Before you turn in the homework, make sure everything runs as expected. To do so, select **Kernel→Restart & Run All** in the toolbar above. Remember to submit both on **DataHub** and **Gradescope**.

Please fill in your name and include a list of your collaborators below.

In [88]:
```
NAME = "Connor McCormick"
COLLABORATORS = ""
```

Project 2: NYC Taxi Rides

Part 1: Data Wrangling

In this notebook, we will first query a database to fetch our data and generate training and test sets.

Imports

In [89]:
```
import os
import pandas as pd
import numpy as np
from pathlib import Path
from sqlalchemy import create_engine
from utils import timeit
```

SQLite

SQLite (https://www.sqlite.org/whentouse.html) is a SQL database engine that excels at managing data stored locally in a file. We will be using SQLite to query for our data. First let's check that our database is accessible and set up properly. Run the following line to make sure the data is there and pay attention to how big the data is.

In practice, data is stored in a distributed SQL database that spans machines (e.g. Hive (https://stackoverflow.com/questions/20030436/what-is-hive-is-it-a-database)) or even continents (e.g. Spanner (https://en.wikipedia.org/wiki/Spanner_(database))). However, how you query the data will remain the same: the SQL language.

In [90]:
```
!ls -lh /srv/db/taxi_2016_student_small.sqlite
```

```
-rw-r--r-- 1 root root 2.1G Nov  7 04:43 /srv/db/taxi_2016_student_smal
l.sqlite
```

Running this line will connect to SQLite engine and test the connection by printing out the total number of rows.

In [91]:
```
DB_URI = "sqlite:////srv/db/taxi_2016_student_small.sqlite"
TABLE_NAME = "taxi"

sql_engine = create_engine(DB_URI)
with timeit():
    print(f"Table {TABLE_NAME} has {sql_engine.execute(f'SELECT COUNT(*)
 FROM {TABLE_NAME}').first()[0]} rows!")
```

```
Table taxi has 15000000 rows!
1.02 s elapsed
```

Quick note: One piece of syntax above that you may not be familiar with is the Python f-string (https://realpython.com/python-f-strings/), a relatively new feature to the language.

Basically, it automatically replaces text inside curly braces with the results of the given expression. For example:

In [92]:
```
bloop = "wet egg"
print(f"{bloop} gets replaced, oh also {3 + 5}.")
```

```
wet egg gets replaced, oh also 8.
```

NYC Taxi Data

We are working with a much larger dataset (15,000,000 rows!), larger than anything we have worked with before. If you are not careful in writing your queries, you may crash your kernel. Please do not `"SELECT * FROM taxi"`. This is a reality that we must face; we do not always get to work with supercomputers that can load everything in memory.

Data Overview

Below is the schema for the `taxi` database:

```
CREATE TABLE taxi_train(
  "record_id" integer primary key,
  "VendorID" INTEGER,
  "tpep_pickup_datetime" TEXT,
  "tpep_dropoff_datetime" TEXT,
  "passenger_count" INTEGER,
  "trip_distance" REAL,
  "pickup_longitude" REAL,
  "pickup_latitude" REAL,
  "RatecodeID" INTEGER,
  "store_and_fwd_flag" TEXT,
  "dropoff_longitude" REAL,
  "dropoff_latitude" REAL,
  "payment_type" INTEGER,
  "fare_amount" REAL,
  "extra" REAL,
  "mta_tax" REAL,
  "tip_amount" REAL,
  "tolls_amount" REAL,
  "improvement_surcharge" REAL,
  "total_amount" REAL
);
```

Here is a description for your convenience:

- `recordID`: primary key of this dataset
- `VendorID`: a code indicating the provider associated with the trip record
- `passenger_count`: the number of passengers in the vehicle (driver entered value)
- `trip_distance`: trip distance
- `dropoff_datetime`: date and time when the meter was engaged
- `pickup_datetime`: date and time when the meter was disengaged
- `pickup_longitude`: the longitude where the meter was engaged
- `pickup_latitude`: the latitude where the meter was engaged
- `dropoff_longitude`: the longitude where the meter was disengaged
- `dropoff_latitude`: the latitude where the meter was disengaged
- `duration`: duration of the trip in seconds
- `payment_type`: the payment type
- `fare_amount`: the time-and-distance fare calculated by the meter
- `extra`: miscellaneous extras and surcharges
- `mta_tax`: MTA tax that is automatically triggered based on the metered rate in use
- `tip_amount`: the amount of credit card tips, cash tips are not included
- `tolls_amount`: amount paid for tolls
- `improvement_surcharge`: fixed fee
- `total_amount`: total amount paid by passengers, cash tips are not included

Question 1: SQL Warmup

Let's begin with some SQL questions! Remember, be careful not to select too many entries in your query. Your kernel **will** crash! Please write your queries in the provided triple quotes and format them with proper SQL style. Below is an example which grabs the first 5 rows from the `taxi` database.

We will use the `timeit` contextmanager from the `utils` file to time each SQL execution. *Beware that SQL can be slow sometimes; enterprise SQL quries often run for hours or days!* (several minutes execution time is [considered fast (https://hortonworks.com/blog/benchmarking-apache-hive-13-enterprise-hadoop/)](https://hortonworks.com/blog/benchmarking-apache-hive-13-enterprise-hadoop/)). In each cell, we have added anitipated execution time to use as a guideline for writing your quries.

In [93]:
```
q1x_query = f"""
            SELECT *
            FROM {TABLE_NAME}
            LIMIT 5;
            """

with timeit(): # this query should take less than a second
    q1x_df = pd.read_sql(q1x_query, sql_engine)
q1x_df.head()
```

```
0.01 s elapsed
```

Out[93]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	1	2	2016-01-01 00:00:00	2016-01-01 00:00:00	2
1	8	1	2016-01-01 00:00:01	2016-01-01 00:11:55	1
2	17	2	2016-01-01 00:00:05	2016-01-01 00:07:14	1
3	18	1	2016-01-01 00:00:06	2016-01-01 00:04:44	1
4	22	2	2016-01-01 00:00:08	2016-01-01 00:18:51	1

Question 1a

Select the top 1000 rows from the `taxi` database ordered by descending `total_amount`. Note that this data is real uncleaned data, with all the strange quirks that come from such datasets, e.g. you'll see that the most expensive taxi ride was $153,296.22, which is certainly some sort of error in the data.

In [94]:
```
q1a_query = """
            SELECT *
            FROM taxi
            ORDER BY total_amount DESC
            LIMIT 1000
            """
with timeit(): # This query is expected to run for less than 20 seconds.
    q1a_df = pd.read_sql(q1a_query, sql_engine)
q1a_df.head()
```

```
21.33 s elapsed
```

Out[94]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	15958593	1	2016-02-16 18:20:33	2016-02-16 18:36:33	1
1	28810418	1	2016-03-20 11:44:34	2016-03-20 12:03:29	2
2	63007353	1	2016-06-13 15:06:32	2016-06-13 15:07:36	1
3	58271050	2	2016-05-27 14:38:36	2016-05-27 15:10:15	1
4	50682006	1	2016-05-11 22:26:52	2016-05-11 22:32:08	1

In [95]:
```
assert len(q1a_df) == 1000
assert q1a_df.loc[0, 'total_amount'] >= q1a_df.loc[999, "total_amount"]
```

Question 1b

Get the `mean`, `max` and `min` `total_amount` for each vendor. As above, you'll get strange answers, since finding the min and max of a big uncleaned dataset captures the most extreme outliers. Make sure your query outputs the columns in this exact order.

In [96]:
```
q1b_query = """
            SELECT
                AVG(total_amount) AS mean,
                MAX(total_amount) AS max,
                MIN(total_amount) AS min
            FROM taxi
            GROUP BY VendorID
            """


with timeit(): # This query is expected to run for about 10 seconds.
    q1b_df = pd.read_sql_query(q1b_query, sql_engine)
q1b_df.head()
```

```
9.31 s elapsed
```

Out[96]:

	mean	max	min
0	15.981053	153296.22	0.0
1	16.276753	4887.30	-958.4

In [97]:
```
assert q1b_df.shape == (2, 3)
assert 15 < q1b_df.iloc[0, 0] < 17
assert q1b_df.iloc[1, 1] == 4887.30
assert q1b_df.iloc[1, 2] == -958.4
```

Question 1c

Find the total amount paid and pickup time for all rides that started June 28th, 2016, then order the result by total amount in descending order. Again, make sure your query outputs the columns in this exact order.

