New generation datamodels and DBMSS Project

January 31, 2025

Author: Luca Maccarini, project edition: 2023 / april 2025

This notebook has been developed in accordance with the Project Guidelines provided by the professor.

1 Transaction Data Simulator Tool

This section focuses on how the various provided scripts were combined to create a single versatile script that, through the use of parameters, is capable of generating CSV files containing all the data that will be inserted into the database. We will not explain the functionality of the Python scripts or the meaning of the data generated by the tool, as these aspects are clearly detailed on the Tool page.

To proceed, the following Python packages and Python sources (from this repository) are required:

```
import os
import sys
import numpy as np
import pandas as pd
import warnings
from IPython.display import SVG, Image, display

sys.path.append(os.path.join(os.getcwd(), '../GenerationScript/Transaction_data_simulator_code'))
from add_frauds import add_frauds
from generate_dataset import generate_dataset

pd.set_option('display.max_rows', 20)
warnings.filterwarnings('ignore')
pd.set_option('display.width', 1000)
```

1.1 Parameters

To manage the parameters for the script in a simple way, I decided to use an array of objects. Each object represents the entire configuration for creating a single database, allowing the script to create multiple databases with different characteristics and data volumes in one run.

Each object in the array contains:

- DB_name: The name of the database.
- n_customers: The number of customers to create.
- n_terminals: The number of terminals to create.
- start_date: The start date for generating transaction data.
- n_days: The number of days after the start_date to use for generating transaction data.

- radius: The action radius for customers. A customer can only perform transactions at a terminal within their radius.

Here is an example:

```
DBs = [
   {
       "DB_name": "DB-410KB",
       "n_customers": 500,
       "n_terminals": 300,
       "n_days": 7,
       "start_date": '2024-12-30',
       "radius": 10
    },
    {
        "DB_name": "DB-14MB",
        "n_customers": 200,
        "n_terminals": 50,
        "n_days": 700,
        "start_date": '2022-01-01',
        "radius": 15
    }
]
```

1.2 Generation Script

Below is the commented code for generating the databases using the parameters defined above.

```
output_dir = ""
# Loop through the databases defined in the configuration file
for db in DBs:
    # Generate database tables using configuration values
    (customer_profiles_table, terminal_profiles_table, transactions_df) = generate_dataset(
        n_customers=db["n_customers"],
        n_terminals=db["n_terminals"],
        nb_days=db["n_days"],
        start_date=db["start_date"],
        r=db["radius"]
    )

    # Add fraud data to the transactions
    transactions_df = add_frauds(customer_profiles_table, terminal_profiles_table, transactions_df)


    # Convert the values of the 'available_terminals' series, as the integers in the list are numpy integers
    customer_profiles_table['available_terminals'] = customer_profiles_table['available_terminals'].apply(
        lambda lst: [int(i) if isinstance(i, np.integer) else i for i in lst] if isinstance(lst, (list, np.array)) else lst
    )
```

```python
    # Prepare for saving the database
    output_dir = os.path.join(os.getcwd(), '..', 'Generated_DBs', db["DB_name"])


    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Saving customers
    customer_profiles_table.to_csv(output_dir + '/customers.csv', sep=';', encoding='utf-8', index=False)

    # Saving terminals
    terminal_profiles_table.to_csv(output_dir + '/terminals.csv', sep=';', encoding='utf-8', index=False)

    # Saving transactions
    transactions_df.to_csv(output_dir + '/transactions.csv', sep=';', encoding='utf-8', index=False)

    print(f"Database data saved in: {os.path.abspath(output_dir)}/\n")


print("DONE! All DBs have been created")
```

```
Time to generate customer profiles table: 0.00s
Time to generate terminal profiles table: 0.00s
Time to associate terminals to customers: 0.05s
Time to generate transactions: 0.41s
Number of frauds from scenario 1: 1
Number of frauds from scenario 2: 127
Number of frauds from scenario 3: 46
Database data saved in: /mnt/1364D0FF74AFABFF/unimi/new generation/progetto/NewGenerationDBMSSProject/Generated_DBs/DB-410KB/

Time to generate customer profiles table: 0.00s
Time to generate terminal profiles table: 0.00s
Time to associate terminals to customers: 0.02s
Time to generate transactions: 4.21s
Number of frauds from scenario 1: 160
Number of frauds from scenario 2: 177216
Number of frauds from scenario 3: 5540
Database data saved in: /mnt/1364D0FF74AFABFF/unimi/new generation/progetto/NewGenerationDBMSSProject/Generated_DBs/DB-14MB/

DONE! All DBs have been created
```

1.3 Generated CSVs

1.3.1 Customers

The following dataFrame shows the generated Customers CSV

```python
pd.read_csv(os.path.join(output_dir, 'customers.csv'), sep=';', encoding='utf-8', index_col=0)
```

```
             x_customer_id  y_customer_id  mean_amount  std_amount  mean_nb_tx_per_day   available_terminals
CUSTOMER_ID
0                54.881350      71.518937    62.262521   31.131260            2.179533        [0, 5, 29, 44]
1                42.365480      64.589411    46.570785   23.285393            3.567092  [0, 4, 5, 8, 11, 46]
2                96.366276      38.344152    80.213879   40.106939            2.115580          [16, 23, 38]
3                56.804456      92.559664    11.748426    5.874213            0.348517              [18, 43]
4                 2.021840      83.261985    78.924891   39.462446            3.480049              [19, 36]
…                      …             …            …          …                  …                   …
195              13.907270      42.690436    85.071214   42.535607            3.272133   [3, 15, 22, 30, 32]
196              10.241376      15.638335    33.898876   16.949438            0.301436            [2, 9, 13]
197              42.466300      10.761771    58.980671   29.490336            0.986228      [24, 27, 37, 47]
198              59.643307      11.752564    97.708967   48.854484            3.730245              [27, 28]
199              39.179694      24.217859    28.787830   14.393915            1.933574              [37, 47]

[200 rows x 6 columns]
```

1.3.2 Terminals

The following dataFrame shows the generated Terminals CSV

```python
pd.read_csv(os.path.join(output_dir, 'terminals.csv'), sep=';', encoding='utf-8', index_col=0)
```

```
             x_terminal_id  y_terminal_id
TERMINAL_ID
0                41.702200      72.032449
1                 0.011437      30.233257
2                14.675589       9.233859
3                18.626021      34.556073
4                39.676747      53.881673
…                      …             …
45               11.474597      94.948926
46               44.991213      57.838961
47               40.813680      23.702698
48               90.337952      57.367949
49                0.287033      61.714491

[50 rows x 2 columns]
```

1.3.3 Transactions

The following dataFrame shows the generated Transactions CSV

```python
pd.read_csv(os.path.join(output_dir, 'transactions.csv'), sep=';', encoding='utf-8', index_col=0)
```

TRANSACTION_ID	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX_TIME_SECONDS	TX_TIME_DAYS	TX_FRAUD	TX_FRAUD_SCENARIO
0	2022-01-01 00:07:56	2	16	146.00	476	0	0	0
1	2022-01-01 00:32:35	183	47	39.30	1955	0	0	0
2	2022-01-01 01:11:00	8	5	2.08	4260	0	0	0
3	2022-01-01 01:56:44	55	18	35.06	7004	0	0	0
4	2022-01-01 01:59:15	159	9	54.22	7155	0	0	0
…	…	…	…	…	…	…	…	…
262558	2023-12-01 22:34:42	57	40	21.72	60474882	699	1	2
262559	2023-12-01 22:45:52	9	33	161.55	60475552	699	1	2
262560	2023-12-01 22:47:16	41	20	9.64	60475636	699	1	2
262561	2023-12-01 22:59:15	1	46	38.33	60476355	699	0	0
262562	2023-12-01 23:07:15	115	26	43.46	60476835	699	1	2

[262563 rows x 8 columns]

1.4 Generated DBs

The project guidelines require three databases to be generated with sizes of 50 MB, 100 MB, and 200 MB. The database generation script does not allow you to directly specify the desired database size. After several tests, I determined the parameters needed to generate the three databases of the desired sizes.

It is important to note that the generated databases simulate scenarios with a high transaction volume and a limited number of customers and terminals. This feature reflects a worst-case scenario for our workload, which should be taken into account when evaluating performance.

Unfortunately, none of the three databases requested by the project can be loaded on a free Neo4j Aura instance due to the excessive number of relationships, which exceeds the 400K limit. So for the demonstration purposes of this notebook, and to ensure that the provided code can run without requiring a Neo4j enterprise instance, I decided to use a 14MB database that we had previously generated with a free Neo4j Aura instance that I had created. Since the free version goes offline after a period of inactivity, you can replace the code I prepared in section 4 by entering the link and credentials of your free neo4j Aura instance.

Despite the performance limitation, in section 6 the queries in this notebook will also be executed on the 50MB, 100MB, and 200MB databases, but on a local enterprise instance that doesn't have any limitations. The parameters used to generate these three databases are as follows:

```
DBs = [
    {
        "DB_name": "50MB",
        "n_customers": 1000,
        "n_terminals": 500,
        "n_days": 500,
        "start_date": '2022-01-01',
        "radius": 5
    },
    {
        "DB_name": "100MB",
        "n_customers": 1200,
        "n_terminals": 600,
        "n_days": 800,
        "start_date": '2022-01-01',
        "radius": 5
```

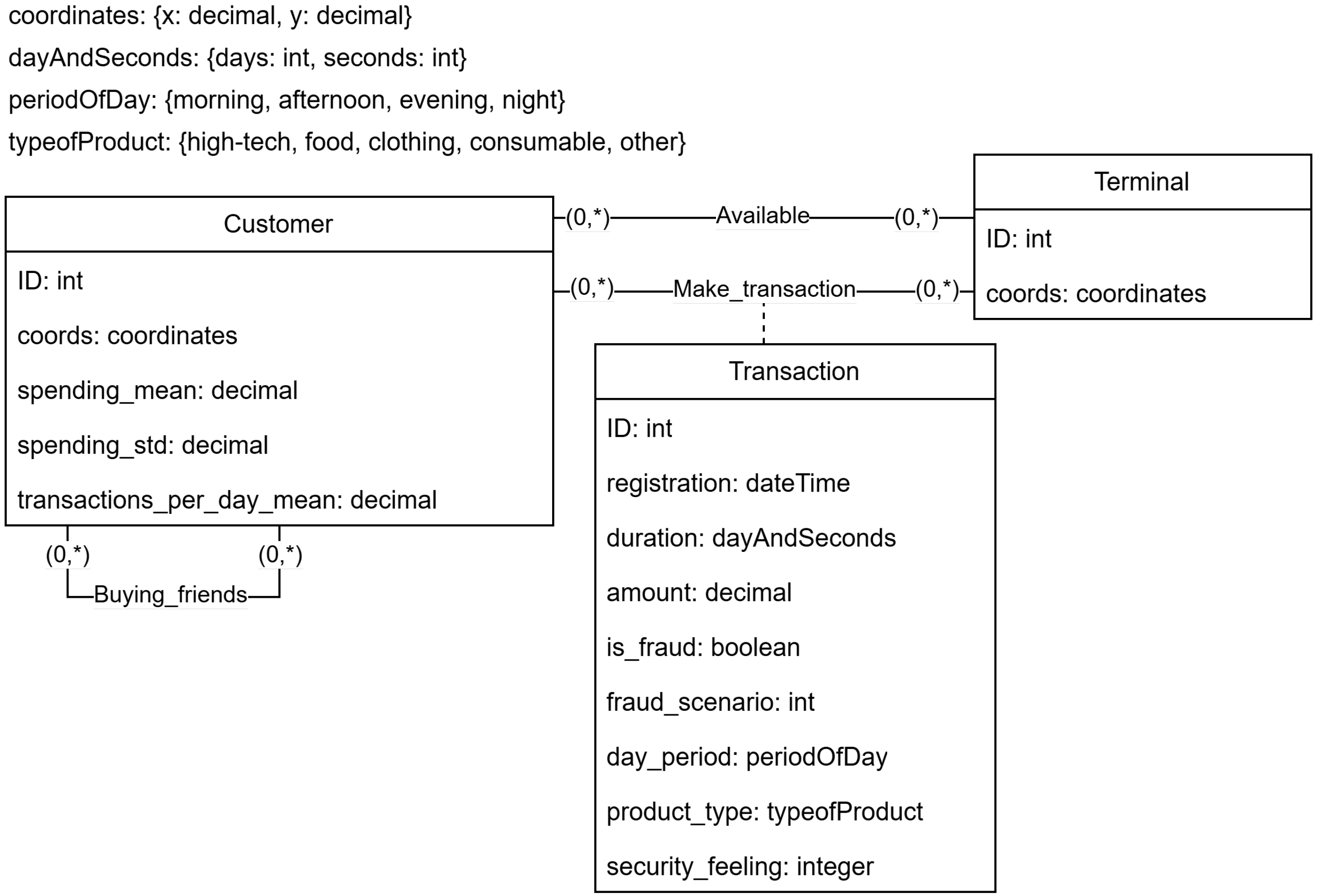
```
    },
    {
        "DB_name": "200MB",
        "n_customers": 2000,
        "n_terminals": 1000,
        "n_days": 900,
        "start_date": '2022-01-01',
        "radius": 5
    }
]
```

2 Conceptual Model

To create the following conceptual model, I analyzed the CSV files generated by the *Transaction Data Simulator* tool and looked at its web page. This analysis helped me understand the semantics of the data and allowed me to design a clear and simple structure that illustrates the relationships between the data.

2.1 UML Class Diagram

```
# To include images in the generated PDF, I used this workaround to embed them correctly.
# If images don't display, re-run the cell displaying the image.
# Ensure IPython.display is imported first (you can find it in the first code cell in side this notebook).
display(SVG(filename="./assets/Conceptual model UML.svg"))
```



2.2 Costraints

2.2.1 Terminal

- $0 <=$ `coords.x` $<= 100$
- $0 <=$ `coords.y` $<= 100$

2.2.2 Customer

- $0 <=$ `coords.x` $<= 100$
- $0 <=$ `coords.y` $<= 100$
- `spending_mean` $>= 0$
- `spending_std` $>= 0$
- `transactions_per_day_mean` $>= 0$

2.2.3 Transactions

- `amount` > 0
- $0 <=$ `fraud_scenario` $<= 3$
- $0 <=$ `security_feeling` $<= 5$
- transactions can be stored only if the customer and the terminal involved are related by an `Available` relationship, therefore the terminal is within the range of the customer

3 Logical Model

Before proceeding with the logical model, it is important to indicate which database I have chosen to manage the data and what decisions I have made for the data representation to meet the workload requirements.

3.1 Database

I chose Neo4j as the database for three main reasons:

- The nature of the data suggests a graph structure;
- All the relationships present are of the N:N type, and such relationships are well handled by graph databases;
- The workload, specifically query 3C, involves continuous traversal of relationships up to a certain `K` value that determines when to stop. Executing this query would be extremely costly if we had to perform a join (or lookup) for each relationship traversed.

In addition, as we will see later, Cypher, Neo4j's query language, provides a library called APOC that allows us to execute query 3C with impressive performance.

3.2 Data representation (workload friendly)

Since Neo4j does not allow the definition of custom types or the insertion of objects within node properties, I decided to eliminate all custom types and implement them using primitive types. For the custom types representing objects, I created a property for each attribute with its corresponding primitive type. For enums, I used simple strings.

