Database Management

Database description:
Initially, we have 7 tables, but 3 of them can be merged into one, as they are 3 tables with the same user columns for Canadian, US, and UK nationalities.

Therefore, we have 4 tables: companies, users, credit_card, and transactions.

This is a relational database structured in a dimensional star model with the 'transactions' table as the fact table and the 'users,' credit_card, 'products,' and 'companies' tables as dimension tables that are related to the fact table 1 to N. There is also a relationship between the users and credit_cards tables 1 to N, which we consider unnecessary and will delete. We will also make the necessary modifications to optimize and organize the database: changing the birth_date and expiring_date types from VARCHAR to DATE.

We created the database that we called sales:
CREATE DATABASE IF NOT EXISTS sales;

We create the tables:
```
CREATE TABLE IF NOT EXISTS credit_cards (
        id VARCHAR(20) PRIMARY KEY,
    user_id INT NOT NULL,
    iban VARCHAR(50) UNIQUE NOT NULL,
    pan VARCHAR(50) UNIQUE NOT NULL,
    pin VARCHAR(50) NOT NULL,
    cvv INT UNIQUE NOT NULL,
    track1 VARCHAR(50),
    track2 VARCHAR(50),
    expiring_date VARCHAR(10) NOT NULL);

CREATE TABLE IF NOT EXISTS companies (
        company_id VARCHAR(20) PRIMARY KEY,
    company_name VARCHAR(50) NOT NULL,
    phone VARCHAR(20) UNIQUE,
    email VARCHAR(50),
    country VARCHAR(20),
    website VARCHAR(100));

CREATE TABLE IF NOT EXISTS users (
        id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    phone VARCHAR(20) UNIQUE,
    email VARCHAR(50) UNIQUE NOT NULL,
    birth_date VARCHAR(20),
    country VARCHAR(20),
    city VARCHAR(20),
    postal_code VARCHAR(20),
    adress VARCHAR(100));
```

```
CREATE TABLE IF NOT EXISTS transactions (
        id VARCHAR(100) PRIMARY KEY,
    card_id VARCHAR(20) NOT NULL,
    business_id VARCHAR(20) NOT NULL,
    timestamp TIMESTAMP,
    amount DECIMAL(10,2) NOT NULL,
    declined boolean,
    product_ids VARCHAR(50) NOT NULL,
    user_id INT NOT NULL,
    lat FLOAT,
    longitude FLOAT);
```

Data Insertion:
To insert a local file to insert data, you must verify that the option is enabled. The initial response is negative, appearing as OFF, so we activate it. We check after activating it, and it appears as ON.

```
SHOW VARIABLES LIKE 'local_infile';
SET GLOBAL local_infile = 1;
```

We also had to go from the Mysql toolbar to ‘Database’ to ‘Manage connections’ and in the ‘Advanced’ tab add the code in ‘Others’: OPT_LOCAL_INFILE=1.

We insert the data:

```
LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/credit_cards.csv'
INTO TABLE credit_cards
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;

SELECT * FROM credit_cards;

LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/companies.csv'
INTO TABLE companies
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;

SELECT * FROM companies;
```

When inserting user data, an error occurred. At first, we thought it was because there were fields containing commas, which was confusing, but upon checking the Excel spreadsheet, we saw that the line break was correct. Therefore, we thought the problem might be the line break, and changing \n to \r\n worked. The credit_cards, companies, and transactions files were created in Unix/Mac OS X format, and the users file was created in Windows format. Below, I'll specify the differences between the different types of line breaks.

\r = CR (Carriage Return) → Used as a newline character on Mac OS before X
\n = LF (Line Feed) → Used as a newline character on Unix/Mac OS X
\r\n = CR + LF → Used as a newline character on Windows

```
LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/users_ca.csv'
INTO TABLE users
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;

LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/users_uk.csv'
INTO TABLE users
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;

LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/users_usa.csv'
INTO TABLE users
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;

SELECT * FROM users;

LOAD DATA LOCAL INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/transactions.csv'
INTO TABLE transactions
FIELDS TERMINATED BY ';'
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;

SELECT * FROM transactions;
```

Modifications in the tables:
CREDIT_CARDS
We remove the credit_card.user_id field:

```
ALTER TABLE credit_card DROP COLUMN user_id;
```

