Task 2: Catching fraud

Challenge 1: Explaining SQL

- The WITH clause creates a table called *processed_users*. This table exists only for the duration of the query.
- The contents of the table *processed_users* are defined in the AS ( ) clause that follows:
  - FROM the table *users*
  - SELECT two columns: *phone_country* and *id*
  - LEFT(u.phone_country) extracts 2 characters from the *phone_country* field, starting from the left. For example, "GB|| JE||IM||GG" becomes "GB".
  - The shortened *u.phone_country* column in the *processed_users* table is named AS *short_phone_country*.
- In this query, we only want to SELECT (and show) three columns in the resultant table: t.user_id, t.merchant_country and amount. The third column, aliased AS amount, is calculated with the function Sum(). This function adds together all the transactions made by a single (user, merchant_country) group. The expression t.amount / fx.rate / Power(10, cd.exponent) converts the amount to EUR. More about GROUPs later…
- The FROM clause is used to specify which tables to select data from. In this query, data is taken from a combination of tables. Because there is a common column between the *transactions* table with the other tables, rows from the tables can combined. For example:
  - The table *transaction* has the column *currency*. The fx_rates table has the column *ccy*. Therefore, JOIN the table *fx_rates* to the table *transactions* based ON these matching columns, *fx.ccy = t.currency*. Here, we only want to extract from the table *fx_rates* rows where the base currency is 'EUR'. This is denoted by the AND operator.
  - Similarly, JOIN table *currency_details* to table *transactions*, based ON the matching columns *cd.currency* and *t.currency.*
  - Same goes for the table *pu.*
- The WHERE clause is used to extract records that fulfill a specified condition. Therefore, only rows with
  - Transaction source 'GAIA' and
  - short_phone_country equal to transaction merchant_country

  will be extracted.
- The GROUP BY clause combines transactions that have the same (user, merchant_country) couple into groups.
- Finally, the query ORDERS the resultant table-set by the amount made by each (user, merchant_country) group, from the largest amount to the smallest amount (DESCending order).

Sadly though, the query does not work.

Problem 1:

- Recall that in line 2 SELECT LEFT(u.phone_country, 2), when the temporary table *processed_users* is created, the column *pu.short_phone_country* is extracted from the column *u.phone_country*, in the format of 2 characters. For example, "GB||JE||IM||GG" becomes "GB".
- This becomes a problem later when data is SELECTed by the query . In line 17, a second condition AND *pu.short_phone_country = t. merchant_country* is required for data to be selected.
- However, in the column *t.merchant_country*, data are written in 2 or 3 characters. For example, Great Britain is written in 3 characters "GBR".
- In this case, because "GB" in *pu.short_phone_country* is not equal to "GBR" in *t.merchant_country*, the data row is not selected to be shown by the query.
- Therefore, in the case where the merchant_country is in 3 letters, the query fails to match the field *pu.short_phone_country* to the field *t.merchant_country* even when the countries are equivalent.
- As a result, fewer entries are returned by the query.

Problem 2:

- The query intends to convert the amount of all transactions into Euros. This is shown in the query, command line 9 to 11: JOIN fx_rates fx ON (fx.ccy = t.currency AND fx.base_ccy = 'EUR').
- To convert other currencies into Euros, the formula of conversion should not be t. amount / fx.rate / Power(10, cd.exponent). Rather, it should be t. amount * fx.rate / Power(10, cd.exponent).

Problem 3:

- There is another logical problem related to the command line AND fx.base_ccy = 'EUR'.
- The problem with this clause is that it does not consider cases where the transaction is made in Euros, i.e. *t.currency* = 'EUR'.
- This is because in the table *fx_rates,* there is no information for base_ccy = 'EUR' and ccy = 'EUR'. (Although common sense tells us the rate of converting Euros to Euros is just, 1.)
- Therefore, the query does not work. The resultant table does not include transactions that are made in Euros.

Problem 4:

- Problem 2 is an issue due to how *pu.short_phone_country* is extracted from *u.phone_country.*
- In line 2 SELECT LEFT(u.phone_country, 2), when the temporary table *processed_users* is created, only the first 2 letters in the column *u.phone_country* is extracted to form the column *pu.short_phone_country.*
- However, this is not a very smart way to extract information. Users may have more than one phone country, with separate countries delimited with a "||" symbol. For example: user number 53, id 0aa06081 has phone_country "US||PR||CA||".
- Line 2 only extracts "US" into the column *pu.short_phone_country.*
- Recall that data need to fulfill the second condition *pu.short_phone_country = t. merchant_country* (line 17) in order to be selected.
- A problem arises when the data in *t.merchant_country* is not the first country in the field. For example, one of the transactions made by this user has merchant_country "CA" (*transaction_id = ffed4da9)*
- In this case, because "CA" in *t.merchant_country* is not equal to "US" in *pu.short_phone_country*, the transaction *ffed4da9* is not selected to be shown by the query table-set.
- Therefore, the algorithm used by line 2 in the query is weak. It does not effectively capture all data entries desired, even when *pu.short_phone_country = t. merchant_country.*

Challenge 2: Successful users

The users that fulfill the requirement are as follows:

user_id	first_date	usd_amount
2eb7c137-056b-4a3f-9f98-f2bc4bc2d982	04-08-15 21:36	27.71803
eb3a952f-1949-46e9-bbf2-a7e243207dd7	22-03-16 10:52	48.29018
ef051a6c-c0fc-4b29-aea1-2d5c8eec1ade	04-05-16 11:26	21.22412
e18b2729-3b60-4a93-932d-a66551870ea7	23-05-16 22:10	19.08051
84382c07-626d-4220-bdd7-79e0d88aa850	03-06-16 8:34	26.37172
f63f5c96-2726-4781-8ca4-417c66602e0e	16-06-16 2:14	16.60442
484253ae-3dd7-402e-8565-0b2b612554b3	21-09-16 16:50	20.19943
20a16a2a-ffbf-4e9c-b6fa-9a85c2350647	10-11-16 13:14	13.991

See Appendix A for the steps and SQL queries taken to solve this challenge.

Challenge 3: Fraudsters

The five likely fraudsters are as follows:

	user_id
1	06bb2d68-bf61-4030-8447-9de64d3ce490
2	41a3f59d-8b1d-49a7-aef4-2aa2205d96e9
3	642800ef-f622-4adf-bac4-145e5858afad
4	86459777-d822-49ef-a9e0-95c2c82adeea
5	ec4fd825-7167-450b-aa9f-0cbcf681978b

The fraudsters are identified because their transactions failed more than 5 times in a single minute. See Appendix B for the queries used to solve this challenge.

Appendix A: Steps and query to solving first transaction

- Using SQLite, four tables are created: *transactions, users, fx_rates,* and *currency_details*. The schema for each table is designed as advised by Rita the data analyst.
- Data under the CREATED_DATE column in transactions.xls is edited to conform to the SQL TIMESTAMP format: YYYY-MM-DD HH:MI:SS. The following steps are taken:
    - Right-click on the CREATED_DATE column
    - Under "Format Cells", choose Category Date and then the desired timestamp format.
- Because transactions spreadsheet is given in the .xls format, it is converted to the .csv format before being imported to the SQL terminal.
- The following SQL query is used:

```
INSERT INTO fx_rates VALUES("USD", "USD", 1);

WITH first_transaction
    AS (SELECT user_id,
            MIN(created_date) AS first_date
        FROM transactions t
        GROUP BY t.user_id)
SELECT t.number,
        t.user_id,
        ft.first_date,
        t.amount * fx.rate / 100 AS usd_amount
FROM transactions t
    JOIN first_transaction ft
    ON ( t.created_date = ft.first_date
        AND t.user_id = ft.user_id )
    JOIN fx_rates fx
        ON (fx.ccy = t.currency
            AND fx.base_ccy = 'USD')
        JOIN currency_details cd
        ON cd.currency = t.currency
WHERE t.amount * fx. rate / 100 > 10
        AND t.type = "CARD_PAYMENT"
        AND t.state = "COMPLETED"
ORDER BY first_date;
```

- Because SQLite does not have the Power function, there needs to be a work-around for exchanging currencies.
- It is found that all transactions were carried out in either USD, GBP or EUR. As the exponent for all these currencies is 2, we can safely divide the amount by 100 (as seen in line 9) when calculating the usd-equivalent amount.

Appendix B: Query to find 5 fraudsters

```
SELECT ROW_NUMBER() OVER (ORDER BY (t.user_id)) AS "",
    t.user_id
FROM transactions t
    JOIN (
        SELECT t.user_id, GROUP_CONCAT(t.created_date) AS time_involved
        FROM transactions t
        WHERE t.state != "COMPLETED"
        GROUP BY t.user_id, ROUND(strftime('%s', t.created_date) / 60 / 1 - 0.5, 0)
        HAVING COUNT(*) > 5
    ) frauds
```

```
    ON t.user_id = frauds.user_id
GROUP BY t.user_id;
```