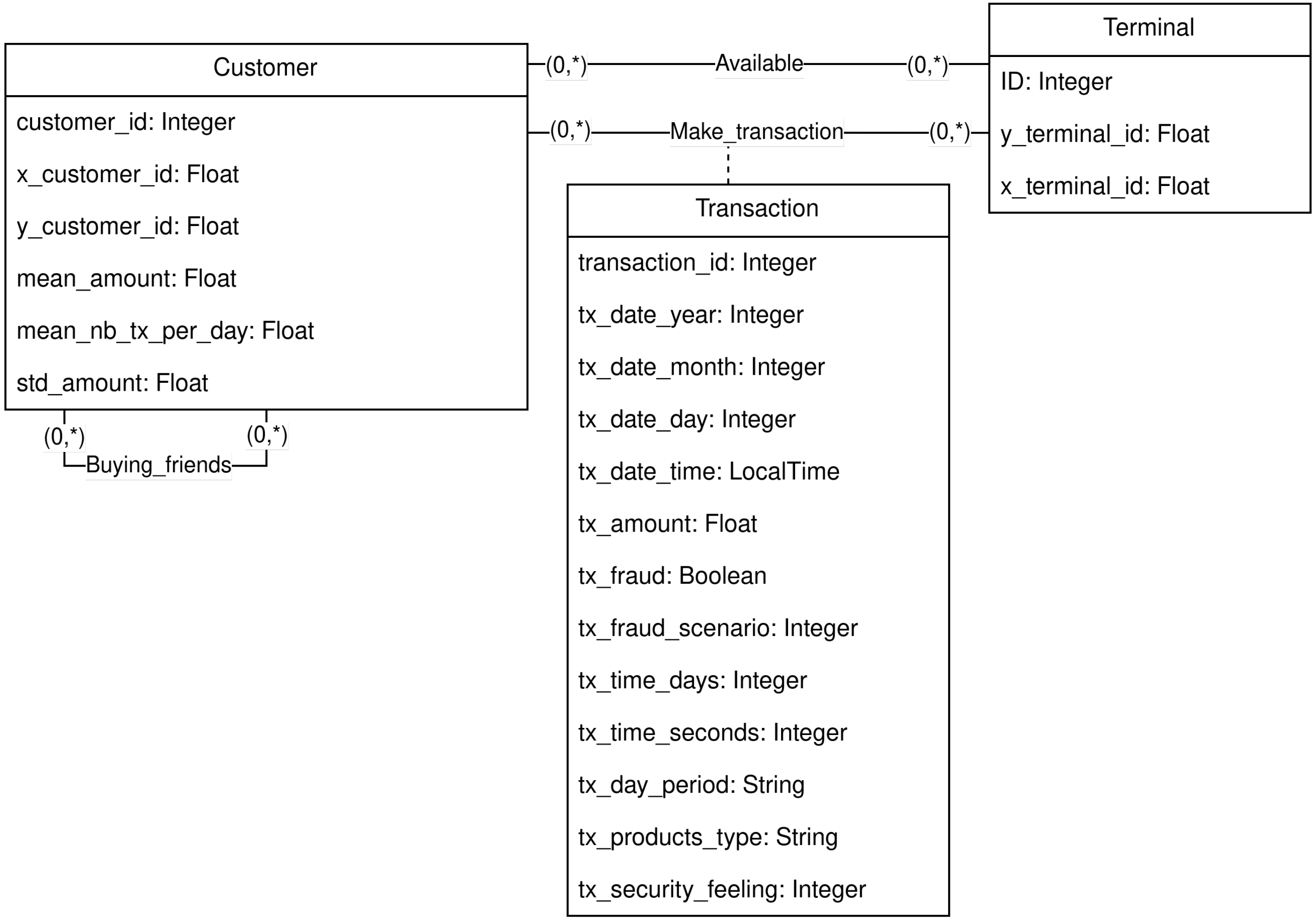
The attribute names in the logical model differ from those in the conceptual model because they are based on those used by the *Transaction Data Simulator* tool. The meaning of any ambiguous or newly introduced fields can be determined by:

- Referring to the *Transaction Data Simulator* tool web page. for fields generated by the tool.
- Reading the following section, which explains the new fields I have added.
- Consulting the project guidelines, which detail and justify the fields explicitly required in the extended database.

As we will see later, in order to improve the efficiency of the workload, I decided to split the `transactions.registration` field into its components: day, month, year, and time. These components are now represented as `tx_date_day`, `tx_date_month`, `tx_date_year` and `tx_date_time` respectively. This division was made because many queries in the workload filter data using only the month and year of the `transactions.registration` field. If I had created an index on the entire field, it would not have been used because the filters in the queries would only use a subset of the entire field. Therefore, the division was made to create a composite index on the year and month fields.

The data types specified in the following UML class diagram are those that exist in Neo4j.

```
display(SVG(filename="./assets/Logical model UML.svg"))
```



3.3 Costraints

3.3.1 Terminal

- $0 <= \texttt{x\_terminal\_id} <= 100$
- $0 <= \texttt{y\_terminal\_id} <= 100$

3.3.2 Customer

- $0 <= \texttt{x\_customer\_id} <= 100$
- $0 <= \texttt{y\_customer\_id} <= 100$
- $\texttt{mean\_amount} >= 0$
- $\texttt{std\_amount} >= 0$
- $\texttt{mean\_nb\_tx\_per\_day} >= 0$

3.3.3 Transactions

- `tx_amount` > 0
- $0 <=$ `tx_fraud_scenario` $<= 3$
- $0 <=$ `tx_security_feeling` $<= 5$
- `tx_date_day`, `tx_date_month`, `tx_date_year` form a correct date type object
- `tx_date_time` forms a correct localTime object
- `tx_day_period` is in [“morning”, “afternoon”, “evening”, “night”]
- `tx_products_type` is in [“high-tech”, “food”, “clothing”, “consumable”, “other”]
- transactions can be stored only if the customer and the terminal involved are linked by an `Available` relationship

3.3.4 Assumptions

Since the constraints implementable in Neo4j focus only on data structure and type. I am not able to define constraints on the actual values or the direction of the relationships, so I assume that whatever software is providing the data to be inserted into the database has correctly implemented all the constraints listed above (except for the constraints on the `tx_date_...` properties, since these can be validated at the database level). In our case, we assume that the values generated by the *Transaction Data Simulator* tool are correct and satisfy the constraints.

Since Neo4j constraints do not allow us to define the direction of relationships, it is our responsibility to ensure that the queries used to create relationships are correctly formulated. We must be careful to avoid creating relationships in the wrong direction.

For more detailed information, I refer you to the Neo4j documentation.

4 Neo4j Data Loading

To proceed the following Python packages are required:

```
import time
import neo4j
import logging
logging.getLogger("neo4j").setLevel(logging.ERROR)
```

To facilitate interactions with Neo4j, we will define some *kernel* functions that will be used to interface with the database. These functions will simplify data management with Neo4j and provide reusable methods for the rest of the project.

To keep the code simple and easy to understand, the *kernel* functions will receive queries with parameters directly embedded through string concatenation. While this approach simplifies the code, it introduces potential security risks, such as code injection, due to the direct concatenation of parameters into the queries. However, since the goal of this project is to demonstrate how the database is managed to optimize workload, and not to focus on addressing security concerns, I have chosen to prioritize simplicity over security in this case.

Before defining the *kernel* functions, we set some configuration parameters that will be useful not only for the *kernel* functions themselves, but also for the various queries that will be executed by the *kernel* functions later in the project. Among the configuration parameters we have:

- `customers_csv_link`, `terminals_csv_link`, `transactions_csv_link`: These parameters refer to the CSV files containing the data to be imported into the database. These three parameters can either contain local file paths or network links. We will see later why network links are preferred in this specific case. The provided network links reference the previously generated 14MB database CSV files hosted on Dropbox. However, in the performance analysis section, we will also use local links for the 50MB, 100MB, and 200MB databases to show the comparison.

- `lines_per_commit_call` and `lines_per_commit_apoc`: These parameters define the number of operations included in each batch, with changes to the database being committed after every batch. I have defined two separate parameters because the optimal batch size depends on the specific job. Jobs that use Cypher’s `CALL {} IN TRANSACTIONS OF ... ROWS` generally support larger batch sizes compared to those using the `APOC.periodic.iterate( ... )` function from the `APOC` library. In this notebook, the value is set to 1000 for both parameters, as it works well for the given context. However, on my local instance, I have used values of 105 and 104, respectively, to further optimize performance.

- `parallel_loading`: useful for the `APOC.periodic.iterate( ... )` batch operations mentioned in the previous point. This parameter indicates whether the database should perform the batch operations in parallel or sequentially.

```
#config parameters
config = {
    "customers_csv_link":  "https://www.dropbox.com/scl/fi/ofi4fd99aydhnp30i2spy/customers.csv?rlkey=iqfr9uaty48gc4toxlssqcvf1&st=h3vqznsz&dl=1",
    "terminals_csv_link":  "https://www.dropbox.com/scl/fi/4tt3cyhnpj4q3y49xksrp/terminals.csv?rlkey=1881everw81e38nc0xa2n32ct&st=8eurat39&dl=1",
    "transactions_csv_link":  "https://www.dropbox.com/scl/fi/we51epibb3p98syq67kcq/transactions.csv?rlkey=4bm84xkt9b7rub9rs0u7cough&st=j1xhtfsa&dl=1",
    "lines_per_commit_call": 1000,
    "lines_per_commit_apoc": 1000,
    "parallel_loading": "true"
}

def get_neo4j_connection():
    try:
        #Using environment variables (recommended): This method securely stores credentials outside the code by using environment variables.
        #uri = os.getenv('NEO4J_URI')
        #user = os.getenv('NEO4J_USERNAME')
        #password = os.getenv('NEO4J_PASSWORD')

        #Using plain strings (not recommended): This method directly includes credentials in the code, which exposes them to potential security risks.
        #In this case, to keep things as simple as possible, I will use plain text credentials since they are for a free version of Neo4j.
        #You can create it by following this link: https://neo4j.com/product/auradb
        uri = "neo4j+s://45d4bc57.databases.neo4j.io"
        user = "neo4j"
        password = "o8mbh0hFGILahScLJw2yTYWIwQ6z7lPhQT6m-U2W1c8"

        #local db
        #uri = "bolt://localhost:7687"
        #user = "neo4j"
        #password = "abcdefgh"

        return neo4j.GraphDatabase.driver(uri, auth=(user, password))

    except Exception as e:
        print(f"ERROR: An unexpected error occurred while connecting to Neo4j: {e}")
        return None

def close_neo4j_connection(driver):
    if driver is not None:
        driver.close()

def clear_database():
    driver = get_neo4j_connection()
    delete_nodes_query = """
        MATCH (n)
        CALL apoc.nodes.delete(n, $lines_per_commit_apoc) YIELD value
```

```
        RETURN value
    """

    try:
        start_time = time.time()
        with driver.session() as session:
            session.run(delete_nodes_query, {"lines_per_commit_apoc": config["lines_per_commit_apoc"]})

            constraints_result = session.run("SHOW CONSTRAINTS")
            for record in constraints_result:
                drop_constraint_query = "DROP CONSTRAINT $name"
                session.run(drop_constraint_query, {"name": record["name"]})

            indexes_result = session.run("SHOW INDEXES")
            for record in indexes_result:
                drop_index_query = "DROP INDEX $name"
                session.run(drop_index_query, {"name": record["name"]})

            print("clear_database execution time: {:.2f}s".format(time.time() - start_time))
            return True
    except Exception as e:
        print(f"ERROR clear_database: {e}")
        return False

    finally:
        close_neo4j_connection(driver)

def execute_query_commands(name, queries):
    driver = get_neo4j_connection()
    try:
        with driver.session() as session:
            start_time = time.time()
            for query in queries:
                try:
                    session.run(query)
                except Exception as e:
                    return False

        print(f"{name} execution time: {{:.2f}}s".format(time.time() - start_time))
        return True

    except Exception as e:
        print(f"ERROR {name}: {e}")
        return False

    finally:
```

```python
        close_neo4j_connection(driver)

def execute_query_df(name, query):
    driver = get_neo4j_connection()
    if driver is None:
        return False

    try:
        start_time=time.time()
        result = driver.execute_query(query, result_transformer_= neo4j.Result.to_df)
        print(f"{name} execution time: {{:.2f}}s".format(time.time() - start_time))

        return result
    except Exception as e:
        print(f"ERROR {name}: {e}")
        return None
    finally:
        close_neo4j_connection(driver)
```

4.1 Database Cleanup

This step is unnecessary if you have just created a new database instance, but **if you are reusing an instance on which you have already performed some operations**, such as running this notebook, **it is necessary to restore it to its original state** by clearing everything. This is where the `clear_database()` function comes in handy.

```python
clear_database()
```

```
clear_database execution time: 18.20s
```

```
True
```

4.2 Schema

Neo4j's constraints focus solely on data structure, as they are used to define a schema for the data. The schemaless nature of Neo4j, or the schemaless nature of NoSQL databases in general, allows data to be inserted with maximum flexibility without the need to define a formal schema in advance. This flexibility allows for handling heterogeneous data and adapting to changes over time, making it ideal for scenarios where the data structure may evolve.

Despite this flexibility, defining a schema is still considered good practice. It provides several benefits, particularly in terms of performance when running queries that filter data or when calculations need to be performed on the data. By enforcing data types and data existence through the schema, the database can optimize certain operations, especially those that involve processing existing values. On the other hand, a disadvantage of using a schema is that it requires additional processing during insertions and modifications, as the database must validate that each new piece of data conforms to the defined constraints.

