

MACHINE LEARNING LAB 03

STEFANO FOCESATTO

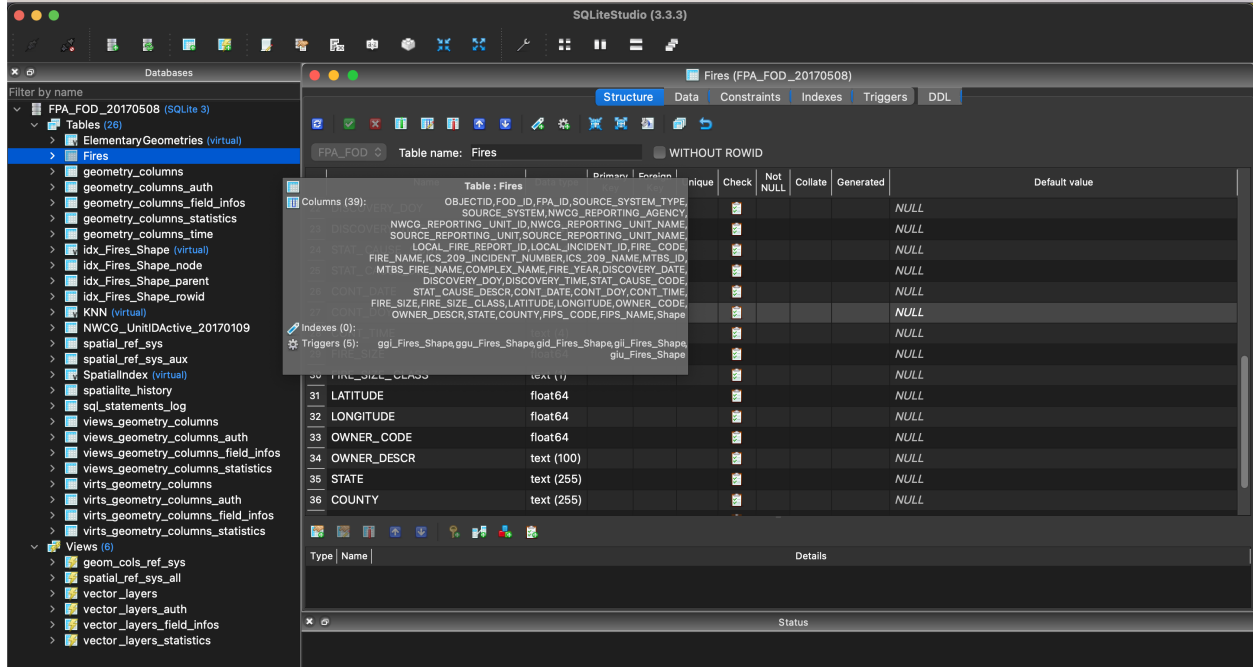
1. INTRODUCTION

The goal of this lab is to explore, and develop a workflow for storing and querying data using cloud platforms. In this lab we create a MySQL server on the three largest cloud computing platforms, Amazon Web Services, Microsoft Azure, and Google Cloud. We will also build a small web application using Dash, deployed on Heroku to query the MySQL servers for data. The data we will use for this lab, can be found on Kaggle, and contains 24 years of geo-referenced wildfire records from 1992 to 2015.

2. PRELIMINARY DATA PREPARATION

The data when pulled from Kaggle comes in the form of a sqlite database. We would like to get this data into a MySQL database as that is the management system that we will be using on the three cloud platforms. There are several ways to view data from a sqlite database, I chose to use a tool called SQLiteStudio, which is an sqlite equivalent to MySQL Workbench. In SQLiteStudio we can view tables and write queries. In doing so I found that a majority of the data is stored in a single table called Fires, and with SQLiteStudio we can also export it as a csv.

FIGURE 1. SQLiteStudio



It should be noted that we could also export the table using the sqlite3 command line interface with the following command,

Code:

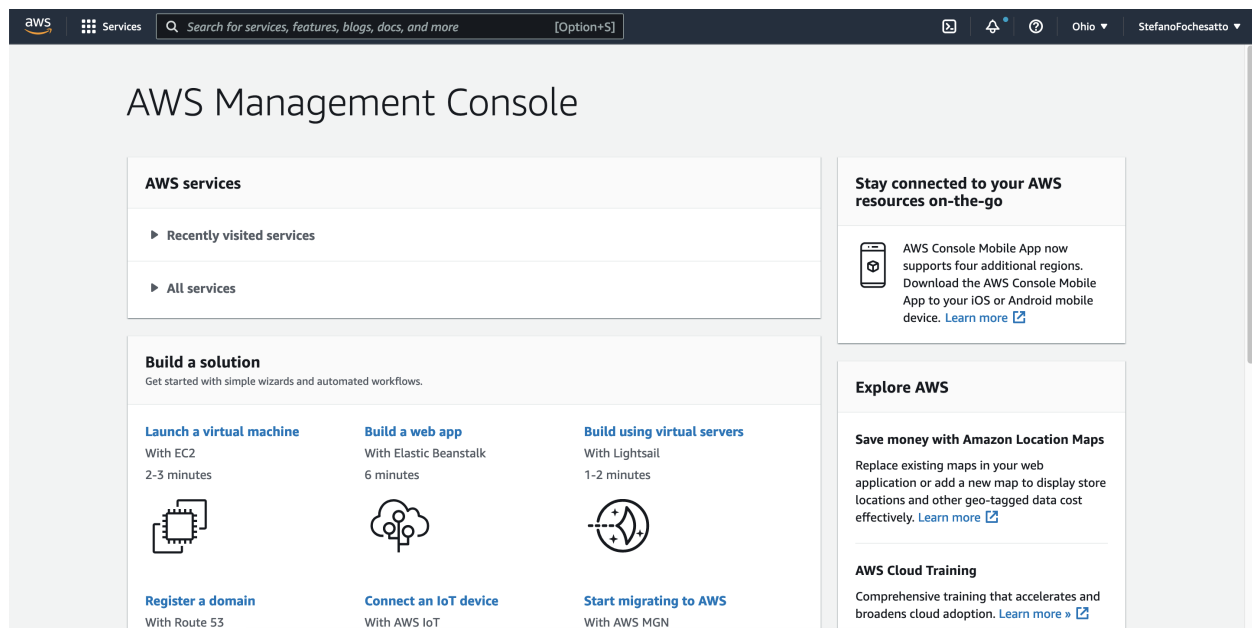
```
sqlite3 -header -csv c:/Downloads/FPA.FOD.20170508.sqlite "select * from Fires;" > data.csv
```

With the data as a .csv we are ready to initialize the cloud MySQL servers and begin importing the data.

