FoodPro – Online Food Delivery Application Database System & Analytics

By:
Prajwal Srinivas

Executive Summary: The objective of the study was to build a robust data management system for an online food ordering/delivery application called FoodPro. The system is to handle both transactional data (payment and order information) and Reference Data (User Account data, Address Information etc), and should be the single source of information about anything related to order customer, restaurant, and the delivery agents.

We were able to not only implement the system but also utilise the power of analytics to predict optimization activities to the restaurants and the application alike. We first created an EER model, which was later translated into a UML class diagram. The relation modelling was done considering the cardinalities and the relationships mentioned in the EER diagram, and later the relations were later normalised to 3.5 Normal Form. The relational model was materialised in the form of a database inside MySQL. A NoSQL component (in the form of MongoDB) was also implemented to utilise the power of document method of storage of information and the power of NoSQL pipelines. The storage system was later connected to a Python environment which enabled advanced analytical capabilities which enables using predictive analytics to derive actionable insights.

The whole development was done iteratively rather than in a waterfall flow, the EER/UML diagram was revisited to accommodate for the information which was made visible at the later stages of development of the system. This enabled us to explore a multitude of solutions prior agreeing on a particular pathway and increased our learning greatly in the process.

I. Introduction

FoodPro is an application where users can order food from various restaurants. For users to be able to use the application and place orders, they must create an account on the application with their email and an address; post which each accounts given a unique account_id.

After that the users are given two options regarding their account type,
1. Free Account
2. Gold Account

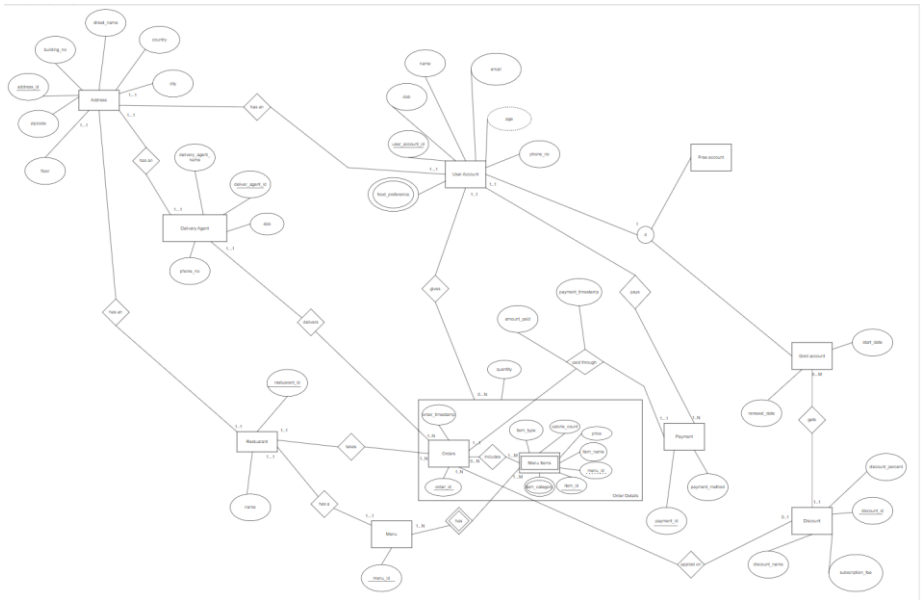
For a Gold Account, a user must pay a subscription fee, post which they get access to discounts for a period of 100 days. The different subscription options are gold, gold_plus and gold_pro.

The users get to choose various restaurants, where every restaurant has a unique restaurant_id; after choosing the restaurant the user selects the items to order from the restaurant menu. Every menu has a unique menu_id and different course_names. The order placed by the user account is assigned to delivery agent who would deliver the order. The delivery agent has a unique agent_id and is given order details and user address to fulfil the order.

The focus of this project is to effectively maintain the order management system and subsequently come up with optimization of the process assignment of delivery agents across the cities. It would also aid the restaurant decide on the inventory for the popular dishes and understand the customer demography/preferences of the user better, and appropriately come up with effective campaigns for selling the appropriate gold subscription plans. The organised structure of the relational database system also enables us in generating useful metrics/economic dashboards (like a mini balance sheet). We would be implementing the required system using a combination of excel csv's, MySQL, MongoDB, and Python.

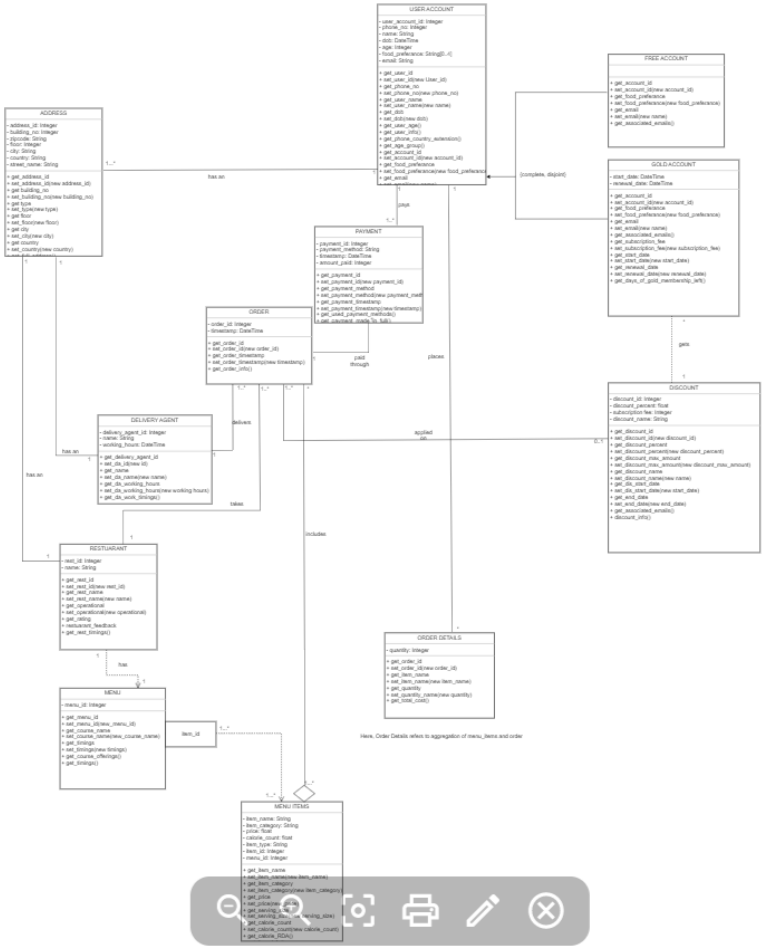
II. Conceptual Data Modelling

EER Diagram:



DMA_Project_EER.html

UML Class Diagram:



DMA_Project_UML.html

III. Mapping Conceptual Model to Relational Model

All relations are normalised to 3.5NF.

(open in MySQL)

Relation – 1:
address (address_id, building_no, street_name, floor, city, zipcode) (each user_account, delivery_agent, and a restaurant can have a single unique address and not multiple addresses)
The address relation contains the address information of the people involved in the FoodPro app – users, restaurants, and the delivery agents.
address_id is the primary_key

Relation – 2:
user_account (user_account_id, dob, email, phone_no, name, age, *address_id*)
The user_account relation contains the information of the user who would be creating an account with the FoodPro application and use the user account details to make an order.
user_account_id is the primary key
address_id is the foreign key referencing the primary key of the relation address – NOT NULL

Relation – 3:
user_account_food_preference (food_preference_type, *user_account_id*) (ER diagram's multi values attribute's relation)
The user_account_food_preference relation holds the information about the user accounts and their associated food preferences (veg, non-veg, egg, vegan, keto, lactose free, diabetes friendly, atleast one preference to be selected)
Food_preference_type, user_account_id – both together form the primary key
user_account_id is the foreign key referring to the primary key of the relation user_account - NOT NULL

Relation – 4:
free_account (*user_account_id)*
Specialisation of relation Account.
user_account_id is the primary key; also, it is the foreign key referencing the user_account parent class – NOT NULL

Relation – 5:
gold_account (*user_account_id*, start_date, renewal_date, *discount_id*)
Specialisation of the relation user_account. Gold account refers to a subscription based membership where the gold users are given discounts on the order from time to time. Renewal_date is 100 days from the start_date. Each subscription is valid for 100 days.
discount_id is the foreign key referencing the primary key of the relation discount – discount_id is NOT NULL
user_account_id is the primary key; also, it is the foreign key referencing the user_account parent class – NOT NULL

Relation – 6:
discount (discount_id, discount_percent, subscription_fee, discount_name)

The discount relation contains the information about the different discounts offered to the gold user based on the subscription package they opted for.
discount_id is the primary key

Relation – 7:
payment (payment_id, payment_method, payment_timestamp, amount_paid, *order_id, user_account_id*)
The payment relation contains information about the payment made by the accounts for the orders they placed.
payment_id is the primary key
user_account_id is the foreign key referring to the primary key of the relation user_account – NOT NULL
order_id is the foreign key referring to the relation order – NOT NULL

Relation – 8:
order_details (*order_id*, *menu_id , item_id*, quantity, *user_account_id*, cost) (This if for the order details aggregation)
The order details relation contains the information of the individual order details places by any user account (the details of the order, items ordered, the time).
order_id, item_id, menu_id together form the primary key
user_account_id is the foreign key referring to the primary key of the relation user_account
item_id is the foreign key referring to the primary key of the relation menu item
menu_id is the foreign key referring to the primary key of the relation menu (as menu item is a weak entity)

Relation – 9:
orders (order_id, order_timestamp, *delivery_agent_id, restaurant_id, discount_id*)
The order relation contains the information of the individual order.
order_id is the primary key
restaurant_id is the foreign key referring to the primary key of the relation restaurant – NOT NULL
delivery_agent_id is the foreign key referring to the primary key of the relation delivery agent – NOT NULL
discount_id is the foreign key referring to the primary key of the relation discount – can be NULL (not applicable)

Relation – 10:
menu (menu_id, *restaurant_id*)
The menu relation contains the information of the menu which is associated with a particular restaurant.
menu_id is the primary key
restaurant_id is the foreign key referring to the primary key of the relation restaurant – NOT NULL

Relation – 11:
menu_item (*menu_id,* item_id, item_name, calorie_count, item_type, price)
The menu_item relation contains the information of items in a particular menu. It is a weak entity (according to the ER diagram), so it borrows the primary key from the menu relation.
menu_id, item_id together form the primary key
menu_id is the foreign key referring to the primary key of the relation menu – NOT NULL

Relation – 12:
menu_item_category (*menu_id, item_id*, item_category)
The menu_item_category relation contains the information of category (veg, non-veg, egg, keto, diabetes friendly) of the different menu items.
menu_id, item_id, item_category together form the primary key
menu_id is the foreign key referring to the primary key of the relation menu – NOT NULL
menu_item_id is the foreign key referring to the primary key of the relation menu_item – NOT NULL

Relation – 13:
restaurant (restaurant_id, restaurant_name, *address_id*) (all restaurants are cloud kitchens which accept orders throughout the day)
The restaurant relation contains the information of the restaurants which are registered with the FoodPro app from where the customers place the order.
restaurant_id is the primary key
address_id is the foreign key referring to the primary key of the relation Address – NOT NULL