The database schema we are about to define simply imlements the previous showed logical model in the neo4j DB by defining the following constraints:

- attributes type: each attribute will be associated with its corresponding data type;
- primary key: for each entity the attributes that form the primary key will be explicitly defined;
- mandatory attributes: All attributes not included in the primary key will be marked as mandatory to ensure data integrity. (Primary key attributes are already mandatory due to their primary key constraint).

```python
def create_terminals_schema():
    queries = [
        "CREATE CONSTRAINT terminal_id_is_integer FOR (t:Terminal) REQUIRE t.terminal_id IS :: INTEGER;",
        "CREATE CONSTRAINT terminal_id_key FOR (t:Terminal) REQUIRE t.terminal_id IS NODE KEY;",
        "CREATE CONSTRAINT terminal_x_is_float FOR (t:Terminal) REQUIRE t.x_terminal_id IS :: FLOAT;",
        "CREATE CONSTRAINT terminal_x_required FOR (t:Terminal) REQUIRE t.x_terminal_id IS NOT NULL;",
        "CREATE CONSTRAINT terminal_y_is_float FOR (t:Terminal) REQUIRE t.y_terminal_id IS :: FLOAT;",
        "CREATE CONSTRAINT terminal_y_required FOR (t:Terminal) REQUIRE t.y_terminal_id IS NOT NULL;"
    ]

    return execute_query_commands("create_terminals_schema", queries)

def create_customers_schema():
    queries = [
        "CREATE CONSTRAINT customer_id_is_integer FOR (c:Customer) REQUIRE c.customer_id IS :: INTEGER;",
        "CREATE CONSTRAINT customer_id_key FOR (c:Customer) REQUIRE c.customer_id IS NODE KEY;",
        "CREATE CONSTRAINT customer_x_is_float FOR (c:Customer) REQUIRE c.x_customer_id IS :: FLOAT;",
        "CREATE CONSTRAINT customer_x_required FOR (c:Customer) REQUIRE c.x_customer_id IS NOT NULL;",
        "CREATE CONSTRAINT customer_y_is_float FOR (c:Customer) REQUIRE c.y_customer_id IS :: FLOAT;",
        "CREATE CONSTRAINT customer_y_required FOR (c:Customer) REQUIRE c.y_customer_id IS NOT NULL;",
        "CREATE CONSTRAINT customer_mean_amount_is_float FOR (c:Customer) REQUIRE c.mean_amount IS :: FLOAT;",
        "CREATE CONSTRAINT customer_mean_amount_required FOR (c:Customer) REQUIRE c.mean_amount IS NOT NULL;",
        "CREATE CONSTRAINT customer_std_amount_is_float FOR (c:Customer) REQUIRE c.std_amount IS :: FLOAT;",
        "CREATE CONSTRAINT customer_std_amount_required FOR (c:Customer) REQUIRE c.std_amount IS NOT NULL;",
        "CREATE CONSTRAINT customer_mean_nb_tx_per_day_is_float FOR (c:Customer) REQUIRE c.mean_nb_tx_per_day IS :: FLOAT;",
        "CREATE CONSTRAINT customer_mean_nb_tx_per_day_required FOR (c:Customer) REQUIRE c.mean_nb_tx_per_day IS NOT NULL;"
    ]
    return execute_query_commands("create_customers_schema", queries)

def create_transaction_schema():
    queries = [
        "CREATE CONSTRAINT transaction_id_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.transaction_id IS :: INTEGER;",
        "CREATE CONSTRAINT transaction_id_key FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.transaction_id IS RELATIONSHIP KEY;",
        "CREATE CONSTRAINT tx_time_seconds_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_time_seconds IS :: INTEGER;",
        "CREATE CONSTRAINT tx_time_seconds_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_time_seconds IS NOT NULL;",
        "CREATE CONSTRAINT tx_time_days_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_time_days IS :: INTEGER;",
        "CREATE CONSTRAINT tx_time_days_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_time_days IS NOT NULL;",
        "CREATE CONSTRAINT tx_amount_is_float FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_amount IS :: FLOAT;",
        "CREATE CONSTRAINT tx_amount_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_amount IS NOT NULL;",
        "CREATE CONSTRAINT tx_date_day_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_day IS NOT NULL;",
        "CREATE CONSTRAINT tx_date_day_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_day IS :: INTEGER;",
        "CREATE CONSTRAINT tx_date_month_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_month IS :: INTEGER;",
        "CREATE CONSTRAINT tx_date_month_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_month IS NOT NULL;",
        "CREATE CONSTRAINT tx_date_year_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_year IS :: INTEGER;",
        "CREATE CONSTRAINT tx_date_year_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_year IS NOT NULL;",
        "CREATE CONSTRAINT tx_date_time_is_localtime FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_time IS :: LOCAL TIME;",
```

```
        "CREATE CONSTRAINT tx_date_time_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_date_time IS NOT NULL;",
        "CREATE CONSTRAINT tx_fraud_is_boolean FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_fraud IS :: BOOLEAN;",
        "CREATE CONSTRAINT tx_fraud_is_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_fraud IS NOT NULL;",
        "CREATE CONSTRAINT tx_fraud_scenario_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_fraud_scenario IS :: INTEGER;
↪",
        "CREATE CONSTRAINT tx_fraud_scenario_is_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_fraud_scenario IS NOT NULL;"
    ]
    return execute_query_commands("create_transaction_schema", queries)

create_terminals_schema()
create_customers_schema()
create_transaction_schema()
```

```
create_terminals_schema execution time: 0.66s
create_customers_schema execution time: 0.99s
create_transaction_schema execution time: 1.33s
```

```
True
```

4.3 Data loading

To load data into Neo4j using CSV files, it's important to consider the location of the Neo4j instance, as the CSV files must be accessible from the machine running Neo4j. There are two possible scenarios:

- the CSV files reside on the machine running the Neo4j instance;
- the CSV files are network resources that can be downloaded directly from a link.

Since we are using a Neo4j instance managed by an external company, Aura, that do not give us access to their servers, we must choose the second option.

This will have an impact on the performance of the data load, because the time indicated by the load procedure will include not only the time it takes to load the data from the file into the database, but also the time it takes the Neo4j instance to download the file. The download time is not negligible because, as we know, the network is much slower than a completely local approach. You can check this yourself by pasting the URL of the transaction CSV file into your browser and see how long it takes your machine to download the file.

It's crucial to use a direct download link for the CSV files to ensure everything works. To share the files quickly and easily, I chose Dropbox, as it provides a file sharing option with links that include a query parameter. This parameter, `&dl=1`, ensures the link is a direct download, which is essential for the Neo4j instance to correctly fetch the file. I considered other cloud storage services, but obtaining a direct download link was more complicated.

Now let's look at the queries used to load the data into the database. Initially, I considered loading the data using the same example that the professor provided during the lessons: `USING PERIODIC COMMIT 1000 LOAD CSV FROM ...`, which is used to load data from a CSV file in batches of 1000 rows per commit. However, since this directive is deprecated, I decided to use `LOAD CSV WITH HEADERS FROM ... CALL {...} IN TRANSACTIONS OF 1000 ROWS`, which gave me the same behavior.

All three data loading functions work similarly: each function downloads the CSV file specified by the link, then starts the batch job inside the `CALL{}` statement where the query creates the data instances in the database. At the end of the query in the `IN TRANSACTIONS OF 1000 ROWS` statement, we specify how many rows from the CSV to process before committing the changes in the database.

In all 3 queries, the instances are created with a `MERGE` statement that sets the properties of the instance using the `ON CREATE SET` clause.

- The `load_customers_with_available_terminals_from_csv()` function not only creates the customer, but also opens the list of terminals where the customer can operate on, matches them, and creates an `available` relationship between the customer and all matched terminals.
- The `load_transactions_from_csv()` function before creating the transaction must match the customer and terminal to create the relationship between them.

```python
def load_terminals_from_csv():
    query = f"""
        LOAD CSV WITH HEADERS FROM "{config["terminals_csv_link"]}" AS row FIELDTERMINATOR ';'
        CALL {{
            WITH row
            CREATE (:Terminal {{terminal_id: toInteger(row.TERMINAL_ID),
                                x_terminal_id: toFloat(row.x_terminal_id),
                                y_terminal_id: toFloat(row.y_terminal_id)}})
        }} IN TRANSACTIONS OF {config["lines_per_commit_call"]} ROWS
    """
    return execute_query_commands("load_terminals_from_csv", [query])

def load_customers_with_available_terminals_from_csv():
    query = f"""
        LOAD CSV WITH HEADERS FROM "{config["customers_csv_link"]}" AS row FIELDTERMINATOR ";"
        CALL {{
            WITH row
            MERGE (c:Customer {{customer_id: toInteger(row.CUSTOMER_ID)}})
            ON CREATE SET
                c.x_customer_id = toFloat(row.x_customer_id),
                c.y_customer_id = toFloat(row.y_customer_id),
                c.mean_amount = toFloat(row.mean_amount),
                c.std_amount = toFloat(row.std_amount),
                c.mean_nb_tx_per_day = toFloat(row.mean_nb_tx_per_day)
            WITH c, row
            WITH c, apoc.convert.fromJsonList(row.available_terminals) AS available_terminal_ids
            UNWIND available_terminal_ids AS available_terminal_id
            MATCH (t:Terminal {{terminal_id: available_terminal_id}})
            MERGE (c)-[:Available]->(t)
        }} IN TRANSACTIONS OF {config["lines_per_commit_call"]} ROWS
    """

    return execute_query_commands("load_customers_with_available_terminals_from_csv", [query])

def load_transactions_from_csv():
    query = f"""
        LOAD CSV WITH HEADERS FROM "{config["transactions_csv_link"]}" AS row FIELDTERMINATOR ";"
        CALL{{
            WITH row

            WITH row,
                 split(row.TX_DATETIME, " ") AS splitted_date_time

            WITH row,
                 date(splitted_date_time[0]) AS parsed_date,
                 localtime(splitted_date_time[1]) AS parsed_local_time
```

```
            MATCH (c:Customer {{customer_id: toInteger(row.CUSTOMER_ID)}}),
                (t:Terminal {{terminal_id: toInteger(row.TERMINAL_ID)}})
            MERGE (c)-[transaction:Make_transaction {{transaction_id: toInteger(row.TRANSACTION_ID)}}]->(t)
            ON CREATE SET
                transaction.tx_time_seconds = toInteger(row.TX_TIME_SECONDS),
                transaction.tx_time_days = toInteger(row.TX_TIME_DAYS),
                transaction.tx_amount = toFloat(row.TX_AMOUNT),
                transaction.tx_fraud = toBoolean(toInteger(row.TX_FRAUD)),
                transaction.tx_fraud_scenario = toInteger(row.TX_FRAUD_SCENARIO),

                transaction.tx_date_day = parsed_date.day,
                transaction.tx_date_month = parsed_date.month,
                transaction.tx_date_year = parsed_date.year,
                transaction.tx_date_time = parsed_local_time
        }} IN TRANSACTIONS OF {config["lines_per_commit_call"]} ROWS
    """
    return execute_query_commands("load_transactions_from_csv", [query])


load_terminals_from_csv()
load_customers_with_available_terminals_from_csv()
load_transactions_from_csv()
```

```
load_terminals_from_csv execution time: 1.47s
load_customers_with_available_terminals_from_csv execution time: 2.03s
load_transactions_from_csv execution time: 28.61s
```

```
True
```

5 Workload

In this section, I will explain how I implemented the queries to efficiently respond to the various requirements outlined in the project specifications. Since the requested queries were not always precise in every detail, the analysis of each query will follow these key points:

- Present the query as expressed in the project specifications;
- Explain my interpretation of the requested query;
- Explain how I built the query, providing the query code;
- Look at the results;
- Evaluate the performance of the query. Where necessary, to demonstrate the optimizations I have added, the execution plan will also be provided.

Other query performance details are included in section 6, where the execution times of different queries are compared across databases of different sizes.

Important: Since I could not find a way to clear the caches in the free Neo4j instance (and I don't believe it is possible), when comparing the execution times of different versions of the same query, or the same query on different databases, it is crucial to ensure the accuracy of the timings by running them multiple times. Of course queries that change the state of the database, such as those that create schema, insert data, or modify existing data, should be run at most once per clean database instance. To run them again, it's necessary to restart the instance using the `clear_database()` function. This is because the schema-building functions are designed to fail if a schema rule already exists, ensuring that you are not using an unclean instance. The only exception to the rule for queries that change the state of the database and can be run as many times as needed is `create_transaction_date_index()`. This query creates an index to optimize queries. If an index with the same name already exists, the function does nothing and does not create a new one. If the existing index does not match the one defined by the function, it is not critical for the database, but queries may not be optimized.

5.1 Query A

5.1.1 Query Request

"For each customer checks that the spending frequency and the spending amounts of the last month is under the usual spending frequency and the spending amounts for the same period".

- "For each customer": indicates that the query results must include all customers, even those for which it is not possible to calculate the requested data.
- "last month": refers to the month before the one specified as a parameter in the query. To call the Python function that executes this query, you must specify a partial date in "yyyy-MM" format as a parameter. This date is then used to calculate the `first_of_previous_month` variable within the query. This variable represents the first day of the month immediately preceding the given partial date.
- "Usual spending frequency and spending amounts for the same period": I interpreted this to mean that the spending frequency and amount must be calculated as the average of all spending frequencies and amounts recorded in the database that match the same month but correspond to a year earlier than the `first_of_previous_month` variable.

5.1.2 A1 query code

Let's provide a first version of the A query.

The query starts by calculating the date corresponding to the first day of the previous month relative to the partial date provided to the Python function. This date is stored in the `first_of_previous_month` variable.

Next, all customers are matched to ensure that none are excluded from the final result of the query. This is done because the following `WHERE` clauses do not filter out customers, and all subsequent matches are `OPTIONAL MATCH`.

The first `OPTIONAL MATCH` is used to retrieve the transaction history for the same period, these transactions are stored in the variable `tx_prev_month_all_prev_year`.

The following `WITH` clause is special because, instead of directly calculating the spending frequency and total amount for each year, it returns `NULL` for both values if no `tx_prev_month_all_prev_year` records are found for the relative year. This approach helps distinguish, in the final result, customers with no significant transaction history (and thus no calculations can be performed) from those with a transaction history, for whom calculations can be made as required by the query.

The next `WITH` clause calculates the averages over the years of the results just calculated, `tx_prev_month_prev_year_total_amount` and `tx_prev_month_prev_year_montly_freq`, yielding `tx_prev_month_all_prev_year_total_amount_avg` and `tx_prev_month_all_prev_year_montly_freq_avg`. The `AVG` operator preserves the `NULL` value when calculating based on `NULL`, so if there is no transactions history, `AVG(NULL)` will return `NULL`.

The last `OPTIONAL MATCH` performs the same calculations as the previous one, but now on transactions `tx` that have the same month and year as `first_of_previous_month`. Unlike before, there is no need to distinguish between customers with and without transactions at this stage, as this distinction is made in the `RETURN` clause by referencing the historical data.

The last `WITH` calculates `total_amount_prev_month` and `monthly_freq_prev_month` which represent the total transaction amount and transaction frequency of all `tx`. These two values are then used in the `RETURN` stage to determine if they are below the usual average transaction amount and frequency.

In the `RETURN` statement, if the customer has historical data for the same period (indicated by `tx_prev_month_all_prev_year_monthly_freq_avg IS NOT NULL`), then we check whether `total_amount_prev_month < tx_prev_month_all_prev_year_total_amount_avg` and `monthly_freq_prev_month < tx_prev_month_all_prev_year_monthly_freq_avg`. It is important to note that in this scenario the customer may not have any `tx`. However, since historical data is available, the absence of `tx` does not indicate missing data in the database. Instead, it means that the customer has not made any transactions in the same month and year as `first_of_previous_month`.

