New generation datamodels and DBMSS Project

December 27, 2024

2023 / april 2025 edition

This notebook has been developed in accordance with the project guidelines provided by the professor. You can consult the guidelines at the following link: Project Guidelines.

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 to 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 linked page.

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

```
import os
import sys
import numpy as np
import pandas as pd
import warnings
from IPython.display import SVG, 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, so each database configuration, 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: 0.01s
Time to generate terminal profiles table: 0.00s
Time to associate terminals to customers: 0.13s
Time to generate transactions: 1.54s
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: C:\Users\luca.maccarini\Desktop\luca\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: 0.07s
Time to generate transactions: 16.42s
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: C:\Users\luca.maccarini\Desktop\luca\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

```
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

```
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

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

```
                    TX_DATETIME  CUSTOMER_ID  TERMINAL_ID  TX_AMOUNT  TX_TIME_SECONDS  TX_TIME_DAYS  TX_FRAUD  TX_FRAUD_SCENARIO
TRANSACTION_ID
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. Instead, all of the previously identified parameters must be specified. 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 paid Neo4j instance, I decided to use a 14MB database that we had previously generated with a free Neo4j Aura instance that I had created. Obviously since the free version goes offline after a period of inactivity you can substitute in the code I have prepared in section 4 by entering link and credentials of your free instance.

Despite the performance limitation in the last section, the queries run in this notebook will also be applied to 50MB, 100MB, and 200MB databases, but on a local instance that doesn't have any limitations.

Since creating these databases is time-consuming, I will not run the database creation script during this demonstration. However, the script can be used to generate them if desired, below are the parameters to generate the desired databases:

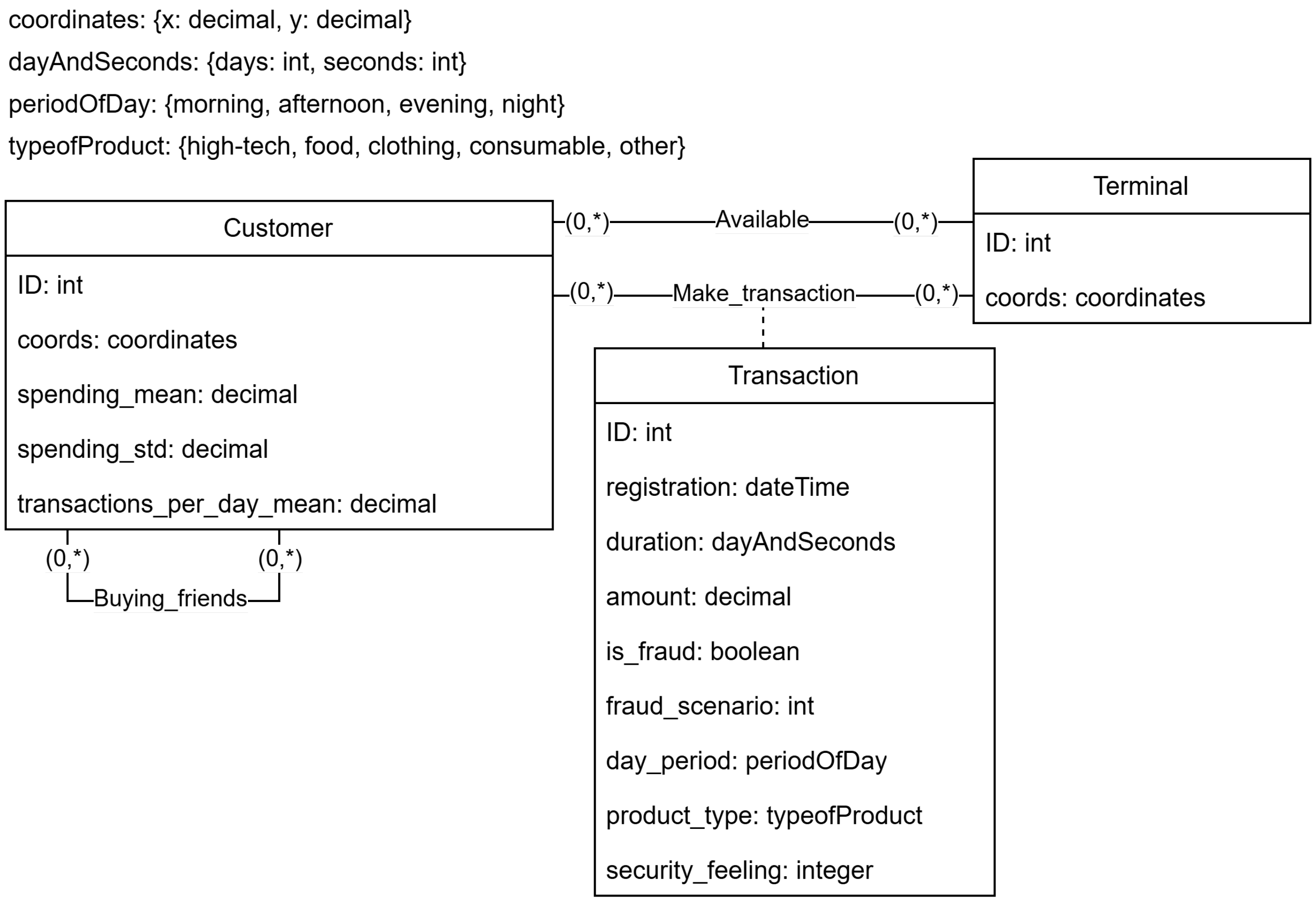
```
DBs = [
    {
        "DB_name": "50MB",
        "n_customers": 1000,
        "n_terminals": 500,
        "n_days": 500,
        "start_date": '2022-01-01',
        "radius": 5
    },
    {
        "DB_name": "100MB",
        "n_customers": 1200,
        "n_terminals": 600,
```

```
        "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. This analysis allowed me to understand the semantics of the data and to design a clear and simple structure that illustrates the relationships between the data to be stored in the database.

2.1 UML Class Diagram



2.2 Costraints

2.2.1 Terminal

- $0 <= \texttt{coords.x} <= 100$
- $0 <= \texttt{coords.y} <= 100$

2.2.2 Customer

- $0 <= \texttt{coords.x} <= 100$
- $0 <= \texttt{coords.y} <= 100$
- $\texttt{spending\_mean} >= 0$
- $\texttt{spending\_std} >= 0$
- $\texttt{transactions\_per\_day\_mean} >= 0$

2.2.3 Transactions

- `amount` > 0
- 0 <= `fraud_scenario` <= 3
- 0 <= `security_feeling` <= 5

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 about how to represent the data to meet the workload requirements.

3.1 Database

I chose Neo4j as the database because the nature of the data suggests a graph structure. In fact, all the relationships present are of the N:N type and such relationships are well handled by graph databases.

Furthermore, this choice was confirmed by the workload, in particular by query 3C, which involves continuous traversal of the 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)

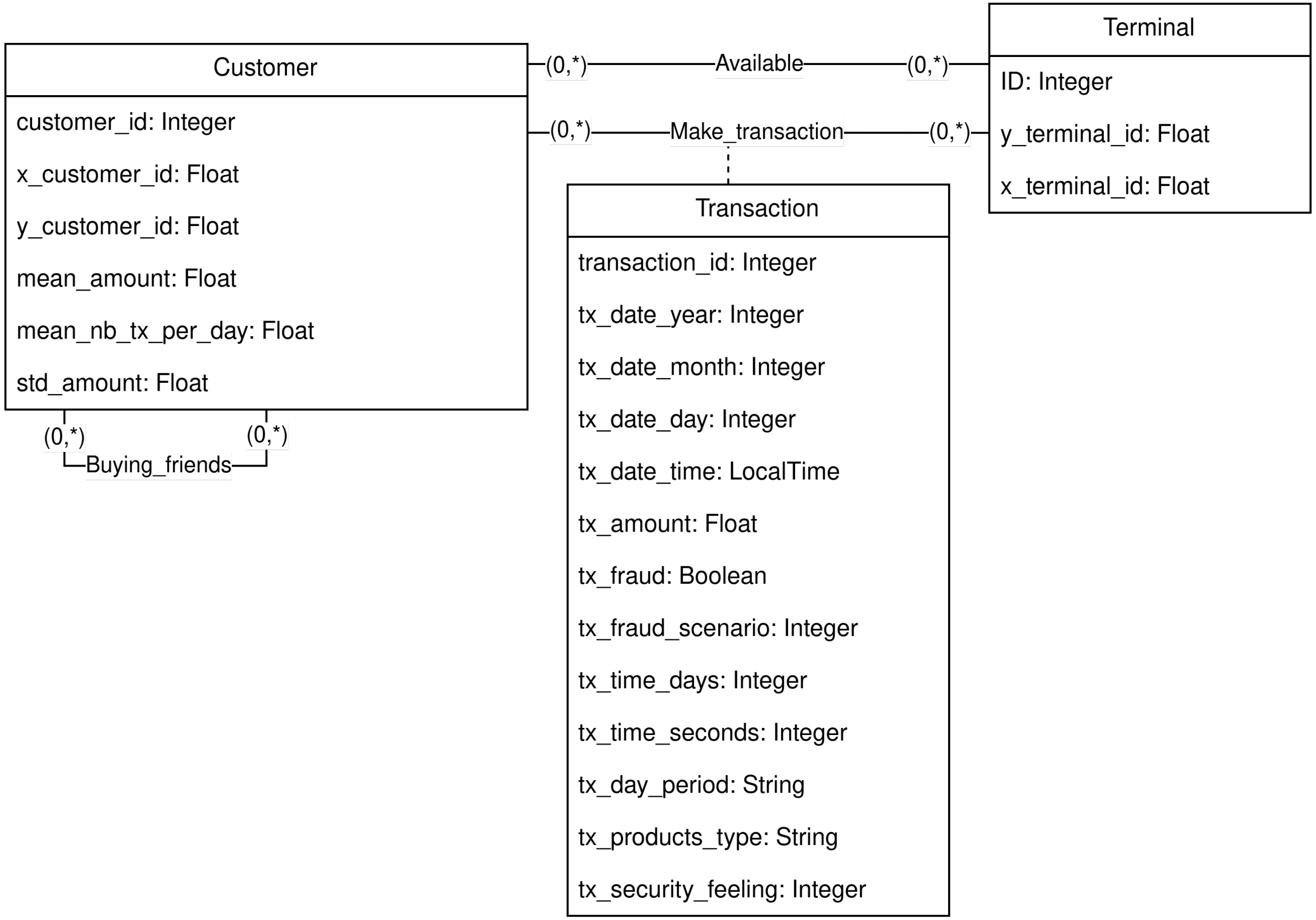
Since Neo4j does not allow the definition of custom types or the insertion of objects within node properties, I decided to eliminate all custom types and implement them using primitive types. For the custom types representing objects, I created a property for each attribute with its corresponding primitive 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 documentation 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 indexing 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 and a composite index was created only on the year and month fields.

The data types specified are those that exist in Neo4j.



3.3 Costraints

3.3.1 Terminal

- $0 <=$ `x_terminal_id` $<= 100$
- $0 <=$ `y_terminal_id` $<= 100$

3.3.2 Customer

- $0 <=$ `x_customer_id` $<= 100$
- $0 <=$ `y_customer_id` $<= 100$
- `mean_amount` $>= 0$
- `std_amount` $>= 0$
- `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 one of the following strings ["morning", "afternoon", "evening", "night"]
- `tx_products_type` is one of the following strings ["high-tech", "food", "clothing", "consumable", "other"]

3.3.4 Assumptions

Since the constraints that can be implemented in Neo4j focus only on the structure and data type, and do not allow constraints on the actual values or the direction of relationships, 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 produced by the *Transaction Data Simulator* tool are correct and satisfy the constraints.

Since Neo4j constraints also do not allow us to define the direction of relationships, it is our responsibility to ensure that we do not make mistakes in the queries we use to create relationships, and 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 be passed queries with parameters embedded directly through string concatenation. While this approach allows for simpler coding, it exposes potential vulnerabilities related to direct parameter concatenation in queries. Since addressing these security concerns is not the goal of this project, but rather demonstrating how the database was managed to optimize workload, I chose to keep the code as simple as possible.

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 generated for the 14MB database. They can be either local file paths or network links. A separate section will explain why network links are preferred in this case. Additionally, in the performance analysis section, we will include the database load times for the 50MB, 100MB, and 200MB databases to provide a comprehensive comparison.

- `lines_per_commit_call` and `lines_per_commit_apoc`: these parameters are used to define the number of operations included in a single batch, where the changes on the DB are committed after each batch. I have defined 2 different parameters because, in order to maximise performance, the batch size depends on how the job is defined. Jobs using Cypher `CALL {} IN TRANSACTIONS OF ... ROWS` will generally allow larger batch sizes than those defined with the 'APOC' library.ROWS

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

