# SQL Query Analysis Report

## Case Study 1: Job Data Analysis

### 1. Jobs Reviewed Over Time

```sql
-- CASE STUDY 1 ---- TASK A --- JOBS REVIEWED OVER TIME
-- Calculate the number of jobs reviewed per hour for each day in Nov 2020

SELECT ds AS Date,
    COUNT(job_id) AS Joint_Job_Id,
    ROUND((SUM(time_spent) / 3600), 2) AS Total_Time_Sp_Hr,
    ROUND((COUNT(job_id) / (SUM(time_spent) / 3600)), 2) AS Job_Rview_PHr_PDay
FROM job_data
WHERE STR_TO_DATE(ds, '%m/%d/%Y') BETWEEN '2020-11-01' AND '2020-11-30'
GROUP BY ds
ORDER BY STR_TO_DATE(ds, '%m/%d/%Y');
```

**Explanation:**

1. **SELECT Clause:**
   - ds AS Date: Renames the ds column to "Date" for clarity, likely representing the date of job reviews.
   - COUNT(job_id) AS Joint_Job_Id: Counts the number of job_id entries per group, renamed to "Joint_Job_Id".
   - ROUND((SUM(time_spent) / 3600), 2) AS Total_Time_Sp_Hr: Calculates the total time spent (in seconds) divided by 3600 (seconds per hour), rounded to 2 decimal places, renamed to "Total_Time_Sp_Hr".
   - ROUND((COUNT(job_id) / (SUM(time_spent) / 3600)), 2) AS Job_Rview_Phr_PDay: Computes the number of jobs reviewed per hour by dividing the count of jobs by the total time in hours, rounded to 2 decimal places, renamed to "Job_Rview_Phr_PDay**".**
2. **FROM Clause:**
   - job_data: The table containing the data, with columns ds, job_id, and time_spent.
3. **WHERE Clause:**
   - STR_TO_DATE(ds, '%m/%d/%Y') BETWEEN '2020-11-01' AND '2020-11-30': Filters the data to include only records where the ds (date string) falls between November 1, 2020, and November 30, 2020, after converting it from the format %m/%d/%Y (e.g., "11/15/2020") to a date.
4. **GROUP BY Clause:**
   - ds: Groups the results by the ds column (date), allowing aggregation (COUNT and SUM) per day.
5. **ORDER BY Clause:**
   - STR_TO_DATE(ds, '%m/%d/%Y'): Sorts the results by date in ascending order.

## Why We Use This Query:

- The query is designed to analyze productivity or efficiency in reviewing jobs over time. By calculating the number of jobs reviewed per hour for each day in November 2020, it helps identify trends or patterns in workload or performance.
- Converting time spent to hours and computing the review rate provides a standardized metric for comparison across days.
- The filtering to November 2020 and grouping by day align with the case study's goal of tracking jobs reviewed over time within a specific month.

# 2. Weekly and Daily Average Throughput

```sql
#B Weekly Average Throughput (Overall)
SELECT ROUND(COUNT(event) / SUM(time_spent), 2) AS weekly_avg_throughput
FROM job_data;

-- Daily Average Throughput (Per Day)
SELECT ds AS Dates,
    ROUND(COUNT(event) / SUM(time_spent), 2) AS daily_avg_throughput
FROM job_data
GROUP BY ds
ORDER BY STR_TO_DATE(ds, '%m/%d/%Y');
```

**This SQL query consists of two parts: one calculating the weekly average throughput (overall) and the other calculating the daily average throughput (per day). Here's the breakdown:**

## 1. Weekly Average Throughput (Overall)

**Explanation:**

- **SELECT** ROUND(COUNT(event) / SUM(time_spent), 2) AS weekly_avg_throughput:
    - COUNT(event): Counts the total number of events (e.g., job reviews or tasks).
    - SUM(time_spent): Sums the total time spent (likely in seconds) across all events.
    - COUNT(event) / SUM(time_spent): Divides the number of events by the total time to get a throughput rate.
    - ROUND(..., 2): Rounds the result to 2 decimal places**.**
    - Renamed to weekly_avg_throughput for clarity.
- **FROM job_data:** Retrieves data from the job_data table.

## Why We Use This:

- This query provides an overall average throughput rate for all events in the dataset, useful for assessing general productivity or efficiency across a week without breaking it down by day.

## 2. Daily Average Throughput (Per Day)

**Explanation:**

- **SELECT ds AS Dates, ROUND(COUNT(event) / SUM(time_spent), 2) AS daily_avg_throughput:**
    - ds AS Dates: Renames the ds column (likely a date string) to "Dates".
    - COUNT(event): Counts the number of events per group (day).
    - SUM(time_spent): Sums the time spent per group (day).
    - COUNT(event) / SUM(time_spent): Calculates the throughput rate per day.
    - ROUND(..., 2): Rounds the result to 2 decimal places.
    - Renamed to daily_avg_throughput.
- **FROM job_data:** Retrieves data from the job_data table**.**
- **GROUP BY ds**: Groups the data by the ds column to calculate throughput for each day.
- **ORDER BY STR_TO_DATE(ds, '%m/%d/%Y'):** Sorts the results by date in ascending order after converting ds from the format %m/%d/%Y (e.g., "11/15/2020") to a date.

### Why We Use This:

- This query breaks down the throughput rate by day, allowing for a detailed analysis of performance trends or variations over time. It helps identify days with higher or lower efficiency.

  Overall Purpose:
- These queries are used to measure productivity or efficiency (throughput) in terms of events processed per unit of time spent. The weekly average gives a high-level overview, while the daily average provides granular insights, which can be useful for scheduling, resource allocation, or identifying patterns in workload.

## 3. Language Share Percentage

```sql
#C Language Share Percentage
SELECT language,
    ROUND(100 * COUNT(*) / total, 2) AS Percentage,
    jd.total
FROM job_data
CROSS JOIN (
    SELECT COUNT(*) AS total
    FROM job_data
) AS jd
GROUP BY language, jd.total;
```

**This SQL query calculates the percentage share of each language based on the total number of records. Here's the breakdown:**

**Explanation:**

- **SELECT language, ROUND(100 * COUNT(*) / total, 2) AS Percentage:**
    - language: Selects the language column, representing different languages in the data.
    - ROUND(100 * COUNT(*) / total, 2): Calculates the percentage by multiplying the count of records per language by 100 and dividing by the total number of records, then rounding to 2 decimal places.
    - Renamed to Percentage for clarity.
- **FROM job_data:**
    - Retrieves data from the job_data table.
- **CROSS JOIN (SELECT COUNT(*) AS total FROM job_data) AS jd:**

- Creates a cross join with a subquery that counts the total number of records (COUNT(*)) in job_data and aliases it as total in a derived table jd.
- A cross join ensures every row in job_data is paired with the total count.

- **GROUP BY language, jd.total:**
  - Groups the results by language to calculate the count per language, and includes jd.total to ensure the total is available for the percentage calculation.

### Why We Use This Query:

- The query is used to determine the proportion of each language in the dataset, expressed as a percentage of the total records. This is useful for analyzing the distribution of languages (e.g., in job reviews or tasks), identifying dominant languages, or understanding data composition for further analysis or reporting.

## 4. Duplicate Rows Detection

```
#D Duplicate Rows Detection:
SELECT actor_id, COUNT(*) AS Duplicate
FROM job_data
GROUP BY actor_id
HAVING COUNT(*) > 1;
```

**This SQL query detects duplicate rows in the job_data table based on the actor_id column. Here's the breakdown:**

**Explanation:**

- **SELECT actor_id, COUNT(*) AS Duplicate:**
  - actor_id: Selects the actor_id column, which likely represents an identifier for individuals or entities performing jobs.
  - COUNT(*): Counts the number of occurrences of each actor_id.
  - Renamed to Duplicate to indicate the count of duplicates.
- **FROM job_data:**
  - Retrieves data from the job_data table.
- **GROUP BY actor_id:**
  - Groups the data by actor_id to aggregate the count of occurrences for each unique actor_id.
- **HAVING COUNT(*) > 1:**
  - Filters the groups to include only those where the count of actor_id is greater than 1, identifying rows with duplicates.

### Why We Use This Query:

- The query is used to identify and quantify duplicate entries in the dataset where the same actor_id appears more than once. This is helpful for data cleaning, detecting potential errors (e.g., repeated job assignments), or ensuring data integrity by highlighting records that may need review or removal.