3. INSTANTIATING MySQL SERVERS

Amazon Web Services. After we create an account with amazon web services we are met with the following dashboard.

FIGURE 2. AWS Main Dashboard



In the search bar we type 'RDS' which stands for relational database service, and follow the prompts to create a database. Finally we have reached the 'Create Database' menu.

FIGURE 3. AWS Create Database Menu

Create database


Choose a database creation method [Info](#)


☐ **Standard create**
You set all of the configuration options, including ones for availability, security, backups, and maintenance.


☒ **Easy create**
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.


Configuration


Engine type [Info](#)


☐ **Amazon Aurora**


☒ **MySQL**


☐ **MariaDB**


☐ **PostgreSQL**


☐ **Oracle**


☐ **Microsoft SQL Server**


Here we will select the 'Easy Create' option as for now we have a relatively simple use case. Then we will select the MySQL engine type and our database instance size, for our project we will continue with the free tier.

FIGURE 4. AWS Create Database Menu

DB instance size

<input type="radio"/> Production db.r6g.xlarge 4 vCPUs 32 GiB RAM 500 GiB 1.017 USD/hour	<input type="radio"/> Dev/Test db.r6g.large 2 vCPUs 16 GiB RAM 100 GiB 0.231 USD/hour	<input checked="" type="radio"/> Free tier db.t2.micro 1 vCPUs 1 GiB RAM 20 GiB 0.020 USD/hour
--	---	--

DB instance identifier
Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

database-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

Master username [Info](#)
Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. First character must be a letter.

☐ **Auto generate a password**
Amazon RDS can generate a password for you, or you can specify your own password.

Master password [Info](#)

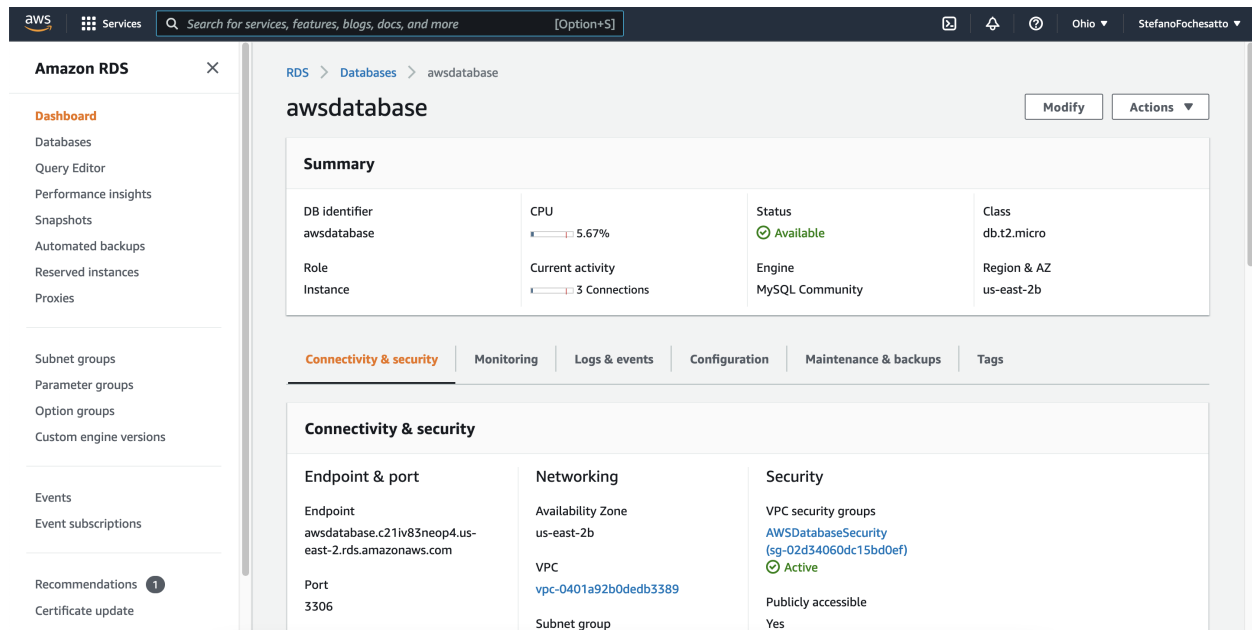
Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm password [Info](#)

Then we have to set the database instance identifier, which is used to differentiate instances in the RDS dashboard, and the master username and password. These will be important when we connect to the server using MySQL Workbench.

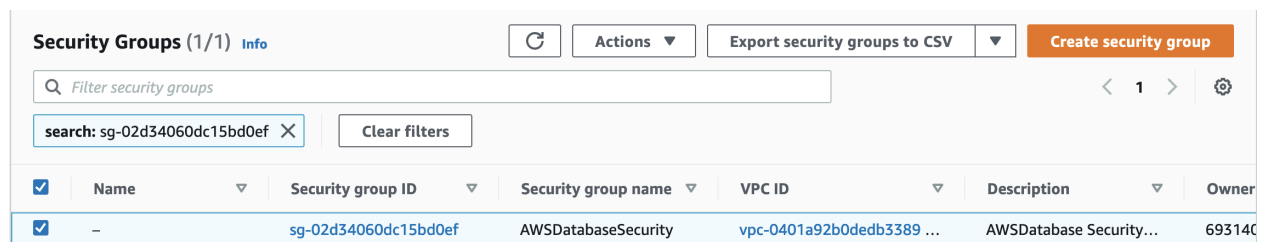
In a few minutes our RDS instance should be ready, and we should be at the following server management dashboard,

FIGURE 5. AWS Database Dashboard



but before we can connect to it we need to change the security group setting to allow an outside connection. To do this we click the link under VPC Security and to open the security groups dashboard.

