CROWN CASE STUDY

By: Swathi Subramanyan

Date: 19-12-2024

Environment: Python 3

Libraries used:

- pandasql (to write sql commands on pandas dfs)
- pandas (for data manipulation)

Table of Contents

1. Introduction

This notebook contains the code for data exploration, data quality checks and the first 3 SQl questions from the exercise

The data (Restaurant members and orders dataset) can be downloaded from Kaggle at: https://www.kaggle.com/datasets/vainero/restaurants-customers-orders-dataset (https://www.kaggle.com/datasets/vainero/restaurants-customers-orders-dataset)

The data details the operations of a group of restaurants and covers locations, types of restaurants, meal types, order and member details.

My Assumptions:

1. The restaurants belong to 'The Crown Group'
2. Member definition - The resrtaurants award paid memberships to people who frequent it. They are provided a monthly budget within which meals are free. If they exceed the limit, they are awarded a commission beteen 10-12% on the exceeded amount.

2. Importing Libraries & Files

In [1]:
```
import pandas as pd
import csv
import pandasql as ps
```

In [2]:
```
# importing data
cities_df = pd.read_csv('cities.csv')
meal_types_df = pd.read_csv('meal_types.csv')
meals_df = pd.read_csv('meals.csv')
members_df = pd.read_csv('members.csv')
monthly_member_totals_df = pd.read_csv('monthly_member_totals.csv')
order_details_df = pd.read_csv('order_details.csv')
orders_df = pd.read_csv('orders.csv')
restaurant_types_df = pd.read_csv('restaurant_types.csv')
restaurants_df = pd.read_csv('restaurants.csv')
serve_types_df = pd.read_csv('serve_types.csv')
```

3. Data Exploration

In [41]:
```
cities_df.head()
```

Out[41]:

	id	city
0	1	Tel Aviv
1	2	Ramat Gan
2	3	Ramat Hasharon
3	4	Herzelia
4	5	Givatayim

All the restaurants are located in Israel. Currency will be in New Shekel.

In [4]:
```
meal_types_df.head()
```

Out[4]:

	id	meal_type
0	1	Vegan
1	2	Cheese
2	3	Beef
3	4	Chicken

They have 4 types of meals offered across restaurants

In [5]:
```
meals_df.head()
```

Out[5]:

	id	restaurant_id	serve_type_id	meal_type_id	hot_cold	meal_name	price
0	1	1	1	2	Cold	Meal 1	43.22
1	2	1	2	4	Hot	Meal 2	29.22
2	3	1	3	1	Cold	Meal 3	37.34
3	4	1	3	2	Hot	Meal 4	52.41
4	5	1	3	2	Cold	Meal 5	27.75

In [6]:
```
members_df.head()
```

Out[6]:

	id	first_name	surname	sex	email	city_id	monthly_budget
0	1	Ollie	Kinney	M	Ol.Kinney@walla.co.il	4	1000.0
1	2	Landon	Bishop	F	La.Bi@gmail.com	4	500.0
2	3	Jia	Delarosa	M	Jia.De@gmail.com	4	500.0
3	4	Valentina	Ratcliffe	F	Va.Ratcliffe@gmail.com	4	600.0
4	5	Stacie	Patel	F	St.Patel@hotmail.com	3	500.0

In [7]:
```
monthly_member_totals_df.head()
```

Out[7]:

	member_id	first_name	surname	sex	email	city	year	month	order_count	meals_count	monthly_budget	total_expense	balance	commission
0	47	Joyce	Newton	F	Joyce.Ne@gmail.com	Herzelia	2020	1	17	37	1836.15	500.0	-1336.15	136.27950
1	126	Macey	Almond	M	Macey.Almond@yahoo.com	Tel Aviv	2020	1	30	64	2676.98	1000.0	-1676.98	214.98500
2	68	Aydin	Hirst	M	Aydin.Hirst@hotmail.com	Tel Aviv	2020	1	24	52	2286.53	1000.0	-1286.53	164.93850
3	193	Mira	Kent	M	Mi.Kent@walla.co.il	Tel Aviv	2020	1	24	54	2547.62	500.0	-2047.62	193.59125
4	53	Lilly-Ann	Frey	F	Li.Fr@hotmail.com	Tel Aviv	2020	1	23	50	2456.64	1000.0	-1456.64	193.97650

Members had to be defined according to the information available in this table. Monthly Budget and Expense columns are interchanged. They will be addressed when visualising with PowerBI.