# Case Study 2: Investigating Metric Spike

## A. Weekly User Engagement

```sql
#[Case Study 2: Investigating Metric Spike]
#Weekly User Engagement:
SELECT DATE_SUB(occurred_at, INTERVAL (DAYOFWEEK(occurred_at) - 1) DAY) AS week_start,
    COUNT(*) AS event_count
FROM events
GROUP BY DATE_SUB(occurred_at, INTERVAL (DAYOFWEEK(occurred_at) - 1) DAY)
ORDER BY week_start;
```

**This SQL query calculates the weekly user engagement by counting events per week, starting each week on Monday. Here's the breakdown:**

**Explanation:**

- **SELECT DATE_SUB(occurred_at, INTERVAL (DAYOFWEEK(occurred_at) - 1) DAY) AS week_start, COUNT(*) AS event_count:**
    - DATE_SUB(occurred_at, INTERVAL (DAYOFWEEK(occurred_at) - 1) DAY) AS week_start: Adjusts the occurred_at date to the Monday of the week by subtracting the number of days since the previous Monday. DAYOFWEEK(occurred_at) returns 1 (Sunday) to 7 (Saturday), and subtracting 1 aligns the week to start on Monday. Renamed to week_start.
    - COUNT(*): Counts the total number of events for each week.
    - Renamed to event_count**.**
- **FROM events:**
    - Retrieves data from the events table, which likely contains event timestamps in the occurred_at column.
- **GROUP BY DATE_SUB(occurred_at, INTERVAL (DAYOFWEEK(occurred_at) - 1) DAY):**
    - Groups the data by the calculated week_start (Monday of each week) to aggregate event counts per week.
- **ORDER BY week_start:**
    - **S**orts the results by week_start in ascending order (chronologically).

### Why We Use This Query:

- The query is used to investigate weekly user engagement trends, likely as part of a case study to identify metric spikes. By aggregating event counts per week and aligning weeks to start on Monday, it provides a clear view of engagement patterns over time. This can help detect unusual spikes or drops in activity, aiding in further analysis or troubleshooting**.**

## B. User Growth Analysis

```sql
#User Growth Analysis
SELECT DATE(created_at) AS date,
    COUNT(DISTINCT user_id) AS new_users
FROM users
GROUP BY DATE(created_at)
ORDER BY date;
```

**This SQL query performs a user growth analysis by counting new users per day. Here's the breakdown:**

**Explanation:**

- **SELECT DATE(created_at) AS date, COUNT(DISTINCT user_id) AS new_users:**
  - DATE(created_at) AS date: Extracts the date portion from the created_at timestamp and renames it to "date".
  - COUNT(DISTINCT user_id) AS new_users: Counts the number of unique user_id values per date, renamed to "new_users", to represent new users created on that day.
- **FROM users:**
  - Retrieves data from the users table, which likely contains user creation timestamps in the created_at column and user identifiers in the user_id column.
- **GROUP BY DATE(created_at):**
  - Groups the data by the date extracted from created_at to aggregate the count of new users per day.
- **ORDER BY date:**
  - Sorts the results by date in ascending order (chronologically).

## Why We Use This Query:

- The query is used to analyze user growth over time by calculating the number of new, unique users added each day. This helps track the rate of user acquisition, identify growth trends, or detect anomalies in user sign-ups, which is valuable for business or product development insights.

# C. Weekly Retention Analysis

```sql
#Weekly Retention Analysis:
SELECT DATE(u.created_at) AS sign_up_week,
    DATEDIFF(DATE(e.occurred_at), DATE(u.created_at)) / 7 AS retention_week,
    COUNT(DISTINCT e.user_id) AS retained_users
FROM users u
LEFT JOIN events e ON u.user_id = e.user_id
GROUP BY DATE(u.created_at), DATEDIFF(DATE(e.occurred_at), DATE(u.created_at)) / 7
ORDER BY sign_up_week, retention_week;
```

**This SQL query performs a weekly retention analysis by calculating the number of retained users per week after their sign-up week. Here's the breakdown:**