We modify the data type of the expiring_date column:

```
ALTER TABLE credit_cards ADD COLUMN expiring_date_temp DATE;
UPDATE credit_cards SET expiring_date_temp = STR_TO_DATE(expiring_date, '%m/%d/%y');
ALTER TABLE credit_cards DROP expiring_date;
ALTER TABLE credit_cards CHANGE expiring_date_temp expiring_date DATE NOT NULL;
SHOW COLUMNS FROM credit_cards;
```

USERS
In user, we modify the data type of the birth_date column:

```
ALTER TABLE user ADD COLUMN birth_date_temp DATE;
UPDATE user SET birth_date_temp = STR_TO_DATE(birth_date, '%b %d, %Y');
ALTER TABLE user DROP birth_date;
ALTER TABLE user CHANGE birth_date_temp birth_date DATE NOT NULL;
```

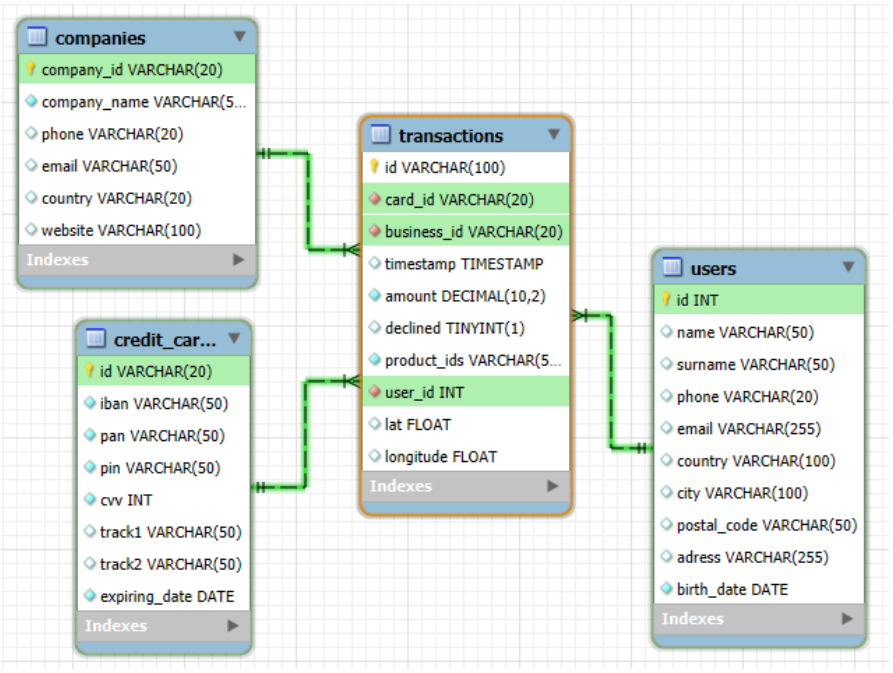
Show that it has been created successfully:
The database:
SHOW TABLES FROM sales;

Tables:
SHOW COLUMNS FROM credit_cards;
SHOW COLUMNS FROM companies;
SHOW COLUMNS FROM users;
SHOW COLUMNS FROM transactions;

We create the Foreign key relationships between the tables
ALTER TABLE transactions
ADD CONSTRAINT fk_transactions_credit_cards FOREIGN KEY (card_id) REFERENCES credit_cards (id),
ADD CONSTRAINT fk_transactions_companies FOREIGN KEY (business_id) REFERENCES companies (company_id),
ADD CONSTRAINT fk_transactions_users FOREIGN KEY (user_id) REFERENCES users (id);

We check that they have been created correctly
SHOW INDEXES FROM transactions;

Initial schema of the 'sales' database:



We create a new table that reflects the status of credit cards based on whether the last three transactions were declined and generate the following query:

We create the new table that is updatable:
We created the new table called credit_status that shows the active or blocked status of the cards based on whether the last 3 transactions have been declined or not.
We use card_id type VARCHAR as Primary Key (the same as the id of the credit_cards table with which we relate it), the VARCHAR type was used as in the credit_cards table and we created a new column called status which could be of Boolean type, but we thought that the ENUM type would be better since it allows certain elements to be displayed, in this case two groups: 'active' for the cards

that have passed the filter such that they have not been declined in the last 3 transactions and 'blocked' for those cards that have passed the filter as they do have their last 3 transactions declined. We relate table 1 to N with the credit_cards table. We check that it has been done correctly.

```
CREATE TABLE credit_status (
   card_id VARCHAR(20) PRIMARY KEY,
   status ENUM('activa', ‘bloqueada’) NOT NULL,
   FOREIGN KEY (card_id) REFERENCES credit_cards(id));
```

We make the query to identify the declined transactions and insert in the table:
We insert into the credit_status table the data in the relevant columns: card_id and status that we took from the following query and using to insert between blocked and active using the CASE method. For the query we use DENSE_RANK() to order the transactions according to the timestamp descending and partitioning by card_id. We call this rank. In this subquery we filter it to select the first 3. When doing the CASE we filter the declines where we take the sum of the declines in the range limited to 3. So those that have less than 3 (last 3 declined transactions) are considered 'active' in status. Those card_ids that have the sum of declinations = 3 will then be marked as 'blocked' in status.

```
INSERT INTO credit_status (card_id, status)
SELECT card_id,
CASE WHEN sum(declined) < 3
        THEN 'activa'
   ELSE 'bloqueada'
END AS status
FROM (SELECT transactions.card_id, transactions.timestamp, declined,
        DENSE_RANK() OVER (PARTITION BY transactions.card_id
                                        ORDER BY transactions.timestamp DESC) AS rango
        FROM transactions) AS consulta_rango
WHERE rango <= 3
GROUP BY card_id
ORDER BY card_id;
```

We add the Foreign Key

```
ALTER TABLE credit_status
ADD CONSTRAINT fk_credit_status_credit_cards FOREIGN KEY (card_id) REFERENCES
credit_cards (id);

SHOW INDEXES FROM credit_status;
```

(Option B:

```
USE transactions;
DROP TABLE IF EXISTS active_credit_cards;
CREATE TABLE active_credit_cards AS
SELECT
   t.credit_card_id AS id,
   CASE
      WHEN COUNT(CASE WHEN t.declined = 1 THEN 1 END) = 3 THEN 'inactive'
      ELSE 'active'
   END AS status
FROM
   (SELECT
      credit_card_id,
      declined,
```

```
    ROW_NUMBER() OVER(PARTITION BY credit_card_id ORDER BY timestamp DESC) AS rn
  FROM transaction) t
WHERE t.rn <= 3
GROUP BY t.credit_card_id;
ALTER TABLE active_credit_cards
ADD CONSTRAINT fk_active_credit_cards_credit_card
FOREIGN KEY (id) REFERENCES credit_card(id);)
```

We create a table with which we can join the data from the new products.csv file with the created database, taking into account that from transaction you have product_ids where there are more than one product

First we will create the products table:

```
CREATE TABLE IF NOT EXISTS products (
  id INT PRIMARY KEY,
  product_name VARCHAR(50) NOT NULL,
  price VARCHAR(10) NOT NULL,
  color VARCHAR(20),
  weight FLOAT,
  warehouse_id VARCHAR(10));
```

We insert the data into the products table:

```
LOAD DATA INFILE 'C:/Users/Nicola Korff/Desktop/SQL/da/sprint_04/products.csv'
INTO TABLE products
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;
```

Check:

```
SHOW TABLES FROM sales;
SHOW COLUMNS FROM products;
```

In the price column we separate the dollar symbol into a new column called currency:

```
ALTER TABLE products ADD COLUMN currency VARCHAR(1);
UPDATE products
SET currency = LEFT(price, 1),
  price = CAST(SUBSTRING(price, 2) AS DECIMAL(10,2));
```

To integrate the data from products.csv into the database and relate it to transactions.product_ids, we need an intermediate table to manage the relationship between transactions and products, since product_ids can contain multiple values per transaction in the same field.

We'll create the orders table from transactions by importing the transaction IDs and products from the products_id column:

```
CREATE TABLE orders SELECT id, products_id FROM transaction;
```

Check:

```
SELECT * FROM orders;
```

We create a temporary table called temp_orders where we can insert the values of the products_id separated by commas:

```
CREATE TABLE temp_orders (
        id VARCHAR(100),
        product_id1 INT,
        product_id2 INT,
        product_id3 INT,
        product_id4 INT );
```

We insert the data into the temps_orders table, extracting and separating the values from the product_ids column of the orders table we just created, which contains the product IDs separated by commas as in the transactions table. We do this using the double SUBSTRING_INDEX function. The last argument of the function, which ranges from 1 to 4, returns the elements of the row. 1 returns the first element; 2 obtains the first two elements, and -1 takes the last of those obtained; the same with 3 and 4. That is, each row of orders will be transformed into a row in temp_orders, where the product_ids values will be divided into up to four columns. If there is no value, it is converted to NULL. With CAST... AS UNSIGNED, we transform the extracted values from the VARCHAR type and convert them into integers (UNSIGNED) so that they are correctly stored in the table.

```
INSERT INTO temp_orders (id, product_id1, product_id2, product_id3, product_id4)
SELECT id,
   CAST(NULLIF(SUBSTRING_INDEX(product_ids, ',', 1), '') AS UNSIGNED) AS product_id1,
   CAST(NULLIF(IF(LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 1,
        SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 2), ',', -1), NULL), '')
        AS UNSIGNED) AS product_id2,
   CAST(NULLIF(IF(LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 2,
        SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 3), ',', -1), NULL), '')
        AS UNSIGNED) AS product_id3,
   CAST(NULLIF(IF(LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 3,
        SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 4), ',', -1), NULL), '')
        AS UNSIGNED) AS product_id4
FROM orders;
```

Once the separation of values is resolved, they must all be transferred to a single field, respecting their IDs. Therefore, we create an auxiliary table called trasp_order to transfer the values from the temp_orders table. We will then delete the orders table so we can rename the trasp_order table to orders. Finally, we will also delete the temporary table.

We create the transfer table:

```
CREATE TABLE trasp_order (
        id VARCHAR(100),
   product_id INT);
```

We insert the values from the temp_orders table:

To insert the data, we normalize the multi-product table into different columns (product_id1, product_id2, product_id3, product_id4) and convert them into individual rows within the transp_order table. We combine the different queries using the UNION ALL method.

```
INSERT INTO trasp_order (id, product_id)
SELECT id, product_id1
FROM temp_orders
WHERE product_id1 IS NOT NULL
UNION ALL
SELECT id, product_id2
```

```
FROM temp_orders
WHERE product_id2 IS NOT NULL
UNION ALL
SELECT id, product_id3
FROM temp_orders
WHERE product_id3 IS NOT NULL
UNION ALL
SELECT id, product_id4
FROM temp_orders
WHERE product_id4 IS NOT NULL;
```

Check:

```
SELECT * FROM trasp_order;
```

We delete the original table "orders" which will be replaced with "trasp_order"

```
DROP table orders;
```

Rename the table "trasp_order" to its final name "orders".

```
ALTER TABLE trasp_order RENAME orders;
```

Finally we delete the temporary table "temp_orders"

```
DROP TABLE temp_orders;
```

We check how the tables turned out:

```
SELECT * FROM orders ORDER BY id;
```

(Option B:

```
INSERT INTO orders2 (transaction_id, product_id)
SELECT transactions.id as transaction_id, products.id AS product_id
FROM transactions
JOIN products ON FIND_IN_SET(products.id, REPLACE (transactions.product_ids, " ", "")) > 0;
```

*SELECT * FROM orders2;*
In each case, the WHERE returns the number of commas.
Here's a step-by-step explanation:
LENGTH(product_ids)
Returns the total length of the string, counting all characters (including commas).
REPLACE(product_ids, ',', '')
Removes all commas from the string.
LENGTH(REPLACE(...))
Returns the length of the string without commas.
Subtract both lengths: LENGTH(original) - LENGTH(without commas)
Gives you the number of commas in product_ids.

```
INSERT INTO orders (order_id, product_id)
SELECT id, CAST(SUBSTRING_INDEX(product_ids, ',', 1) AS UNSIGNED)
FROM orders
WHERE product_ids IS NOT NULL
UNION ALL
SELECT id, CAST(SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 2), ',', -1) AS UNSIGNED)
FROM orders
WHERE LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 1
UNION ALL
SELECT id, CAST(SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 3), ',', -1) AS UNSIGNED)
FROM orders
```

WHERE LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 2
UNION ALL
SELECT id, CAST(SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', 4), ',', -1) AS UNSIGNED)
FROM orders
WHERE LENGTH(product_ids) - LENGTH(REPLACE(product_ids, ',', '')) >= 3;

Option C:
You can also use FIND_IN_SET. This is simpler and more useful code, although it's resource-intensive:
CREATE TABLE IF NOT EXISTS orders2 (
transaction_id VARCHAR(100) NOT NULL,
product_id INT NOT NULL);)

Finally, we add the Foreign Keys:
We create an id index on orders to link to the Foreign Key of the transactions table and the Primary Key of the products table.
ALTER TABLE orders
ADD CONSTRAINT fk_orders_products FOREIGN KEY (product_id) REFERENCES products (id),
ADD CONSTRAINT fk_orders_transactions FOREIGN KEY (id) REFERENCES transactions (id);

Check:
SHOW INDEXES FROM orders;

FINAL OUTLINE: