TRAINITY PROJECT 3: DATA ANALYTICS PROCESS

TOPIC: Operation Analytics and Investigating Metric Spike

PROJECT DESCRIPTION:

Operational Analytics is a crucial process that involves analyzing a company's end-to-end operations. This analysis helps identify areas for improvement within the company. As a Data Analyst, one needs to work closely with various teams, such as operations, support, and marketing, helping them derive valuable insights from the data they collect. One of the key aspects of Operational Analytics is investigating metric spikes. This involves understanding and explaining sudden changes in key metrics, such as a dip in daily user engagement or a drop in sales. As a Data Analyst, one needs to answer these questions daily, making it crucial to understand how to investigate these metric spikes.

In this project, with various datasets and tables, task is to derive insights from this data to answer questions posed by different departments within the company. Here advanced SQL skills is used to analyze the data and provide valuable insights that can help improve the company's operations and understand sudden changes in key metrics.

APPROACH:

Create a Database: The database file was provided in the attachments. The necessay tables were created using the provided table structures and links. The csv file was imported into MySQL Workbench.

TECH- STACK USED:

The software I am using to do the analysis is MySQL Workbench, Version: 8.0.34 (MySQL Community Server – GPL) and Microsoft Excel to prepare data. I used these two softwares for the project as it is easy to use and freely available.

INSIGHTS:

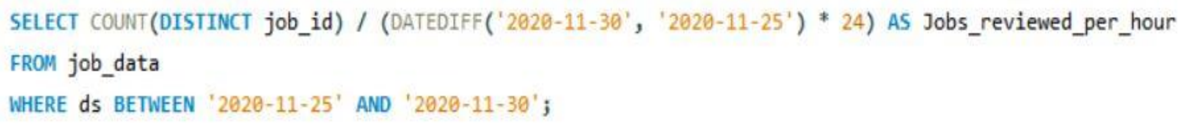
Case Study 1: Job Data Analysis

A table named job_data with the following columns was created:

- **job_id:** Unique identifier of jobs
- **actor_id:** Unique identifier of actor
- **event:** The type of event (decision/skip/transfer).
- **language:** The Language of the content
- **time_spent:** Time spent to review the job in seconds.
- **org:** The Organization of the actor
- **ds:** The date in the format yyyy/mm/dd (stored as text).

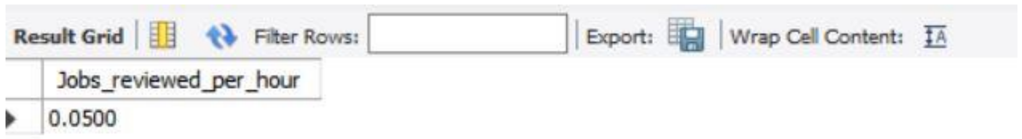
Tasks:

A. **Jobs Reviewed Over Time:**
   - Objective: Calculate the number of jobs reviewed per hour for each day in November 2020.
   - Task: Write an SQL query to calculate the number of jobs reviewed per hour for each day in November 2020.



```sql
SELECT COUNT(DISTINCT job_id) / (DATEDIFF('2020-11-30', '2020-11-25') * 24) AS Jobs_reviewed_per_hour
FROM job_data
WHERE ds BETWEEN '2020-11-25' AND '2020-11-30';
```

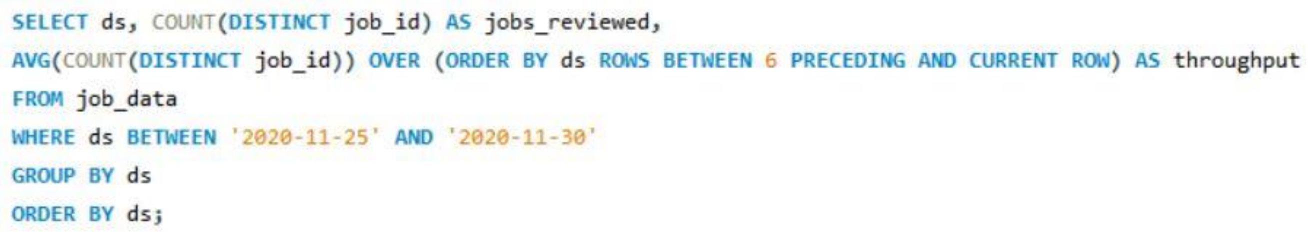
The SQL Query calculates the number of jobs reviewed per day per hour from date 25-11-2020 to 30-11-2020 from the table job_data.



Jobs_reviewed_per_hour
0.0500

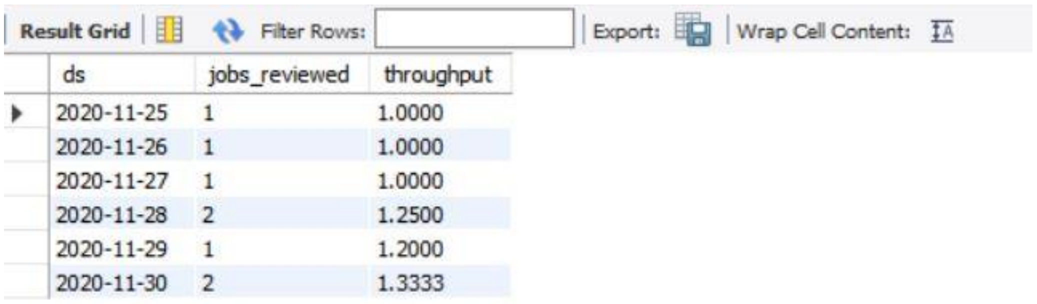
Only 0.5 jobs were reviewed per day per hour from date 25-11-2020 to 30-11-2020

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



```sql
SELECT ds, COUNT(DISTINCT job_id) AS jobs_reviewed,
AVG(COUNT(DISTINCT job_id)) OVER (ORDER BY ds ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS throughput
FROM job_data
WHERE ds BETWEEN '2020-11-25' AND '2020-11-30'
GROUP BY ds
ORDER BY ds;
```

We directly calculate the count of distinct job IDs reviewed (**jobs_reviewed**) and use it within the window function **AVG()** to calculate the 7-day rolling average. We utilize **ORDER BY ds** to ensure the rows are ordered by date. The **ROWS BETWEEN 6 PRECEDING AND CURRENT ROW** clause specifies the window size as the current row plus the previous 6 rows, enabling the calculation of the 7-day rolling average. The **GROUP BY ds** groups the data by date. The **ORDER BY ds** orders the results by date.



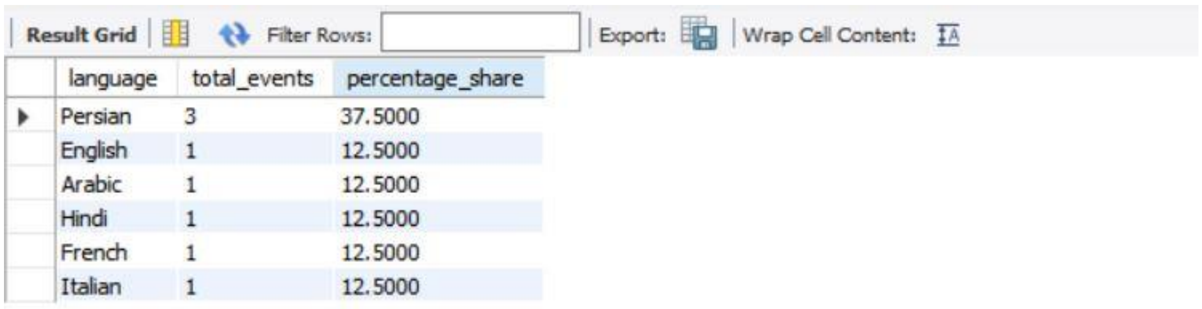
ds	jobs_reviewed	throughput
2020-11-25	1	1.0000
2020-11-26	1	1.0000
2020-11-27	1	1.0000
2020-11-28	2	1.2500
2020-11-29	1	1.2000
2020-11-30	2	1.3333

The 7-day rolling average of throughput is maximum on 30-11-2020.
Also preference either to use the daily metric or the 7-day rolling average depends on granularity needed. Daily metrics are for short-term insights; 7-day rolling average is for smoother, long-term trends. It helps to highlight longer-term trends and patterns while reducing the impact of daily variations.

C. **Language Share Analysis:**
- Objective: Calculate the percentage share of each language in the last 30 days.
- Task: Write an SQL query to calculate the percentage share of each language over the last 30 days.

```
SELECT
    language,
    COUNT(*) AS total_events,
    (COUNT(*) / (SELECT COUNT(*) FROM job_data WHERE ds BETWEEN '2020-11-25' AND '2020-11-30')) * 100 AS percentage_share
FROM
    job_data
WHERE
    ds BETWEEN '2020-11-25' AND '2020-11-30'
GROUP BY
    language
ORDER BY
    percentage_share DESC;
```

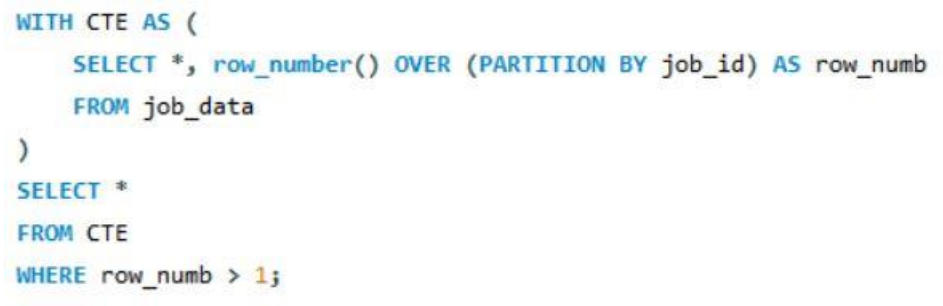
We use COUNT (*) to calculate the total number of events for each language, which is consistent throughout the query. The subquery (SELECT COUNT(*) FROM job_data WHERE ds BETWEEN ‘2020-11-25’ AND ‘2020-11-30’)) * 100 retrieves the total count of events which is then used to calculate the percentage share.
The percentage share calculation is accurate, as it divides the count of events for each language by the total count of events.



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

So, the percentage share of Persian is more as compared to other languages in the given time frame.

D. **Duplicate Rows Detection:**
   - Objective: Identify duplicate rows in the data.
   - Task: Write an SQL query to display duplicate rows from the job_data table.

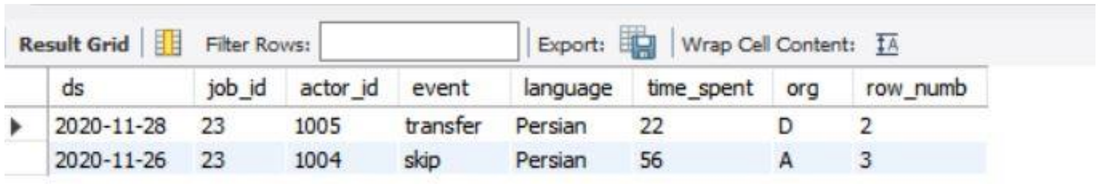


```sql
WITH CTE AS (
    SELECT *, row_number() OVER (PARTITION BY job_id) AS row_numb
    FROM job_data
)
SELECT *
FROM CTE
WHERE row_numb > 1;
```

The WITH clause is used to define a Common Table Expression (CTE) named CTE. Inside the CTE, we select all columns (*) from the job_data table. We use the **ROW_NUMBER()** window function combined with the OVER clause to assign a unique row number to each row within each partition defined by the job_id column. The **PARTITION BY** job_id clause partitions the rows of the result set into groups based on the job_id column. This means that the row number is reset for each distinct job_id. The assigned row number is aliased as **row_numb** in the result set.

Selecting Duplicate Rows from the CTE:

In this part, we select all columns (*) from the CTE named CTE. We filter the rows by specifying WHERE row_numb > 1. This condition ensures that only rows with a row number greater than 1 are selected. Rows with a row number greater than 1 indicate duplicates because they have more than one occurrence within the partition defined by the job_id.



ds	job_id	actor_id	event	language	time_spent	org	row_numb
2020-11-28	23	1005	transfer	Persian	22	D	2
2020-11-26	23	1004	skip	Persian	56	A	3

There were two duplicate rows of job_id as depicted above in the executed query.

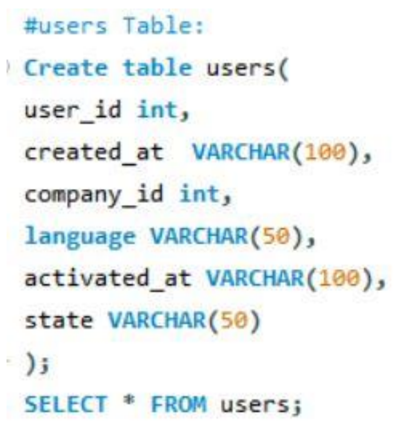
Case Study 2: Investigating Metric Spike

There are three tables:

- **users**: Contains one row per user, with descriptive information about that user's account.
- **events**: Contains one row per event, where an event is an action that a user has taken (e.g., login, messaging, search).
- **email_events**: Contains events specific to the sending of emails.

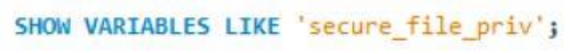
Before moving on to do the analysis, we first load the data from excel to MySQL Workbench. Though data contains millions of records, importing through “Table Data Import Wizard” will take hours to import. For quick importing the data, we use “LOAD DATA INFILE” command which helps in loading large datasets within seconds.

First we create a Table users:

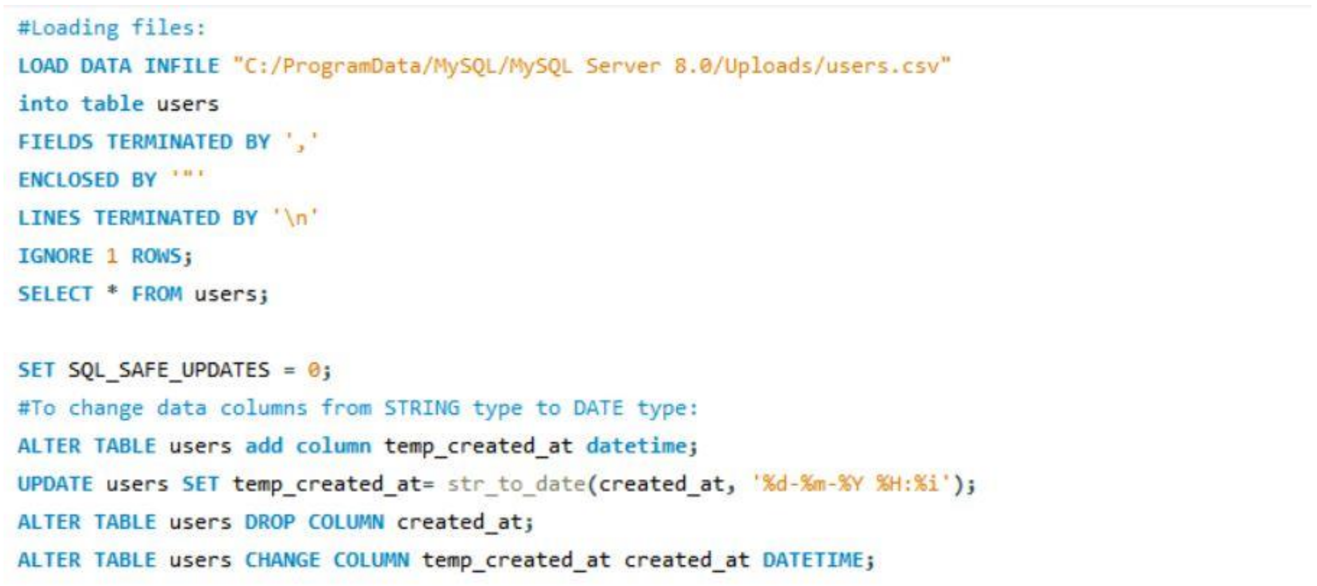


```
#users Table:
Create table users(
user_id int,
created_at  VARCHAR(100),
company_id int,
language VARCHAR(50),
activated_at VARCHAR(100),
state VARCHAR(50)
);
SELECT * FROM users;
```

Executing this command will show the path where the datasets that need to imported to be kept.

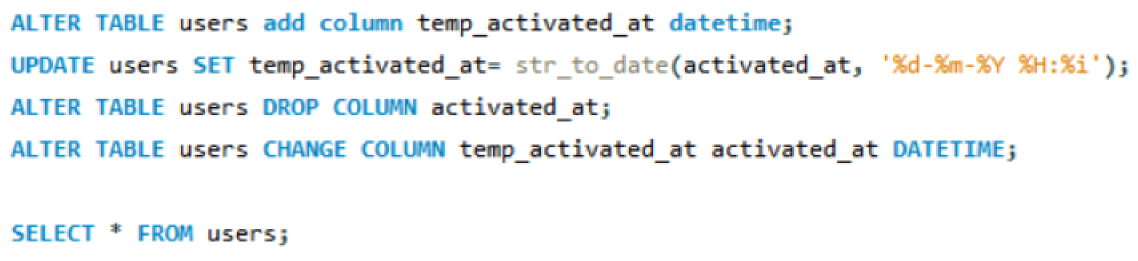


```
SHOW VARIABLES LIKE 'secure_file_priv';
```



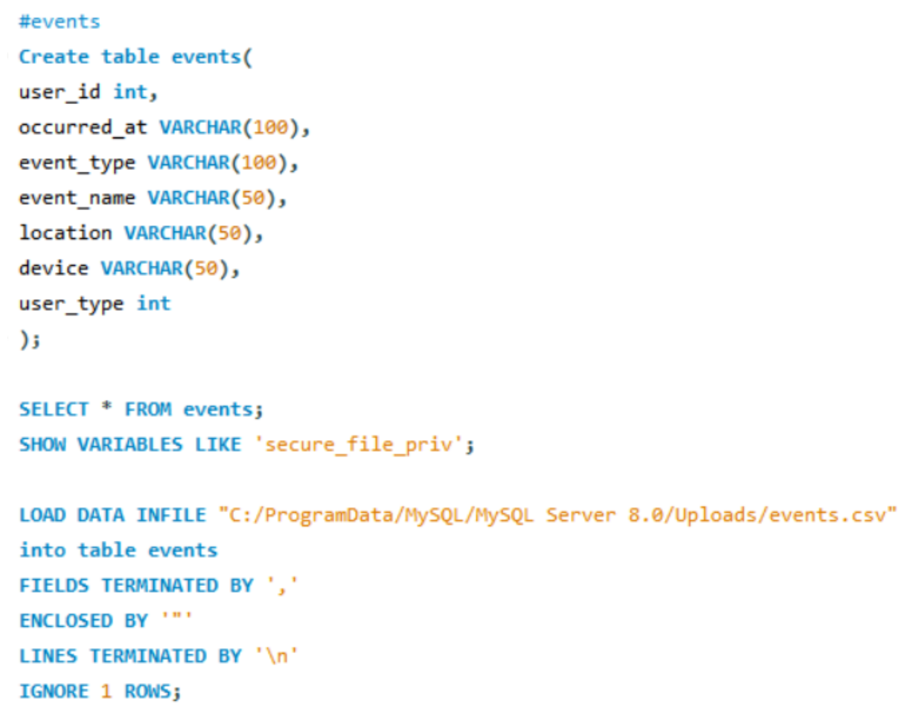
```
#Loading files:
LOAD DATA INFILE "C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/users.csv"
into table users
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
SELECT * FROM users;

SET SQL_SAFE_UPDATES = 0;
#To change data columns from STRING type to DATE type:
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;
```

In the dataset, the date column was in STRING format. For effctive execution of queries, we need to convert it into date type format using **str_to_date.**

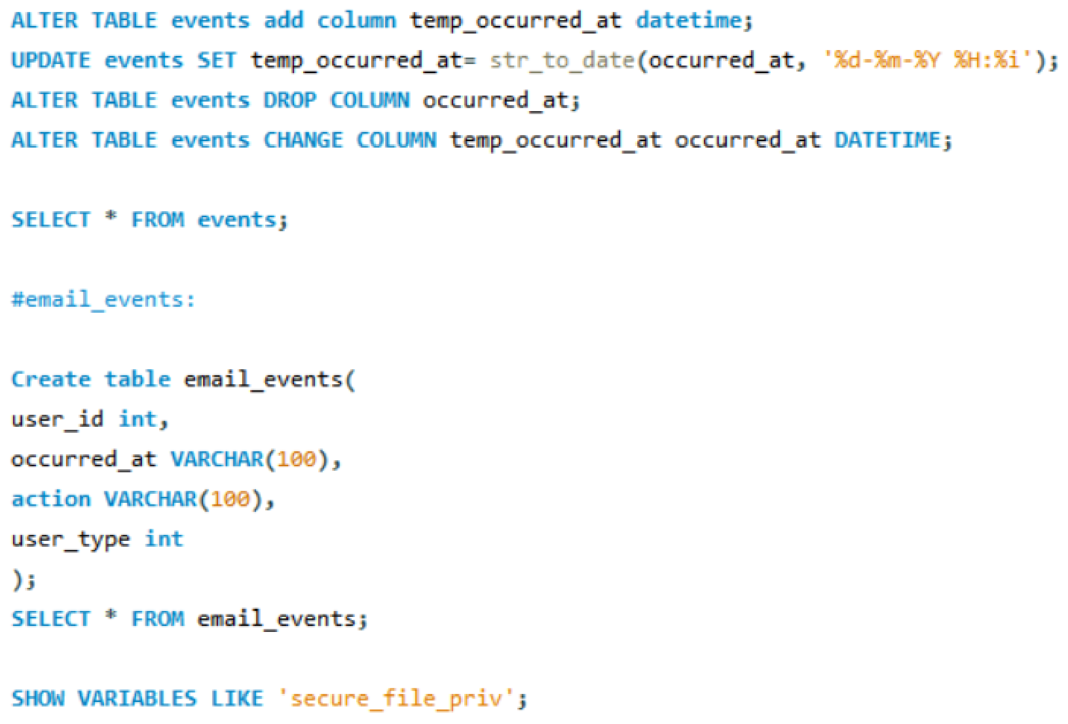


```
ALTER TABLE users add column temp_activated_at datetime;
UPDATE users SET temp_activated_at= str_to_date(activated_at, '%d-%m-%Y %H:%i');
ALTER TABLE users DROP COLUMN activated_at;
ALTER TABLE users CHANGE COLUMN temp_activated_at activated_at DATETIME;

SELECT * FROM users;
```

Similarly, events table and email_events table were created and data were imported.



```
#events
Create table events(
user_id int,
occurred_at VARCHAR(100),
event_type VARCHAR(100),
event_name VARCHAR(50),
location VARCHAR(50),
device VARCHAR(50),
user_type int
);

SELECT * FROM events;
SHOW VARIABLES LIKE 'secure_file_priv';

LOAD DATA INFILE "C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/events.csv"
into table events
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```



```
ALTER TABLE events add column temp_occurred_at datetime;
UPDATE events SET temp_occurred_at= str_to_date(occurred_at, '%d-%m-%Y %H:%i');
ALTER TABLE events DROP COLUMN occurred_at;
ALTER TABLE events CHANGE COLUMN temp_occurred_at occurred_at DATETIME;