In [8]:
```
order_details_df.head()
```

Out[8]:

	id	order_id	meal_id
0	1	3	176
1	2	5	349
2	3	5	348
3	4	5	344
4	5	5	348

In [9]:
```
orders_df.head()
```

Out[9]:

	id	date	hour	member_id	restaurant_id	total_order
0	1	2020-01-01	11:00:00.0000000	25	6	0.0
1	2	2020-01-01	11:08:00.0000000	122	4	0.0
2	3	2020-01-01	11:10:00.0000000	62	16	39.0
3	4	2020-01-01	11:13:00.0000000	171	9	0.0
4	5	2020-01-01	11:13:00.0000000	152	30	153.0

The data type of the hour column will need to be sorted to time.

In [10]:
```
restaurant_types_df.head()
```

Out[10]:

	id	restaurant_type
0	1	Fast Food
1	2	Asian
2	3	Italian
3	4	Homemade
4	5	Indian

There are only 5 cuisines offered

In [11]:
```
restaurants_df.head()
```

Out[11]:

	id	restaurant_name	restaurant_type_id	income_persentage	city_id
0	1	Restaurant 1	3	0.075	3
1	2	Restaurant 2	5	0.100	3
2	3	Restaurant 3	2	0.075	4
3	4	Restaurant 4	4	0.100	4
4	5	Restaurant 5	5	0.050	2

In [12]:
```
serve_types_df.head()
```

Out[12]:

	id	serve_type
0	1	Starter
1	2	Main
2	3	Desert

In [42]:
```
# storing the data in a dict for easy parsing
dataframes = {
    'cities': cities_df,
    'meal_types': meal_types_df,
    'meals': meals_df,
    'members': members_df,
    'monthly_member_totals': monthly_member_totals_df,
    'order_details': order_details_df,
    'orders': orders_df,
    'restaurant_types': restaurant_types_df,
    'restaurants': restaurants_df,
    'serve_types': serve_types_df,
}

# parsing the dict for other summary info
for name, df in dataframes.items():
    print(f"\n=========== Summary of {name} DataFrame: ===========\n")
    print(f"Shape: {df.shape}\n")
    print(f"Columns: {df.columns.tolist()}\n")
    print("Missing values:\n", df.isnull().sum(),"\n")
    print("Data Types:\n", df.dtypes,"\n")
```

```
city_id                int64
dtype: object


=========== Summary of serve_types DataFrame: ===========

Shape: (3, 2)

Columns: ['id', 'serve_type']

Missing values:
 id            0
serve_type    0
dtype: int64

Data Types:
 id             int64
serve_type    object
dtype: object
```

The above result was used to check the following:

1. **Number of rows and columns of the data**
2. **Missing values**
3. **Data Types**

Results show that there is no missing data and the data types have all been read correctly.

orders, order_details and monthly_member_totals are the biggest files.

The tables - **cities, meal_types, restaurant_types and serve_types do not need any cleaning.**

In [43]:
```
# Check for duplicates in all DataFrames
print("Number of duplicated rows per table\n")

for name, df in dataframes.items():
    duplicate_count = df.duplicated().sum()
    print(f"{name}: {duplicate_count}")
```

```
Number of duplicated rows per table

cities: 0
meal_types: 0
meals: 0
members: 0
monthly_member_totals: 0
order_details: 0
orders: 0
restaurant_types: 0
restaurants: 0
serve_types: 0
```