Relation – 14:
delivery_agent (delivery_agent_id, delivery_agent_name, dob, phone_no, *address_id*)
The delivery_agent relation contains the information of the delivery agents who are registered with the FoodPro app, who would be delivering the orders placed by the user accounts.
delivery_agent_id is the primary key
address_id is the foreign key referring to the primary key of the relation address – NOT NULL

IV. Implementation of Relation Model via MySQL and NoSQL

Analytical SQL Queries:

DMA_Group_28_SQL_Scripts.sql (Open in MySQL)

1. First, let us look at the order count split between the free and the gold users:

```
WITH non_disc AS
(SELECT o.order_id, ua.user_account_id, o.discount_id
FROM orders o, order_details od, user_account ua
WHERE o.order_id = od.order_id AND od.user_account_id = ua.user_account_id AND o.discount_id = 'not applicable'),

count_non_disc AS
(SELECT COUNT(DISTINCT non_disc.order_id) count_of_non_discounted_orders FROM non_disc),

count_disc AS
(SELECT COUNT(DISTINCT od.order_id) - count_non_disc.count_of_non_discounted_orders count_of_discounted_orders FROM order_details od, count_non_disc)

SELECT count_non_disc.count_of_non_discounted_orders count_of_free_user_orders, count_disc.count_of_discounted_orders count_of_gold_user_order
FROM count_non_disc, count_disc
WHERE count_non_disc.count_of_non_discounted_orders IS NOT NULL;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

count_of_free_user_orders	count_of_gold_user_order
182	110

From the output, it seems that the number of orders made by the free users are higher than the gold users. To better understand this split, we need to further investigate the number of orders made by each group of users and the average order per customer of the two groups.

2. Now, let us find the average no of orders per gold customer and free account customer.

```
WITH non_disc AS
(SELECT o.order_id, ua.user_account_id, o.discount_id
FROM orders o, order_details od, user_account ua
WHERE o.order_id = od.order_id AND od.user_account_id = ua.user_account_id AND o.discount_id = 'not applicable'),

count_non_disc AS
(SELECT COUNT(DISTINCT non_disc.order_id) count_of_non_discounted_orders FROM non_disc),

count_disc AS
(SELECT COUNT(DISTINCT od.order_id) - count_non_disc.count_of_non_discounted_orders count_of_discounted_orders FROM order_details od, count_non_disc),

order_count AS
(SELECT count_non_disc.count_of_non_discounted_orders, count_disc.count_of_discounted_orders
FROM count_non_disc, count_disc
WHERE count_non_disc.count_of_non_discounted_orders IS NOT NULL)

SELECT ROUND((oc.count_of_non_discounted_orders/(SELECT COUNT(*) FROM free_account)), 2) avg_order_per_free_user,
       ROUND((oc.count_of_discounted_orders/(SELECT COUNT(*) FROM gold_account)), 2) avg_order_per_gold_user
FROM order_count oc;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

avg_order_per_free_user	avg_order_per_gold_user
1.53	2.75

The gold users order more often than the regular free user, this could be attributed to the fact that the gold user enjoys the discount associated with the gold account benefits.

3. Now, let us also see the most popular gold subscription, and the average order per respective gold subscription plan of that customer.

```
WITH gold_non_gold_order_count AS
(SELECT o.discount_id, COUNT(o.discount_id) no_of_orders
FROM orders o
GROUP BY o.discount_id),

gold_user_dist AS
(SELECT ga.discount_id, COUNT(DISTINCT ga.user_account_id) no_of_users
FROM gold_account ga
GROUP BY ga.discount_id)

SELECT gud.discount_id, gud.no_of_users, gc.no_of_orders, (gc.no_of_orders/gud.no_of_users) avg_order_per_user
FROM gold_user_dist gud, gold_non_gold_order_count gc
WHERE gud.discount_id = gc.discount_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

discount_id	no_of_users	no_of_orders	avg_order_per_user
d_gold	13	37	2.8462
d_gold_plus	18	50	2.7778
d_gold_pro	9	23	2.5556

Looks like the gold plus subscription plan is the most popular, there are on average 100% and 40% more orders in comparison to the gold pro and gold orders respectively. Also, in contrast to the subscribers for each gold package the average order per user is highest for gold users, followed by gold plus and then gold pro.

4. What is the average order value - overall, by non-gold and gold customers?

```
WITH tot_avg_val AS
(SELECT 1, od.order_id, SUM(cost) total_order_value
FROM order_details od
GROUP BY od.order_id),

gol_avg_val AS
(SELECT 1, ga.user_account_id, tot_avg_val.total_order_value
FROM tot_avg_val, order_details od2, user_account ua, gold_account ga
WHERE tot_avg_val.order_id = od2.order_id AND ua.user_account_id = od2.user_account_id AND ga.user_account_id = ua.user_account_id),

free_avg_val AS
(SELECT 1, fa.user_account_id, tot_avg_val.total_order_value
FROM tot_avg_val, order_details od2, user_account ua, free_account fa
WHERE tot_avg_val.order_id = od2.order_id AND ua.user_account_id = od2.user_account_id AND fa.user_account_id = ua.user_account_id),

t1 AS
(SELECT AVG(tot_avg_val.total_order_value) total_order_value FROM tot_avg_val),

t2 AS
(SELECT AVG(gol_avg_val.total_order_value) gold_total_order_value FROM gol_avg_val),

t3 AS
(SELECT AVG(free_avg_val.total_order_value) free_total_order_value FROM free_avg_val)

SELECT ROUND(t1.total_order_value, 2) cumulative_avg_order_value, ROUND(t2.gold_total_order_value, 2) gold_avg_order_value,
       ROUND(t3.free_total_order_value, 2) free_avg_order_value
FROM t1, t2, t3
WHERE t1.total_order_value IS NOT NULL;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

cumulative_avg_order_value	gold_avg_order_value	free_avg_order_value
338.45	356.85	341.81

We can see that the average order value of gold members is larger in comparison to the free account users.
It would be interesting to see if the story remains even after we take into account the gold discount offered to the gold members.

5. Now let's see average order value of the gold members after taking into account the discount offered on the orders.

```
SELECT ROUND(AVG(t1.amt_paid_gold), 2) gold_total_order_value_after_discount
FROM (SELECT p.order_id, SUM(p.amount_paid) amt_paid_gold
FROM payment p, user_account ua, gold_account ga
WHERE p.user_account_id = ua.user_account_id AND ua.user_account_id = ga.user_account_id
GROUP BY p.order_id) t1;
```

Result Grid | Filter Rows:

gold_total_order_value_after_discount
312.43

The average order value is around 8% lesser than the total average order value for gold users, when we consider the discounts provided. In conclusion, the average order value is greater for gold users than free members, considering the additional subscription fee, it is beneficial for the company if a particular user moves from the free group to the gold group. Also, the gold plus is the most popular gold subscription, probably attributable to the anchoring effect of the gold plus subscription. We could optimise the pricing of the gold subscriptions even more, optimise the discount rates by looking into predicting future order/subscription patterns or even perform A/B testing for the optimal pricing of the subscription plan/discount rates. The above analysis completely focuses on the customer and their past behaviour/pattern.

6. Now, let's explore the dataset based on geography, restaurant, menu item etc. Let's look at the most popular location, restaurant, and the menu items.
   First, the top 5 locations

```
SELECT a.city, COUNT(DISTINCT od.order_id) no_of_orders
FROM address a, user_account ua, order_details od
WHERE od.user_account_id = ua.user_account_id AND ua.address_id = a.address_id
GROUP BY a.city
ORDER BY no_of_orders DESC
LIMIT 5;
```

Result Grid | Filter Rows:

city	no_of_orders
Albuquerque	41
Indianapolis	36
Louisville	36
Fresno	31
Jacksonville	31

Result 218

Considering, the output of the above the top 5 cities are Albuquerque, Indianapolis, Louisville, Fresno, Jacksonville. We should move towards assigning a higher percentage of delivery agents in the above cities to optimise the delivery timings. Also, we could push more restaurants in the above areas to tie up with the FoodPro app, considering the high number of orders in the app in the above cities.

Next, the top 5 restaurants

```
WITH t1 AS
(SELECT r.restaurant_name, COUNT(o.order_id) no_of_orders
FROM restaurant r
JOIN orders o
ON r.restaurant_id = o.restaurant_id
GROUP BY r.restaurant_name
ORDER BY no_of_orders DESC)

SELECT * FROM t1;
```

Result Grid | Filter Rows:

restaurant_name	no_of_orders
Kitchen Al Dente	41
Pickled Kitchen	34
Flambe Kitchen	32
Exquisite Kitchen	30
Kitchen Piquant	29
Ambrosial Kitchen	28

Result 219

From the above list we could approach the restaurants to make optimizations in cooking time to deliver the online orders through FoodPro. We could also consider starting a collaboration with the restaurant, to automatically sign up with FoodPro whenever they open a new branch; considering the popularity of the restaurant, the new branches are also expected to do well. This would further help in reducing the wait time of the delivery agents who would deliver the order from the above restaurants to the customer.

Now, let's look at the most popular menu items

```
SELECT mi.item_name, SUM(od.quantity) no_of_units_ordered
FROM order_details od, menu_item mi
WHERE od.menu_id = mi.menu_id AND od.item_id = mi.item_id
GROUP BY mi.item_name
ORDER BY no_of_units_ordered DESC;
```

Result Grid | Filter Rows: | Export:

item_name	no_of_units_ordered
Spaghetti with meatballs	353
French fries	292
Paneer Pesto	286
Veg Pizza	272
Diet Coke	271
Chicken Pizza	269

Result 220

Spaghetti with meatballs, French fries, Paneer Pesto, Veg Pizza, Diet Coke are the top 5 items that ordered by the users; the restaurants can be guided to increase the inventory needed to fulfil the above orders. Also, interesting to note that 3 out of the top 5 most ordered items are fast food. Therefore, we can guide the restaurant to revamp the menu items to include more fast food, as they are the most selling and they ideally have higher margins and shorter cooking time.

7. Now, let's see the most popular food_preference that was ordered.

```
SELECT mic.item_category, COUNT(*) no_of_items_ordered
FROM menu_item mi, menu_item_category mic, order_details od
WHERE mi.menu_id = mic.menu_id AND mi.item_id = mic.menu_item_id
        AND od.menu_id = mi.menu_id AND od.item_id = mi.item_id
GROUP BY mic.item_category
ORDER BY no_of_items_ordered DESC;
```

Result Grid | Filter Rows:

item_category	no_of_items_ordered
veg	525
non-veg	233
egg	224
keto	221
diabetes friendly	143

Result 221

Looks like the people preferred to order veg items by more than 100% than the second most ordered food category - non-veg. This could guide the restaurants in looking into adding more vegetarian options on their menu. It is also interesting to note that keto being a category which would be ordered by a niche group of people, sits high up along with non-veg and egg items. This could be an opportunity to come up with more specific diet focused meal plans (vegan/keto/lean) - because these orders are usually ordered by the same group of people in a periodic manner - this would help us in streamlining the delivery process as well, as we would be prepared with a clear schedule about the delivery.

8. Now, let's check the revenue generated (by orders) split based on the day of the week
First, we generate a CTE table to clean the order date column and get the day of the order

```
WITH doy AS
(SELECT DISTINCT DAYOFWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-", SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE)) day_of_week_no,
        CASE DAYOFWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-", SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE))
            WHEN 1 THEN 'Sunday'
            WHEN 2 THEN 'Monday'
            WHEN 3 THEN 'Tuesday'
            WHEN 4 THEN 'Wednesday'
            WHEN 5 THEN 'Thursday'
            WHEN 6 THEN 'Friday'
            WHEN 7 THEN 'Saturday'
            END day_name
        FROM orders o
        ORDER BY day_of_week_no)

SELECT doy.day_name, DAYOFWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-",
                    SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE)) order_day_of_week,
                    SUM(od.cost) total_order_value
FROM order_details od
JOIN orders o
ON o.order_id = od.order_id
JOIN doy
ON doy.day_of_week_no = DAYOFWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-", SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE))
GROUP BY order_day_of_week
ORDER BY total_order_value DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

day_name	order_day_of_week	total_order_value
Sunday	1	18468
Tuesday	3	15764
Thursday	5	15098
Friday	6	14008
Saturday	7	12233
Monday	2	11858
Wednesday	4	11399

Expectedly, the most revenue was generated from orders that were made on Sundays and the least from mid-week - Wednesday. Interestingly, the second least revenue was generated from Saturdays, this could be attributed to the fact that most people would eat out on Saturdays as opposed to ordering in. Considering the above observations, the restaurants and the delivery agents can be guided to accommodate (increase in inventory, open extra time) for the increased orders on Sundays/Wednesdays.

9. Now, let’s create a mini week-wise balance sheet and look at the revenue numbers

```
WITH t1
AS(SELECT DISTINCT YEARWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-", SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE)) week_number,
        ROUND(SUM(od.cost) OVER (PARTITION BY YEARWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(o.order_timestamp, 2), "-",
        SUBSTR(o.order_timestamp, 4, 7)), "%d-%m-%Y") AS DATE))), 2) total_order_value

FROM order_details od, orders o
WHERE o.order_id = od.order_id),

t2 AS
(SELECT YEARWEEK(ga.start_date) week_number, SUM(d.subscription_fee) subscription_fee_collected
FROM gold_account ga, discount d
WHERE ga.discount_id = d.discount_id
GROUP BY 1
ORDER BY 1),

t3 AS
(SELECT DISTINCT YEARWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(p.payment_timestamp, 2), "-", SUBSTR(p.payment_timestamp, 4, 7)), "%d-%m-%Y") AS DATE)) week_number,
        ROUND(SUM(p.amount_paid) OVER (PARTITION BY YEARWEEK(CAST(STR_TO_DATE(CONCAT(LEFT(p.payment_timestamp, 2), "-",
        SUBSTR(p.payment_timestamp, 4, 7)), "%d-%m-%Y") AS DATE))), 2) total_order_value_after_discounts
FROM payment p),

t4 AS
(SELECT t1.week_number, t1.total_order_value, t3.total_order_value_after_discounts, t2.subscription_fee_collected
FROM t1
LEFT JOIN t2
ON t1.week_number = t2.week_number
LEFT JOIN t3
ON t2.week_number = t3.week_number),

t5 AS
(SELECT CONCAT("Year-", LEFT(t4.week_number, 4), ", ", "Week-", RIGHT(t4.week_number, 2)) AS week_number,
        t4.total_order_value, t4.total_order_value_after_discounts,
        ROUND((t4.total_order_value - t4.total_order_value_after_discounts), 2) total_discount_value,
        t4.subscription_fee_collected
FROM t4)

SELECT * FROM t5;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

week_number	total_order_value	total_order_value_after_discounts	total_discount_value	subscription_fee_collected
Year-2022, Week-39	2933	NULL	NULL	NULL
Year-2022, Week-40	9458	9177.96	280.04	105
Year-2022, Week-41	8991	8698.83	292.17	395
Year-2022, Week-42	9826	9452.4	373.6	120
Year-2022, Week-43	14090	13570.66	519.34	500
Year-2022, Week-44	11829	11365.38	463.62	50
Year-2022, Week-45	8220	7931.37	288.63	185
Year-2022, Week-46	14163	13609.26	553.74	205
Year-2022, Week-47	13481	12930.67	550.33	115
Year-2022, Week-48	5837	5582.37	254.63	50

The order value is the highest for the combined consecutive week 46/47, this could be attributed to the thanksgiving week festivities. With people travelling from different places, and staying home, more orders can be expected around the holidays. We can further think about providing some extra discount during the holiday season to boost sales of both the orders as well as the subscriptions plans.

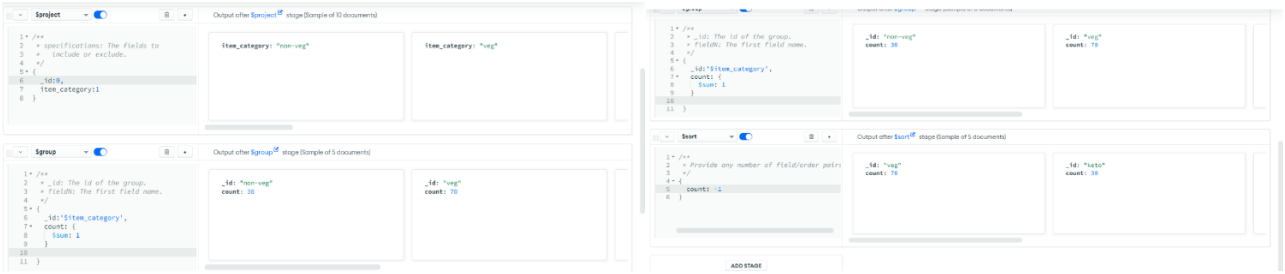
Let's calculate the profit made on a weekly basis considering a flat profit margin of 14% (after deducting the operational cost, rest payments etc on the order value (query is similar to the previous one).

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

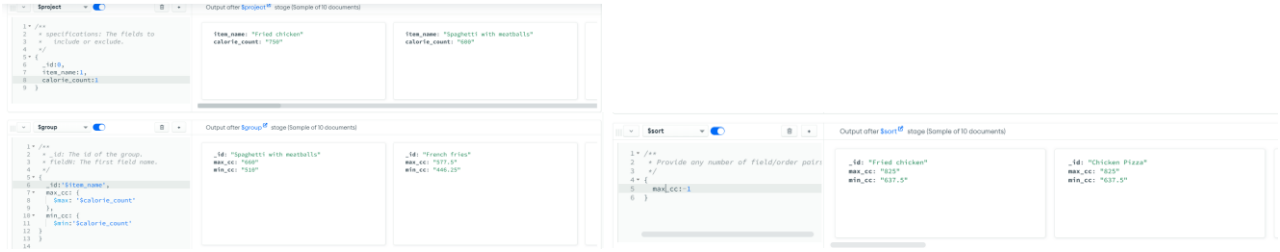
week_number	total_order_value	total_order_value_after_discounts	total_discount_value	subscription_fee_collected	final_profit_value	profit_divided_by_tot_ord_value
Year-2022, Week-39	2933	NULL	NULL	NULL	NULL	NULL
Year-2022, Week-40	9458	9177.96	280.04	105	1149.08	12.15%
Year-2022, Week-41	8991	8698.83	292.17	395	1361.57	15.14%
Year-2022, Week-42	9826	9452.4	373.6	120	1122.04	11.42%
Year-2022, Week-43	14090	13570.66	519.34	500	1953.26	13.86%
Year-2022, Week-44	11829	11365.38	463.62	50	1242.44	10.5%
Year-2022, Week-45	8220	7931.37	288.63	185	1047.17	12.74%
Year-2022, Week-46	14163	13609.26	553.74	205	1634.08	11.54%
Year-2022, Week-47	13481	12930.67	550.33	115	1452.01	10.77%
Year-2022, Week-48	5837	5582.37	254.63	50	612.55	10.49%

Now, looking at the final profit values at a profit margin of 14%, we can clearly see that the final_profit per week is the highest for week 43, and a look at the profit percentages as a proportion of the total_order_value, we ca see that the profit % is highest for week 41, which can be attributed to the high number of gold subscriptions and the lesser discounts because fewer people were using the gold subscription discounts.

NoSQL Queries:

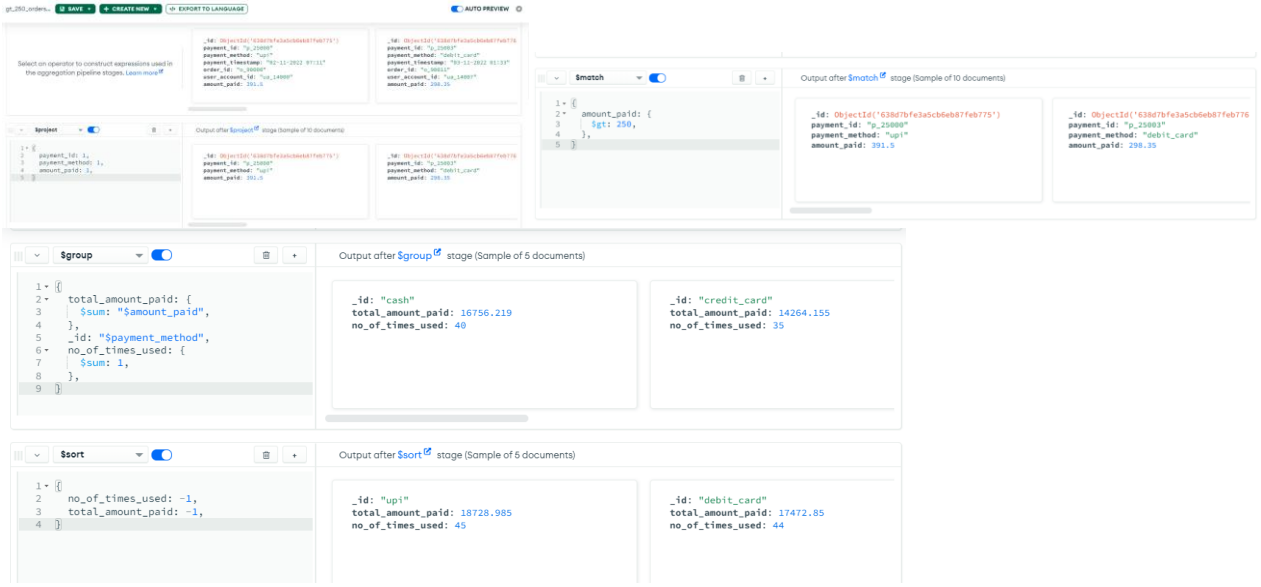


From the above query, we get the most frequently ordered food item category, thus helping the restaurants about inventory stocking based on popularity of the food items, for example preparing for veg dishes early by making certain sauces and chopping certain vegetables, would ensure to quick TAT.



Here we are getting the dish having the highest value of calories.

Next, we create a simple aggregate pipeline. It calculates the count of the payment methods used, and the aggregate value of the amount paid, for order values of over 250.



The final two stages and the group and the sort where we group the key value pairs by the payment method to calculate the total amount and number of times each payment method is used. The sort stage sorts the output key value pair in descending order by the no of times a particular payment method was used and then the total amount paid using that particular payment method.