```python
#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": 100000,
    "lines_per_commit_apoc": 10000,
    "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:
        close_neo4j_connection(driver)
```

```python
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)
```

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: 5.32s
```

```
True
```

4.1 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 builds upon the previously documented logical model by incorporating the following elements:

- Defining attribute constraints: Each attribute will be associated with its corresponding data type.
- Primary key specification: For each entity in the logical model, the attributes that form the primary key will be explicitly defined.
- Mandatory attribute constraints: Attributes not included in the primary key will be marked as mandatory, ensuring data integrity. (Primary keys are inherently mandatory due to their constraint.)

```python
def create_terminals_schema():
    queries = [
        "CREATE CONSTRAINT terminal_id_is_integer FOR (t:Terminal) REQUIRE t.terminal_id IS :: INTEGER;",
```

```python
        "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: 1.20s
create_customers_schema execution time: 1.28s
create_transaction_schema execution time: 1.78s
```

```
True
```

4.2 Data loading

In order to load data into Neo4j using CSV files, we must first consider where the Neo4j instance is. This is critical because the CSV files must be accessible from the machine running the Neo4j instance. 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, they obviously do not give us access to their servers, so 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 important to use a direct download link for the CSV files to make sure everything works. To share these files easily and quickly, I chose Dropbox because it offers a file sharing option with links that include a query parameter in the URL. This parameter, which appears as `&dl=1` at the end of the link, allows me to specify whether the link should be a direct download. This feature is critical for the Neo4j instance to download the file correctly. I also looked at other cloud storage systems, but the process of getting a direct download link was unnecessarily complex.

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 N 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 functions work similarly, with only the changes they make to the database changing. 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 to 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 that 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 as described above, must match the customer and terminal to create the relationship.

```
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: 2.44s
load_customers_with_available_terminals_from_csv execution time: 2.57s
```

```
[#D1E7]  _: <CONNECTION> error: Failed to read from defunct connection IPv4Address(('45d4bc57.databases.neo4j.io', 7687)) (ResolvedIPv4Address(('35.189.
↪250.174', 7687))): OSError('No data')
```

```
False
```

5 Workload

In this section, I'll 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 requirement;
- 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 the dedicated section, 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. 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 date. When determining the value of `first_of_previous_month`, only the month and year are taken into account, ensuring that the query correctly filters data relevant to the previous month.
- "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 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 counting the `tx_prev_month_all_prev_year` and summing their amounts, it returns `NULL` for both values if no transactions are found in the history. This is useful for distinguishing, in the final result, customers for whom no significant transaction history is found (and therefore no calculations can be performed) from those for whom a history is available (and calculations can be performed as required by the query).

The next `WITH` clause calculates the averages 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 are no transactions, `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)
```

```
Unable to retrieve routing information
Unable to retrieve routing information
Unable to retrieve routing information
Unable to retrieve routing information
Unable to retrieve routing information
Unable to retrieve routing information
```

```
ERROR query_a1: Unable to retrieve routing information
```

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()
```

```
Unable to retrieve routing information
```

```
False
```

```
query_a1(month_and_year_under_analesis)
```

```
query_a1 execution time: 0.62s
```

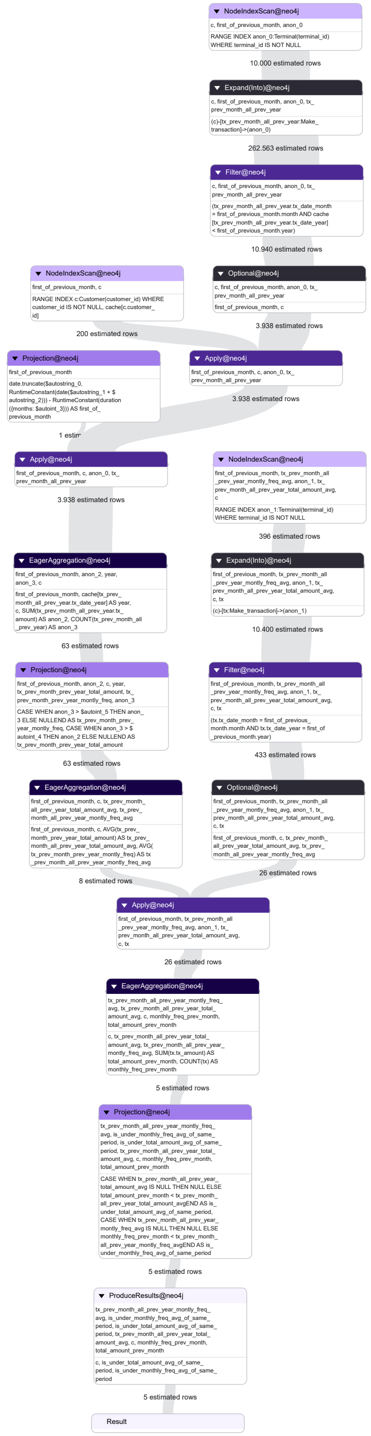
	c	is_under_total_amount_avg_of_same_period	is_under_monthly_freq_avg_of_same_period
0	(mean_amount, x_customer_id, mean_nb_tx_per_da…	None	None
1	(mean_amount, x_customer_id, mean_nb_tx_per_da…	None	None
2	(mean_amount, x_customer_id, mean_nb_tx_per_da…	None	None
3	(mean_amount, x_customer_id, mean_nb_tx_per_da…	None	None
4	(mean_amount, x_customer_id, mean_nb_tx_per_da…	None	None

```
..                                              …                                         …                                         …
195  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    None                                      None
196  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    None                                      None
197  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    None                                      None
198  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    None                                      None
199  (mean_amount, x_customer_id, mean_nb_tx_per_da…                                    None                                      None

[200 rows x 3 columns]
```

As you can see in the execution plan image below, the query does not use the index at all. This is because in the initial `MATCH` clause, we do not directly filter the transactions. Instead, we first match the customers, which prevents the query from using the index efficiently.

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 Neo4j.


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 and only display 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.

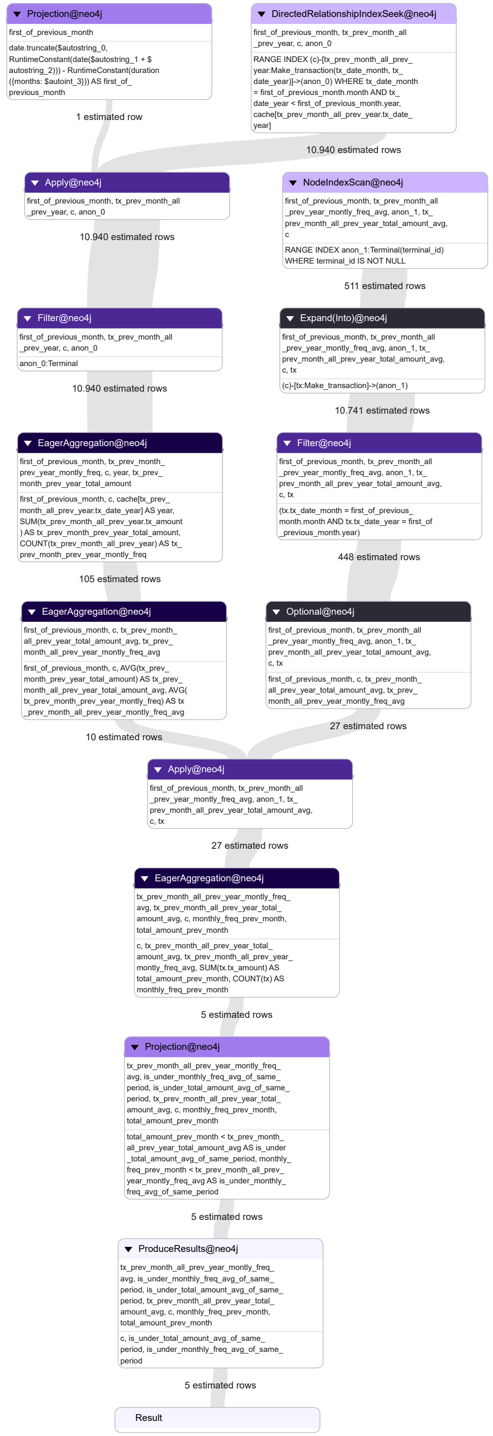
```
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_a2(year_and_month_under_analesis):
    query = f"""
            WITH date.truncate('month', date("{year_and_month_under_analesis}" + "-01") ) - duration({{months: 1}}) AS first_of_previous_month

            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: 1.11s
```

```
Empty DataFrame
Columns: [c, is_under_total_amount_avg_of_same_period, is_under_monthly_freq_avg_of_same_period]
Index: []
```

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.



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 prior to the month specified as a parameter. Similar to the previous query, this query is parameterized by passing a partial date in "yyyy-MM" format to Python. This date is used to calculate the `first_of_previous_month` variable, which represents the first day of the month prior to the given date. In addition, the query includes a reference to the first day of the current month, stored in the `today` variable, for further calculations or filtering as needed.

5.2.2 B1 query code

The query starts by storing the given date in the `today` variable and calculating the first day of the previous month stored in `first_of_previous_month`.

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 remains `NULL`.

The next step uses another `OPTIONAL MATCH` to retrieve transactions for the current month, filtering by the same month and year as `today`. These transactions are stored in the variable `tx_current_month`. Using the calculated fraud amount limit, the query identifies fraudulent transactions by collecting those in `tx_current_month` where the transaction amount exceeds `tx_amount_fraud_limit`. This collection is stored in `fraud_txs_current_month`. If `tx_amount_fraud_limit` is `NULL`, the condition will always evaluate false, resulting in an empty collection for the terminal.

