Operation Analytics and Investigating Metric Spike

Advanced SQL

Project Description:

Operational Analytics is a process involving analysis of a company's end-to-end operations. This analysis helps identify areas for improvement within the company.
One of the key aspects of Operational Analytics is investigating metric spikes. It involves understanding and explaining sudden changes in key metrics, like sudden changes in daily user engagement or product sales.
We have derived insights from the provided data and answered the questions that have been presented. We make use of advanced SQL skills to analyze the given data and provide insights to help improve the company's operations and understand sudden spikes and lows in key metrics.
This will help provide valuable insights to developers and investors that can help the business grow.

Approach:

We start by creating our database in MySQL.
Next we run a command to use the said database that has been created.
The data is stored in a tabular format.
The problem statement is divided as Case Study 1 and Case Study 2.
Case Study 1 includes a dataset called job_data.
We create and use a database named job_data.
Then we make use of Table Data Import Wizard to import the csv file on our MySQL Workbench.
Case Study 2 has three datasets as users, events and email_events.
For Case Study 2, datasets are loaded in the form of tables in a database called metric_square using load data infile query.
Various Operational Analysis queries and Metric Spike queries have been used to retrieve information from the database according to our requirement.
We use various DDL and DML queries like SELECT, FROM, GROUP BY, ORDER BY, etc to retrieve and show information from the database.

Tech-Stack Used:

Intel Core i5
Windows 10
MySql 8.0 CE Workbench
Local host MySQ80

Insights:

For Case Study 1:

Creating a database in MySQL:

CREATE DATABASE job_data;
Command has been used to create a database named job_data.

USE job_data;
USE database command is run to select the database that we work on.

Loading Dataset in table job_data:

In the **schemas** column, under job_data database, right click on **Table Data Import Wizard**. Next, choose the required file, choose required data types for each column and import.

Result:

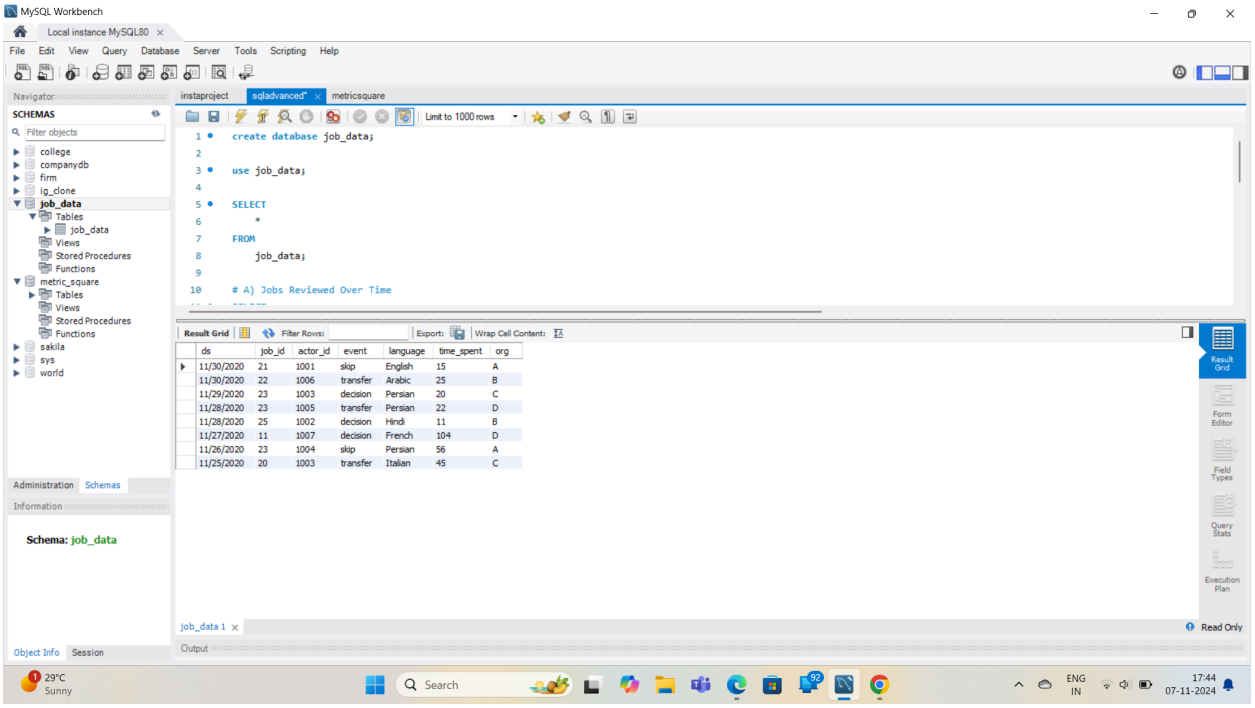
In order to visualize the data in tables we use query

SELECT * FROM tablename;

To visualize data from users table we use command

select * from job_data;

we get output as



A) Jobs Reviewed Over Time

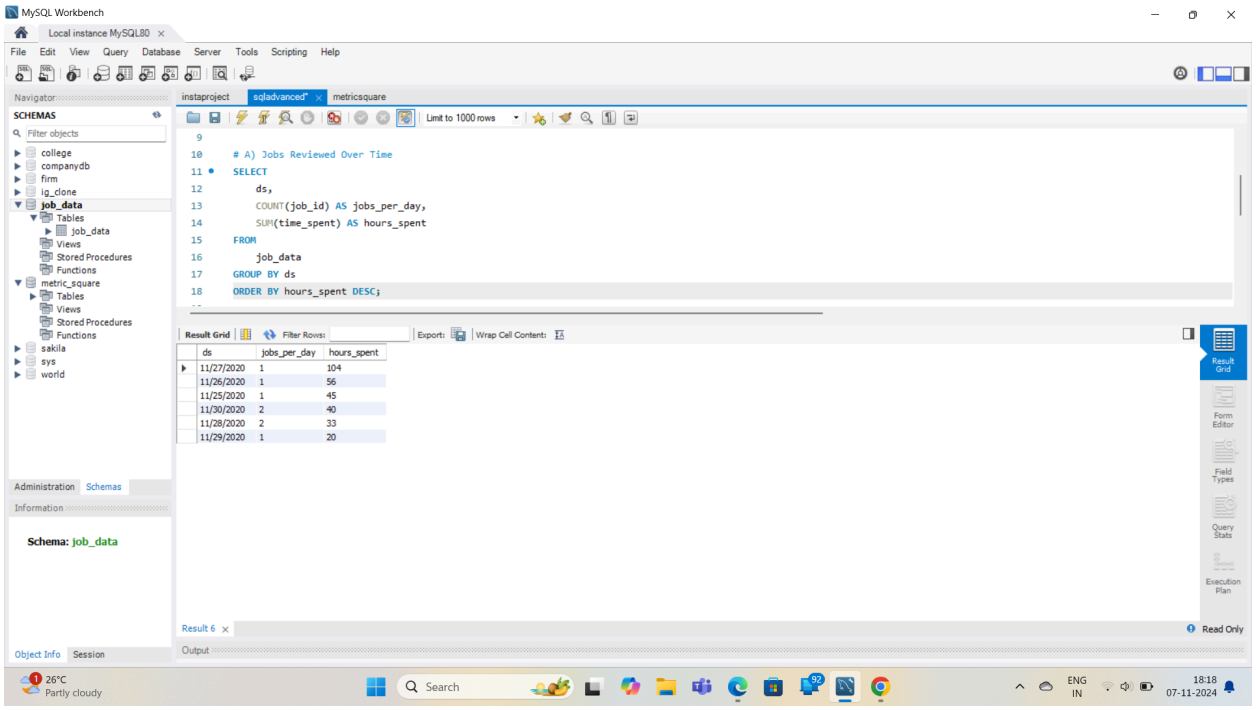
Objective: Calculate the number of jobs reviewed per hour for each day in November 2020.
Your Task: Write an SQL query to calculate the number of jobs reviewed per hour for each day in November 2020.

Query:

```
SELECT
    ds,
    COUNT(job_id) AS jobs_per_day,
    SUM(time_spent) AS hours_spent
FROM
    job_data
GROUP BY ds
ORDER BY hours_spent DESC;
```

Explanation:
We use **SELECT** to show column **ds** as it is, **COUNT(job_id)** gives the total number of entries in the job_id column and **AS** is used to show this in a column **jobs_per_day**.
SUM(time_spent) gives the total number of entries in the time_spent column and **AS** is used to show this in a column **hours_spent FROM** job_data table.
GROUP BY ds is used to group the values by ds column.
DESC is used to give descending order of values in hours_spent column using **ORDER BY.**

Result:



Conclusion:
On 11/27/2020, 1 job took 104 hours. On 11/30/2020, 2 jobs took 40 hours and similar.

B) **Throughput Analysis:**

Objective: Calculate the 7-day rolling average of throughput (number of events per second).

Your Task: Write an SQL query to calculate the 7-day rolling average of throughput. Additionally, explain whether you prefer using the daily metric or the 7-day rolling average for throughput, and why.

Query:

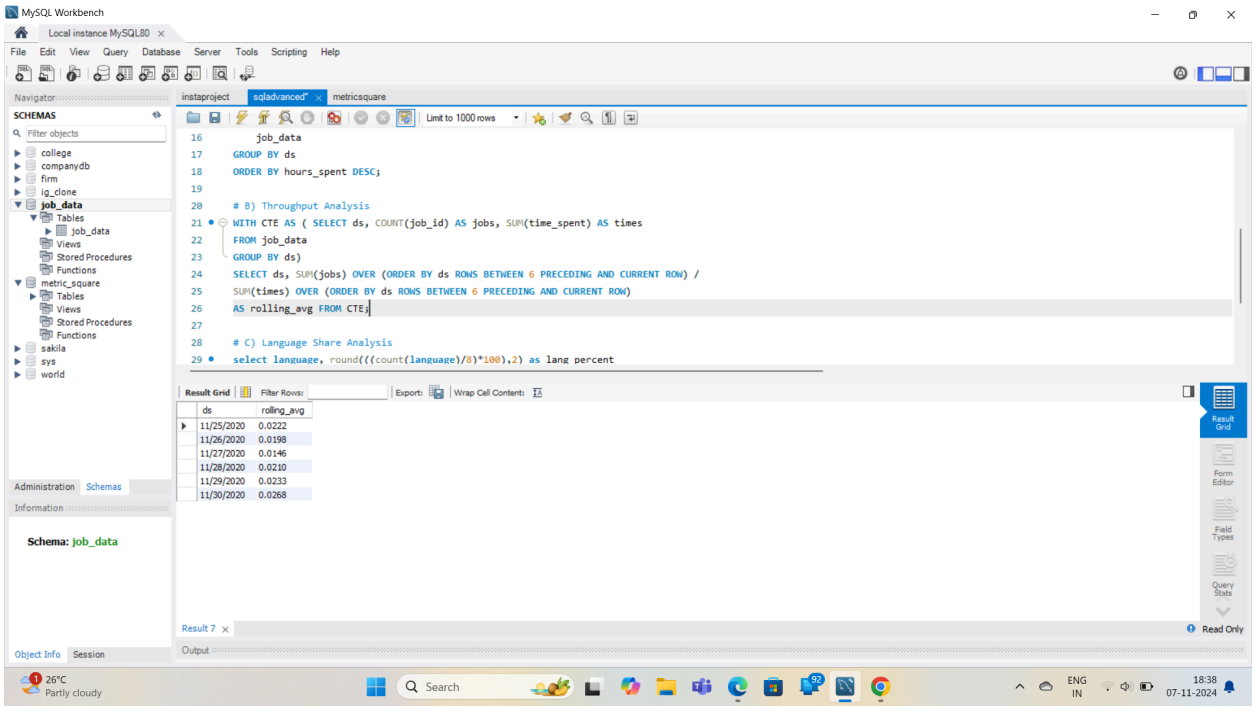
```
WITH CTE AS ( SELECT ds, COUNT(job_id) AS jobs, SUM(time_spent) AS times
FROM job_data
GROUP BY ds)
SELECT ds, SUM(jobs) OVER (ORDER BY ds ROWS BETWEEN 6 PRECEDING AND
CURRENT ROW) /
SUM(times) OVER (ORDER BY ds ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
AS rolling_avg FROM CTE;
```

Explanation:

CTE is used as Common Table Expression.
The **OVER** clause constructs a window that includes all the records returned by the query.
ROWS BETWEEN is used to calculate the running measure / aggregation in Analytical Functions.

CURRENT ROW: Range starts or ends at CURRENT ROW.

Result:



Conclusion:

Preference of daily metric or 7 day rolling average depends on what is required.

Daily metric is preferred on small and stable data, where insights from a particular day are required to make real time decisions.

7 day rolling average is more useful for large and noisy datasets, where we require a smooth outcome to highlight longer and broader trends.

C) Language Share Analysis:

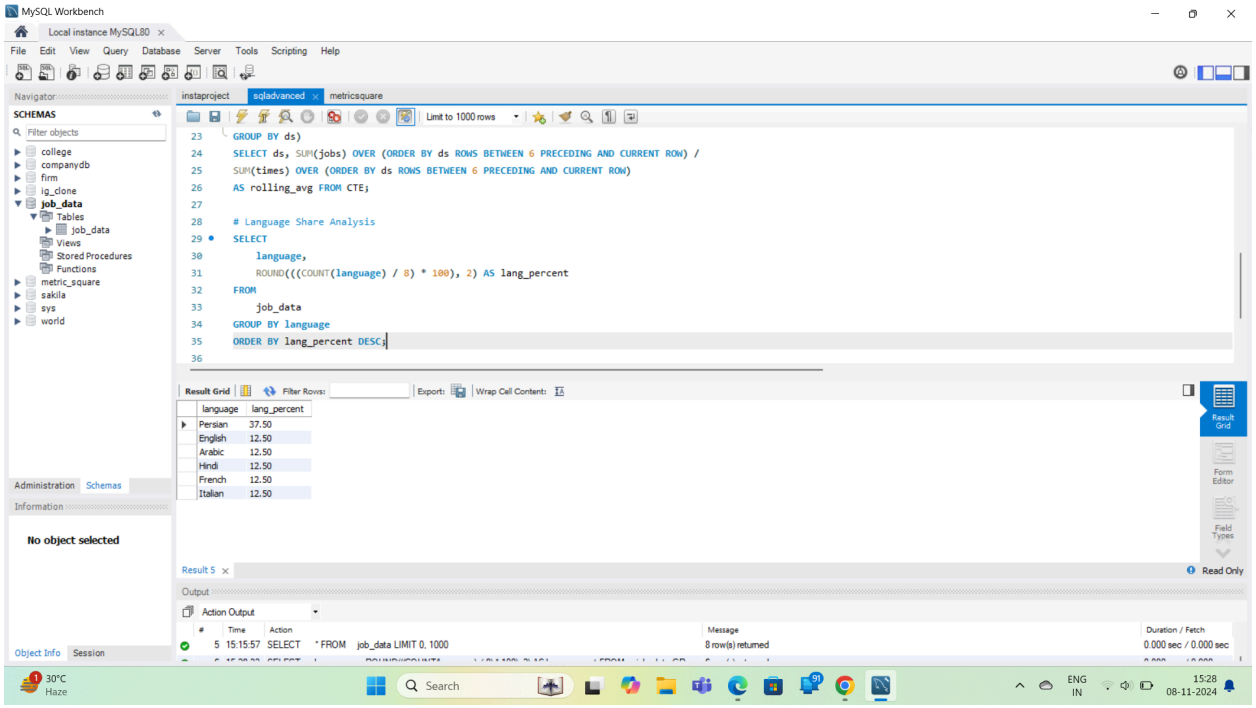
Objective: Calculate the percentage share of each language in the last 30 days.
Your Task: Write an SQL query to calculate the percentage share of each language over the last 30 days.

Query:

```
SELECT
    language,
    ROUND(((COUNT(language) / 8) * 100), 2) AS lang_percent
FROM
    job_data
GROUP BY language
ORDER BY lang_percent DESC;
```

Explanation:
The **ROUND** function rounds a number to a specified number of decimal places, which we have defined as **2**.
The expression **COUNT(language) / 8) * 100** applies the formula of percentage on the **language** column.

Result:



Conclusion:
Persian language has **37.50%** weightage, while the rest of the languages **Arabic, Hindi, French, Italian** each have a similar weightage of **12.50%**.

D) Duplicate Rows Detection

Objective: Identify duplicate rows in the data.
Your Task: Write an SQL query to display duplicate rows from the job_data table.

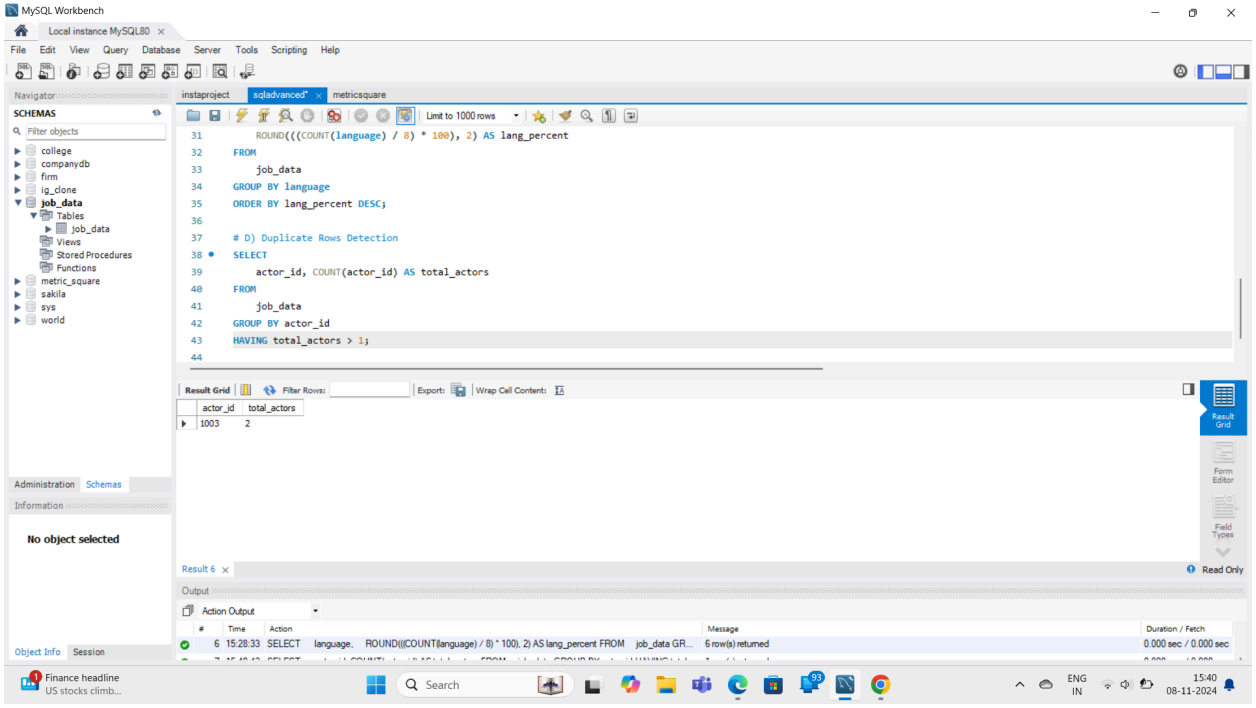
Query:

```
SELECT
    actor_id, COUNT(actor_id) AS total_actors
FROM
    job_data
GROUP BY actor_id
HAVING total_actors > 1;
```

Explanation:

Having clause is used to filter rows that satisfy specific conditions along with aggregate functions.

Query:



Conclusion:

Actor ID 1003 has two rows.

For Case Study 2:

Creating a database in MySQL:

CREATE DATABASE metric_square;
Command has been used to create a database named metric_square.

USE metric_square;
USE database command is run to select the database that we work on.

Creating a table in MySQL:

Creating table USERS in our database.

Query:

```
CREATE TABLE users (
    user_id INT,
    created_at VARCHAR(50),
    company_id INT,
    language VARCHAR(50),
    activated_at VARCHAR(100),
    state VARCHAR(50)
);
```

We use the CREATE command to create tables.
We define names and datatypes of the columns in the table.

Loading Data In The Table:

show variables like 'secure_file_priv';

Secure_file_priv gives the location where dataset.csv needs to be saved.

```
load data infile "C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/users.csv"
into table users
fields terminated by ','
enclosed by '"'
lines terminated by '\n'
```

ignore 1 rows;

Load data infile gives location of the file that needs to be loaded on MySQL Workbench.
Ignore 1 rows to ignore column headings.

Changing datatype in column created_at from string to datetime

```
alter table users add column temp_created_at datetime;
UPDATE users
SET
    temp_created_at = STR_TO_DATE(created_at, '%d-%m-%Y %H:%i');
alter table users drop column created_at;
alter table users change column temp_created_at created_at datetime;
```

ALTER and DROP commands are used.
We have changed the data type from the created_at column from STR_TO_DATE.

Similarly, we'll change, datatype of activated_at table.

Visualizing The Table Data:

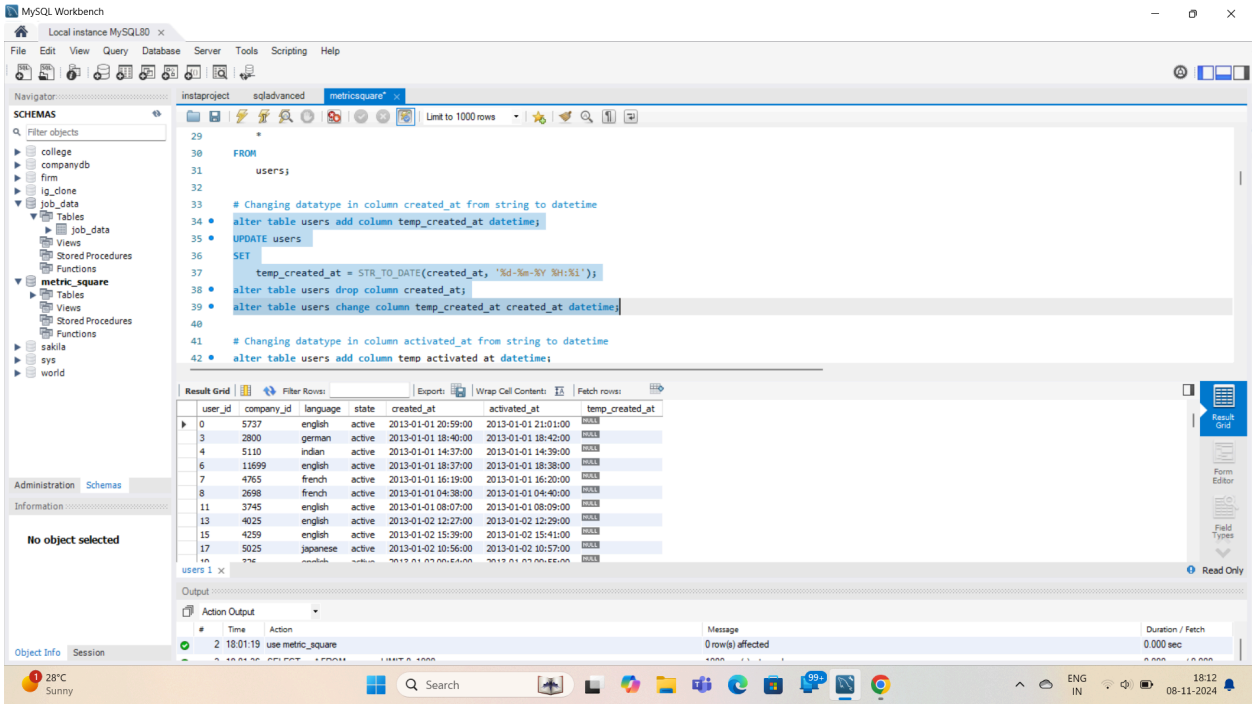
In order to visualize the data in tables we use query

SELECT * FROM tablename;

To visualize data from users table we use command

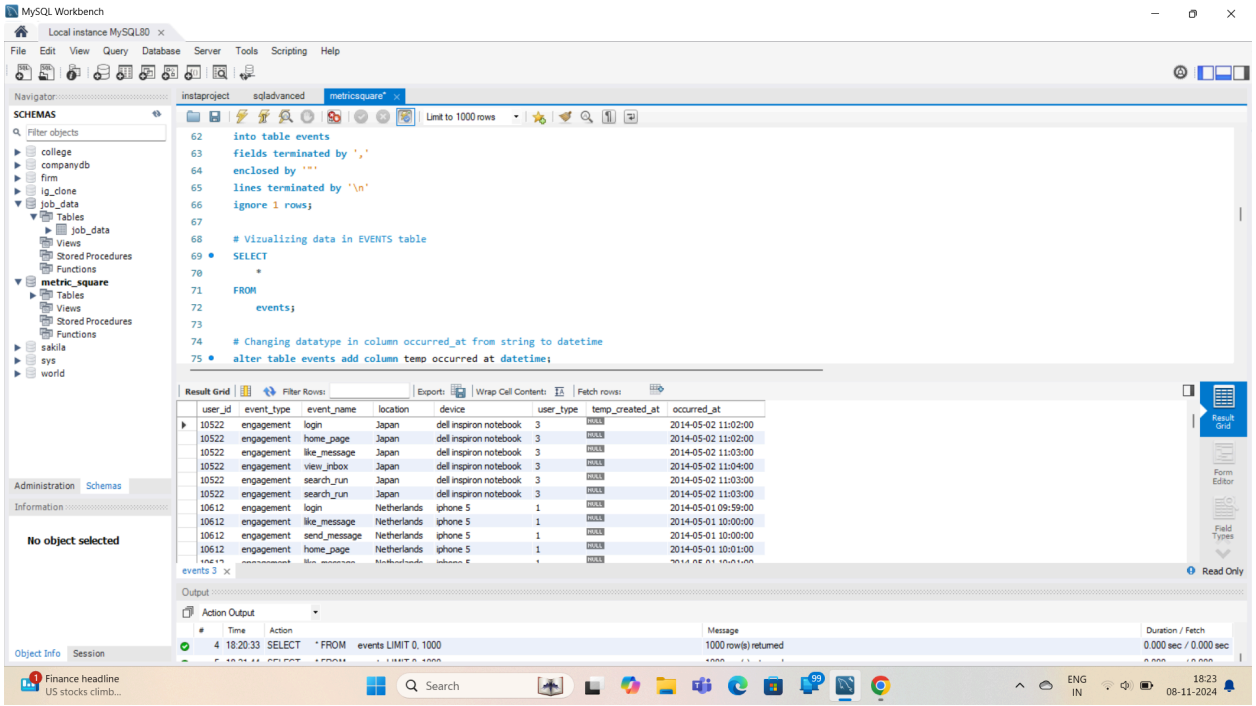
select * from users;

We get output as

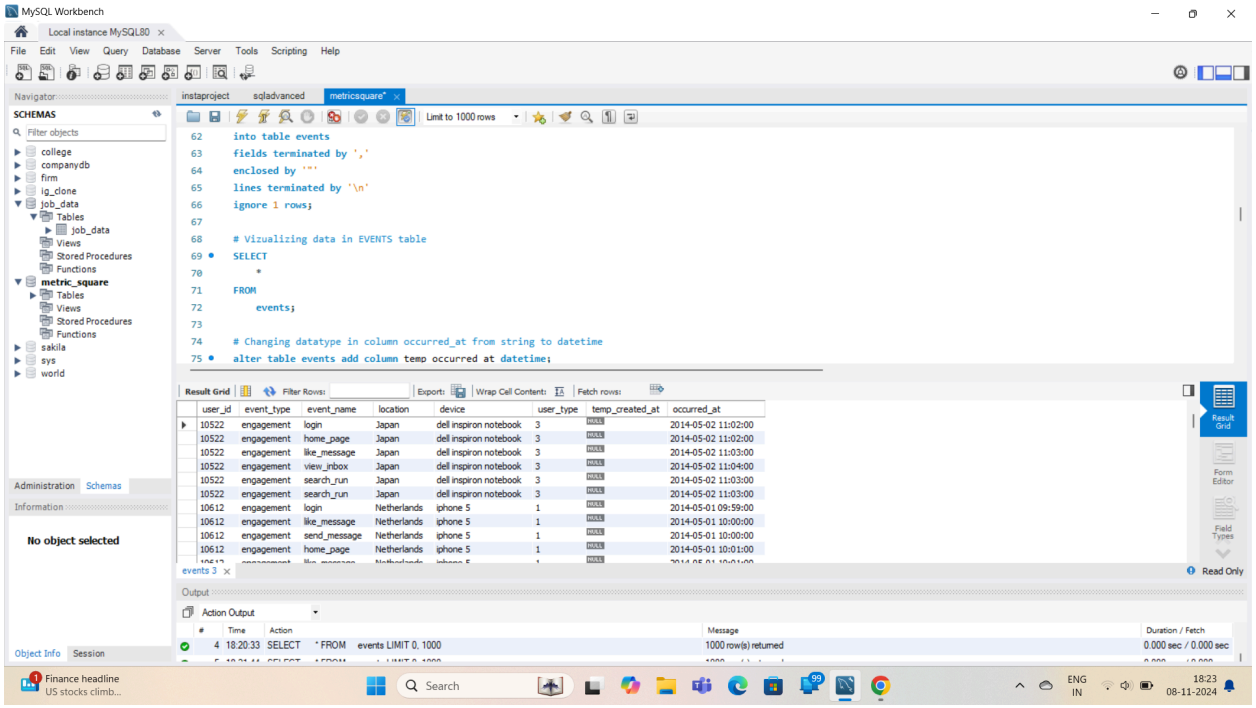


Similarly, we create tables EVENTS and EMAIL_EVENTS and change data types from string to datetime.

Events Table



Email_Events Table



A) Weekly User Engagement

Objective: Measure the activeness of users on a weekly basis.

Your Task: Write an SQL query to calculate the weekly user engagement.

Query:

```
SELECT
    EXTRACT(WEEK FROM occurred_at) AS no_of_weeks,
    COUNT(DISTINCT user_id) AS user_engagement
FROM
    events
WHERE
    event_type = 'Engagement'
GROUP BY no_of_weeks
ORDER BY no_of_weeks;
```

Explanation:

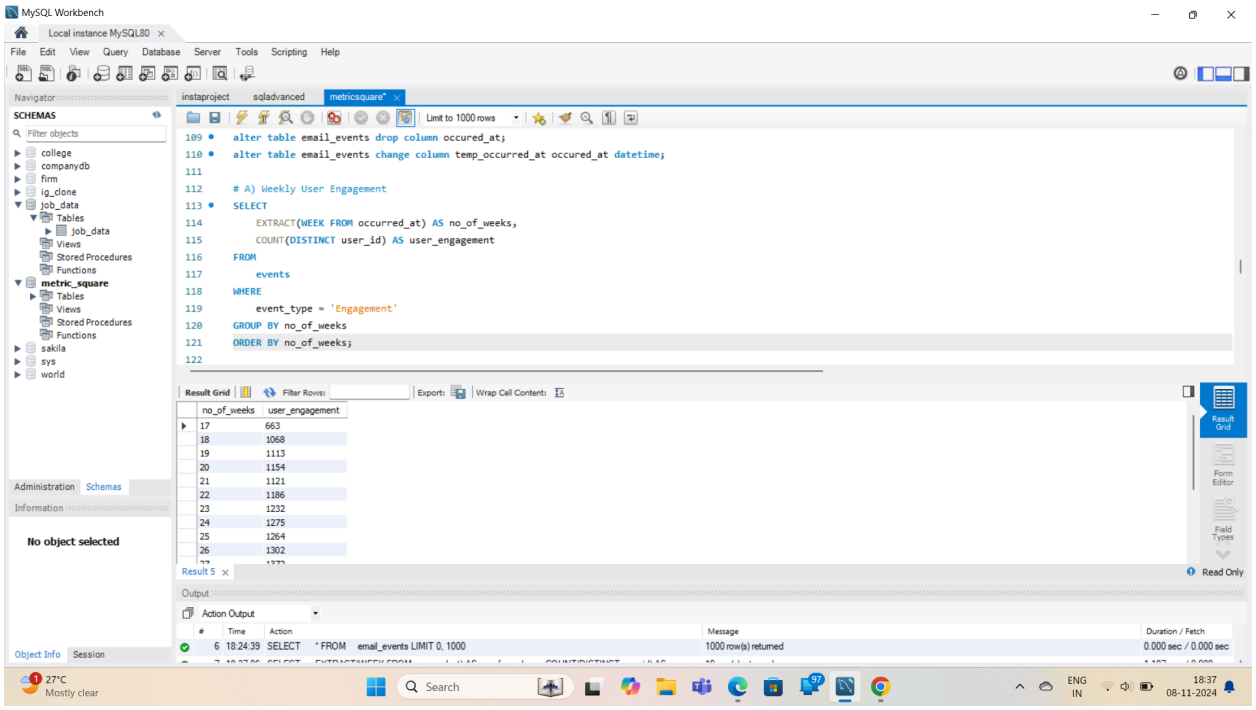
EXTRACT function extracts a part from a given date.

The **WEEK** function returns the week number for a given date

WHERE clause is used to filter records. It is used to extract only those records that fulfill a specified condition.

DISTINCT statement is used to return only **distinct** (different) values.

Result:



Conclusion:

Number of users that are engaged for how many weeks is mentioned in the result grid.

B) User Growth Analysis:

Objective: Analyze the growth of users over time for a product.
Your Task: Write an SQL query to calculate the user growth for the product.

Query:

```
SELECT year, month, active_users,
                sum(active_users) OVER(ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS user_growth,
```

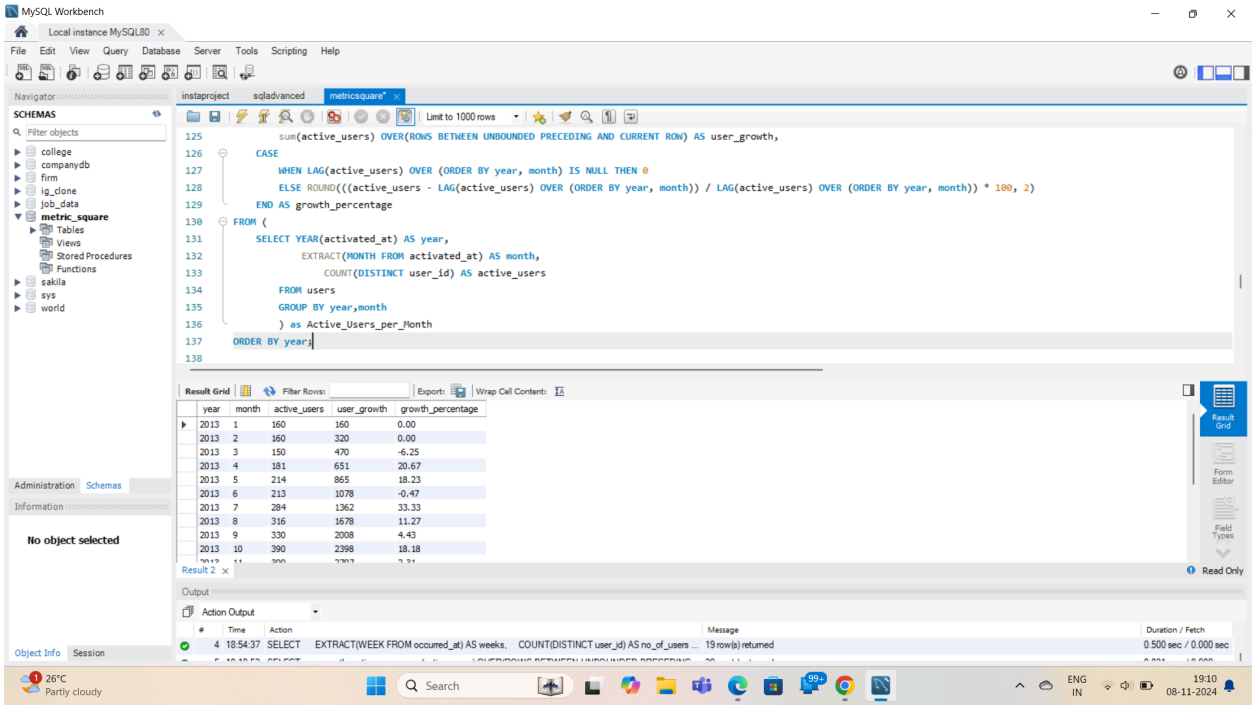
```
    CASE
        WHEN LAG(active_users) OVER (ORDER BY year, month) IS NULL THEN 0
        ELSE ROUND(((active_users - LAG(active_users) OVER (ORDER BY year, month)) /
LAG(active_users) OVER (ORDER BY year, month)) * 100, 2)
    END AS growth_percentage
FROM (
            SELECT YEAR(activated_at) AS year,
                                EXTRACT(MONTH FROM activated_at) AS month,
                                        COUNT(DISTINCT user_id) AS active_users
                    FROM users
                    GROUP BY year,month
                    ) as Active_Users_per_Month
ORDER BY year;
```

Explanation:

CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement).

LAG function is used to get value from the row that precedes the current row.

Result:



Conclusion:

The result grid shows columns for year, month, active_users, user_growth and growth_percentage.

C) Weekly Retention Analysis:

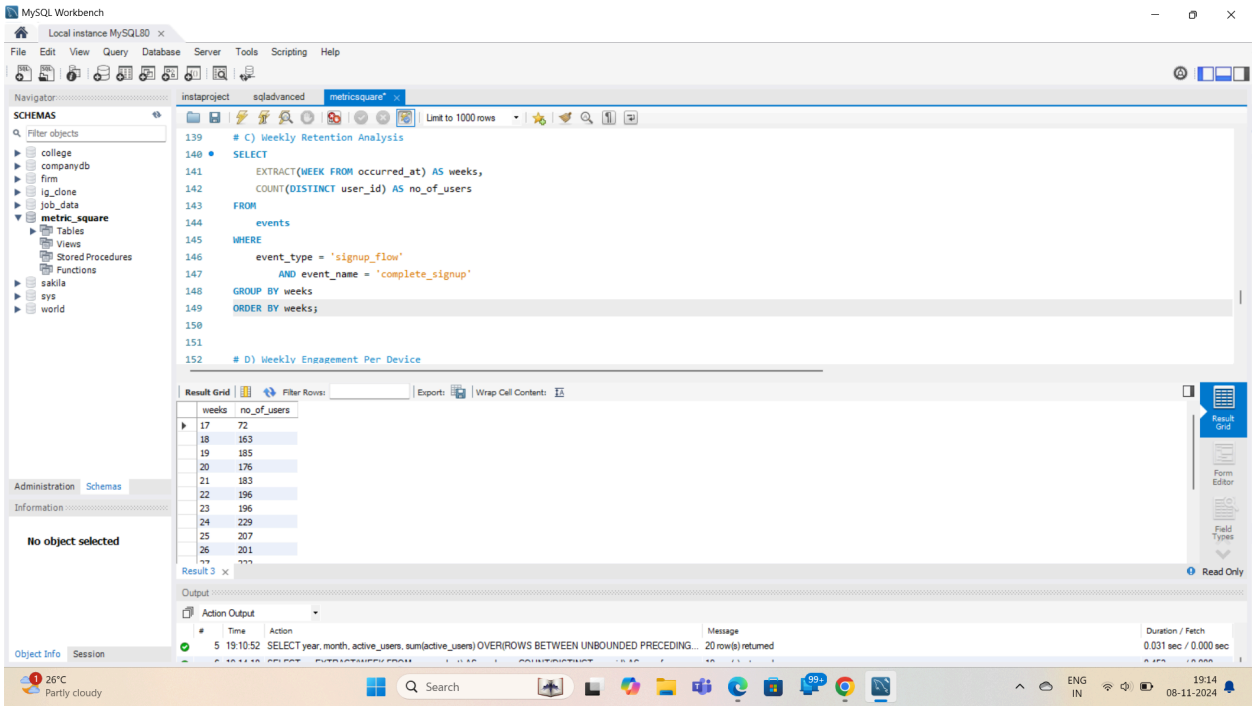
Objective: Analyze the retention of users on a weekly basis after signing up for a product.
Your Task: Write an SQL query to calculate the weekly retention of users based on their sign-up cohort.

Query:

```
SELECT
    EXTRACT(WEEK FROM occurred_at) AS weeks,
    COUNT(DISTINCT user_id) AS no_of_users
FROM
    events
WHERE
    event_type = 'signup_flow'
        AND event_name = 'complete_signup'
GROUP BY weeks
ORDER BY weeks;
```

Explanation:
event_type = 'signup_flow' AND event_name = 'complete_signup' are used as conditions in the query.

Result:



Conclusion:
Number of users for weeks is given in the result grid.

D) Weekly Engagement Per Device:

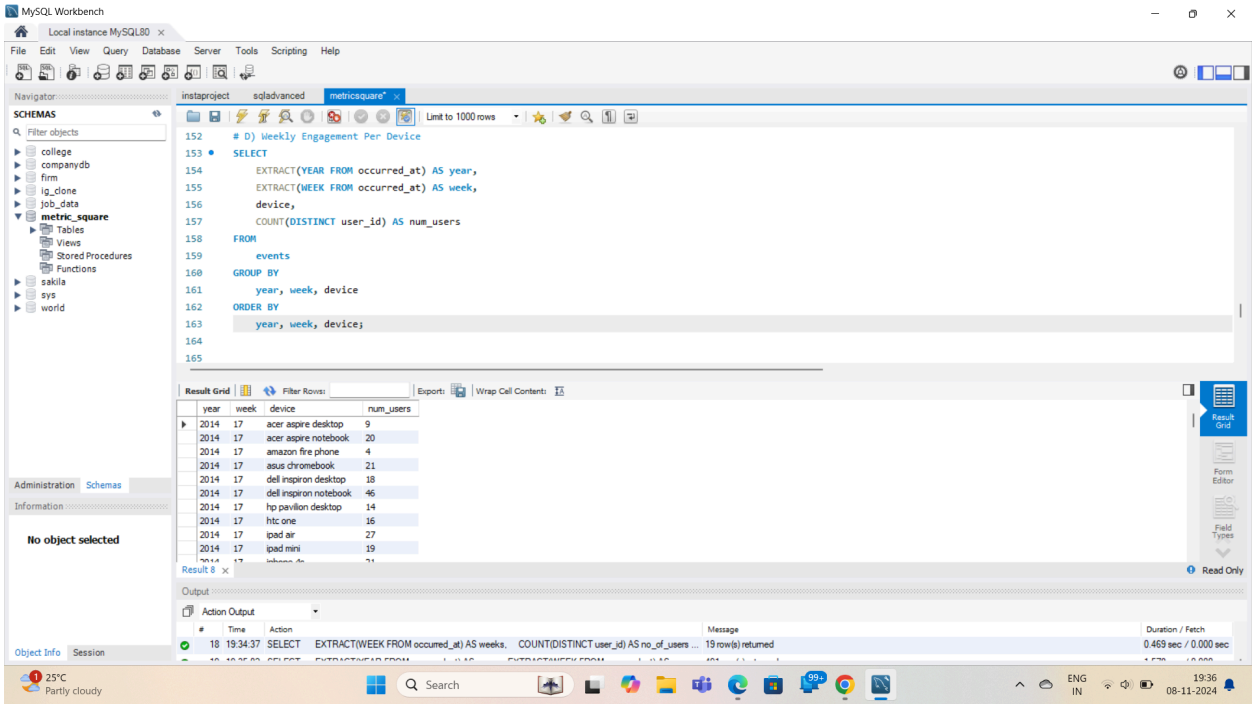
Objective: Measure the activeness of users on a weekly basis per device.

Your Task: Write an SQL query to calculate the weekly engagement per device.

Query:

```
SELECT
    EXTRACT(YEAR FROM occurred_at) AS year,
    EXTRACT(WEEK FROM occurred_at) AS week,
    device,
    COUNT(DISTINCT user_id) AS num_users
FROM
    events
GROUP BY
    year, week, device
ORDER BY
    year, week, device;
```

Result:



Conclusion:

Number of users engaging for number of weeks per year is given along with the names of devices that were used.

E) Email Engagement Analysis:

Objective: Analyze how users are engaging with the email service.
Your Task: Write an SQL query to calculate the email engagement metrics.

Query:

```
SELECT
    user_id,
    Emails_sent,
    Emails_opened,
    Emails_clicked,
    ROUND(SUM(Emails_opened) / SUM(Emails_sent), 2) * 100 AS Opening_rate,
    ROUND(SUM(Emails_clicked) / SUM(Emails_opened),
            2) * 100 AS Engagement_rate
FROM
    (SELECT
        user_id,
            SUM(CASE
```

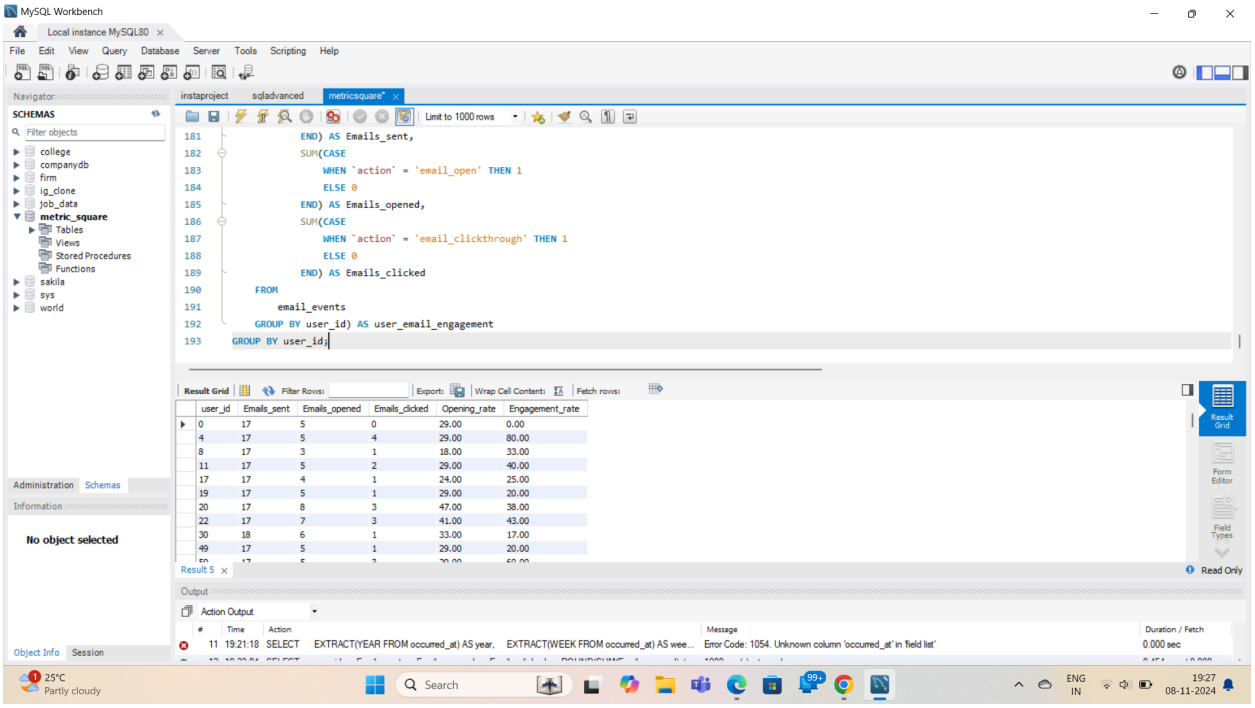
```
        WHEN `action` = 'sent_weekly_digest' THEN 1
        ELSE 0
    END) AS Emails_sent,
    SUM(CASE
        WHEN `action` = 'email_open' THEN 1
        ELSE 0
    END) AS Emails_opened,
    SUM(CASE
        WHEN `action` = 'email_clickthrough' THEN 1
        ELSE 0
    END) AS Emails_clicked
  FROM
    email_events
  GROUP BY user_id) AS user_email_engagement
GROUP BY user_id;
```

Explanation:

USER ENGAGEMENT
Clickthrough rate, or CTR, is the number of clicks per email delivered. If your emails include links this metric can help you gauge how effective that is.
Formula to calculate rate
(email clicks / emails delivered) x 100

WHEN - - ELSE is similar to the if else statement in coding languages.

Result:



Conclusion:

Number of emails that have been sent, opened, clicked, along with their opening rate and engagement rate are given.