V. Database Access via Python

We connect the MySQL database to Python and try to derive insightful analytics visualizations. Only for the first visualization, we have shown the connection procedure and the parsing of the retrieved information (this would remain the same for the other visualizations as well).

DMA_Group_28_Python_HTML.html

1. First, we visualize the order value information among the different group of customers (gold and non-gold)

```
cnx = sql.connect(
    host = "localhost",
    database = "foodpro",
    port = 3306,
    user = "root",
    password = "rootuser")

try:
    sql_select_Query = "SELECT * FROM discount;"
    cursor = cnx.cursor()
    cursor.execute(sql_select_Query)
    records = cursor.fetchall()
    print("Different Gold Subscriptions Plans:\n")
    for i, row in enumerate(records):
        print(f'Plan {i + 1} =',row,"\n")
#
except Error as e:
    print("Error while connecting to MySQL", e)

finally:
    if (cnx.is_connected()):
        cursor.close()
        cnx.close()
        print("MySQL connection is closed")
```

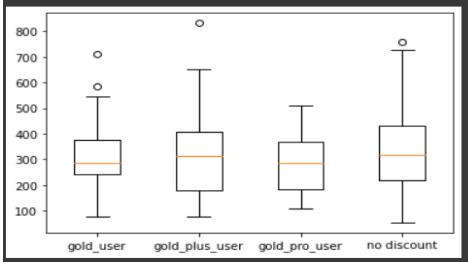
```
Different Gold Subscriptions Plans:

Plan 1 = ('d_gold', 'gold_user', 6, 15)

Plan 2 = ('d_gold_plus', 'gold_plus_user', 10, 35)

Plan 3 = ('d_gold_pro', 'gold_pro_user', 14, 100)

Plan 4 = ('not applicable', 'no discount', 0, 0)

MySQL connection is closed
```

```
cnx = sql.connect(
    host = "localhost",
    database = "foodpro",
    port = 3306,
    user = "root",
    password = "rootuser")

try:
    sql_select_Query = '''  SELECT d.discount_name, p.amount_paid
                            FROM discount d, orders o, payment p
                            WHERE o.discount_id = d.discount_id AND p.order_id = o.order_id;  '''
    cursor = cnx.cursor()
    cursor.execute(sql_select_Query)
    records = cursor.fetchall()

    print("The discounts availed on individual orders and the respective amount paid by the customer who made the order\n")
    for i, row in enumerate(records):
        print(f'Record {i + 1} =',row,"\n")
#
except Error as e:
    print("Error while connecting to MySQL", e)

finally:
    if (cnx.is_connected()):
        cursor.close()
        cnx.close()
        print("MySQL connection is closed")
```

```
The discounts availed on individual orders and the respective amount paid by the customer who made the order

Record 1 = ('gold_user', 586.56)

Record 2 = ('gold_user', 201.63)

Record 3 = ('gold_user', 711.345)
```

```
rec = {}
for i, j in records:
    if rec.get(i):
        rec[i].append(j)
    else:
        rec[i] = [j]
```

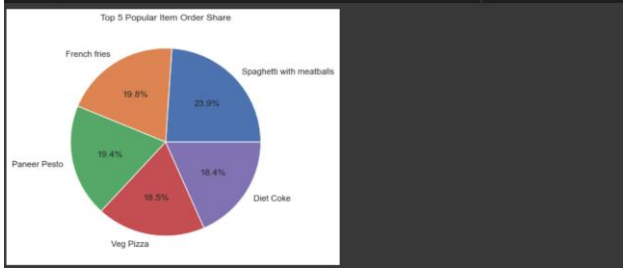
```
fig, ax = plt.subplots()
ax.boxplot(rec.values())
ax.set_xticklabels(rec.keys());
```



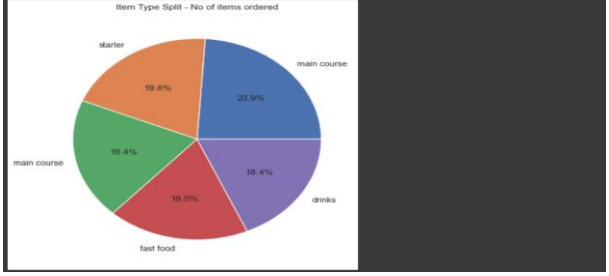
The gold subscription plan users show lesser variations in their order ranges.

2. Now, lets look at the different dishes offered and the number of times they were ordered

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.pie(df_temp.nlargest(5, 'no_of_units_ordered')['no_of_units_ordered'],
       labels = df_temp.nlargest(5, 'no_of_units_ordered')['item_name'], autopct = '%.1f%%')
ax.set_title('Top 5 Popular Item Order Share')
plt.tight_layout();
```



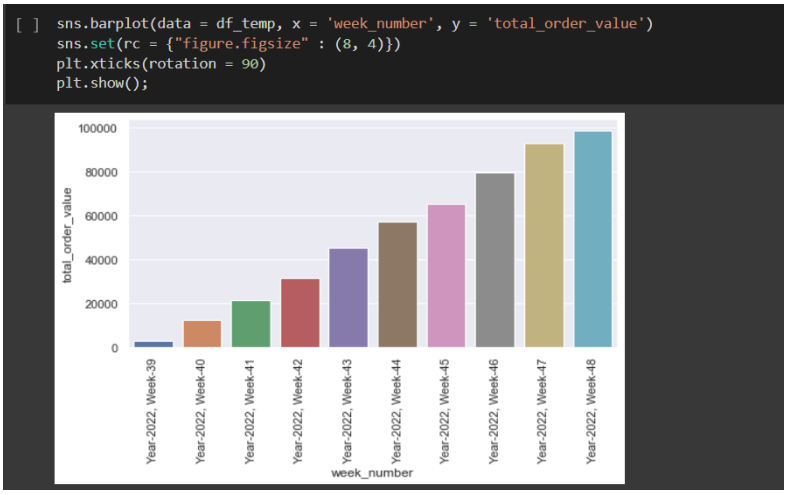

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.pie(df_temp.nlargest(5, 'no_of_units_ordered')['no_of_units_ordered'],
       labels = df_temp.nlargest(5, 'no_of_units_ordered')['item_type'], autopct = '%.1f%%')
ax.set_title('Item Type Split - No of items ordered')
plt.tight_layout();
```



Spaghetti with meatballs, French fries, Paneer Pesto, Veg Pizza, Diet Coke are the top 5 items that ordered by the users; the restaurants can be guided to increase the inventory needed to fulfill the above orders. Also, interesting to note that 3 out of the top 5 most ordered items are fast food.

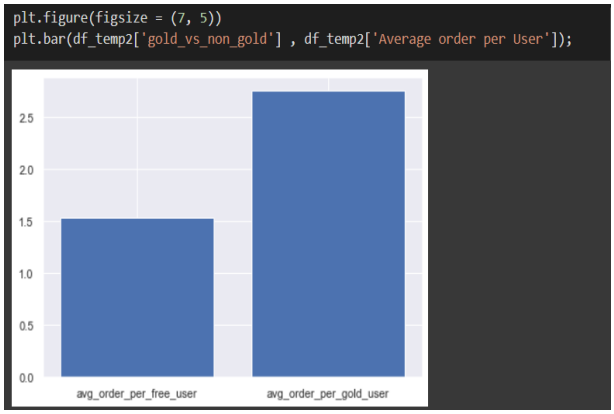
Therefore, we can guide the restaurant to revamp the menu items to include more fast food, as they are the most selling and they ideally have higher margins and shorter cooking time.

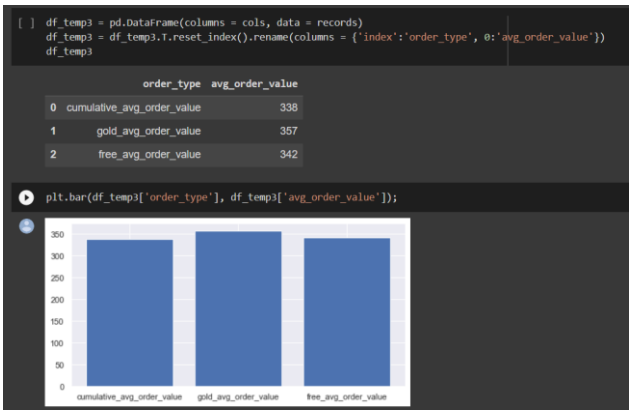
3. Let's visualize the mini balance sheet in a data frame - Note that the values given are cumulative in nature and not the individual week values:



The sales week over week have been consistently increasing which is a good sign. Considering the higher discounts (in terms of volume) offered to customers, it would be interesting to see if the graph pattern remains the same when looking into the final profit. Also, the order value is the highest for the combined consecutive week 46/47, this could be attributed to the thanksgiving week festivities. With people travelling from different places, and staying home, more orders can be expected around the holidays. We can further think about providing some extra discount during the holiday season to boost sales of both the orders as well as the subscriptions plans.

4. Now we will try to visualize the average number of orders between gold and non-gold customers and, average order value of gold and non-gold customers:





The gold users order more often than the regular free user, this could be attributed to the fact that the gold user enjoys the discount associated with the gold account benefits.
We can also, see that the average order value of gold members is larger in comparison to the free account users. It would be interesting to see if the story remains even after we consider the gold discount offered to the gold members

5. Now, well try to visualize the change in revenue/profit by looking at the income growth/decl ine in a waterfall graph

The order value (with and without discount) and the discounts given show constant increase with the rate reducing in the latest weeks. The subscription fee collected has also reduced in comparison to the earlier days of the app. So, we can focus on revamping the subscription plan structure; think of new ways of incentivizing buying a subscription plan, because it has clearly been established that gold users use the app to buy larger orders than non-gold users.

VI. Summary and Recommendation

We were able to successfully implement a database system which was robust for both kinds of transactions – transactional and referential in nature. The MySQL database and the MongoDB tie up together nicely as the former provides a neatly structured/normalised database to store the structured data in, while the latter provides us with the flexibility to store unstructured data. Future scalability can be done both vertically and horizontally on MongoDB. The queries we wrote could be used to draw important information regarding the company's performance/direction thus far and can be utilized to appropriately adjust the strategy to further improve service on both the restaurants/delivery agents and the customers alike. The structured data inside the MySQL database makes it easy to produce repeatable results/information – like a mini balance sheet, as the company grows, and see the order patterns in restaurants by different customer groups. The Python connections enables advanced analytical capabilities and provides us with intuitive visualizations.

The advantage of our current design would be the significant rework required to accommodate the upgrade in cardinality (from 1-1/1-M to M-N); this would include the addition of new relations/referential keys in the MySQL database. Also, because the application has been launched there would be constant changes in the company's strategy about the gold plans itself (above the changes in discount %), eventually gold members could be offered faster deliveries and could offer from restaurants from additional radius compared to a free customer – such futuristic changes are not yet captured in the database and would have to be reworked when the changes are introduced. Another improvement we could think of implementing is an automated weekly balance sheet/performance sheet generation code; where on a periodic basis, the report would be generated and shared across the respective stakeholders/decision makers. Also, we could think of utilising the additional visualization capabilities provided by powerful BI tools like Power BI/Tableau to be able to capture more nuanced interactions of our data that would be missed otherwise.