

Operational Analytics And Investigating Metric Spikes

Project Description

To perform operational Analytics that involves analyzing company's end-to-end operations on daily basis. This involves understanding and explaining sudden changes in key metrics, such as a dip in daily user engagement or a drop in sales.

Tech Stack Used

For this project, I have used Mysql workbench 8.0.35 CE. I have chosen this because it is an easy-to-use Mysql platform. It is used to create databases and perform various SQL queries. I have also used MS Excel to change the date formats in all the sheets and MS Word to generate the report on operational analytics and then converted into PDF format for submission.

Case Study-1

Approach

After analyzing the dataset that contains information on jobs reviewed by people looking to become actor on daily basis and the other activities performed by them, I have found

1. Number of jobs reviewed per hour for each day in November 2020

The output is :

ds	total_jobs_per_day	hours_spent_each_day
2020-11-30	2	0.0111
2020-11-28	2	0.0092
2020-11-08	1	0.0106
2020-11-27	1	0.0289
2020-11-23	1	0.0144
2020-11-17	1	0.0258
2020-11-19	1	0.0064
2020-11-15	1	0.0272
2020-11-04	1	0.0272
2020-11-18	1	0.0147
2020-11-10	1	0.0172

2020-11-21	1	0.0211
2020-11-13	1	0.0111
2020-11-14	1	0.0089
2020-11-12	1	0.0042
2020-11-22	1	0.0186
2020-11-03	1	0.0186
2020-11-24	1	0.0103
2020-11-16	1	0.0086
2020-11-25	1	0.0125
2020-11-02	1	0.0067
2020-11-09	1	0.0208
2020-11-29	1	0.0056
2020-11-26	1	0.0156
2020-11-11	1	0.0106
2020-11-05	1	0.0086
2020-11-07	1	0.0172
2020-11-20	1	0.0219

SQL Query With Output

The screenshot displays the MySQL Workbench interface. The central pane shows a SQL query (Query 1) that calculates the number of jobs reviewed per hour for each day. The query is as follows:

```

37
38
39
40 # number of jobs reviewed per hour for each day
41 • SELECT
42     ds,
43     COUNT(job_id) AS total_jobs_per_day,
44     sum(time_spent) / 3600 as hours_spent_each_day # as multiple jobs can be reviewed on the same day
45 FROM
46     job_data
47 GROUP BY ds
48 order by total_jobs_per_day desc;
49
50
51

```

The bottom pane shows the Result Grid with the following data:

ds	total_jobs_per_day	hours_spent_each_day
2020-11-30	2	0.0111
2020-11-28	2	0.0092
2020-11-08	1	0.0106
2020-11-27	1	0.0289
2020-11-23	1	0.0144

On the left, the Schemas pane shows the database structure, including the 'job_data' table. The bottom left pane shows the table structure for 'job_data'.

Table: job_data

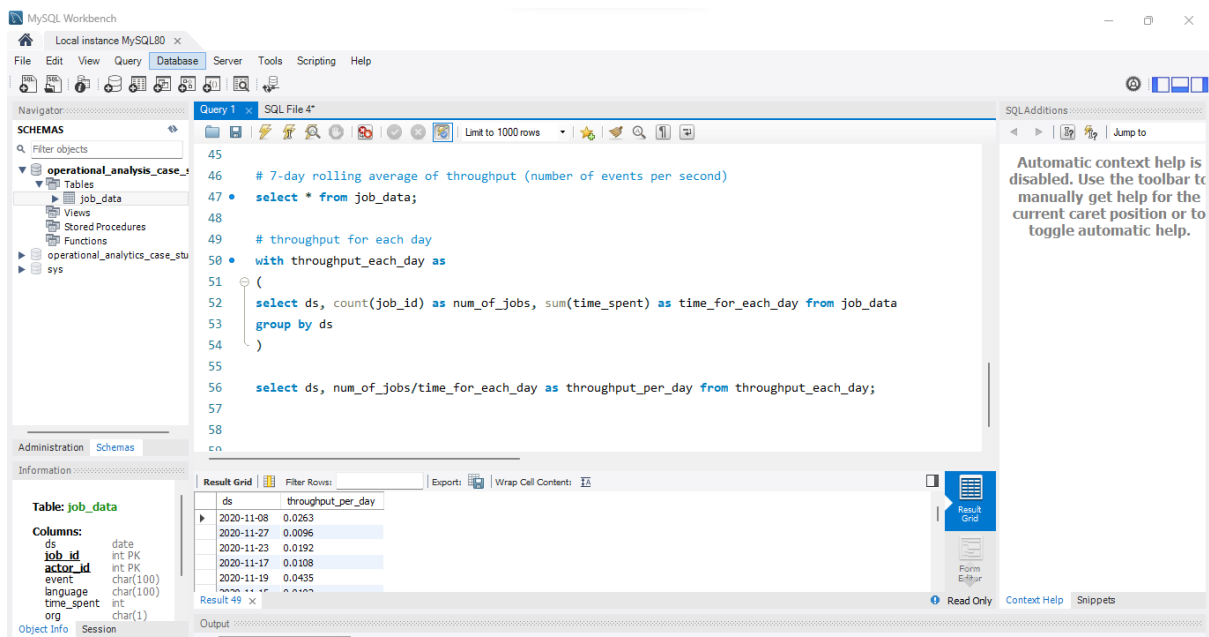
Columns:

- ds: date
- job_id: int PK
- actor_id: int PK
- event: char(100)
- language: char(100)
- time_spent: int
- org: char(1)

2. 7-day rolling average of throughput (number of events per second)

After analysis, it has been observed that calculating every-day average throughput is much better than 7-day rolling average throughput as it provides a better view of fluctuations happening each day. It helps to closely monitor everything.

SQL Query With Output for each day



The screenshot shows the MySQL Workbench interface. The 'Query 1' window contains the following SQL code:

```
45
46 # 7-day rolling average of throughput (number of events per second)
47 select * from job_data;
48
49 # throughput for each day
50 with throughput_each_day as
51 (
52   select ds, count(job_id) as num_of_jobs, sum(time_spent) as time_for_each_day from job_data
53   group by ds
54 )
55
56 select ds, num_of_jobs/time_for_each_day as throughput_per_day from throughput_each_day;
57
58
59
```

The 'Result Grid' shows the output of the query:

ds	throughput_per_day
2020-11-08	0.0263
2020-11-27	0.0096
2020-11-23	0.0192
2020-11-17	0.0108
2020-11-19	0.0435
2020-11-15	0.0102
2020-11-04	0.0102
2020-11-18	0.0189
2020-11-10	0.0161
2020-11-21	0.0132
2020-11-13	0.0250
2020-11-14	0.0313
2020-11-12	0.0667
2020-11-22	0.0149
2020-11-03	0.0149

Output is:

ds	throughput_per_day
2020-11-08	0.0263
2020-11-27	0.0096
2020-11-23	0.0192
2020-11-17	0.0108
2020-11-19	0.0435
2020-11-15	0.0102
2020-11-04	0.0102
2020-11-18	0.0189
2020-11-10	0.0161
2020-11-21	0.0132
2020-11-13	0.0250
2020-11-14	0.0313
2020-11-12	0.0667
2020-11-22	0.0149
2020-11-03	0.0149

2020-11-24	0.0270
2020-11-16	0.0323
2020-11-25	0.0222
2020-11-30	0.0500
2020-11-02	0.0417
2020-11-09	0.0133
2020-11-29	0.0500
2020-11-26	0.0179
2020-11-28	0.0606
2020-11-11	0.0263
2020-11-05	0.0323
2020-11-07	0.0161
2020-11-20	0.0127

SQL Query With Output for 7-day rolling Average

The screenshot shows the MySQL Workbench interface. The SQL Editor contains the following query:

```

53 group by ds
54 )
55
56 select ds, num_of_jobs/time_for_each_day As throughput_per_day from throughput_each_day;
57
58 # 7-day rolling average of throughput
59 # I have used window function as I need to calculate aggregation for 7-days only
60 with rolling_average_throughput as
61 (
62 select ds, count(job_id) over (order by ds rows between 6 preceding and current row) as num_of_jobs,
63 sum(time_spent) over (order by ds rows between 6 preceding and current row) as time_spent_for_a_week
64 from job_data
65 )
66 select ds, num_of_jobs/time_spent_for_a_week as 7_day_rolling_average from rolling_average_throughput;
67

```

The Result Grid shows the output of the query:

ds	7_day_rolling_average
2020-11-02	0.0417
2020-11-03	0.0220
2020-11-04	0.0159
2020-11-05	0.0182
2020-11-07	0.0177
2020-11-08	0.0188
2020-11-09	0.0177
2020-11-20	0.0127

Output is:

ds	7_day_rolling_average
2020-11-02	0.0417
2020-11-03	0.0220
2020-11-04	0.0159
2020-11-05	0.0182
2020-11-07	0.0177
2020-11-08	0.0188
2020-11-09	0.0177

