**Case Study #1 - Danny's Diner**

**Introduction**

In early 2021, Danny follows his passion for Japanese food and opens "Danny's Diner," a charming restaurant offering sushi, curry, and ramen. However, lacking data analysis expertise, the restaurant struggles to leverage the basic data collected during its initial months to make informed business decisions. Danny's Diner seeks assistance in using this data effectively to keep the restaurant thriving.

**Problem Statement**

Danny aims to utilize customer data to gain valuable insights into their visiting patterns, spending habits, and favorite menu items. By establishing a deeper connection with his customers, he can provide a more personalized experience for his loyal patrons.

He plans to use these insights to make informed decisions about expanding the existing customer loyalty program. Additionally, Danny seeks assistance in generating basic datasets for his team to inspect the data conveniently, without requiring SQL expertise.

Due to privacy concerns, he has shared a sample of his overall customer data, hoping it will be sufficient for you to create fully functional SQL queries to address his questions.

The case study revolves around three key datasets:

* Sales
* Menu
* Members

**Entity Relationship Diagram**

A screenshot of a computer

Description automatically generated

**Case Study Questions & Solutions**

1. **What is the total amount each customer spent at the restaurant?**

SELECT S.customer\_id, SUM(M.price) AS total\_amnt

FROM sales S

JOIN menu M ON S.product\_id = M.product\_id

GROUP BY S.customer\_id

ORDER BY customer\_id

* The SQL query retrieves the customer\_id and calculates the total amount spent (total\_amnt) by each customer at the restaurant.
* It combines data from the sales and menu tables based on matching product\_id.
* The results are grouped by customer\_id.
* The query then calculates the total sum of price for each group of sales records with the same customer\_id.
* Finally, the results are sorted in ascending order based on the customer\_id.

**Output-**

A screenshot of a computer

Description automatically generated

1. **How many days has each customer visited the restaurant?**

SELECT customer\_id, COUNT(DISTINCT order\_date) AS No\_Days

FROM sales

GROUP BY customer\_id

* + The SQL query selects the customer\_id and counts the number of distinct order dates (No\_Days) for each customer.
  + It retrieves data from the sales table.
  + The results are grouped by customer\_id.
  + The COUNT(DISTINCT order\_date) function calculates the number of unique order dates for each customer.
  + Finally, the query presents the total number of unique order dates as No\_Days for each customer.

**Output-**

A screenshot of a computer

Description automatically generated

1. **What was the first item from the menu purchased by each customer?**

WITH CTE AS

(SELECT S.customer\_id,DENSE\_RANK() OVER(PARTITION BY S.customer\_id ORDER BY S.order\_date)AS rn,M.product\_name

FROM sales S

JOIN menu M ON S.product\_id=M.product\_id)

SELECT customer\_id,product\_name

FROM CTE

WHERE rn=1

* The SQL query uses a Common Table Expression (CTE) named CTE to generate a temporary result set.
* Within the CTE, it selects the customer\_id, assigns a dense rank to each row based on the order date for each customer, and retrieves the corresponding product\_name from the menu table.
* The sales table is joined with the menu table on matching product\_id.
* The DENSE\_RANK() function assigns a rank to each row within the partition of each customer\_id based on the order\_date in ascending order.
* Each customer\_id has its own partition and separate ranks based on the order dates of their purchases.
* Next, the main query selects the customer\_id and corresponding product\_name from the CTE.
* It filters the results and only includes rows where the rank rn is equal to 1, which means the earliest purchase for each customer\_id.
* As a result, the query returns the first purchased product for each customer.

**Output-**

A screenshot of a computer

Description automatically generated

1. **What is the most purchased item on the menu and how many times was it purchased by all customers?**

SELECT M.product\_name,COUNT(S.product\_id)AS most\_ordered

FROM Sales S

JOIN menu M ON S.product\_id=M.product\_id

GROUP BY M.product\_name

ORDER BY most\_ordered DESC

LIMIT 1

* The SQL query selects the product\_name from the menu table and counts the number of times each product was ordered (most\_ordered).
* It retrieves data from the Sales table and joins it with the menu table based on matching product\_id.
* The results are grouped by product\_name.
* The COUNT(S.product\_id) function calculates the number of occurrences of each product\_id in the Sales table.
* The query then presents the product\_name and its corresponding count as most\_ordered for each product.
* Next, the results are sorted in descending order based on the most\_ordered column, so the most ordered product appears first.
* The LIMIT 1 clause is used to restrict the result to only one row, effectively showing the most ordered product.

**Output-**

A screenshot of a computer

Description automatically generated

1. **Which item was the most popular for each customer?**

WITH CTE AS

(SELECT S.customer\_id,M.product\_name,COUNT(M.product\_id)AS order\_count,DENSE\_RANK() OVER(PARTITION BY S.customer\_id ORDER BY COUNT(S.product\_id)DESC) AS rnk

FROM sales S

JOIN menu M ON S.product\_id=M.product\_id

GROUP BY S.customer\_id,M.product\_name)

SELECT customer\_id,product\_name,order\_count

FROM CTE

WHERE rnk=1

* The SQL query uses a Common Table Expression (CTE) named CTE to generate a temporary result set.
* Within the CTE, it selects the customer\_id, product\_name, and counts the number of times each product was ordered (order\_count) for each customer.
* It retrieves data from the sales table and joins it with the menu table based on matching product\_id.
* The results are grouped by customer\_id and product\_name to get the count of orders for each product of each customer.
* The COUNT(M.product\_id) function calculates the number of occurrences of each product\_id in the menu table.
* The DENSE\_RANK() function assigns a rank to each row within the partition of each customer\_id based on the order count of products in descending order.
* Each customer\_id has its own partition and separate ranks based on the number of product orders.
* Next, the main query selects the customer\_id, product\_name, and order\_count from the CTE.
* It filters the results and only includes rows where the rank rnk is equal to 1, which means the most ordered product for each customer\_id.
* As a result, the query returns the customer's ID, the most ordered product, and the number of times it was ordered by that customer.

**Output-**

A screenshot of a computer

Description automatically generated

1. **Which item was purchased first by the customer after they became a member?**

SELECT DISTINCT ON (s.customer\_id)

s.customer\_id,

m.product\_name

FROM sales s

JOIN members mbr ON s.customer\_id = mbr.customer\_id

JOIN menu m ON s.product\_id = m.product\_id

WHERE s.order\_date > mbr.join\_date

ORDER BY s.customer\_id;

* The SQL query retrieves distinct rows for each unique customer\_id with their corresponding product\_name from the sales and menu tables.
* It filters the data based on the condition that the order\_date in the sales table is greater than the join\_date of the customer in the members table.
* The sales table is aliased as s, the members table is aliased as mbr, and the menu table is aliased as m.
* The query performs inner joins between sales and members tables on matching customer\_id and between sales and menu tables on matching product\_id.
* Only rows that meet the join condition and the order\_date > join\_date condition are considered in the result set.
* The query selects the customer\_id and corresponding product\_name for each customer who has placed an order after their join\_date.
* The results are sorted in ascending order based on the customer\_id.
* The DISTINCT ON (s.customer\_id) clause ensures that only the first occurrence of each customer\_id is included in the result set.
* As a result, the query returns a unique list of customer\_id along with the first product\_name they ordered after joining as a member.

**Output-**

A screenshot of a computer

Description automatically generated

1. **Which item was purchased just before the customer became a member?**

SELECT DISTINCT ON (s.customer\_id)

s.customer\_id,

m.product\_name

FROM sales s

JOIN members mbr ON s.customer\_id = mbr.customer\_id

JOIN menu m ON s.product\_id = m.product\_id

WHERE s.order\_date < mbr.join\_date

ORDER BY s.customer\_id;

* The SQL query retrieves distinct rows for each unique customer\_id with their corresponding product\_name from the sales and menu tables.
* It filters the data based on the condition that the order\_date in the sales table is less than the join\_date of the customer in the members table.
* The sales table is aliased as s, the members table is aliased as mbr, and the menu table is aliased as m.
* The query performs inner joins between sales and members tables on matching customer\_id and between sales and menu tables on matching product\_id.
* Only rows that meet the join condition and the order\_date < join\_date condition are considered in the result set.
* The query selects the customer\_id and corresponding product\_name for each customer who has placed an order before their join\_date.
* The results are sorted in ascending order based on the customer\_id.
* The DISTINCT ON (s.customer\_id) clause ensures that only the first occurrence of each customer\_id is included in the result set.
* As a result, the query returns a unique list of customer\_id along with the first product\_name they ordered before joining as a member.

**Output-**

A screenshot of a computer

Description automatically generated

1. **What is the total items and amount spent for each member before they became a member?**

SELECT S.customer\_id,

COUNT(S.product\_id) AS total\_item,

SUM(M.price) AS total\_amont

FROM sales S

JOIN menu M ON S.product\_id=M.product\_id

JOIN members ME ON S.customer\_id=ME.customer\_id

WHERE S.order\_date<ME.join\_date

GROUP BY S.customer\_id

ORDER BY S.customer\_id

* The SQL query retrieves the customer\_id along with the total count of items ordered (total\_item) and the total amount spent (total\_amont) by each customer.
* It retrieves data from the sales table and joins it with the menu table based on matching product\_id.
* It also joins the sales table with the members table based on matching customer\_id.
* The results are filtered based on the condition that the order\_date in the sales table is less than the join\_date of the customer in the members table.
* The COUNT(S.product\_id) function calculates the number of occurrences of each product\_id in the sales table, giving the total number of items ordered by each customer.
* The SUM(M.price) function calculates the sum of the price from the menu table, providing the total amount spent by each customer.
* Results are grouped by customer\_id to get the totals for each customer.
* The query then presents the customer\_id, total\_item, and total\_amont for each customer who placed orders before joining as a member.
* Finally, the results are sorted in ascending order based on the customer\_id.

**Output-**

A screenshot of a computer

Description automatically generated

1. **If each $1 spent equates to 10 points and sushi has a 2x points multiplier - how many points would each customer have?**

SELECT s.customer\_id,

SUM(CASE

WHEN m.product\_name = 'sushi' THEN price \* 2

ELSE price

END) \* 10 AS total\_points

FROM sales s

JOIN menu m ON s.product\_id = m.product\_id

GROUP BY s.customer\_id

ORDER BY s.customer\_id;

* The SQL query retrieves the customer\_id and calculates the total points (total\_points) earned by each customer based on their purchases from the sales and menu tables.
* It retrieves data from the sales table and joins it with the menu table based on matching product\_id.
* The query uses a CASE statement to differentiate between 'sushi' and other products.
* If the product name is 'sushi', the price is multiplied by 2 to give double points.
* Otherwise, the regular price is considered.
* The SUM function calculates the total points for each customer by adding up the points earned from their purchases.
* The total points are then multiplied by 10 to give a scaled value.
* Results are grouped by customer\_id to get the total points for each customer.
* The query then presents the customer\_id and the scaled total\_points for each customer based on their purchases.
* Finally, the results are sorted in ascending order based on the customer\_id.

**Output-**

A screenshot of a computer

Description automatically generated

1. **In the first week after a customer joins the program (including their join date) they earn 2x points on all items, not just sushi - how many points do customer A and B have at the end of January?**

WITH dates\_cte AS (

SELECT

customer\_id,

join\_date,

join\_date + INTERVAL '6 days' AS valid\_date,

DATE\_TRUNC('month', '2021-01-31'::DATE) + INTERVAL '1 month' - INTERVAL '1 day' AS last\_date

FROM members

)

SELECT

s.customer\_id,

SUM(CASE

WHEN m.product\_name = 'sushi' OR (s.order\_date BETWEEN dates.join\_date AND dates.valid\_date) THEN 2 \* 10 \* m.price

ELSE 10 \* m.price END) AS points

FROM sales s

INNER JOIN dates\_cte AS dates

ON s.customer\_id = dates.customer\_id

AND dates.join\_date <= s.order\_date

AND s.order\_date <= dates.last\_date

INNER JOIN menu m

ON s.product\_id = m.product\_id

GROUP BY s.customer\_id

ORDER BY s.customer\_id;

* The SQL query starts by creating a Common Table Expression (CTE) named dates\_cte.
* Within the CTE, it selects customer\_id, join\_date, join\_date + INTERVAL '6 days' as valid\_date, and the last day of the month for the date '2021-01-31' as last\_date.
* The CTE is used to generate date ranges for each customer, from their join\_date to 6 days later, and the last day of the month for January 2021.
* Next, the main query selects the customer\_id and calculates the total points (points) earned by each customer based on their purchases from the sales and menu tables.
* It retrieves data from the sales table and joins it with the dates\_cte CTE using a combination of JOIN and ON clauses.
* The query uses a CASE statement to differentiate between 'sushi' purchases and other products.
* If the product name is 'sushi' or the order date falls within the range of join\_date to valid\_date, the points are calculated as 2 times 10 times the price of the product.
* Otherwise, for other products, the points are calculated as 10 times the price of the product.
* The SUM function calculates the total points for each customer by adding up the points earned from their purchases.
* Results are grouped by customer\_id to get the total points for each customer.
* The query then presents the customer\_id and the calculated points for each customer based on their purchases.
* Finally, the results are sorted in ascending order based on the customer\_id.

**Output-**

A screenshot of a computer

Description automatically generated

**Key Insights**

**Customer Spending**: There's a big difference in how much each customer spends at Danny's Diner. Some customers spend a lot more than others, which shows we might have some really loyal customers.

**Customer Visits**: How often customers come to the diner varies too. Some come a lot, while others only a few times.

**First Purchases**: Knowing what items new customers buy first helps us figure out which dishes attract people to Danny’s Diner.

**Most Popular Item**: Finding out which item is ordered the most helps Danny manage his stock better and focus on what customers like most.

**Personalized Recommendations**: If we know which items are favorites for each customer, Danny can suggest dishes they might like, making their visits even better.

**Customer Loyalty**: Looking at what customers buy before and after they join our loyalty program shows us how well the program is working.

**Bonus Points for New Members**: Giving double points in the first week encourages new loyalty members to spend more.

**Member Points**: Tracking the points each member earns helps us see who our most loyal customers are and lets us offer them special deals.

**Data Visualization**: Charts and graphs based on our data help Danny spot trends and make decisions based on facts.

**Customer Segmentation**: By understanding how different customers spend, Danny can create marketing that's more likely to appeal to different groups.

**Expanding Membership**: Insights from the data can help improve the loyalty program and draw in new members.

**Inventory Management**: Knowing which items are least and most popular can help Danny reduce waste and increase profits.

**Menu Optimization**: This data can also help Danny decide which menu items to keep or change and think about adding new dishes that customers might like.

**Customer Engagement**: Understanding why customers keep coming back helps Danny make the diner more appealing.

**Long-Term Growth**: Using all this data, Danny can make informed choices that help the diner grow and succeed over time.