**Explanation:**

- **SELECT DATE(u.created_at) AS sign_up_week, DATEDIFF(DATE(e.occurred_at), DATE(u.created_at)) / 7 AS retention_week, COUNT(DISTINCT e.user_id) AS retained_users:**
  - DATE(u.created_at) AS sign_up_week: Extracts the date from the created_at column in the users table and renames it to "sign_up_week", representing the week a user signed up.
  - DATEDIFF(DATE(e.occurred_at), DATE(u.created_at)) / 7 AS retention_week: Calculates the difference in days between the event date (e.occurred_at) and the sign-up date (u.created_at), then divides by 7 to convert it to weeks, renamed to "retention_week".
  - COUNT(DISTINCT e.user_id) AS retained_users: Counts the number of unique user_id values from the events table, renamed to "retained_users", indicating users who performed an event in a given retention week.
- **FROM users u:**
  - Retrieves data from the users table, aliased as u.
- **LEFT JOIN events e ON u.user_id = e.user_id:**
  - Performs a left join with the events table (aliased as e) on the user_id column, ensuring all users are included even if they have no events.
- **GROUP BY DATE(u.created_at), DATEDIFF(DATE(e.occurred_at), DATE(u.created_at)) / 7:**
  - Groups the data by the sign-up week and retention week to aggregate the count of retained users for each combination.
- **ORDER BY sign_up_week, retention_week:**
  - Sorts the results by sign_up_week and retention_week in ascending order for a chronological view.

## Why We Use This Query:

- The query is used to analyze user retention by tracking how many users remain active (based on events) in subsequent weeks after signing up. This helps identify retention rates over time, assess the effectiveness of engagement strategies, and detect drops or spikes in user activity, which is critical for improving user retention and product success.

## D. Weekly Engagement Per Device

```sql
#Weekly Engagement Per Device
SELECT DATE(occurred_at) AS engagement_week,
    device,
    COUNT(*) AS event_count
FROM events
GROUP BY DATE(occurred_at), device
ORDER BY engagement_week, device;
```

**This SQL query analyzes weekly engagement per device by counting events. Here's the breakdown:**

**Explanation:**

- **SELECT DATE(occurred_at) AS engagement_week, device, COUNT(*) AS event_count:**
  - DATE(occurred_at) AS engagement_week: Extracts the date from the occurred_at timestamp and renames it to "engagement_week", representing the week of the event.
  - device: Selects the device column, indicating the device type used for the event.
  - COUNT(*) AS event_count: Counts the total number of events for each group, renamed to "event_count".
- **FROM events:**
  - Retrieves data from the events table, which likely contains event timestamps (occurred_at) and device information (device).
- **GROUP BY DATE(occurred_at), device:**
  - Groups the data by the date (week) and device to aggregate the event counts for each combination.
- **ORDER BY engagement_week, device:**
  - Sorts the results by engagement_week and device in ascending order for a chronological and organized view.

### Why We Use This Query:

- The query is used to measure weekly engagement levels across different devices, helping to understand user activity patterns. This can reveal which devices are most active, identify trends in device usage, and inform decisions on device-specific features or optimizations.

## E. Email Engagement Analysis

```sql
#Email Engagement Analysis
SELECT DATE(occurred_at) AS engagement_date,
    action,
    COUNT(*) AS total_actions,
    COUNT(DISTINCT user_id) AS unique_users
FROM email_events
GROUP BY DATE(occurred_at), action
ORDER BY engagement_date, action;
```

**This SQL query performs an email engagement analysis by counting total actions and unique users per action and date. Here's the breakdown:**

**Explanation:**

- **SELECT DATE(occurred_at) AS engagement_date, action, COUNT(*) AS total_actions, COUNT(DISTINCT user_id) AS unique_users:**
  - DATE(occurred_at) AS engagement_date: Extracts the date from the occurred_at timestamp and renames it to "engagement_date".
  - action: Selects the action column, representing different types of email interactions (e.g., open, click).
  - **COUNT(*) AS total_actions:** Counts the total number of events (actions) for each group, renamed to "total_actions".
  - **COUNT(DISTINCT user_id) AS unique_users:** Counts the number of unique user_id values for each group, renamed to "unique_users", indicating distinct users performing actions.
- **FROM email_events:**
  - Retrieves data from the email_events table, which likely contains event timestamps (occurred_at), user identifiers (user_id), and action types (action).
- **GROUP BY DATE(occurred_at), action:**
  - Groups the data by the date and action to aggregate the counts for each combination.
- **ORDER BY engagement_date, action:**
  - Sorts the results by engagement_date and action in ascending order for a chronological and organized view.

## Why We Use This Query:

- The query is used to analyze email engagement by tracking the total number of actions and the number of unique users performing those actions per date and action type. This helps evaluate the effectiveness of email campaigns, identify popular actions (e.g., opens vs. clicks), and assess user participation trends over time.