Finally, the `RETURN` statement distinguishes between two problematic cases when a terminal has an empty `fraud_txs_current_month` collection. In the first case, the fraud amount limit could not be calculated, making it impossible to determine whether the terminal had fraudulent transactions. In the second case, the limit was calculated but no fraudulent transactions were identified for that terminal in the current month. To resolve this ambiguity, the query replaces empty collections in `fraud_txs_current_month` with the value `NULL` whenever `tx_amount_fraud_limit IS NULL`. This approach ensures clarity in the results by distinguishing between the two scenarios.

```python
#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: 1.17s

                                              t fraud_txs_current_month
0   (y_terminal_id, terminal_id, x_terminal_id)                    None
1   (y_terminal_id, terminal_id, x_terminal_id)                    None
2   (y_terminal_id, terminal_id, x_terminal_id)                    None
3   (y_terminal_id, terminal_id, x_terminal_id)                    None
4   (y_terminal_id, terminal_id, x_terminal_id)                    None
..                                          …                      …
45  (y_terminal_id, terminal_id, x_terminal_id)                    None
46  (y_terminal_id, terminal_id, x_terminal_id)                    None
47  (y_terminal_id, terminal_id, x_terminal_id)                    None
48  (y_terminal_id, terminal_id, x_terminal_id)                    None
49  (y_terminal_id, terminal_id, x_terminal_id)                    None

[50 rows x 2 columns]
```

5.2.3 B1 Performance

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 reuse the composite index previously created with the Python function `create_transaction_date_index()`.

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.

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]
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.

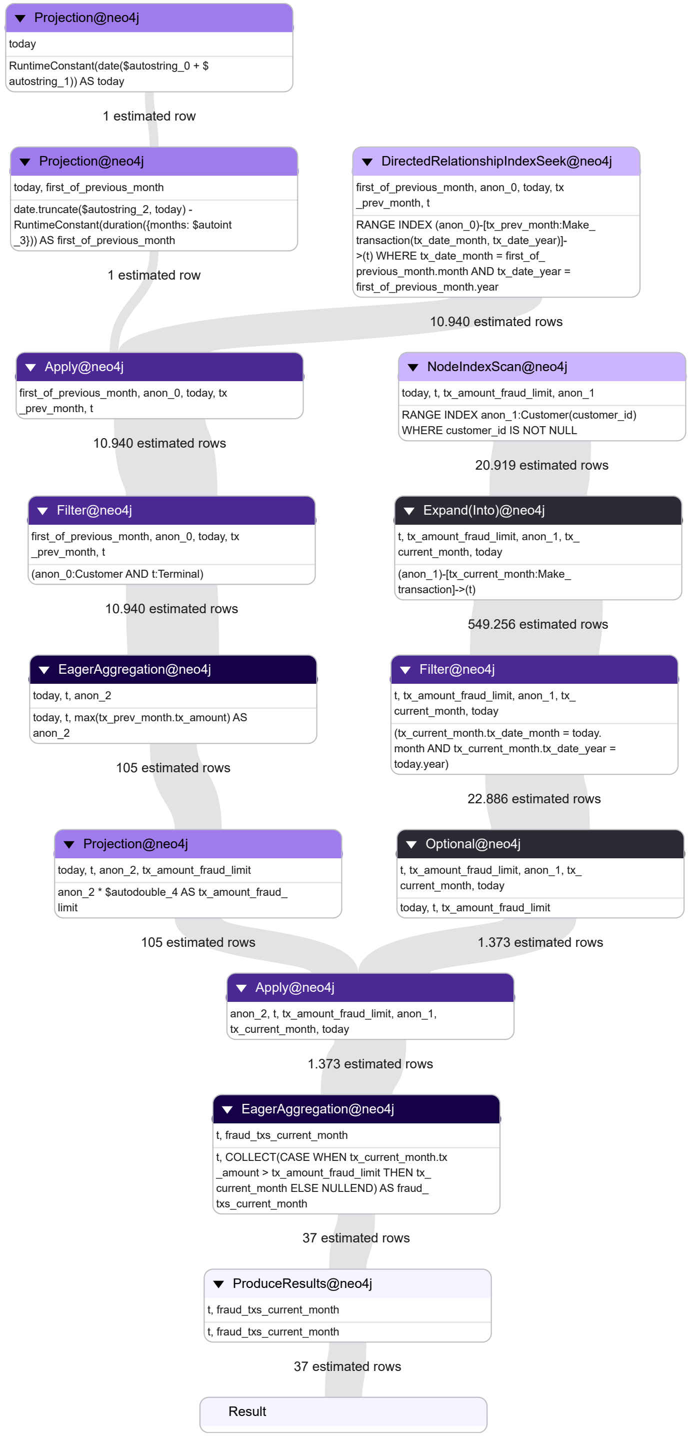
```
#year_and_month_under_analesis is a string that contains a year and a month in the format yyyy-MM
def query_b2(year_and_month_under_analesis):
    query = f"""
            WITH date("{year_and_month_under_analesis}" + "-01") AS today
            WITH today, date.truncate('month', today ) - duration({{months: 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: 1.24s
```

```
Empty DataFrame
Columns: [t, fraud_txs_current_month]
Index: []
```

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.



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 NULL END) 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, allowing us to surpass the co-client of degree `k` in remarkably short processing times.

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's `expandConfig` function to efficiently explore relationships up to a specified level. Starting from the customer node with the same ID as the passed `customer_id`, it navigates through `make_transaction` relationships to `terminal` or other `customer` nodes. The `relationshipFilter` and `labelFilter` parameters allow the query to specify the types of relationships and node labels to be considered. The `maxLevel` parameter limits the exploration depth, ensuring that only paths with length <= `k` are returned. The `uniqueness: 'NODE_GLOBAL'` setting guarantees 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. Finally, 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 reworked 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 lower-level co-customer, the Python `k` becomes `(k - 1) * 2` in the query.

```
#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, 2)
```

```
query_c execution time: 1.06s
```

```
Empty DataFrame
Columns: [CO_Customer]
```

```
Index: []
```

5.3.3 C Performance

I was pleasantly surprised by the performance of this solution, especially considering that the query's requirements represent a potentially exponential task. Before arriving at this query, I tried several approaches with very poor results. Even calculating (CC_3(...)) (the co-customer of degree (k = 3) starting from the customer with `customer_id = ...`) took an enormous amount of time, and attempting `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. This happens because, despite having a large number of `make_transactions` relationships, the customers and terminals have fewer relationships to the transactions. Since the requirement is that customers and terminals must be unique within the path, many paths are filtered out, further reducing the computational load.

With the proposed solution, however, it is possible to go well beyond `k = 3` while still maintaining remarkably low execution times.

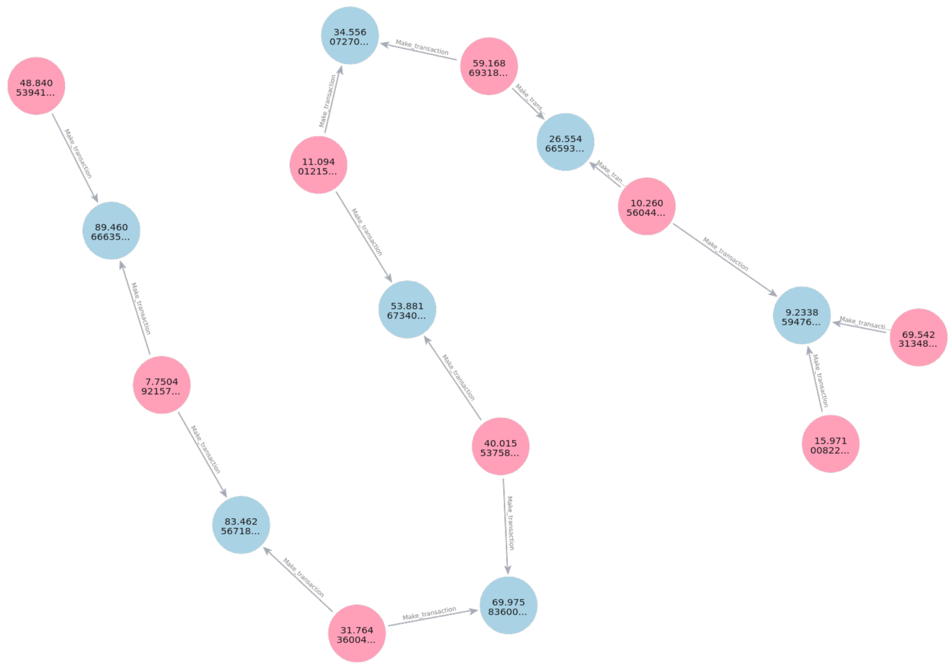
```
query_c(5, 8)
```

```
query_c execution time: 0.92s
```

```
Empty DataFrame
Columns: [CO_Customer]
Index: []
```

To visualise the chains of customers and terminals, I ran the query in the Neo4j console, which returned all the paths starting from `customer_id = 5` and reaching the customers returned by *query_c(5, 8)*.

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.



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 clearly worded 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's `iterate` function, which allows batch tasks to be defined and executed in parallel, similar to the `CALL{}` used earlier in Section 4. 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, and proceeds to do the work.

5.4.2 Di query code

The `query_di` itself has been split into two queries, each with its own Python function: 1. the first query is the core one that uses the `iterate` function to modify the data, it retrieves all the transactions with the `MATCH` function and adds the 3 requested properties, selecting them randomly with the `CASE` function and using `rand()` to calculate the condition;

2. the second query adds the constraints for the new properties to the transactions schema. Unlike the data loading process, the schema creation is done after the data modification. This is because the data already exists and creating the schema for the new data before adding them the costraint creation would not work because the existing data wouldn't satisfy the new constraints.

```
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: 1.25s
create_transaction_extended_schema execution time: 1.32s
```

```
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`).

```python
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: 1.32s
```

```
True
```

5.4.4 Di and Dii Performances

For both queries the performance is excellent and I have not produced optimised versions, the execution plan is not shown below as it is unnecessary as all the work is done in a single block `APOC.iterate` which ensures parallelised batch work giving us efficient queries.

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 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 average **montly** count 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 `startDate` 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, which addresses the previously described problem of fraudulent transactions appearing early in the database records.

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

The next `WITH` aggregates the transactions in `tx` based on the triple (`tx.tx_date_year`, `tx.tx_date_month`, terminal) and calculates the `tx_amount_fraud_limit` for each of these tuples. Note that the grouping does not use the year and month directly, but rather their associated date value, using the first day of the month incremented by one 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 (`tx.tx_date_year`, `tx.tx_date_month`, terminal). Therefore, we can proceed to count the total number of transactions and the fraudulent transactions associated with each daily period and store 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 terminal and we 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 (`tx.tx_date_year`, `tx.tx_date_month`, terminal, `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.

Finally, the `RETURN` clause aggregates the data by day period only, summing the `tx_count` values into `total_transactions` and calculating the 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]
```

```python
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)
```

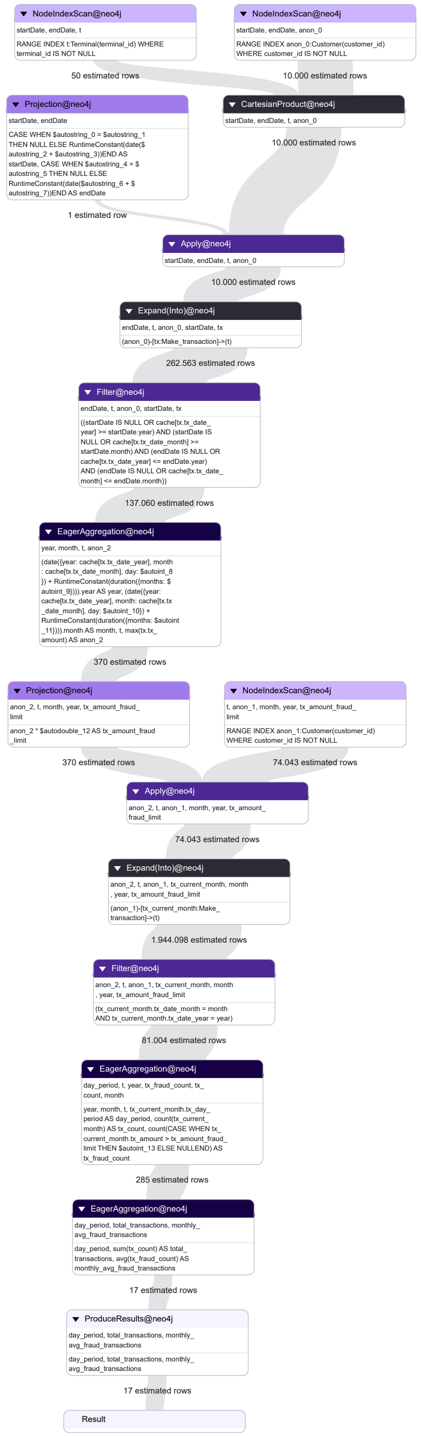
```
query_e1("2023-01" , month_and_year_under_analesis)
```

```
query_e1 execution time: 1.59s

 Empty DataFrame
 Columns: [day_period, total_transactions, monthly_avg_fraud_transactions]
 Index: []
```

5.5.3 E1 Performances

This query is the most computationally intensive of the whole workload, as it potentially operates on all the relationships (if no interva is defined) of all the terminals in the DB, and we are not using an optimised and convenient APOC function. 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.

During the development of this query, I expected it to leverage the same composite index created to optimize query A, given that the filtering of transactions is done by breaking down `startDate` and `endDate` into their year and month components: `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 <= endDate.month`. However, after reviewing the execution plan, as shown below, this is not the case. This is due to the fact that, in the condition, we check if `startDate` and `endDate` are `NULL`, and in those cases, the filter is not applied.


NodeIndexScan@neo4j
startDate, endDate, t
RANGE INDEX t:Terminal(terminal_id) WHERE terminal_id IS NOT NULL
50 estimated rows
NodeIndexScan@neo4j
startDate, endDate, anon_0
RANGE INDEX anon_0:Customer(customer_id) WHERE customer_id IS NOT NULL
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
1 estimated row
CartesianProduct@neo4j
startDate, endDate, t, anon_0
10.000 estimated rows
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
370 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
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` and instead enforcing the definition of an interval, we can take advantage of the composite index we discussed earlier. This would allow the query to efficiently filter transactions based on the `tx.tx_date_year` and `tx.tx_date_month` fields, which are indexed in the composite index, improving performance and making the filtering process more efficient.

```
#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
            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
                 (tx.tx_date_year >= startDate.year OR ( tx.tx_date_year = startDate.year AND tx.tx_date_month >= startDate.month)) AND
                 (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
```

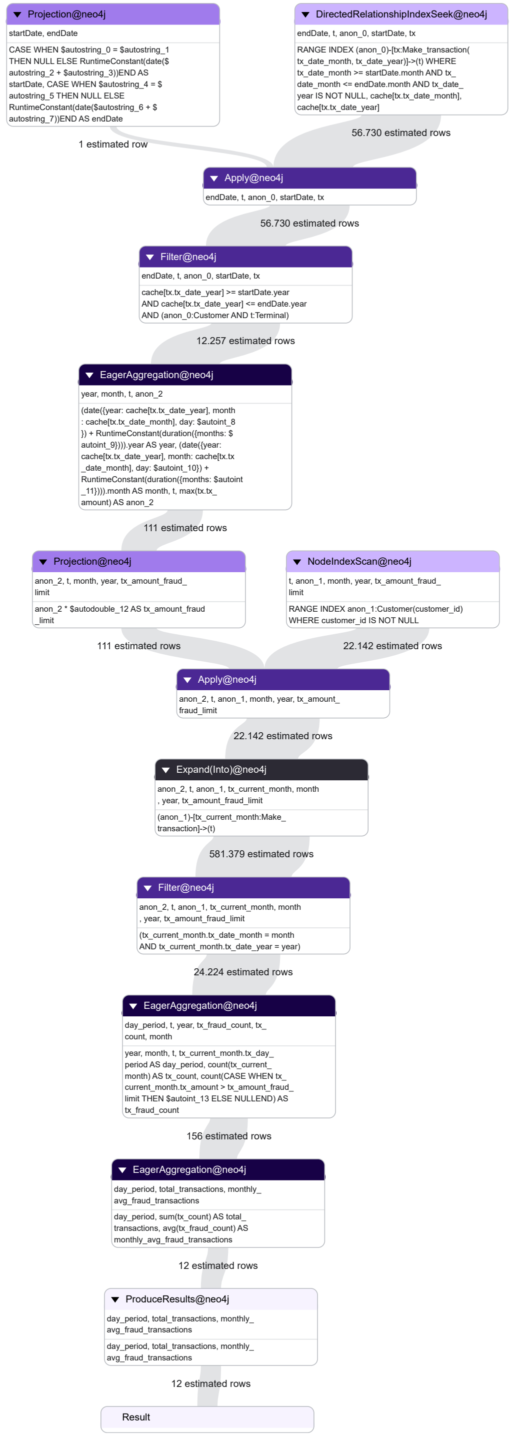
```
                )AS tx_fraud_count

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

    return execute_query_df("query_e2",query)

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