There are **no duplicated rows in any of the tables**

In [16]:

```python
# Check the number of unique values per column in each DataFrame
for name, df in dataframes.items():

    print(f"\nUnique value count for {name}:\n")
    print(df.nunique())
    print(f"Shape: {df.shape}")
```

```
Unique value count for cities:

id      5
city    5
dtype: int64
Shape: (5, 2)

Unique value count for meal_types:

id           4
meal_type    4
dtype: int64
Shape: (4, 2)

Unique value count for meals:

id               350
restaurant_id     30
serve_type_id      3
meal_type_id       4
hot_cold           2
meal_name        350
price            332
dtype: int64
Shape: (350, 7)

Unique value count for members:

id                200
first_name        198
surname           192
sex                 2
email             200
city_id             5
monthly_budget      4
dtype: int64
Shape: (200, 7)

Unique value count for monthly_member_totals:

member_id          200
first_name         198
surname            192
sex                  2
email              200
city                 5
year                 1
month                6
order_count         35
meals_count         76
monthly_budget    1200
total_expense        4
balance           1199
commission        1199
dtype: int64
Shape: (1200, 14)

Unique value count for order_details:

id          70577
order_id    30782
meal_id       350
dtype: int64
Shape: (70577, 3)

Unique value count for orders:

id               36000
date               182
hour               780
member_id          200
restaurant_id       30
total_order       9413
dtype: int64
Shape: (36000, 6)

Unique value count for restaurant_types:

id                 5
restaurant_type    5
dtype: int64
Shape: (5, 2)
```

```
Unique value count for restaurants:

id                   30
restaurant_name      30
restaurant_type_id    5
income_persentage     3
city_id               5
dtype: int64
Shape: (30, 5)

Unique value count for serve_types:

id            3
serve_type    3
dtype: int64
Shape: (3, 2)
```

This information was useful to **cross validate the integrity** of the data across different tables, for example, is the number of restaurant IDs the same in the restaurants and orders.

This shows that there are **5 unique cities, 30 restaurants, 5 cuisines, 200 members, 350 meal types, 3 serve types. **

Most importantly, we can know **the primary keys of each table** based on this output. Most tables have unique ID columns except monthly_member_totals in which the primary key is the combination of member, year and month. (adding year assuming the data is updated)

The number of unique prices are less than number of meals. Let us investigate this:

In [17]:
```python
# Group by 'price' and count the number of unique meal names
meals_with_same_price = meals_df.groupby('price')['meal_name'].nunique()

# Filter prices where more than one unique meal shares the same price
prices_with_multiple_meals = meals_with_same_price[meals_with_same_price > 1]

# Display results
if not prices_with_multiple_meals.empty:
    print("Prices shared by more than one meal:")
    print(prices_with_multiple_meals)
else:
    print("No prices are shared by more than one meal.")
```

```
Prices shared by more than one meal:
price
24.43    2
31.07    3
31.69    2
32.18    2
33.51    2
34.95    2
35.57    3
36.65    2
39.21    2
39.87    2
40.00    2
48.23    2
50.87    2
64.13    2
71.06    2
76.28    2
Name: meal_name, dtype: int64
```

Thus, we conclude that **more than meal has the same price.**

In [18]:
```python
full_name_duplicates = members_df.duplicated(subset=['first_name', 'surname']).sum()
print(f"Number of duplicate 'first_name' and 'surname' pairs: {full_name_duplicates}")
```

```
Number of duplicate 'first_name' and 'surname' pairs: 0
```

In [19]:
```
# Check the summary stats in each DataFrame
for name, df in dataframes.items():

    print(f"\nSummary Stats for {name}:\n")

    print(df.describe())
```

```
                id      order_id       meal_id
count  70577.000000  70577.000000  70577.000000
mean   35289.000000  18022.653386    174.097992
std    20373.969311  10407.579887    105.387164
min        1.000000      3.000000      1.000000
25%    17645.000000   8957.000000     80.000000
50%    35289.000000  18070.000000    175.000000
75%    52933.000000  27049.000000    268.000000
max    70577.000000  36000.000000    350.000000

Summary Stats for orders:

                id    member_id  restaurant_id   total_order
count  36000.000000  36000.00000   36000.000000  36000.000000
mean   18000.500000    100.94375      15.530083     87.899680
std    10392.449182     57.72601       8.654903     64.939033
min        1.000000      1.00000       1.000000      0.000000
25%     9000.750000     51.00000       8.000000     36.710000
50%    18000.500000    101.00000      16.000000     78.660000
75%    27000.250000    151.00000      23.000000    129.620000
```