2020-11-10	0.0162
2020-11-11	0.0173
2020-11-12	0.0218
2020-11-13	0.0212
2020-11-14	0.0233
2020-11-15	0.0194
2020-11-16	0.0222
2020-11-17	0.0202
2020-11-18	0.0193
2020-11-19	0.0189
2020-11-20	0.0171
2020-11-21	0.0155
2020-11-22	0.0166
2020-11-23	0.0158
2020-11-24	0.0181
2020-11-25	0.0185
2020-11-26	0.0170
2020-11-27	0.0160
2020-11-28	0.0183
2020-11-28	0.0214
2020-11-29	0.0237
2020-11-30	0.0256
2020-11-30	0.0277

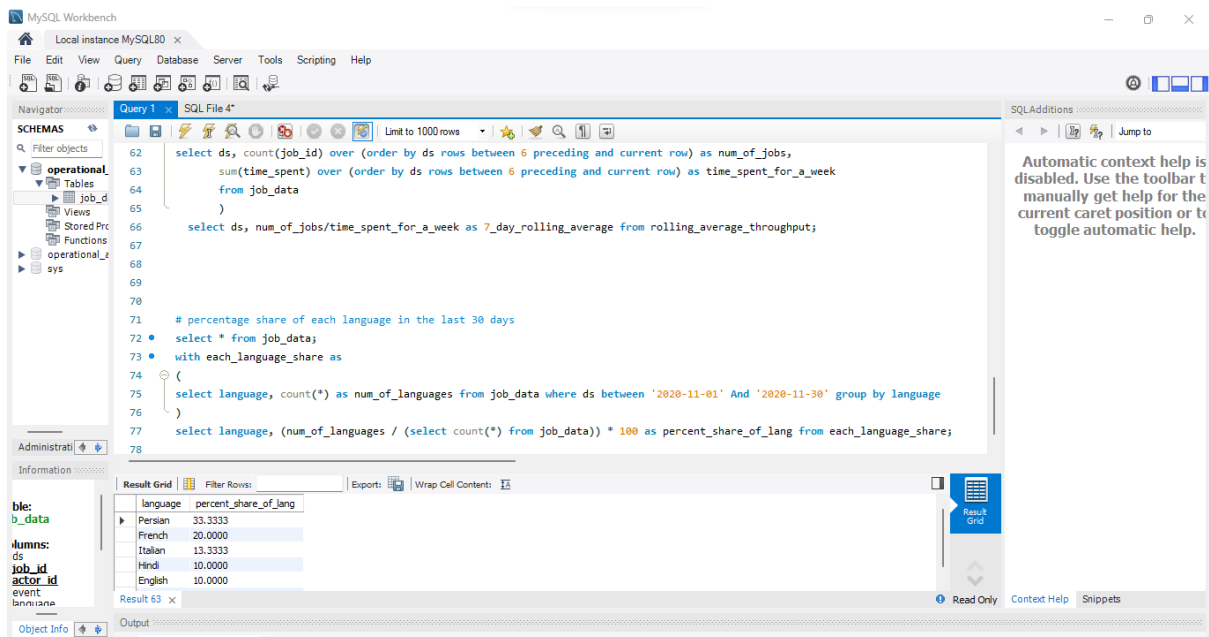
3. Percentage share of each language in the last 30 days

After analysis, it has been found that Persian is the language whose share is more than all other languages.

Output is:

language	percent_share_of_lang
Persian	33.3333
French	20.0000
Italian	13.3333
Arabic	13.3333
Hindi	10.0000
English	10.0000

SQL Query With Output



The screenshot shows the MySQL Workbench interface. The SQL editor contains a query that calculates the percentage share of each language in the last 30 days. The query is as follows:

```
62 select ds, count(job_id) over (order by ds rows between 6 preceding and current row) as num_of_jobs,
63 sum(time_spent) over (order by ds rows between 6 preceding and current row) as time_spent_for_a_week
64 from job_data
65 )
66 select ds, num_of_jobs/time_spent_for_a_week as 7_day_rolling_average from rolling_average_throughput;
67
68
69
70
71 # percentage share of each language in the last 30 days
72 select * from job_data;
73 with each_language_share as
74 (
75 select language, count(*) as num_of_languages from job_data where ds between '2020-11-01' And '2020-11-30' group by language
76 )
77 select language, (num_of_languages / (select count(*) from job_data)) * 100 as percent_share_of_lang from each_language_share;
78
```