```
query_e2 execution time: 1.66s
```

```
Empty DataFrame
Columns: [day_period, total_transactions, monthly_avg_fraud_transactions]
Index: []
```

5.5.5 E2 performances

From the execution plan shown below, we can see that the composite index is now being used.


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
1 estimated row
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 >= startDate.month AND tx_ date_month <= endDate.month AND tx_date_ year IS NOT NULL, cache[tx.tx_date_month], cache[tx.tx_date_year]
56.730 estimated rows
Apply@neo4j
endDate, t, anon_0, startDate, tx
56.730 estimated rows
Filter@neo4j
endDate, t, anon_0, startDate, tx
cache[tx.tx_date_year] >= startDate.year AND cache[tx.tx_date_year] <= endDate.year AND (anon_0:Customer AND t:Terminal)
12.257 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
111 estimated rows
Projection@neo4j
anon_2, t, month, year, tx_amount_fraud_ limit
anon_2 * $autodouble_12 AS tx_amount_fraud _limit
111 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
22.142 estimated rows
Apply@neo4j
anon_2, t, anon_1, month, year, tx_amount_ fraud_limit
22.142 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)
581.379 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)
24.224 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
156 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
12 estimated rows
ProduceResults@neo4j
day_period, total_transactions, monthly_ avg_fraud_transactions
day_period, total_transactions, monthly_ avg_fraud_transactions
12 estimated rows
Result

6 Performance Analysis and Future Developments

In this section, I will analyze and compare the execution times of all queries presented in the notebook, based on databases generated according to the project requirements. The 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: Because these queries require analyzing data from past relationships, I ran them against the penultimate month in which relationships were recorded, 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 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 execution times reported below are collected in the file `documentation/outputs.txt`. These times were obtained by running Python scripts located in the `Neo4j` directory: `Import`, `Workload_DBextension`, and `Workload_queries`. These scripts are executable versions of all the code in this notebook, with configuration parameters adjusted to point to a local Neo4j instance, as well as local references to the CSV files.

```python
# Dati delle query e dei tempi di esecuzione per le dimensioni del database 50MB, 100MB, 200MB
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", "query_e2"
    ],
    "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, 1.01
    ],
    "100MB": [
        0.02, 0.03, 0.03, 0.02, 0.09, 41.80, 0.00, 0.65, 2.70, 0.60, 0.41, 0.22, 3.34, 1.56, 64.91, 5.04, 5.06
    ],
    "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, 11.37
    ],
}

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	0.65	1.13
query_a2	0.31	2.70	0.97
query_b1	0.36	0.60	1.18
query_b2	0.28	0.41	0.76
query_c	0.12	0.22	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
query_e2	1.01	5.06	11.37

Given the type of workload defined in the project guidelines, we can divide the queries into two categories, for which we will analyze the performance using different criteria:

6.1 Queries executed only once

In this category, we prefer queries with low execution times. However, for queries with higher execution times, we do not consider it a problem as long as the longer duration is justified by the large volume of data being processed. This is because these queries are executed only once and do not require real-time responses from the user.

- `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 exhibit consistent performance due to the relatively small order of magnitude of nodes ~103.
- `load_transactions_from_csv`: This query is inherently more demanding, as it loads relationships with an order of magnitude of ~106 and the execution time scale with database size. I do not think the query time can be improved because the limitation comes from the hardware capacity related to the data volume and not from the query design as I followed the documented Neo4j massive datasets pattern.
- `create_transaction_date_index`: This query completes almost instantly across all databases.
- `create_transaction_extended_schema`: This query demonstrates excellent execution performance, despite the order of magnitude of ~106 transactions. 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 efficiently modifies all the transactions across all database sizes. Although its execution time exceeds one second, it remains a small fraction of the initial time required to load the transactions into the database. Since, according to the project guidelines, it only needs to be executed once, the execution time is not a significant concern. I don't believe there is much room for improving its performance, as the query only carries out the necessary operations and the primary limitation seems to be the hardware capacity in handling the data volume, rather than inefficiencies in the query design.

- `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 prefer queries with low execution times, ideally under 1 or 2 seconds, due to their frequent execution as part of the regular workload. This is because they directly impact the application's response time, and optimizing their performance ensures a smooth user experience.

- `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`: Although this query might initially appear to be the most computationally intensive, due to the complexity of calculating co-customers at a high degree, it performs exceptionally well when leveraging APOC. In the case of calculating the 15th-degree co-customer of the customer with `customer_id = 2` (CC15(2), because it have results) on the 200MB database, the query returns results in about half a second. This demonstrates that even complex graph traversals can be executed rapidly with the proper use of APOC, providing excellent performance even on large datasets.
- `query_e1`, `query_e2`: Query E has proven to be the most computationally intensive query, as it effectively needs to build a history over potentially all data. The reported times are based on building a history for 4 months. Despite the excellent execution times, given the amount of data that needs to be analyzed and computed, some waiting time is required from the user. This suggests implementing the history functionality asynchronously on the application side, possibly by calculating it in the background and sending an email with the requested history to the user's inbox, especially when dealing with a history of all data in the database. A noticeable point when comparing the execution times of the two versions of query E is that the times are almost identical, even though the second version should be an improved, more efficient version. As shown earlier in this notebook, the second version performs better as expected, but I noticed that on the local database, the same query does not use the predefined index, unlike on the free instance on Aura, where the index is used. For future development, I would investigate why the index is not used on the local instance and find a way to ensure its use, which would further reduce execution time.