Summary stats is useful to check **the range of values in a numerical column, any anomalous max or min values.** Based on the above results, **the orders column shows a minimum price of 0 New Shekel. This seems to be suspicious**

Let us investigate this:

In [20]:
```
# Sort the orders_df by 'total_price' in descending order
orders_sorted_df = orders_df.sort_values(by='total_order')

print("First 5 rows of the sorted orders table:")
print(orders_sorted_df.head())
```

```
First 5 rows of the sorted orders table:
          id        date              hour  member_id  restaurant_id  \
0          1  2020-01-01  11:00:00.0000000         25              6
23508  23509  2020-04-29  11:04:00.0000000        113             10
23514  23515  2020-04-29  11:18:00.0000000         78             29
23515  23516  2020-04-29  11:21:00.0000000        110             12
6552    6553  2020-02-03  17:21:00.0000000         18             21

       total_order
0              0.0
23508          0.0
23514          0.0
23515          0.0
6552           0.0
```

The above sample table suggests that atleast a few members have purchased specifc orders at the restaurant that costed them nothing.

which meals have they purchsed, for those whom the total_order is 0.0. if it is a miscalculation we can calculate the actual price for it. What is the original price supposed to be?

In [22]:
```
# to find the meal ids of those orders whch have 0 price. maybe i can calculate new pricebased on meals?

q1 = """
    WITH zero_cost_df AS (
        SELECT id
        FROM orders_df
        WHERE total_order = '0.0'
    ),

    zero_meals_df AS (
    SELECT meal_id
    FROM zero_cost_df zc
    INNER JOIN order_details_df od
    ON zc.id == od.order_id)

    SELECT meal_id
    FROM zero_meals_df
    """

print(ps.sqldf(q1, locals()))
```

```
Empty DataFrame
Columns: [meal_id]
Index: []
```

The above SQL code results in no results for corresponding meal_ids for these orders where price was. I believe there is **more investigation is needed to decipher why these transactions exist in the orders table if a meal has not been bought.**

In [23]:
```
# to find count of orders that have 0 price

q2 = """
    SELECT total_order AS price, count(distinct id) AS count_orders
    FROM orders_df
    WHERE total_order == '0.0'
    GROUP BY 1
    """

print(ps.sqldf(q2, locals()))
```

```
   price  count_orders
0    0.0          5218
```

5218 orders YTD have no price attached to them

In [ ]:
```
# each meal is served only at one restaurant

q6 = """
    SELECT id, count(distinct restaurant_id)
    FROM meals_df
    GROUP BY 1
    ORDER BY 2 desc
    """

print(ps.sqldf(q6, locals()))
```

4. Exercise SQL Questions

1. How many members in each of the cities?

In [35]:
```
q3 = """
    SELECT c.city as City, count(distinct m.id) as "Total Members"
    FROM members_df m
    LEFT JOIN cities_df c
    ON m.city_id = c.id
    GROUP BY c.city
    ORDER BY 2 desc
    """

print(ps.sqldf(q3, locals()))
```

```
            City  Total Members
0       Herzelia             48
1  Ramat Hasharon             43
2      Givatayim             42
3       Tel Aviv             38
4      Ramat Gan             29
```

The SQL query retrieves the total number of distinct members for each city by joining the members_df table with the cities_df table using a LEFT JOIN on the city_id field. The query counts the distinct member IDs (m.id) for each city, groups the results by city name, and orders the cities in descending order based on the total number of members. The final output includes the city name and the corresponding member count.