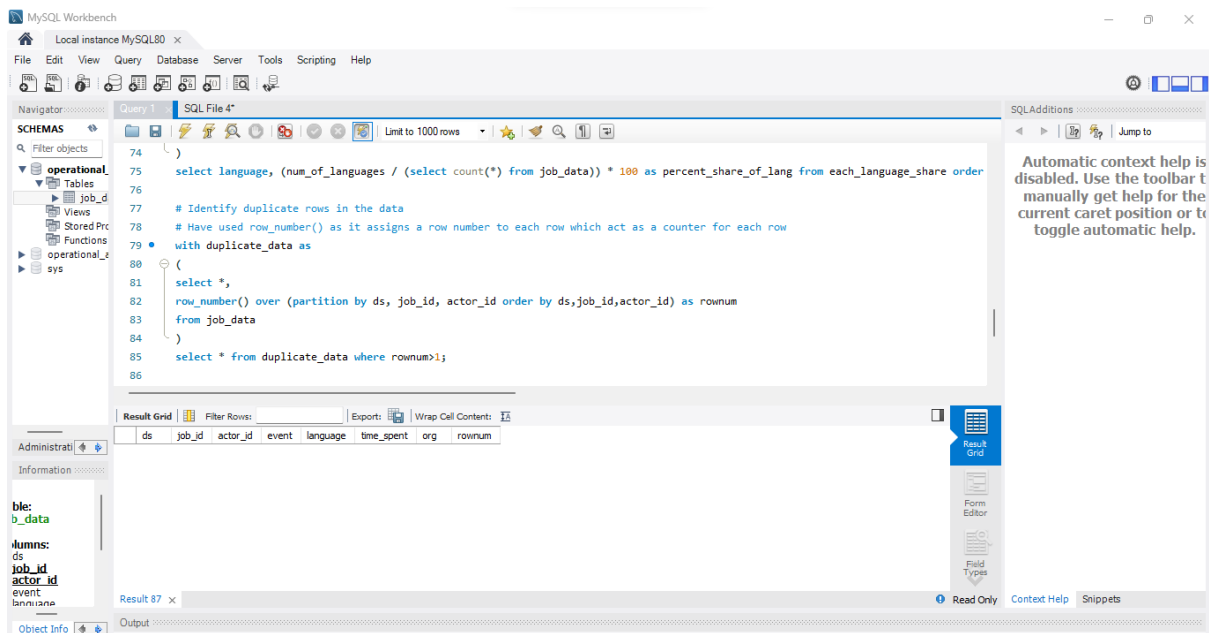
The Result Grid shows the output of the query, displaying columns: language, percent_share_of_lang. The data is as follows:

language	percent_share_of_lang
Persian	33.3333
French	20.0000
Italian	13.3333
Hindi	10.0000
English	10.0000

4. Identify duplicate rows in the data

After analysis, it has been found that the given dataset doesn't contain any duplicate records. Although, each column does contain duplicate values individually. But, no duplicate rows has been found.

SQL Query With Output



The screenshot shows the MySQL Workbench interface. The SQL editor contains a query to identify duplicate rows in the data. The query is as follows:

```
74 )
75 select language, (num_of_languages / (select count(*) from job_data)) * 100 as percent_share_of_lang from each_language_share order
76
77 # Identify duplicate rows in the data
78 # Have used row_number() as it assigns a row number to each row which act as a counter for each row
79 with duplicate_data as
80 (
81 select *,
82 row_number() over (partition by ds, job_id, actor_id order by ds, job_id, actor_id) as rownum
83 from job_data
84 )
85 select * from duplicate_data where rownum>1;
86
```

The Result Grid shows the output of the query, displaying columns: ds, job_id, actor_id, event, language, time_spent, org, rownum. The data is as follows:

ds	job_id	actor_id	event	language	time_spent	org	rownum
----	--------	----------	-------	----------	------------	-----	--------

Case Study – 2

Approach

After analyzing the dataset that contains information on users and their engagement and the other activities performed by them, I have found

1. Measure the activeness of users on a weekly basis

Below is the list that shows the count of active user engagement per week. The overall user engagement per week is good.

event_type	week_of_the_year	weekly_user_engagement
engagement	17	8019
engagement	18	17341
engagement	19	17224
engagement	20	17911
engagement	21	17151
engagement	23	18280
engagement	22	18413
engagement	24	19052
engagement	25	18642
engagement	29	20067
engagement	26	19061
engagement	30	21533
engagement	28	20776
engagement	27	19881
engagement	31	18556
engagement	32	16612
engagement	33	16145
engagement	34	16127
engagement	35	784

