Allison Day

Part 1: SQL

1. Write a SQL query to list all users and their total order amount

```
SELECT u.user_id, SUM(o.amt) AS total
FROM user u
LEFT JOIN order_header o ON u.user_id = o.user_id
GROUP BY user_id;
```

Returns a table that looks like this:

user_id	total
1	52
2	169
3	*null*

The users who haven't spent money will have a `null` in the total column. It would be interesting to sort desc or filter to only orders of that week, or maybe to market to find which is the most productive. Pretty basic but opens the door to interesting insights.

2. Write a SQL query to list users that have not placed an order

```
SELECT u.*
FROM user u
WHERE NOT EXISTS (SELECT 1 FROM order_header o WHERE o.user_id = u.user_id);
```

returns a table that looks like this:

user_id	name	market
3	Ruby	North

I chose `WHERE NOT EXISTS` over `WHERE o.user_ID IS NULL` because I think exists future proofs the query more since we are joining tables and may want to add more constraints in the future.

3. Write a SQL query to add an index to speed up the query

```sql
CREATE INDEX idx_market
ON user (market);

CREATE INDEX idx_order_date
ON order_header (order_date);
```

If I had millions of rows, I would want to make an index of the different markets so that I can easily filter by a specific market.

My logic for the order table falls along those same lines. When I'm working with time data I often filter by date and creating an index would speed up the `WHERE` statement.

Part 2: My Code

This is a snippet of code from my internship that I'm particularly proud of. I had to go through a dataset of points, get the overlapping ones and then rank as they rose (lowest was 1, highest was n). But if a height was missing then it needed to skip that height (start at 2). Speed was the most important factor. This code needed to comb through over thousands of points every few seconds.

```python
# python using pandas
centers["position"] = centers.groupby(["x", "y"]).ngroup()
centers["heightlist"] = centers.groupby("position")["z"].transform("count")
avg_level = ceil(centers["heightlist"].mean())

for position in centers["position"].unique():
    pos = centers.loc[(centers["position"] == position)]
    pos.sort_values(by=["z"], ascending=True, inplace=True)

    if pos["heightlist"].min() < avg_level:
        high = avg_level
    else:
        high = pos["heightlist"].min()

    levels = []
    for i in range(len(pos)):
        levels.append(high - i)

    centers.loc[(centers["position"] == position), "level"] = levels
```

Line 1

```
centers["position"] = centers.groupby(["x", "y"]).ngroup()
```

Creates a new column named `[position]` and has a unique id for each unique (x,y). This is very helpful to help us filter each position later.

Line 2

```
centers["heightlist"] = centers.groupby("position")["z"].transform("count")
```

Groups by position, and counts the number of unique z values are in that position.

Line 3

```
avg_level = ceil(centers["heightlist"].mean())
```

The average heightlist for the whole dataset

Line 4

```
for position in centers["position"].unique():
```

This line ittorated through the centers.csv to find each unique position (x,y) for filtering.

Line 5

```
pos = centers.loc[(centers["position"] == position)]
```

Creates a filtered dataset named *pos* containing only those with the same position as the loop.

Line 6

```
pos.sort_values(by=["z"], ascending=True, inplace=True)
```

Sorts all of the values by their z in ascending order so that when we rank it will be from lowest to highest (1, ..., n). Decided to do it inplace so that we don't have to redefine *pos* again, and because we're not losing any data.

Lines 7-10

```
if pos["heightlist"].min() < avg_level:
    high = avg_level
else:
    high = pos["heightlist"].min()
```

Even though *pos* should have the same heightlist for every position (since it was generated using position), I decided to use the `.min()` aggrigation anyway because we're checking if it's above the `average_level` . If it is above or equal to average level then it will just use the number of heights. However, if it is below average then it is missing heights and needs to start at a different number, which will be calculated from the average (so high = average).

Lines 11-13

```
levels = []
for i in range(len(pos)):
    levels.append(high - i)
```

`levels` is a list of all of the heights. As the loop ittorates, `i` will become bigger and as we append make levels count down. Usually `high` matches the count of rows, so levels counts down to one `[5, ..., 1]`. However, if it was determined to not have enough levels it will only count down from average `[5, ..., 2]`.

Line 14

```
centers.loc[(centers["position"] == position), "level"] = levels
```

This maps the correct height to the centers.csv using the levels list using `.loc`. It filters down to all the rows with the same position, then defines the column `[level]` to match the list `levels`. The filtered centers.csv and the `levels` list must be the same length or else there will be errors, but because we calculated the `levels` list using the length of the filtered dataframe they should always match.