Final Report

Thomas Gourley
Robert Upton

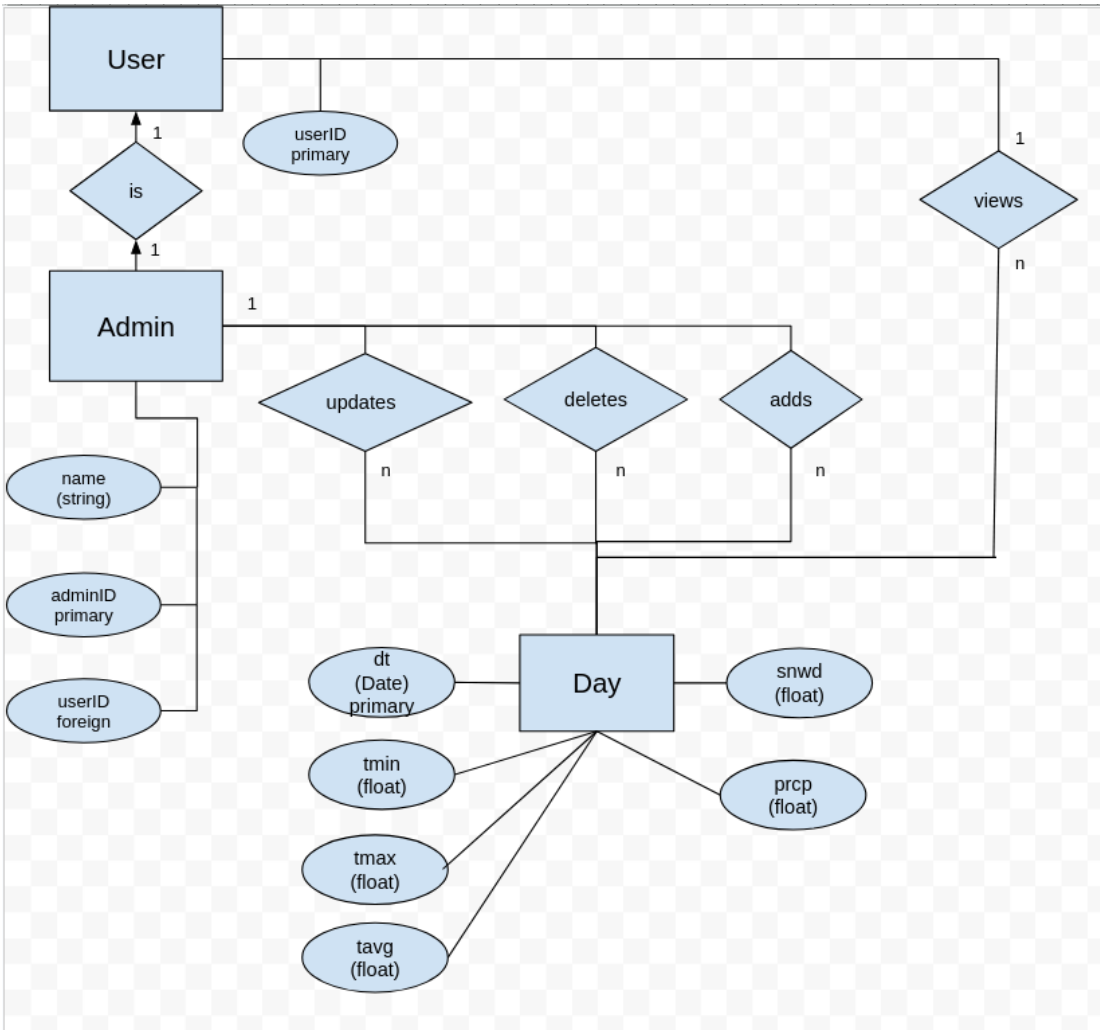
Introduction

Our application revolves around the retrieval, processing and visualization of weather data from Base Orcadas, a scientific station in Antarctica managed by Argentina and distinguished as the oldest operational station in the region. Originally intending to develop a Mars weather tracking application, we encountered roadblocks, particularly in data availability. The Mars Insight rover has become inoperable due to Martian dust covering its solar panels, and as a result, its associated API data became outdated following the rover's shutdown in December 2022. Further complicating matters, the API we found to gather the Curiosity rover's measurements was later discovered to be unreachable. In response, the focus was instead shifted to tracking Antarctic weather using a government-sponsored API provided by NOAA and NCEI.

The National Centers for Environmental Information (NCEI) hosts a free API to access their weather data, which includes data from hundreds of weather stations across the world and, in some cases, data available as far back as the mid-1700s. Base Orcadas was chosen because it has the highest data coverage (91%) with the longest time scale (from Jan 1st, 1957 to Nov 2023) out of the all other Antarctic weather stations found through NCEI. The API for this base provides minimum temperature, maximum temperature, average temperature, precipitation, and snow depth for each day. By plotting Antarctic weather data, we hope to provide a unique tool to visualize long-term weather and overall climate trends at one of Earth's most important environments.

Database Details

The Entity-Relationship (ER) model illustrates the relationships among entities in our database. It adheres to the principles of third Normal Form (3NF) by satisfying the requirements of 2NF and ensuring the absence of transitive dependencies in the diagram. We have ensured the presence of primary keys for each table and incorporated foreign keys where necessary to establish meaningful relationships between tables.



The diagram comprises entries such as "Day," intended for holding information encompassing date, high and low temperatures, as well as precipitation details for a given day and snow depth.

The database schema exhibits a comprehensive structure that defines relationships between Users, Admins, and Days. Users maintain one-to-many relationships with Days, allowing them to associate with multiple instances of each time unit. Admins, inheriting search and view capabilities from Users, additionally possess the authority to create new entries, delete records, and modify existing data. Admins also engage in one-to-many relationships with Days.

Functionality Details

Our application offers a suite of fundamental features, including record insertion, search and listing capabilities, record updates, and deletion. During record insertion, the function seamlessly integrates weather data from the API, empowering database administrators to add new days with corresponding weather details. Users can explore the database for weather information, revealing daily high, low, and average temperatures, precipitation, and snow depth using search and listing operations. The intriguing queries function enables advanced searches,

allowing users to filter days based on specific data types, such as identifying the coldest days of winter for a particular year. Record updates empower database administrators to refresh existing records with new or revised weather data for a specific day, including the addition of new days. The deletion feature allows administrators to remove arbitrary weather data for existing days or delete the entire record for specific days.

The advanced functionalities encompass weather data collection and weather data trend visualization. Weather data collection involves making requests to the weather API to gather new data when adding a new day entry into the database. The weather data trend visualization function graphs temperature, precipitation, and snow depth over a specified time scale, providing users with a visual representation of changes over time.

Implementation Details

For the final implementation of the app, MySQL was used for the database and a combination of HTML and CSS was used for the web frontend. The Python library Flask was used to serve the webpages as HTTP endpoints and respond to user interaction, while the MySQLConnector library was used to manage queries to the database. These also handle user and admin logins. Generation of the data plots was done with the Python library matplotlib, and this was also displayed to a webpage by using Flask and HTML.

The first step in development was acquiring the data, which was done through the NCEI weather data API. To interact with the NCEI API, an API token was obtained from their website and a url was crafted which contained start date, end date, datatype, and station id arguments. A single request can be made through the command line using `curl`:

```
curl -X GET -H "token:<your-token-here>"
"https://www.ncei.noaa.gov/cdo-web/api/v2/data?datasetid=GSOM&stationid=GHCND:AYM00088968&units=standard&startdate=1957-01-01&enddate=1967-01-01&limit=1000"
```

and the API will respond with a JSON object which can be saved to a file. In this case, the large number of requests to be sent warranted the use of Python and the requests library, and the responses were saved in a JSON file for their corresponding year.

```python
args["startdate"] = f"{year}-01-01"
args["enddate"]   = f"{year}-12-31"
args["offset"] = "0"
req1 = requests.get(url, params=args, headers=headers)

# Sleep 1 sec to prevent more than 5 requests from ever happening in 1 sec
time.sleep(1)

args["offset"] = "1001"
req2 = requests.get(url, params=args, headers=headers)
```

`download_data.py`

Once the data for each year was downloaded, the datatypes needed to be grouped into their corresponding days. Another Python script was written to iterate through each file, grouping individual tmin, tmax, etc responses by their date since the date acts as the unique primary key for a single Day entity. In order to insert each day with the day's high temperature,

low temperature, average temperature, precipitation, and snow depth into the database, SQL code had to be generated with Python because of the size of this dataset. In total, 23,869 lines of SQL code were generated, and this `generated_sql.sql` file was imported into MySQL to populate the database.

```
82      code = open("generated_sql.sql", "a+")
83      code.write(f"-- {year}\n")
84      #last_post = code.tell()
85      code.write("INSERT INTO Days(dt, tavg, tmin, tmax, prcp, snwd) VALUES\n")
86
87      keys_list = result.keys()
88      for k in keys_list:
89          tavg = result.get(k).tavg if (result.get(k).tavg != None) else "NULL"
90          tmin = result.get(k).tmin if (result.get(k).tmin != None) else "NULL"
91          tmax = result.get(k).tmax if (result.get(k).tmax != None) else "NULL"
92          precip = result.get(k).prcp if (result.get(k).prcp != None) else "NULL"
93          snowd  = result.get(k).snwd if (result.get(k).snwd != None) else "NULL"
94
95          ch = ""
96          klist = list(keys_list)
97          if(klist.index(k) == len(klist)-1):
98              ch = ";"
99          else:
00              ch = ","
01          code_str = f"('{k}', {tavg}, {tmin}, {tmax}, {precip}, {snowd}){ch}\n"
02          code.write(code_str)
```

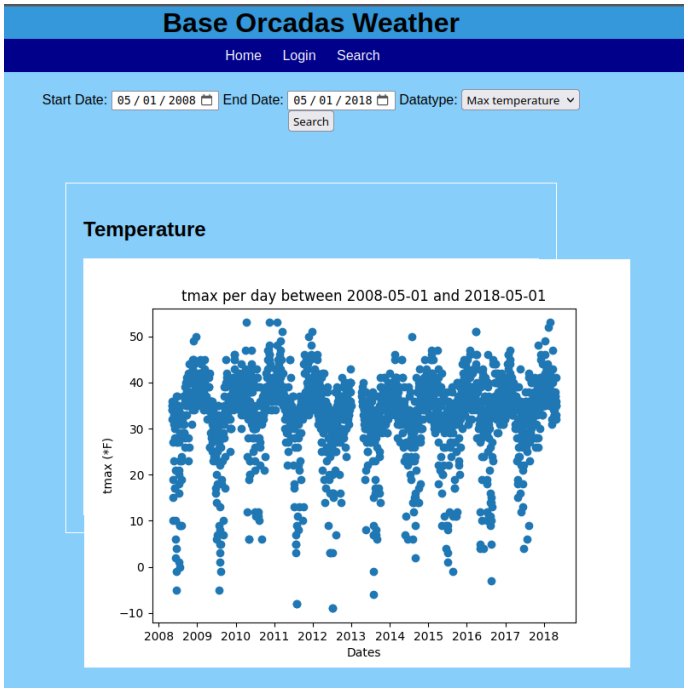
`populate_db.py`

```
23866  ('2023-11-07', 32.0, 30.0, 33.0, NULL, 27.6),
23867  ('2023-11-08', 32.0, 30.0, NULL, 0.0, 28.3),
23868  ('2023-11-09', 30.0, 24.0, 37.0, 0.0, 27.6),
23869  ('2023-11-10', 32.0, 28.0, NULL, NULL, 27.6);
23870
```

`generated_sql.sql`

After populating the Days table, Users and Admins tables were created according to our ER diagram.

At this point we could now use Flask and the MySQLConnector libraries to connect a web frontend to the database backend. We used CSS to style HTML templates for each webpage, and each webpage was treated as an HTTP endpoint by Flask. Flask uses an HTML template designer called Jinja to conditionally display Python variables on a webpage. These parts all work together to implement our advanced functions. A user first logs into the system. If the user's ID and username match a valid pair in the database, they are redirected to the search page. The user specifies a date range and datatype (avg temp, min temp, max temp, precip, or snow depth) to search the database. MySQLConnector searches the database and if successful, a plot of data vs. time is generated with matplotlib and passed back to the webpage for display.



Basic user search and advanced plotting functionality

Experiences

The first major hurdle in the development of this project was acquiring the data. The initial idea was to do a NASA-style Mars weather tracker, however we discovered rather late that this would not be possible. The problem was that the APIs were either completely unreachable (they would not respond to a request with any data) or the latest data they had was years old. This resulted in us having to change the project idea at Stage 3, and we settled on an Antarctica weather tracker due to similarity to the original idea, being a unique take on weather data visualization, but now possible because of the availability of data from government weather stations and APIs. A further objective of this was to see if an obvious trend could be found in temperature or climate change over the years.

The large dataset also proved to be a challenge, including the format our chosen API decided to return our data. A single Day in our database relates a Date and multiple datatype columns, with each type returned by the API in an individual JSON object. These min temp, max temp, etc JSON objects are marked with their date, type, and value and collected in a list of results to make up a single response for a request. Couple this multiple-JSON-objects-per-day issue with the need to make two requests per year (only a maximum of 1000 data points can be returned from any one request) and the sheer volume of data from every day between Jan 1st, 1957 to Nov 10th, 2023 and the challenge of organizing the data to fit our design becomes clear. Once a script was made to download the data for each year, another one had to be made to parse these files and organize the data in such a way to fit according to the Days relation found in our ER diagram. Only *then* could we populate the database, but of course it would not

be possible to manually populate a database with 23,000+ days, so SQL code to insert these days into the database was generated in Python.

The final hurdle was in the connection between backend and frontend. The group determined early on that we were going to develop a web-based interface for the application, and this plan was not to be scrapped like the others. Initially the idea was to use mysqlconnector and a Javascript framework called pyscript to run Python code in the browser. The reasoning was that this would effectively allow us to reuse the Python code from the plotting demo shown during the presentation, only now running in the browser. But this was not the case. After much research it was discovered that there is a conflict between the MySQL server connection and the server running our website, and due to security reasons the MySQL server connection initiated inside the browser is also blocked by the browser. This issue led us to explore Flask, which required much modification of our existing HTML to fit the endpoints that were needed, and a lot more Python code had to be written, but this solution ultimately proved to be a much better and less “hacky” solution than the previous.

Overall, this project has been an incredible learning opportunity. We learned how to interact with external APIs, gained further experience in problem-solving and troubleshooting obscure errors, and learned how to operate between frontend and backend code. In the future, the project could be enhanced by automatically updating the database when a new day is recorded by NCEI. The website is also still vulnerable to SQL injection, which could be dangerous for a web app based around a large dataset. Furthermore, a very simple and useful feature would be the ability to customize the user’s search operation, for example plotting all the coldest days of the year for a given time range. This is already possible in the backend Python code, but unfortunately has yet to be implemented in the frontend.

References

Flask Quickstart guide
https://flask.palletsprojects.com/en/3.0.x/quickstart/

Jinja Template Designer
https://jinja.palletsprojects.com/en/3.1.x/templates/#

W3Schools SQL
https://www.w3schools.com/SQL/

W3Schools HTML
https://www.w3schools.com/html/default.asp

Matplotlib
https://matplotlib.org/stable/

NOAA NCEI
https://www.ncei.noaa.gov/