**Revenue per customer**

You've been hired at Delivr as a data analyst! A customer just called Delivr's Customer Support team; she wants to double-check whether her receipts add up. Going by her receipts, she calculated that her total bill on Delivr is $271, and she wants to make sure of that. Her user ID is 15.

Help the Customer Support team by calculating her total bill! Sum up everything she's spent on Delivr orders; in other words, calculate the total revenue that Delivr has generated from her.

* Write the expression for revenue.
* Keep only the records of user ID 15.
* -- Calculate revenue
* SELECT sum(meal\_price\*order\_quantity) AS revenue
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* -- Keep only the records of customer ID 15
* WHERE user\_id = 15;

# Revenue per week

Delivr's first full month of operations was June 2018. At launch, the Marketing team ran an ad campaign on popular food channels on TV, with the number of ads increasing each week through the end of the month. The Head of Marketing asks you to help her assess that campaign's success.

Get the revenue per week for each week in June and check whether there's any consistent growth in revenue.

**Note**: Don't be surprised if you get a date in May in the result. DATE\_TRUNC('week', '2018-06-02') returns '2018-05-28', since '2018-06-02' is a Saturday and the preceding Monday is on '2018-05-28'.

* Write the expression for revenue.
* Keep only the records of June 2018.
* SELECT DATE\_TRUNC('week', order\_date) :: DATE AS delivr\_week,
* -- Calculate revenue
* sum(meal\_price\*order\_quantity) AS revenue
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* -- Keep only the records in June 2018
* WHERE DATE\_TRUNC('MONTH',order\_date):: DATE ='2018-06-01'
* GROUP BY delivr\_week
* ORDER BY delivr\_week ASC;

# Total cost

What is Delivr's total cost since it began operating?

SELECT sum(meal\_cost\*order\_quantity) AS TOTALCOST

FROM meals

JOIN orders ON meals.meal\_id = orders.meal\_id

# Top meals by cost

Alice from Finance wants to know what Delivr's top 5 meals are by overall cost; in other words, Alice wants to know the 5 meals that Delivr has spent the most on for stocking.

You're provided with an aggregate query; you'll need to fill in the blanks to get the output Alice needs.

**Note**: Recall that in the meals table, meal\_price is what the user pays Delivr for the meal, while meal\_cost is what Delivr pays its eateries to stock this meal.

* Calculate cost per meal ID.
* Set the LIMIT to 5.
* SELECT
* -- Calculate cost per meal ID
* MEALS.meal\_id,
* SUM(meal\_cost\*stocked\_quantity) AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY meals.meal\_id
* ORDER BY cost DESC
* -- Only the top 5 meal IDs by purchase cost
* LIMIT 5;

**Using CTEs**

Alice wants to know how much Delivr spent per month on average during its early months (before September 2018). You'll need to write two queries to solve this problem:

* A query to calculate cost per month, wrapped in a CTE,
* A query that averages monthly cost before September 2018 by referencing the CTE.
* Calculate cost per month.
* SELECT
* -- Calculate cost
* DATE\_TRUNC('month', stocking\_date)::DATE AS delivr\_month,
* SUM(meal\_cost\*stocked\_quantity)  AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY delivr\_month
* ORDER BY delivr\_month ASC;

**Using CTEs**

Alice wants to know how much Delivr spent per month on average during its early months (before September 2018). You'll need to write two queries to solve this problem:

* A query to calculate cost per month, wrapped in a CTE,
* A query that averages monthly cost before September 2018 by referencing the CTE.
* Wrap the query you just wrote in a CTE named monthly\_cost.
* -- Declare a CTE named monthly\_cost
* WITH monthly\_cost AS (
* SELECT
* DATE\_TRUNC('month', stocking\_date)::DATE AS delivr\_month,
* SUM(meal\_cost \* stocked\_quantity) AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY delivr\_month)
* SELECT \*
* FROM monthly\_cost;
* Now that you've set up the monthly\_cost CTE, find the average cost incurred **before** September 2018.
* -- Declare a CTE named monthly\_cost
* WITH monthly\_cost AS (
* SELECT
* DATE\_TRUNC('month', stocking\_date)::DATE AS delivr\_month,
* SUM(meal\_cost \* stocked\_quantity) AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY delivr\_month)
* SELECT
* -- Calculate the average monthly cost before September
* AVG(COST)
* FROM monthly\_cost
* WHERE delivr\_month < '2018-09-01';

# Profit per eatery

Delivr is renegotiating its contracts with its eateries. The higher the profit that an eatery generates, the higher the rate that Delivr is willing to pay this eatery for the bulk purchase of meals.

The Business Development team asks you to find out how much profit each eatery is generating to strengthen their negotiating positions.

**Note**: You don't need to GROUP BY eatery in the final query. You've already grouped by eatery in the revenue and cost CTEs; all that's required is joining them to each other to get each eatery's revenue and cost in one row. Since revenue and cost take up one row each per eatery, there are no additional groupings to be made.

* Calculate revenue per eatery in the revenue CTE.
* Calculate cost per eatery in the cost CTE.
* Join the two CTEs and calculate profit per eatery.
* WITH revenue AS (
* -- Calculate revenue per eatery
* SELECT eatery,
* SUM(meal\_cost \* order\_quantity)  AS revenue
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* GROUP BY eatery),
* cost AS (
* -- Calculate cost per eatery
* SELECT eatery,
* SUM(meal\_cost \* stocked\_quantity)  AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY eatery)
* -- Calculate profit per eatery
* SELECT revenue.eatery,
* revenue - cost as profit
* FROM revenue
* JOIN cost ON revenue.eatery = cost.eatery
* ORDER BY profit DESC;

# Profit per month

After prioritizing and making deals with eateries by their overall profits, Alice wants to track Delivr profits per month to see how well it's doing. You're here to help.

You're provided with two CTEs. The first stores revenue and the second stores cost. To access revenue and cost in one query, the two CTEs are joined in the last query. From there, you can apply the formula for profit Profit = Revenue - Cost to calculate profit per month.

Remember that revenue is the sum of each meal's price times its order quantity, and that cost is the sum of each meal's cost times its stocked quantity.

* Calculate revenue per month in the revenue CTE.
* Calculate cost per month in the cost CTE.
* Join the two CTEs and calculate profit per month.
* -- Set up the revenue CTE
* WITH revenue AS (
* SELECT
* DATE\_TRUNC('month', order\_date) :: DATE AS delivr\_month,
* SUM(meal\_price \* order\_quantity)   AS revenue
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* GROUP BY delivr\_month),
* -- Set up the cost CTE
* cost AS (
* SELECT
* DATE\_TRUNC('month', stocking\_date) :: DATE AS delivr\_month,
* SUM(meal\_cost \* stocked\_quantity)   AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* GROUP BY delivr\_month)
* -- Calculate profit by joining the CTEs
* SELECT
* revenue.delivr\_month,
* revenue - cost as profit
* FROM revenue
* JOIN cost ON revenue.delivr\_month = cost.delivr\_month
* ORDER BY revenue.delivr\_month ASC;

# Registrations by month

Usually, registration dates are stored in a table containing users' metadata. However, Delivr only considers a user registered if that user has ordered at least once. A Delivr user's registration date is the date of that user's first order.

Bob, the Investment Relations Manager at Delivr, is preparing a pitch deck for a meeting with potential investors. He wants to add a line chart of registrations by month to highlight Delivr's success in gaining new users.

Send Bob a table of registrations by month.

* Return a table of user IDs and their registration dates.
* Order by user\_id in ascending order.
* SELECT
* -- Get the earliest (minimum) order date by user ID
* user\_id,
* min(order\_date) AS reg\_date
* FROM orders
* GROUP BY user\_id
* -- Order by user ID
* ORDER BY user\_id ASC;

# Registrations by month

Usually, registration dates are stored in a table containing users' metadata. However, Delivr only considers a user registered if that user has ordered at least once. A Delivr user's registration date is the date of that user's first order.

Bob, the Investment Relations Manager at Delivr, is preparing a pitch deck for a meeting with potential investors. He wants to add a line chart of registrations by month to highlight Delivr's success in gaining new users.

Send Bob a table of registrations by month.

* Wrap the query you just wrote in a CTE named reg\_dates.
* Using reg\_dates, return a table of registrations by month.
* -- Wrap the query you wrote in a CTE named reg\_dates
* WITH reg\_dates AS (
* SELECT
* user\_id,
* MIN(order\_date) AS reg\_date
* FROM orders
* GROUP BY user\_id)
* SELECT
* -- Count the unique user IDs by registration month
* DATE\_TRUNC ('month', reg\_date) :: DATE AS delivr\_month,
* count(distinct user\_id ) AS regs
* FROM reg\_dates
* GROUP BY delivr\_month
* ORDER BY delivr\_month ASC;

# Monthly active users (MAU)

Bob predicts that the investors won't be satisfied with only registrations by month. They will want to know how many users actually used Delivr as well. He's decided to include another line chart of Delivr's monthly active users (MAU); he's asked you to send him a table of monthly active users.