The screenshot displays the MySQL Workbench interface. The top menu bar includes File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The left sidebar shows the 'SCHEMAS' tab with a tree view of the database structure, including 'operational_analysis_case_study1' and 'operational_analysis_case_study2'. The main editor window shows a SQL query in 'Query 1' with the following code:

```

1 use operational_analytics_case_study_2;
2
3 # weekly user engagement
4
5 SELECT
6   event_type, WEEK(DATE(occured_at)) AS week_of_the_year,
7   COUNT(user_id) AS weekly_user_engagement
8 FROM
9   events
10  where event_type="engagement"
11 GROUP BY week_of_the_year;
12
13
14

```

Below the query editor, the 'Result Grid' tab is active, showing the results of the query. The grid has three columns: 'event_type', 'week_of_the_year', and 'weekly_user_engagement'. The data is as follows:

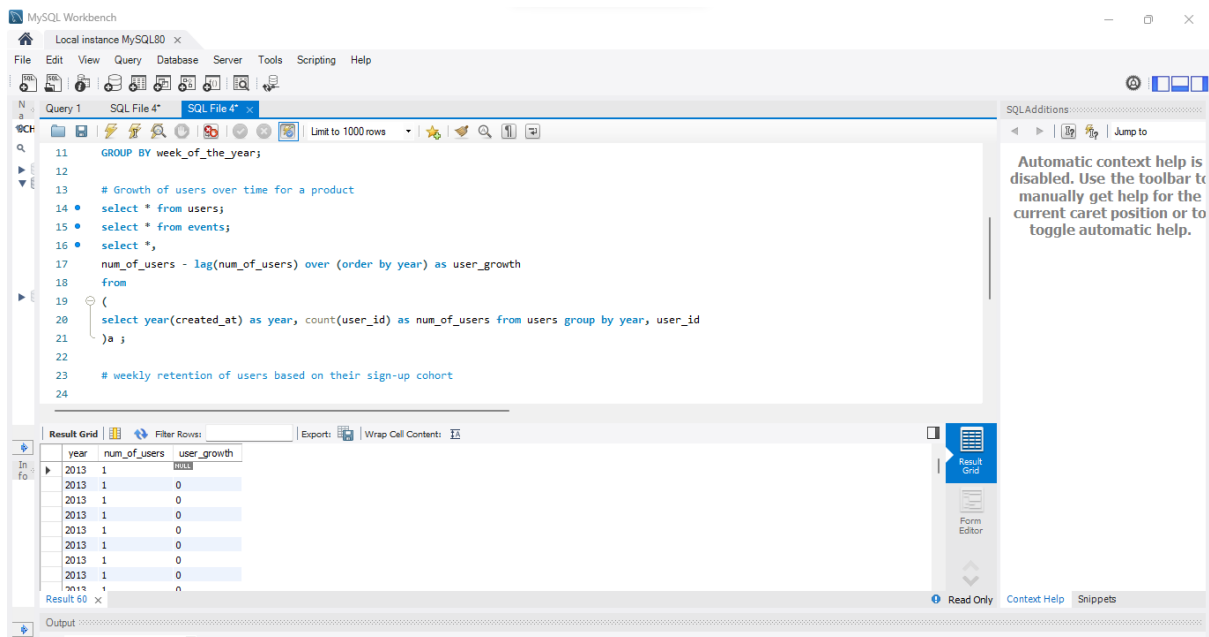
event_type	week_of_the_year	weekly_user_engagement
engagement	17	8019
engagement	18	17341
engagement	19	17224
engagement	20	17911
engagement	21	17151
engagement	23	18280
engagement	22	18413
engagement	24	19052

The bottom status bar indicates 'Read Only', 'Context Help', and 'Snippets'.

Below are results showing the number of users joined each year. As, it is clear that number of users joined in 2014 are more than number of users in 2013 which means that more users have engaged with the product.

[illegible]

SQL Query With Output



The screenshot shows the MySQL Workbench interface. The SQL editor contains a query for cohort analysis. The output grid displays the results of the query, showing columns for year, num_of_users, and user_growth.

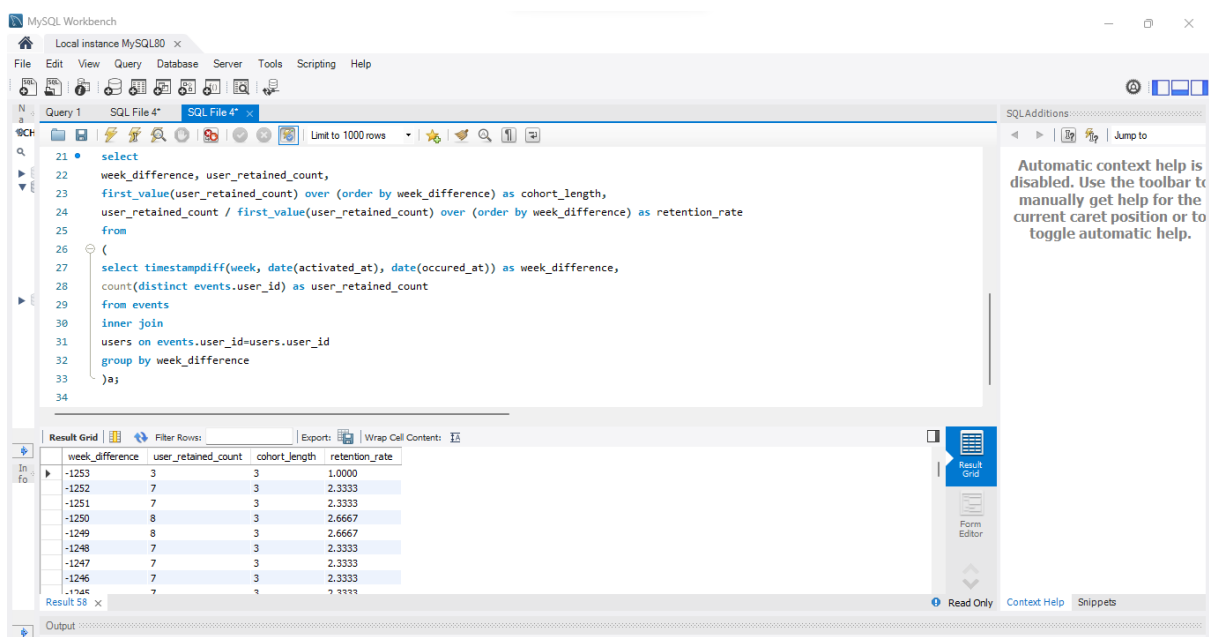
```
11 GROUP BY week_of_the_year;
12
13 # Growth of users over time for a product
14 • select * from users;
15 • select * from events;
16 • select *,
17   num_of_users - lag(num_of_users) over (order by year) as user_growth
18   from
19   (
20     select year(created_at) as year, count(user_id) as num_of_users from users group by year, user_id
21   ) a ;
22
23 # weekly retention of users based on their sign-up cohort
24
```

year	num_of_users	user_growth
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0
2013	1	0

3. Retention of users on a weekly basis after signing up for a product

Here we calculated Cohort Analysis which is an analysis of multiple cohorts or group of people with the objective of getting a deeper understanding of user behaviors, market trends, etc. It is a method of analyzing a metric by comparing its behavior between different groups of users.

SQL Query With Output



The screenshot shows the MySQL Workbench interface. The SQL editor contains a query for cohort analysis. The output grid displays the results of the query, showing columns for week_difference, user_retained_count, cohort_length, and retention_rate.

```
21 • select
22   week_difference, user_retained_count,
23   first_value(user_retained_count) over (order by week_difference) as cohort_length,
24   user_retained_count / first_value(user_retained_count) over (order by week_difference) as retention_rate
25   from
26   (
27     select timestampdiff(week, date(activated_at), date(occured_at)) as week_difference,
28     count(distinct events.user_id) as user_retained_count
29   from events
30   inner join
31   users on events.user_id=users.user_id
32   group by week_difference
33   ) a;
34
```

week_difference	user_retained_count	cohort_length	retention_rate
-1253	3	3	1.0000
-1252	7	3	2.3333
-1251	7	3	2.3333
-1250	8	3	2.6667
-1249	8	3	2.6667
-1248	7	3	2.3333
-1247	7	3	2.3333
-1246	7	3	2.3333
-1245	7	3	2.3333

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

Below is the list of devices with number of users using them weekly. User engagement seems good.

	device_name	week_of_engagement	num_of_active_devices	num_of_users
▶	macbook pro	17	2	2
	asus chromebook	17	8	8
	macbook pro	17	14	14
	mac mini	17	22	22
	macbook air	17	15	15
	iphone 5	17	5	5
	nexus 10	17	4	4
	ipad air	17	31	31
	lenovo thinkpad	17	8	8
	hp pavilion desktop	17	9	9
	lenovo thinkpad	17	14	14
	macbook pro	17	4	4
	iphone 5	17	16	16
	dell inspiron noteb...	17	5	5
	macbook pro	17	5	5
	iphone 5	17	7	7
	samsung galaxy s4	17	8	8
	macbook air	17	3	3
	samsung galaxy s4	17	5	5
	asus chromebook	17	8	8

Result 74 x

SQL Query With Output

The screenshot shows the MySQL Workbench interface. The top toolbar includes icons for File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The main window is titled 'Query 1' and contains the following SQL code:

```
36 group by week_difference
37 );
38
39 # weekly engagement per device
40 select * from events;
41
42 SELECT
43     device as device_name, WEEK(date(occurred_at)) AS week_of_engagement,
44     COUNT(device) AS num_of_active_devices,
45     COUNT(user_id) AS num_of_users
46 FROM
47     events
48 group by week_of_engagement, user_id, device;
49
50 # email engagement metrics
51
```

The bottom panel displays the 'Result Grid' with the following data:

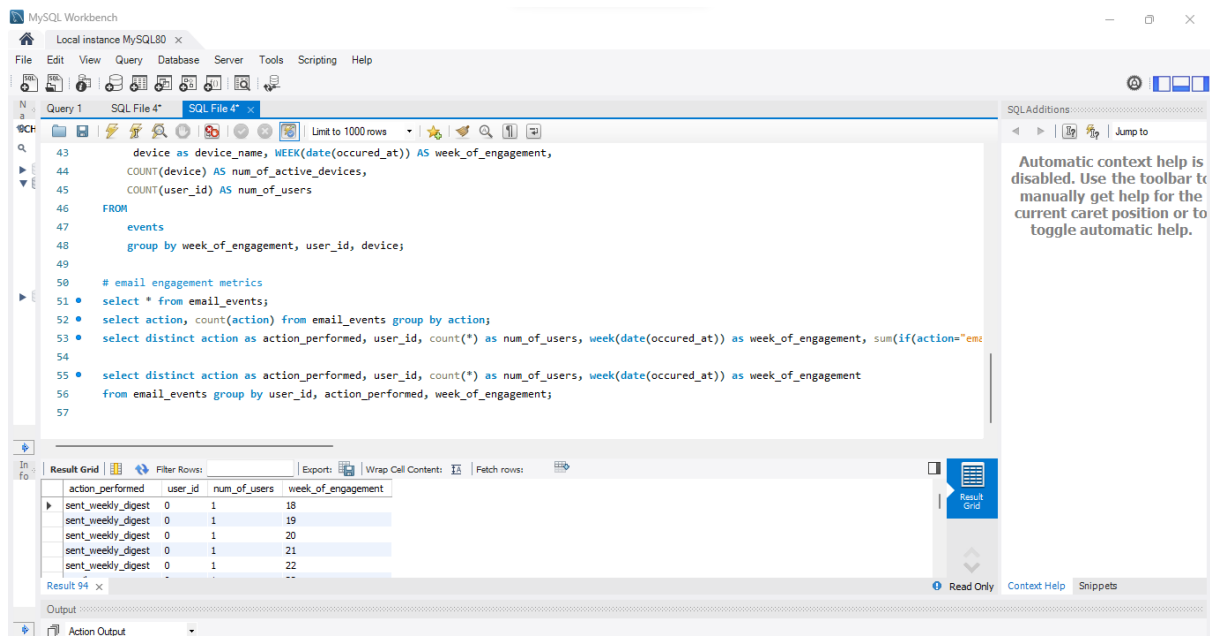
device_name	week_of_engagement	num_of_active_devices	num_of_users
macbook pro	17	2	2
asus chromebook	17	8	8
macbook pro	17	14	14
mac mini	17	22	22
macbook air	17	15	15
iphone 5	17	5	5

The right sidebar shows a 'SQLAdditions' panel with a message: 'Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.'

5. How users are engaging with the email service

Below is the list of users and the action taken by them weekly along with number of times each action is performed. Overall email engagements seems to be increasing.

SQL Query With Output

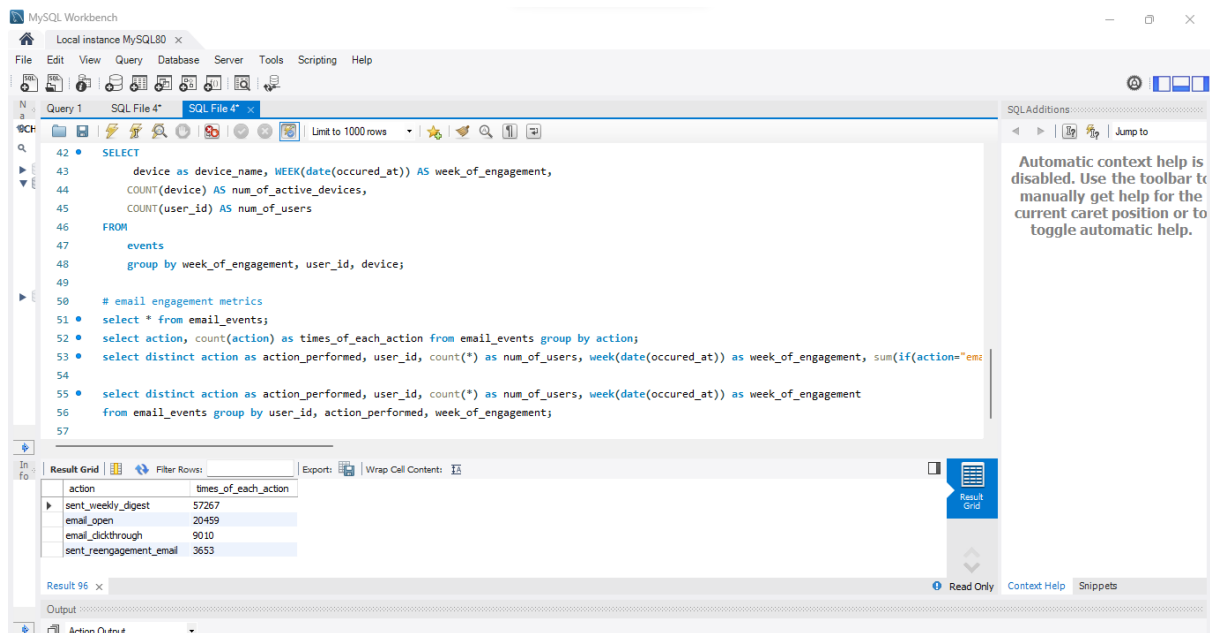


The screenshot shows the MySQL Workbench interface with a SQL query in the editor and its output in the Result Grid. The query is as follows:

```
43 device as device_name, WEEK(date(occured_at)) AS week_of_engagement,  
44 COUNT(device) AS num_of_active_devices,  
45 COUNT(user_id) AS num_of_users  
46 FROM  
47 events  
48 group by week_of_engagement, user_id, device;  
49  
50 # email engagement metrics  
51 select * from email_events;  
52 select action, count(action) from email_events group by action;  
53 select distinct action as action_performed, user_id, count(*) as num_of_users, week(date(occured_at)) as week_of_engagement, sum(if(action="sent_weekly_digest", 1, 0)) as num_of_weekly_digests  
54  
55 select distinct action as action_performed, user_id, count(*) as num_of_users, week(date(occured_at)) as week_of_engagement  
56 from email_events group by user_id, action_performed, week_of_engagement;  
57
```

The Result Grid shows the output of the query, with columns: action_performed, user_id, num_of_users, and week_of_engagement. The data is as follows:

action_performed	user_id	num_of_users	week_of_engagement
sent_weekly_digest	0	1	18
sent_weekly_digest	0	1	19
sent_weekly_digest	0	1	20
sent_weekly_digest	0	1	21
sent_weekly_digest	0	1	22



The screenshot shows the MySQL Workbench interface with a SQL query in the editor and its output in the Result Grid. The query is as follows:

```
42 select  
43 device as device_name, WEEK(date(occured_at)) AS week_of_engagement,  
44 COUNT(device) AS num_of_active_devices,  
45 COUNT(user_id) AS num_of_users  
46 FROM  
47 events  
48 group by week_of_engagement, user_id, device;  
49  
50 # email engagement metrics  
51 select * from email_events;  
52 select action, count(action) as times_of_each_action from email_events group by action;  
53 select distinct action as action_performed, user_id, count(*) as num_of_users, week(date(occured_at)) as week_of_engagement, sum(if(action="sent_weekly_digest", 1, 0)) as num_of_weekly_digests  
54  
55 select distinct action as action_performed, user_id, count(*) as num_of_users, week(date(occured_at)) as week_of_engagement  
56 from email_events group by user_id, action_performed, week_of_engagement;  
57
```

The Result Grid shows the output of the query, with columns: action, and times_of_each_action. The data is as follows:

action	times_of_each_action
sent_weekly_digest	57267
email_open	20459
email_clickthrough	9010
sent_reengagement_email	3653

Result

This project is really engaging and at the same time difficult too. It not only level up my understanding about window functions in SQL but also introduced me to terms like cohort analysis, rolling average. It does gave me the confidence in using window functions for future.

Note:

It's a request to please include a explanation of the each task so that it becomes clear that what insight we should be looking for in each task.