**SQL Query Complete query and Explanation**

**Step 1: Data Exploration**

**Description:** This code gives the information of about the tables present in the database.

**Implemented Query:**

PRAGMA table\_info(users);

PRAGMA table\_info(emotional\_data);

PRAGMA table\_info(loans);

SELECT \* FROM users;

SELECT \* FROM emotional\_data;

SELECT \* FROM loans;

SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM users;

SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM emotional\_data;

SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM loans;

SELECT \* FROM emotional\_data WHERE user\_id = '551' AND intensity = '5.0';

SELECT \* FROM loans WHERE loan\_id = '734';

**Code explanation:**

**PRAGMA table\_info(table\_name);**

* **Function:** Retrieves metadata about the structure of the specified table.
* **Usage:** Used to obtain details about the columns in a table, including their names, data types, constraints (like NOT NULL), default values, and primary key status.
* **Queries:**
  + PRAGMA table\_info(users);
  + PRAGMA table\_info(emotional\_data);
  + PRAGMA table\_info(loans);

**SELECT \* FROM table\_name;**

* **Function:** Selects all columns and rows from the specified table.
* **Usage:** Used to retrieve all data from the given tables for examination.
* **Queries:**
  + SELECT \* FROM users;
  + SELECT \* FROM emotional\_data;
  + SELECT \* FROM loans;

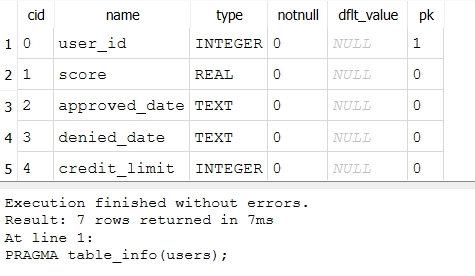
**SELECT COUNT(\*) - COUNT(column\_name) AS missing\_column\_name FROM table\_name;**

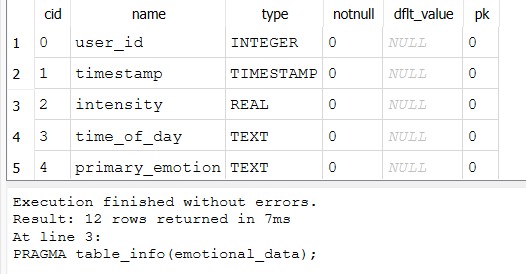
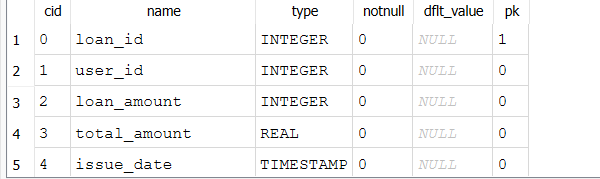
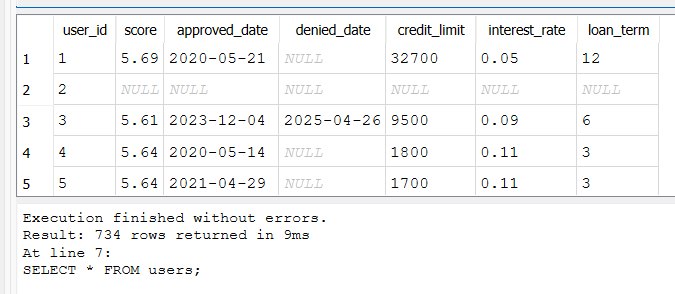
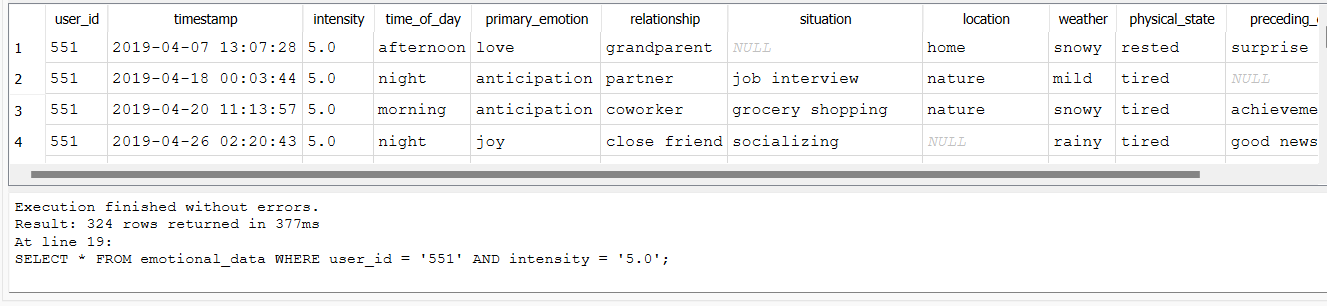
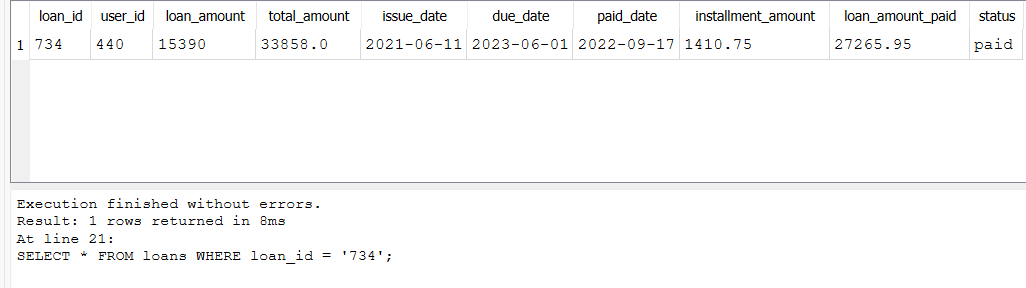
* **Function:** Counts the number of missing values in a specified column of a table by subtracting the count of non-null values from the total count.
* **Usage:** Used to determine how many entries in the specified column are missing (i.e., have null values).
* **Queries:**
  + SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM users;
  + SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM emotional\_data;
  + SELECT COUNT(\*) - COUNT(user\_id) AS missing\_user\_ids FROM loans;