* Select the month by truncating the order dates.
* Calculate MAU by counting the users for every month.
* Order by month in ascending order.
* SELECT
* -- Truncate the order date to the nearest month
* DATE\_TRUNC('month',order\_date) :: DATE AS delivr\_month,
* -- Count the unique user IDs
* count(distinct user\_id) AS mau
* FROM orders
* GROUP BY delivr\_month
* -- Order by month
* ORDER BY delivr\_month ASC;

# Registrations running total

You have a suggestion for Bob's pitch deck: Instead of showing registrations by month in the line chart, he can show the registrations running total by month. The numbers are bigger that way, and investors always love bigger numbers! He agrees, and you begin to work on a query that returns a table of the registrations running total by month.

* Select the month and the registrations in each month.
* Order by month in ascending order.
* WITH reg\_dates AS (
* SELECT
* user\_id,
* MIN(order\_date) AS reg\_date
* FROM orders
* GROUP BY user\_id)
* SELECT
* -- Select the month and the registrations
* DATE\_TRUNC('month', reg\_date) :: DATE AS delivr\_month,
* COUNT(DISTINCT user\_id) AS regs
* FROM reg\_dates
* GROUP BY delivr\_month
* -- Order by month in ascending order
* ORDER BY delivr\_month;

# Registrations running total

You have a suggestion for Bob's pitch deck: Instead of showing registrations by month in the line chart, he can show the registrations running total by month. The numbers are bigger that way, and investors always love bigger numbers! He agrees, and you begin to work on a query that returns a table of the registrations running total by month.

* Return a table of the registrations running total by month.
* Order by month in ascending order.
* WITH reg\_dates AS (
* SELECT
* user\_id,
* MIN(order\_date) AS reg\_date
* FROM orders
* GROUP BY user\_id),
* regs AS (
* SELECT
* DATE\_TRUNC('month', reg\_date) :: DATE AS delivr\_month,
* COUNT(DISTINCT user\_id) AS regs
* FROM reg\_dates
* GROUP BY delivr\_month)
* SELECT
* -- Calculate the registrations running total by month
* delivr\_month,
* sum(regs) over (order by delivr\_month asc) AS regs\_rt
* FROM regs
* -- Order by month in ascending order
* ORDER BY delivr\_month ASC;

# MAU monitor (I)

Carol from the Product team noticed that you're working with a lot of user-centric KPIs for Bob's pitch deck. While you're at it, she says, you can help build an idea of hers involving a user-centric KPI. She wants to build a monitor that compares the MAUs of the previous and current month, raising a red flag to the Product team if the current month's active users are less than those of the previous month.

To start, write a query that returns a table of MAUs and the previous month's MAU for every month.

* Select the month and the MAU.
* Fetch the previous month's MAU.
* Order by month in ascending order.
* WITH mau AS (
* SELECT
* DATE\_TRUNC('month', order\_date) :: DATE AS delivr\_month,
* COUNT(DISTINCT user\_id) AS mau
* FROM orders
* GROUP BY delivr\_month)
* SELECT
* -- Select the month and the MAU
* delivr\_month,
* mau,
* COALESCE(
* lag(mau) over (order by delivr\_month ASC),
* 0) AS last\_mau
* FROM mau
* -- Order by month in ascending order
* ORDER BY delivr\_month ASC;

# MAU monitor (II)

Now that you've built the basis for Carol's MAU monitor, write a query that returns a table of months and the deltas of each month's current and previous MAUs.

If the delta is negative, less users were active in the current month than in the previous month, which triggers the monitor to raise a red flag so the Product team can investigate.

* Fetch the previous month's MAU in the mau\_with\_lag CTE..
* Select the month and the delta between its MAU and the previous month's MAU.
* Order by month in ascending order.
* WITH mau AS (
* SELECT
* DATE\_TRUNC('month', order\_date) :: DATE AS delivr\_month,
* COUNT(DISTINCT user\_id) AS mau
* FROM orders
* GROUP BY delivr\_month),
* mau\_with\_lag AS (
* SELECT
* delivr\_month,
* mau,
* -- Fetch the previous month's MAU
* COALESCE(
* lag(mau) over (order by delivr\_month ASC),
* 0) AS last\_mau
* FROM mau)
* SELECT
* -- Calculate each month's delta of MAUs
* delivr\_month,
* mau-last\_mau AS mau\_delta
* FROM mau\_with\_lag
* -- Order by month in ascending order
* ORDER BY delivr\_month;

# MAU monitor (III)

Carol is very pleased with your last query, but she's requested one change: She prefers to have the month-on-month (MoM) MAU growth rate over a raw delta of MAUs. That way, the MAU monitor can have more complex triggers, like raising a yellow flag if the growth rate is -2% and a red flag if the growth rate is -5%.

Write a query that returns a table of months and each month's MoM MAU growth rate to finalize the MAU monitor.

