ETL Process

Step 1: Connecting the datasets

To connect the datasets we needed to join on the geocodes. Joining by the geocode wasn't feasible so we needed to map the codes to zip codes and join on that column. We researched many ways to get the zip code from the provided longitude and latitude and came to a consensus that the best way to approach it was using an API. BigDataCloud.com provided an API that can be reached that interprets the geocode and responds with the respective Zip Code.

Steps to initiate and reach the API

1. Create an account on https://www.bigdatacloud.com/bdc/
2. Propagate your API key on site
3. Import the BigDataCloudAPI library on Python
4. Pass your API Key and client in your code to get responses

```
import bigdatacloudapi
import requests
```

We wrote up code in python to loop through our datasets to pass our GeoCodes and get a json response with the ZipCode and we parse that data.

```
citi_jan = pd.read_csv('CitibikeJanuary2021.csv')


import bigdatacloudapi
import requests


a_lat = [i for i in citi_jan['start station latitude']]
a_long = [i for i in citi_jan['start station longitude']]



apiKey = "69136c65e9e44449aad09e93c691d334"
client = bigdatacloudapi.Client(apiKey)
resultObject,httpResponseCode = client.getIpGeolocationFull({"ip":"8.8.8.8"})


base_url = "https://api.bigdatacloud.net/data/reverse-geocode?latitude="
key = "69136c65e9e44449aad09e93c691d334"

zipcode = []
for i in range(len(citi_jan)):
    address = f"{a_lat[i]}&longitude={a_long[i]}&localityLanguage=en&key="
    zipcode.append(requests.get(f"{base_url}{address}{key}").json()['postcode'])
    print(requests.get(f"{base_url}{address}{key}").json()['postcode'])
```

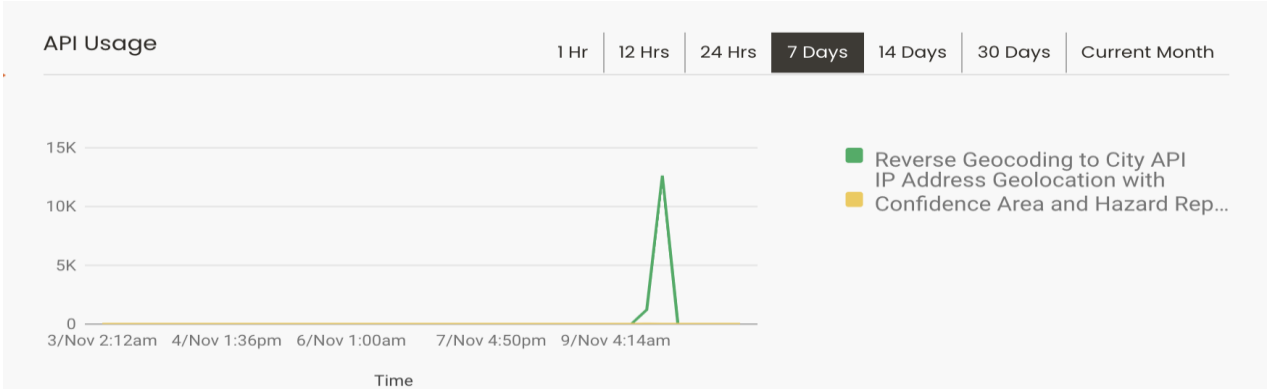
For both dataset (citibike and listing) we stored the zipcodes in a list before storing them in their respective datasets.

To mitigate the amount of requests sent to the API we grouped by the station ID of the city bikes for unique values. This was sufficient as we know the unique values of the station ids would have unique zip codes regardless of the slight change in geocoding. Moreover, due to the limit of API calls and the extensive amount of time required to get the zip codes (around an hour for 10,000 rows), not all 3 of us could run the code. We then split it in two parts, with one group member taking care of the citibike location and the other taking care of the airbnb's listings location.

Below is an example of what we've done to create the zipcode for listings.

```
listings = pd.read_csv('listings.csv')

listings = listings.head(10000)

a_lat = [i for i in listings['latitude']]
a_long = [i for i in listings['longitude']]


apiKey = "c0a952790b8742dd812fc84d7ffd938d"
client = bigdatacloudapi.Client(apiKey)
resultObject,httpResponseCode = client.getIpGeolocationFull({"ip":"8.8.8.8"})


base_url = "https://api.bigdatacloud.net/data/reverse-geocode?latitude="
key = "c0a952790b8742dd812fc84d7ffd938d"

zipcode = []


for i in range(len(listings)):

    address = f"{a_lat[i]}&longitude={a_long[i]}&localityLanguage=en&key="

    zipcode.append(requests.get(f"{base_url}{address}{key}").json()['postcode'])

    print(requests.get(f"{base_url}{address}{key}").json()['postcode'])



listings['zipcode'] = zipcode
listings.to_csv('listing_w_zip.csv')
```

Below is a representation of our calls to the API



Step 2: Creation of our data attributes

After managing to connect our citibikes and airbnb_listings dataset based on their location, we realized that we needed to create new attributes for the sake of our project. We added numerous attributes to our dataset based on other attributes' calculations which will help us keep track of our KPIs. Below is an example of what we have performed:

```
#First we get the number of rows per zipcode for start_zip which corresponds to the bike pickups

pickups = citi_2021.groupby('start_zip').size().to_frame().reset_index().rename(columns={'start_zip':'zipcode',
                                                                                         0:'n_pickup'})
pickups = pickups[pickups['zipcode'] != 0]
pickups.head()
```

	zipcode	n_pickup
1	10001.0	488
2	10002.0	383
3	10003.0	326
4	10004.0	51
5	10005.0	24

```
#We do the same for the dropoffs (zipcode of the end station)
dropoffs = citi_2021.groupby('end_zip').size().to_frame().reset_index().rename(columns={'end_zip':'zipcode',
                                                                                        0:'n_dropoff'})
dropoffs = dropoffs[dropoffs['zipcode'] != 0]
dropoffs.head()
```

	zipcode	n_dropoff
1	10001.0	283
2	10002.0	751
3	10003.0	718
4	10004.0	91
5	10005.0	57

```
trips = pd.merge(
    pickups,
    dropoffs,
    how="left",
    on='zipcode')

trips.fillna(0,inplace=True)

#Calculate the total number of trips in the first two months of the year
trips['Number_of_citibikes_trips_ov2M'] = trips['n_pickup'] + trips['n_dropoff']

#Use the previous results to have an estimate of the total number of trips for the year
```

In the example below, after calculating the number of pickup and drop off per zip code for the period of January to February 2021, we added those numbers which gave us the number of trips over these 2 months. Based on these numbers we also estimated the number of trips this location will experience over the year, although we acknowledge that this might be far off reality due to the limits of our data. A more detailed observation is available on our jupyter notebook submitted along this PDF file.

```
trips = pd.merge(
    pickups,
    dropoffs,
    how="left",
    on='zipcode')

trips.fillna(0,inplace=True)

#Calculate the total number of trips in the first two months of the year
trips['Number_of_citibikes_trips_ov2M'] = trips['n_pickup'] + trips['n_dropoff']

#Use the previous results to have an estimate of the total number of trips for the year
trips['Avg_trip_a_year'] = (12 * trips['Number_of_citibikes_trips_ov2M'])/2

trips.head()
```

	zipcode	n_pickup	n_dropoff	Number_of_citibikes_trips_ov2M	Avg_trip_a_year
0	10001.0	488	283.0	771.0	4626.0
1	10002.0	383	751.0	1134.0	6804.0
2	10003.0	326	718.0	1044.0	6264.0
3	10004.0	51	91.0	142.0	852.0
4	10005.0	24	57.0	81.0	486.0

```
#Merge the listings and trips dataset
listings = pd.merge(
    listings,
    total,
    how="left",
    on='zipcode')
```

```
listings.head(2)
```

...es_value	reviews_per_month	approximate_revenue_total	approximate_revenue_ltm	zipcode	n_pickup	n_dropoff	Number_of_citibikes_trips_ov2M	Avg_trip_a_year
4.41	0.34	$367,200.00	$0.00	10036.0	324.0	239.0	563.0	3378.0
4.65	5.09	$52,020.00	$8,670.00	11238.0	161.0	179.0	340.0	2040.0

Step 3a: ETL process - Data Profiling and Cleaning

Creating a python ETL pipeline object based on the example provided in class, we extracted our data through a csv file using the extract_CSV method built inside the pipeline. The data file extracted is the csv file completed before we start our ETL process, called listings_complete.csv, which contains all the attributes necessary for our project.
After extracting our data we performed a quick data profiling which provides us with the number of rows, columns, the name of our columns, the number of null values etc.

Following our data profiling, we cleaned our data by using an imputer that would change the null values of numeric attributes by the column's median, and replace the null values of categorical columns by the column's most frequent values. Concerning the outliers, we developed a formula that would define an outlier based on the corresponding value's zscore, and change any outliers by the column's median. Concerning the duplicates, there were none.

Step 3b: Creating dimensions and fact tables

Concerning our create_dimension method, we made the surrogate key an optional argument considering that we already had keys for the host and the listing. However, we created a surrogate key for our location dimension.

```
def create_dimension(df, dimension_columns:list,
                         surrogate_key_name:None,
                         surrogate_key_integer_start:None):

    dim = df.copy()
    dim = dim[dimension_columns]

    dim = dim.drop_duplicates(subset=dimension_columns, keep='first')

    if surrogate_key_name != None and surrogate_key_integer_start != None:
        dim.insert(0, surrogate_key_name, range(surrogate_key_integer_start, surrogate_key_integer_start+len(dim)))

    return dim
```

Dimension tables

1. host_dim

	host_id	host_name	host_is_superhost	host_response_time	host_total_listings_count
0	2845.0	Jennifer	f	within an hour	6.0
1	4869.0	LisaRoxanne	f	within a day	1.0
2	7356.0	Garon	f	within a day	1.0
3	7378.0	Rebecca	f	within a day	1.0
4	8967.0	Shunichi	f	within an hour	1.0

2. listing_dim

	listing_id	property_type	accommodates
0	2595.0	Entire rental unit	1.0
1	3831.0	Entire guest suite	3.0
2	5121.0	Private room in rental unit	2.0
3	5136.0	Entire rental unit	4.0
4	5178.0	Private room in rental unit	2.0

3. location_dim

	location_id	host_location	host_neighbourhood	latitude	longitude	zipcode
0	1000	New York, New York, United States	Midtown	40.75356	-73.98559	10036.0
1	1001	New York, New York, United States	Clinton Hill	40.68494	-73.95765	11238.0
2	1002	New York, New York, United States	Bedford-Stuyvesant	40.68535	-73.95512	11216.0
3	1003	Brooklyn, New York, United States	Greenwood Heights	40.66265	-73.99454	11232.0
4	1004	New York, New York, United States	Hell's Kitchen	40.76457	-73.98317	10019.0

Finally, we decided to create our fact table by first merging, using a left join, the cleaned_airbnb dataframe with the location_dim table on all their common columns (which are location attributes), as location_dim was the only dimension in which we created a surrogate key. Thanks to this join, our merged table had all the necessary columns to complete our fact table, with the last step being to drop the non-necessary columns to keep only the measures required for our project.
We can see below how we created our fact table:

```
def create_fact(df,dimensions:list):

    print(f"-----\n creating fact table")
    # copy full dataframe to create fact table
    fact = df.copy()

    # Location_dim is the only dimension where we created a surrogate key.
    # Therefore, we left join location_dim with fact based on the location columns which will
    # provide our fact table with the accurate location_id column.
    for d in dimensions:
        if d is location_dim:
            fact = pd.merge(fact, d, on = ['host_location','host_neighbourhood','latitude',
                                           'longitude','zipcode'], how='left')


    # Drop columns that would duplicate as they'd already be present in cleaned_airbnb (df here)
    # By dropping these column, we make sure that our fact table doesn't contain attributes that are
    # already present in our dimensions, besides the foreign key whose names finish by 'id'
    cols_to_drop = []
    for d in dimension:
        for c in d.columns:
            if c in fact.columns and not c.endswith('id'):
                cols_to_drop.append(c)


    fact = fact.drop(cols_to_drop, axis=1)


    #Reorder columns
    location_id = fact['location_id']
    fact = fact.drop(columns=['location_id'], axis=1)
    fact.insert(loc=0, column='location_id', value=location_id)

    # return fact table as a dataframe
    print(f"fact table created with {len(fact)} rows \n-----")
    return fact
```

4. Revenue_fact_table

	location_id	listing_id	host_id	price	number_of_reviews	review_scores_location	review_scores_value	reviews_per_month	approximate_revenue_total	approxi
0	1000	2595.0	2845.0	150.0	48.0	4.86	4.41	0.34	367200.0	
1	1001	3831.0	4869.0	75.0	7.0	4.72	4.65	0.23	52020.0	
2	1002	5121.0	7356.0	60.0	50.0	4.47	4.52	0.55	153000.0	
3	1003	5136.0	7378.0	275.0	1.0	4.00	5.00	0.01	2337.5	
4	1004	5178.0	8967.0	61.0	7.0	4.86	4.35	3.63	100589.0	

We can see in the first half of our fact table that it contains the 3 foreign keys established in our dimensional model.