2. Which cities have the most vegan meals by members and orders?

In [40]:
```
q4 = """

WITH vegan_orders as (
    SELECT m.id as vo_id

    FROM
    order_details_df od
    LEFT JOIN meals_df m
    ON od.meal_id = m.id

    LEFT JOIN meal_types_df mt
    ON m.meal_type_id = mt.id

    WHERE mt.meal_type = 'Vegan'),

city_vegan as (

    SELECT r.city_id as city_id, count(distinct vo.vo_id) as count_orders

    FROM vegan_orders vo
    LEFT JOIN orders_df o
    ON vo.vo_id = o.id

    LEFT JOIN restaurants_df r
    on o.restaurant_id = r.id

    GROUP BY r.city_id)

SELECT c.city as City, cv.count_orders as "# Vegan Orders"
FROM city_vegan cv
LEFT JOIN cities_df c on cv.city_id = c.id
ORDER BY 2 desc

    """

print(ps.sqldf(q4, locals()))
```

```
            City  # Vegan Orders
0      Ramat Gan              30
1  Ramat Hasharon              30
2       Herzelia              21
3       Tel Aviv              19
4      Givatayim               8
```

This query calculates the number of vegan orders per city by utilizing **Common Table Expressions (CTEs)** to break the process into manageable steps:

vegan_orders CTE: This step retrieves the IDs of orders containing vegan meals. It joins the order_details_df table with the meals_df and meal_types_df tables to filter for vegan meals based on the meal_type field.

city_vegan CTE: This step aggregates the number of distinct vegan orders for each city. It joins the results from the vegan_orders CTE with the orders_df and restaurants_df tables to group the data by city_id and count the vegan orders.

The final SELECT query retrieves the city name and the corresponding count of vegan orders. It joins the city_vegan results with the cities_df table to map the city_id to the actual city name and orders the cities by the count of vegan orders in descending order.

3. What is the proportion of serve types for the Italian restaurant?

In [37]:
```
q5= """

WITH italian_orders as (

    SELECT o.id as order_id

    FROM orders_df o
    LEFT JOIN restaurants_df r
    ON o.restaurant_id = r.id

    LEFT JOIN restaurant_types_df rt
    ON r.restaurant_type_id = rt.id

    where rt.restaurant_type == 'Italian'

    ),

italian_serve as (

    SELECT io.order_id as order_id, m.serve_type_id as serve_type_id

    FROM italian_orders io
    LEFT JOIN order_details_df od
    ON io.order_id = od.order_id

    LEFT JOIN meals_df m
    ON od.meal_id = m.id

),

serve_counts as (

    SELECT st.serve_type as serve_type, count(its.serve_type_id) as total_meals_served
    FROM italian_serve its
    LEFT JOIN serve_types_df st
    ON its.serve_type_id = st.id
    WHERE its.serve_type_id IS NOT NULL
    GROUP BY 1

)

SELECT
    serve_type as "Serve Type",
    total_meals_served as "Total Meals",
    ROUND(total_meals_served * 1.0 / SUM(total_meals_served) OVER (), 4) AS Proportion
    FROM serve_counts
    ORDER BY 2 DESC;


    """

print(ps.sqldf(q5,locals()))
```

```
  Serve Type  Total Meals  Proportion
0     Desert         3842      0.3935
1       Main         3813      0.3906
2    Starter         2108      0.2159
```

This query calculates the distribution of meal serve types for Italian restaurant orders. It uses **Common Table Expressions (CTEs)** to organize the process into distinct steps:

italian_orders CTE: This step identifies the order IDs from Italian restaurants by joining the orders_df table with the restaurants_df and restaurant_types_df tables to filter orders from Italian restaurants based on the restaurant_type field.

italian_serve CTE: This step retrieves the serve type information for the identified Italian orders. It joins the italian_orders CTE with the order_details_df and meals_df tables to associate the meals with their corresponding serve types.

serve_counts CTE: This step counts the number of meals served by each serve type, grouping by the serve_type field. It uses the serve_types_df table to get the name of the serve type and calculates the total meals served for each serve type.

The final SELECT query retrieves the serve type, the total number of meals served, and the proportion of each serve type relative to the total number of meals served. The results are ordered by the total meals served in descending order.

In [ ]: