

Operation Analytics and Investigating Metric Spike

PROJECT – 3

Charmy Raj

Project Description:

Operational Analytics is essential for analysing a company's whole operations. This research identifies company improvement areas. You'll help operations, support, and marketing teams get insights from their data as a Data Analyst. Investigating metric spikes is crucial to Operational Analytics. This requires recognising and explaining abrupt changes in critical indicators like daily user engagement or sales. Data Analysts must answer these questions regularly, therefore understanding how to evaluate metric spikes is vital.

Tech Stack used:

To facilitate data analysis and querying, MySQL was used as the underlying database management system. Thanks to SQL, complex queries could be written to retrieve the necessary data from the database with ease. For writing SQL queries and communicating with the MySQL backend, we used SQL Workbench. The platform's intuitive design made SQL query creation and execution much more efficient.

Insights:

I was supposed to pretend I worked for Microsoft as a Lead Data Analyst for this assignment. I was given several tables and information and tasked with analysing them to provide answers to questions raised by various teams around the firm. I hoped that by using my expert SQL abilities to the data, I could assist the company enhance its operations and better understand why some of its important metrics had suddenly changed.

Approach:

To begin, I made a database for the project, and then use the provided table structures and connections to make the appropriate tables. I used MySQL Workbench to import the.csv file.

Use Structured Query Language (SQL) to do the research and obtain solutions to the issues raised by the case studies. I would assume that it's important to know how to read a database table and what information each column represents before beginning a review.

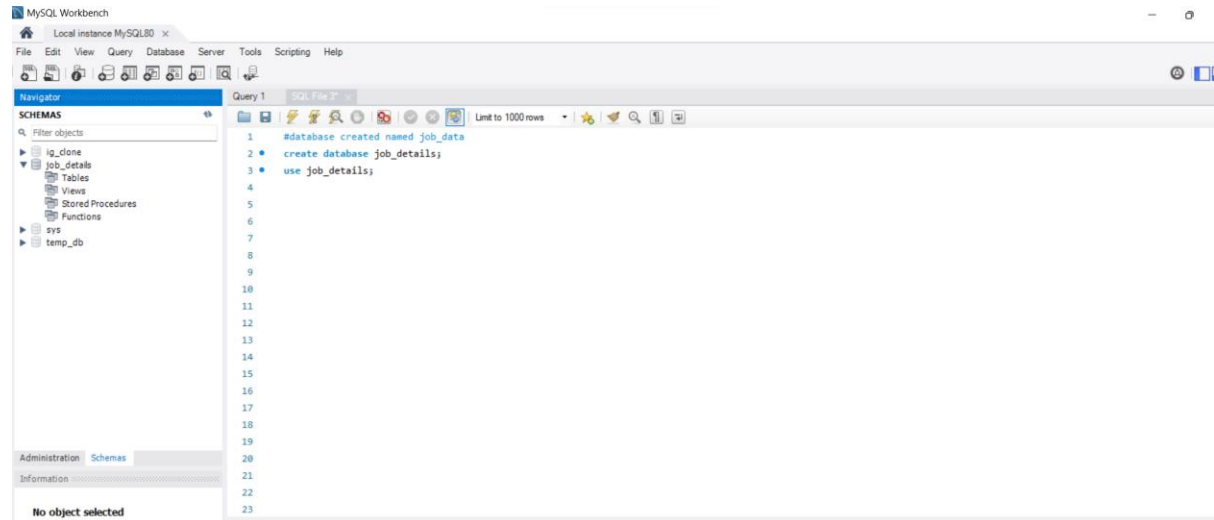
Table of Contents

Project Description:.....	1
Tech Stack used:.....	1
Insights:.....	1
Approach:.....	1
CASE STUDY 1.....	4
Creation of database.....	4
Importing the table data.....	4
Browsed the CSV file.....	5
Selected the destination of file.....	5
Successfully imported the data.....	6
Query 1: Over Time Jobs Reviewed.....	7
Output:.....	7
Code:.....	7
Explanation:.....	7
Query 2: Throughput Analysis.....	8
Output:.....	8
Code:.....	8
Explanation:.....	8
Preference:.....	9
Query 3: Language Share Analysis.....	10
Output:.....	10
Code:.....	10
Explanation:.....	10
Query 4: Duplicate Rows Detection.....	11
Output:.....	11
Code:.....	11
Explanation:.....	12
CASE STUDY 2.....	13
Importing The Data.....	13
All the data has been successfully imported.....	13
Query 1: Weekly User Engagement.....	14
Output:.....	14
Code:.....	14
Explanation:.....	14
Query 2: User Growth Analysis.....	15

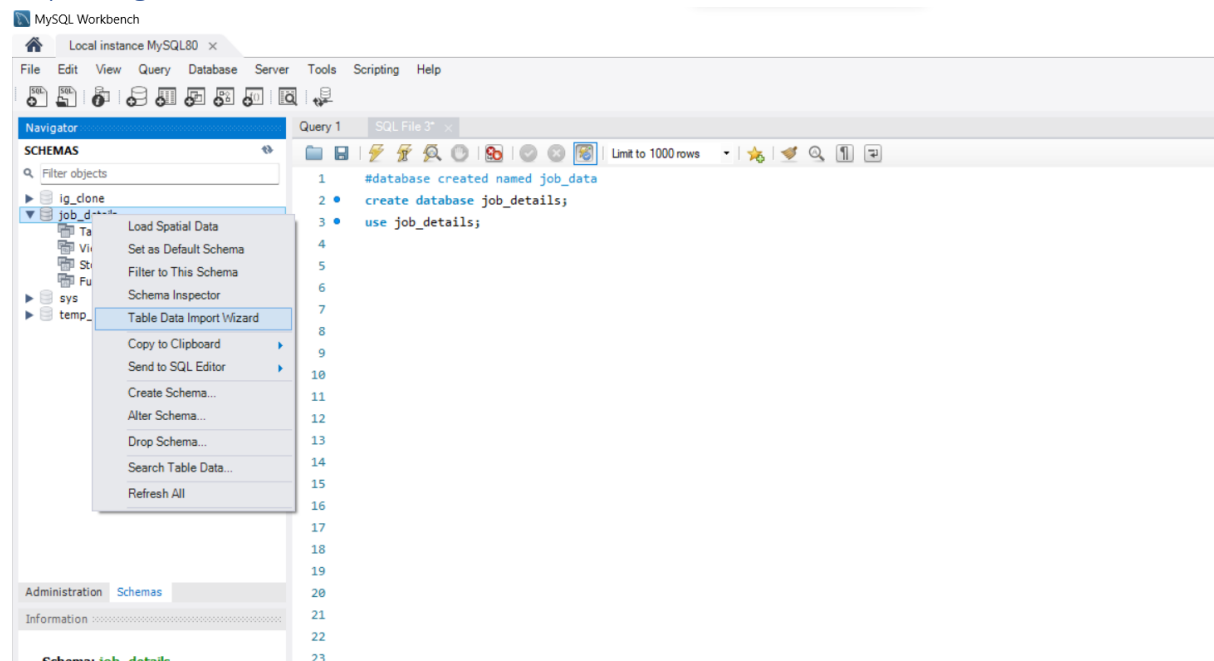
Output:.....	15
Code:	15
Explanation:	15
Query 3: Weekly Retention Analysis	16
Output:.....	16
Code:	16
Explanation:	16
Query 4: Weekly Engagement Per Device.....	17
Output:.....	17
Code:	17
Explanation:	17
Query 5: Email Engagement Analysis.....	18
Output:.....	18
Code:	18
Explanation:	18

CASE STUDY 1

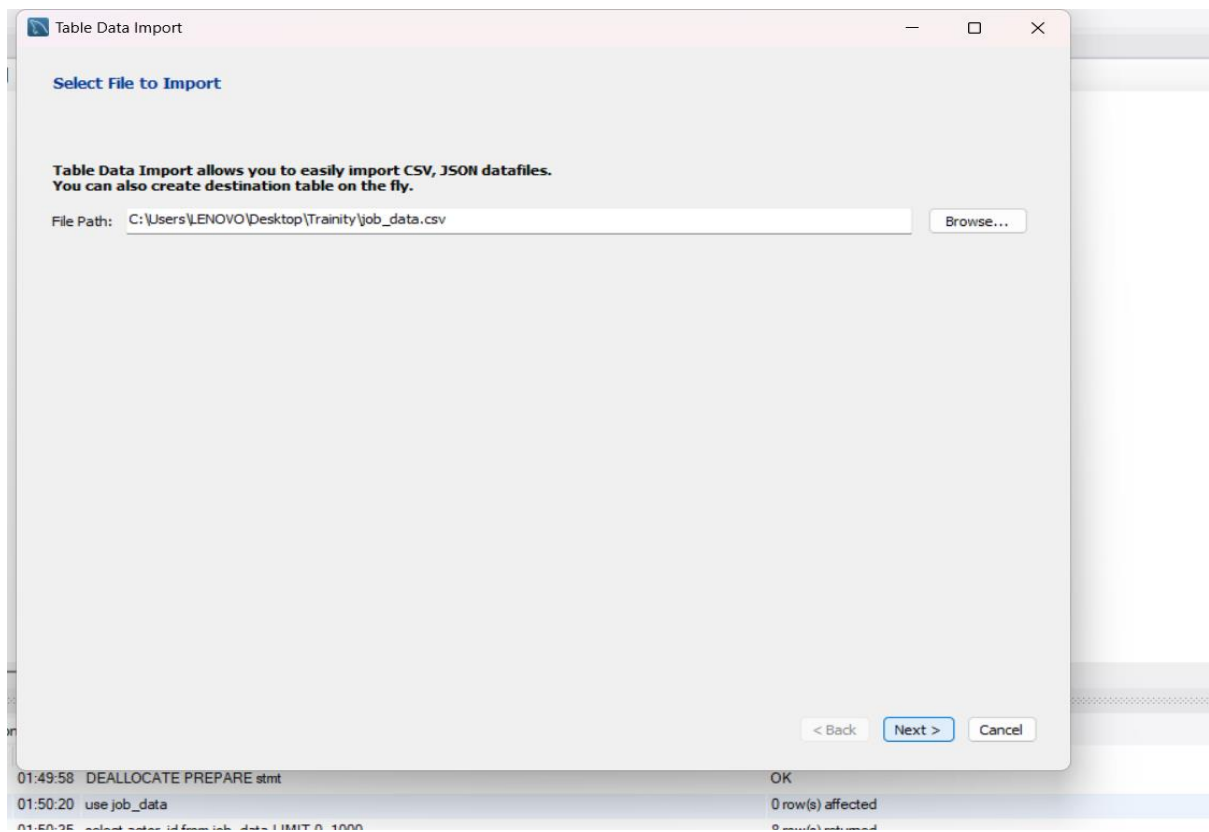
Creation of database



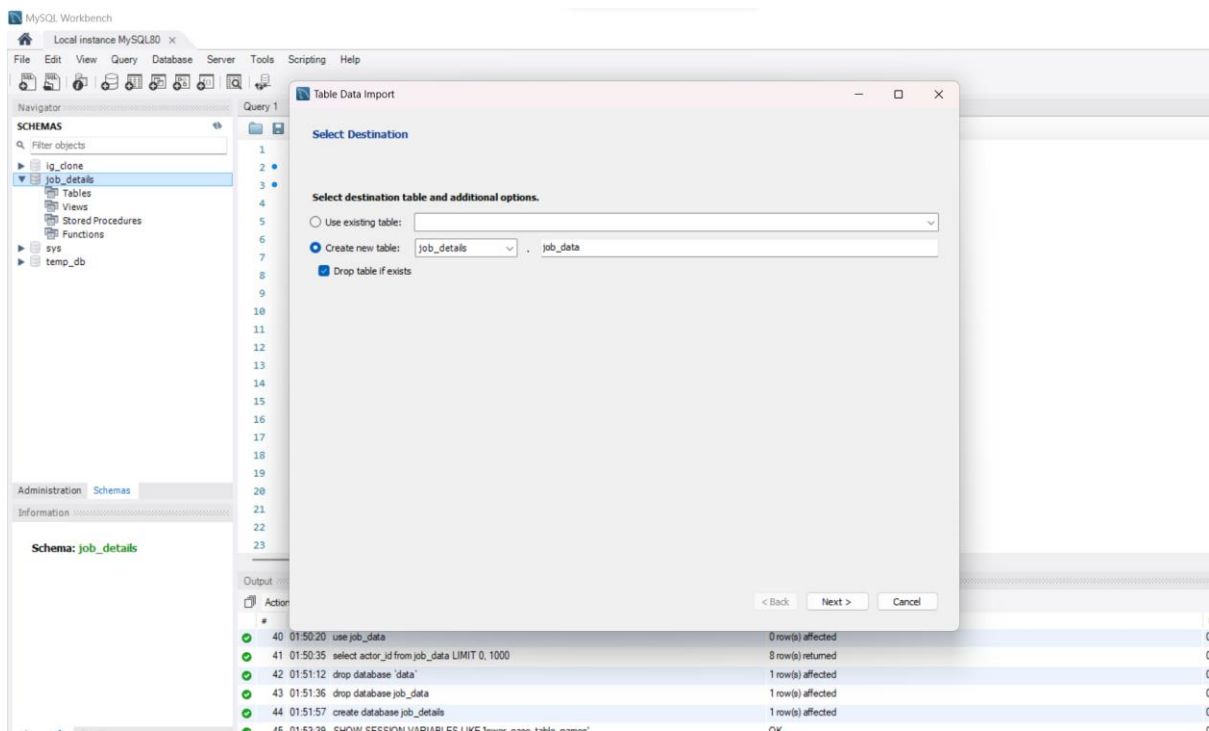
Importing the table data



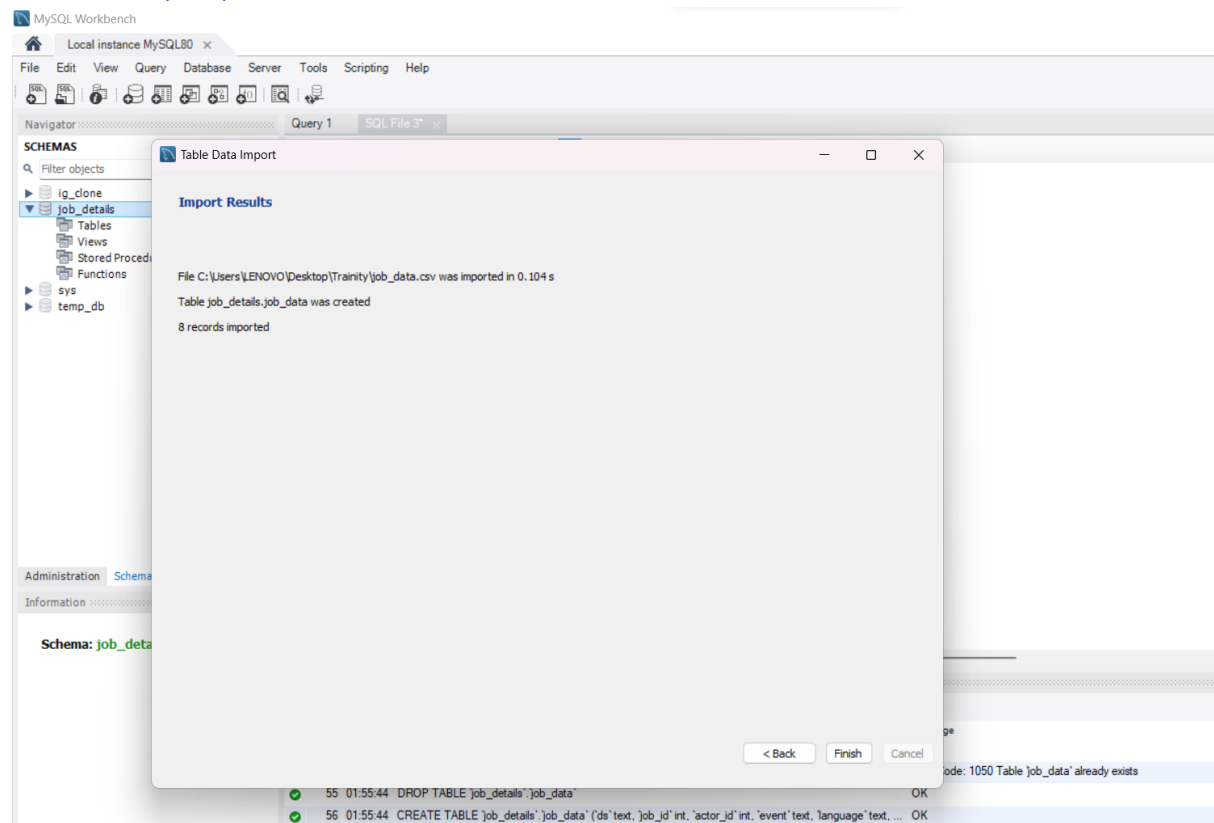
Browsed the CSV file



Selected the destination of file



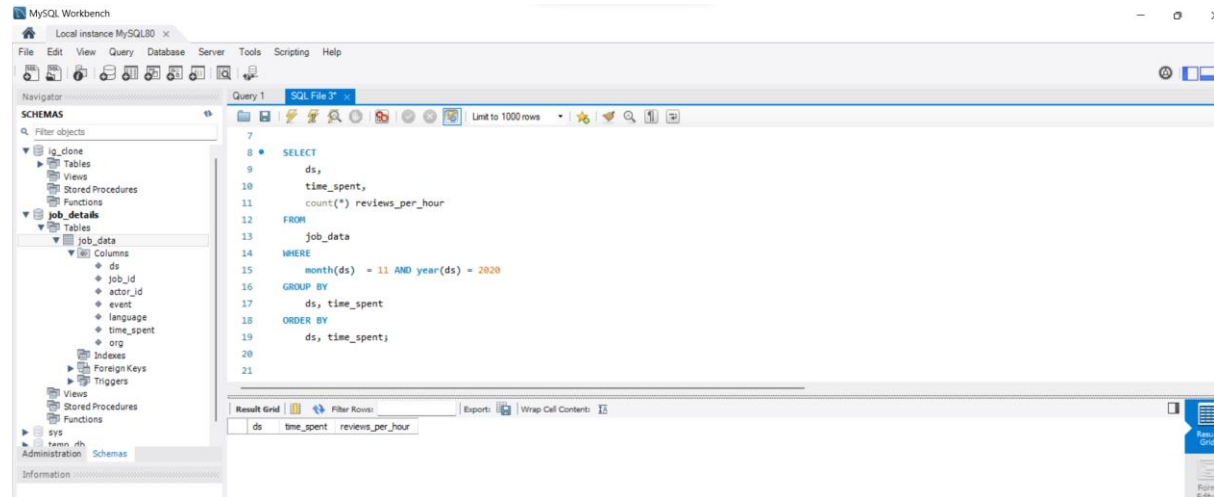
Successfully imported the data.



Query 1: Over Time Jobs Reviewed

Approach - The goal is to determine the average rate of job reviews per hour for the month of November 2020. To find out how many jobs will be examined per hour daily in November 2020, you can write a SQL query to find out.

Output:



Code:

SELECT

ds,

time_spent,

count(*) reviews_per_hour

FROM

job_data

WHERE

month(ds) = 11 AND year(ds) = 2020

GROUP BY

ds, time_spent

ORDER BY

ds, time_spent;

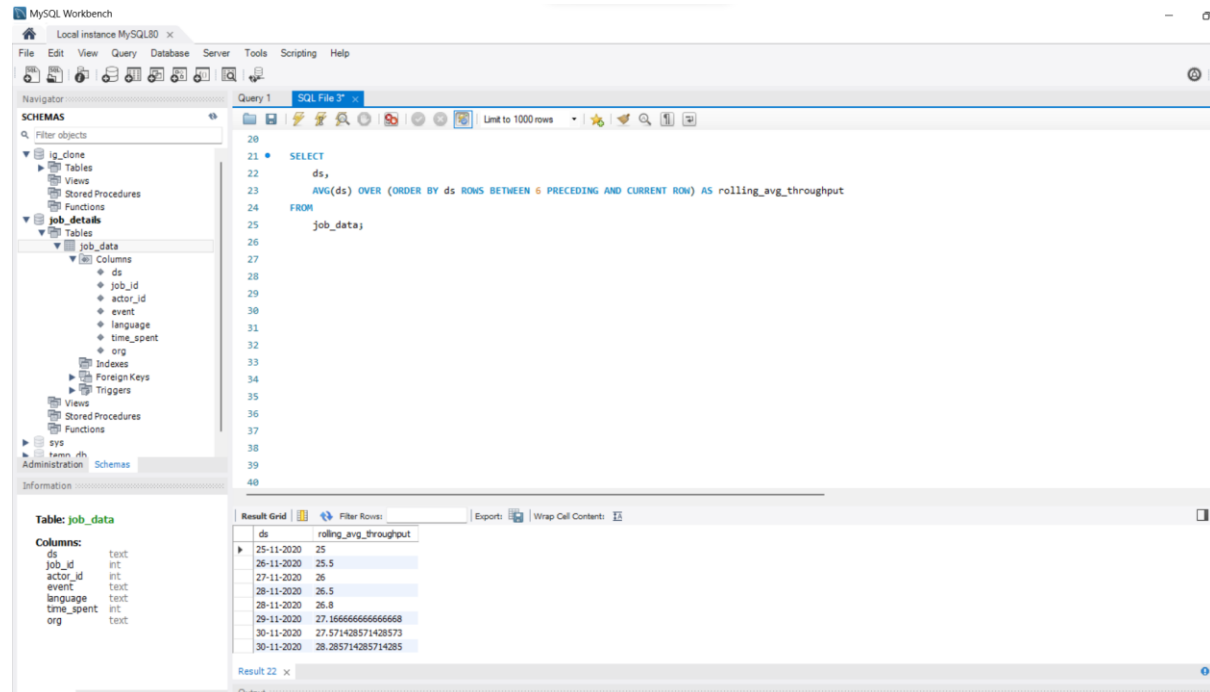
Explanation:

- Using the WHERE clause, we're filtering the rows to include just the reviews that occurred in November 2020.
- The MONTH() and YEAR() functions, respectively, extract the month and year from the ds.
- We're grouping the results by ds and time_spent using the GROUP BY clause.
- The COUNT(*) function determines the number of reviews per hour.
- Finally, the ORDER BY clause orders the results by ds, followed by time_spent.

Query 2: Throughput Analysis

Approach -The goal is to determine the throughput (events per second) on a rolling 7-day basis. Create a SQL query to determine the 7-day moving throughput average. Also, please elaborate on why you favour a daily metric over a rolling 7-day average for throughput.

Output:



The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' pane shows the 'job_details' database selected, with the 'job_data' table highlighted. The 'Columns' pane for 'job_data' lists: ds (text), job_id (int), actor_id (int), event (text), language (text), time_spent (int), and org (text). The main editor shows a SQL query for 'Query 1' (SQL File 37) with the following code:

```
20
21 SELECT
22   ds,
23   AVG(ds) OVER (ORDER BY ds ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_avg_throughput
24 FROM
25   job_data;
```

The 'Result Grid' at the bottom shows the output of the query, with columns 'ds' and 'rolling_avg_throughput'. The data is as follows:

ds	rolling_avg_throughput
25-11-2020	25
26-11-2020	25.5
27-11-2020	26
28-11-2020	26.5
28-11-2020	26.8
29-11-2020	27.166666666666668
30-11-2020	27.571428571428573
30-11-2020	28.285714285714285

Code:

SELECT

ds,

AVG(ds) OVER (ORDER BY ds ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS
rolling_avg_throughput

FROM

job_data;

Explanation:

- To calculate the rolling average, use the AVG() function with the OVER clause and ROWS BETWEEN.
- ORDER BY ds determines how rows are ordered depending on the ds column.
- THE FRAMEWORK FOR THE ROLLING AVERAGE IS DEFINED BY THE ROWS BETWEEN 6 AND CURRENT ROW. It considers the current row as well as the previous six rows, which spans a seven-day period.

Preference:

Here's an explanation of why I favour the daily number over the 7-day rolling average for throughput:

Daily Throughput Metric: The daily measure for throughput gives an easy method to understand the number of events per second on a given day. It provides a clear picture of throughput for individual days. This might help you spot patterns, trends, or abnormalities on specific days.

Throughput 7-Day Rolling Average: The 7-day rolling average smooths out the evolution of throughput over time. It considers the fluctuation that may occur from day to day owing to circumstances such as weekends, holidays, or other abnormalities. The rolling average can assist you in identifying longer-term patterns and variations in throughput while filtering out the noise caused by daily fluctuations.

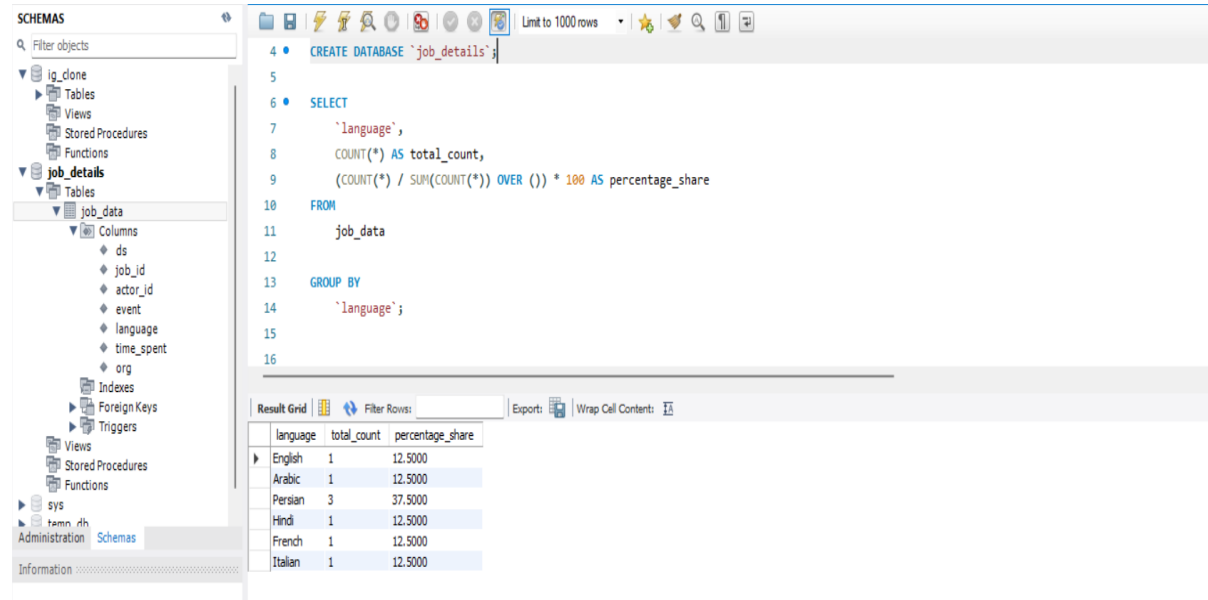
Preference: The choice between the daily metric and the 7-day rolling average is determined by the exact insights you need. The daily statistic may be more appropriate if you want to study day-to-day fluctuations and discover abrupt spikes or dips in throughput. The 7-day rolling average, on the other hand, might be a better choice if you're looking for patterns and a steadier representation of throughput.

Finally, all metrics have advantages, and the choice is dependent on your analytical goals and the level of detail required. It's common practise to employ a combination of to acquire a thorough grasp of the data.

Query 3: Language Share Analysis

Approach -The aim is to determine the relative popularity of each language during the past 30 days. To find out how many people spoke each language in the last 30 days, you can write a SQL query to find out.

Output:



The screenshot shows a SQL IDE interface. On the left, a 'SCHEMAS' pane displays a tree view with 'job_details' expanded, showing 'job_data' and its columns: 'ds', 'job_id', 'actor_id', 'event', 'language', 'time_spent', and 'org'. The main editor displays a SQL query:

```
4 CREATE DATABASE `job_details`;
5
6 SELECT
7     `language`,
8     COUNT(*) AS total_count,
9     (COUNT(*) / SUM(COUNT(*) OVER ())) * 100 AS percentage_share
10 FROM
11     job_data
12
13 GROUP BY
14     `language`;
15
16
```

 Below the query editor, a 'Result Grid' shows the output of the query. It has columns 'language', 'total_count', and 'percentage_share'. The data is grouped by language: English (1, 12.5000), Arabic (1, 12.5000), Persian (3, 37.5000), Hindi (1, 12.5000), French (1, 12.5000), and Italian (1, 12.5000).

language	total_count	percentage_share
English	1	12.5000
Arabic	1	12.5000
Persian	3	37.5000
Hindi	1	12.5000
French	1	12.5000
Italian	1	12.5000

Code:

SELECT

 `language`,

 COUNT(*) AS total_count,

 (COUNT(*) / SUM(COUNT(*) OVER ())) * 100 AS percentage_share

FROM

 job_data

GROUP BY

 `language`;

Explanation:

- The SELECT statement chooses the language column and uses the COUNT(*) aggregate function to get the total number of records for each language.
- The SUM(COUNT(*) OVER ()) function computes the total number of records across all languages. This number is used to compute the percentage share.
- (COUNT(*) / SUM(COUNT(*) OVER ())) * 100 computes the percentage share for each language by dividing the overall count by the count for that language and then multiplying by 100.
- The GROUP BY clause categorises the results based on the language column.

Query 4: Duplicate Rows Detection

Approach -The goal is to find all instances of duplicate data in the table. Write a query in SQL to show duplicate records in the job_data table.

Output:

The screenshot shows a SQL IDE interface. On the left, the 'SCHEMAS' pane displays the database structure, including the 'job_data' table with columns: ds (text), job_id (int), actor_id (int), event (text), language (text), time_spent (int), and org (text). The main editor displays the following SQL query:

```
SELECT *
FROM job_data
WHERE (
  ds,
  job_id,
  actor_id,
  `event`,
  `language`,
  time_spent,
  org
)
IN (
  SELECT
    ds, job_id, actor_id, `event`, `language`, time_spent, org
  FROM job_data
  GROUP BY
    ds, job_id, actor_id, `event`, `language`, time_spent, org
  HAVING
    COUNT(*) > 1
);
```

At the bottom, the 'Result Grid' shows the columns: ds, job_id, actor_id, event, language, time_spent, org.

Code:

```
SELECT *
FROM
  job_data
WHERE
  (
    ds,
    job_id,
    actor_id,
    `event`,
    `language`,
    time_spent,
    org
  )
IN
  (
    SELECT
      ds, job_id, actor_id, `event`, `language`, time_spent, org
    FROM
```

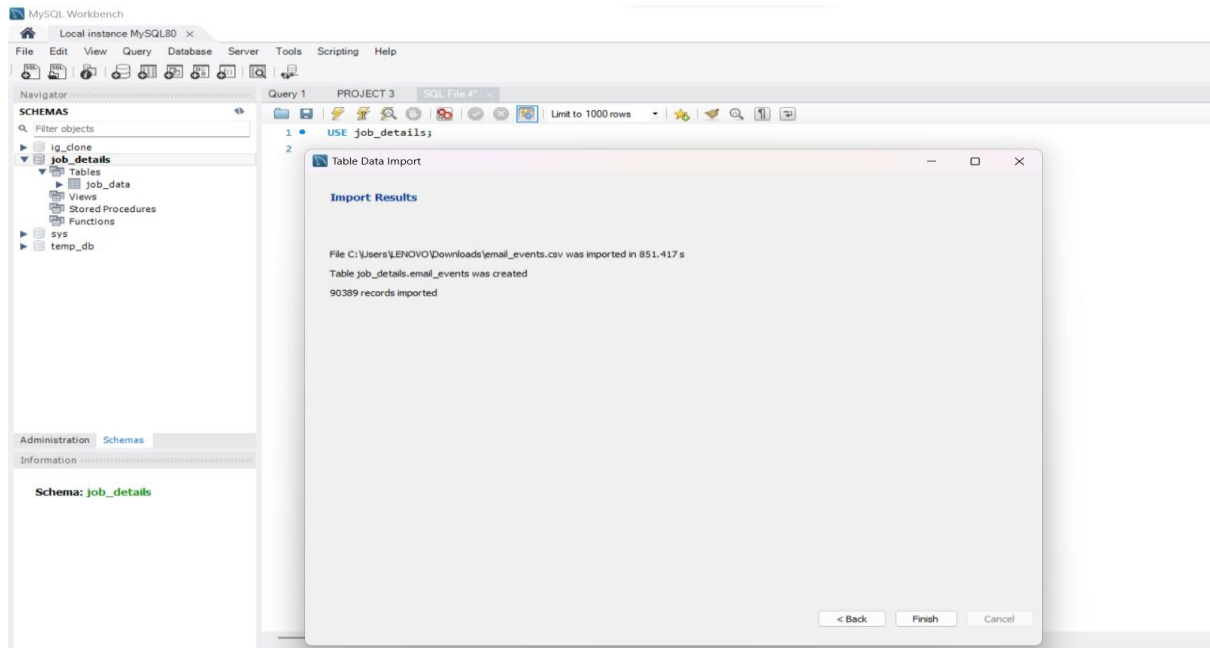
```
    job_data
GROUP BY
    ds, job_id, actor_id, `event`, `language`, time_spent, org
HAVING
    COUNT(*) > 1
);
```

Explanation:

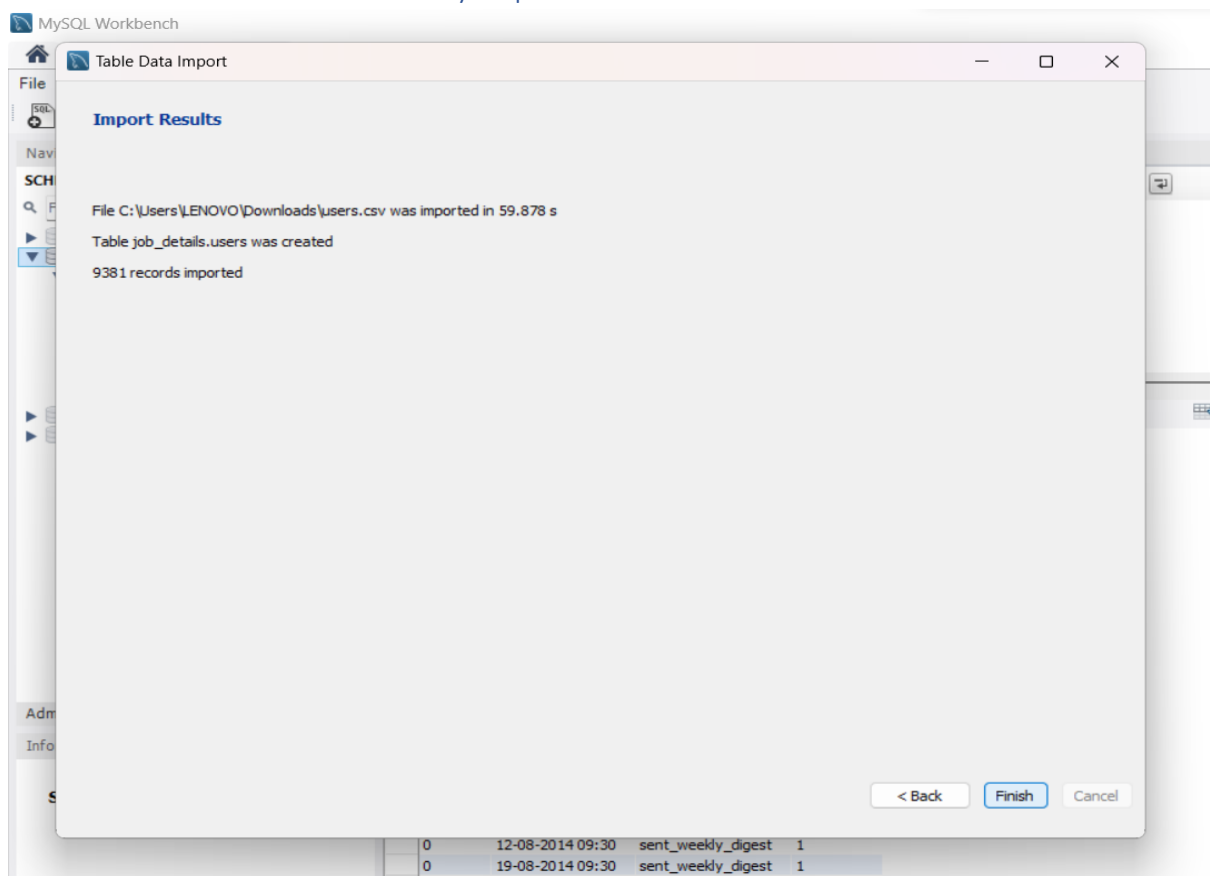
- The outer SELECT statement queries the job_data table for all columns (*).
- The WHERE clause restricts the rows to those that match a subquery.
- The IN clause subquery selects rows with duplicate values in specified fields (ds, job_id, actor_id, 'event', 'language', time_spent, org).
- A aggregate BY clause is used in the subquery to aggregate rows based on the same set of columns (ds, job_id, actor_id, 'event', 'language', time_spent, org).
- The HAVING clause in the subquery eliminates groups with more than one row, indicating duplicates.

CASE STUDY 2

Importing The Data



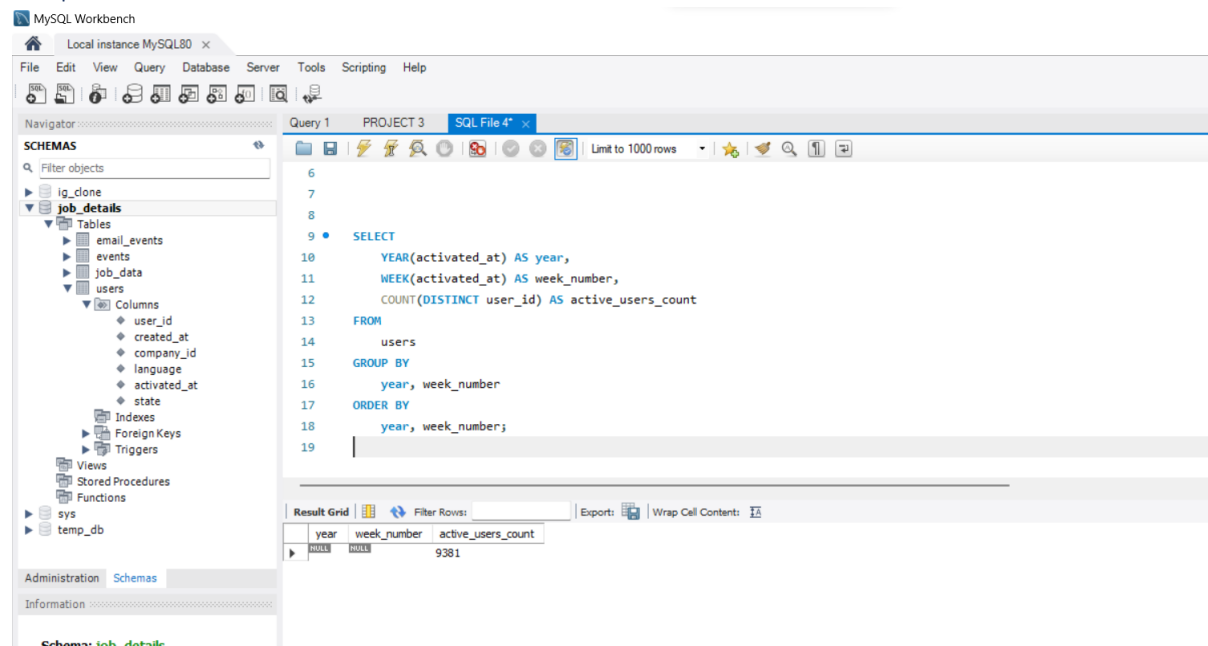
All the data has been successfully imported.



Query 1: Weekly User Engagement

Approach -The goal is to track user engagement on a weekly basis. Create a SQL query to determine the weekly activity of the users.

Output:



Code:

SELECT

YEAR(activated_at) AS year,

WEEK(activated_at) AS week_number,

COUNT(DISTINCT user_id) AS active_users_count

FROM

users

GROUP BY

year, week_number

ORDER BY

year, week_number;

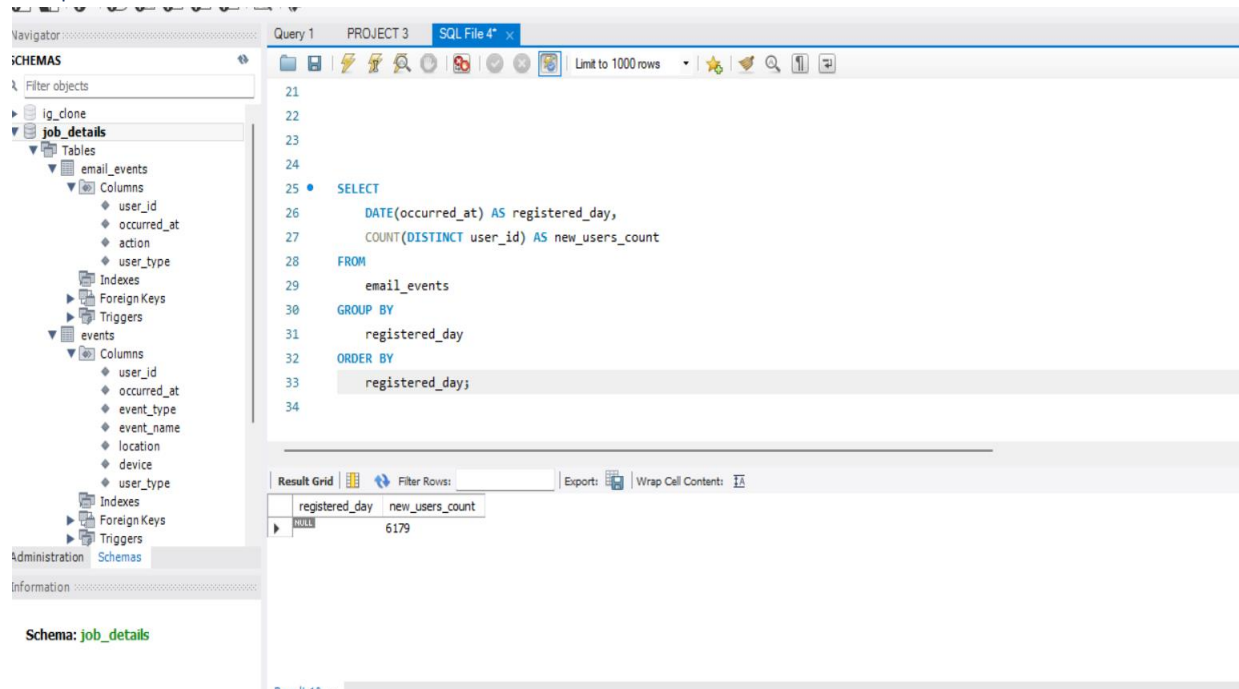
Explanation:

- The SELECT statement uses the YEAR() and WEEK() procedures to derive the year and week number from the activated_at.
- The COUNT(DISTINCT user_id) function counts the number of unique active users each week. Using DISTINCT ensures that each user is tallied just once every week, regardless of how many encounters they have.
- The GROUP BY clause categorises the results based on the year and week number.
- The ORDER BY clause organises the results by year and week number in ascending order.

Query 2: User Growth Analysis

Approach -Analysis of a product's user base as it expands over time is the goal. To determine the rate of product adoption, compose a SQL query.

Output:



The screenshot shows a SQL IDE interface. On the left is a 'Navigator' pane with a tree view of a database schema named 'job_details'. It includes tables like 'email_events' and 'events', each with columns such as 'user_id', 'occurred_at', 'action', 'event_type', 'location', 'device', and 'user_type'. The main area displays a SQL query for 'Query 1' in 'PROJECT 3'. The query is as follows:

```
21
22
23
24
25 • SELECT
26     DATE(occurred_at) AS registered_day,
27     COUNT(DISTINCT user_id) AS new_users_count
28 FROM
29     email_events
30 GROUP BY
31     registered_day
32 ORDER BY
33     registered_day;
34
```

Below the query editor, the 'Result Grid' shows the output of the query:

registered_day	new_users_count
NULL	6179

The interface also includes a toolbar with various icons and a status bar at the bottom indicating 'Schema: job_details'.

Code:

SELECT

DATE(occurred_at) AS registered_day,

COUNT(DISTINCT user_id) AS new_users_count

FROM

email_events

GROUP BY

registered_day

ORDER BY

registered_day;

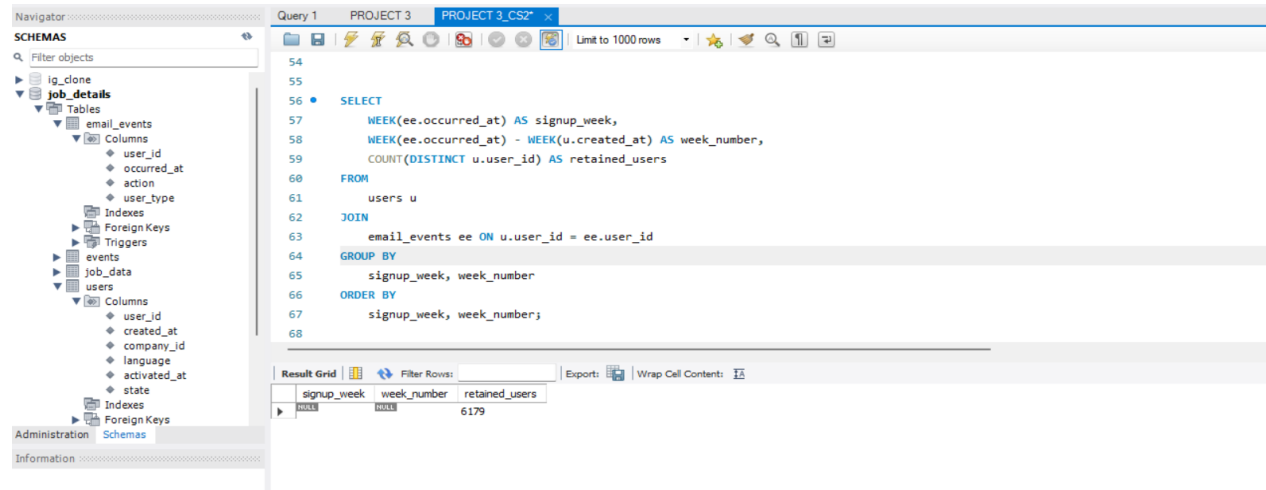
Explanation:

- The SELECT statement uses the DATE() function to get the date from the occurred_at.
- The COUNT(DISTINCT user_id) function counts the number of distinct new users registered on each signup day.
- Using DISTINCT ensures that each user is only tallied once, even if they sign up more than once on the same day.
- The GROUP BY clause categorises the results based on the recorded day.
- The ORDER BY clause organises the results by registered day in ascending order.

Query 3: Weekly Retention Analysis

Approach -The goal is to examine how many customers continue using a service or product each week after signing up. Create a SQL query to determine user retention on a weekly basis by cohort of signups.

Output:



```
54
55
56 • SELECT
57     WEEK(ee.occurred_at) AS signup_week,
58     WEEK(ee.occurred_at) - WEEK(u.created_at) AS week_number,
59     COUNT(DISTINCT u.user_id) AS retained_users
60 FROM
61     users u
62 JOIN
63     email_events ee ON u.user_id = ee.user_id
64 GROUP BY
65     signup_week, week_number
66 ORDER BY
67     signup_week, week_number;
68
```

signup_week	week_number	retained_users
6179	6179	

Code:

SELECT

WEEK(ee.occurred_at) AS signup_week,

WEEK(ee.occurred_at) - WEEK(u.created_at) AS week_number,

COUNT(DISTINCT u.user_id) AS retained_users

FROM

users u

JOIN

email_events ee ON u.user_id = ee.user_id

GROUP BY

signup_week, week_number

ORDER BY

signup_week, week_number;

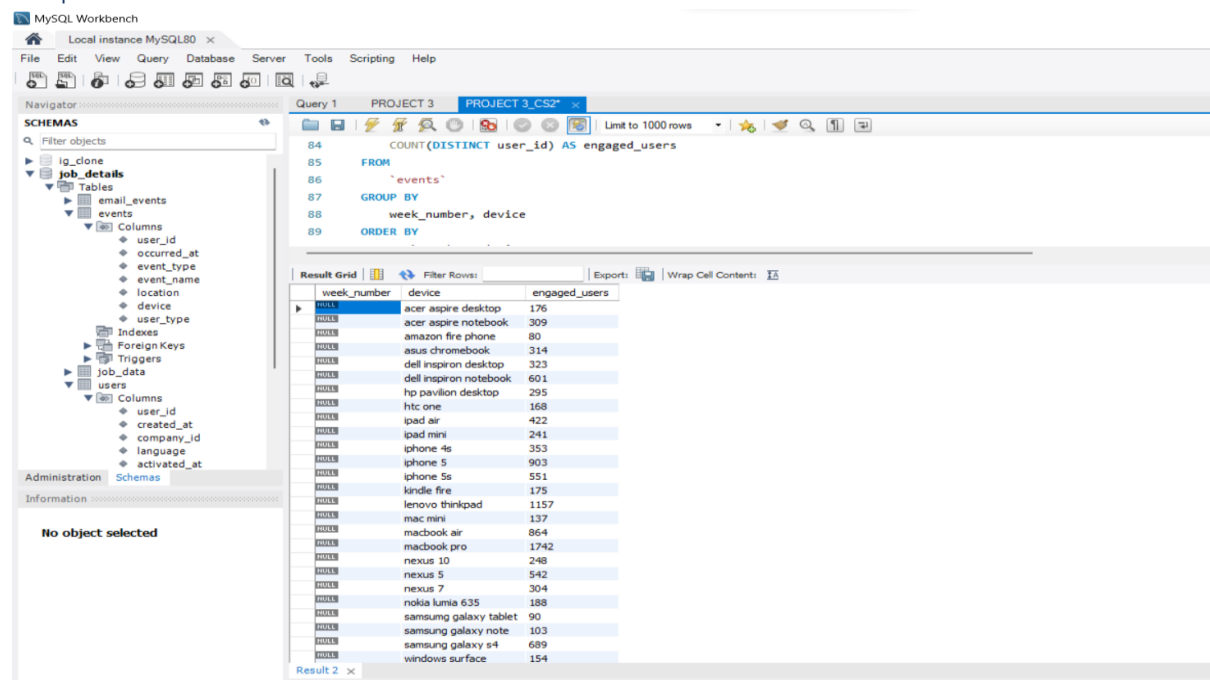
Explanation:

- The users table has the alias u, while the email_events table has the alias ee.
- The user_id field is used to link the two tables.
- We compute the signup_week using WEEK(ee.occurred_at) and the week_number by subtracting WEEK(u.created_at) from WEEK(ee.occurred_at).
- The GROUP BY and ORDER BY clauses remain unchanged, grouping and arranging the results by signup_week and week_number, respectively.

Query 4: Weekly Engagement Per Device

Approach -The goal is to track how often each gadget is used each week. Create a SQL query to determine the average weekly usage by device.

Output:



The screenshot shows the MySQL Workbench interface. On the left is the 'SCHEMAS' navigator showing a tree view of the database structure, including tables like 'email_events', 'events', 'users', and 'job_data'. The main area displays a SQL query in the 'Query 1' editor. The query is as follows:

```
84 COUNT(DISTINCT user_id) AS engaged_users
85 FROM
86 `events`
87 GROUP BY
88 week_number, device
89 ORDER BY
```

Below the query editor is the 'Result Grid' showing the output of the query. The results are sorted by week_number and device. The columns are week_number, device, and engaged_users.

week_number	device	engaged_users
2015-01-05	acer aspire desktop	176
2015-01-05	acer aspire notebook	309
2015-01-05	amazon fire phone	80
2015-01-05	asus chromebook	314
2015-01-05	dell inspiron desktop	323
2015-01-05	dell inspiron notebook	601
2015-01-05	hp pavilion desktop	295
2015-01-05	htc one	168
2015-01-05	ipad air	422
2015-01-05	ipad mini	241
2015-01-05	iphone 4s	353
2015-01-05	iphone 5	903
2015-01-05	iphone 5s	551
2015-01-05	kindle fire	175
2015-01-05	lenovo thinkpad	1157
2015-01-05	mac mini	137
2015-01-05	macbook air	864
2015-01-05	macbook pro	1742
2015-01-05	nexus 10	248
2015-01-05	nexus 5	542
2015-01-05	nexus 7	304
2015-01-05	nokia lumia 635	188
2015-01-05	samsung galaxy tablet	90
2015-01-05	samsung galaxy note	103
2015-01-05	samsung galaxy s4	689
2015-01-05	windows surface	154

Code:

SELECT

WEEK(occurred_at) AS week_number,

device,

COUNT(DISTINCT user_id) AS engaged_users

FROM

`events`

GROUP BY

week_number, device

ORDER BY

week_number, device;

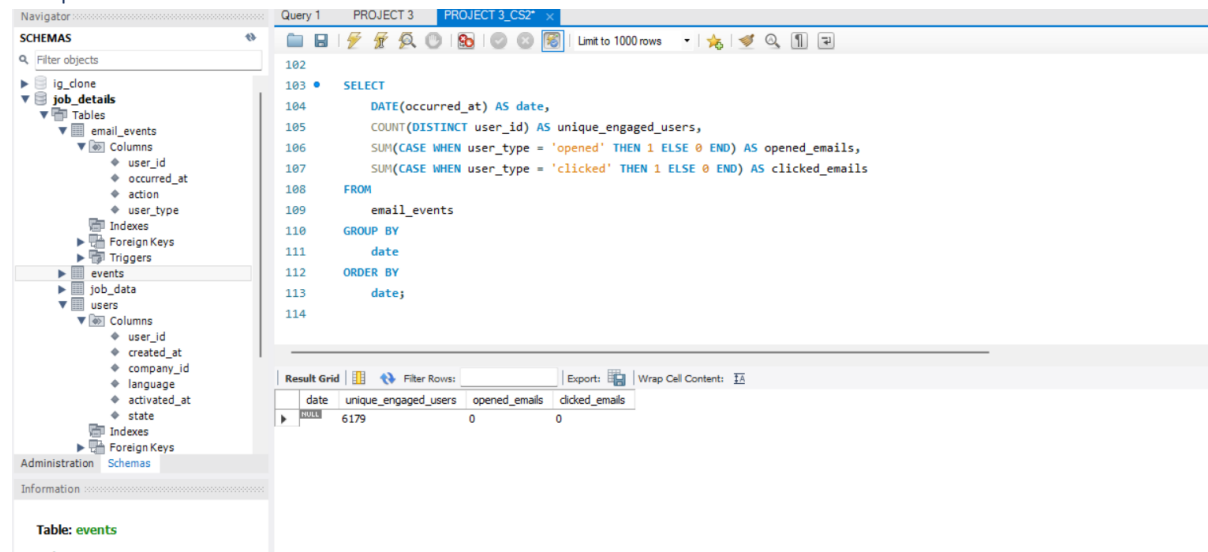
Explanation:

- WEEK(occurred_at) calculates the week number of the occurred_at.
- COUNT(DISTINCT user_id) calculates the count of distinct users engaged with each device in each week.
- The GROUP BY clause groups the results by week number and device.
- The ORDER BY clause sorts the results in ascending order of week number and device.

Query 5: Email Engagement Analysis

Approach -Examine how people interact with the email service. To calculate the email engagement metrics, create a SQL query.

Output:



The screenshot shows a SQL IDE interface. On the left is a 'SCHEMAS' pane with a tree view containing 'ig_clone', 'job_details', 'email_events', 'events', 'job_data', and 'users'. The 'email_events' table is selected, showing its columns: 'user_id', 'occurred_at', 'action', and 'user_type'. The main pane displays a SQL query (lines 103-114) and its results in a 'Result Grid' (lines 115-116). The query calculates email engagement metrics grouped by date. The result grid shows one row for the date '2015-01-01' with 6179 unique engaged users, 0 opened emails, and 0 clicked emails.

```
103 SELECT
104     DATE(occurred_at) AS date,
105     COUNT(DISTINCT user_id) AS unique_engaged_users,
106     SUM(CASE WHEN user_type = 'opened' THEN 1 ELSE 0 END) AS opened_emails,
107     SUM(CASE WHEN user_type = 'clicked' THEN 1 ELSE 0 END) AS clicked_emails
108 FROM
109     email_events
110 GROUP BY
111     date
112 ORDER BY
113     date;
114
```

date	unique_engaged_users	opened_emails	clicked_emails
2015-01-01	6179	0	0

Code:

SELECT

DATE(occurred_at) AS date,

COUNT(DISTINCT user_id) AS unique_engaged_users,

SUM(CASE WHEN user_type = 'opened' THEN 1 ELSE 0 END) AS opened_emails,

SUM(CASE WHEN user_type = 'clicked' THEN 1 ELSE 0 END) AS clicked_emails

FROM

email_events

GROUP BY

date

ORDER BY

date;

Explanation:

- The SELECT statement uses the DATE() function to get the date from the email_date.
- COUNT(DISTINCT user_id) computes the number of unique users that opened emails on each day.
- SUM(CASE WHEN user_type = 'opened' THEN 1 ELSE 0 END) computes the total number of emails opened for each date.
- SUM(CASE WHEN user = 'clicked' THEN 1 ELSE 0 END) computes the total number of emails clicked for each date.
- The GROUP BY clause categorises the results based on the date.
- The ORDER BY clause arranges the results in ascending date order.