ETL Project #2, Asylum Data

Kori Y. | Daniel C. | Brad P.

**Extraction**

Our dataset comes from a Kaggle user named Saurabh Patra  – providing data on applications and decisions of asylum seekers worldwide from the year 2000 to 2020. These datasets come in two separate .csv files (application file and decision file) coming in a structured format that contains information on country of origins, country of asylum, and the files’ respective unique data on the applications and decisions. A few things we noticed about the datasets upon the initial upload were unknown values from country of origin, majority of zero values in the column “Complementary protection”, and the “years” column of our applications file not in order which caused some initial misinterpretation during our planning stage. Despite early confusion, both .csv files contained the necessary information for us to proceed with our transformation phase.

**Transformation**

Upon extraction of the data, our plan involved the eventual merge of our two datasets into one through the use of a key that would merge on a form of concatenation between a country of origin, country of asylum, and the year. Additionally, we wanted to narrow our data frame’s scope to store data on asylum in the United States. This meant both of our data frames would only contain the United States as  “the country of asylum”. In order to get to this point, we began by cleaning both the application and decision data frames. For the application dataset, we re-labeled our country of origin and country of asylum headers in conjunction with our decision data frame to create a universal header that would provide us with cleaner references between the two. Once re-labeled, we decided to remove the ISO columns (the country’s abbreviation) for both the country of origin and asylum. We felt it was unnecessary space within the data frame and did not provide any insight into future analysis once stored. We repeated the cleaning of the headers for our decision data frame and kept its unique decision stats (recognized decisions, rejected decisions, etc.). Once cleaned, we then filtered out nations of asylum that weren’t the United States of America. In doing so, we had two clean data frames that only contained asylum seekers to the United States.

It was noted that our two data frames, however, had different lengths; our application data frame contained 191 rows while our decision data frame had 184. Although this doesn’t signal error in our transformation, we wanted to ensure our data contained the correct information and did not hold duplicate values. These values came from stateless and unknown country of origin which we deemed to be necessary information in our dataset. Despite not being labeled as a formal nation-state, it still provided insight as to who would be seeking asylum in the United States and could be easily removed - if necessary - in future analysis.

When we tried to combine the two data frames, we realized that there were several rows for the same country of origin seeking asylum in the US in the same year. The root cause was that there were several data entries for the same countries in the same years in the original dataset. As there was little information on Kaggle page explaining why that happened, we decide to take all the details into consideration instead of randomly removing certain repetitive values. So we have grouped the data frame by the “year”, “country of origin”, and “country of asylum” which in  turn gave us a more accurate total value.

**Load**

To load our dataset into SQL, we created a table in Postgres called us\_asylum and added columns to match the columns we had in our final dataframe from our jupyter notebook. We used the id column as our primary key. Since we already merged the two separate dataframes for “applied” & “decisions”  into one dataframe in our Jupyter notebook, we would not need to merge these later in SQL. As such, we only needed to create one table in our new database. Once we created the table, we then created a connection to our database in our jupyter notebook to our Postgres database and created an engine using this connection string. To confirm that we successfully connected, we input a command in the jupyter notebook to confirm the tables that we had available. Once we confirmed the table is the same as the one we created in Postgres, we started the merge of our dataframe into the SQL database from the Jupyter notebook using the engine that we previously created. At this point our dataframe should have successfully merged into our SQL database and to confirm this, we run a SQL query to show the table.