If a customer doesn't have the same period of historical data, we can't give a meaningful answer, so we respond with a `NULL` value in both the `is_under_total_amount_avg_of_same_period` and `is_under_monthly_freq_avg_of_same_period` columns.

```python
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_a1(year_and_month_under_analesis):
    query = f"""
            WITH date.truncate('month', date("{year_and_month_under_analesis}" + "-01") ) - duration({{months: 1}}) AS first_of_previous_month

            MATCH (c:Customer)

            OPTIONAL MATCH (c)-[tx_prev_month_all_prev_year:Make_transaction]->(:Terminal)
            WHERE
                tx_prev_month_all_prev_year.tx_date_month = first_of_previous_month.month
                AND tx_prev_month_all_prev_year.tx_date_year < first_of_previous_month.year
            WITH
                first_of_previous_month,
                c,
                tx_prev_month_all_prev_year.tx_date_year as year,
                CASE
                    WHEN COUNT(tx_prev_month_all_prev_year)>0 THEN SUM(tx_prev_month_all_prev_year.tx_amount)
                    ELSE NULL
                END AS tx_prev_month_prev_year_total_amount,

                CASE
                    WHEN  COUNT(tx_prev_month_all_prev_year)>0 THEN COUNT(tx_prev_month_all_prev_year)
                    ELSE NULL
                END AS tx_prev_month_prev_year_montly_freq
            WITH
            first_of_previous_month,
            c,
            AVG(tx_prev_month_prev_year_total_amount) AS tx_prev_month_all_prev_year_total_amount_avg,
            AVG(tx_prev_month_prev_year_montly_freq) AS tx_prev_month_all_prev_year_montly_freq_avg

            OPTIONAL MATCH (c)-[tx:Make_transaction]->(:Terminal)
            WHERE
                tx.tx_date_month = first_of_previous_month.month AND
                tx.tx_date_year = first_of_previous_month.year
            WITH
                c,
                SUM(tx.tx_amount) AS total_amount_prev_month,
                COUNT(tx) AS monthly_freq_prev_month,
                tx_prev_month_all_prev_year_total_amount_avg,
                tx_prev_month_all_prev_year_montly_freq_avg

            RETURN
                c,

                CASE
                    WHEN tx_prev_month_all_prev_year_total_amount_avg IS NULL THEN NULL
```

```
                ELSE total_amount_prev_month < tx_prev_month_all_prev_year_total_amount_avg
            END AS is_under_total_amount_avg_of_same_period,

            CASE
                WHEN tx_prev_month_all_prev_year_montly_freq_avg IS NULL THEN NULL
                ELSE monthly_freq_prev_month < tx_prev_month_all_prev_year_montly_freq_avg
            END AS is_under_monthly_freq_avg_of_same_period
    """

    return execute_query_df("query_a1",query)

month_and_year_under_analesis = "2023-05"
query_a1(month_and_year_under_analesis)
```

```
query_a1 execution time: 3.12s
```

```
                                                   c is_under_total_amount_avg_of_same_period is_under_monthly_freq_avg_of_same_period
0    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                    False
1    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                     True
2    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                     True
3    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                    False
4    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                    False
..                                               …                                       …                                        …
195  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                     True
196  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    False
197  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                     True
198  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                     True
199  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                     True

[200 rows x 3 columns]
```

5.1.3 A1 Performances

In order to improve the performance of the query, since it matches the data on `make_transaction.tx_date_month` and `make_transaction.tx_date_year`, we can create a compound index on these two fields. After that, we can call the query again, passing the same argument, and look at the execution time.

```
def create_transaction_date_index():
    query = "CREATE INDEX composite_index_on_tx_date_year_and_month IF NOT EXISTS FOR ()-[tx:Make_transaction]-() ON (tx.tx_date_month, tx.
 ↪tx_date_year)"
    return execute_query_commands("create_transaction_date_index", [query])

create_transaction_date_index()
```

```
create_transaction_date_index execution time: 0.40s
```

```
True
```

```
query_a1(month_and_year_under_analesis)
```

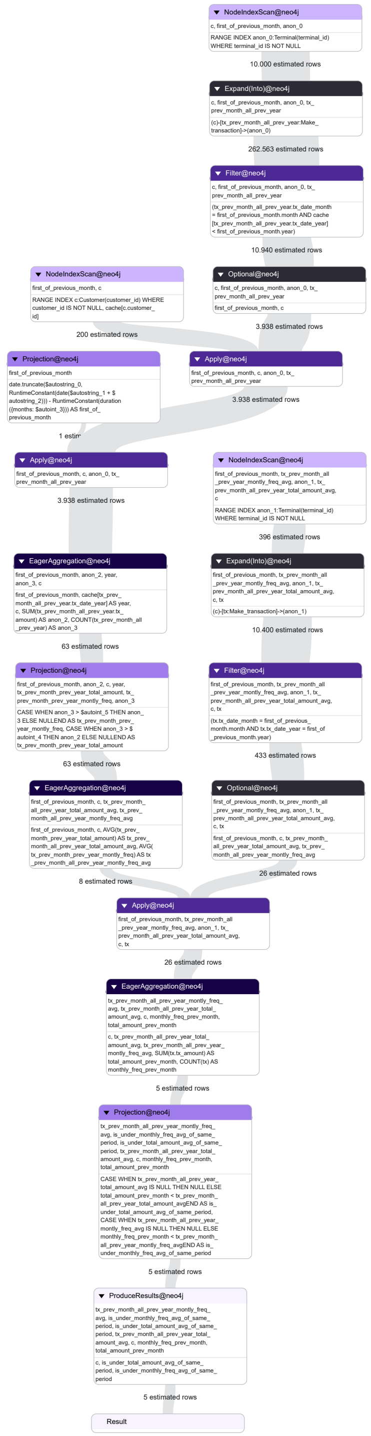
```
query_a1 execution time: 3.20s

                                                     c is_under_total_amount_avg_of_same_period is_under_monthly_freq_avg_of_same_period
0    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                   False
1    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    True
2    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    True
3    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                   False
4    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                   False
..                                                 …                                        …                                       …
195  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    False                                    True
196  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                   False
197  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    True
198  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    True
199  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     True                                    True

[200 rows x 3 columns]
```

The execution time remains nearly the same because the query doesn't utilize the index. As shown in the execution plan below, this is due to the initial `MATCH` clause, where customers are matched first without directly filtering the transactions, not using the index.

In fact, the only index used is on the customers, and it is only used to retrieve all the customer nodes without doing any filtering. As for the transactions, no index is used either in the initial filtering or in the subsequent `OPTIONAL MATCH`, which further contributes to the inefficiency of the query.

To generate the execution plan shown in the image, you simply need to prefix the query with the word `EXPLAIN` in the Neo4j console.

```
display(SVG(filename="./assets/Execution plan query A1.svg"))
```



NodeIndexScan@neo4j
c, first_of_previous_month, anon_0
RANGE INDEX anon_0:Terminal(terminal_id) WHERE terminal_id IS NOT NULL
10.000 estimated rows
Expand(Into)@neo4j
c, first_of_previous_month, anon_0, tx_prev_month_all_prev_year
(c)-[tx_prev_month_all_prev_year:Make_transaction]->(anon_0)
262.563 estimated rows
Filter@neo4j
c, first_of_previous_month, anon_0, tx_prev_month_all_prev_year
(tx_prev_month_all_prev_year.tx_date_month = first_of_previous_month.month AND cache[tx_prev_month_all_prev_year.tx_date_year] < first_of_previous_month.year)
10.940 estimated rows
NodeIndexScan@neo4j
first_of_previous_month, c
RANGE INDEX c:Customer(customer_id) WHERE customer_id IS NOT NULL, cache[c.customer_id]
200 estimated rows
Optional@neo4j
c, first_of_previous_month, anon_0, tx_prev_month_all_prev_year
first_of_previous_month, c
3.938 estimated rows
Projection@neo4j
first_of_previous_month
date.truncate($autostring_0, RuntimeConstant(date($autostring_1 + $autostring_2))) - RuntimeConstant(duration({months: $autoint_3})) AS first_of_previous_month
1 estim
Apply@neo4j
first_of_previous_month, c, anon_0, tx_prev_month_all_prev_year
3.938 estimated rows
Apply@neo4j
first_of_previous_month, c, anon_0, tx_prev_month_all_prev_year
3.938 estimated rows
NodeIndexScan@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c
RANGE INDEX anon_1:Terminal(terminal_id) WHERE terminal_id IS NOT NULL
396 estimated rows
EagerAggregation@neo4j
first_of_previous_month, anon_2, year, anon_3, c
first_of_previous_month, cache[tx_prev_month_all_prev_year.tx_date_year] AS year, c, SUM(tx_prev_month_all_prev_year.tx_amount) AS anon_2, COUNT(tx_prev_month_all_prev_year) AS anon_3
63 estimated rows
Expand(Into)@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
(c)-[tx:Make_transaction]->(anon_1)
10.400 estimated rows
Projection@neo4j
first_of_previous_month, anon_2, c, year, tx_prev_month_prev_year_total_amount, tx_prev_month_prev_year_montly_freq, anon_3
CASE WHEN anon_3 > $autoint_5 THEN anon_3 ELSE NULLEND AS tx_prev_month_prev_year_montly_freq, CASE WHEN anon_3 > $autoint_4 THEN anon_2 ELSE NULLEND AS tx_prev_month_prev_year_total_amount
63 estimated rows
Filter@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
(tx.tx_date_month = first_of_previous_month.month AND tx.tx_date_year = first_of_previous_month.year)
433 estimated rows
EagerAggregation@neo4j
first_of_previous_month, c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg
first_of_previous_month, c, AVG(tx_prev_month_prev_year_total_amount) AS tx_prev_month_all_prev_year_total_amount_avg, AVG(tx_prev_month_prev_year_montly_freq) AS tx_prev_month_all_prev_year_montly_freq_avg
8 estimated rows
Optional@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
first_of_previous_month, c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg
26 estimated rows
Apply@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
26 estimated rows
EagerAggregation@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg, SUM(tx.tx_amount) AS total_amount_prev_month, COUNT(tx) AS monthly_freq_prev_month
5 estimated rows
Projection@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, is_under_monthly_freq_avg_of_same_period, is_under_total_amount_avg_of_same_period, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
CASE WHEN tx_prev_month_all_prev_year_total_amount_avg IS NULL THEN NULL ELSE total_amount_prev_month < tx_prev_month_all_prev_year_total_amount_avgEND AS is_under_total_amount_avg_of_same_period, CASE WHEN tx_prev_month_all_prev_year_montly_freq_avg IS NULL THEN NULL ELSE monthly_freq_prev_month < tx_prev_month_all_prev_year_montly_freq_avgEND AS is_under_monthly_freq_avg_of_same_period
5 estimated rows
ProduceResults@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, is_under_monthly_freq_avg_of_same_period, is_under_total_amount_avg_of_same_period, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
c, is_under_total_amount_avg_of_same_period, is_under_monthly_freq_avg_of_same_period
5 estimated rows
Result

5.1.4 A2 Query Code

By slightly modifying the query to omit the "for all customers" clause displaying only customers with historical data, we can significantly improve performance by leveraging the index. This tweak involves removing the first `MATCH` clause and changing the second `OPTIONAL MATCH` to a regular `MATCH`.

This change means that the results will no longer include customers with `NULL` values in the columns `tx_prev_month_all_prev_year_total_amount_avg` and `tx_prev_month_all_prev_year_montly_freq_avg`, as these customers are directly excluded by the first `MATCH` clause.

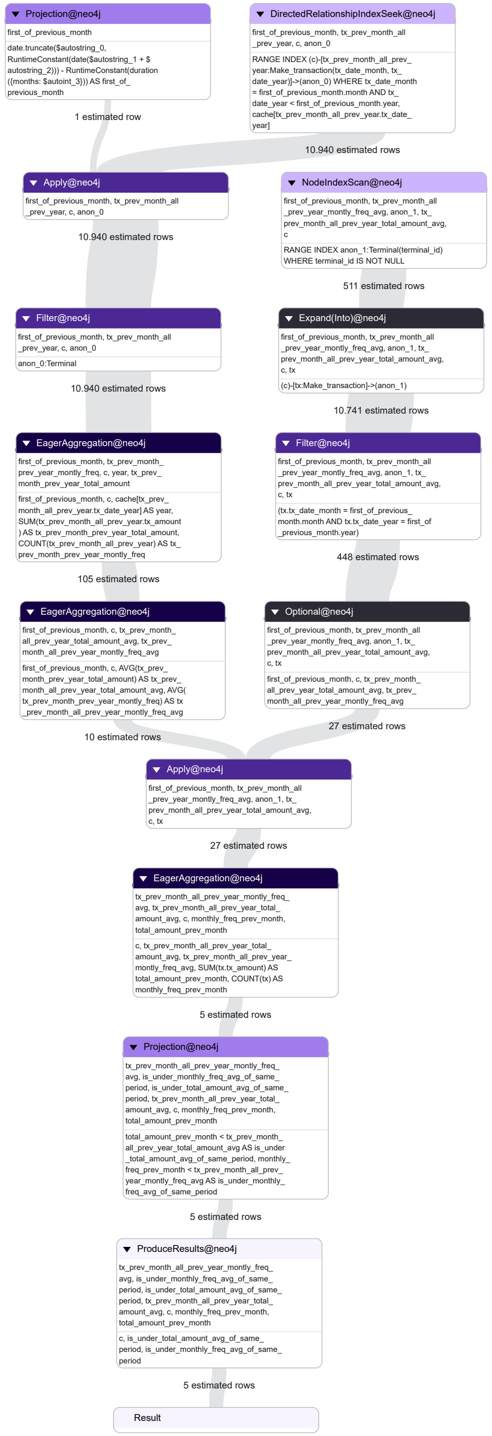
```
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_a2(year_and_month_under_analesis):
    query = f"""
            WITH date.truncate('month', date("{year_and_month_under_analesis}" + "-01") ) - duration({{months: 1}}) AS first_of_previous_month

            MATCH (c)-[tx_prev_month_all_prev_year:Make_transaction]->(:Terminal)
            WHERE
                tx_prev_month_all_prev_year.tx_date_month = first_of_previous_month.month
                AND tx_prev_month_all_prev_year.tx_date_year < first_of_previous_month.year
            WITH
                first_of_previous_month,
                c,
                tx_prev_month_all_prev_year.tx_date_year as year,
                SUM(tx_prev_month_all_prev_year.tx_amount)  AS tx_prev_month_prev_year_total_amount,
                COUNT(tx_prev_month_all_prev_year) AS tx_prev_month_prev_year_montly_freq
            WITH
            first_of_previous_month,
            c,
            AVG(tx_prev_month_prev_year_total_amount) AS tx_prev_month_all_prev_year_total_amount_avg,
            AVG(tx_prev_month_prev_year_montly_freq) AS tx_prev_month_all_prev_year_montly_freq_avg

            OPTIONAL MATCH (c)-[tx:Make_transaction]->(:Terminal)
            WHERE
                tx.tx_date_month = first_of_previous_month.month AND
                tx.tx_date_year = first_of_previous_month.year
            WITH
                c,
                SUM(tx.tx_amount) AS total_amount_prev_month,
                COUNT(tx) AS monthly_freq_prev_month,
                tx_prev_month_all_prev_year_total_amount_avg,
                tx_prev_month_all_prev_year_montly_freq_avg

            RETURN
                c,
                total_amount_prev_month < tx_prev_month_all_prev_year_total_amount_avg  AS is_under_total_amount_avg_of_same_period,
                monthly_freq_prev_month < tx_prev_month_all_prev_year_montly_freq_avg AS is_under_monthly_freq_avg_of_same_period
            """

    return execute_query_df("query_a2",query)
query_a2(month_and_year_under_analesis)
```

```
query_a2 execution time: 1.75s

                                                     c  is_under_total_amount_avg_of_same_period  is_under_monthly_freq_avg_of_same_period
0    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                      True
1    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                      True
2    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                      True                                     False
3    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                      True                                     False
4    (mean_amount, x_customer_id, mean_nb_tx_per_da…                                      True                                      True
..                                                 …                                       …                                       …
190  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                     False
191  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                      True
192  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                      True                                     False
193  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                     False
194  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                     False                                     False

[195 rows x 3 columns]
```

5.1.5 A2 Performances

As shown in the execution plan image below, the query now uses the index we created specifically for filtering transactions. Unlike the initial version, which did not use an index on the transactions, this optimized approach ensures that the query uses the index effectively to improve performance during the filtering process.

```
display(SVG(filename="./assets/Execution plan query A2.svg"))
```