Hint: From the schema, note that `tpep_pickup_datetime` is a text field. We're effectively looking for strings that have a start time that comes after `2016-06-28 00:00:00` but before `2016-06-29 00:00:00`.

In [98]:
```
q1c_query = """
            SELECT
                total_amount,
                tpep_pickup_datetime
            FROM taxi
            WHERE tpep_pickup_datetime LIKE '%2016-06-28%'
            ORDER BY total_amount DESC
            """

with timeit(): # This query should take about 3 seconds.
    q1c_df = pd.read_sql_query(q1c_query, sql_engine)
q1c_df.head()
```

```
3.45 s elapsed
```

Out[98]:

	total_amount	tpep_pickup_datetime
0	390.99	2016-06-28 12:23:13
1	289.12	2016-06-28 15:14:42
2	286.30	2016-06-28 00:01:13
3	285.80	2016-06-28 13:34:12
4	275.30	2016-06-28 21:38:13

In [99]:
```
assert q1c_df.iloc[0, 0] == 390.99
assert q1c_df.shape == (74857, 2)
```

Question 1d

Find all rides starting in the month of January in the year 2016, selecting only those entries whose `record_id` ends in 00.

Note: The rest of our questions in Part 1, Part 2 and Part 3 will be based off of the results of this query. In part 4, you will be to use anything else in the database for fitting a model (more later). Because of its importance for the rest of the assignment, your query must be correct for this question.

In [100]:
```
q1d_query = f"""
            SELECT *
            FROM {TABLE_NAME}
            WHERE tpep_pickup_datetime
                BETWEEN '2016-01-01' AND '2016-02-01'
                AND record_id % 100 == 0
            ORDER BY tpep_pickup_datetime
            """


with timeit(): # This query should take less than 3 second
    q1d_df = pd.read_sql_query(q1d_query, sql_engine)
q1d_df.head()
```

```
2.83 s elapsed
```

Out[100]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	37300	1	2016-01-01 00:02:20	2016-01-01 00:11:58	2
1	37400	1	2016-01-01 00:03:04	2016-01-01 00:28:54	1
2	37500	2	2016-01-01 00:03:40	2016-01-01 00:12:47	6
3	37900	2	2016-01-01 00:05:38	2016-01-01 00:10:02	3
4	38500	1	2016-01-01 00:07:50	2016-01-01 00:23:42	1

In [101]:
```
assert q1d_df.iloc[0].loc['tpep_pickup_datetime'] >= "2016-01-01"
assert q1d_df.iloc[-1].loc['tpep_pickup_datetime'] <= "2016-02-01"
assert q1d_df.shape == (23674, 20)
```

Question 2: Data Inspection

We will refer to the table generated by Question 1d as `Jan16`. Note that we have not explicitly built a table called `Jan16` in our SQL database. We are instead using `Jan16` to represent the mathematical object that results from Question 1d. Let us now check some basic properties of `Jan16`. We will be addressing the following properties within our dataset:

- missing data values
- duplicated values
- range of duration values
- range of latitude and longitude values
- range of passenger count values

It is good practice to check these properties when presented with a new dataset. There are two ways to check these properties: Approach one is to write SQL queries that directly interact with the database. Approach two is to create a pandas dataframe and use pandas methods. Since you've already gotten similar practice with pandas earlier in the semester, we'll stick with approach one.

In the following problems, you'll check these properties using SQL queries. We'll also provide you with the pandas solution so that you can compare with your SQL based solution. In order to be able to provide these pandas solutions, we need to store the result of your `q1d_query` into a dataframe, which we'll call `jan_16_df`.