**SELECT \* FROM table\_name WHERE condition;**

* **Function:** Selects all columns from the specified table based on a given condition.
* **Usage:** Used to filter records that meet specific criteria.
* **Queries:**
  + SELECT \* FROM emotional\_data WHERE user\_id = '551' AND intensity = '5.0';
  + SELECT \* FROM loans WHERE loan\_id = '734';

**Output: (Screenshot is attached below)**

****

**** ****   

These queries primarily retrieve information about the tables' structures, examine the data contained within those tables, check for missing values in critical columns, and filter records based on specific conditions. The use of PRAGMA commands enhances the understanding of the database schema, while standard SELECT queries allow for detailed inspection of the data itself.

**Step 2: Finding Null values**

**Description:** The provided SQL queries are designed to assess the completeness of data within three critical tables in the SQLite database: users, loans, and emotional\_data. Each query focuses on identifying missing values across key columns to evaluate the data's quality before any analytical processes or machine learning applications.

**Implemented Query:**

-- For the 'users' table:

SELECT

COUNT(\*) - COUNT(score) AS missing\_score,

COUNT(\*) - COUNT(credit\_limit) AS missing\_credit\_limit,

COUNT(\*) - COUNT(interest\_rate) AS missing\_interest\_rate,

COUNT(\*) - COUNT(loan\_term) AS missing\_loan\_term,

COUNT(\*) - COUNT(approved\_date) AS missing\_approved\_date,

COUNT(\*) - COUNT(denied\_date) AS missing\_denied\_date

FROM users;

-- For the 'loans' table:

SELECT

COUNT(\*) - COUNT(loan\_amount) AS missing\_loan\_amount,

COUNT(\*) - COUNT(total\_amount) AS missing\_total\_amount,

COUNT(\*) - COUNT(installment\_amount) AS missing\_installment\_amount,

COUNT(\*) - COUNT(loan\_amount\_paid) AS missing\_loan\_amount\_paid,

COUNT(\*) - COUNT(issue\_date) AS missing\_issue\_date,

COUNT(\*) - COUNT(due\_date) AS missing\_due\_date,

COUNT(\*) - COUNT(paid\_date) AS missing\_paid\_date,

COUNT(\*) - COUNT(status) AS missing\_status

FROM loans;

-- For the 'emotional\_data' table:

SELECT

COUNT(\*) - COUNT(intensity) AS missing\_intensity,

COUNT(\*) - COUNT(grade) AS missing\_grade,

COUNT(\*) - COUNT(timestamp) AS missing\_timestamp,

COUNT(\*) - COUNT(time\_of\_day) AS missing\_time\_of\_day,

COUNT(\*) - COUNT(primary\_emotion) AS missing\_primary\_emotion,

COUNT(\*) - COUNT(relationship) AS missing\_relationship,

COUNT(\*) - COUNT(situation) AS missing\_situation,

COUNT(\*) - COUNT(location) AS missing\_location,

COUNT(\*) - COUNT(weather) AS missing\_weather,

COUNT(\*) - COUNT(physical\_state) AS missing\_physical\_state,

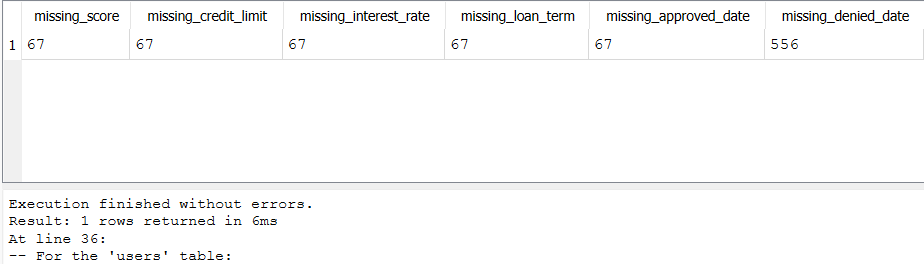
COUNT(\*) - COUNT(preceding\_event) AS missing\_preceding\_event

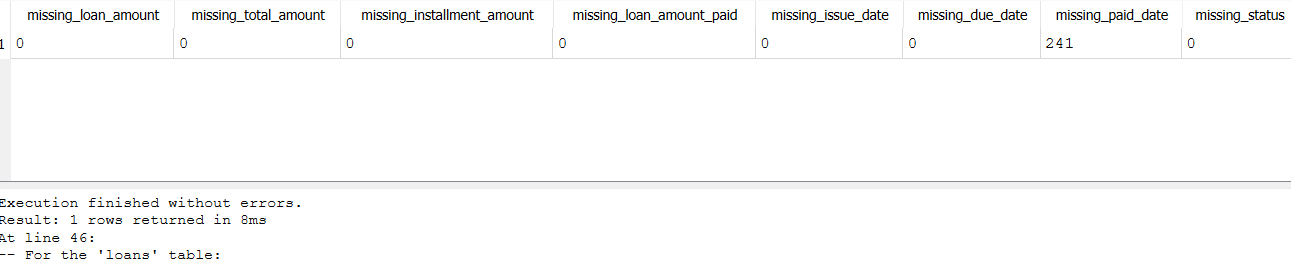
FROM emotional\_data;

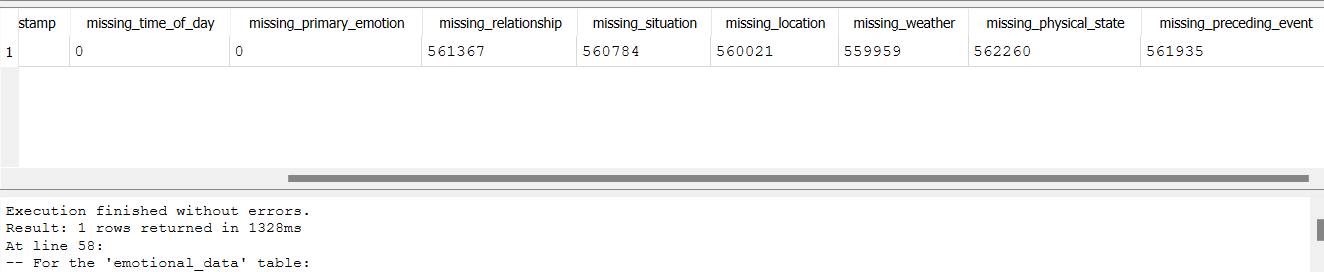
**Explanation of Code:**

* **COUNT(\*):** Counts the total number of rows in the table.
* **COUNT(column\_name):** Counts the number of non-null values in the specified column.
* **Missing Values Calculation:** By subtracting the count of non-null values from the total count, the query determines how many entries are missing for each specified column.

**Output: (Screenshot is attached below)**

****

****

****

**Analysis:**

* In users tables for all the numerical columns we have the 67 null values.
* In loans tables we can see the241 null values in missing\_paid\_date column.
* In emotional\_data column we have ,many null values in the column missing\_relationship, situation, location,. Weather, physical\_state, preceeding\_event.

Hence, we need to remove the null values for better data structure.

**Step 3: Treating Null values**

**Description:** This SQL script is focused on **data preprocessing**, specifically **handling missing values** in both numerical and categorical columns in the users, emotional\_data, and loans tables.

