Write an SQL query to find the confirmation rate of each user.
The confirmation rate of a user is the number of 'confirmed'
messages divided by the total number of requested confirmation messages. The
confirmation rate of a user that did not request any confirmation messages is 0. Round
the confirmation rate to two decimal places. */

-- **Table -1**
CREATE TABLE Signups (user_id INT PRIMARY KEY, time_stamp
DATETIME);

-- **Insert the data**
INSERT INTO Signups (user_id, time_stamp)
VALUES (3, '2020-03-21 10:16:13'),(7, '2020-01-04 13:57:59'),
(2, '2020-07-29 23:09:44'),(6, '2020-12-09 10:39:37');

-- **Table - 2**
CREATE TABLE Confirmations (user_id INT,time_stamp DATETIME,
action VARCHAR(10) CHECK (action IN ('confirmed', 'timeout')),
PRIMARY KEY (user_id, time_stamp),
FOREIGN KEY (user_id) REFERENCES Signups(user_id));

-- **Insert the data**
INSERT INTO Confirmations (user_id, time_stamp, action)
VALUES
(3, '2021-01-06 03:30:46', 'timeout'),(3, '2021-07-14 14:00:00', 'timeout'),
(7, '2021-06-12 11:57:29', 'confirmed'),(7, '2021-06-13 12:58:28', 'confirmed'),
(7, '2021-06-14 13:59:27', 'confirmed'),(2, '2021-01-22 00:00:00', 'confirmed'),
(2, '2021-02-28 23:59:59', 'timeout');

%sql
SELECT s.user_id, ROUND(
CASE WHEN COUNT(c.action) = 0 THEN 0 ELSE SUM(CASE WHEN c.action = 'confirmed'
THEN 1 ELSE 0 END) * 1.0 / COUNT(c.action) END, 2) AS confirmation_rate FROM
Signups s LEFT JOIN Confirmations c ON s.user_id = c.user_id GROUP BY s.user_id
ORDER BY s.user_id;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, count, sum as _sum, round

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Create Signups DataFrame
signups_data = [
    (3, '2020-03-21 10:16:13'),
    (7, '2020-01-04 13:57:59'),
    (2, '2020-07-29 23:09:44'),
    (6, '2020-12-09 10:39:37')]

signups_df = spark.createDataFrame(signups_data, ['user_id', 'time_stamp'])

# Create Confirmations DataFrame
confirmations_data = [
    (3, '2021-01-06 03:30:46', 'timeout'),

```
    (3, '2021-07-14 14:00:00', 'timeout'),
    (7, '2021-06-12 11:57:29', 'confirmed'),
    (7, '2021-06-13 12:58:28', 'confirmed'),
    (7, '2021-06-14 13:59:27', 'confirmed'),
    (2, '2021-01-22 00:00:00', 'confirmed'),
    (2, '2021-02-28 23:59:59', 'timeout')]

confirmations_df = spark.createDataFrame(confirmations_data, ['user_id',
'time_stamp', 'action'])

# Join Signups with Confirmations (left join)
joined_df = signups_df.join(confirmations_df, on='user_id', how='left')

# Compute confirmation rate
result_df = joined_df.groupBy("user_id").agg(round(when(count("action") == 0,
0).otherwise(_sum(when(col("action") == "confirmed", 1).otherwise(0)) /
count("action")), 2).alias("confirmation_rate"))

# Show result
result_df.orderBy("user_id").show()
```

Write a SQL query to determine the count of delayed orders for each delivery partner.

```
-- Table
CREATE TABLE order_details (orderid INT PRIMARY KEY,custid INT,
city VARCHAR(50),order_date DATE,del_partner VARCHAR(50),
 order_time TIME,deliver_time TIME,predicted_time INT,
aov DECIMAL(10, 2));
```



```
-- Insert the data
INSERT INTO order_details
VALUES
(1, 101, 'Bangalore', '2024-01-01', 'PartnerA', '10:00:00', '11:30:00', 60, 100.00),
(2, 102, 'Chennai', '2024-01-02', 'PartnerB', '12:00:00', '13:15:00', 45, 200.00),
(3, 103, 'Bangalore', '2024-01-03', 'PartnerA', '14:00:00', '15:45:00', 60, 300.00),
(4, 104, 'Chennai', '2024-01-04', 'PartnerB', '16:00:00', '17:30:00', 90, 400.00);

%sql
SELECT del_partner, COUNT(*) AS delayed_orders FROM order_details WHERE
EXTRACT(HOUR FROM deliver_time - order_time) * 60 + EXTRACT(MINUTE FROM
deliver_time - order_time) > predicted_time GROUP BY del_partner;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, expr, unix_timestamp, count

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Create DataFrame for order_details
data = [
    (1, 101, 'Bangalore', '2024-01-01', 'PartnerA', '10:00:00', '11:30:00', 60, 100.00),
    (2, 102, 'Chennai', '2024-01-02', 'PartnerB', '12:00:00', '13:15:00', 45, 200.00),
    (3, 103, 'Bangalore', '2024-01-03', 'PartnerA', '14:00:00', '15:45:00', 60, 300.00),
```

```
    (4, 104, 'Chennai', '2024-01-04', 'PartnerB', '16:00:00', '17:30:00', 90, 400.00)]

columns = ['orderid', 'custid', 'city', 'order_date', 'del_partner', 'order_time',
'deliver_time', 'predicted_time', 'aov']

df = spark.createDataFrame(data, columns)

# Combine order_date with times to create full timestamps
df = df.withColumn("order_ts", expr("to_timestamp(concat(order_date, ' ',
order_time))")) \
       .withColumn("deliver_ts", expr("to_timestamp(concat(order_date, ' ',
deliver_time))"))

# Calculate actual duration in minutes
df = df.withColumn("actual_minutes",
                   (unix_timestamp(col("deliver_ts")) - unix_timestamp(col("order_ts"))) / 60)

# Filter delayed orders
delayed_df = df.filter(col("actual_minutes") > col("predicted_time"))

# Count delayed orders per delivery partner
result_df = delayed_df.groupBy("del_partner").agg(count("*").alias("delayed_orders"))

# Show result
result_df.show()
-----------------------------------------------
```

Write a query to obtain a breakdown of the time spent sending vs. opening snaps as a percentage of total time spent on these activities grouped by age group. Round the percentage to 2 decimal places in the output.

Notes:-
Calculate the following percentages:
time spent sending / (Time spent sending + Time spent opening)
Time spent opening / (Time spent sending + Time spent opening)
To avoid integer division in percentages, multiply by 100.0 and not 100.
*/

```
-- Table -1
CREATE TABLE age_breakdown (
 user_id INT PRIMARY KEY,
 age_bucket VARCHAR(10));

-- Insert the data
INSERT INTO age_breakdown(user_id, age_bucket) VALUES
(123, '31-35'),
(456, '26-30'),
(789, '21-25');

-- Table - 2
CREATE TABLE activities (
 activity_id INT PRIMARY KEY,
 user_id INT,
 activity_type VARCHAR(10),
 time_spent DECIMAL(5,2),
```

```
 activity_date DATETIME,
 FOREIGN KEY (user_id) REFERENCES age_breakdown(user_id) ON DELETE CASCADE);

-- Insert the data
INSERT INTO activities(activity_id, user_id, activity_type, time_spent, activity_date)
VALUES
(7274, 123, 'open', 4.50, '2022-06-22 12:00:00'),
(2425, 123, 'send', 3.50, '2022-06-22 12:00:00'),
(1413, 456, 'send', 5.67, '2022-06-23 12:00:00'),
(2536, 456, 'open', 3.00, '2022-06-25 12:00:00'),
(8564, 456, 'send', 8.24, '2022-06-26 12:00:00'),
(5235, 789, 'send', 6.24, '2022-06-28 12:00:00'),
(4251, 123, 'open', 1.25, '2022-07-01 12:00:00'),
(1414, 789, 'chat', 11.00, '2022-06-25 12:00:00'),
(1314, 123, 'chat', 3.15, '2022-06-26 12:00:00'),
(1435, 789, 'open', 5.25, '2022-07-02 12:00:00');

%sql
SELECT ab.age_bucket,ROUND(
SUM(CASE WHEN a.activity_type = 'send' THEN a.time_spent ELSE 0 END) * 100.0 /
SUM(CASE WHEN a.activity_type IN ('send', 'open') THEN a.time_spent ELSE 0 END), 2)
AS send_pct, ROUND(
SUM(CASE WHEN a.activity_type = 'open' THEN a.time_spent ELSE 0 END) * 100.0 /
SUM(CASE WHEN a.activity_type IN ('send', 'open') THEN a.time_spent ELSE 0 END), 2)
AS open_pct
FROM age_breakdown ab JOIN activities a ON ab.user_id = a.user_id WHERE
a.activity_type IN ('send', 'open') GROUP BY ab.age_bucket ORDER BY
ab.age_bucket;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, sum as _sum, round

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Sample DataFrames

# Age Breakdown Table
age_data = [
    (123, '31-35'),
    (456, '26-30'),
    (789, '21-25')
]
age_columns = ['user_id', 'age_bucket']
age_df = spark.createDataFrame(age_data, age_columns)

# Activities Table
activities_data = [
    (7274, 123, 'open', 4.50, '2022-06-22 12:00:00'),
    (2425, 123, 'send', 3.50, '2022-06-22 12:00:00'),
    (1413, 456, 'send', 5.67, '2022-06-23 12:00:00'),
    (2536, 456, 'open', 3.00, '2022-06-25 12:00:00'),
    (8564, 456, 'send', 8.24, '2022-06-26 12:00:00'),
    (5235, 789, 'send', 6.24, '2022-06-28 12:00:00'),
```

```
    (4251, 123, 'open', 1.25, '2022-07-01 12:00:00'),
    (1414, 789, 'chat', 11.00, '2022-06-25 12:00:00'),
    (1314, 123, 'chat', 3.15, '2022-06-26 12:00:00'),
    (1435, 789, 'open', 5.25, '2022-07-02 12:00:00')]

activities_columns = ['activity_id', 'user_id', 'activity_type', 'time_spent', 'activity_date']
activities_df = spark.createDataFrame(activities_data, activities_columns)

# Join tables
joined_df = activities_df.join(age_df, on="user_id")

# Filter for only 'send' and 'open'
filtered_df = joined_df.filter(col("activity_type").isin("send", "open"))

# Aggregate and calculate percentages
agg_df = filtered_df.groupBy("age_bucket").agg(
    _sum(when(col("activity_type") == "send",
col("time_spent")).otherwise(0)).alias("send_time"),
    _sum(when(col("activity_type") == "open",
col("time_spent")).otherwise(0)).alias("open_time"))

# Final percentage calculation
result_df = agg_df.withColumn("send_pct", round(col("send_time") * 100.0 /
(col("send_time") + col("open_time")), 2)).withColumn("open_pct",
round(col("open_time") * 100.0 / (col("send_time") + col("open_time")),
2)).select("age_bucket", "send_pct", "open_pct")

# Show result
result_df.orderBy("age_bucket").show()
```

Write a query to calculate the sum of odd-numbered and even-numbered measurements
separately for a particular day and display the results in two different columns.
Refer to the Example Output below for the desired format.
*/


```
-- Table
CREATE TABLE Measurements (
 measurement_id INT,
 measurement_value DECIMAL(10, 2),
 measurement_time DATETIME);

-- Insert the data
INSERT INTO Measurements (measurement_id, measurement_value,
measurement_time) VALUES
(131233, 1109.51, '2022-07-10 09:00:00'),(135211, 1662.74, '2022-07-10 11:00:00'),
(143562, 1124.50, '2022-07-11 13:15:00'),(346462, 1234.14, '2022-07-11 15:00:00'),
(124245, 1252.62, '2022-07-11 16:45:00'),(523542, 1246.24, '2022-07-10 14:30:00'),
(143251, 1246.56, '2022-07-11 18:00:00'),(141565, 1452.40, '2022-07-12 08:00:00'),
(253622, 1244.30, '2022-07-12 14:00:00'),(353625, 1451.00, '2022-07-12 15:00:00');


%sql
SELECT TO_CHAR(measurement_time, 'YYYY-MM-DD') AS measurement_date,
```

```
ROUND(SUM(CASE WHEN MOD(measurement_id, 2) = 0 THEN measurement_value
ELSE 0 END), 2) AS even_sum, ROUND(SUM(CASE WHEN MOD(measurement_id, 2) <> 0
THEN measurement_value ELSE 0 END), 2) AS odd_sum FROM Measurements WHERE
TO_CHAR(measurement_time, 'YYYY-MM-DD') = '2022-07-11' GROUP BY
TO_CHAR(measurement_time, 'YYYY-MM-DD');

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum as _sum, to_date, when, round

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Sample data
data = [
    (131233, 1109.51, '2022-07-10 09:00:00'),
    (135211, 1662.74, '2022-07-10 11:00:00'),
    (143562, 1124.50, '2022-07-11 13:15:00'),
    (346462, 1234.14, '2022-07-11 15:00:00'),
    (124245, 1252.62, '2022-07-11 16:45:00'),
    (523542, 1246.24, '2022-07-10 14:30:00'),
    (143251, 1246.56, '2022-07-11 18:00:00'),
    (141565, 1452.40, '2022-07-12 08:00:00'),
    (253622, 1244.30, '2022-07-12 14:00:00'),
    (353625, 1451.00, '2022-07-12 15:00:00')]

columns = ['measurement_id', 'measurement_value', 'measurement_time']
df = spark.createDataFrame(data, columns)

# Convert string to timestamp and extract date
df = df.withColumn("measurement_time", col("measurement_time").cast("timestamp"))
df = df.withColumn("measurement_date", to_date("measurement_time"))

# Filter for specific date
filtered_df = df.filter(col("measurement_date") == '2022-07-11')

# Aggregate even and odd separately
result_df = filtered_df.agg(
    round(_sum(when(col("measurement_id") % 2 == 0,
col("measurement_value")).otherwise(0)), 2).alias("even_sum"),
    round(_sum(when(col("measurement_id") % 2 != 0,
col("measurement_value")).otherwise(0)), 2).alias("odd_sum"))

result_df.show()
```

The Bloomberg terminal is the go-to resource for financial professionals, offering convenient access to a wide array of financial datasets. As a Data Analyst at Bloomberg,you have access to historical data on stock performance.

Currently, you're analyzing the highest and lowest open prices for each FAANG stock by month over the years.

For each FAANG stock, display the ticker symbol, the month and year ('Mon-YYYY') with the corresponding highest and lowest open prices (refer to the Example Output

format).Ensure that the results are sorted by ticker symbol.
*/

-- **Table**
```
CREATE TABLE stockprices (stock_date DATETIME,ticker VARCHAR(10),
 openprice DECIMAL(10,2),highprice DECIMAL(10,2),
 lowprice DECIMAL(10,2),closeprice DECIMAL(10,2));

%sql
SELECT ticker, TO_CHAR(stock_date, 'Mon-YYYY') AS month_year, MAX(openprice) AS
highest_open, MIN(openprice) AS lowest_open FROM stockprices WHERE ticker IN ('FB',
'AAPL', 'AMZN', 'NFLX', 'GOOG')  GROUP BY ticker,TO_CHAR(stock_date, 'Mon-YYYY')
ORDER BY ticker,TO_DATE(TO_CHAR(stock_date, 'Mon-YYYY'), 'Mon-YYYY');

%python

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, date_format, max as _max, min as _min

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Sample FAANG data (replace with actual data as needed)
data = [
    ('2022-01-03', 'AAPL', 170.0, 175.0, 169.0, 174.0),
    ('2022-01-10', 'AAPL', 172.0, 176.0, 170.0, 175.0),
    ('2022-01-04', 'GOOG', 2800.0, 2825.0, 2790.0, 2810.0),
    ('2022-01-11', 'GOOG', 2780.0, 2795.0, 2750.0, 2775.0)]

columns = ['stock_date', 'ticker', 'openprice', 'highprice', 'lowprice', 'closeprice']
df = spark.createDataFrame(data, columns)

# Cast date and filter FAANG
df = df.withColumn("stock_date", col("stock_date").cast("date"))
faang = ['FB', 'AAPL', 'AMZN', 'NFLX', 'GOOG']
df = df.filter(col("ticker").isin(faang))

# Create month-year column
df = df.withColumn("month_year", date_format("stock_date", "MMM-yyyy"))

# Aggregate by ticker and month
result_df = df.groupBy("ticker", "month_year").agg(
    _max("openprice").alias("highest_open"),
    _min("openprice").alias("lowest_open")).orderBy("ticker", "month_year")

result_df.show(truncate=False)
```

Write an SQL query to find the percentage of immediate orders in
the first orders of all customers, rounded to 2 decimal places.

The first order of a customer is the order with the earliest order
date that customer made. It is guaranteed that a customer has exactly one first order.

If the preferred delivery date of the customer is the same as
the order date then the order is called immediate otherwise it's called scheduled.

```
*/

-- Table
CREATE TABLE Delivery (
 delivery_id INT PRIMARY KEY,
 customer_id INT NOT NULL,
 order_date DATE NOT NULL,
 customer_pref_delivery_date DATE NOT NULL)

-- Insert the data
INSERT INTO Delivery (delivery_id, customer_id, order_date,
customer_pref_delivery_date) VALUES
(1, 1, '2019-08-01', '2019-08-02'),(2, 2, '2019-08-02', '2019-08-02'),
(3, 1, '2019-08-11', '2019-08-12'),(4, 3, '2019-08-24', '2019-08-24'),
(5, 3, '2019-08-21', '2019-08-22'),(6, 2, '2019-08-11', '2019-08-13'),
(7, 4, '2019-08-09', '2019-08-09');

%sql
WITH FirstOrders AS (
SELECT * FROM Delivery d WHERE order_date = (
SELECT MIN(order_date) FROM Delivery WHERE customer_id = d.customer_id))
SELECT ROUND(
(COUNT(CASE WHEN order_date = customer_pref_delivery_date THEN 1 END) * 100.0) /
COUNT(*),2) AS immediate_percentage FROM FirstOrders;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, min as _min, count, when, round as _round

# Create Spark session
spark = SparkSession.builder.getOrCreate()

# Create DataFrame
data = [
    (1, 1, '2019-08-01', '2019-08-02'),
    (2, 2, '2019-08-02', '2019-08-02'),
    (3, 1, '2019-08-11', '2019-08-12'),
    (4, 3, '2019-08-24', '2019-08-24'),
    (5, 3, '2019-08-21', '2019-08-22'),
    (6, 2, '2019-08-11', '2019-08-13'),
    (7, 4, '2019-08-09', '2019-08-09')]

columns = ['delivery_id', 'customer_id', 'order_date', 'customer_pref_delivery_date']
df = spark.createDataFrame(data, columns)

# Convert date columns to date type
df = df.withColumn("order_date", col("order_date").cast("date"))
df = df.withColumn("customer_pref_delivery_date",
col("customer_pref_delivery_date").cast("date"))

# Get first order per customer
first_orders = df.join(
    df.groupBy("customer_id").agg(_min("order_date").alias("first_order_date")),
    on=["customer_id", "order_date"])
```

```
# Calculate stats
result_df =
first_orders.select(count("*").alias("total_first_orders"),count(when(col("order_date") ==
col("customer_pref_delivery_date"),
True)).alias("immediate_orders")).withColumn("immediate_percentage",
_round((col("immediate_orders") * 100.0) / col("total_first_orders"), 2))

result_df.show()
```

Write an SQL query to find the salaries of the employees after applying taxes.

The tax rate is calculated for each company based on the following criteria:
0% If the max salary of any employee in the company is less than 1000$. 24% If the max salary of any employee in the company is in the range [1000, 10000] inclusive. 49% If the max salary of any employee in the company is greater than 10000$.

Return the result table in any order. Round the salary to the nearest integer.
*/

-- **Table**


```
CREATE TABLE Salaries (
 company_id INT,
 employee_id INT,
 employee_name VARCHAR(50),
 salary INT,
 PRIMARY KEY (company_id, employee_id));
```

- **Insert the data**

```
INSERT INTO Salaries (company_id, employee_id, employee_name, salary) VALUES
(1, 1, 'Tony', 2000),(1, 2, 'Pronub', 21300),
(1, 3, 'Tyrrox', 10800),(2, 1, 'Pam', 300),
(2, 7, 'Bassem', 450),(2, 9, 'Hermione', 700),
(3, 7, 'Bocaben', 100),(3, 2, 'Ognjen', 2200),
(3, 13, 'Nyancat', 3300),(3, 15, 'Morninngcat', 7777);

%sql
WITH CompanyTax AS (
SELECT company_id, CASE WHEN MAX(salary) < 1000 THEN 0 WHEN MAX(salary)
BETWEEN 1000 AND 10000 THEN 0.24 ELSE 0.49 END AS tax_rate FROM Salaries GROUP
BY company_id)
SELECT s.company_id, s.employee_id, s.employee_name, ROUND(s.salary * (1 -
ct.tax_rate)) AS after_tax_salary FROM Salaries s JOIN CompanyTax ct ON
s.company_id = ct.company_id;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, max as _max, when, round as _round

# Initialize Spark
spark = SparkSession.builder.getOrCreate()

# Sample Data
data = [
    (1, 1, 'Tony', 2000),
    (1, 2, 'Pronub', 21300),
```

```
    (1, 3, 'Tyrrox', 10800),
    (2, 1, 'Pam', 300),
    (2, 7, 'Bassem', 450),
    (2, 9, 'Hermione', 700),
    (3, 7, 'Bocaben', 100),
    (3, 2, 'Ognjen', 2200),
    (3, 13, 'Nyancat', 3300),
    (3, 15, 'Morninngcat', 7777)]

columns = ['company_id', 'employee_id', 'employee_name', 'salary']
df = spark.createDataFrame(data, columns)

# Step 1: Get max salary per company
max_salary_df = df.groupBy("company_id").agg(_max("salary").alias("max_salary"))

# Step 2: Assign tax rate based on max salary
tax_df = max_salary_df.withColumn(
    "tax_rate",
    when(col("max_salary") < 1000, 0.0)
    .when((col("max_salary") >= 1000) & (col("max_salary") <= 10000), 0.24)
    .otherwise(0.49))

# Step 3: Join with original data to compute post-tax salary
result_df = df.join(tax_df, on="company_id", how="inner").withColumn(
    "after_tax_salary", _round(col("salary") * (1 - col("tax_rate"))))

# Step 4: Select desired columns
final_df = result_df.select("company_id", "employee_id", "employee_name",
"after_tax_salary")

# Show result
final_df.show()
```

/*Write a query to calculate the percentage of total transaction
volume processed via PayPal in year 2024.
*/


```
-- Table
CREATE TABLE Transactions (
TransactionID VARCHAR(10) PRIMARY KEY,
UserID VARCHAR(10),
PaymentMethod VARCHAR(20),
TransactionAmount DECIMAL(18,10),
TransactionDate DATE );

%sql
SELECT ROUND(100.0 * SUM(CASE WHEN PaymentMethod = 'PayPal' THEN
TransactionAmount ELSE 0 END) / SUM(TransactionAmount),2) AS paypal_percentage
FROM Transactions WHERE EXTRACT(YEAR FROM TransactionDate) = 2024;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year, when, sum as _sum, round as _round

# Initialize SparkSession
```

```
spark = SparkSession.builder.getOrCreate()

# Sample Data
data = [
    ("TXN1", "U1", "PayPal", 100.00, "2024-01-10"),
    ("TXN2", "U2", "CreditCard", 250.00, "2024-02-15"),
    ("TXN3", "U3", "PayPal", 150.00, "2024-03-20"),
    ("TXN4", "U4", "ApplePay", 200.00, "2023-12-25"),
    ("TXN5", "U5", "PayPal", 50.00, "2024-05-05")]

columns = ["TransactionID", "UserID", "PaymentMethod", "TransactionAmount",
"TransactionDate"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Convert TransactionDate to DateType
from pyspark.sql.functions import to_date
df = df.withColumn("TransactionDate", to_date(col("TransactionDate")))

# Filter for 2024 transactions
df_2024 = df.filter(year("TransactionDate") == 2024)

# Calculate total and PayPal transaction volumes
result_df = df_2024.withColumn(
    "paypal_amount",
    when(col("PaymentMethod") == "PayPal", col("TransactionAmount")).otherwise(0)
).agg(_round((_sum("paypal_amount") / _sum("TransactionAmount")) * 100,
2).alias("paypal_percentage"))

# Show result
result_df.show()
-----------------------------------------------------
Risk is calculated by the addition of CBC, RBH, and LBH. If the sum is more than 20,
"High", between 16-20 then "Medium", else "Low"
*/
```



```
-- Table -1
CREATE TABLE insurance (
 Patient_id INT PRIMARY KEY,
 Insurance_id INT,
 Insurance_Name VARCHAR(50));

-- Insert the data
INSERT INTO insurance (Patient_id, Insurance_id, Insurance_Name)
VALUES
 (1, 1001, 'India Insurance'),(2, 1002, 'ICICI Lombard'),
 (3, 1001, 'India Insurance'),(4, 1003, 'Star Health'),
 (5, 1003, 'Star Health');

-- Table - 2
CREATE TABLE test (
 Patient_id INT,
 Test_type VARCHAR(10),
 Test_score INT,
```

```
 FOREIGN KEY (Patient_id) REFERENCES insurance(Patient_id));

 -- Insert the data
INSERT INTO test (Patient_id, Test_type, Test_score)
VALUES
(1, 'CBC', 7),(1, 'RBC', 6),(1, 'LBH', 6),(2, 'CBC', 7),(2, 'RBC', 8),(2, 'LBH', 8),
(3, 'CBC', 5),(3, 'RBC', 4),(3, 'LBH', 4),(4, 'CBC', 4),(4, 'RBC', 6),(4, 'LBH', 6),
(5, 'CBC', 5),(5, 'RBC', 6),(5, 'LBH', 7);

%sql
SELECT
  i.Patient_id,
  i.Insurance_Name,
  SUM(CASE WHEN t.Test_type = 'CBC' THEN t.Test_score ELSE 0 END) +
  SUM(CASE WHEN t.Test_type = 'RBC' THEN t.Test_score ELSE 0 END) +
  SUM(CASE WHEN t.Test_type = 'LBH' THEN t.Test_score ELSE 0 END) AS total_score,

  CASE
    WHEN (
      SUM(CASE WHEN t.Test_type = 'CBC' THEN t.Test_score ELSE 0 END) +
      SUM(CASE WHEN t.Test_type = 'RBC' THEN t.Test_score ELSE 0 END) +
      SUM(CASE WHEN t.Test_type = 'LBH' THEN t.Test_score ELSE 0 END)) > 20 THEN 'High'

    WHEN (
      SUM(CASE WHEN t.Test_type = 'CBC' THEN t.Test_score ELSE 0 END) +
      SUM(CASE WHEN t.Test_type = 'RBC' THEN t.Test_score ELSE 0 END) +
      SUM(CASE WHEN t.Test_type = 'LBH' THEN t.Test_score ELSE 0 END)) BETWEEN 16 AND
20 THEN 'Medium' ELSE 'Low' END AS Risk_Level FROM insurance i JOIN test t ON
i.Patient_id = t.Patient_id GROUP BY i.Patient_id, i.Insurance_Name;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import when, col, sum as spark_sum

# Initialize Spark
spark = SparkSession.builder.getOrCreate()

# Sample data for insurance
insurance_data = [
    (1, 1001, 'India Insurance'),
    (2, 1002, 'ICICI Lombard'),
    (3, 1001, 'India Insurance'),
    (4, 1003, 'Star Health'),
    (5, 1003, 'Star Health')]

insurance_columns = ['Patient_id', 'Insurance_id', 'Insurance_Name']
insurance_df = spark.createDataFrame(insurance_data, insurance_columns)

# Sample data for test
test_data = [
    (1, 'CBC', 7), (1, 'RBC', 6), (1, 'LBH', 6),
    (2, 'CBC', 7), (2, 'RBC', 8), (2, 'LBH', 8),
    (3, 'CBC', 5), (3, 'RBC', 4), (3, 'LBH', 4),
    (4, 'CBC', 4), (4, 'RBC', 6), (4, 'LBH', 6),
    (5, 'CBC', 5), (5, 'RBC', 6), (5, 'LBH', 7)]
```

```
test_columns = ['Patient_id', 'Test_type', 'Test_score']
test_df = spark.createDataFrame(test_data, test_columns)

# Pivot test scores to columns
pivot_df = test_df.groupBy("Patient_id").pivot("Test_type").sum("Test_score")

# Join with insurance data
joined_df = insurance_df.join(pivot_df, on="Patient_id")

# Fill nulls in case any test type is missing
filled_df = joined_df.fillna(0, subset=["CBC", "RBC", "LBH"])

# Calculate total score and risk level
result_df = filled_df.withColumn(
    "Total_Score", col("CBC") + col("RBC") + col("LBH")
).withColumn(
    "Risk_Level",
    when(col("Total_Score") > 20, "High")
    .when((col("Total_Score") >= 16) & (col("Total_Score") <= 20), "Medium")
    .otherwise("Low"))

# Show the result
result_df.select("Patient_id", "Insurance_Name", "Total_Score", "Risk_Level").show()
```

Write a SQL query to fix the names so that only first character is Upper-Case and the rest are Lower-Case. Return the result table ordered by user_id.
*/

```
-- Table
CREATE TABLE Users (
 user_id INT PRIMARY KEY,
 name NVARCHAR(50));

-- Insert the data
INSERT INTO Users (user_id, name)
VALUES (1, 'aLice'),(2, 'bOB');

%sql
SELECT user_id, UPPER(SUBSTR(name, 1, 1)) || LOWER(SUBSTR(name, 2)) AS name
FROM Users ORDER BY user_id;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, upper, lower, substring, concat

# Create SparkSession
spark = SparkSession.builder.getOrCreate()

# Sample data
data = [(1, 'aLice'), (2, 'bOB')]
columns = ['user_id', 'name']

# Create DataFrame
```

```
df = spark.createDataFrame(data, columns)

# Fix name formatting
formatted_df = df.select(col("user_id"), concat(upper(substring("name", 1, 1)),
lower(substring("name", 2, 100))).alias("name")).orderBy("user_id")

# Show result
formatted_df.show()
```

Assume you're given a table containing information about Wayfair user transactions for different products. Write a query to calculate the year-on-year growth rate for the total spend of each product, grouping the results by product ID.

The output should include the year in ascending order, product ID, current year's spend, previous year's spend and year-on-year growth percentage, rounded to 2 decimal places.
*/

```
-- Table
CREATE TABLE user_Transactions (transaction_id INT PRIMARY KEY,product_id INT,spend
DECIMAL(10,2),transaction_date DATETIM);

-- Insert the data
INSERT INTO user_Transactions (transaction_id, product_id, spend, transaction_date)
VALUES
(1341, 123424, 1500.60, '2019-12-31 12:00:00'),(1423, 123424, 1000.20, '2020-12-31
12:00:00'),(1623, 123424, 1246.44, '2021-12-31 12:00:00'),(1322, 123424, 2145.32,
'2022-12-31 12:00:00'),(1344, 234412, 1800.00, '2019-12-31 12:00:00'),(1435, 234412,
1234.00, '2020-12-31 12:00:00'),(4325, 234412, 889.50, '2021-12-31 12:00:00'),(5233,
234412, 2900.00, '2022-12-31 12:00:00'),
(2134, 543623, 6450.00, '2019-12-31 12:00:00'),(1234, 543623, 5348.12, '2020-12-31
12:00:00'),(2423, 543623, 2345.00, '2021-12-31 12:00:00'),
(1245, 543623, 5680.00, '2022-12-31 12:00:00');
```



```
%sql
SELECT
   product_id,
   year,
   current_year_spend,
   prev_year_spend, ROUND(( (current_year_spend - prev_year_spend) /
prev_year_spend ) * 100, 2) AS yoy_growth_pct
FROM (
   SELECT
      product_id,
      EXTRACT(YEAR FROM transaction_date) AS year,
      SUM(spend) AS current_year_spend,
      LAG(SUM(spend)) OVER (
         PARTITION BY product_id
         ORDER BY EXTRACT(YEAR FROM transaction_date)
      ) AS prev_year_spend
   FROM user_transactions
   GROUP BY product_id, EXTRACT(YEAR FROM transaction_date)) ORDER BY
product_id, year;
```

```
%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import year, sum, round, col, lag
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.getOrCreate()

# Sample data
data = [
    (1341, 123424, 1500.60, '2019-12-31 12:00:00'),
    (1423, 123424, 1000.20, '2020-12-31 12:00:00'),
    (1623, 123424, 1246.44, '2021-12-31 12:00:00'),
    (1322, 123424, 2145.32, '2022-12-31 12:00:00'),
    (1344, 234412, 1800.00, '2019-12-31 12:00:00'),
    (1435, 234412, 1234.00, '2020-12-31 12:00:00'),
    (4325, 234412, 889.50, '2021-12-31 12:00:00'),
    (5233, 234412, 2900.00, '2022-12-31 12:00:00'),
    (2134, 543623, 6450.00, '2019-12-31 12:00:00'),
    (1234, 543623, 5348.12, '2020-12-31 12:00:00'),
    (2423, 543623, 2345.00, '2021-12-31 12:00:00'),
    (1245, 543623, 5680.00, '2022-12-31 12:00:00')]

columns = ["transaction_id", "product_id", "spend", "transaction_date"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Extract year and compute total spend per product per year
df_with_year = df.withColumn("year", year("transaction_date"))

total_spend = df_with_year.groupBy("product_id", "year") \
    .agg(sum("spend").alias("current_year_spend"))

# Define window partitioned by product and ordered by year
window_spec = Window.partitionBy("product_id").orderBy("year")

# Add previous year's spend
result = total_spend.withColumn("prev_year_spend",
lag("current_year_spend").over(window_spec))

# Calculate YoY growth
result = result.withColumn(
    "yoy_growth_pct",
    round(((col("current_year_spend") - col("prev_year_spend")) /
col("prev_year_spend")) * 100, 2))

# Show final output
result.orderBy("product_id", "year").show()
```

-- Q) Write an SQL and pyspark Query to find out call duration (in minute) for every call.

```
-- Table - 1
CREATE TABLE call_start(
```

```
ph_no varchar(10),
start_time DATETIME);

-- Insert the data
INSERT INTO call_start VALUES
('contact_1','2024-05-01 10:20:00'),
('contact_1','2024-05-01 16:25:00'),
('contact_2','2024-05-01 12:30:00'),
('contact_3','2024-05-02 10:00:00'),
('contact_3','2024-05-02 12:30:00'),
('contact_3','2024-05-03 09:20:00');

-- Table - 2
 CREATE TABLE call_end(
ph_no VARCHAR(10),
end_time DATETIME);

 -- Insert the data
 INSERT INTO call_end VALUES
('contact_1','2024-05-01 10:45:00'),
('contact_1','2024-05-01 17:05:00'),
('contact_2','2024-05-01 12:55:00'),
('contact_3','2024-05-02 10:20:00'),
('contact_3','2024-05-02 12:50:00'),
('contact_3','2024-05-03 09:40:00'

%sql
WITH start_ranked AS (
SELECT ph_no, start_time,ROW_NUMBER() OVER (PARTITION BY ph_no ORDER BY
start_time) AS rn FROM call_start),
end_ranked AS (
SELECT ph_no, end_time, ROW_NUMBER() OVER (PARTITION BY ph_no ORDER BY
end_time) AS rn FROM call_end)
SELECT
  s.ph_no,
  s.start_time,
  e.end_time,
  ROUND((e.end_time - s.start_time) * 24 * 60, 2) AS duration_minutes FROM
start_ranked s JOIN end_ranked e ON s.ph_no = e.ph_no AND s.rn = e.rn ORDER BY
s.ph_no, s.start_time;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import row_number, col, unix_timestamp, round as
spark_round
from pyspark.sql.window import Window

spark = SparkSession.builder.getOrCreate()

# Sample data
start_data = [
    ("contact_1", "2024-05-01 10:20:00"),
    ("contact_1", "2024-05-01 16:25:00"),
    ("contact_2", "2024-05-01 12:30:00"),
    ("contact_3", "2024-05-02 10:00:00"),
```

```
    ("contact_3", "2024-05-02 12:30:00"),
    ("contact_3", "2024-05-03 09:20:00"),
]

end_data = [
    ("contact_1", "2024-05-01 10:45:00"),
    ("contact_1", "2024-05-01 17:05:00"),
    ("contact_2", "2024-05-01 12:55:00"),
    ("contact_3", "2024-05-02 10:20:00"),
    ("contact_3", "2024-05-02 12:50:00"),
    ("contact_3", "2024-05-03 09:40:00"),
]

# Create DataFrames
df_start = spark.createDataFrame(start_data, ["ph_no", "start_time"])
df_end = spark.createDataFrame(end_data, ["ph_no", "end_time"])

# Convert to timestamps
df_start = df_start.withColumn("start_time", col("start_time").cast("timestamp"))
df_end = df_end.withColumn("end_time", col("end_time").cast("timestamp"))

# Add row numbers to ensure correct matching
windowSpec = Window.partitionBy("ph_no").orderBy("start_time")
df_start = df_start.withColumn("rn", row_number().over(windowSpec))

windowSpecEnd = Window.partitionBy("ph_no").orderBy("end_time")
df_end = df_end.withColumn("rn", row_number().over(windowSpecEnd))

# Join and calculate duration
df_result = df_start.join(df_end, on=["ph_no", "rn"]) \
.withColumn("duration_minutes", spark_round((unix_timestamp("end_time") -
unix_timestamp("start_time")) / 60, 2)).select("ph_no", "start_time", "end_time",
"duration_minutes") \.orderBy("ph_no", "start_time")

df_result.show(truncate=False)
```