In [102]:

```
with timeit(): # less than 3 seconds
    jan_16_df = pd.read_sql_query(q1d_query, sql_engine)
jan_16_df['tpep_pickup_datetime'] = pd.to_datetime(jan_16_df['tpep_picku
p_datetime'])
jan_16_df['tpep_dropoff_datetime'] = pd.to_datetime(jan_16_df['tpep_drop
off_datetime'])
jan_16_df.head()
```

```
2.82 s elapsed
```

Out[102]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	37300	1	2016-01-01 00:02:20	2016-01-01 00:11:58	2
1	37400	1	2016-01-01 00:03:04	2016-01-01 00:28:54	1
2	37500	2	2016-01-01 00:03:40	2016-01-01 00:12:47	6
3	37900	2	2016-01-01 00:05:38	2016-01-01 00:10:02	3
4	38500	1	2016-01-01 00:07:50	2016-01-01 00:23:42	1

For the remaining questions in part 1, you'll be using nested queries. For example, the nested query below selects all rides with passenger count equal to 2 from `Jan16`. Reminder that Python automatically replaces the `"q1d_query"` in `temporary_table_query_example` with the contents of the string variable named `q1d_query`.

In [103]:
```
# Jan16 to dataframe using temporary table
temporary_table_query_example = f"""
SELECT *
FROM ({q1d_query})
WHERE passenger_count = 2;"""
print(temporary_table_query_example)
```

```
SELECT *
FROM (
            SELECT *
            FROM taxi
            WHERE tpep_pickup_datetime
                BETWEEN '2016-01-01' AND '2016-02-01'
                AND record_id % 100 == 0
            ORDER BY tpep_pickup_datetime
            )
WHERE passenger_count = 2;
```

The cell below executes this nested query.

In [104]:
```
with timeit(): # less than 3 seconds
    pd.read_sql_query(temporary_table_query_example, sql_engine)
```

```
2.63 s elapsed
```

Question 2a

Write a SQL query to check if `Jan16` contains any missing values. Unfortunately, in this table, missing values are *not* specified with NaN nor empty strings. For example, take a look at record ID 136700. What do you observe about the location information?

Write a SQL query `q2a_query` that collects all rows that have a missing `tpep_pickup_datetime`, `tpep_dropoff_datetime`, `pickup_longitude`, or `pickup_latitude`. Then set `number_of_rows_with_missing_values` to the number of rows that have at least one missing value.

In pandas, we could use boolean indexing to filter out these values.

In [105]:
```
# Inspecting record 136700 for your convience.

pd.read_sql_query(f"""
SELECT *
FROM {TABLE_NAME}
WHERE record_id = 136700
""", sql_engine)
```

Out[105]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	136700	1	2016-01-01 03:13:07	2016-01-01 03:28:48	1

In [106]:
```
q2a_query = f"""
            SELECT *
            FROM ({q1d_query})
            WHERE cast(pickup_longitude as decimal(5,1)) = 0.0;
            """

number_of_rows_with_missing_values = 371


with timeit(): # Should take < 3 seconds
    q2a_df = pd.read_sql_query(q2a_query, sql_engine)
q2a_df.head()
```

```
2.61 s elapsed
```

Out[106]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	136700	1	2016-01-01 03:13:07	2016-01-01 03:28:48	1
1	216000	1	2016-01-01 11:46:23	2016-01-01 11:57:50	4
2	228400	2	2016-01-01 12:40:12	2016-01-01 12:46:35	1
3	340400	2	2016-01-01 19:37:19	2016-01-01 20:17:57	5
4	360600	1	2016-01-01 21:06:29	2016-01-01 21:09:41	1

In [107]:
```
# Hidden Test
```

Question 2b

Write a SQL query `q2b_query` to help determine if there are any duplicate records in `Jan16`. Set the boolean `has_duplicates` variable to `True` or `False` based on what you learn. You may use `len(jan_16_df)` in your solution.

For comparison, approach two (pandas) for duplicate checking looks like `num_duplicates = jan_16_df.duplicated(subset=jan_16_df.columns).sum()`.