**Implemented Query:**

-- Replace NULL values in numerical columns with their respective mean

UPDATE users

SET score = (SELECT AVG(score) FROM users WHERE score IS NOT NULL)

WHERE score IS NULL;

UPDATE users

SET credit\_limit = (SELECT AVG(credit\_limit) FROM users WHERE credit\_limit IS NOT NULL)

WHERE credit\_limit IS NULL;

UPDATE users

SET interest\_rate = (SELECT AVG(interest\_rate) FROM users WHERE interest\_rate IS NOT NULL)

WHERE interest\_rate IS NULL;

UPDATE users

SET loan\_term = (SELECT AVG(loan\_term) FROM users WHERE loan\_term IS NOT NULL)

WHERE loan\_term IS NULL;

-- Replace NULL values in categorical columns with the most frequent value

UPDATE users

SET approved\_date = (SELECT approved\_date FROM users WHERE approved\_date IS NOT NULL GROUP BY approved\_date ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE approved\_date IS NULL;

UPDATE users

SET denied\_date = (SELECT denied\_date FROM users WHERE denied\_date IS NOT NULL GROUP BY denied\_date ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE denied\_date IS NULL;

-- Replace NULL values in numerical columns with their respective mean

UPDATE emotional\_data

SET intensity = (SELECT AVG(intensity) FROM emotional\_data WHERE intensity IS NOT NULL)

WHERE intensity IS NULL;

UPDATE emotional\_data

SET grade = (SELECT AVG(grade) FROM emotional\_data WHERE grade IS NOT NULL)

WHERE grade IS NULL;

-- Replace NULL values in categorical columns with the most frequent value

UPDATE emotional\_data

SET primary\_emotion = (SELECT primary\_emotion FROM emotional\_data WHERE primary\_emotion IS NOT NULL GROUP BY primary\_emotion ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE primary\_emotion IS NULL;

UPDATE emotional\_data

SET time\_of\_day = (SELECT time\_of\_day FROM emotional\_data WHERE time\_of\_day IS NOT NULL GROUP BY time\_of\_day ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE time\_of\_day IS NULL;

UPDATE emotional\_data

SET relationship = (SELECT relationship FROM emotional\_data WHERE relationship IS NOT NULL GROUP BY relationship ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE relationship IS NULL;

UPDATE emotional\_data

SET situation = (SELECT situation FROM emotional\_data WHERE situation IS NOT NULL GROUP BY situation ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE situation IS NULL;

UPDATE emotional\_data

SET location = (SELECT location FROM emotional\_data WHERE location IS NOT NULL GROUP BY location ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE location IS NULL;

UPDATE emotional\_data

SET weather = (SELECT weather FROM emotional\_data WHERE weather IS NOT NULL GROUP BY weather ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE weather IS NULL;

UPDATE emotional\_data

SET physical\_state = (SELECT physical\_state FROM emotional\_data WHERE physical\_state IS NOT NULL GROUP BY physical\_state ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE physical\_state IS NULL;

UPDATE emotional\_data

SET preceding\_event = (SELECT preceding\_event FROM emotional\_data WHERE preceding\_event IS NOT NULL GROUP BY preceding\_event ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE preceding\_event IS NULL;

-- Replace NULL values in numerical columns with their respective mean

UPDATE loans

SET loan\_amount = (SELECT AVG(loan\_amount) FROM loans WHERE loan\_amount IS NOT NULL)

WHERE loan\_amount IS NULL;

UPDATE loans

SET total\_amount = (SELECT AVG(total\_amount) FROM loans WHERE total\_amount IS NOT NULL)

WHERE total\_amount IS NULL;

UPDATE loans

SET installment\_amount = (SELECT AVG(installment\_amount) FROM loans WHERE installment\_amount IS NOT NULL)

WHERE installment\_amount IS NULL;

UPDATE loans

SET loan\_amount\_paid = (SELECT AVG(loan\_amount\_paid) FROM loans WHERE loan\_amount\_paid IS NOT NULL)

WHERE loan\_amount\_paid IS NULL;

-- Replace NULL values in categorical columns with the most frequent value

UPDATE loans

SET status = (SELECT status FROM loans WHERE status IS NOT NULL GROUP BY status ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE status IS NULL;

UPDATE loans

SET issue\_date = (SELECT issue\_date FROM loans WHERE issue\_date IS NOT NULL GROUP BY issue\_date ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE issue\_date IS NULL;

UPDATE loans

SET due\_date = (SELECT due\_date FROM loans WHERE due\_date IS NOT NULL GROUP BY due\_date ORDER BY COUNT(\*) DESC LIMIT 1)

WHERE due\_date IS NULL;

UPDATE loans

SET paid\_date = (SELECT paid\_date FROM loans WHERE paid\_date IS NOT NULL GROUP BY paid\_date ORDER BY COUNT(\*) DESC LIMIT 1)