FIGURE 6. AWS Security Group Dashboard



From here we want to create a new security group, that allows all ip addresses to connect (Note that we could have the server only accept a list of ip addresses for added security).

FIGURE 7. AWS Security Group Dashboard

Create security group [Info](#)

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a new security group, complete the fields below.

Basic details

Security group name [Info](#)

Name cannot be edited after creation.

Description [Info](#)

VPC [Info](#)

Inbound rules [Info](#)

This security group has no inbound rules.

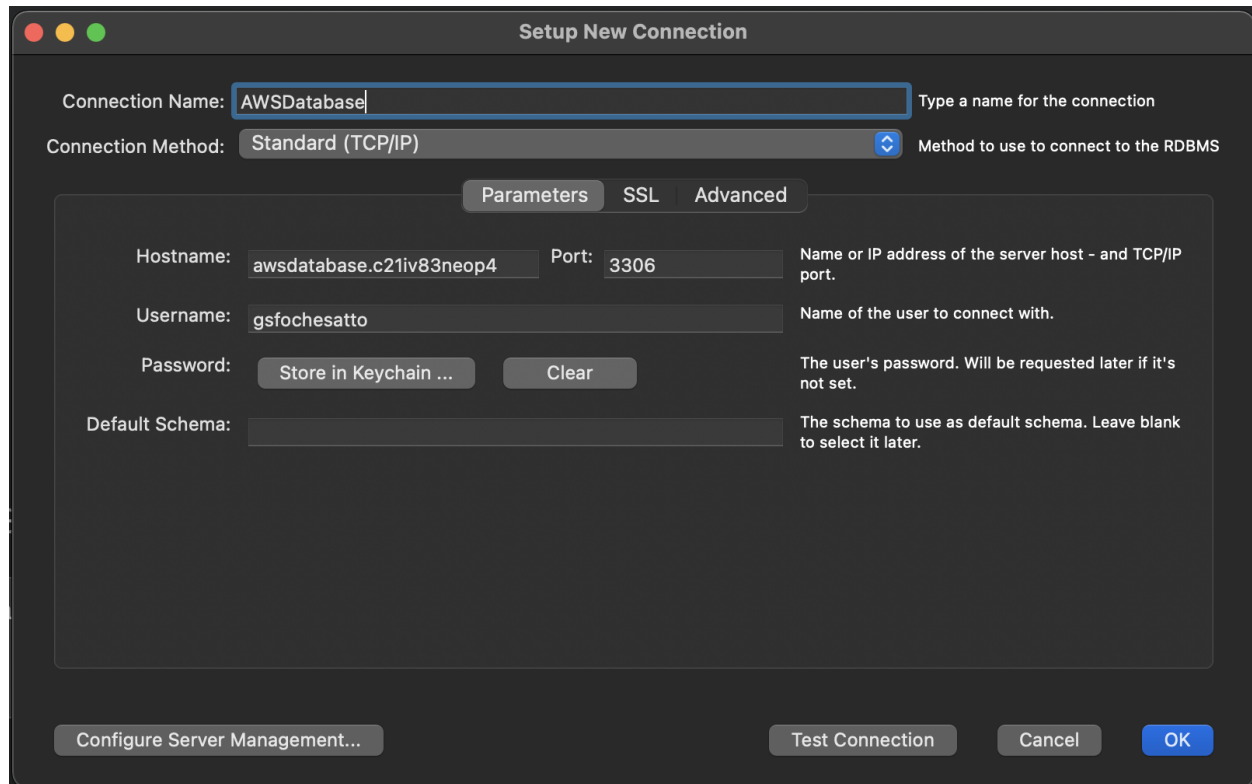
Add rule

Outbound rules [Info](#)

Type Info	Protocol Info	Port range Info	Destination Info	Description - optional Info	
<div>All traffic</div>	<div>All</div>	<div>All</div>	<div>Custom</div> <div><input type="text" value="0.0.0.0/0"/></div>	<input type="text"/>	<div>Delete</div>

Now we are ready to connect to the server using MySQL Workbench. On startup we are met with a menu of connections, to create a new connection we just hit the plus icon and we get the following menu,

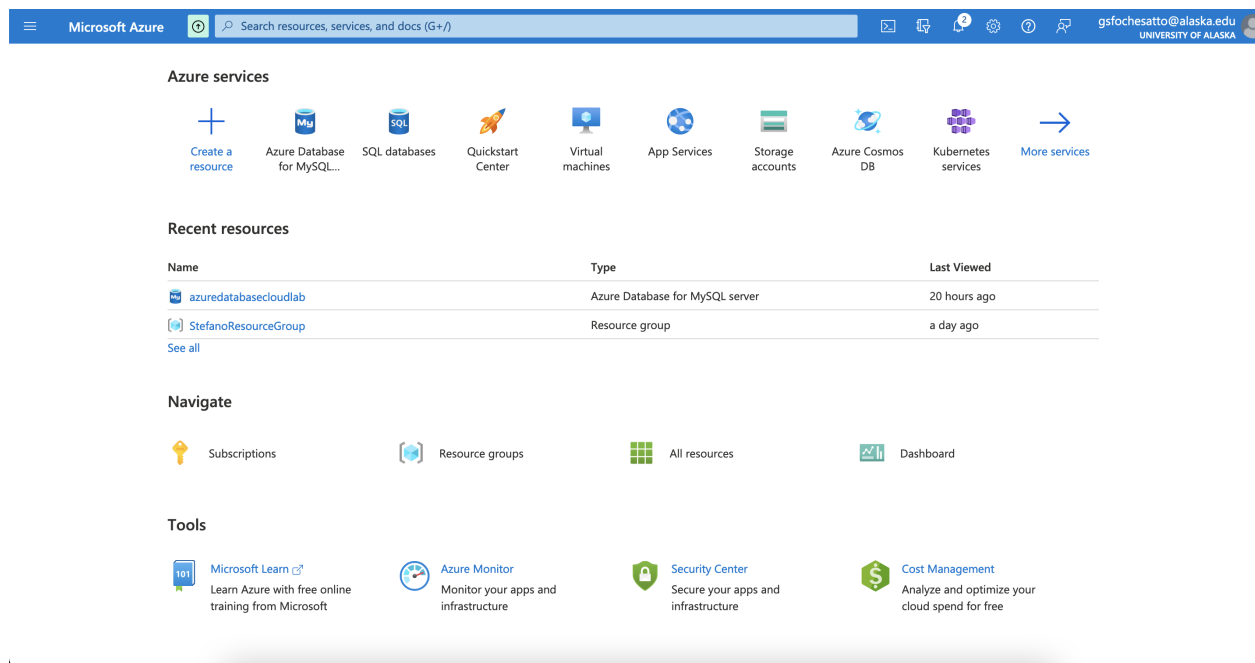
FIGURE 8. MySQL Workbench New Connection Menu



What is most important here is to set the "Hostname" to the endpoint address that we saw in the RDS Instance Dashboard, and set the username and password to the master username and password we decided when creating the RDS instance. After that we should be able to connect to the server.

Microsoft Azure. After creating a Microsoft Azure account we are met with the following dashboard.

FIGURE 9. Azure Main Dashboard



From here we will want to search "Azure Database for MySQL servers" open the service and then select create.

FIGURE 10. Azure Create Database Menu

Create MySQL server

Microsoft

Basics Additional settings Tags Review + create

Create an Azure Database for MySQL server. [Learn more](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Resource group * ⓘ
[Create new](#)

Server details

Enter required settings for this server, including picking a location and configuring the compute and storage resources.

Server name * ⓘ

Data source * ⓘ ☒ None ☐

Location * ⓘ

Version * ⓘ

Since we have a simple use case we'll just use the single server service. Under the project detail menu we can specify subscription and billing details, Under the server details menu we can specify it's name, location, MySQL version (we will use ver.8.0), and compute and storage capabilities.

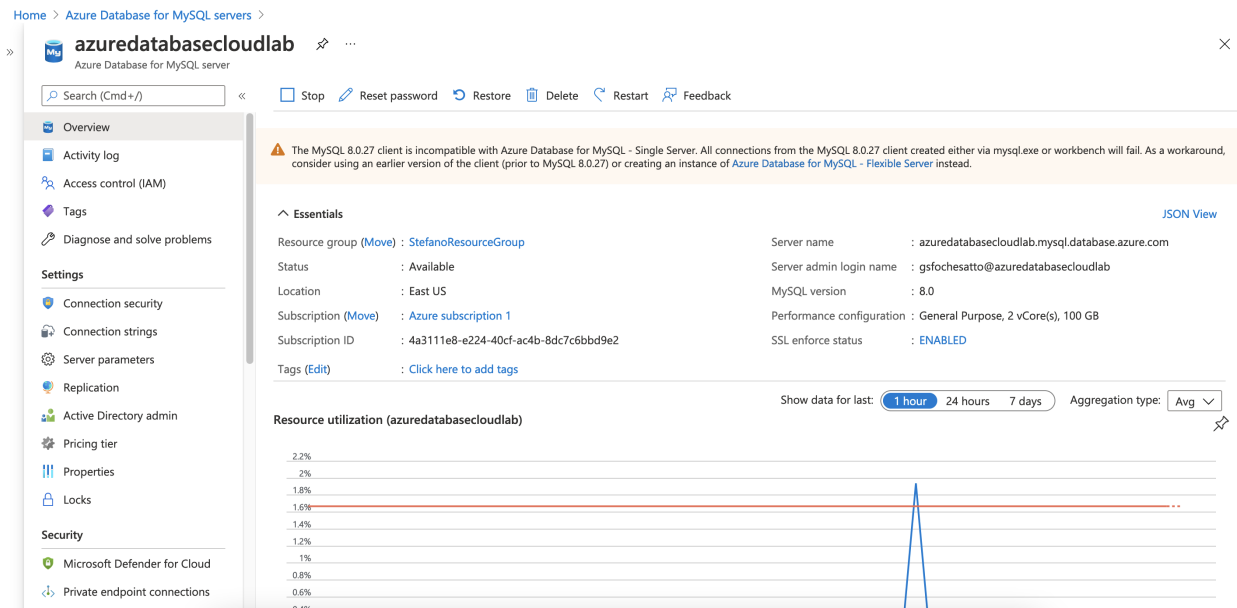
FIGURE 11. Azure Create Database Menu

Administrator account

Admin username * ⓘ	<input type="text" value="Enter server admin login name"/>
Password * ⓘ	<input type="password" value="Enter password"/>
Confirm password *	<input type="password" value="Confirm the above password"/>

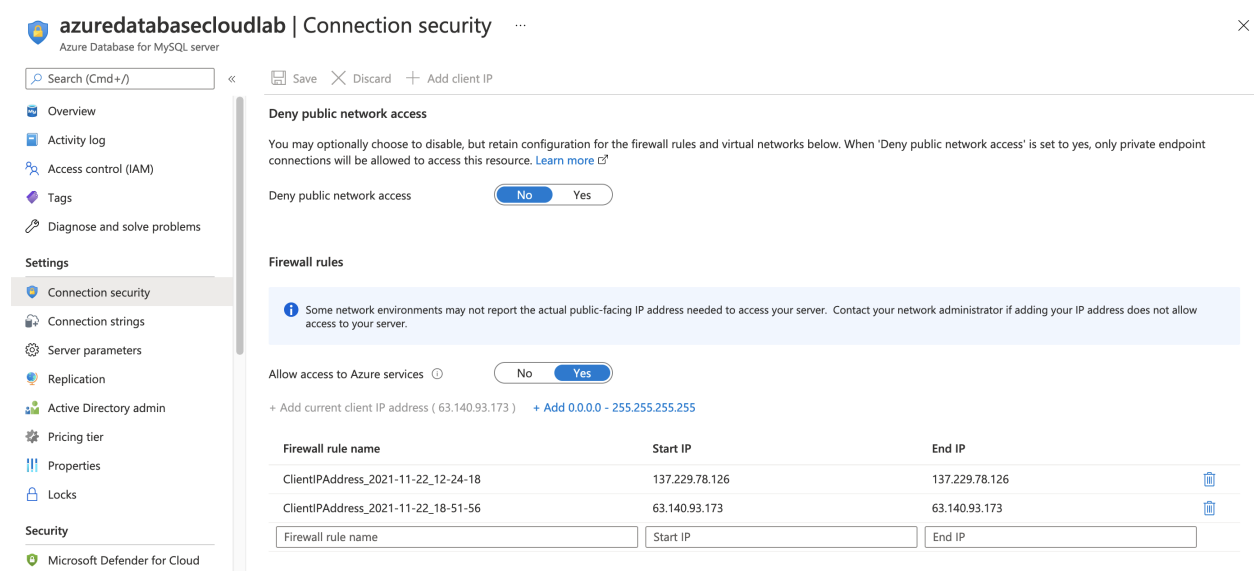
The administrator account menu is important as we will use these login details to gain access to the server. With our server configured we can click create and we are met with the following dashboard.

FIGURE 12. Azure Database Dashboard



To connect to the server we must first configure the security settings. To do so click on 'connection security' on the left hand menu.

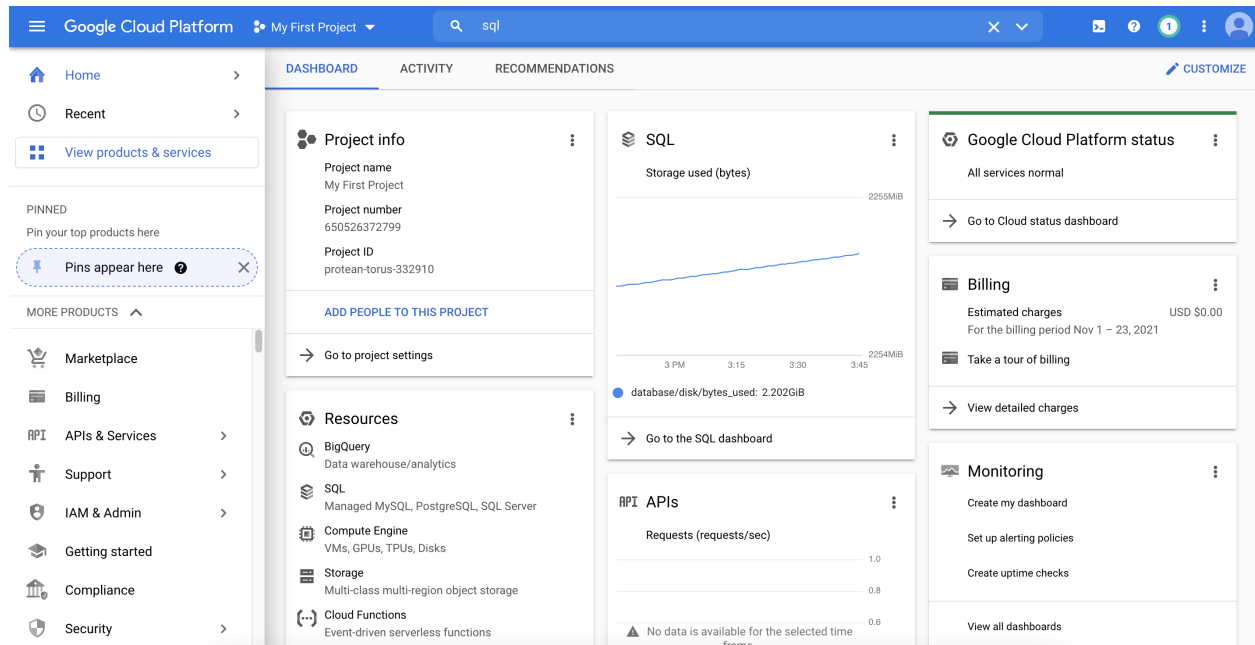
FIGURE 13. Azure Connection Security Menu



Here all we have to do is make sure that 'Deny Public Network Access' is set to no, 'Allow access to Azure services' is set to yes, and click on 'Add Current Client IP Address'. Now we should be ready to login in to the server in MySQL Workbench. To do so we proceed as before but this time we set the "Hostname" to the Server name in the Azure Dashboard, and the 'Username' to the Server admin login name, which also in the Azure Dashboard and the password is whatever was set in the administrator account menu.

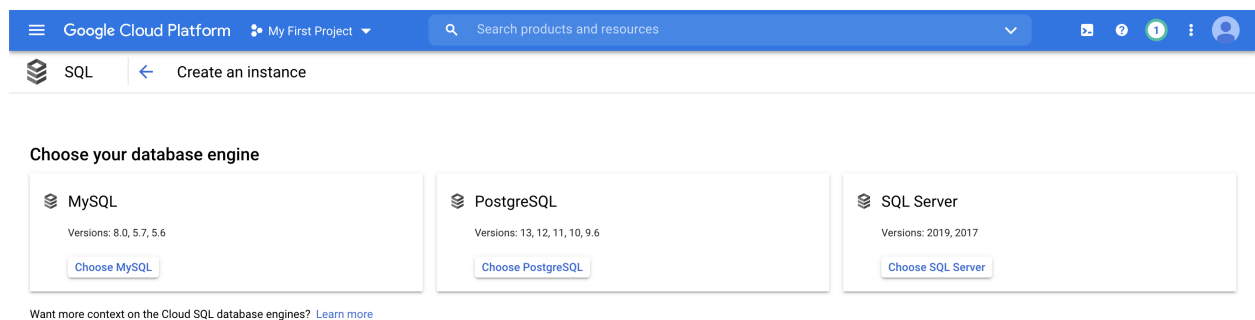
Google Cloud Platform. After creating a Google Cloud Platform account we are met with the following dash.

FIGURE 14. Google Cloud Platform Main Dashboard



From here we can search 'SQL' to find the Google SQL service, and continue to create and instance.

FIGURE 15. Google Cloud Platform Create Database Menu



Here we are prompted with what type of database we want to create, for our project we will select MySQL. The following menu allows us to customize our MySQL instance, generally we want to keep track of the Instance ID and Password.

FIGURE 16. Google Cloud Platform Create Database Menu

Google Cloud Platform

My First Project

Search products and resources

Create a MySQL instance

Instance info

Instance ID *

Use lowercase letters, numbers, and hyphens. Start with a letter.

Password *

GENERATE

Set a password for the root user. [Learn more](#)

☐ No password

Database version *

MySQL 8.0

Choose region and zonal availability

For better performance, keep your data close to the services that need it. Region is permanent, while zone can be changed any time.

Region

us-central1 (Iowa)

Zonal availability

☐ Single zone

In case of outage, no failover. Not recommended for production.

☒ Multiple zones (Highly available)

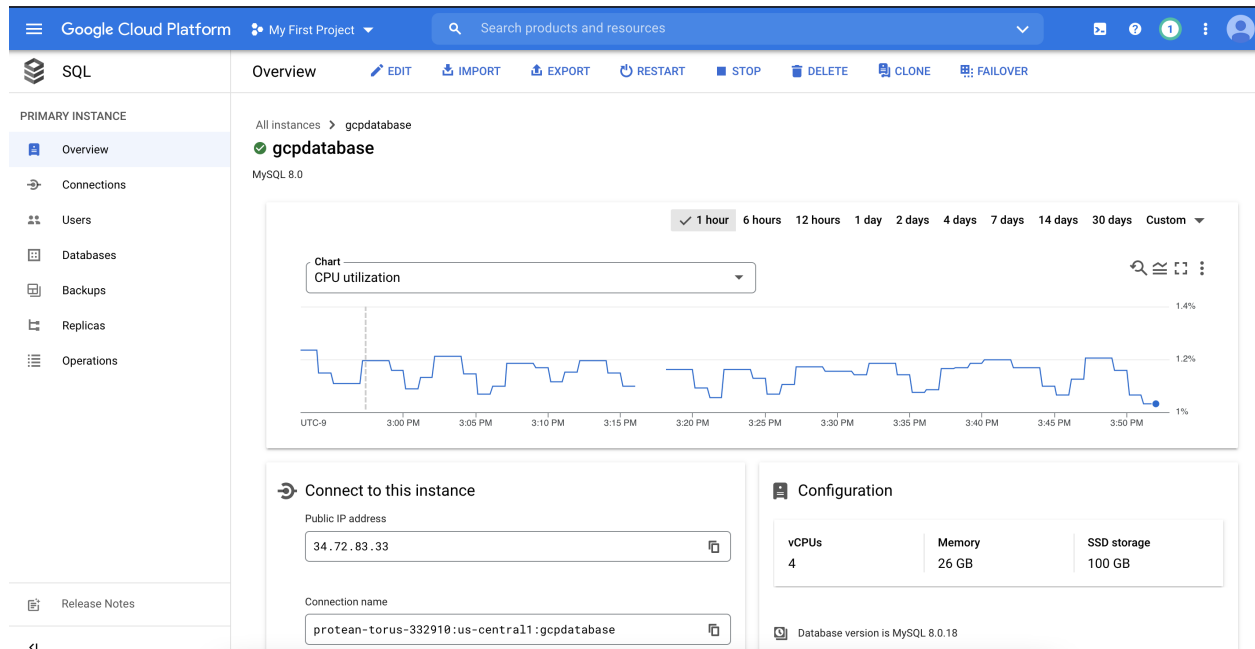
Automatic failover to another zone within your selected region. Recommended for production instances. Increases cost.

Summary

Region	us-central1 (Iowa)
DB Version	MySQL 8.0
vCPUs	4 vCPU
Memory	26 GB
Storage	100 GB
Network throughput (MB/s) ?	1,000 of 2,000
Disk throughput (MB/s) ?	Read: 48.0 of 240.0 Write: 48.0 of 240.0
IOPS ?	Read: 3,000 of 15,000 Write: 3,000 of 15,000
Connections	Public IP
Backup	Automated
Availability	Multiple zones (Highly available)
Point-in-time recovery	Enabled

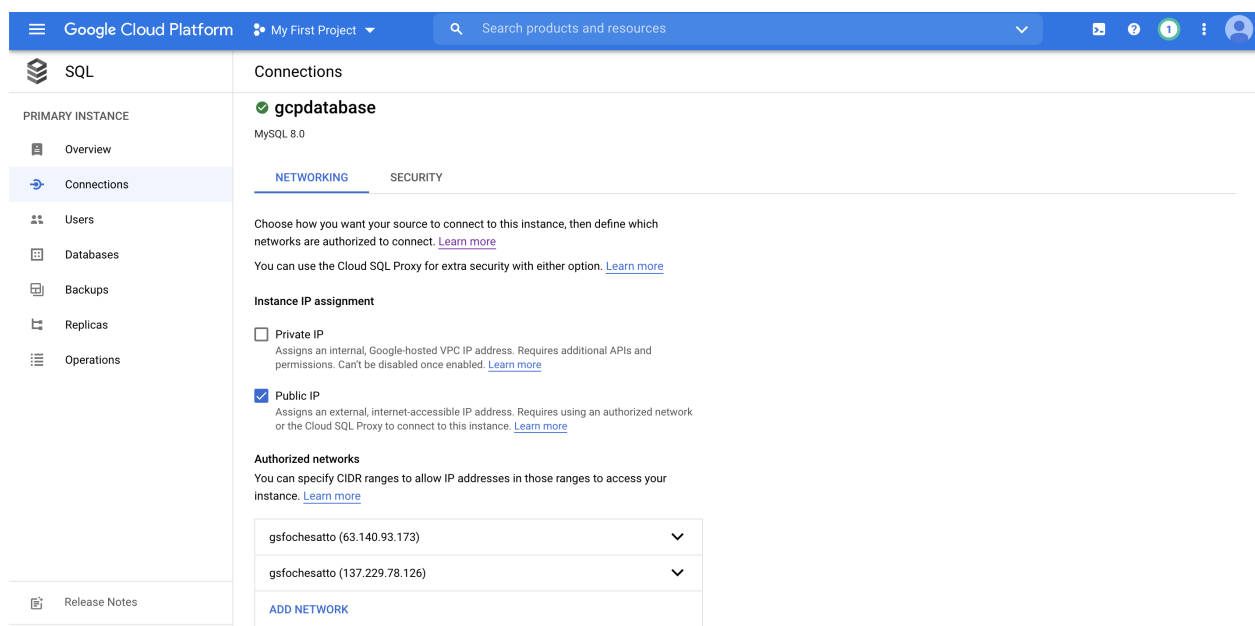
Creating our instance we see the following dashboard.

FIGURE 17. Google Cloud Platform Database Dashboard



Before we can connect to the server we need to change the security settings. To do so just click on connections in the left side menu.

FIGURE 18. Google Cloud Platform Security Settings Menu



In the connections menu we need to set 'Instance IP Assignment' to Public IP and then under Authorized Networks we need to add a network with our current IP address. Now we should have access to the server. To connect to the server in MySQL Workbench we just create a new connection with the 'Hostname' set to the public ip address, the 'Username' set to root and the password set to whatever we decided when we created the MySQL instance.

Now we have successfully initialized and connected to 3 different MySQL servers on 3 different cloud platforms.

Loading Data to MySQL Servers. So we now are connected to 3 different MySQL servers, but they are empty and have no data. Luckily since they all use the same management system(MySQL) we can use the same SQL script to load the data from the csv that we exported earlier on all 3 servers. Consider the following script,

Code:

```
## Create Database Using MySQL workbench GUI or with the following command.
CREATE DATABASE FireData;

## We won't use almost all of these entries but they are the full data.
CREATE TABLE `FireData`.`Fires` (
  `OBJECTID` int,
  `FOD_ID` int,
  `FPA_ID` text,
  `SOURCE.SYSTEM.TYPE` text,
  `SOURCE.SYSTEM` text,
  `NWCG.REPORTING.AGENCY` text,
  `NWCG.REPORTING.UNIT.ID` text,
  `NWCG.REPORTING.UNIT.NAME` text,
  `SOURCE.REPORTING.UNIT` text,
  `SOURCE.REPORTING.UNIT.NAME` text,
  `LOCAL.FIRE.REPORT.ID` int,
  `LOCAL.INCIDENT.ID` text,
  `FIRE.CODE` text,
  `FIRE.NAME` text,
  `ICS_209.INCIDENT.NUMBER` text,
  `ICS_209.NAME` text,
  `MTBS.ID` text,
  `MTBS.FIRE.NAME` text,
  `COMPLEX.NAME` text,
  `FIRE.YEAR` int,
  `DISCOVERY.DATE` double,
  `DISCOVERY.DOY` int,
  `DISCOVERY.TIME` int,
  `STAT.CAUSE.CODE` int,
  `STAT.CAUSE.DESCR` text,
  `CONT.DATE` double,
  `CONT.DOY` int,
  `CONT.TIME` int,
  `FIRE.SIZE` double,
  `FIRE.SIZE.CLASS` text,
  `LATITUDE` double,
  `LONGITUDE` double,
  `OWNER.CODE` int,
  `OWNER.DESCR` text,
  `STATE` text,
  `COUNTY` int,
  `FIPS.CODE` text,
  `FIPS.NAME` text,
  `Shape` text);
```



```
## Mount Database and load data from csv.
USE FireData;
LOAD DATA LOCAL INFILE  '/Users/stefanofochesatto/Desktop/Fires.csv' IGNORE
INTO TABLE Fires
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n';

## Create Index for FIRE_YEAR which will be used by our web app to query results faster.
CREATE INDEX FireYearIndex ON Fires (FIRE_YEAR);

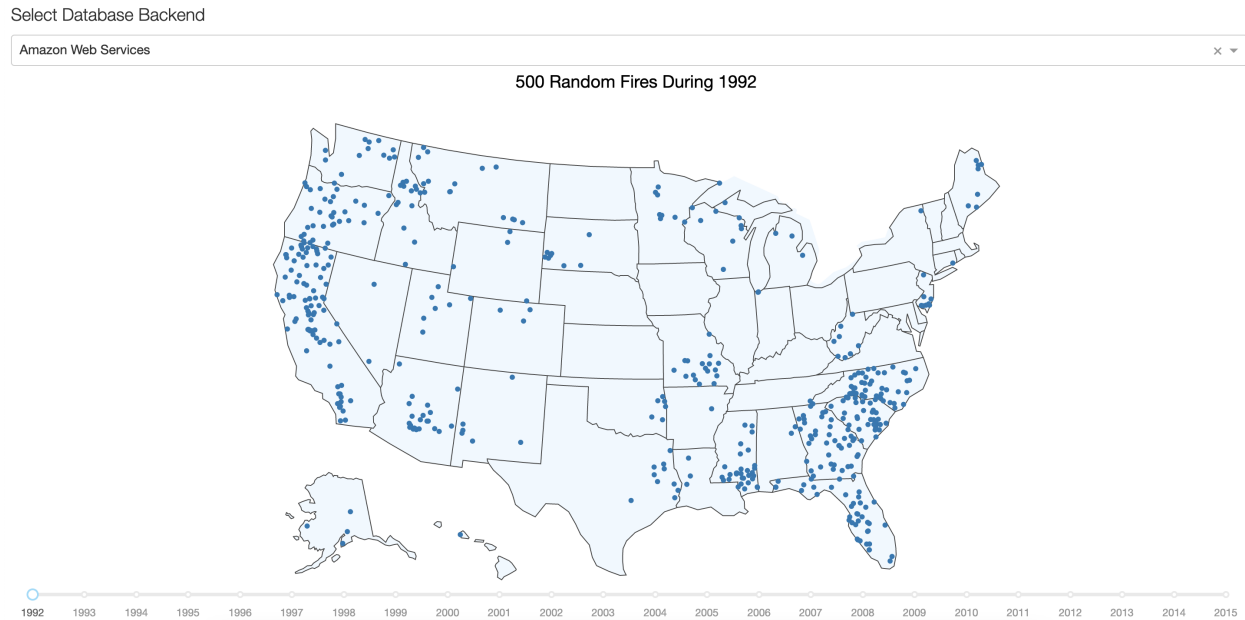
## Demonstrating Querying data with query that will be used in web app.
select FIRE_YEAR,LATITUDE, LONGITUDE from
    Fires use index(FireYearIndex) where FIRE_YEAR = 2005 ORDER BY RAND() limit 500;
```

This script first creates a database called FireData, and a table called Fires. It then mounts the FireData database and loads in the csv file from my local machine (This part takes the longest since we need to upload the data to each server). Then it creates an index called FireYearIndex on the FIRE_YEAR column of data, we need this since our web app will display fires, using latitude and longitude by year and creating an index like this makes the queries much faster. The final line is an example of the type of query we will be making in our web app.

4. DASH APP AND QUERYING IN PYTHON

So now we should have our databases setup up with our data, we can write queries to them using MySQL Workbench and use them to store and serve data to collaborators. The rest of this lab we will be exploring how we can programmatically query these databases using python, in the form of a data visualization web application. The web app I built for this lab can be found [here](#) and it simply plots the location of 500 random fires given a year. It also allows us to select which cloud platform we are querying our data from. Here a screenshot of the application,

FIGURE 19. Lab 3 Web Application



For the rest of this section, I will go through the web application code, block by block and explain what is going on. The first block of code defined the application dependencies. I used the Dash library to build the application and the SQLAlchemy library to query data from the servers. As an introduction, all we need to know is that Dash is a high level javascript wrapper, that is why you'll see elements of html and css throughout the code. The last block just deals with setting up the sql login credentials as environment variables, this is necessary to share the code without exposing my login information.

Code:

```
## Data IO\Manipulation libraries
import pandas as pd
import numpy as np
from sqlalchemy import create_engine

## Dashboard Libraries
import dash
from dash import dcc
from dash import html
from dash.dependencies import Input, Output
```

```
import plotly.graph_objs as go
from os import environ

AWSKEY = str(environ.get('AWSENGINE'))
AZUREKEY = str(environ.get('AZUREENGINE'))
GCPKEY = str(environ.get('GCPENGINE'))
```

The next block of code is primarily used to define the layout of the web application. We can see that the first line of code is used to bring in a css style sheet, this really just changes the fonts used in the application. The rest of this code block is for defining the layout of the application and the different components i.e. dropdown menu, geo-scatter plot, and year slider. We also initialize our connection to the AWS server(it is the default when loading the page) using SQLAlchemy's createngine command.

Code:

```
## CSS Style Sheet for Dash Components
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css ']

## Init Dash App
app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
server = app.server

## Make connection to MySQL database, AWS by default
engine = create_engine(AWSKEY)

## dash app layout
app.layout = html.Div([
    ## Database dropdown menu
    html.Div(
        [
            html.H6(""" Select Database Backend""",
                    style={'margin-right ': '2em'}),
        ],
    ),
```

```

    dcc.Dropdown(
        id='demo-dropdown',
        options=[
            {'label': 'Amazon Web Services', 'value': 'AWS'},
            {'label': 'Microsoft Azure', 'value': 'AZR'},
            {'label': 'Google Cloud Platform', 'value': 'GCP'}
        ],
        value='AWS'
    ),

    ## Graph
    dcc.Graph(
        id='map-with-slider',
        style={'width': '100vw', 'height': '80vh', 'textAlign': 'left', 'font-weight': '300'}
    ),

    ## Year Slider
    dcc.Slider(
        id='year-slider',
        min = 1992,
        max = 2015,
        value = 1992,
        step = 1,
        marks = {year: str(year) for year in range(1992,2016)}
    ),
    dcc.Store(id='intermediate-value')
])

```

The last block of code defines all our callback functions, which make the web application interactive. The first call back function is called `change_backend` and it is used to defined the behavior of the dropdown menu. When you select a different cloud service in the dropdown menu, the `change_backend` function will initialize the connection to one of the three servers using the `createengine` command we mentioned before. You can see this in the block of `else if` statements.

The second callback function is called `display_map`, and it is used to update the map whenever

the server dropdown menu or year slider are interacted with. The most important part of this code is where we initialize the query string, and then pass it through read_sql function to create a pandas dataframe from our sql query. The rest of the function involves updating the map with the queried data.

Code:

```

### Callback for Database menu
@app.callback(Output('intermediate-value', 'data'), [Input("demo-dropdown", "value")])
def change_backend(value):
    # reference global engine object
    global engine
    # if statement to change databases
    if (value == 'AWS'):
        engine = create_engine(AWSKEY)
    elif (value == 'AZR'):
        engine = create_engine(AZUREKEY)
    elif (value == 'GCP'):
        engine = create_engine(GCPKEY)

    return 1

### Callback for Year slider
@app.callback(
    Output("map_with_slider", "figure"),
    Input("intermediate-value", "data"),
    Input("year_slider", "value"))
def display_map(extra, year):

    ##### This is the most important part.
    query = 'select FIRE.YEAR,LATITUDE,LONGITUDE from Fires use
    index(FireYearIndex) where FIRE.YEAR = {} ORDER BY RAND() limit 500'.format(year)

    with engine.connect() as connection:
        result_dataframe = pd.read_sql(query, connection)

```

```
##### We right the query, send it the the server and put the response in dataframe.
```

```
data = [
    go.Scattergeo(
        lat = result_dataframe['LATITUDE'],
        lon = result_dataframe['LONGITUDE']
    )
]

layout = go.Layout(
    title = dict(
        text= '500 Random Fires During {}'.format(year),
        font=dict(
            family="Helvetica Neue",
            size=20,
            color='#000000'
        )
    ),

    margin = dict(
        autoexpand=True,
        l=0, #left margin
        r=0, #right margin
        b=0, #bottom margin
        t=50 #top margin
    ),

    geo = dict(
        scope='usa',
        landcolor='aliceblue',
        projection=dict(type='albers usa'),
        showland = True,
    )
)

return {'data':data, 'layout':layout}
```

5. CONCLUSION

This lab felt incredibly challenging and rewarding. Before we started the module on relational databases I wasn't very familiar with SQL and dynamic web applications. I learned a whole variety of new tools throughout this lab, like MySQL Workbench, AWS, Microsoft Azure, and Google Cloud Platform. I had used Dash before for small presentations and demos, but nothing that emulated a full web application with a separate front and back end. I struggled a lot with trying to convert the sqlite database into one that could be used in MySQL, and in the end I found a workaround because all the data was stored in a single table in the sqlite database. I was also worried with the performance of the web application, for a long time I didn't have an index for the Fire_Year which was making the queries take several minutes. I think in the future I would like to work with a database with a more complicated schema, and maybe even a different engine like PostgreSQL. As always here is a link to the data we used in the lab <https://www.kaggle.com/rtatman/188-million-us-wildfires>. The web application can be found here, <https://exploringclouddatabases.herokuapp.com/> and code for the web application can be found on my github <https://github.com/StefanoFochesatto/ExploringCloudDatabases>.