Step 4: Loading and joining tables

Lastly, we loaded the three dimensions tables and the one fact table into BigQuery.

```
ETL_pipeline.load_table_to_bigquery(bigquery_client, listing_dim, dataset_name='Airbnb',
                                    table_name='listing_dim')
```

```
-----
 loading listing_dim to BigQuery dataset Airbnb
Loaded 10000 rows and 3 columns to cis9440-323912.Airbnb.listing_dim
-----
```

```
ETL_pipeline.load_table_to_bigquery(bigquery_client, host_dim, dataset_name='Airbnb',
                                    table_name='host_dim')
```

```
-----
 loading host_dim to BigQuery dataset Airbnb
Loaded 8718 rows and 5 columns to cis9440-323912.Airbnb.host_dim
-----
```

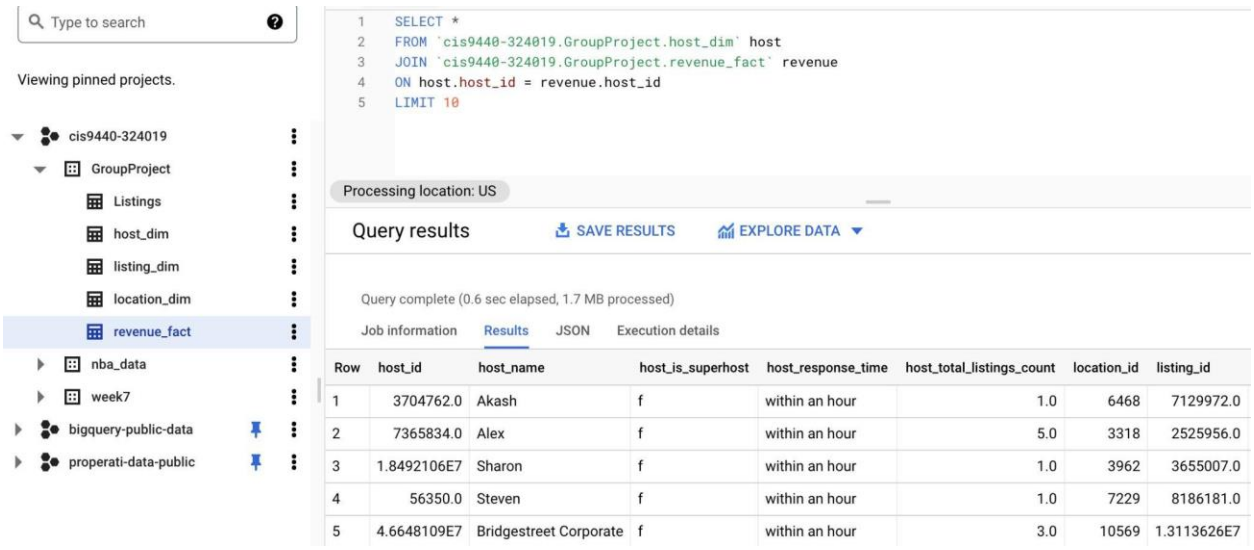
```
ETL_pipeline.load_table_to_bigquery(bigquery_client, location_dim, dataset_name='Airbnb',
                                    table_name='location_dim')
```

```
-----
 loading location_dim to BigQuery dataset Airbnb
Loaded 9886 rows and 6 columns to cis9440-323912.Airbnb.location_dim
-----
```

```
ETL_pipeline.load_table_to_bigquery(bigquery_client, revenue_fact_table, dataset_name='Airbnb',
                                    table_name='revenue_fact_table')
```

```
-----
 loading revenue_fact_table to BigQuery dataset Airbnb
Loaded 10000 rows and 19 columns to cis9440-323912.Airbnb.revenue_fact_table
-----
```

We can now see below all the tables loaded in BigQuery. For validation, we joined host_dim and revenue_fact_table on host_id as an example.



Row	host_id	host_name	host_is_superhost	host_response_time	host_total_listings_count	location_id	listing_id
1	3704762.0	Akash	f	within an hour	1.0	6468	7129972.0
2	7365834.0	Alex	f	within an hour	5.0	3318	2525956.0
3	1.8492106E7	Sharon	f	within an hour	1.0	3962	3655007.0
4	56350.0	Steven	f	within an hour	1.0	7229	8186181.0
5	4.6648109E7	Bridgestreet Corporate	f	within an hour	3.0	10569	1.3113626E7