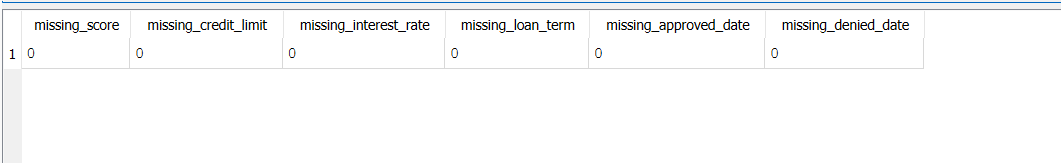
WHERE paid\_date IS NULL;

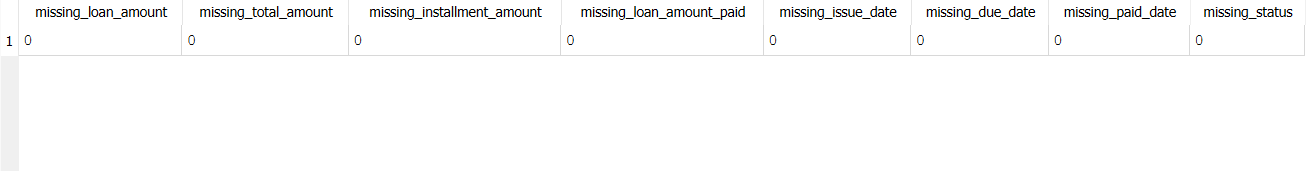
**Explanation of Code:**

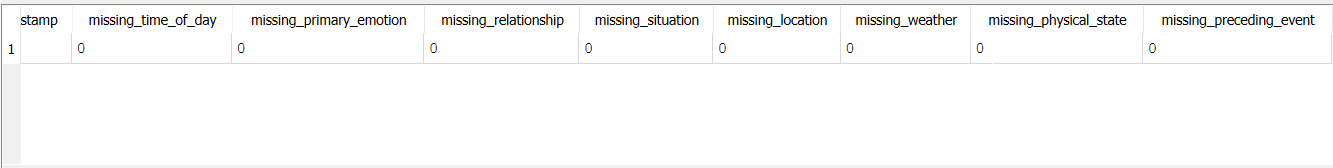
* **Purpose of the Query:** This query is designed to update the status column in the loans table by replacing any NULL values with the most frequently occurring (mode) non-NULL value from the same status column.
* **UPDATE Statement:**
* UPDATE loans: This part of the query specifies that we are updating the loans table.
* SET status = (...): This indicates that we are setting the value of the status column based on the result of the subquery provided within the parentheses.
* **Subquery:**
* The subquery is responsible for selecting the most frequent non-NULL status from the status column.
* SELECT status FROM loans WHERE status IS NOT NULL: This selects all non-NULL values of the status column from the loans table.
* GROUP BY status: This groups the results by each unique value in the status column. This means that for each distinct status (e.g., 'Paid', 'Unpaid', etc.), there will be a group.
* ORDER BY COUNT(\*) DESC: This orders the grouped results by the count of occurrences for each unique status in descending order. The most frequently occurring status will be at the top.
* LIMIT 1: This limits the result of the subquery to only the top row, which corresponds to the most frequent status.
* **WHERE Clause:** WHERE status IS NULL: This condition ensures that only those rows in the loans table with NULL values in the status column are affected by the update. Therefore, the query updates only the rows where status is NULL.

**Step 3(a): Again Checking for null after null value treatment**

**Description:** The query used in finding null values above, same query is used to check if null values are present or not after applying the null value treatment.

**Output:  
  
**

****

**  
  
Analysis:** After the applying the query to remove values. Finally, all the existing null values are removed successfully.

**Step 4: Emotional Pattern Analysis**

**Description:** This query identifies the most common emotions (primary\_emotion) for each month, based on the data from the emotional\_data table. Also correlates emotions with different contexts such as relationship, situation, time\_of\_day, location, and weather to understand how emotions are associated with specific scenarios.