In [108]:
```
q2b_query = f"""
            SELECT COUNT(*)
            FROM ({q1d_query})
            GROUP BY record_id
            HAVING COUNT(*) > 1
            """

has_duplicates = False

with timeit(): # should take < 3 seconds
    q2b_df = pd.read_sql_query(q2b_query, sql_engine)
q2b_df.head()
```

```
2.69 s elapsed
```

Out[108]:

	COUNT(*)

In [109]:
```
# Hidden test
```

Question 2c

Find the min and max trip duration in `Jan16`. You may manually fill in the `min_duration`, `max_duration` placeholders.

Hint: check `julianday` (https://www.techonthenet.com/sqlite/functions/julianday.php) in `SQLite`. Your answer should be decimal representations of a day (e.g. 6 hours = 0.25).

In [110]:
```
q2c_query = f"""
            SELECT (julianday(tpep_dropoff_datetime) - julianday(tpep_pi
ckup_datetime)) as time
            FROM ({q1d_query})
            ORDER BY time DESC
            """

min_duration = 0.0
max_duration = 0.999190

with timeit(): # should take < 3 seconds
    q2c_df = pd.read_sql_query(q2c_query, sql_engine)
q2c_df.head()
```

```
2.69 s elapsed
```

Out[110]:

	time
0	0.999190
1	0.998889
2	0.998611
3	0.998472
4	0.998472

In [111]:
```
df_min_seconds = min(jan_16_df["tpep_dropoff_datetime"] - jan_16_df["tpe
p_pickup_datetime"]).total_seconds()
df_max_seconds = max(jan_16_df["tpep_dropoff_datetime"] - jan_16_df["tpe
p_pickup_datetime"]).total_seconds()
assert min_duration == df_min_seconds/86400
assert np.isclose(max_duration,df_max_seconds/86400)
```

The cell above should have shown that some trips are extremely long (almost a day)! What is up with this? There may be several reasons why we have a handful of taxi rides with abnormally high durations.

Using our domain knowledge about taxi businesses in NYC, we might believe that taxi drivers accidentally left their meters running, which causes high duration values to be recorded. This is a plausible explanation. Because of this, we will only train and predict on taxi ride data that has a duration of at most 12 hours.

Question 3: Data Cleaning

Now let's use domain knowledge and clean up our data. You will use SQL while we perform the equivalent operations in pandas on `cleaned_jan_16_df`.

In [112]:
```
cleaned_jan_16_df = jan_16_df.copy()
```

Question 3a

Write a SQL Query to find all rides in `Jan16` that are less than 12 hours, or 0.5 days. We will use this query as a nested query `q3a_query` in question 3b.

Hint: Ideas in `q1d_query` can be heavily reused

In [113]:
```
q3a_query = f"""
            SELECT *
            FROM ({q1d_query})
            WHERE (julianday(tpep_dropoff_datetime) - julianday(tpep_pic
kup_datetime)) < 0.5
            """

with timeit(): # should take < 3 seconds
    q3a_df = pd.read_sql_query(q3a_query, sql_engine)
q3a_df.head()
```

```
2.89 s elapsed
```

Out[113]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	37300	1	2016-01-01 00:02:20	2016-01-01 00:11:58	2
1	37400	1	2016-01-01 00:03:04	2016-01-01 00:28:54	1
2	37500	2	2016-01-01 00:03:40	2016-01-01 00:12:47	6
3	37900	2	2016-01-01 00:05:38	2016-01-01 00:10:02	3
4	38500	1	2016-01-01 00:07:50	2016-01-01 00:23:42	1

In [114]:
```
cleaned_jan_16_df['duration'] = cleaned_jan_16_df["tpep_dropoff_datetim
e"]-cleaned_jan_16_df["tpep_pickup_datetime"]
cleaned_jan_16_df['duration'] = cleaned_jan_16_df['duration'].dt.total_s
econds()
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['duration'] < 12
 * 3600]
assert len(q3a_df) == len(cleaned_jan_16_df)
```

Question 3b

Our objective is to predict the duration of taxi rides in the New York City region. Therefore, we should verify that our dataset contains only rides that are either starting or ending in New York (or are contained within the NY region).

