    agencyDF = paginate(after=pullAfter, before=today, agency=agency)
    print(f'agency count: {agencyDF.count()}')

    allAgencyDF = allAgencyDF.union(agencyDF)

allAgencyDF.count()

# Write data to "visit" table in mysql database
allAgencyDF.write    \
    .format("jdbc")    \
    .option("url","jdbc:mysql://localhost:3306")    \
    .option("dbtable", "websites.visits")    \
    .option("user", "root")    \
    .option("password", "password")    \
    .mode("append").save()
```

5. **transform.py (Task 2 Called by Apache Airflow pipeline)**

transform.py loads the raw *visits* table from the mysql database into a Spark dataframe called df_visits. I transform the raw data to monthly counts so it is ready for analysis and visualization. I use a select statement to group the website visits by month. For each agency, I use the sum() function to add together all website visits across all website domains for each month. Then I write the transformed data to a new table in the mysql database called *monthly_agency*. The write function "overwrites" the *monthly_agency* table because we need to re-aggregate the data with the new data pull. The monthly_agency table has one row for each agency for each month of the year.

Exhibit: transform.py script

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import window, column, desc, col, year, month, dayofmonth

spark = SparkSession.builder.getOrCreate()

# Show that pyspark is working
df = spark.sql("select 'spark' as hello ")
df.show()

# Read in visits data from websites database
df_visits = spark.read \
    .format("jdbc") \
    .option("url","jdbc:mysql://localhost:3306") \
    .option("dbtable", "websites.visits") \
    .option("user", "root") \
    .option("password", "password") \
    .load()

df_visits.show(1)

# Aggregate Table: Agency Visits by Month
```

```
monthly_agency = df_visits\
.select(
   "report_agency", "visits", "date",
   year("date").alias('year'),
   month("date").alias('month'),
   dayofmonth("date").alias('day'))\
.groupBy(col("report_agency"), col("year"), col("month"))\
.sum("visits").withColumnRenamed("sum(visits)", "visits_agg").orderBy("report_agency", "year","month")

monthly_agency.show(1)

# Save transformed table into database
# Over-write stored data
monthly_agency.write \
   .format("jdbc") \
   .option("url","jdbc:mysql://localhost:3306") \
   .option("dbtable", "websites.monthly_agency") \
   .option("user", "root") \
   .option("password", "password")\
   .mode("overwrite").save()
```

6. **visualize.py (Task 3 Called by Apache Airflow pipeline)**

visualize.py loads the data from the mysql database into a pandas dataframe and then visualizes it using the python altair package. First, I installed 3 python packages (PyMySQL, SQLAlchemy, altiar) in to my virtual environment using pip install. I use PyMySQL and SQLAlch to load in the monthly_agency table from mysql database into a pandas dataframe called df.

I also load in the visits table in order to pull out the last date extracted. I leverage the datetime python package to parse the year, month, and day from the string (dt.year, dt.month, dt.day). I use this information to populate the title of the 2023 chart.

Next, I use the pandas dataframe df as the source data for my altair heatmap chart. The heatmap chart is interactive where you can hover over each square to view its data. Notably, I will have to update this code when we get to 2024. A future improvement to this code would make it fully automated so manual coding changes are not needed in future years. I save the altair chart as an html file so it is ready to be served up on a website.

Exhibit: visualize.py

```
from sqlalchemy import create_engine
import pymysql
import altair as alt
import pandas as pd
from datetime import datetime

# Connect to mysql database
mysql_string = 'mysql+pymysql://root:password@localhost:3306/websites'
conn = create_engine(mysql_string)
df = pd.read_sql('SELECT * FROM monthly_agency', con=conn)
```

```
# Get Last Day of Data
df_raw = pd.read_sql('SELECT date FROM visits  order by date desc limit 1', con=conn)
datestring = df_raw.iloc[0,0]
dt = datetime.strptime(datestring, '%Y-%m-%d')
print(dt.year, dt.month, dt.day)

# Create Heatmap and save as html file
# Will need to update the code when we get to 2024

tooltip=[alt.Tooltip('report_agency:O', title = 'Agency'),
        alt.Tooltip('year:O', title = "Year"),
        alt.Tooltip('month:O', title = 'Month'),
        alt.Tooltip('visits_agg:Q', title = 'Monthly Website Visits', format=",.0f")]

chart2019 = alt.Chart(df.loc[df.year==2019], title="Monthly Website Visits in 2019").mark_rect().encode(
    alt.X("month:O").title("Month"),
    alt.Y("report_agency:O").title("Agency"),
    tooltip,
    alt.Color("visits_agg").legend(title='Monthly Website Visits', format=",.0f"))

chart2020 = alt.Chart(df.loc[df.year==2020], title="2020").mark_rect().encode(
    alt.X("month:O").title("Month"),
    alt.Y("report_agency:O").title("Agency").axis(labels=False),
    tooltip,
    alt.Color("visits_agg"))

chart2021 = alt.Chart(df.loc[df.year==2021], title="2021").mark_rect().encode(
    alt.X("month:O").title("Month"),
    alt.Y("report_agency:O").title("Agency").axis(labels=False),
    tooltip,
    alt.Color("visits_agg"))


chart2022 = alt.Chart(df.loc[df.year==2022], title="2022").mark_rect().encode(
    alt.X("month:O").title("Month"),
    alt.Y("report_agency:O").title("Agency").axis(labels=False),
    tooltip,
    alt.Color("visits_agg"))

chart2023 = alt.Chart(df.loc[df.year==2023], title="2023 through " + str(dt.month) + '/' + str(dt.day) + '/' +
str(dt.year)).mark_rect().encode(
    alt.X("month:O").title("Month"),
    alt.Y("report_agency:O").title("Agency").axis(labels=False),
    tooltip,
    alt.Color("visits_agg"))

heatmap = chart2019 | chart2020 | chart2021 | chart2022 | chart2023
heatmap.save('/Users/Jess/Documents/e63/apiPull/heatmap.html')
```

7. api_dag.py

The api_dag.py is the file used by Apache Airflow to schedule the data pipeline job. I must import several airflow functions, including DAG and BashOperator

The dag_id is 'api_dag.' With the first block of code, I schedule this data pipeline to run once a week, every Monday at 17:30 UTC, defined by the schedule_interval variable. There are five positions in the syntax: minute, hour, day of month, month, day of the week. The syntax here is the same as for the crontab bash utility. The pipeline starts on a date in the past, arbitrarily April 4, 2023, per the start_data variable. Catchup=False means that I don't want the DAG to go back and pull data from the past.

I define three Tasks in the dag via the BashOperator. The BashOperator allows me to submit bash commands. Task 1 (task_api) runs the batch_api.py script through the spark-submit bash command. Task 2 (task_transform) runs the transform.py script through the spark-submit bash command. Task 3 (task_viz) runs the visualize.py through the python3 bash command.

I set the order of the three tasks with the last statement with the >> statement, first running task_api, then task_transform, then task_viz.

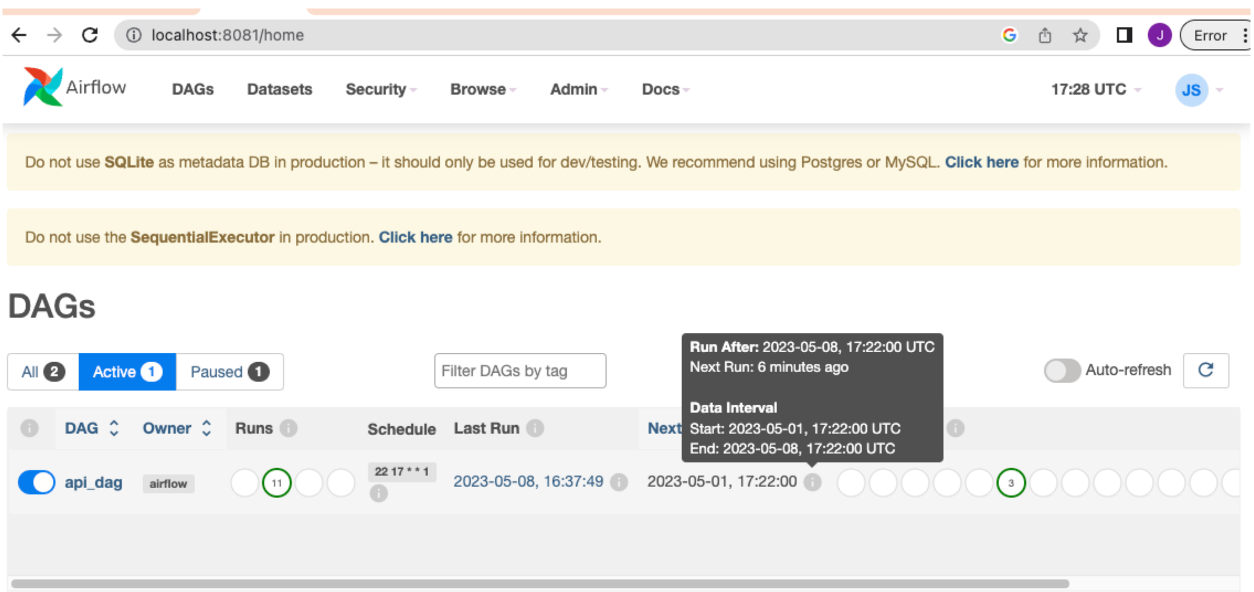
Exhibit: api.dag.py

```
import os
import pandas as pd
from datetime import datetime
from airflow.models import DAG
from airflow.operators.bash import BashOperator


with DAG(
    dag_id='api_dag',
    schedule_interval='30 17 * * 4',
    start_date=datetime(year=2022, month=2, day=1),
    catchup=False
) as dag:
  # 1. Run pyspark script to pull api data and add raw data to database
  task_api = BashOperator(
    task_id='run_pyspark_api_script',
    bash_command='$SPARK_HOME/bin/spark-submit /Users/Jess/Documents/e63/apiPull/batch_api.py'
  )

  # 2. Transform data, save transformed data in a database table
  task_transform = BashOperator(
    task_id='run_pyspark_transform_script',
    bash_command='$SPARK_HOME/bin/spark-submit /Users/Jess/Documents/e63/apiPull/transform.py'
  )

  #3. Visualize data with python altair
  task_viz = BashOperator(
    task_id='run_python_viz_script',
    bash_command='python3 /Users/Jess/Documents/e63/apiPull/visualize.py'
```

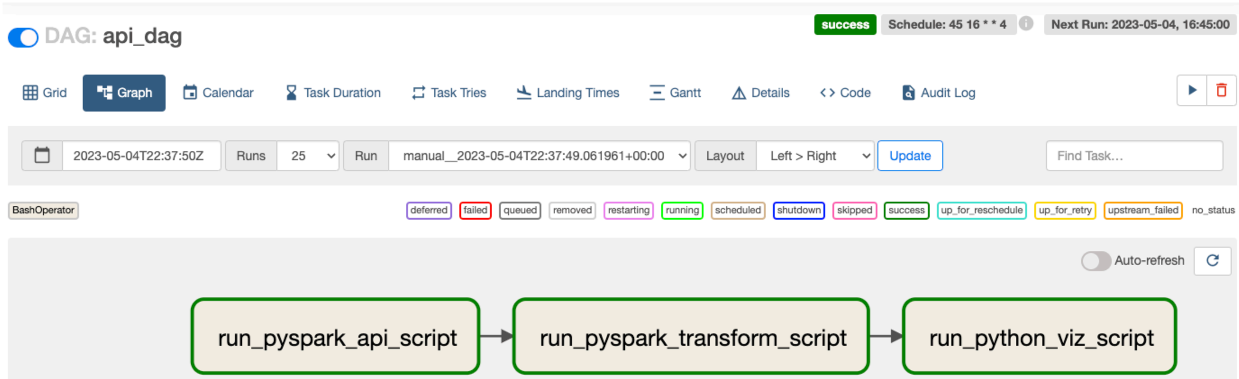
```
)

# Set order of DAG
task_api >> task_transform >> task_viz
```

5. Demonstration

I can visualize my dag in the browser at http://localhost:8081. It shows my Schedule, the last time it was run, and the next scheduled time it will run. Notably, it shows a previous run in the image below because I did some test runs first by hitting the "play triangle button." My demonstration is the first batch run of my DAG, happening on Monday, May 8, 2023. When I hover over the info circle, I can see that my next run is scheduled in 24 minutes, at 17:30 UTC on May 8, 2023. This batch run pulls data from May 1 – May 7 because the latest data available from the API is "yesterday."



If I click the api_dag and then hit "Graph", I can view my Tasks in the order that I specified. I also toggled on the DAG so it is active with the blue toggle button in the upper left.



Once my DAG ran on May 8, 2023, I pulled down the latest data from May 1 – May 7. A sql query of the visits table shows a total of 3,510,428 rows, which is an additional 13,808 rows of data from the last 7 days.

Exhibit: SQL Queries from "visits" Table (raw data pulled from API)

```
mysql> select max(date) from visits;
+------------+
| max(date)  |
+------------+
| 2023-05-07 |
+------------+
1 row in set (1.25 sec)

mysql> select count(*) from visits;
+----------+
| count(*) |
+----------+
|  3510428 |
+----------+
1 row in set (0.17 sec)

mysql> 
```

In the exhibit below, can also observe the total number of government website visits per year from 2019 through 2023, noting of course that the 2023 data is only through May 7, 2023. There was peak traffic in 2021, which tapered down in 2023. In 2023, the most popular website to visit so far is Heath and Human Services with over 2 billion visits. The next top three are postal service, commerce, and treasury. The bottom two agencies are the

Exhibit: Queries from the "monthly_agency" table.

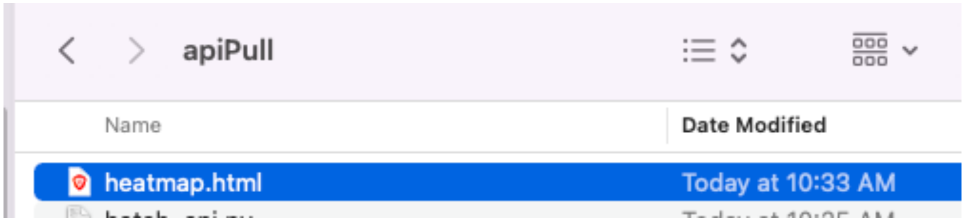
```
mysql> select * from monthly_agency order by report_agency, year, month limit 20;;
+----------------------------------+------+-------+------------+
| report_agency                    | year | month | visits_agg |
+----------------------------------+------+-------+------------+
| agency-international-development | 2020 |     1 |     842922 |
| agency-international-development | 2020 |     2 |     915566 |
| agency-international-development | 2020 |     3 |    1046642 |
| agency-international-development | 2020 |     4 |     975087 |
| agency-international-development | 2020 |     5 |     909230 |
| agency-international-development | 2020 |     6 |     851920 |
| agency-international-development | 2020 |     7 |     836459 |
| agency-international-development | 2020 |     8 |     828150 |
| agency-international-development | 2020 |     9 |     864525 |
| agency-international-development | 2020 |    10 |     909520 |
| agency-international-development | 2020 |    11 |     863583 |
| agency-international-development | 2020 |    12 |     825515 |
| agency-international-development | 2021 |     1 |     873075 |
| agency-international-development | 2021 |     2 |    1049867 |
| agency-international-development | 2021 |     3 |    1047693 |
| agency-international-development | 2021 |     4 |     971735 |
| agency-international-development | 2021 |     5 |     976631 |
| agency-international-development | 2021 |     6 |     917497 |
| agency-international-development | 2021 |     7 |     843364 |
| agency-international-development | 2021 |     8 |     892809 |
```

```
mysql> SELECT year, sum(visits_agg) from monthly_agency GROUP BY year;
+------+-----------------+
| year | sum(visits_agg) |
+------+-----------------+
| 2019 |     13786646685 |
| 2020 |     21475958918 |
| 2021 |     22376835800 |
| 2022 |     20626019311 |
| 2023 |      7491601466 |
```

```
mysql> SELECT report_agency, sum(visits_agg) as 2023_visits from monthly_agency WHERE
 year=2023 GROUP BY report_agency ORDER BY
sum(visits_agg) DESC;
+--------------------------------------------+-------------+
| report_agency                              | 2023_visits |
+--------------------------------------------+-------------+
| health-human-services                      |  2136882600 |
| postal-service                             |  1835121684 |
| commerce                                   |   595116163 |
| treasury                                   |   481753066 |
| homeland-security                          |   380755782 |
| state                                      |   293562633 |
| general-services-administration            |   242673808 |
| veterans-affairs                           |   187862136 |
| social-security-administration             |   164876341 |
| interior                                   |   157068831 |
| defense                                    |   134782672 |
| national-aeronautics-space-administration  |   127239356 |
| office-personnel-management                |   124490771 |
| justice                                    |   115479522 |
| education                                  |    96778292 |
| labor                                      |    84157041 |
| agriculture                                |    66160472 |
| transportation                             |    57374756 |
| national-archives-records-administration   |    56069355 |
| environmental-protection-agency            |    44426156 |
| executive-office-president                 |    34203007 |
| small-business-administration              |    21170490 |
| energy                                     |    21052578 |
| housing-urban-development                  |    15941308 |
| national-science-foundation                |    10087413 |
| agency-international-development           |     4659320 |
| nuclear-regulatory-commission              |     1855913 |
+--------------------------------------------+-------------+
27 rows in set (0.00 sec)
```

The resulting visualization is saved in the "heatmap.html" file as displayed in the exhibit below.

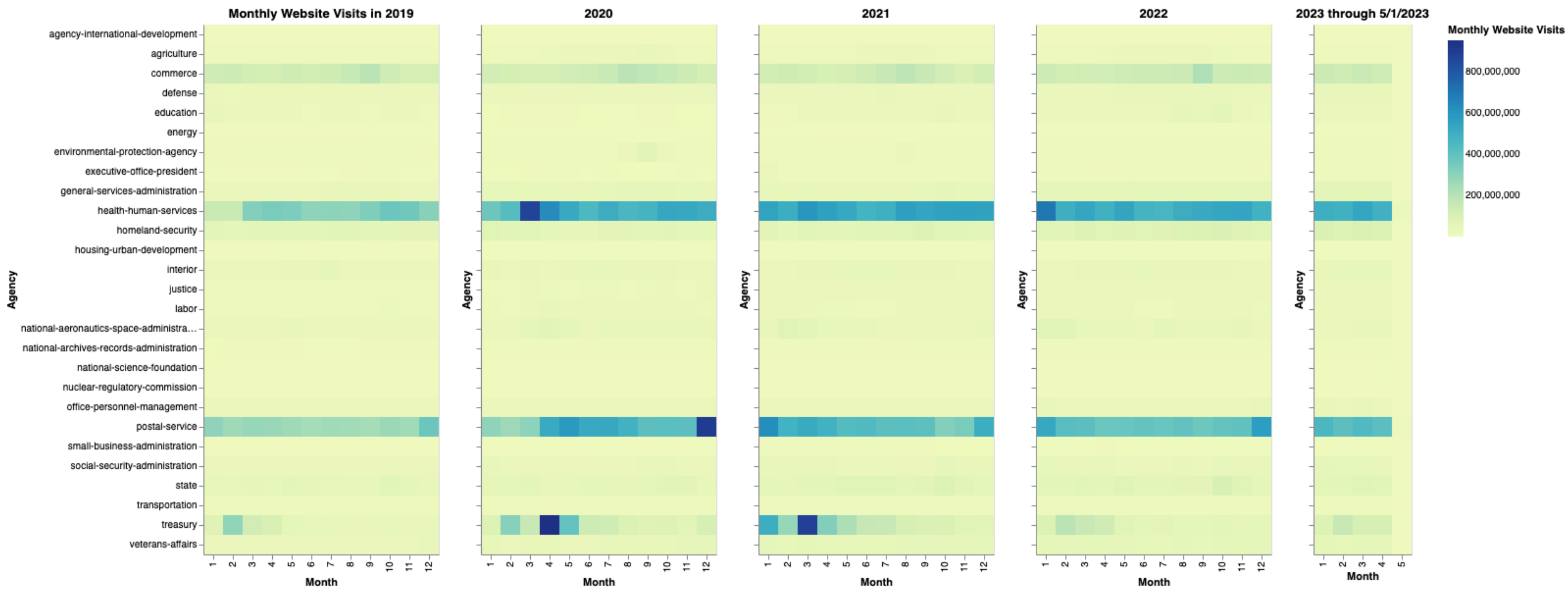
Exhibit: heatmap.html file in directory



As shown in the exhibit on the next page, the visualization in heatmap.html shows all the data through May 7, 2023. The title of the chart automatically also shows this cutoff date. This visualization shows that the websites of the department of health and human services, the postal service, and commerce are the most popular consistently over time. The treasury department has spikes in traffic during the tax season.

In the future, there is an opportunity to dig much more into the data to show individual trends for each agency, and for each agency's various website domains. I could also pull data from additional API endpoints on the device type for each visitor (e.g., desktop vs mobile). I would also like to integrate additional data sources into the pipeline, including weekly or monthly economic data (e.g., S&P 500, jobs reports) and twitter data to highlight the key conversation topics for each government agency and key influencers.

Heatmap.html



6. Summarize:

General Lessons Learned: Start as simple as possible when learning a new tool. Make sure a basic trivial example is working. Then, test every little step of your code and data pipeline before you try to string a pipeline together. I did this both with my API code and with the airflow code.

Airflow Lesson Learned/Challenges:

1. If you need to change something in the configuration file airflow.cfg, you have to reset the airflow db (airflow db reset) and also shutdown and restart the webserver and the scheduler. To shutdown the webserver and scheduler, you can use kill <pid>. I got the list of the active pids by running lsof -i tcp:8081. For example, in the image below, I found PID 71088. I used the command "kill 71088" to shutdown the airflow web server.

```
(airflow_env) Jess$ lsof -i tcp:8081
COMMAND      PID USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
python3.9 71088 Jess    5u  IPv4 0x9f611099e9c80803      0t0  TCP *:sunproxyadmin (LISTEN)
python3.9 96563 Jess    5u  IPv4 0x9f611099e9c80803      0t0  TCP *:sunproxyadmin (LISTEN)
python3.9 96564 Jess    5u  IPv4 0x9f611099e9c80803      0t0  TCP *:sunproxyadmin (LISTEN)
python3.9 96566 Jess    5u  IPv4 0x9f611099e9c80803      0t0  TCP *:sunproxyadmin (LISTEN)
python3.9 96568 Jess    5u  IPv4 0x9f611099e9c80803      0t0  TCP *:sunproxyadmin (LISTEN)
```

2. Airflow uses the UTC timezone for scheduling jobs by default. I was originally thinking it automatically detected my timezone but it does not.
3. If you want your DAG to start running right away at the first opportunity, make sure that your start_date in your schedule is far enough back. For example, I wanted to run my job weekly at 17:00 UTC every Monday (starting on Monday, May 8, 2023). However, I had my start_date as May 4, 2023. The job wouldn't run on Monday, May 8 because it was waiting to start 7 days after my start_date. I moved my start_date back to April 4, 2023 and it would run. Here is the article that I read that helped me figure out my problem: Troubleshooting the Apache Airflow Scheduler: DAG Not Triggered at Scheduled Time

Pros: Airflow has a user-friendly visual display of the pipeline. APIs from data.gov offer robust, clean data.

Cons: I did not find any specific cons of airflow.

7. Project URLs

YouTube 2-minute video: https://youtu.be/AYaaRbIt7I4
YouTube 15-minute video: https://youtu.be/CKeJR5OuxSQ
git Repo: **https://github.com/jhsmith22/e63finalproject.git**

8. References

Primary Sources:

Bill Chambers and Matei Zaharia. "Spark_The Definitive Guide." O'Reilly Publisher.
Tutorial on installing apache airflow: https://betterdatascience.com/apache-airflow-install/
Tutorial on writing your first apache airflow dag: https://betterdatascience.com/apache-airflow-write-your-first-dag/
Apache Airflow Quickstart Guide: https://airflow.apache.org/docs/apache-airflow/stable/start.html

Took a line a code from the following sources:

https://stackoverflow.com/questions/30483977/python-get-yesterdays-date-as-a-string-in-yyyy-mm-dd-format
https://stackoverflow.com/questions/32490629/getting-todays-date-in-yyyy-mm-dd-in-python
https://stackoverflow.com/questions/62977067/error-while-creating-data-frame-from-rest-api-in-pyspark
https://softhints.com/convert-mysql-table-pandas-dataframe-python-dictionary/