**Implemented query:**

-- Emotional Pattern Analysis:

-- Identify the most common emotions over time

SELECT strftime('%Y-%m', timestamp) AS month, primary\_emotion, COUNT(\*) AS emotion\_count

FROM emotional\_data

GROUP BY month, primary\_emotion

ORDER BY month ASC, emotion\_count DESC;

-- Correlate emotions with different contexts like relationship, situation, time\_of\_day

SELECT

primary\_emotion,

relationship,

situation,

time\_of\_day,

location,

weather,

COUNT(\*) AS emotion\_count

FROM emotional\_data

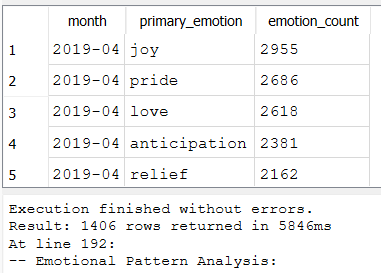
GROUP BY primary\_emotion, relationship, situation, time\_of\_day

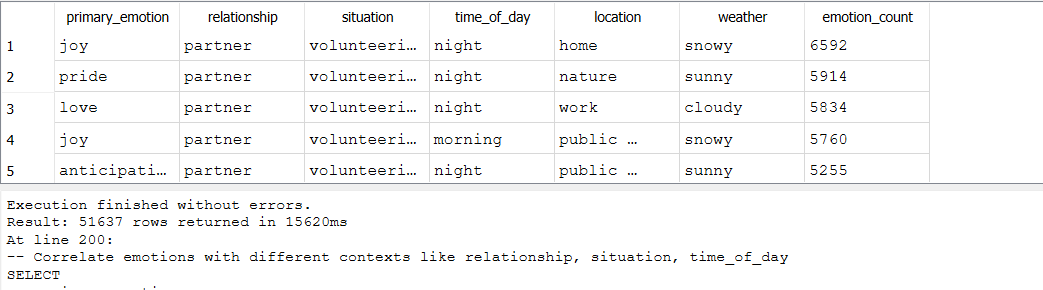
ORDER BY emotion\_count DESC;

**Explanation of Code:**

* strftime('%Y-%m', timestamp) AS month: This function extracts the year and month (e.g., '2024-01') from the timestamp column and renames it as month. The timestamp field is assumed to store the date and time of the emotional data.
* primary\_emotion: This selects the primary emotion recorded during the timestamp.
* COUNT(\*) AS emotion\_count: This counts how many times each emotion occurs within a specific month.
* FROM emotional\_data: The data is fetched from the emotional\_data table.
* GROUP BY month, primary\_emotion: The results are grouped by each combination of month and primary\_emotion, so you can see how many times each emotion appeared in a given month.
* ORDER BY month ASC, emotion\_count DESC: The results are ordered first by the month in ascending order, and then by emotion\_count in descending order. This means that for each month, the emotions will be listed in order of frequency, from most common to least common.
* primary\_emotion: This selects the recorded emotion from the emotional\_data table.
* relationship: This selects the relationship context during which the emotion was recorded (e.g., "friend", "partner").
* situation: This selects the situation or event (e.g., "work", "vacation") where the emotion occurred.
* time\_of\_day: This selects the time of day (e.g., "morning", "afternoon") the emotion was recorded.
* location: This selects the location (e.g., "home", "office") related to the emotion.
* weather: This selects the weather condition (e.g., "sunny", "rainy") at the time the emotion was recorded.
* COUNT(\*) AS emotion\_count: This counts how many times each combination of primary\_emotion, relationship, situation, time\_of\_day, location, and weather occurs.
* FROM emotional\_data: The data is fetched from the emotional\_data table.
* GROUP BY primary\_emotion, relationship, situation, time\_of\_day: This groups the data by emotion and context (relationship, situation, time\_of\_day).
* ORDER BY emotion\_count DESC: The results are ordered by the frequency of occurrences in descending order, so the most common emotion-context combinations are listed first.

**Output: (Screenshot is attached below)**

****

****

**Analysis:** These queries analyze emotional patterns over time and across various contexts such as relationships, situations, and time of day. The first query gives a timeline of dominant emotions, while the second query provides insights into how emotions are tied to specific scenarios. The above code shows the code shows the relationship between primary\_emotion and emotion\_count with respect the timestamp, i.e., month and year as seen in the above output.

**Step 5: Loan – Emotion Correlation**

**Description:** This query examines how emotions are correlated with loan amounts and interest rates by calculating the average loan amount and average interest rate for each emotion. Alsohelps identify which emotional factors (i.e., primary emotions) are associated with different loan results, such as loan statuses (e.g., paid, unpaid). It counts how many loans are associated with each emotion and loan status.