Projection@neo4j
first_of_previous_month
date.truncate($autostring_0, RuntimeConstant(date($autostring_1 + $autostring_2))) - RuntimeConstant(duration({months: $autoint_3})) AS first_of_previous_month
1 estimated row
DirectedRelationshipIndexSeek@neo4j
first_of_previous_month, tx_prev_month_all_prev_year, c, anon_0
RANGE INDEX (c)-[tx_prev_month_all_prev_year:Make_transaction(tx_date_month, tx_date_year)]->(anon_0) WHERE tx_date_month = first_of_previous_month.month AND tx_date_year < first_of_previous_month.year, cache[tx_prev_month_all_prev_year.tx_date_year]
10.940 estimated rows
Apply@neo4j
first_of_previous_month, tx_prev_month_all_prev_year, c, anon_0
10.940 estimated rows
NodeIndexScan@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c
RANGE INDEX anon_1:Terminal(terminal_id) WHERE terminal_id IS NOT NULL
511 estimated rows
Filter@neo4j
first_of_previous_month, tx_prev_month_all_prev_year, c, anon_0
anon_0:Terminal
10.940 estimated rows
Expand(Into)@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
(c)-[tx:Make_transaction]->(anon_1)
10.741 estimated rows
EagerAggregation@neo4j
first_of_previous_month, tx_prev_month_prev_year_montly_freq, c, year, tx_prev_month_prev_year_total_amount
first_of_previous_month, c, cache[tx_prev_month_all_prev_year.tx_date_year] AS year, SUM(tx_prev_month_all_prev_year.tx_amount) AS tx_prev_month_prev_year_total_amount, COUNT(tx_prev_month_all_prev_year) AS tx_prev_month_prev_year_montly_freq
105 estimated rows
Filter@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
(tx.tx_date_month = first_of_previous_month.month AND tx.tx_date_year = first_of_previous_month.year)
448 estimated rows
EagerAggregation@neo4j
first_of_previous_month, c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg
first_of_previous_month, c, AVG(tx_prev_month_prev_year_total_amount) AS tx_prev_month_all_prev_year_total_amount_avg, AVG(tx_prev_month_prev_year_montly_freq) AS tx_prev_month_all_prev_year_montly_freq_avg
10 estimated rows
Optional@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
first_of_previous_month, c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg
27 estimated rows
Apply@neo4j
first_of_previous_month, tx_prev_month_all_prev_year_montly_freq_avg, anon_1, tx_prev_month_all_prev_year_total_amount_avg, c, tx
27 estimated rows
EagerAggregation@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
c, tx_prev_month_all_prev_year_total_amount_avg, tx_prev_month_all_prev_year_montly_freq_avg, SUM(tx.tx_amount) AS total_amount_prev_month, COUNT(tx) AS monthly_freq_prev_month
5 estimated rows
Projection@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, is_under_monthly_freq_avg_of_same_period, is_under_total_amount_avg_of_same_period, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
total_amount_prev_month < tx_prev_month_all_prev_year_total_amount_avg AS is_under_total_amount_avg_of_same_period, monthly_freq_prev_month < tx_prev_month_all_prev_year_montly_freq_avg AS is_under_monthly_freq_avg_of_same_period
5 estimated rows
ProduceResults@neo4j
tx_prev_month_all_prev_year_montly_freq_avg, is_under_monthly_freq_avg_of_same_period, is_under_total_amount_avg_of_same_period, tx_prev_month_all_prev_year_total_amount_avg, c, monthly_freq_prev_month, total_amount_prev_month
c, is_under_total_amount_avg_of_same_period, is_under_monthly_freq_avg_of_same_period
5 estimated rows
Result

5.2 Query B

5.2.1 Query Request

"For each terminal identify the possible fraudulent transactions. The fraudulent transactions are those whose import is higher than 20% of the maximal import of the transactions executed on the same terminal in the last month".

- "For each terminal": This means that the query results must include all terminals, even those for which it is not possible to identify fraudulent transactions.
- "Last month": Refers to data from the month preceding the specified date provided as a parameter. Similar to the previous query, this one is parameterized by passing a partial date in the "yyyy-MM" format to Python. The `first_of_previous_month` variable is then calculated to represent the first day of the previous month relative to the given date. Additionally, the query uses the `today` variable, which holds the first day of the current month, for further calculations or filtering.

5.2.2 B1 query code

The query begins by completing the partial date provided as input with the first day of the passed month, storing it in the `today` variable, and calculating the first day of the previous month, which is stored in the `first_of_previous_month` variable.

Next, all terminals are matched to ensure that none are excluded from the final result of the query. This is done because the following `WHERE` clauses do not filter out any terminals, and all subsequent matches are `OPTIONAL MATCH`.

The first `OPTIONAL MATCH` retrieves transactions made on terminals during the month and year corresponding to `first_of_previous_month`. These transactions are stored in the variable `tx_prev_month`. However, some terminals may not have any transactions for the specified period, in which case `tx_prev_month` will be empty for those terminals.

The query then calculates the fraud detection threshold using a `WITH` statement. The fraud amount limit, stored in the variable `tx_amount_fraud_limit`, is defined as 20% above the maximum transaction amount from the previous month. For terminals where no transactions were found in `tx_prev_month`, the fraud amount limit is `NULL`.

The next step uses an `OPTIONAL MATCH` clause to retrieve transactions from the current month by filtering based on the same month and year as the `today` variable. These transactions are stored in the `tx_current_month` variable. Then the query uses the previously calculated `tx_amount_fraud_limit` to identify fraudulent transactions. It collects transactions from `tx_current_month` where the transaction amount exceeds `tx_amount_fraud_limit`, storing these in the `fraud_txs_current_month` collection. If `tx_amount_fraud_limit` is `NULL`, the condition always evaluates to `false`, resulting in an empty collection for that terminal.

The `RETURN` statement at the end of the query enables distinguishing between two specific scenarios when a terminal has an empty `fraud_txs_current_month` collection:

- the fraud amount limit could not be calculated, making it impossible to determine whether the terminal had any fraudulent transactions;
- the fraud amount limit was calculated, but no fraudulent transactions were identified for that terminal in the current month.

To address these scenarios, the query substitutes empty collections in `fraud_txs_current_month` with `NULL` whenever `tx_amount_fraud_limit IS NULL`.

```
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_b1(year_and_month_under_analesis):
    query = f"""
            WITH date("{year_and_month_under_analesis}" + "-01") AS today
            WITH today, date.truncate('month', today ) - duration({{months: 1}}) AS first_of_previous_month

            MATCH (t:Terminal)

            OPTIONAL MATCH (:Customer)-[tx_prev_month:Make_transaction]->(t)
            WHERE
                tx_prev_month.tx_date_month = first_of_previous_month.month
                AND tx_prev_month.tx_date_year = first_of_previous_month.year
```

```
        with today, t, max(tx_prev_month.tx_amount) * 1.2 as tx_amount_fraud_limit

        OPTIONAL MATCH (:Customer)-[tx_current_month:Make_transaction]->(t)
        WHERE
            tx_current_month.tx_date_month = today.month
            AND tx_current_month.tx_date_year = today.year

        WITH
            t,
            tx_amount_fraud_limit,
            COLLECT(CASE
                WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN tx_current_month
                ELSE NULL
            END) AS fraud_txs_current_month

        RETURN
            t,
            CASE
                WHEN tx_amount_fraud_limit IS NULL THEN NULL
                ELSE fraud_txs_current_month
            END AS fraud_txs_current_month
        """

    return execute_query_df("query_b1",query)
query_b1(month_and_year_under_analesis)
```

```
query_b1 execution time: 3.02s

                                             t fraud_txs_current_month
0   (y_terminal_id, terminal_id, x_terminal_id)                      []
1   (y_terminal_id, terminal_id, x_terminal_id)                      []
2   (y_terminal_id, terminal_id, x_terminal_id)                      []
3   (y_terminal_id, terminal_id, x_terminal_id)                      []
4   (y_terminal_id, terminal_id, x_terminal_id)                      []
..                                           …                      …
45  (y_terminal_id, terminal_id, x_terminal_id)                      []
46  (y_terminal_id, terminal_id, x_terminal_id)                      []
47  (y_terminal_id, terminal_id, x_terminal_id)                      []
48  (y_terminal_id, terminal_id, x_terminal_id)                      []
49  (y_terminal_id, terminal_id, x_terminal_id)                      []

[50 rows x 2 columns]
```

5.2.3 B1 Performance

To improve the performance of the query, since it filters the data on `make_transaction.tx_date_month` and `make_transaction.tx_date_year`, we can reuse the composite index previously created.

As we can see in the execution plan of the query shown below, the same behavior observed in the previous query occurs here as well. In particular, the first `MATCH` clause, which matches all terminals, prevents the index from being used to filter the transactions.

In fact, the only index used is on the terminals, and it is only used to retrieve all the terminal nodes without performing any filtering. As for the transactions, no index is used either in the initial filtering or in the subsequent `OPTIONAL MATCH`, which further contributes to the inefficiency of the query.

```
display(SVG(filename="./assets/Execution plan query B1.svg"))
```


NodeIndexScan@neo4j
t, first_of_previous_month, today, anon_0
RANGE INDEX anon_0:Customer(customer_id) WHERE customer_id IS NOT NULL
10.000 estimated rows
Expand(Into)@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
(anon_0)-[tx_prev_month:Make_transaction]->(t)
262.563 estimated rows
Filter@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
(tx_prev_month.tx_date_month = first_of_previous_month.month AND tx_prev_month.tx_date_year = first_of_previous_month.year)
10.940 estimated rows
Projection@neo4j
today
RuntimeConstant(date($autostring_0 + $autostring_1)) AS today
1 estimated row
NodeIndexScan@neo4j
today, first_of_previous_month, t
RANGE INDEX t:Terminal(terminal_id) WHERE terminal_id IS NOT NULL, cache[t.terminal_id]
50 estimated
Optional@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
today, first_of_previous_month, t
656 estimated rows
Projection@neo4j
today, first_of_previous_month
date.truncate($autostring_2, today) - RuntimeConstant(duration({months: $autoint_3})) AS first_of_previous_month
1 estimated row
Apply@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
656 estimated rows
NodeIndexScan@neo4j
today, t, tx_amount_fraud_limit, anon_1
RANGE INDEX anon_1:Customer(customer_id) WHERE customer_id IS NOT NULL
5.124 estimated rows
Apply@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
656 estimated rows
Expand(Into)@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
(anon_1)-[tx_current_month:Make_transaction]->(t)
134.540 estimated rows
EagerAggregation@neo4j
today, t, anon_4
today, t, max(tx_prev_month.tx_amount) AS anon_4
26 estimated rows
Filter@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
(tx_current_month.tx_date_month = today.month AND tx_current_month.tx_date_year = today.year)
5.606 estimated rows
Projection@neo4j
today, t, anon_4, tx_amount_fraud_limit
anon_4 * $autodouble_4 AS tx_amount_fraud_limit
26 estimated rows
Optional@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
today, t, tx_amount_fraud_limit
336 estimated rows
Apply@neo4j
t, anon_1, tx_current_month, today, anon_4, tx_amount_fraud_limit
336 estimated rows
EagerAggregation@neo4j
t, tx_amount_fraud_limit, fraud_txs_current_month
t, tx_amount_fraud_limit, COLLECT(CASE WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN tx_current_month ELSE NULLEND) AS fraud_txs_current_month
18 estimated rows
Projection@neo4j
t, tx_amount_fraud_limit, fraud_txs_current_month
CASE WHEN tx_amount_fraud_limit IS NULL THEN NULL ELSE fraud_txs_current_month END AS fraud_txs_current_month
18 estimated rows
ProduceResults@neo4j
t, tx_amount_fraud_limit, fraud_txs_current_month
t, fraud_txs_current_month
18 estimated rows
Result

5.2.4) B2 Query Code By slightly modifying the query to omit the "for all terminals" clause and display only terminals with `tx_amount_fraud_limit`, we can improve performance by using the index. This tweak involves removing the first `MATCH` clause and changing the second `OPTIONAL MATCH` to a regular `MATCH`.

This change means that the results will no longer include terminals with `NULL` values in the `fraud_txs_current_month` column, as these terminals are directly excluded by the first `MATCH` clause.

```
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_b2(year_and_month_under_analesis):
    query = f"""
            WITH date("{year_and_month_under_analesis}" + "-01") AS today
            WITH today, date.truncate('month', today ) - duration({{months: 1}}) AS first_of_previous_month

            MATCH (:Customer)-[tx_prev_month:Make_transaction]->(t:Terminal)
            WHERE
                tx_prev_month.tx_date_month = first_of_previous_month.month
                AND tx_prev_month.tx_date_year = first_of_previous_month.year

            with today, t, max(tx_prev_month.tx_amount) * 1.2 as tx_amount_fraud_limit

            OPTIONAL MATCH (:Customer)-[tx_current_month:Make_transaction]->(t)
            WHERE
                tx_current_month.tx_date_month = today.month
                AND tx_current_month.tx_date_year = today.year

            RETURN
                t,
                COLLECT(
                    CASE
                        WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN tx_current_month
                        ELSE NULL
                    END
                )AS fraud_txs_current_month
            """

    return execute_query_df("query_b2",query)
query_b2(month_and_year_under_analesis)
```

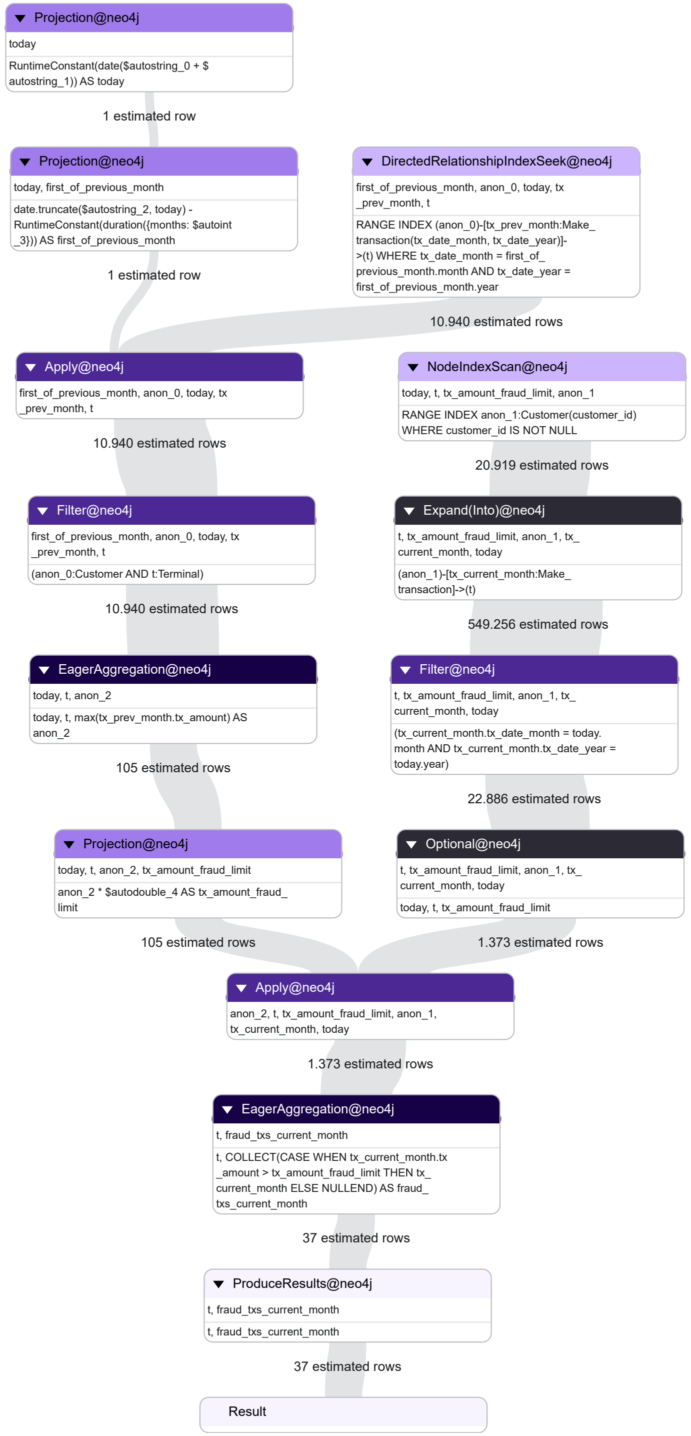
```
query_b2 execution time: 1.86s

                                              t                              fraud_txs_current_month
0   (y_terminal_id, terminal_id, x_terminal_id)                                                    []
1   (y_terminal_id, terminal_id, x_terminal_id)                                                    []
2   (y_terminal_id, terminal_id, x_terminal_id)                                                    []
3   (y_terminal_id, terminal_id, x_terminal_id)                                                    []
4   (y_terminal_id, terminal_id, x_terminal_id)  [(transaction_id, tx_date_month, tx_amount, tx…
..                                            …                                                 …
45  (y_terminal_id, terminal_id, x_terminal_id)                                                    []
46  (y_terminal_id, terminal_id, x_terminal_id)                                                    []
```

```
47  (y_terminal_id, terminal_id, x_terminal_id)                                              []
48  (y_terminal_id, terminal_id, x_terminal_id)                                              []
49  (y_terminal_id, terminal_id, x_terminal_id)                                              []

[50 rows x 2 columns]
```

