**Technical Write-up report for ETL-Project**

Downloaded Data from RedFin in CSVs.

Check for duplicates using MLS Listing as Primary Key, since that is unique

**For new construction listings, it didn’t have a MLS # so we deleted those from our dataset.**

<http://www.city-data.com/zips/94102.html>

94102, 94103, 94104, 94105, 94107, 94108, 94109, 94110, 94111, 94112, 94114, 94115, 94116, 94117, 94118, 94121, 94122, 94123, 94124, 94127, 94129, 94130, 94131, 94132, 94133, 94134, 94158.  
  
Read more: <http://www.city-data.com/zipmaps/San-Francisco-California.html>

Ran the data by zip code

**The sources of data that we extracted:**

For our project, we reviewed search engine result pages data related to travel. Specifically, our data consists of search terms, like “flights to hong kong” and the various websites, like Expedia & Travel Advisor, that had the quickest search result queries, which is under Search Time.

At first, we attempted to load the data in Postgres but the dataset returned several errors and we spent a lot of time trying to fix the errors in the dataset. The instructor suggested that we pull the data into Juypter Notebook and try to transform the data. We made a few attempts to clean the dataset however, the dataset did not serve to be consistent to continue our analysis.

We also attempted to clean the SERP flights data which was also part of the data that we found from Kaggle and it had a lot of the same data that we could not work with in a consistent manner.

We kept searching for datasets and found SQLite database that had airline flight data as well, which contains information on airlines, airports, and routes between airports. Each route represents a repeated flight that an airline flies between a source and a destination airport.

**Importing the data**

Eventually, we were able to import the flight\_tickets .csv files into Postgres. We created the database and tables to support the import of the data. The data was inconsistent and there were random extra lines of bad data that posed a problem in importing the data into Postgres however, there was an exception to be selected which eased the import.

Also included with the flight\_tickets.csv file was a SERP\_flights.csv that we analyzed. The data was not easy to clean up and we could not use it further however, we wanted to mention that as we spent time working on the same.

The flights data was in a SQLite database format so we did not need to import that data. We also found datasets on Airbnb data and RedFin real estate data that we could work with.

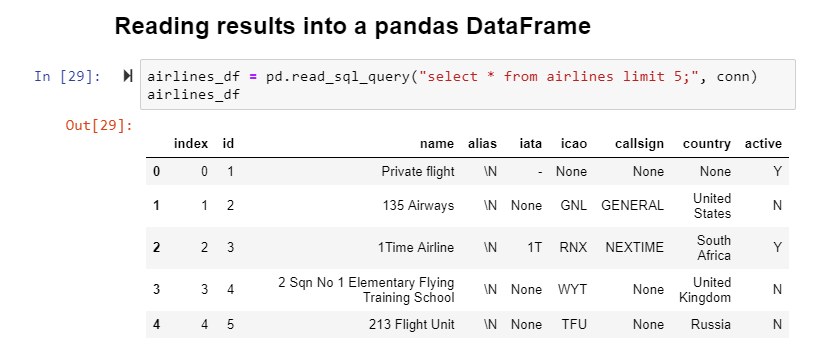
**The type of transformation needed for this data (cleaning, joining, filtering, aggregating, etc).**

1. flights\_tickets/Serp\_flights data

A few notes on the different columns available:

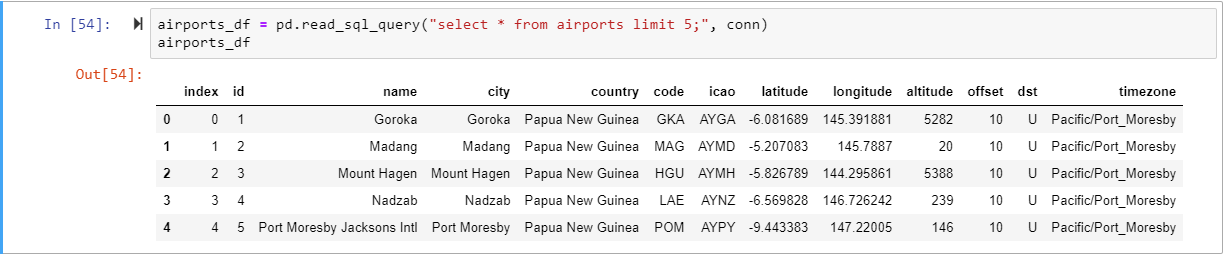
"queryTime" is the time that the query was run (when the request was created). This is different from "searchTime" which is the amount of time it took Google to run the query (usually less than one second). Most of the main columns will always be there, but if one were to pass different parameters there will be more or less columns. For example, one could have columns describing the images, in case we specify the type of search to be "image"

1. Airlines data:



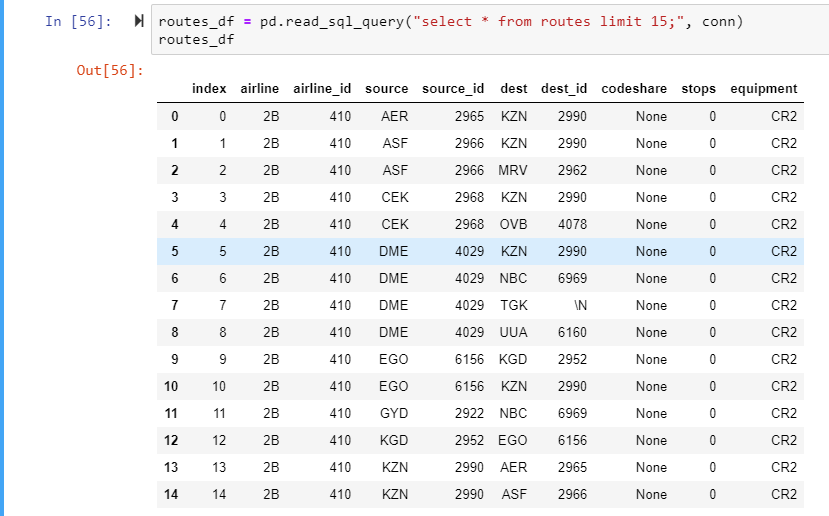
Each row is a different airline, and each column is a property of that airline, such as name, country, etc. Each airline also has a unique id, so we can easily look it up with the id as required.

Here are a few rows from the Airport table:



Each row corresponds to an airport, and contains information on the location of the airport. Each airport also has a unique id, so we can run queries with the same.

Routes table:



Each route contains an airline\_id, which is the id of the airline that flies the route, as well as the source\_id, which is the id of the airport that the route originates from, and dest\_id, which is the id of the destination airport for the flight.

Querying database rows in Python

The read\_sql\_query function would read the results of a SQL query directly into a pandas DataFrame. It automatically reads in the names of the headers from the table. It creates a DataFrame, so we can quickly explore the data. This function gives us the advantage to manipulate the columns.

Modifying database rows

We used sqlite to modify a SQLite database by inserting, updating, and or deleting rows. First we inserted a new row in the airlines table. We specified 9 values to insert, one for each column in the airlines table. This added a new row to the table.

Passing parameters into a query

We added a few columns of data to the airline table, updated the rows and passed queries to view the inserted data. The inserted columns of data were added to the airlines table.

Next we moved on to creating tables, we created a new table daily\_flights and inserted 6 columns of data. We performed a query to check the updated table with the inserted row, then performed a query to delete the data. We also queried to check whether the data from the table was deleted.

Next we moved on to creating a table in daily\_flights and loaded the data into DataFrames and loaded the same data into the SQL database. We created a table to represent each daily flight on a route with the following columns:

* id — integer
* departure — date, when the flight left the airport
* arrival — date, when the flight arrived at the destination
* number — text, the flight number
* route\_id — integer, the id of the route the flight was flying

We also performed some queries to alter tables by adding columns to existing tables within airlines and daily\_flights.

Creating tables with Pandas, we created a DataFrame first to export it to a SQL table.

We created the DataFrame with date time values to be entered into the daily\_flights table. Then, we’ll be able to call the [to\_sql](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_sql.html) method to convert df to a table in a database. We set the keep\_exists parameter to replace to delete and replace any existing tables named daily\_flights. We can ten verify that everything worked by querying the database.

Altering tables with Pandas

Using our airline example, we decided to add an airplanes field to the airlines table that indicates how many airplanes each airline owns. Alter queries are immediately executed and do not require the commit command. All the columns are set to null in SQLite (which translates to None in Python) because there are not any values for the column yet.

**The type of final production database to load the data into (relational or non-relational).**

* SQLite database for the flights data (that data was already in a relational dataset format)
* Postgres to load both the flight\_tickets and SERP\_flights data (relational)

**The final tables or collections that will be used in the production database.**

* SQLite database for the flights data (that data was already in a relational dataset format)
* Postgres to load both the flight\_tickets and SERP\_flights data (relational)

We used Postgres, Jupyter Notebook and Python packages for our work:

* [pandas](https://pandas.pydata.org/): For data manipulation, reshaping, merging, sorting, etc.
* Sqlite3: To create, query and update the database
* Postgres – loading the data into the database and running queries against the same
* Matplotlib for plotting graphs and charts