**Implemented Query:**

-- Loan-Emotion Correlation:

-- Analyze correlation between emotions and loan amounts

SELECT

ed.primary\_emotion,

AVG(l.loan\_amount) AS avg\_loan\_amount,

AVG(u.interest\_rate) AS avg\_interest\_rate

FROM emotional\_data ed

JOIN loans l ON ed.user\_id = l.user\_id

JOIN users u ON ed.user\_id = u.user\_id

GROUP BY ed.primary\_emotion

ORDER BY avg\_loan\_amount DESC;

-- Identify which emotional factors influence loan results

SELECT

ed.primary\_emotion,

l.status,

COUNT(\*) AS loan\_count

FROM emotional\_data ed

JOIN loans l ON ed.user\_id = l.user\_id

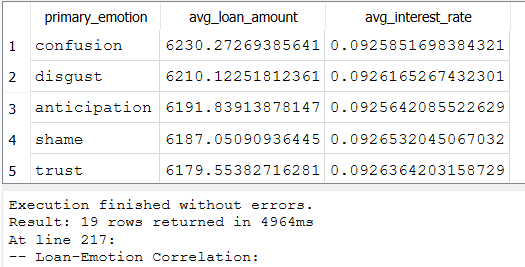
GROUP BY ed.primary\_emotion, l.status

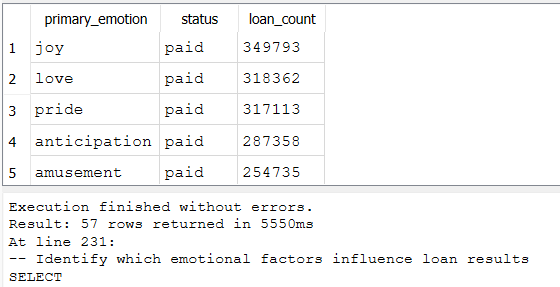
ORDER BY loan\_count DESC;

**Explanation of Code:**

* ed.primary\_emotion: This selects the primary emotion recorded from the emotional\_data table (ed alias).
* AVG(l.loan\_amount) AS avg\_loan\_amount: This calculates the average loan amount from the loans table (l alias) for each emotion.
* AVG(u.interest\_rate) AS avg\_interest\_rate: This calculates the average interest rate from the users table (u alias) for each emotion.
* FROM emotional\_data ed: This specifies the emotional\_data table as the base table.
* JOIN loans l ON ed.user\_id = l.user\_id: This joins the loans table based on the user\_id field, linking the emotions to the loans taken by the same user.
* JOIN users u ON ed.user\_id = u.user\_id: This joins the users table to access user-related fields (like interest rate).
* GROUP BY ed.primary\_emotion: This groups the results by primary\_emotion, so the average loan amount and interest rate are calculated for each emotion.
* ORDER BY avg\_loan\_amount DESC: The results are ordered by the average loan amount in descending order, showing the emotions associated with higher loan amounts at the top.
* ed.primary\_emotion: This selects the primary emotion recorded from the emotional\_data table (ed alias).
* l.status: This selects the loan status (e.g., paid, unpaid) from the loans table (l alias).
* COUNT(\*) AS loan\_count: This counts the number of loans associated with each emotion and status combination.
* FROM emotional\_data ed: This specifies the emotional\_data table as the base table.
* JOIN loans l ON ed.user\_id = l.user\_id: This joins the loans table based on the user\_id, linking emotions with the loan statuses for the same users.
* GROUP BY ed.primary\_emotion, l.status: This groups the results by both primary\_emotion and status, so you can see how emotions correlate with different loan outcomes.
* ORDER BY loan\_count DESC: The results are ordered by the number of loans in descending order, showing which emotion-loan status combinations are most common.

**Output: (Screenshot is attached below)**





**Analysis:**

* **First Query:** This query analyzes the relationship between emotions and loan characteristics, such as average loan amount and average interest rate, for each emotion.
* **Second Query:** This query examines how emotions influence loan outcomes by identifying the emotions most frequently associated with different loan statuses.

**Step 6: Lending Operation Assessment:**

**Description:** This query evaluates loan defaults based on loan amounts and interest rates, providing insight into how different loan characteristics (amount and interest rate) correlate with default risk. Also, analyzes the loan disbursement trends over time by grouping the total loan amounts disbursed per month. It also calculates the revenue trends over time, comparing the actual loan amounts paid versus the total amount expected, and calculating any differences (i.e., revenue gains or losses).