* Select the month and its MoM MAU growth rate.
* Order by month in ascending order.
* WITH mau AS (
* SELECT
* DATE\_TRUNC('month', order\_date) :: DATE AS delivr\_month,
* COUNT(DISTINCT user\_id) AS mau
* FROM orders
* GROUP BY delivr\_month),
* mau\_with\_lag AS (
* SELECT
* delivr\_month,
* mau,
* GREATEST(
* LAG(mau) OVER (ORDER BY delivr\_month ASC),
* 1) AS last\_mau
* FROM mau)
* SELECT
* -- Calculate the MoM MAU growth rates
* delivr\_month,
* ROUND(
* (mau - last\_mau) :: NUMERIC/last\_mau,
* 2) AS growth
* FROM mau\_with\_lag
* -- Order by month in ascending order
* ORDER BY delivr\_month;

# Order growth rate

Bob needs one more chart to wrap up his pitch deck. He's covered Delivr's gain of new users, its growing MAUs, and its high retention rates. Something is missing, though. Throughout the pitch deck, there's not a single mention of the best indicator of user activity: the users' orders! The more orders users make, the more active they are on Delivr, and the more money Delivr generates.

Send Bob a table of MoM order growth rates.

(**Recap**: MoM means month-on-month.)

* Count the unique orders per month.
* Fetch each month's previous and current orders.
* Return a table of MoM order growth rates.
* WITH orders AS (
* SELECT
* DATE\_TRUNC ('month', order\_date) :: DATE AS delivr\_month,
* -- Count the unique order IDS
* COUNT (DISTINCT order\_id) AS orders
* FROM orders
* GROUP BY delivr\_month),
* orders\_with\_lag AS (
* SELECT
* delivr\_month,
* --Fetch each month's current and previous orders
* orders,
* COALESCE(
* LAG (orders) OVER (ORDER BY delivr\_month ASC),
* 1) AS last\_orders
* FROM orders)
* SELECT
* delivr\_month,
* -- Calculate the MoM order growth rate
* ROUND(
* (orders - last\_orders) :: NUMERIC / last\_orders,
* 2) AS growth
* FROM orders\_with\_lag
* ORDER BY delivr\_month ASC;

# Retention rate

Bob's requested your help again now that you're done with Carol's MAU monitor. His meeting with potential investors is fast approaching, and he wants to wrap up his pitch deck. You've already helped him with the registrations running total by month and MAU line charts; the investors, Bob says, would be convinced that Delivr is growing both in new users and in MAUs.

However, Bob wants to show that Delivr not only attracts new users but also retains existing users. Send him a table of MoM retention rates so that he can highlight Delivr's high user loyalty.

* Select the month column from user\_monthly\_activity, and calculate the MoM user retention rates.
* Join user\_monthly\_activity to itself on the user ID and the month, pushed forward one month.
* WITH user\_monthly\_activity AS (
* SELECT DISTINCT
* DATE\_TRUNC ('month', order\_date) :: DATE AS delivr\_month,
* user\_id
* FROM orders)
* SELECT
* -- Calculate the MoM retention rates
* previous.delivr\_month,
* ROUND (
* COUNT (DISTINCT current.user\_id) :: NUMERIC /
* GREATEST (COUNT (DISTINCT previous.user\_id), 1),
* 2) AS retention\_rate
* FROM user\_monthly\_activity AS previous
* LEFT JOIN user\_monthly\_activity AS current
* -- Fill in the user and month join conditions
* ON previous.user\_id = current.user\_id
* AND previous.delivr\_month= (current.delivr\_month - INTERVAL '1 month')
* GROUP BY previous.delivr\_month
* ORDER BY previous.delivr\_month ASC;

# Average revenue per user

Dave from Finance wants to study Delivr's performance in revenue and orders per each of its user base. In other words, he wants to understand its unit economics.

Help Dave kick off his study by calculating the overall average revenue per user (ARPU) using the first way discussed in Lesson 3.1.

Return a table of user IDs and the revenue each user generated

SELECT

  -- Select the user ID and calculate revenue

  user\_id,

  sum(meal\_price\*order\_quantity) AS revenue

FROM meals AS m

JOIN orders AS o ON m.meal\_id = o.meal\_id

GROUP BY user\_id;

# Average revenue per user

Dave from Finance wants to study Delivr's performance in revenue and orders per each of its user base. In other words, he wants to understand its unit economics.

Help Dave kick off his study by calculating the overall average revenue per user (ARPU) using the first way discussed in Lesson 3.1.

* Wrap the previous query in a CTE named kpi.
* Return the average revenue per user (ARPU).
* -- Create a CTE named kpi
* with KPI as (
* SELECT
* -- Select the user ID and calculate revenue
* user\_id,
* SUM(m.meal\_price \* o.order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m.meal\_id = o.meal\_id
* GROUP BY user\_id)
* -- Calculate ARPU
* SELECT ROUND(avg(revenue) :: numeric, 2) AS arpu
* FROM kpi;

# ARPU per week

Next, Dave wants to see whether ARPU has increased over time. Even if Delivr's revenue is increasing, it's not scaling well if its ARPU is decreasing—it's generating less revenue from each of its customers.

Send Dave a table of ARPU by week using the second way discussed in Lesson 3.1.

* Store revenue and the number of unique active users by week in the kpi CTE.
* Calculate ARPU by dividing the revenue by the number of users.
* Order the results by week in ascending order.
* WITH kpi AS (
* SELECT
* -- Select the week, revenue, and count of users
* Date\_Trunc('week', order\_date) :: DATE AS delivr\_week,
* SUM(meal\_price\*order\_quantity) AS revenue,
* count(distinct user\_id) AS users
* FROM meals AS m
* JOIN orders AS o ON m.meal\_id = o.meal\_id
* GROUP BY delivr\_week)
* SELECT
* delivr\_week,
* -- Calculate ARPU
* ROUND(
* revenue :: numeric / Greatest(users,1),
* 2) AS arpu
* FROM kpi
* -- Order by week in ascending order
* ORDER BY delivr\_week ASC;

# Average orders per user

Dave wants to add the average orders per user value to his unit economics study, since more orders usually correspond to more revenue.

Calculate the average orders per user for Dave.

**Note**: The count of distinct orders is different than the sum of ordered meals. One order can have many meals within it. Average orders per user depends on the count of orders, not the sum of ordered meals.

* Store the count of distinct orders and distinct users in the kpi CTE.
* Calculate the average orders per user.
* WITH kpi AS (
* SELECT
* -- Select the count of orders and users
* count(distinct order\_id) AS orders,
* count(distinct user\_id) AS users
* FROM orders)
* SELECT
* -- Calculate the average orders per user
* ROUND(
* orders :: numeric / GREATEST(users,1),
* 2) AS arpu
* FROM kpi;

# Histogram of revenue

After determining that Delivr is doing well at scaling its business model, Dave wants to explore the distribution of revenues. He wants to see whether the distribution is U-shaped or normal to see how best to categorize users by the revenue they generate.

Send Dave a frequency table of revenues by user.

* Store each user ID and the revenue Delivr generates from it in the user\_revenues CTE.
* Return a frequency table of revenues rounded to the nearest hundred and the users generating those revenues.
* WITH user\_revenues AS (
* SELECT
* -- Select the user ID and revenue
* user\_id,
* SUM (meal\_price\* order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m. meal\_id = o. meal\_id
* GROUP BY user\_id)
* SELECT
* -- Return the frequency table of revenues by user
* ROUND (revenue :: NUMERIC, -2) AS revenue\_100,
* COUNT (DISTINCT user\_id) AS users
* FROM user\_revenues
* GROUP BY revenue\_100
* ORDER BY revenue\_100 ASC;

# Histogram of orders

Dave also wants to plot the histogram of orders to see if it matches the shape of the histogram of revenues.

Send Dave a frequency table of orders by user.

* Set up the frequency tables query by getting each user's count of orders.
* SELECT
* -- Select the user ID and the count of orders
* user\_id,
* count(distinct order\_id) AS orders
* FROM orders
* GROUP BY user\_id
* ORDER BY user\_id ASC
* LIMIT 5;
* Return a frequency table of orders and the count of users with those orders.

WITH user\_orders AS (

  SELECT

    user\_id,

    COUNT(DISTINCT order\_id) AS orders

  FROM orders

  GROUP BY user\_id)

SELECT

  -- Return the frequency table of orders by user

  orders,

  count(distinct user\_id) AS users

FROM user\_orders

GROUP BY orders

ORDER BY orders ASC;

# Bucketing users by revenue

Based on his analysis, Dave identified that $150 is a good cut-off for low-revenue users, and $300 is a good cut-off for mid-revenue users. He wants to find the number of users in each category to tweak Delivr's business model.

Split the users into low, mid, and high-revenue buckets, and return the count of users in each group.

* Store each user ID and the revenue it generates in the user\_revenues CTE.
* Return a table of the revenue groups and the count of users in each group.
* WITH user\_revenues AS (
* SELECT
* -- Select the user IDs and the revenues they generate
* user\_id,
* SUM (meal\_price\* order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m. meal\_id = o. meal\_id GROUP BY user\_id)
* SELECT
* -- Fill in the bucketing conditions
* CASE
* WHEN revenue < 150
* THEN 'Low-revenue users'
* WHEN revenue < 300 THEN 'Mid-revenue users'
* ELSE 'High-revenue users'
* END AS revenue\_group,
* COUNT (DISTINCT user\_id) AS users
* FROM user\_revenues
* GROUP BY revenue\_group;

# Bucketing users by orders

Dave is repeating his bucketing analysis on orders to have a more complete profile of each group. He determined that 8 orders is a good cut-off for the low-orders group, and 15 is a good cut-off for the medium orders group.

Send Dave a table of each order group and how many users are in it.

* Store each user ID and its count of orders in a CTE named user\_orders.
* Set the cut-off point for the low-orders bucket to 8 orders, and set the cut-off point for the mid-orders bucket to 15 orders.
* Count the distinct users in each bucket.

WITH user\_orders AS (

SELECT

user\_id,

COUNT (DISTINCT order\_id) AS orders

FROM orders

GROUP BY user\_id)

SELECT

-- Write the conditions for the three buckets

CASE

WHEN orders < 8 THEN 'Low-orders users'

WHEN orders < 15 THEN 'Mid-orders users'

ELSE 'High-orders users'

END AS order\_group,

-- Count the distinct users in each bucket

COUNT (DISTINCT user\_id) AS users

FROM user\_orders

GROUP BY order\_group;

# Revenue quartiles

Dave is wrapping up his study, and wants to calculate a few more figures. He wants to find out the first, second, and third revenue quartiles. He also wants to find the average to see in which direction the data is skewed.

Calculate the first, second, and third revenue quartiles, as well as the average.

**Note**: You can calculate the 30th percentile for a column named column\_a by using PERCENTILE\_CONT(0.30) WITHIN GROUP (ORDER BY column\_a ASC).

* Store each user ID and the revenue Delivr generates from it in the user\_revenues CTE.
* Calculate the first, second, and third revenue quartile.
* Calculate the average revenue.
* WITH user\_revenues AS (
* -- Select the user IDs and their revenues
* SELECT
* user\_id,
* SUM (meal\_price\* order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m. meal\_id = o.meal\_id
* GROUP BY user\_id)
* SELECT
* --Calculate the first, second, and third quartile
* ROUND (
* PERCENTILE\_CONT (0.25) WITHIN GROUP (ORDER BY revenue ASC) :: NUMERIC,
* 2) AS revenue\_p25,
* ROUND(
* PERCENTILE\_CONT (0.5) WITHIN GROUP (ORDER BY revenue ASC) :: NUMERIC,
* 2) AS revenue\_p50,
* ROUND(
* PERCENTILE\_CONT (0.75) WITHIN GROUP (ORDER BY revenue ASC) :: NUMERIC,
* 2) AS revenue\_p75,
* --Calculate the average
* ROUND (AVG (revenue) :: NUMERIC, 2) AS avg\_revenue
* FROM user\_revenues;

# Interquartile range

The final value that Dave wants is the count of users in the revenue interquartile range (IQR). Users outside the revenue IQR are outliers, and Dave wants to know the number of "typical" users.

Return the count of users in the revenue IQR.

* Return a table of user IDs and generated revenues for each user.
* SELECT
* -- Select user\_id and calculate revenue by user
* user\_id,
* sum(meal\_price \* order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m.meal\_id = o.meal\_id
* GROUP BY user\_id;

# Interquartile range

The final value that Dave wants is the count of users in the revenue interquartile range (IQR). Users outside the revenue IQR are outliers, and Dave wants to know the number of "typical" users.

Return the count of users in the revenue IQR.

* Wrap the previous query in a CTE named user\_revenues.
* Calculate the first and third revenue quartiles.

WITH user\_revenues AS (

SELECT

-- Select user\_id and calculate revenue by user

user\_id,

SUM(m.meal\_price\* o.order\_quantity) AS revenue

FROM meals AS m

JOIN orders AS o ON m. meal\_id = o.meal\_id GROUP BY user\_id)

SELECT

--Calculate the first and third revenue quartiles

ROUND (

PERCENTILE\_CONT (0.25) WITHIN GROUP (ORDER BY revenue ASC) :: NUMERIC,

2) AS revenue\_p25,

ROUND (

PERCENTILE\_CONT (0.75) WITHIN GROUP (ORDER BY revenue ASC) :: NUMERIC,

2) AS revenue\_p75

FROM user\_revenues;

# Interquartile range

The final value that Dave wants is the count of users in the revenue interquartile range (IQR). Users outside the revenue IQR are outliers, and Dave wants to know the number of "typical" users.

Return the count of users in the revenue IQR.