5.2.4 B2 Execution

As shown in the execution plan image below, the query now uses the index we created specifically for filtering transactions. Unlike the initial version, where no index was used on the transactions, this optimized approach ensures that the query uses the index effectively to improve performance during the filtering process.

```
display(SVG(filename="./assets/Execution plan query B2.svg"))
```


Projection@neo4j
today
RuntimeConstant(date($autostring_0 + $autostring_1)) AS today
1 estimated row
Projection@neo4j
today, first_of_previous_month
date.truncate($autostring_2, today) - RuntimeConstant(duration({months: $autoint_3})) AS first_of_previous_month
1 estimated row
DirectedRelationshipIndexSeek@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
RANGE INDEX (anon_0)-[tx_prev_month:Make_transaction(tx_date_month, tx_date_year)]->(t) WHERE tx_date_month = first_of_previous_month.month AND tx_date_year = first_of_previous_month.year
10.940 estimated rows
Apply@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
10.940 estimated rows
NodeIndexScan@neo4j
today, t, tx_amount_fraud_limit, anon_1
RANGE INDEX anon_1:Customer(customer_id) WHERE customer_id IS NOT NULL
20.919 estimated rows
Filter@neo4j
first_of_previous_month, anon_0, today, tx_prev_month, t
(anon_0:Customer AND t:Terminal)
10.940 estimated rows
Expand(Into)@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
(anon_1)-[tx_current_month:Make_transaction]->(t)
549.256 estimated rows
EagerAggregation@neo4j
today, t, anon_2
today, t, max(tx_prev_month.tx_amount) AS anon_2
105 estimated rows
Filter@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
(tx_current_month.tx_date_month = today.month AND tx_current_month.tx_date_year = today.year)
22.886 estimated rows
Projection@neo4j
today, t, anon_2, tx_amount_fraud_limit
anon_2 * $autodouble_4 AS tx_amount_fraud_limit
105 estimated rows
Optional@neo4j
t, tx_amount_fraud_limit, anon_1, tx_current_month, today
today, t, tx_amount_fraud_limit
1.373 estimated rows
Apply@neo4j
anon_2, t, tx_amount_fraud_limit, anon_1, tx_current_month, today
1.373 estimated rows
EagerAggregation@neo4j
t, fraud_txs_current_month
t, COLLECT(CASE WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN tx_current_month ELSE NULLEND) AS fraud_txs_current_month
37 estimated rows
ProduceResults@neo4j
t, fraud_txs_current_month
t, fraud_txs_current_month
37 estimated rows
Result

5.3 Query C

5.3.1 Query request

"Given a user u, determine the "co-customer-relationships CC of degree k". A user u' is a co-customer of u if you can determine a chain "u1-t1-u2-t2-...tk-1-uk" such that u1=u, uk=u', and for each 1<=I,j<=k, ui <> uj, and t1,..tk-1 are the terminals on which a transaction has been executed. Therefore, CCk(u)={u'| a chain exists between u and u' of degree k}. Please, note that depending on the adopted model, the computation of CCk(u) could be quite complicated. Consider therefore at least the computation of CC3(u) (i.e. the co-costumer relationships of degree 3)".

This request is very precise and needs no further elaboration. What I would like to emphasize is the proposed solution, which uses an APOC function for efficient graph traversal. This approach will prove to be highly efficient.

5.3.2 C query code

The Python function that executes the query takes two parameters: `customer_id`, representing the starting customer, and `k`, representing the degree of the co-customer. The query uses `APOC.path.expandConfig` function to efficiently explore relationships up to a specified level. Starting from the customer node with same `customer_id` as the passed one, it navigates through `make_transaction` relationships to `terminal` or other `customer` nodes.

Looking at the `APOC.path.expandConfig` parameters:

- the `relationshipFilter` specifies which relationships can be traversed based on their type;
- the `labelFilter` defines which nodes can be traversed based on their label;
- the `maxLevel` parameter limits the exploration depth, ensuring only paths with a length `k` are returned;
- the `uniqueness` parameter defines the level of uniqueness for nodes in the path; when set to `'NODE_GLOBAL'`, it ensures that each node in the path appears only once.

To focus only on paths of exact length `k`, a `WHERE` clause filters the results after the `WITH` clause. At the end, the `RETURN` statement selects only the last node in each qualified path that represents the desired co-customer of interest.

The `k` passed to the Python function is chenged in the query because the `maxLevel` parameter must specify the maximum number of nodes in the path. Since each co-customer needs a terminal between itself and the immediately next co-customer, the Python `k` becomes `(k - 1) * 2` in the query.

```python
#customer_id is an integer that indicates the customer_id property of :Customer
#k is an integer that indicates the different customers involved in the chain described in the project track
def query_c(customer_id, k):
    query = f"""
            WITH {k-1} * 2 AS k
            MATCH (start:Customer {{customer_id: {customer_id}}})
            CALL apoc.path.expandConfig(start, {{
                relationshipFilter: 'Make_transaction',
                labelFilter: 'Terminal|Customer',
                maxLevel: k,
                uniqueness: 'NODE_GLOBAL'
            }}) YIELD path

            WITH path
            WHERE length(path) = k
            RETURN nodes(path)[-1].customer_id AS CO_Customer
            """
    return execute_query_df("query_c",query)
```

```
query_c(1, 3)
```

```
query_c execution time: 0.46s

    CO_Customer
0             9
1            11
2            66
3           134
4           154
..          …
48           40
49           61
50           97
51          151
52          178

[53 rows x 1 columns]
```

5.3.3 C Performance

I was pleasantly surprised by the performance of this solution. Before its design, I had tried several approaches with very poor results. In fact, even calculating CC3(…) tooks an enormous amount of time. Attempts with k > 3 resulted in no response, likely due to the excessive computation time required.

The query is also highly efficient because by using the `uniqueness: 'NODE_GLOBAL'` many paths are discarded, significantly reducing the number of possible paths because customers and terminals must be unique within the path.

With the proposed solution, however, it is possible to go well beyond `k = 3` while still maintaining remarkably low execution times, as indicated below where CC8(5) is calculated in half a second.

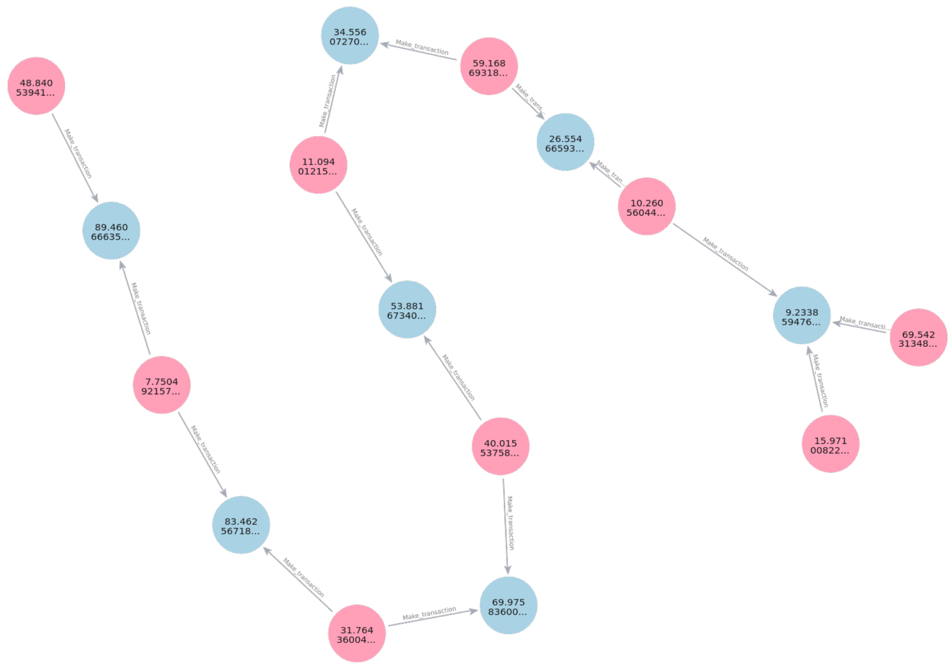
```
query_c(5, 8)
```

```
query_c execution time: 0.59s

   CO_Customer
0           51
1          104
```

To visualize the chains of customers and terminals, I ran a slightly modified version of the query in the Neo4j console so that it would return the paths related to CC8(5).

The data displayed inside the nodes in the image is not particularly meaningful, as it shows one of the properties of the nodes, which in this case is not relevant to the visualization.

```
display(Image(filename="./assets/Query C path.png"))
```



5.4 Query D

5.4.1 Query request

"i. Each transaction should be extended with:

1. The period of the day {morning, afternoon, evening, night} in which the transaction has been executed.

2. The kind of products that have been bought through the transaction {hightech, food, clothing, consumable, other}.

3. The feeling of security expressed by the user. This is an integer value between 1 and 5 expressed by the user when conclude the transaction.

The values can be chosen randomly.

ii. Customers that make more than three transactions from the same terminal expressing a similar average feeling of security should be connected as "buying_friends". Therefore also this kind of relationship should be explicitly stored in the NOSQL database and can be queried. Note, two average feelings of security are considered similar when their difference is lower than 1".

The query is clear and leaves no room for alternative interpretations, so there is no need to explain it further. For simplicity, we will split this query into two separate queries: `query_di`, which performs point i, and `query_dii`, which performs point ii.

The approach for both queries is similar, as both use `APOC.periodic.iterate` function, which allows batch tasks to be defined and executed in parallel, similar to the `CALL{} IN TRANSACTIONS OF ... ROWS`. The `iterate` function takes three parameters: the query to be run, the size of the batch, and whether the task should be run in parallel.

5.4.2 Di query code

The `query_di` process has been split into two distinct queries, each handled by its own Python function:

- `query_di()` function executes the core query, which uses the `iterate` function to modify the data. It retrieves all transactions with the `MATCH` clause and adds the three requested properties. These properties are selected randomly using the `CASE` function, with conditions determined by the `rand()` function.
- The `create_transaction_extended_schema()` function executes the query to add constraints for the new properties in the transactions schema. Unlike the data loading process, schema creation is performed after data modification. This sequence is necessary because the schema creation would fail if attempted before the execution of `query_di()`, as the existing transaction data lacks the new values required to satisfy the constraints.

```python
def query_di():
    query = f"""
        CALL apoc.periodic.iterate(
            'MATCH (c:Customer)-[transaction:Make_transaction]->(t:Terminal)
            RETURN transaction',
            'SET transaction.tx_day_period = CASE toInteger(rand() * 4)
                                            WHEN 0 THEN "morning"
                                            WHEN 1 THEN "afternoon"
                                            WHEN 2 THEN "evening"
                                            ELSE "night"
                                        END,
                transaction.tx_products_type = CASE toInteger(rand() * 5)
                                                WHEN 0 THEN "high-tech"
                                                WHEN 1 THEN "food"
                                                WHEN 2 THEN "clothing"
                                                WHEN 3 THEN "consumable"
                                                ELSE "other"
                                            END,
                transaction.tx_security_feeling = toInteger(rand() * 5) + 1',
            {{batchSize: {config["lines_per_commit_apoc"]}, parallel: {config["parallel_loading"]}}}
        )
    """
    return execute_query_commands("query_di", [query])

def create_transaction_extended_schema():
    queries = [
        "CREATE CONSTRAINT tx_day_period_is_string FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_day_period IS :: STRING;",
        "CREATE CONSTRAINT tx_day_period_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_day_period IS NOT NULL;",
        "CREATE CONSTRAINT tx_products_type_is_string FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_products_type IS :: STRING;",
        "CREATE CONSTRAINT tx_products_type_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_products_type IS NOT NULL;",
        "CREATE CONSTRAINT tx_security_feeling_is_integer FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_security_feeling IS :: 
↪INTEGER;",
        "CREATE CONSTRAINT tx_security_feeling_required FOR ()-[transaction:Make_transaction]->() REQUIRE transaction.tx_security_feeling IS NOT NULL;
↪",
    ]
    return execute_query_commands("create_transaction_extended_schema", queries)
```

```
query_di()
create_transaction_extended_schema()
```

```
query_di execution time: 19.26s
create_transaction_extended_schema execution time: 1.82s
```

```
True
```

5.4.3 Dii Query Code

The query begins with the first `MATCH`, identifying all customers `c1` who have made at least three transactions at a terminal `t` and calculates the average of the `tx_security_feeling` property for these transactions, storing the result in `avg_tx1_security_feeling`. It then searches for other customers `c2` who have also made at least three transactions at the same terminal, calculating their average `tx_security_feeling` and storing it in `avg_tx2_security_feeling`.

Once the pairs of customers `c1` and `c2` sharing the same terminal with at least 3 transactions are identified, the query checks whether the absolute difference between their average security feelings values are less than 1. This condition ensures that the two customers have similar transaction security experiences at the same terminal. If the condition is met, the query creates a `buying_friends` relationship between the two customers.

Since `buying_friends` is a symmetric relationship, the condition `c1 < c2` is used to ensure that the relationship is created only once for each pair. This prevents duplicate relationships from being formed (e.g., both `c1 -> c2` and `c2 -> c1`).

```
def query_dii():
    query = f"""
        CALL apoc.periodic.iterate(
            '
                MATCH (c1:Customer)-[tx1:Make_transaction]->(t:Terminal)
                WITH c1, t, COUNT(tx1) AS count_tx1, avg(tx1.tx_security_feeling) as avg_tx1_security_feeling
                WHERE count_tx1 > 3

                MATCH (c2:Customer)-[tx2:Make_transaction]->(t:Terminal)
                WITH c1, c2, t, avg_tx1_security_feeling, COUNT(tx2) AS count_tx2, avg(tx2.tx_security_feeling) as avg_tx2_security_feeling
                WHERE
                    count_tx2 > 3 AND
                    c1 < c2 AND
                    (abs(avg_tx1_security_feeling - avg_tx2_security_feeling) < 1)

                RETURN c1, c2
            ',
            '
                MERGE (c1)-[:buying_friends]-(c2)
            ',
            {{batchSize: {config["lines_per_commit_apoc"]}, parallel: {config["parallel_loading"]}}}
        )
    """
    return execute_query_commands("query_dii",[query])

query_dii()
```

```
query_dii execution time: 18.43s
```

```
True
```

5.4.4 Di and Dii Performances

Both queries are structured in the same way, relying on the `APOC.periodic.iterate` function, passing two different queries. Since the execution plan does not provide useful information, given that all operations are performed within a single `APOC.periodic.iterate` block, for both queries it has been omitted.

- The `query_di` proves to be highly efficient, as it is capable of modifying all transactions in significantly less time than it took to load them into the database. In fact, it only takes a small fraction of the loading time, thanks to the parallelized processing of the `APOC.periodic.iterate` batches.
- The `query_dii`, while taking a considerable amount of time to complete, still performs its task efficiently considering the large volume of work required to identify the `buying_friends`. The identification process itself is quite costly, and I haven't found an alternative way to make it faster.

5.5 Query E

5.5.1 Query Request

"For each period of the day identifies the number of transactions that occurred in that period, and the average number of fraudulent transactions"

- "For each period of the day": The query result must contain 4 rows, one for each possible value of `Make_transaction.tx_day_period`. Since the detection of fraudulent transactions for a given month relies on data from the previous month (as seen in query B), it is practical to run this query only considering transactions executed after a specified `startMonthYear` and, for completeness, before a given `endMonthYear`. In this way, if a `startMonthYear` is provided and there are data in the database from the previous month, it becomes possible to calculate the fraudulent transactions for transactions with the same `tx_date_year` and `tx_date_month` as those expressed by `startMonthYear`. If the `startMonthYear` is not provided, it would always be impossible to detect fraudulent transactions for the first month and first year transactions in the database because there would be no data available from the preceding month. If it is not possible to calculate fraudulent transactions for a month, they will be included as 0 (indicating the absence of fraudolent transactions) in the average calculation.
- "the number of transactions": This means that for each `Make_transaction.tx_day_period`, you need to count the number of transactions registered after `startMonthYear` and before `endMonthYear`.
- "the average number of fraudulent transactions": means calculating the **montly count** average of fraudulent transactions registered after `startMonthYear` and before `endMonthYear` for each desired `Make_transaction.tx_day_period`."

5.5.2 E1 query code

The query starts by setting the `startDate` and `endDate` variables to the first day of the month and year of the Python variables `startMonthYear` and `endMonthYear`, each of which contains a date in the format yyyy-MM. If the Python variables are empty strings, the corresponding query variables are set to `NULL`. This ensures that they are not used to filter the data in the subsequent `WHERE` clause. This approach allows the interval to be partially or completely unspecified, addressing the previously described issue of detecting fraudulent transactions in the first month and year of transactions in the database.

The first `MATCH` clause extracts all transactions and the subsequent `WHERE` clause filters these transactions, keeping only those within the specified interval storing them in the `tx` variable.

The next `WITH` aggregates the `tx` transactions based on the triple (`tx.tx_date_year`, `tx.tx_date_month`, `t`), where `t` is the terminal, and calculates the `tx_amount_fraud_limit` for each of these tuples. It's important to note that the grouping doesn't use the `tx.tx_date_year` and `tx.tx_date_month` directly; instead, it uses a date object created from these two fields, but referring to the first day of the following month. This is because the `tx_amount_fraud_limit` needs to be calculated based on transactions from the previous month, so the `tx_amount_fraud_limit` values we calculate are for the following month.

At this stage we have the `tx_amount_fraud_limit` for each triple (`year`, `month`, `t`). Therefore, we can proceed to count the total number of transactions and fraudulent transactions associated with each daily period storing them in the variables `tx_count` and `tx_fraud_count` respectively. To achieve this, we use a second `MATCH` clause to extract the transactions corresponding to the same `t` and filter them using the `WHERE` clause keeping only those transactions with the same `year` and `month` as in the triple, storing them in the variable `tx_current_month`. Then, using the `WITH` clause, we group by the quadruple (`year`, `month`, `t`, `tx_current_month.tx_day_period`), counting the number of transactions in the `tx_count` variable and also counting the number of fraudulent transactions, defined as those where `tx_current_month.tx_amount > tx_amount_fraud_limit`, and storing the result in the `tx_fraud_count` variable.

At the end, the `RETURN` clause aggregates the data by only `tx_current_month.tx_day_period`, summing the `tx_count` values into `total_transactions` and calculating the montly count average of the `tx_fraud_count` values as `monthly_avg_fraud_transactions`.

```
#startMonthYear is a string that contains an year and a month in the format yyyy-MM, it could be "" to not filter the results from a starting point
#endMonthYear is a string that contains an year and a month in the format yyyy-MM, it could be "" to not filter the results from an ending point
#the filtering is [startMonthYear, endMonthYear]
def query_e1(startMonthYear, endMonthYear):
    query = f"""
            WITH
            CASE
                WHEN "{startMonthYear}" = "" THEN NULL
                ELSE date("{startMonthYear}" + "-01")
            END AS startDate,
            CASE
                WHEN "{endMonthYear}" = "" THEN NULL
                ELSE date("{endMonthYear}" + "-01")
            END AS endDate

            MATCH (:Customer)-[tx:Make_transaction]->(t:Terminal)
            WHERE
                 (startDate IS NULL OR (tx.tx_date_year >= startDate.year OR (tx.tx_date_year = startDate.year AND tx.tx_date_month >= startDate.
 ↪month))) AND
                 (endDate IS NULL OR (tx.tx_date_year <= endDate.year OR (tx.tx_date_year = endDate.year AND tx.tx_date_month <= endDate.month)))

            WITH (date({{year: tx.tx_date_year, month: tx.tx_date_month, day: 1}}) + duration({{months: 1}})).year AS year,
                 (date({{year: tx.tx_date_year, month: tx.tx_date_month, day: 1}}) + duration({{months: 1}})).month AS month,
                 t,
                 max(tx.tx_amount) * 1.2 as tx_amount_fraud_limit

            MATCH (:Customer)-[tx_current_month:Make_transaction]->(t)
            WHERE
                tx_current_month.tx_date_month = month AND
                tx_current_month.tx_date_year = year

            WITH
                year,
                month,
                t,
                tx_current_month.tx_day_period as day_period,
                count(tx_current_month) as tx_count,
                count(
                    CASE
                        WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN 1
                        ELSE NULL
```

```
                END
            )AS tx_fraud_count

        RETURN day_period, sum(tx_count) AS total_transactions, avg(tx_fraud_count) AS monthly_avg_fraud_transactions
        """

    return execute_query_df("query_e1",query)

query_e1("2023-01" , month_and_year_under_analesis)
```

```
query_e1 execution time: 17.23s

  day_period  total_transactions  monthly_avg_fraud_transactions
0      night               48306                        0.345588
1    morning               28703                        0.230483
2    evening               15944                        0.117202
3  afternoon               21468                        0.123134
```

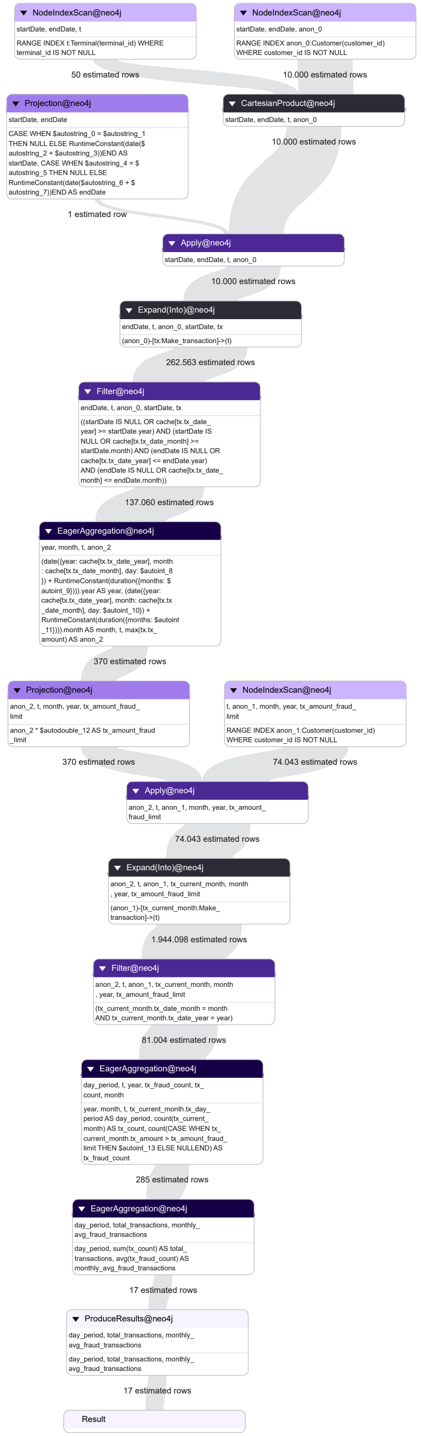
5.5.3 E1 Performances

This query is computationally intensive as it potentially operates on all the relationships (if the interval is not defined) of all the terminals in the DB. Roughly speaking, we can say that it is like running query B for each terminal and for each year and month within the defined interval, then grouping the data by `day_period` and performing the necessary counts and averages. Therefore, its execution time will definitely be equal to (in rare and specific circumstances) or greater than the execution time of query B.

During the development of this query, I expected that it would leverage the same composite index created to optimize Query A, as the filtering of transactions involves breaking down `startDate` and `endDate` into their year and month components. However, upon reviewing the execution plan, as shown below, it became clear that the composite index is not being utilized effectively due to the following reasons:

- the first `WHERE` condition does not always filter values within a defined range. If `startDate` or `endDate`, or both, are `NULL`, the filter ranges become incomplete or undefined, limiting the applicability of the index.
- Even when `startDate` and `endDate` are not `NULL`, the filter condition is not fully suitable for the composite index. This is because some clauses in the condition impose restrictions only on the `tx_date_year` field without including constraints on the `tx_date_month` field. Since the index is a composite index on both `tx_date_year` and `tx_date_month`, it cannot be used when the condition evaluates only `tx.tx_date_year >= startDate.year` or `tx.tx_date_year <= endDate.year`.

```
display(SVG(filename="./assets/Execution plan query E1.svg"))
```


NodeIndexScan@neo4j
startDate, endDate, t
RANGE INDEX t:Terminal(terminal_id) WHERE terminal_id IS NOT NULL
NodeIndexScan@neo4j
startDate, endDate, anon_0
RANGE INDEX anon_0:Customer(customer_id) WHERE customer_id IS NOT NULL
50 estimated rows
10.000 estimated rows
Projection@neo4j
startDate, endDate
CASE WHEN $autostring_0 = $autostring_1 THEN NULL ELSE RuntimeConstant(date($autostring_2 + $autostring_3))END AS startDate, CASE WHEN $autostring_4 = $autostring_5 THEN NULL ELSE RuntimeConstant(date($autostring_6 + $autostring_7))END AS endDate
CartesianProduct@neo4j
startDate, endDate, t, anon_0
10.000 estimated rows
1 estimated row
Apply@neo4j
startDate, endDate, t, anon_0
10.000 estimated rows
Expand(Into)@neo4j
endDate, t, anon_0, startDate, tx
(anon_0)-[tx:Make_transaction]->(t)
262.563 estimated rows
Filter@neo4j
endDate, t, anon_0, startDate, tx
((startDate IS NULL OR cache[tx.tx_date_year] >= startDate.year) AND (startDate IS NULL OR cache[tx.tx_date_month] >= startDate.month) AND (endDate IS NULL OR cache[tx.tx_date_year] <= endDate.year) AND (endDate IS NULL OR cache[tx.tx_date_month] <= endDate.month))
137.060 estimated rows
EagerAggregation@neo4j
year, month, t, anon_2
(date({year: cache[tx.tx_date_year], month : cache[tx.tx_date_month], day: $autoint_8 }) + RuntimeConstant(duration({months: $autoint_9}))).year AS year, (date({year: cache[tx.tx_date_year], month: cache[tx.tx_date_month], day: $autoint_10}) + RuntimeConstant(duration({months: $autoint_11}))).month AS month, t, max(tx.tx_amount) AS anon_2
370 estimated rows
Projection@neo4j
anon_2, t, month, year, tx_amount_fraud_limit
anon_2 * $autodouble_12 AS tx_amount_fraud_limit
NodeIndexScan@neo4j
t, anon_1, month, year, tx_amount_fraud_limit
RANGE INDEX anon_1:Customer(customer_id) WHERE customer_id IS NOT NULL
370 estimated rows
74.043 estimated rows
Apply@neo4j
anon_2, t, anon_1, month, year, tx_amount_fraud_limit
74.043 estimated rows
Expand(Into)@neo4j
anon_2, t, anon_1, tx_current_month, month, year, tx_amount_fraud_limit
(anon_1)-[tx_current_month:Make_transaction]->(t)
1.944.098 estimated rows
Filter@neo4j
anon_2, t, anon_1, tx_current_month, month, year, tx_amount_fraud_limit
(tx_current_month.tx_date_month = month AND tx_current_month.tx_date_year = year)
81.004 estimated rows
EagerAggregation@neo4j
day_period, t, year, tx_fraud_count, tx_count, month
year, month, t, tx_current_month.tx_day_period AS day_period, count(tx_current_month) AS tx_count, count(CASE WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN $autoint_13 ELSE NULLEND) AS tx_fraud_count
285 estimated rows
EagerAggregation@neo4j
day_period, total_transactions, monthly_avg_fraud_transactions
day_period, sum(tx_count) AS total_transactions, avg(tx_fraud_count) AS monthly_avg_fraud_transactions
17 estimated rows
ProduceResults@neo4j
day_period, total_transactions, monthly_avg_fraud_transactions
day_period, total_transactions, monthly_avg_fraud_transactions
17 estimated rows
Result

5.5.4 E2 query code

By removing the possibility of setting `startDate` and `endDate` to `NULL`, we enforced the definition of a fully specified interval. Additionally, I replaced the partial conditions that only filtered by the `tx_date_year` field with a more comprehensive condition by adding a universally true clause, `tx.tx_date_month >= 1`. This ensures that the filter always includes constraints on both the `tx_date_year` and `tx_date_month` fields, enabling the composite index on these fields to be effectively utilized.

```
#startMonthYear is a string that contains an year and a month in the format yyyy-MM
#endMonthYear is a string that contains an year and a month in the format yyyy-MM
#the filtering is [startMonthYear, endMonthYear]
def query_e2(startMonthYear, endMonthYear):
    query = f"""
            WITH
                date("{startMonthYear}" + "-01") AS startDate,
                date("{endMonthYear}" + "-01") AS endDate
            MATCH (:Customer)-[tx:Make_transaction]->(t:Terminal)
            WHERE
                (
                    tx.tx_date_year >= startDate.year AND tx.tx_date_month >= 1 OR
                    tx.tx_date_year = startDate.year AND tx.tx_date_month >= startDate.month
                )
                AND
                (
                    tx.tx_date_year <= endDate.year AND tx.tx_date_month >= 1 OR
                    tx.tx_date_year = endDate.year AND tx.tx_date_month <= endDate.month
                )

            WITH (date({{year: tx.tx_date_year, month: tx.tx_date_month, day: 1}}) + duration({{months: 1}})).year AS year,
                 (date({{year: tx.tx_date_year, month: tx.tx_date_month, day: 1}}) + duration({{months: 1}})).month AS month,
                 t,
                 max(tx.tx_amount) * 1.2 as tx_amount_fraud_limit

            MATCH (:Customer)-[tx_current_month:Make_transaction]->(t)
            WHERE
                tx_current_month.tx_date_month = month AND
                tx_current_month.tx_date_year = year

            WITH
                year,
                month,
                t,
                tx_current_month.tx_day_period as day_period,
                count(tx_current_month) as tx_count,
                count(
                    CASE
                        WHEN tx_current_month.tx_amount > tx_amount_fraud_limit THEN 1
                        ELSE NULL
                    END
```

```
            )AS tx_fraud_count

        RETURN day_period, sum(tx_count) AS total_transactions, avg(tx_fraud_count) AS monthly_avg_fraud_transactions
        """

    return execute_query_df("query_e2",query)

query_e2("2023-01" , month_and_year_under_analesis)
```

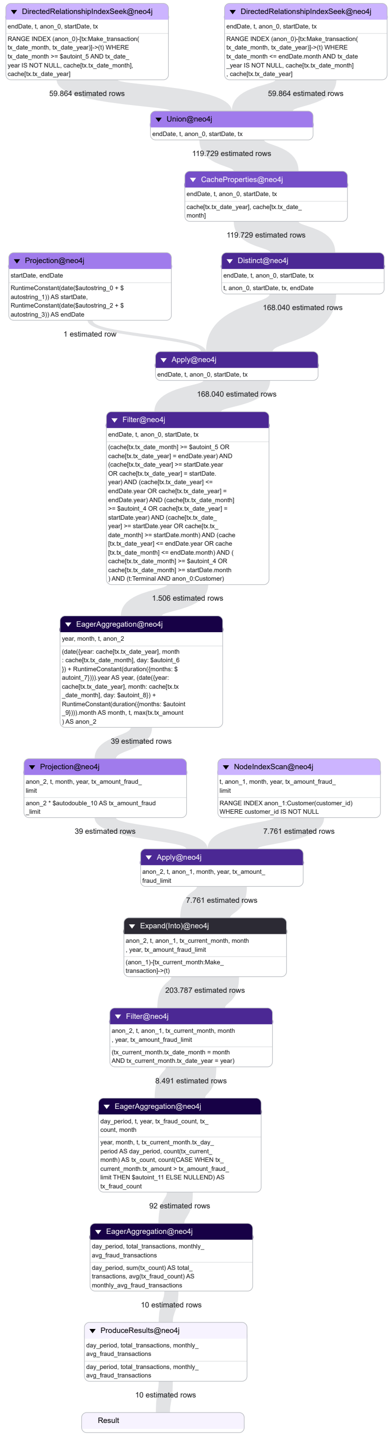
```
query_e2 execution time: 19.93s

  day_period  total_transactions  monthly_avg_fraud_transactions
0      night               48306                        0.345588
1    evening               15944                        0.117202
2    morning               28703                        0.230483
3  afternoon               21468                        0.123134
```

5.5.5 E2 Performances

From the execution plan shown below, we can see that the composite index is now being used. However, despite the index being utilized, the first version remains more efficient. This is likely because, upon reviewing the execution plans, the first version starts by analyzing the nodes, which are much smaller in scale compared to the relationships. In contrast, the second version begins by analyzing the relationships, performing two separate searches using the index, which leads to significantly slower performance. Therefore, the fact that the first version remains the best in terms of execution time can be attributed to the fact that starting with nodes in the first version leads to a more efficient query execution.