**Implemented Code:**

-- Lending Operation Assessment:

-- Evaluate loan defaults based on interest rates and loan amounts

SELECT

l.loan\_amount,

u.interest\_rate,

COUNT(\*) AS loan\_count,

SUM(CASE WHEN l.status = 'default' THEN 1 ELSE 0 END) AS default\_count

FROM loans l

JOIN users u ON l.user\_id = u.user\_id

GROUP BY l.loan\_amount, u.interest\_rate

ORDER BY default\_count DESC;

-- Analyze disbursement trends over time

SELECT

strftime('%Y-%m', issue\_date) AS month,

SUM(loan\_amount) AS total\_disbursed\_amount

FROM loans

GROUP BY month

ORDER BY month ASC;

-- Calculate revenue trends (loan\_amount\_paid vs total\_amount)

SELECT

strftime('%Y-%m', paid\_date) AS month,

SUM(loan\_amount\_paid) AS total\_revenue,

SUM(total\_amount) AS expected\_total\_amount,

SUM(loan\_amount\_paid - total\_amount) AS revenue\_difference

FROM loans

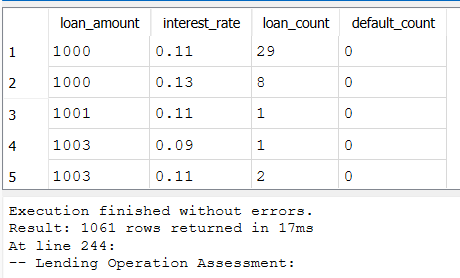
GROUP BY month

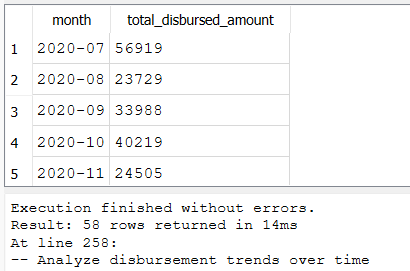
ORDER BY month ASC;

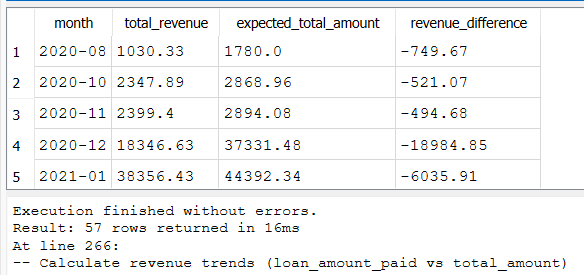
**Explanation of Code:**

* l.loan\_amount: Retrieves the loan amount from the loans table (l alias).
* u.interest\_rate: Retrieves the interest rate from the users table (u alias).
* COUNT(\*) AS loan\_count: Counts the total number of loans for each combination of loan amount and interest rate.
* SUM(CASE WHEN l.status = 'default' THEN 1 ELSE 0 END) AS default\_count: Counts how many of these loans defaulted by using a conditional (CASE) statement that checks if the loan status is 'default'.
* JOIN users u ON l.user\_id = u.user\_id: Joins the users table with the loans table to get user-related information (like interest rate).
* GROUP BY l.loan\_amount, u.interest\_rate: Groups the results by loan amount and interest rate, so the default count is calculated for each combination of these two variables.
* ORDER BY default\_count DESC: Orders the results by the number of defaults in descending order, highlighting which loan amounts and interest rates are associated with the highest default counts.
* strftime('%Y-%m', issue\_date) AS month: Extracts the year and month from the issue\_date of each loan, returning it as a formatted string (e.g., "2024-05").
* SUM(loan\_amount) AS total\_disbursed\_amount: Calculates the total loan amounts disbursed in each month.
* GROUP BY month: Groups the results by month, so the total disbursement is calculated for each month.
* ORDER BY month ASC: Orders the results chronologically (in ascending order by month).
* strftime('%Y-%m', paid\_date) AS month: Extracts the year and month from the paid\_date, returning it as a formatted string for grouping the data by month.
* SUM(loan\_amount\_paid) AS total\_revenue: Calculates the total amount of loans paid in each month.
* SUM(total\_amount) AS expected\_total\_amount: Calculates the total expected amount to be paid for all loans in each month.
* SUM(loan\_amount\_paid - total\_amount) AS revenue\_difference: Calculates the difference between the actual amount paid (loan\_amount\_paid) and the total expected amount (total\_amount), showing any revenue surplus or shortfall.
* GROUP BY month: Groups the results by month.
* ORDER BY month ASC: Orders the results by month in ascending order.

**Output: (Screenshot is attached below)**







**Analysis:**

* **First Query:** Evaluates loan defaults based on the combination of loan amounts and interest rates, helping to identify high-risk lending patterns.
* **Second Query:** Tracks the total amount of loans disbursed each month to observe trends in loan disbursement over time.
* **Third Query:** Compares actual revenue versus expected revenue on a monthly basis, allowing for analysis of any gains or losses in loan payments.