SELECT * FROM events;

#email_events:

Create table email_events(
user_id int,
occurred_at VARCHAR(100),
action VARCHAR(100),
user_type int
);
SELECT * FROM email_events;

SHOW VARIABLES LIKE 'secure_file_priv';
```

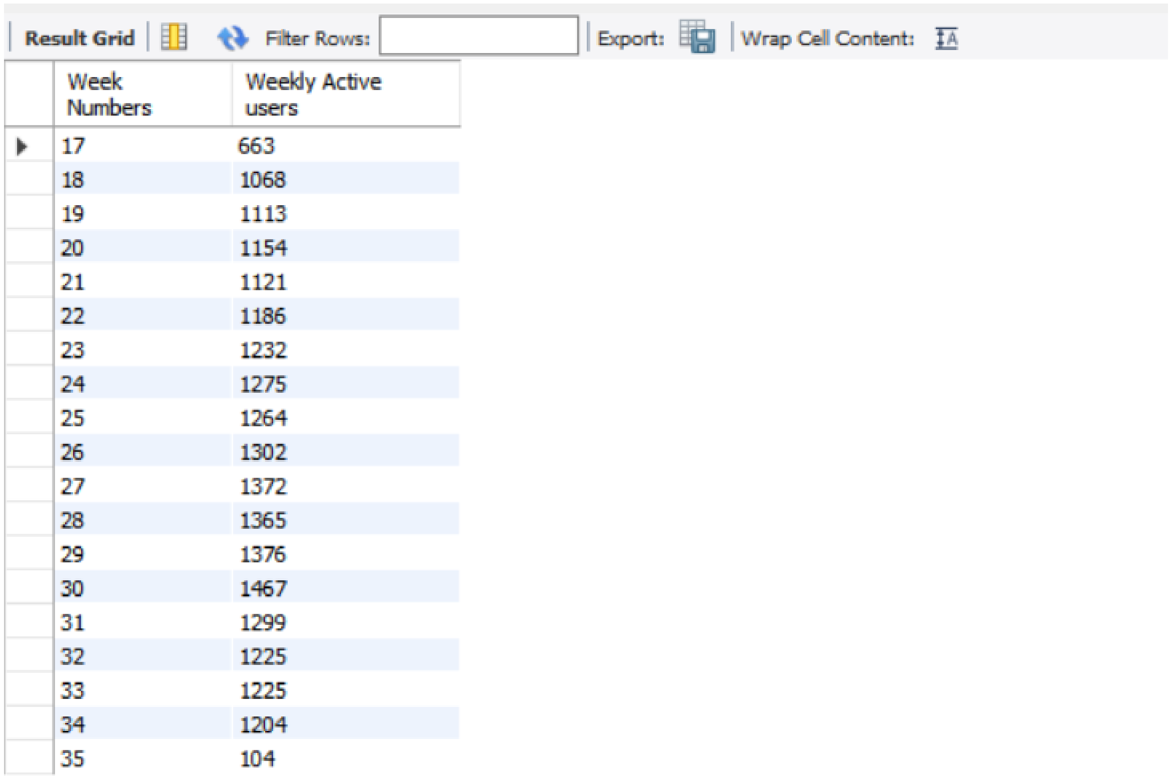
```
LOAD DATA INFILE "C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/email_events.csv"
into table email_events
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
SELECT * FROM email_events;

SET SQL_SAFE_UPDATES = 0;
ALTER TABLE email_events add column temp_occurred_at datetime;
UPDATE email_events SET temp_occurred_at= str_to_date(occurred_at, '%d-%m-%Y %H:%i');
ALTER TABLE email_events DROP COLUMN occurred_at;
ALTER TABLE email_events CHANGE COLUMN temp_occurred_at occurred_at DATETIME;
```

Tasks:

A. **Weekly User Engagement:**
   - Objective: Measure the activeness of users on a weekly basis.
   - Task: Write an SQL query to calculate the weekly user engagement.

```
SELECT EXTRACT(WEEK FROM occurred_at) AS "Week Numbers", COUNT(DISTINCT user_id) as "Weekly Active users"
FROM events
WHERE event_type= "engagement"
GROUP BY 1;
```

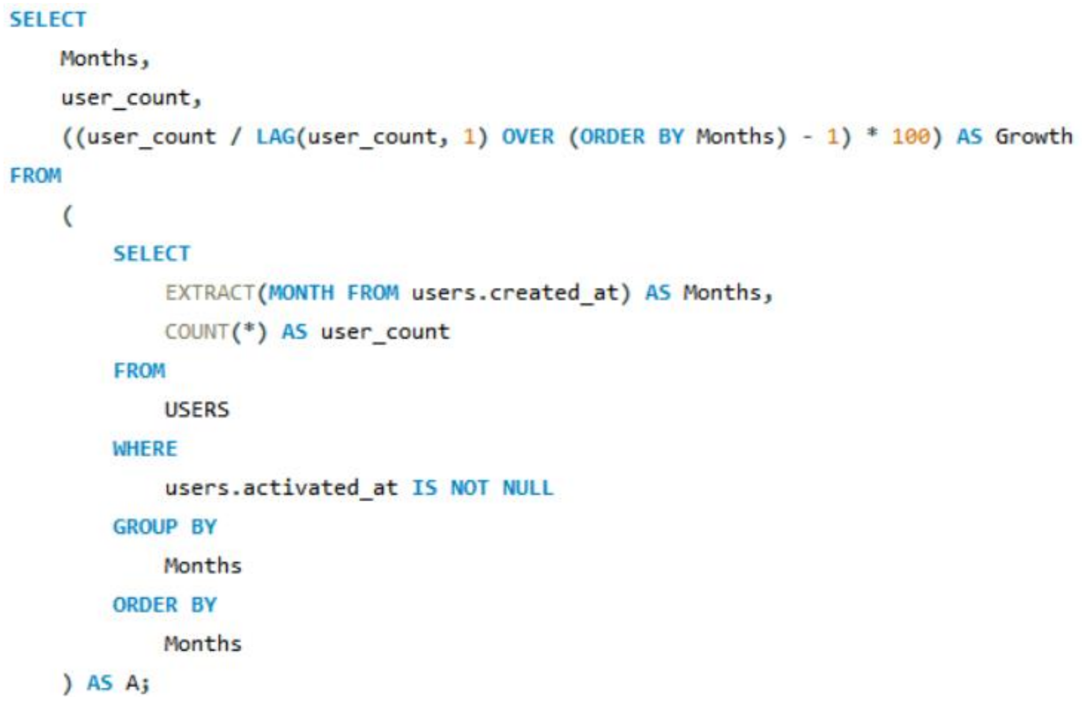
We use the EXTRACT (WEEK FROM occurred_at) function to extract the week number from the occurred_at column. The COUNT (DISTINCT user_id) function calculates the number of distinct users for each week. We filter the events based on the event_type being 'engagement'. Finally, we group the results by the week number. This query will give you the count of weekly active users based on the 'engagement' events.



Week Numbers	Weekly Active users
17	663
18	1068
19	1113
20	1154
21	1121
22	1186
23	1232
24	1275
25	1264
26	1302
27	1372
28	1365
29	1376
30	1467
31	1299
32	1225
33	1225
34	1204
35	104

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.

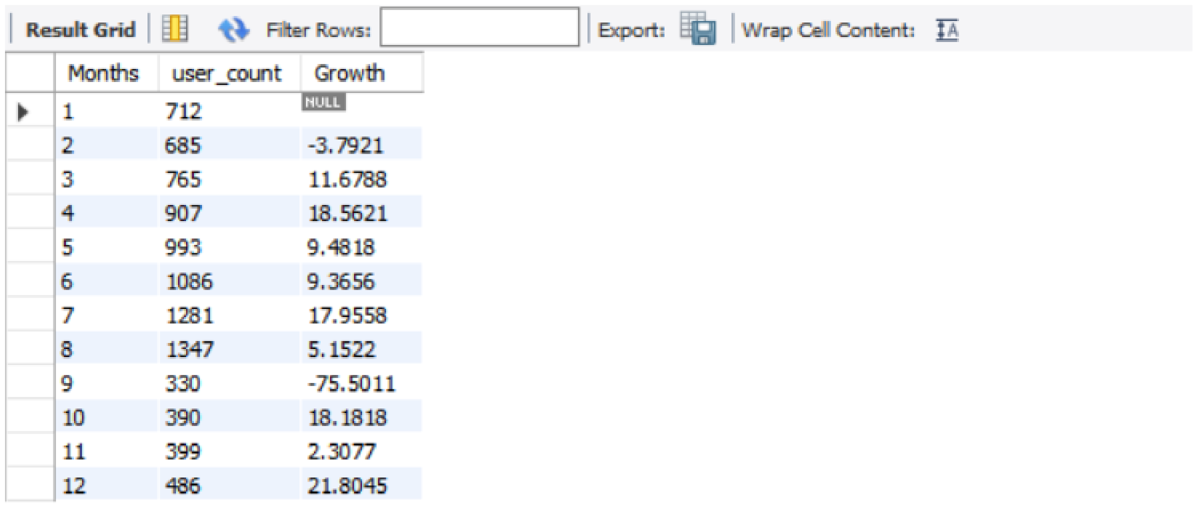


```
SELECT
    Months,
    user_count,
    ((user_count / LAG(user_count, 1) OVER (ORDER BY Months) - 1) * 100) AS Growth
FROM
    (
        SELECT
            EXTRACT(MONTH FROM users.created_at) AS Months,
            COUNT(*) AS user_count
        FROM
            USERS
        WHERE
            users.activated_at IS NOT NULL
        GROUP BY
            Months
        ORDER BY
            Months
    ) AS A;
```

Subquery to Calculate User Count by Month:
This subquery retrieves data from the USERS table. It calculates the month component from the created_at timestamp column using the **EXTRACT()** function and aliases it as Months. It counts the number of users **(COUNT(*))** in each month and aliases it as **user_count**. The WHERE clause filters out rows where the activated_at column is not null, indicating activated users. Results are grouped by the Months column and ordered by Months.

Main Query to Calculate User Growth:
In the main query, we select columns from the subquery. We select Months and user_count directly from the subquery. For calculating growth, we use the **LAG()** window function to retrieve the user count from the previous month. **LAG(user_count, 1) OVER (ORDER BY Months)** retrieves the user count from the previous row, ordered by Months. We calculate **growth as ((user_count / PreviousMonthUserCount) - 1) * 100.**

The results are presented as Months, user_count, and Growth.



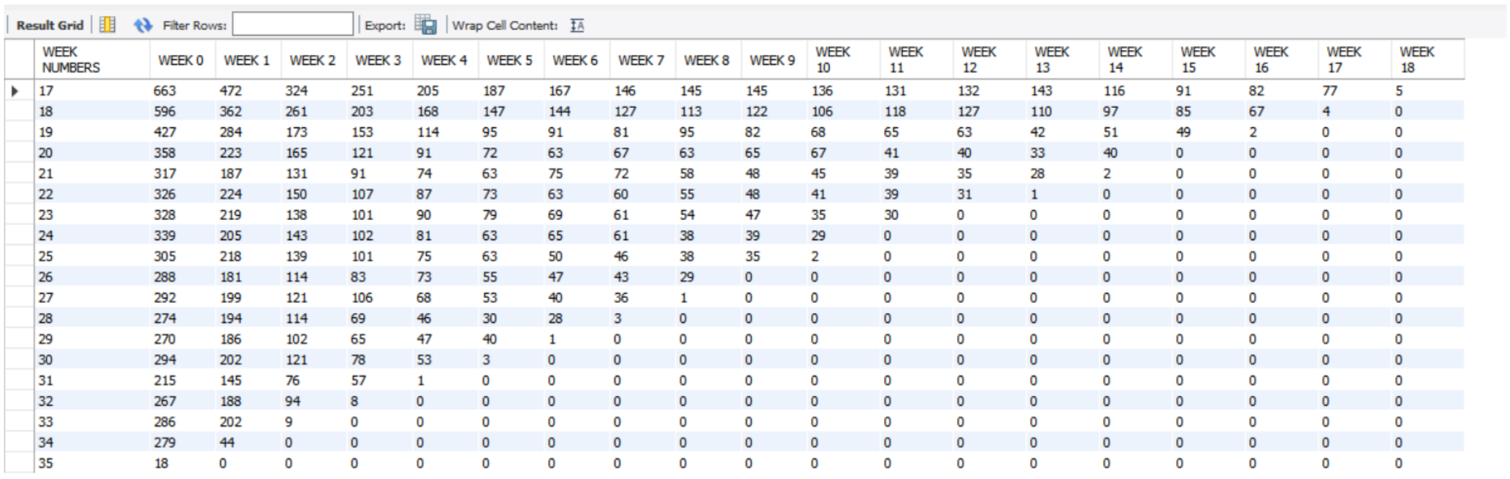
Months	user_count	Growth
1	712	NULL
2	685	-3.7921
3	765	11.6788
4	907	18.5621
5	993	9.4818
6	1086	9.3656
7	1281	17.9558
8	1347	5.1522
9	330	-75.5011
10	390	18.1818
11	399	2.3077
12	486	21.8045

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

Subquery to Calculate Weekly Logins and First Week:
This subquery selects the user_id, login week, and the first week of login for each user.
In the first part of the subquery (A), it selects the user_id and the week number of logins (LOGIN_WEEK) from the events table using the EXTRACT(WEEK FROM OCCURRED_AT) function. It groups the results by user_id and LOGIN_WEEK.
In the second part of the subquery (B), it selects the user_id and calculates the minimum week number of login (FIRST) for each user. The subquery calculates the difference between the login week and the first week (LOGIN_WEEK - FIRST) to determine the week number for each login event.
It then filters the results to include only the rows where the user_id from subquery A matches the user_id from subquery B.

Main Query to Pivot Weekly Retention:
The main query pivots the data to calculate weekly retention for each cohort.
It uses conditional aggregation with SUM (CASE ...) statements to count the number of users who logged in during each week (WEEK 0, WEEK 1, ..., WEEK 18).
The CASE statements check if the week number equals the respective week (WEEK_NUMBER = 0, WEEK_NUMBER = 1, etc.), and if so, it adds 1 to the sum; otherwise, it adds 0.
The data is grouped by the first login week (FIRST), representing the user cohort.
The results are ordered by the first login week.
This query effectively calculates the weekly retention of users based on their sign-up cohort. Adjustments may be needed based on your specific database schema and requirements.

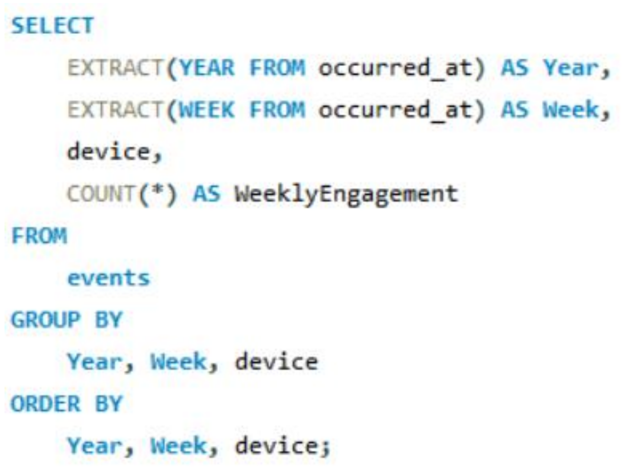
```
SELECT FIRST AS "WEEK NUMBERS",
SUM(CASE WHEN WEEK_NUMBER = 0 THEN 1 ELSE 0 END) AS "WEEK 0",
SUM(CASE WHEN WEEK_NUMBER = 1 THEN 1 ELSE 0 END) AS "WEEK 1",
SUM(CASE WHEN WEEK_NUMBER = 2 THEN 1 ELSE 0 END) AS "WEEK 2",
SUM(CASE WHEN WEEK_NUMBER = 3 THEN 1 ELSE 0 END) AS "WEEK 3",
SUM(CASE WHEN WEEK_NUMBER = 4 THEN 1 ELSE 0 END) AS "WEEK 4",
SUM(CASE WHEN WEEK_NUMBER = 5 THEN 1 ELSE 0 END) AS "WEEK 5",
SUM(CASE WHEN WEEK_NUMBER = 6 THEN 1 ELSE 0 END) AS "WEEK 6",
SUM(CASE WHEN WEEK_NUMBER = 7 THEN 1 ELSE 0 END) AS "WEEK 7",
SUM(CASE WHEN WEEK_NUMBER = 8 THEN 1 ELSE 0 END) AS "WEEK 8",
SUM(CASE WHEN WEEK_NUMBER = 9 THEN 1 ELSE 0 END) AS "WEEK 9",
SUM(CASE WHEN WEEK_NUMBER = 10 THEN 1 ELSE 0 END) AS "WEEK 10",
SUM(CASE WHEN WEEK_NUMBER = 11 THEN 1 ELSE 0 END) AS "WEEK 11",
SUM(CASE WHEN WEEK_NUMBER = 12 THEN 1 ELSE 0 END) AS "WEEK 12",
SUM(CASE WHEN WEEK_NUMBER = 13 THEN 1 ELSE 0 END) AS "WEEK 13",
SUM(CASE WHEN WEEK_NUMBER = 14 THEN 1 ELSE 0 END) AS "WEEK 14",
SUM(CASE WHEN WEEK_NUMBER = 15 THEN 1 ELSE 0 END) AS "WEEK 15",
SUM(CASE WHEN WEEK_NUMBER = 16 THEN 1 ELSE 0 END) AS "WEEK 16",
SUM(CASE WHEN WEEK_NUMBER = 17 THEN 1 ELSE 0 END) AS "WEEK 17",
SUM(CASE WHEN WEEK_NUMBER = 18 THEN 1 ELSE 0 END) AS "WEEK 18"
FROM
(SELECT A.user_id, A.login_week, B.FIRST, A.login_week - FIRST AS WEEK_NUMBER
FROM
(SELECT user_id, EXTRACT(WEEK FROM OCCURRED_AT) AS LOGIN_WEEK FROM events
GROUP BY 1,2)A,
(SELECT user_id, MIN(EXTRACT( WEEK FROM occurred_at)) AS FIRST FROM events
GROUP BY 1)B
WHERE A.user_id = B.user_id
)SUB
GROUP BY FIRST
ORDER BY FIRST;
```


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

WEEK NUMBERS	WEEK 0	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9	WEEK 10	WEEK 11	WEEK 12	WEEK 13	WEEK 14	WEEK 15	WEEK 16	WEEK 17	WEEK 18
17	663	472	324	251	205	187	167	146	145	145	136	131	132	143	116	91	82	77	5
18	596	362	261	203	168	147	144	127	113	122	106	118	127	110	97	85	67	4	0
19	427	284	173	153	114	95	91	81	95	82	68	65	63	42	51	49	2	0	0
20	358	223	165	121	91	72	63	67	63	65	67	41	40	33	40	0	0	0	0
21	317	187	131	91	74	63	75	72	58	48	45	39	35	28	2	0	0	0	0
22	326	224	150	107	87	73	63	60	55	48	41	39	31	1	0	0	0	0	0
23	328	219	138	101	90	79	69	61	54	47	35	30	0	0	0	0	0	0	0
24	339	205	143	102	81	63	65	61	38	39	29	0	0	0	0	0	0	0	0
25	305	218	139	101	75	63	50	46	38	35	2	0	0	0	0	0	0	0	0
26	288	181	114	83	73	55	47	43	29	0	0	0	0	0	0	0	0	0	0
27	292	199	121	106	68	53	40	36	1	0	0	0	0	0	0	0	0	0	0
28	274	194	114	69	46	30	28	3	0	0	0	0	0	0	0	0	0	0	0
29	270	186	102	65	47	40	1	0	0	0	0	0	0	0	0	0	0	0	0
30	294	202	121	78	53	3	0	0	0	0	0	0	0	0	0	0	0	0	0
31	215	145	76	57	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	267	188	94	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	286	202	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	279	44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
35	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

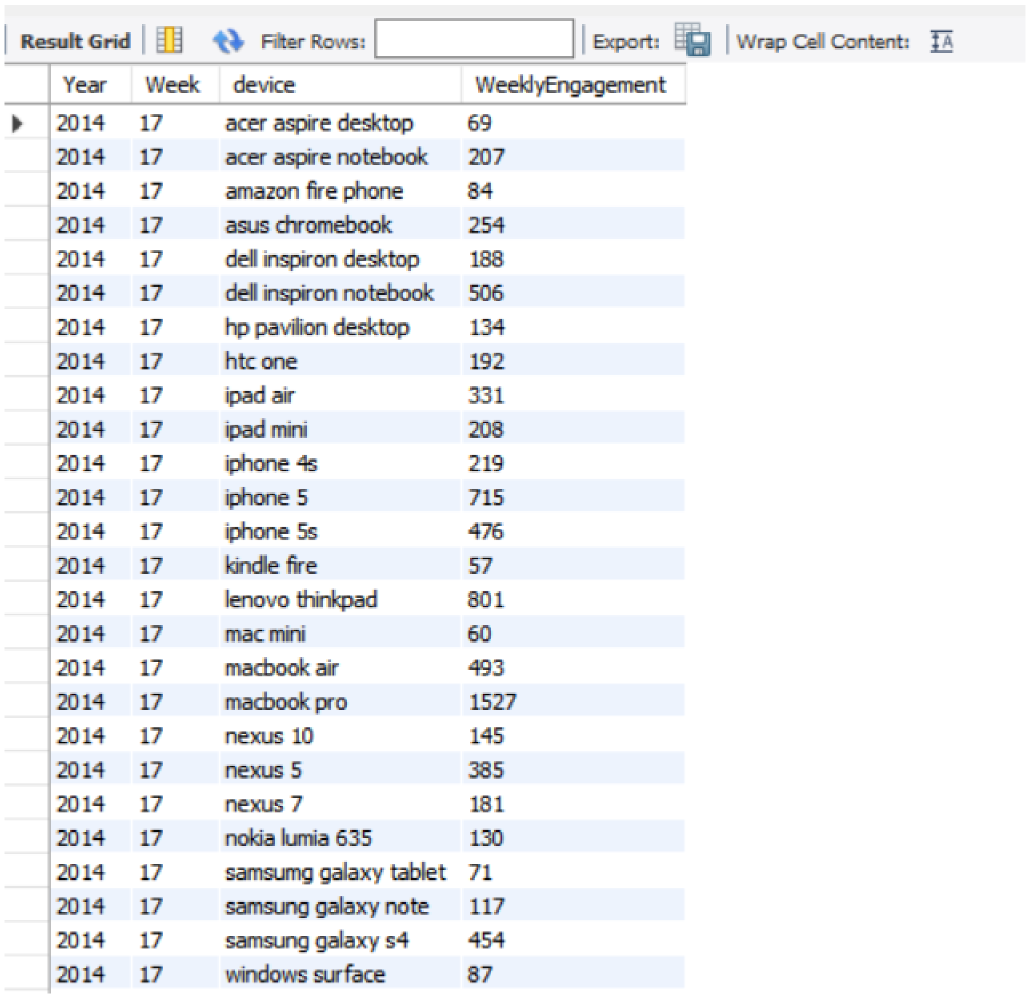
D. **Weekly Engagement Per Device:**

- Objective: Measure the activeness of users on a weekly basis per device.
- Task: Write an SQL query to calculate the weekly engagement per device.



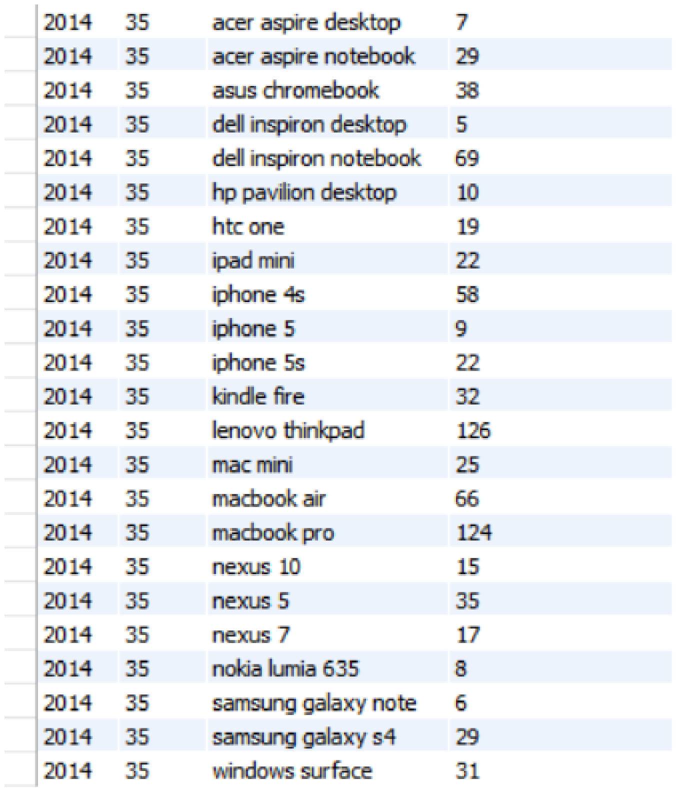
```
SELECT
    EXTRACT(YEAR FROM occurred_at) AS Year,
    EXTRACT(WEEK FROM occurred_at) AS Week,
    device,
    COUNT(*) AS WeeklyEngagement
FROM
    events
GROUP BY
    Year, Week, device
ORDER BY
    Year, Week, device;
```

- EXTRACT(YEAR FROM occurred_at) AS Year: Extracts the year component from the occurred_at timestamp column.
- EXTRACT(WEEK FROM occurred_at) AS Week: Extracts the week number component from the occurred_at timestamp column.
- device: Specifies the device used for the engagement.
- COUNT(*) AS WeeklyEngagement: Counts the number of engagements per week for each device.
- GROUP BY Year, Week, device: Groups the data by year, week, and device to calculate engagement statistics for each device per week.
- ORDER BY Year, Week, device: Orders the results by year, week, and device for better readability.


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

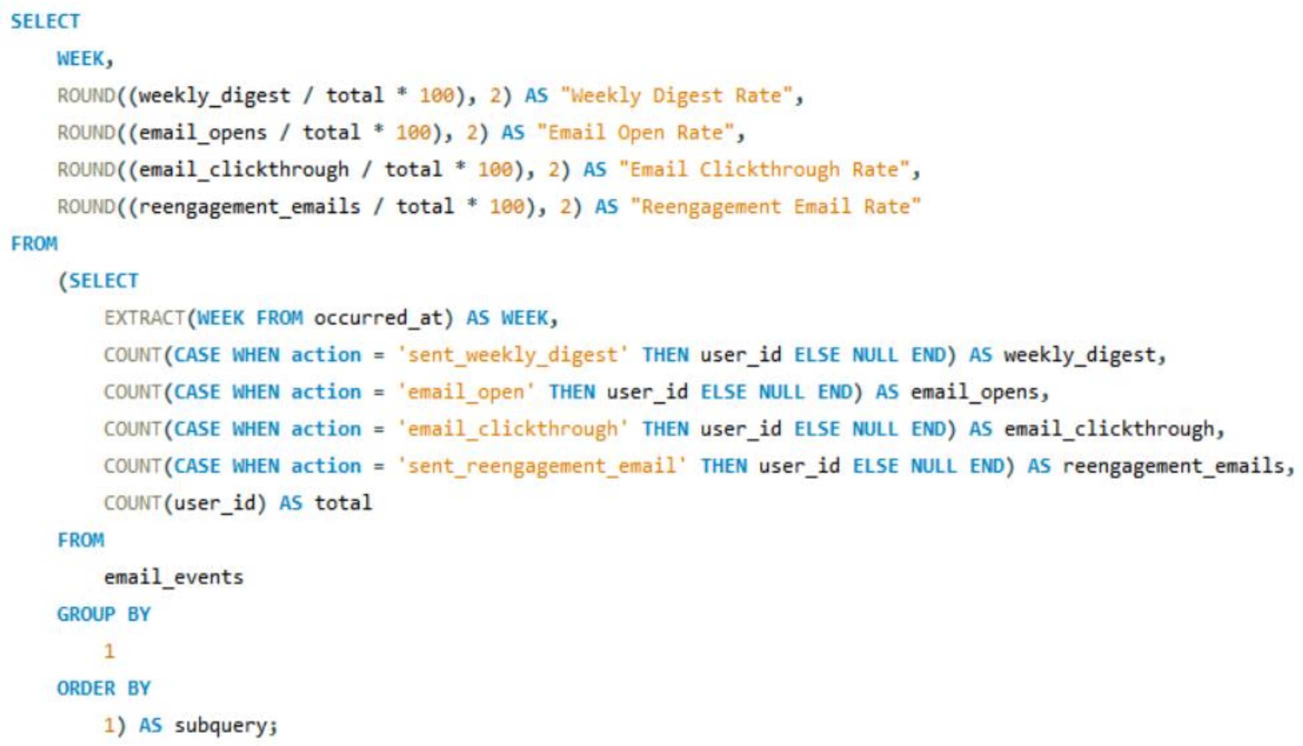
Year	Week	device	WeeklyEngagement
2014	17	acer aspire desktop	69
2014	17	acer aspire notebook	207
2014	17	amazon fire phone	84
2014	17	asus chromebook	254
2014	17	dell inspiron desktop	188
2014	17	dell inspiron notebook	506
2014	17	hp pavilion desktop	134
2014	17	htc one	192
2014	17	ipad air	331
2014	17	ipad mini	208
2014	17	iphone 4s	219
2014	17	iphone 5	715
2014	17	iphone 5s	476
2014	17	kindle fire	57
2014	17	lenovo thinkpad	801
2014	17	mac mini	60
2014	17	macbook air	493
2014	17	macbook pro	1527
2014	17	nexus 10	145
2014	17	nexus 5	385
2014	17	nexus 7	181
2014	17	nokia lumia 635	130
2014	17	samsumg galaxy tablet	71
2014	17	samsung galaxy note	117
2014	17	samsung galaxy s4	454
2014	17	windows surface	87

It goes till 35weeks.



2014	35	acer aspire desktop	7
2014	35	acer aspire notebook	29
2014	35	asus chromebook	38
2014	35	dell inspiron desktop	5
2014	35	dell inspiron notebook	69
2014	35	hp pavilion desktop	10
2014	35	htc one	19
2014	35	ipad mini	22
2014	35	iphone 4s	58
2014	35	iphone 5	9
2014	35	iphone 5s	22
2014	35	kindle fire	32
2014	35	lenovo thinkpad	126
2014	35	mac mini	25
2014	35	macbook air	66
2014	35	macbook pro	124
2014	35	nexus 10	15
2014	35	nexus 5	35
2014	35	nexus 7	17
2014	35	nokia lumia 635	8
2014	35	samsung galaxy note	6
2014	35	samsung galaxy s4	29
2014	35	windows surface	31

E. **Email Engagement Analysis:**
   - Objective: Analyze how users are engaging with the email service.
   - Task: Write an SQL query to calculate the email engagement metrics.

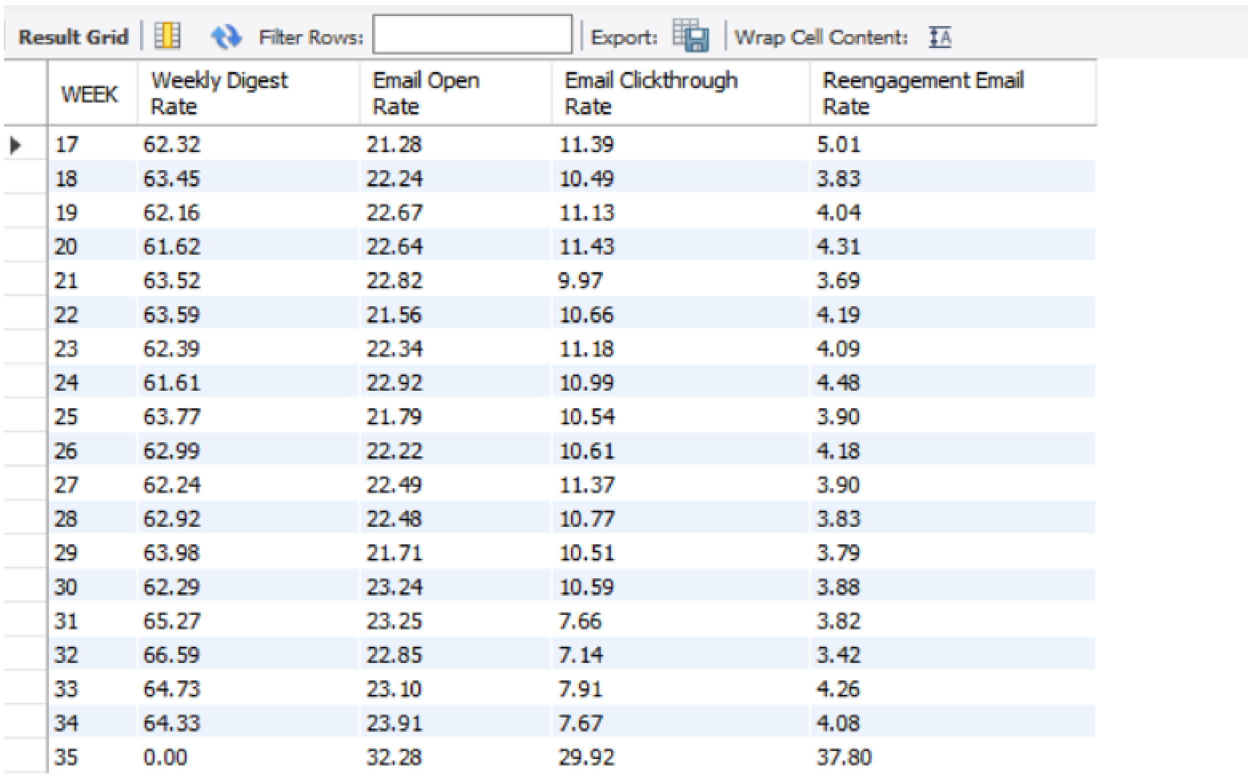


```
SELECT
    WEEK,
    ROUND((weekly_digest / total * 100), 2) AS "Weekly Digest Rate",
    ROUND((email_opens / total * 100), 2) AS "Email Open Rate",
    ROUND((email_clickthrough / total * 100), 2) AS "Email Clickthrough Rate",
    ROUND((reengagement_emails / total * 100), 2) AS "Reengagement Email Rate"
FROM
    (SELECT
        EXTRACT(WEEK FROM occurred_at) AS WEEK,
        COUNT(CASE WHEN action = 'sent_weekly_digest' THEN user_id ELSE NULL END) AS weekly_digest,
        COUNT(CASE WHEN action = 'email_open' THEN user_id ELSE NULL END) AS email_opens,
        COUNT(CASE WHEN action = 'email_clickthrough' THEN user_id ELSE NULL END) AS email_clickthrough,
        COUNT(CASE WHEN action = 'sent_reengagement_email' THEN user_id ELSE NULL END) AS reengagement_emails,
        COUNT(user_id) AS total
    FROM
        email_events
    GROUP BY
        1
    ORDER BY
        1) AS subquery;
```

Main Query:
The main query selects columns for the week number (WEEK) and various email engagement rates. These rates include the percentage of users who engaged in specific email actions (such as opening an email or clicking through a link) relative to the total number of users who received emails. The ROUND function is used to round the calculated rates to two decimal places for better readability.

Subquery:
The subquery is enclosed within parentheses and aliased as subquery. It retrieves data from the email_events table. It calculates the counts of different email actions (sent_weekly_digest, email_open, email_clickthrough, sent_reengagement_email) for each week using COUNT and CASE statements. Additionally, it calculates the total number of email events for each week by counting all occurrences of user_id. The data is grouped by week number using GROUP BY 1, where 1 refers to the first column in the SELECT list (EXTRACT(WEEK FROM occurred_at)).
The results are ordered by the week number using ORDER BY 1.

Joining the Main Query and Subquery:
The main query selects data from the subquery. It calculates the email engagement rates by dividing the counts of specific email actions by the total count of email events for each week. The resulting rates are rounded using the ROUND function.
Finally, the results are returned.



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

WEEK	Weekly Digest Rate	Email Open Rate	Email Clickthrough Rate	Reengagement Email Rate
17	62.32	21.28	11.39	5.01
18	63.45	22.24	10.49	3.83
19	62.16	22.67	11.13	4.04
20	61.62	22.64	11.43	4.31
21	63.52	22.82	9.97	3.69
22	63.59	21.56	10.66	4.19
23	62.39	22.34	11.18	4.09
24	61.61	22.92	10.99	4.48
25	63.77	21.79	10.54	3.90
26	62.99	22.22	10.61	4.18
27	62.24	22.49	11.37	3.90
28	62.92	22.48	10.77	3.83
29	63.98	21.71	10.51	3.79
30	62.29	23.24	10.59	3.88
31	65.27	23.25	7.66	3.82
32	66.59	22.85	7.14	3.42
33	64.73	23.10	7.91	4.26
34	64.33	23.91	7.67	4.08
35	0.00	32.28	29.92	37.80

RESULTS:

This project aimed to analyze user engagement and activity on Instagram using SQL queries. By examining metrics such as likes, comments, and follows, insights were derived to inform decision-making for product enhancements and marketing strategies. Through data-driven analysis, trends in user behavior were uncovered, enabling the identification of key areas for improvement and growth. The utilization of SQL queries allowed for the extraction of meaningful insights from Instagram's vast dataset, aiding in understanding user interactions and preferences. Overall, this project demonstrated the power of SQL in extracting actionable insights from large-scale social media data, contributing to informed decision-making and the continuous improvement of the Instagram platform.

The insights and the key findings I derived in these projects:

- Users exhibit high engagement, with significant likes, comments, and follows.
- Growth trends indicate a steady increase in user activity over time.
- Certain users exhibit suspicious behavior, possibly indicating bot activity.
- Throughput analysis reveals peak activity periods, aiding in resource allocation.
- Language analysis highlights dominant languages among users.
- Retention analysis identifies patterns in user longevity on the platform.
- Email engagement metrics provide insights into user interactions with email services.
- Rolling averages smooth out short-term fluctuations in throughput data.
- Weekly engagement per device showcases platform usage across different devices.
- Duplicate row detection helps maintain data integrity and accuracy.