Since this second version is clearly more inefficient and cannot be considered an improvement over the first version, **it will not be taken into account in the final performance evaluation**.

```
display(SVG(filename="./assets/Execution plan query E2.svg"))
```


DirectedRelationshipIndexSeek@neo4j
endDate, t, anon_0, startDate, tx
RANGE INDEX (anon_0)-[tx:Make_transaction(tx_date_month, tx_date_year)]->(t) WHERE tx_date_month >= $autoint_5 AND tx_date_ year IS NOT NULL, cache[tx.tx_date_month], cache[tx.tx_date_year]
59.864 estimated rows
DirectedRelationshipIndexSeek@neo4j
endDate, t, anon_0, startDate, tx
RANGE INDEX (anon_0)-[tx:Make_transaction(tx_date_month, tx_date_year)]->(t) WHERE tx_date_month <= endDate.month AND tx_date _year IS NOT NULL, cache[tx.tx_date_month] , cache[tx.tx_date_year]
59.864 estimated rows
Union@neo4j
endDate, t, anon_0, startDate, tx
119.729 estimated rows
CacheProperties@neo4j
endDate, t, anon_0, startDate, tx
cache[tx.tx_date_year], cache[tx.tx_date_ month]
119.729 estimated rows
Projection@neo4j
startDate, endDate
RuntimeConstant(date($autostring_0 + $ autostring_1)) AS startDate, RuntimeConstant(date($autostring_2 + $ autostring_3)) AS endDate
1 estimated row
Distinct@neo4j
endDate, t, anon_0, startDate, tx
t, anon_0, startDate, tx, endDate
168.040 estimated rows
Apply@neo4j
endDate, t, anon_0, startDate, tx
168.040 estimated rows
Filter@neo4j
endDate, t, anon_0, startDate, tx
(cache[tx.tx_date_month] >= $autoint_5 OR cache[tx.tx_date_year] = endDate.year) AND (cache[tx.tx_date_year] >= startDate.year OR cache[tx.tx_date_year] = startDate. year) AND (cache[tx.tx_date_year] <= endDate.year OR cache[tx.tx_date_year] = endDate.year) AND (cache[tx.tx_date_month] >= $autoint_4 OR cache[tx.tx_date_year] = startDate.year) AND (cache[tx.tx_date_ year] >= startDate.year OR cache[tx.tx_ date_month] >= startDate.month) AND (cache [tx.tx_date_year] <= endDate.year OR cache [tx.tx_date_month] <= endDate.month) AND (cache[tx.tx_date_month] >= $autoint_4 OR cache[tx.tx_date_month] >= startDate.month) AND (t:Terminal AND anon_0:Customer)
1.506 estimated rows
EagerAggregation@neo4j
year, month, t, anon_2
(date({year: cache[tx.tx_date_year], month : cache[tx.tx_date_month], day: $autoint_6 }) + RuntimeConstant(duration({months: $ autoint_7}))).year AS year, (date({year: cache[tx.tx_date_year], month: cache[tx.tx _date_month], day: $autoint_8}) + RuntimeConstant(duration({months: $autoint _9}))).month AS month, t, max(tx.tx_amount) AS anon_2
39 estimated rows
Projection@neo4j
anon_2, t, month, year, tx_amount_fraud_ limit
anon_2 * $autodouble_10 AS tx_amount_fraud _limit
39 estimated rows
NodeIndexScan@neo4j
t, anon_1, month, year, tx_amount_fraud_ limit
RANGE INDEX anon_1:Customer(customer_id) WHERE customer_id IS NOT NULL
7.761 estimated rows
Apply@neo4j
anon_2, t, anon_1, month, year, tx_amount_ fraud_limit
7.761 estimated rows
Expand(Into)@neo4j
anon_2, t, anon_1, tx_current_month, month , year, tx_amount_fraud_limit
(anon_1)-[tx_current_month:Make_ transaction]->(t)
203.787 estimated rows
Filter@neo4j
anon_2, t, anon_1, tx_current_month, month , year, tx_amount_fraud_limit
(tx_current_month.tx_date_month = month AND tx_current_month.tx_date_year = year)
8.491 estimated rows
EagerAggregation@neo4j
day_period, t, year, tx_fraud_count, tx_ count, month
year, month, t, tx_current_month.tx_day_ period AS day_period, count(tx_current_ month) AS tx_count, count(CASE WHEN tx_ current_month.tx_amount > tx_amount_fraud_ limit THEN $autoint_11 ELSE NULLEND) AS tx_fraud_count
92 estimated rows
EagerAggregation@neo4j
day_period, total_transactions, monthly_ avg_fraud_transactions
day_period, sum(tx_count) AS total_ transactions, avg(tx_fraud_count) AS monthly_avg_fraud_transactions
10 estimated rows
ProduceResults@neo4j
day_period, total_transactions, monthly_ avg_fraud_transactions
day_period, total_transactions, monthly_ avg_fraud_transactions
10 estimated rows
Result

6 Performance Analysis

In this section, I will analyze and compare the execution times of all queries presented in the notebook (except for `query_e2`), on databases generated according to the project requirements. The generated databases have the following characteristics:

- 50MB, containing 1,500 nodes and slightly over 900,000 relationships
- 100MB, containing 1,800 nodes and slightly over 1.8 million relationships
- 200MB, containing 3,000 nodes and slightly under 3.5 million relationships

Here's how I chose the parameters for the queries in the workload

- Queries A and B: Since these queries require analyzing data from past relationships, I ran them against one of the last months in which relationships were recorded, being careful not to execute them on the most recent month, ensuring that all transactions for that month had already been generated.
- Query E: I used the same previous point date for `endMonthYear`, while for `startMonthYear` I chose a date three months earlier, creating a four-month interval since the limits are inclusive.
- Query C: I used a value of `k = 15` to demonstrate the excellent execution times achieved even with higher `k` values (compared to `k = 3`). As for the customer ID, I ran several tests to find one that would return results for the query. Without valid results, the query would have stopped before analyzing the k-th co-customer and the execution time would not have been meaningful.

The query execution times reported below are taken from the file `documentation/outputs.txt`, which contains a detailed report of the execution of all queries, specifying the exact parameters used to call them, the execution time, and a partial output printout. The content of this file was produced by aggregating the various outputs from the Python scripts located in the `Neo4j` directory: `Import`, `Workload_queries`, and `Workload_DBextension`. These scripts are executable versions of all the code in this notebook. The configuration parameters used in these Python scripts are adjusted to point to a local Neo4j instance, as well as to local references for the CSV files.

```
data = {
    "Query": [
        "create_terminals_schema", "create_customers_schema", "create_transaction_schema",
        "load_terminals_from_csv", "load_customers_with_available_terminals_from_csv",
        "load_transactions_from_csv",
        "create_transaction_date_index",
        "query_a1", "query_a2", "query_b1", "query_b2", "query_c", "query_di",
        "create_transaction_extended_schema", "query_dii", "query_e1"
    ],
    "50MB": [
        0.02, 0.03, 0.06, 0.03, 0.10, 21.24, 0.00, 0.38, 0.31, 0.36, 0.28, 0.12, 1.89, 0.73, 29.51, 1.02
    ],
    "100MB": [
        0.02, 0.03, 0.03, 0.02, 0.09, 41.80, 0.00, 0.58, 0.40, 0.56, 0.39, 0.23, 3.34, 1.56, 64.91, 5.04
    ],
    "200MB": [
        0.02, 0.03, 0.04, 0.03, 0.18, 71.11, 0.00, 1.13, 0.97, 1.18, 0.76, 0.59, 6.53, 2.97, 172.95, 11.24
    ],
}

df = pd.DataFrame(data)
df.set_index("Query", inplace=True)
df
```

Query	50MB	100MB	200MB
create_terminals_schema	0.02	0.02	0.02
create_customers_schema	0.03	0.03	0.03
create_transaction_schema	0.06	0.03	0.04
load_terminals_from_csv	0.03	0.02	0.03
load_customers_with_available_terminals_from_csv	0.10	0.09	0.18
load_transactions_from_csv	21.24	41.80	71.11
create_transaction_date_index	0.00	0.00	0.00
query_a1	0.38	0.58	1.13
query_a2	0.31	0.40	0.97
query_b1	0.36	0.56	1.18
query_b2	0.28	0.39	0.76
query_c	0.12	0.23	0.59
query_di	1.89	3.34	6.53
create_transaction_extended_schema	0.73	1.56	2.97
query_dii	29.51	64.91	172.95
query_e1	1.02	5.04	11.24

Given the type of workload defined in the project guidelines, we can divide the queries into two categories, for which we will analyze performance using different criteria: queries executed with very low frequency, if not only once, and queries executed with high frequency.

6.1 Queries executed with low frequency

In this category, we also accept queries with longer execution times, as long as the duration is justified by the large volume of data being processed and not by inefficiency due to poor query design. This is because these queries are executed with low frequency and handle massive data imports or modifications that do not require real-time responses from the user. To be more precise, as described in the project guidelines, the queries in this category are executed only once.

- `create_terminals_schema`, `create_customers_schema`, `create_transaction_schema`: These queries perform consistently across all three databases, with an excellent execution time. The database size has no impact since these queries define constraints on an empty database, eliminating the need to verify existing data.
- `load_terminals_from_csv`, `load_customers_with_available_terminals_from_csv`: Both queries perform consistently across all three databases due to the relatively small nodes cardinality ~103.
- `load_transactions_from_csv`: This query is more demanding because it loads relationships, which have a cardinality of ~106, and its execution time scales with the size of the database. The performance of this query is excellent, as it follows the documented Neo4j best practices for handling massive datasets. However, I found two alternative methods for importing CSV data that could potentially offer better performance, but I did not use them because:
  - The first method involves `APOC.load.csv`, which, as documented in the APOC extended documentation, shows how to pair it with `APOC.periodic.iterate` for importing massive CSVs using parallel batches. Unfortunately, the `APOC.load.csv` function is not included in the standard APOC package, and since I wanted to keep things as simple and reproducible as possible, I chose not to use it.
  - The second method involves using the Neo4j-admin import tool, but I did not use it because the project guidelines required creating a query for data import.
- `create_transaction_date_index`: This query completes almost instantly across all databases.
- `create_transaction_extended_schema`: This query has excellent execution performance, despite the cardinality of transactions ~106. The slight increase in execution time compared to previous schema creation queries is due to the presence of preloaded data requiring validation against the newly introduced constraints. Despite this, the query remains highly efficient and well-optimized for the dataset's scale, especially since it is executed only once.
- `query_di`: This query demonstrates excellent execution performance, even with the high cardinality of transactions ~106. In fact, its execution time represents only a small

fraction of the initial time required to load the transactions into the database. Since it only needs to be executed once, as specified in the project guidelines, the execution time is not a significant concern. I don't believe there is much room for improving its performance, as the query is already optimized to perform only the strictly necessary operations, leveraging parallel batch jobs with the `APOC.periodic.iterate` approach.

- `query_dii`: This query is more time consuming because identifying the `buying_friends` is very expensive. However, the execution times are not excessive compared to the amount of data in the DB, and considering that this query only needs to be executed once, the given times are not a problem. In future development, this is one of the queries I would optimize by finding a way to streamline the search for buying_friends, possibly looking for an `APOC` function that could significantly speed up the process.

6.2 Frequently Called Queries

In this category, we prioritize queries with low execution times, ideally under 1 or 2 seconds, due to their frequent execution as part of the regular workload. This is crucial because these queries directly impact the application's response time, thereby affecting the user experience. The only exception where a query in this category may exceed this threshold, while still remaining within a reasonable execution time, is for asynchronous reporting tasks. In such cases, an immediate response to the user is not required, but results should still be delivered within a short timeframe.

- `query_a1`, `query_a2`, `query_b1`, `query_b2`: These queries consistently deliver excellent performance across all database sizes. By utilizing the indexed versions (`a2`, `b2`), the execution time is reduced, ensuring response times under one second for all three database sizes.
- `query_c`: This query demonstrates exceptional performance by leveraging `APOC`. For example, when calculating the 15th-degree co-customer of the customer with `customer_id = 2` (CC15(2)) on the 200MB database, the query returns results in approximately half a second. This highlights that even complex graph traversals can be executed efficiently with the appropriate use of `APOC`, delivering outstanding performance even on large datasets.
- `query_e1`: Query E is computationally intensive query, as it requires constructing a history that potentially spans all the data in the database. The reported execution times are based on a 4-month history. While the query achieves excellent performance related to the volume of data it processes, some user wait time is still unavoidable. To enhance user experience, it would be advisable to implement this functionality asynchronously at the application level. For example, the history could be computed in the background, and the results delivered to the user's inbox via email, particularly for requests involving the complete history of all data in the database.

7 Conclusions

I am highly satisfied with the solution provided for the entire project, as the recorded execution times have proven to be excellent. These results highlight the solution's ability to scale effectively, even when applied to databases with significantly larger datasets. I can state this because all queries perform their tasks optimally, and those that take more than 10 seconds to execute do so solely due to the large volume of data being processed relative to hardware limitations, not because of poor query design.

The only query I would label as potentially problematic is `query_dii`. As previously mentioned, it leaves room for future improvements in optimizing the search for `buying_friends`. However, despite its relatively high execution time, this is a negligible concern, as the query is meant to be executed only once according to the workload described in the project guidelines.