**Extract, Transform, and Load Project**

**Daniel Furman, Engy Samaan, Todd Wintering**

**Introduction:**

For our project we decided to examine the weather conditions of aerial bombings during the Second World War. From the harsh Russian winter that halted the German invasion, to the planning of the D-Day landings, weather conditions played a pivotal role in the course of the conflict. This was especially true in the case of aviation. To help investigate its effects, we gathered data detailing every known allied bombing mission, including the takeoff and target locations as well as the types of aircraft and weapons employed, and combined them with weather reports from observation stations throughout both theaters of war.

This report details our efforts in extracting, transforming, and loading this data into a MySQL database so as to facilitate further research and analysis.

**Extract:**

Our bombing data comes from the Theater History of Operations (THOR) dataset. THOR contains records of all allied bombing missions from World War I through Vietnam. Our data originates from paper mission reports filed from 1939 to 1945. It comprises missions flown by the United States, Great Britain, Australia, New Zealand, and South Africa. We found this data in a collection hosted at Data World. (See <https://data.world/datamil/world-war-ii-thor-data>).

Our weather data comes from a dataset complied by the National Centers for Environmental Information. We retrieved it from a collection hosted at Kaggle. (See <https://www.kaggle.com/smid80/weatherww2>).

We began by downloading both datasets in comma separated value (CSV) format. Using Jupyter Notebook, we extracted five CSV files into Pandas dataframes using the read\_csv function.

**Transform:**

The transformation process consisted of cleaning the data using Pandas and Python. The first problem we encountered was that the raw dataframes generated from the CSV files contained many extraneous and mostly blank columns. For example, our dataframe on aerial missions from THOR started with 62 columns with hundreds of thousands of NaN values. We pared our frames down by dropping unnecessary columns and only retaining fields useful for analysis.

Another issue we came across was that our source data used different date formats. Since we eventually wanted these columns to be joined in a relational database, we used Python string handling to create a consistent format. This involved splitting dates with the "-" delimiter into year, month, and day variables and the inserting a zero before single-digit days and months. Similarly, we had to use the upper() method to ensure that weapon names and descriptions matched amongst our various dataframes.

We added an "\_id" column to each dataframe to eventually serve as a primary key in a relational database. We also changed several column names to be consistent between tables to make eventual SQL joins easier.

To detect missing data in Pandas we used value\_counts() and isna().sum(). Where appropriate we used the fillna() function to supply missing values. For example, on days where it did not snow we filled our Snowfall column with the value of zero. To avoid potential incompatibilities with MySQL, Pandas "NaN" values were replaced with empty strings to generate SQL nulls in the loading phase. (These would not be automatically converted due to our need to return the data to CSV format as outlined below. Similarly, there was no sense in using Python None types as these would be converted to empty strings anyway.)

Finally, we saved our five cleaned dataframes in CSV format using the Pandas to\_csv function.

**Load:**

Our initial plan was simply to load our Pandas dataframes directly into MySQL using the to\_sql function, but this took too long, causing our server to timeout. We therefore had to find another way to get our data into a relational database. As mentioned above, we ended up exporting our dataframes to CSV files.

Our next task was to find a way to import our CSV files into our MySQL server. One possibility was to use the import wizard in MySQL Workbench, however, the wizard was prohibitively slow, and we would never have finished the project in time waiting for it to complete. We thus turned to the LOAD DATA INFILE command as a faster alternative. Unfortunately, the server we were using did not give us permission to load SQL from files, so we were forced to abandon this approach as well.

We finally settled on using Object Relational Mapping (ORM) and creating Python classes mirroring our desired relational tables, so that we could translate our data using the SQL Alchemy module. After defining our classes and creating the associated metadata, we looped through each row of our CSV files using the Python CSV module. We used indexing to select appropriate columns from each row and assigned their values to variables matching the column names in our classes, in order to create Python objects corresponding to the desired tuples in our database relations. At this point, we were successfully able to use SQL Alchemy session to add and commit each row to our server.

The result of our efforts is that the raw CSV finals are now tables in a MySQL database. They can now be joined and queried according to the relational model. For example, the date attribute can be used to join mission and weather tables to find the weather conditions of a particular bombing run.