* Count the number of distinct users.
* Filter out all users outside the IQR.
* WITH user\_revenues AS (
* SELECT
* -- Select user\_id and calculate revenue by user
* user\_id,
* SUM(m.meal\_price \* o.order\_quantity) AS revenue
* FROM meals AS m
* JOIN orders AS o ON m.meal\_id = o.meal\_id
* GROUP BY user\_id),
* quartiles AS (
* SELECT
* -- Calculate the first and third revenue quartiles
* ROUND(
* PERCENTILE\_CONT(0.25) WITHIN GROUP
* (ORDER BY revenue ASC) :: NUMERIC,
* 2) AS revenue\_p25,
* ROUND(
* PERCENTILE\_CONT(0.75) WITHIN GROUP
* (ORDER BY revenue ASC) :: NUMERIC,
* 2) AS revenue\_p75
* FROM user\_revenues)
* SELECT
* -- Count the number of users in the IQR
* COUNT(DISTINCT user\_id) AS users
* FROM user\_revenues
* CROSS JOIN quartiles
* -- Only keep users with revenues in the IQR range
* WHERE revenue :: NUMERIC >= revenue\_p25
* AND revenue :: NUMERIC <= revenue\_p75;
* **Formatting dates**
* Eve from the Business Intelligence (BI) team lets you know that she's gonna need your help to write queries for reports. The reports are read by C-level execs, so they need to be as readable and quick to scan as possible. Eve tells you that the C-level execs' preferred date format is something like Friday 01, June 2018 for 2018-06-01.
* You have a list of useful patterns.

| **Pattern** | **Description** |
| --- | --- |
| DD | Day number (01 - 31) |
| FMDay | Full day name (Monday, Tuesday, *etc.*) |
| FMMonth | Full month name (January, February, *etc.*) |
| YYYY | Full 4-digit year (2018, 2019, *etc.*) |

Figure out the format string that formats 2018-06-01 as "Friday 01, June 2018" when using TO\_CHAR.

* Select the order date.
* Format the order date so that 2018-06-01 is formatted as Friday 01, June 2018.
* SELECT DISTINCT
* -- Select the order date
* order\_date,
* --Format the order date
* TO\_CHAR (order\_date, 'FMDay DD, FMMonth YYYY') AS format\_order\_date
* FROM orders
* ORDER BY order\_date ASC
* LIMIT 3;

# Rank users by their count of orders

Eve tells you that she wants to report which user IDs have the most orders each month. She doesn't want to display long numbers, which will only distract C-level execs, so she wants to display only their ranks. The top 1 rank goes to the user with the most orders, the second-top 2 rank goes to the user with the second-most orders, and so on.

Send Eve a list of the top 3 user IDs by orders in August 2018 with their ranks.

* Keep only the orders in August 2018.
* SELECT
* user\_id,
* COUNT (DISTINCT order\_id) AS count\_orders
* FROM orders
* --Only keep orders in August 2018
* WHERE DATE\_TRUNC ('month', order\_date) = '2018-08-01'
* GROUP BY user\_id;
* Wrap the previous query in a CTE named user\_count\_orders.
* Select the user ID and rank all user IDs by the count of orders in descending order.
* Only keep the top 3 users by their count of orders.
* -- Set up the user\_count\_orders CTE
* WITH user\_count\_orders AS (
* SELECT
* user\_id,
* COUNT (DISTINCT order\_id) AS count\_orders
* FROM orders
* --Only keep orders in August 2018
* WHERE DATE\_TRUNC ('month', order\_date) = '2018-08-01' GROUP BY user\_id)
* SELECT
* --Select user ID, and rank user ID by count\_orders
* user\_id,
* RANK () OVER (ORDER BY count\_orders DESC) AS count\_orders\_rank
* FROM user\_count\_orders
* ORDER BY count\_orders\_rank ASC
* -- Limit the user IDs selected to 3
* LIMIT 3;

# Pivoting user revenues by month

Next, Eve tells you that the C-level execs prefer wide tables over long ones because they're easier to scan. She prepared a sample report of user revenues by month, detailing the first 5 user IDs' revenues from June to August 2018. The execs told her to pivot the table by month. She's passed that task off to you.

Pivot the user revenues by month query so that the user ID is a row and each month from June to August 2018 is a column.

* Enable CROSSTAB() from tablefunc.
* Declare the new pivot table's columns, user ID and the first three months of operation.
* CREATE EXTENSION IF NOT EXISTS tablefunc;
* SELECT \* FROM CROSSTAB ($$
* SELECT
* user\_id,
* DATE\_TRUNC ('month', order\_date) :: DATE AS delivr\_month, SUM (meal\_price\* order\_quantity) :: FLOAT AS revenue
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* WHERE user\_id IN (0, 1, 2, 3, 4)
* AND order\_date< '2018-09-01'
* GROUP BY user\_id, delivr\_month
* ORDER BY user\_id, delivr\_month;
* $$)
* -- Select user ID and the months from June to August 2018
* AS ct (user\_id INT,
* "2018-06-01" FLOAT,
* "2018-07-01" FLOAT,
* "2018-08-01" FLOAT)
* ORDER BY user\_id ASC;

# Costs

The C-level execs next tell Eve that they want a report on the total costs by eatery in the last two months.

First, write a query to get the total costs by eatery in November and December 2018, then pivot by month.

**Note**: Recall from Chapter 1 that total cost is the sum of each meal's cost times its stocking quantity.

* Select the eatery and calculate total cost per eatery.
* Keep only the records after October 2018.

SELECT

-- Select eatery and calculate total cost

eatery,

DATE\_TRUNC ('month', stocking\_date) :: DATE AS delivr\_month,

SUM (meal\_cost \* stocked\_quantity) :: FLOAT AS cost

FROM meals

JOIN stock ON meals.meal\_id = stock.meal\_id

-- Keep only the records after October 2018

WHERE stocking\_date >

'2018-10-01¹

GROUP BY eatery, delivr\_month

ORDER BY eatery, delivr\_month;

* Enable CROSSTAB from tablefunc.
* Declare the new pivot table's columns, the eatery and the last two months of operation.
* CREATE EXTENSION IF NOT EXISTS tablefunc;
* SELECT \* FROM CROSSTAB ($$
* SELECT
* -- Select eatery and calculate total cost
* eatery,
* DATE\_TRUNC ('month', stocking\_date) :: DATE AS delivr\_month,
* SUM (meal\_cost \* stocked\_quantity) :: FLOAT AS cost
* FROM meals
* JOIN stock ON meals.meal\_id = stock.meal\_id
* --Keep only the records after October 2018
* WHERE DATE\_TRUNC ('month', stocking\_date) > '2018-10-01'
* GROUP BY eatery, delivr\_month
* ORDER BY eatery, delivr\_month;
* $$)
* -- Select the eatery and November and December 2018 as columns
* AS ct (eatery TEXT,
* "2018-11-01" FLOAT,
* "2018-12-01" FLOAT)
* ORDER BY eatery ASC;

# Executive report

Eve wants to produce a final executive report about the rankings of eateries by the number of unique users who order from them by quarter. She said she'll handle the pivoting, so you only need to prepare the source table for her to pivot.

Send Eve a table of unique ordering users by eatery and by quarter.

* Fill in the format string that formats 2018-06-01 as Q2 2018.
* Count the ordering users by eatery and by quarter.
* SELECT
* eatery,
* -- Format the order date so "2018-06-01" becomes "Q2 2018"
* TO\_CHAR (order\_date, '"Q"Q YYYY') AS delivr\_quarter,
* -- Count unique users
* COUNT (DISTINCT user\_id) AS users
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* GROUP BY eatery, delivr\_quarter
* ORDER BY delivr\_quarter, users;
* Select the eatery and the quarter from the CTE.
* Assign a rank to each row, with the top-most rank going to the row with the highest orders.
* WITH eatery\_users AS (
* SELECT
* eatery,
* --Format the order date so "2018-06-01" becomes "Q2 2018"
* TO\_CHAR(order\_date, '"Q"Q YYYY') AS delivr\_quarter,
* --Count unique users
* COUNT (DISTINCT user\_id) AS users
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* GROUP BY eatery, delivr\_quarter
* ORDER BY delivr\_quarter, users)
* SELECT
* -- Select eatery and quarter
* eatery,
* delivr\_quarter,
* -- Rank rows, partition by quarter and order by users
* RANK () OVER
* (PARTITION BY delivr\_quarter
* ORDER BY users DESC) :: INT AS users\_rank
* FROM eatery\_users
* ORDER BY delivr\_quarter, users\_rank;
* Import the tablefunc extension.
* Pivot the table by quarter.
* Select the new columns from the pivoted table.
* CREATE EXTENSION IF NOT EXISTS tablefunc;
* -- Pivot the previous query by quarter
* SELECT \* FROM CROSSTAB ($$
* WITH eatery\_users AS (
* SELECT
* eatery,
* -- Format the order date so "2018-06-01" becomes "Q2 2018"
* TO\_CHAR(order\_date, '"Q"Q YYYY') AS delivr\_quarter,
* --- Count unique users
* COUNT (DISTINCT user\_id) AS users
* FROM meals
* JOIN orders ON meals.meal\_id = orders.meal\_id
* GROUP BY eatery, delivr\_quarter
* ORDER BY delivr\_quarter, users)
* SELECT
* --Select eatery and quarter
* eatery,
* delivr\_quarter,
* -- Rank rows, partition by quarter and order by users
* RANK () OVER
* (PARTITION BY delivr\_quarter
* ORDER BY users DESC) :: INT AS users\_rank
* FROM eatery\_users
* ORDER BY eatery, delivr\_quarter;
* $$)
* -- Select the columns of the pivoted table
* AS ct (eatery TEXT,
* "Q2 2018" INT,
* "Q3 2018" INT,
* "Q4 2018" INT)
* ORDER BY "Q4 2018";