Based on different coordinate estimates of New York City, the (inclusive) latitude and longitude ranges are (roughly) as follows:

- Latitude is between 40.63 and 40.85
- Longitude is between -74.03 and -73.75

Write a SQL query to find all rides in `q3a_query` that are within the New York City region. We will use this query as a temporary table `q3b_query` in question 3c.

- Note: This query can be tedious to write. In practice people use special data types to encode geographical information. For example, if we were using Postgres (made in Berkeley!) instead of SQLite, we could use the geo-spatial data types provided as part of PostGIS (https://postgis.net/).

Hint: Ideas in `q3a_query` can be heavily reused

In [115]:
```
# Try using this function!
def bounding_condition(lat_l, lat_u, lon_l, lon_u):
    return f"""
            pickup_longitude <= {lon_u} AND
            pickup_longitude >= {lon_l} AND
            dropoff_longitude <= {lon_u} AND
            dropoff_longitude >= {lon_l} AND
            pickup_latitude <= {lat_u} AND
            pickup_latitude >= {lat_l} AND
            dropoff_latitude <= {lat_u} AND
            dropoff_latitude >= {lat_l}
            """
lat_l = 40.63
lat_u = 40.85
lon_l = -74.03
lon_u = -73.75

bound_query = bounding_condition(lat_l, lat_u, lon_l, lon_u)

q3b_query = f"""
            SELECT *
            FROM ({q3a_query})
            WHERE ({bound_query})
            """

with timeit(): # should take < 3 seconds
    q3b_df = pd.read_sql_query(q3b_query, sql_engine)
q3b_df.head()
```

```
2.83 s elapsed
```

Out[115]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	37300	1	2016-01-01 00:02:20	2016-01-01 00:11:58	2
1	37400	1	2016-01-01 00:03:04	2016-01-01 00:28:54	1
2	37500	2	2016-01-01 00:03:40	2016-01-01 00:12:47	6
3	37900	2	2016-01-01 00:05:38	2016-01-01 00:10:02	3
4	38500	1	2016-01-01 00:07:50	2016-01-01 00:23:42	1

By contrast, the approach two (pandas) equivalent is given below.

In [116]:
```
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['pickup_longitude'] <= -73.75]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['pickup_longitude'] >= -74.03]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['pickup_latitude'] <= 40.85]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['pickup_latitude'] >= 40.63]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['dropoff_longitude'] <= -73.75]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['dropoff_longitude'] >= -74.03]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['dropoff_latitude'] <= 40.85]
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['dropoff_latitude'] >= 40.63]
assert len(q3b_df) == len(cleaned_jan_16_df)
```

Question 3c

The `passenger_count` variable has a minimum value of 0 passengers and a maximum value of 9 passengers. Having 0 passengers does not make sense in the context of this business case; it is likely an error and should therefore be removed from our dataset.

Write a SQL query to find all rides in `q3b_query` with `passenger_count` greater than 0.

Hint: Ideas in `q3b_query` can be heavily reused

In [117]:
```
q3c_query = f"""
            SELECT *
            FROM ({q3b_query})
            WHERE passenger_count > 0
            """


with timeit():
    q3c_df = pd.read_sql_query(q3c_query, sql_engine)
q3c_df.head()
```

```
2.81 s elapsed
```

Out[117]:

	record_id	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count
0	37300	1	2016-01-01 00:02:20	2016-01-01 00:11:58	2
1	37400	1	2016-01-01 00:03:04	2016-01-01 00:28:54	1
2	37500	2	2016-01-01 00:03:40	2016-01-01 00:12:47	6
3	37900	2	2016-01-01 00:05:38	2016-01-01 00:10:02	3
4	38500	1	2016-01-01 00:07:50	2016-01-01 00:23:42	1

In [118]:
```
cleaned_jan_16_df = cleaned_jan_16_df[cleaned_jan_16_df['passenger_coun
t'] > 0]
assert len(q3c_df) == len(cleaned_jan_16_df)
```

Question 3d

If you passed all the previous tests, then we are done cleaning! We would like to check how many records we have removed to ensure that it is a relatively small number (otherwise we might introduce bias within our dataset). In the cell below calculate the number and proportion of records we removed from the original `jan_16_df` during the data cleaning process.

To avoid possible error propagation, you should use our `cleaned_jan_16_df` in your solution as the final cleaned dataset instead of your `q3c_df`.

In [119]:
```
num_records_removed = len(jan_16_df) - len(cleaned_jan_16_df)
proportion_records_removed = num_records_removed / len(jan_16_df)

print(f'Records removed:{num_records_removed}')
print(f'Proportion records removed:{proportion_records_removed}')
```

```
Records removed:731
Proportion records removed:0.030877756188223367
```

In [120]:
```
assert proportion_records_removed < 0.04
assert proportion_records_removed > 0.03
```

At this point, let's take a look at the final query that cleaned up the data. Nesting SQL queries or creating views for future re-use are common pattern in analytical queries. Pay attention to each WHERE clause.

In [121]:
```
print(q3c_query)
```

```
                SELECT *
                FROM (
                SELECT *
                FROM (
                SELECT *
                FROM (
                SELECT *
                FROM taxi
                WHERE tpep_pickup_datetime
                    BETWEEN '2016-01-01' AND '2016-02-01'
                    AND record_id % 100 == 0
                ORDER BY tpep_pickup_datetime
                )
                WHERE (julianday(tpep_dropoff_datetime) - julianday(tpep_pi
ckup_datetime)) < 0.5
                )
                WHERE (
                pickup_longitude <= -73.75 AND
                pickup_longitude >= -74.03 AND
                dropoff_longitude <= -73.75 AND
                dropoff_longitude >= -74.03 AND
                pickup_latitude <= 40.85 AND
                pickup_latitude >= 40.63 AND
                dropoff_latitude <= 40.85 AND
                dropoff_latitude >= 40.63
                )
                )
                WHERE passenger_count > 0
```

Question 4: Training and Validation Split

Now that we have fetched and cleaned our data, let's create training and validation sets. We will use a 80/20 ratio for training/validation and set `random_state=42` for the purpose of grading.

In [122]:
```
from sklearn.model_selection import train_test_split
train_df, val_df = train_test_split(cleaned_jan_16_df, test_size=0.2, ra
ndom_state=42)
```

In [123]:
```
# Check that 80% records in training and 20% in validation set.
assert len(train_df) < 18500
assert len(train_df) > 17000
assert len(val_df) > 4000
assert len(val_df) < 5000
```

Part 1 Exports

Throughout our analysis, we have formatted and cleaned our data. Since we are ready to begin the feature engineering process, a good practice is to start a new notebook (since this one is getting quite long!). Now, we will save our formatted data, which we will load in part 2. **Be sure to run the cell below!**

Please read the documentation below on saving and loading hdf files.

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_hdf.html (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_hdf.html)

https://pandas.pydata.org/pandas-docs/version/0.22/generated/pandas.read_hdf.html (https://pandas.pydata.org/pandas-docs/version/0.22/generated/pandas.read_hdf.html)

In [124]:
```
Path("data/part1").mkdir(parents=True, exist_ok=True)
data_file = Path("data/part1", "cleaned_data.hdf") # Path of hdf file
train_df.to_hdf(data_file, "train") # Train data of hdf file
val_df.to_hdf(data_file, "val") # Val data of hdf file
```

Part 1 Conclusions

We have downloaded/loaded our data, cleaned the data, and split our data into a training and test set to use in future analysis and modeling.

Please proceed to part 2, where we will be exploring the taxi ride training set.

Submission

You're almost done!

Before submitting this assignment, ensure that you have:

1. Restarted the Kernel (in the menubar, select Kernel→Restart & Run All)
2. Validated the notebook by clicking the "Validate" button.

Then,

1. **Submit** the assignment via the Assignments tab in **Datahub**
2. **Upload and tag** the manually reviewed portions of the assignment on **Gradescope**