A company wants to divide the employees into teams such that all the members on each team have the same salary. The teams should follow these criteria:

Each team should consist of at least two employees.
All the employees on a team should have the same salary.
All the employees of the same salary should be assigned to the same team. If the salary of an employee is unique, we do not assign this employee to any team. A team's ID is assigned based on the rank of the team's salary relative to the other teams' salaries, where the team with the lowest salary has team_id = 1.

Return the result table ordered by team_id in ascending order. In case of a tie, order it by employee_id in ascending order.

```
-- Table
CREATE TABLE Employees_ (
 employee_id INT PRIMARY KEY,
 name VARCHAR(50),
 salary INT)
```

-- **Insert the data**

```
INSERT INTO Employees_ (employee_id, name, salary) VALUES
(2, 'Meir', 3000),(3, 'Michael', 3000),
(7, 'Addilyn', 7400),(8, 'Juan', 6100),
(9, 'Kannon', 7400);

%sql
WITH salary_groups AS (
SELECT salary FROM Employees_ GROUP BY salary HAVING COUNT(*) >= 2),
ranked_teams AS (
SELECT salary, DENSE_RANK() OVER (ORDER BY salary) AS team_id FROM
salary_groups),
final_teams AS (
SELECT e.employee_id, e.name, e.salary, r.team_id FROM Employees_ e JOIN
ranked_teams r ON e.salary = r.salary)
SELECT * FROM final_teams ORDER BY team_id, employee_id;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, dense_rank
from pyspark.sql.window import Window

# Initialize SparkSession
spark = SparkSession.builder.getOrCreate()

# Sample data
data = [
    (2, 'Meir', 3000),
    (3, 'Michael', 3000),
    (7, 'Addilyn', 7400),
    (8, 'Juan', 6100),
    (9, 'Kannon', 7400)]

# Create DataFrame
df = spark.createDataFrame(data, ["employee_id", "name", "salary"])

# Step 1: Filter salaries that appear more than once
salary_counts = df.groupBy("salary").agg(count("*").alias("cnt")).filter(col("cnt") >= 2)

# Step 2: Assign team_id using dense_rank
window_spec = Window.orderBy("salary")
salary_ranked = salary_counts.withColumn("team_id",
dense_rank().over(window_spec)).select("salary", "team_id")

# Step 3: Join back with original data
final_df = df.join(salary_ranked, on="salary", how="inner")

# Step 4: Sort by team_id and employee_id
result = final_df.select("employee_id", "name", "salary", "team_id").orderBy("team_id",
"employee_id")

# Show result
result.show()
```

We need to Find department wise minimum salary emp_name and maximum salary emp_name .

📊Here's the table structure and sample data:-

```
CREATE TABLE emps_tbl (
emp_name VARCHAR(50),
dept_id INT, salary INT);

INSERT INTO emps_tbl
VALUES ('Siva', 1, 30000), ('Ravi', 2, 40000),
('Prasad', 1, 50000), ('Sai', 2, 20000),
('Anna', 2, 10000);

%sql
SELECT dept_id, MAX(CASE WHEN salary = min_salary THEN emp_name END) AS
min_salary_emp, MAX(CASE WHEN salary = max_salary THEN emp_name END) AS
max_salary_emp FROM (
SELECT e.*,  MIN(salary) OVER (PARTITION BY dept_id) AS min_salary, MAX(salary) OVER
(PARTITION BY dept_id) AS max_salary FROM emps_tbl e) sub GROUP BY dept_id;

%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import min, max, col, first
from pyspark.sql.window import Window

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Sample data
data = [
   ('Siva', 1, 30000),
   ('Ravi', 2, 40000),
   ('Prasad', 1, 50000),
   ('Sai', 2, 20000),
   ('Anna', 2, 10000)]

# Create DataFrame
df = spark.createDataFrame(data, ["emp_name", "dept_id", "salary"])

# Window for min and max salary in each dept
win = Window.partitionBy("dept_id")

# Add min and max salary columns
df = df.withColumn("min_salary", min("salary").over(win)).withColumn("max_salary",
max("salary").over(win))

# Filter for min salary employees
min_df = df.filter(col("salary") == col("min_salary")).select("dept_id",
col("emp_name").alias("min_salary_emp"))

# Filter for max salary employees
max_df = df.filter(col("salary") == col("max_salary")).select("dept_id",
col("emp_name").alias("max_salary_emp"))

# Join min and max
```

```
result = min_df.join(max_df, on="dept_id", how="inner").distinct()

# Show result
result.show()
```