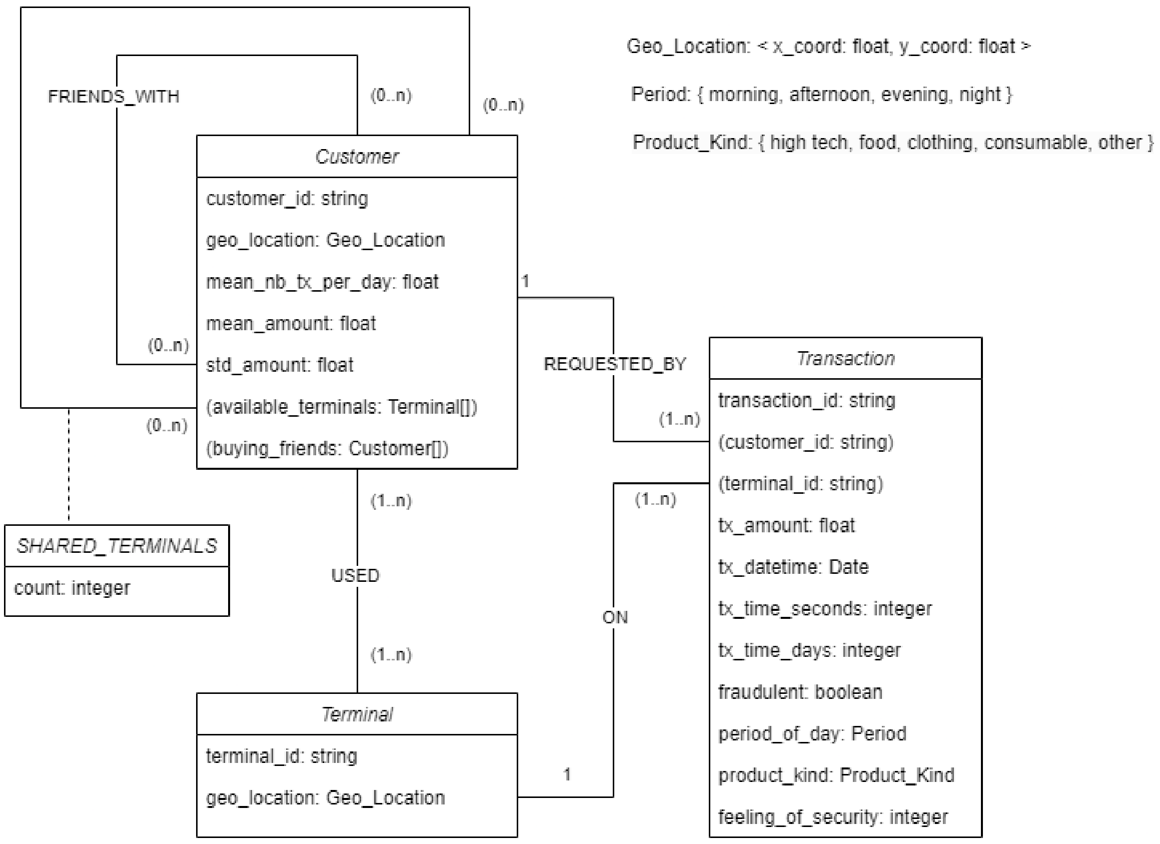
Credit Card Fraud Detection DBMS

27435A Pauletto Jacopo
27436A Nava Stefano

2025

Contents

1 UML Class Diagram



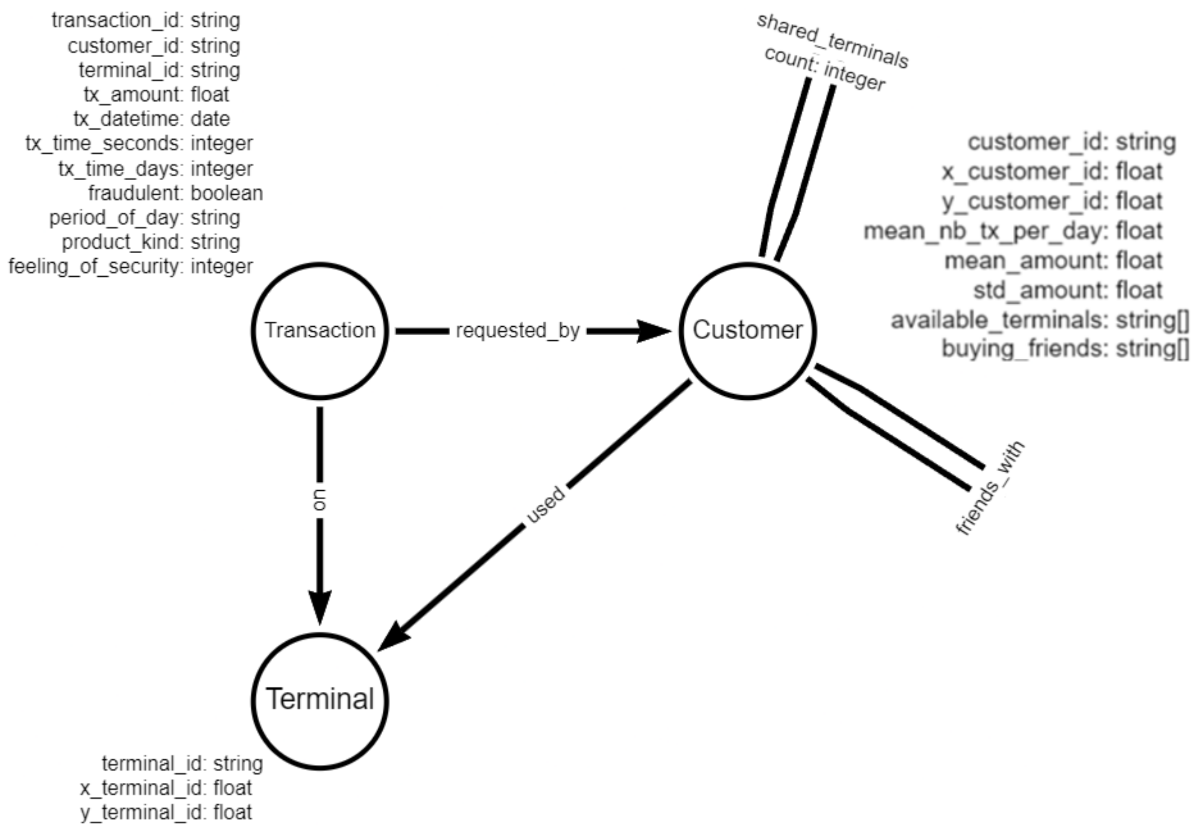
1.1 Assumptions

- We add an ID on Customer in order to univocally identify a customer.
- We add an ID on Terminal in order to univocally identify a terminal.
- We assume that there is a Geo_Location tuple with the x and y coordinates.
- We assume that the period of the day for a Transaction is one of the values contained in the Period enumerator.
- We assume that the product kind for a Transaction is one of the values contained in the Product Kind enumerator.

1.2 Constraints

- The x and y values of the Geo_Location must be between 0 and 100 (the min and max values of the grid).
- The mean_spending_amount and std_spending_amount of a customer can't be greater than the maximum amount_paid for the customer's transactions or lesser than 0.
- The feeling_of_security field of a transaction must be a value between 1 (low security) and 5 (high security).

2 Logical Data Model



We chose to develop this project on a Neo4J database because its mainly focus is on the relationships instead of the entities themselves. We created the `Transaction`, `Terminal` and `Customer` nodes and their respective properties according to the guidelines. The relationships between these three nodes have been generated based on the specifications and we added the two self-relationships on `Customer` following the queries' requests.

- `shared_terminals` identifies the number of terminals shared between two customers;
- `friends_with` identifies the concept of friendship based on the fact that two customers which made more than three transactions from the same terminal expressed a similar average feeling of security.

3 Import scripts

We built a generalized import function which considers the chosen dataset between the three options (50MB, 100MB, 200MB) and creates nodes and relationships based on the contents of the said dataset's CSV files.

```python
with driver.session() as session:
    print("Inizio caricamento nodi")
    for query in nodes_queries:
        session.execute_write(execute_query, query)

    print("Fine caricamento nodi, inizio caricamento relazioni")

    for query in relationships_queries:
        session.execute_write(execute_query, query)

    print("Fine caricamento relazioni")
driver.close()
```

The queries are collected in 2 arrays (one for the nodes, one for the relationships) which are iterated on by the generalized function.

3.1 Queries for importing nodes

We built the importing queries following the specifications of the Fraud Detection Handbook. Each one loads the CSV data (with its headers) from one of the 3 files and creates the related set of nodes.

```
LOAD CSV WITH HEADERS FROM 'file:///{datasets_folders[dataset_number]}/customers.csv' AS row
CREATE (:Customer {{
    customer_id: row.CUSTOMER_ID,
    x_customer_id: toFloat(row.x_customer_id),
    y_customer_id: toFloat(row.y_customer_id),
    mean_amount: toFloat(row.mean_amount),
    std_amount: toFloat(row.std_amount),
    mean_nb_tx_per_day: toFloat(row.mean_nb_tx_per_day),
    available_terminals: row.available_terminals,
    nb_terminals: toInteger(row.nb_terminals)
}});
```

```
LOAD CSV WITH HEADERS FROM 'file:///{datasets_folders[dataset_number]}/terminals.csv' AS row
CREATE (:Terminal {{
    terminal_id: row.TERMINAL_ID,
    x_terminal_id: toFloat(row.x_terminal_id),
    y_terminal_id: toFloat(row.y_terminal_id)
}});
```

```
CALL apoc.periodic.iterate(
    "LOAD CSV WITH HEADERS FROM 'file:///{datasets_folders[dataset_number]}/transactions.csv' AS row RETURN row",
    "CREATE (:Transaction {{
        transaction_id: row.TRANSACTION_ID,
        customer_id: row.CUSTOMER_ID,
        terminal_id: row.TERMINAL_ID,
        tx_datetime: row.TX_DATETIME,
        tx_amount: toFloat(row.TX_AMOUNT),
        tx_time_seconds: toInteger(row.TX_TIME_SECONDS),
        tx_time_days: toInteger(row.TX_TIME_DAYS)
    }})",
    {{ batchSize: 10000, batchMode: "BATCH" }}
);
```

For the transactions we used the APOC module in order to create the nodes in batches to avoid running out of memory with the 100mb and 200mb dataset.

3.2 Queries for importing relationships

Each query creates a relationship between nodes based on the requirements of the project.
As we did for the transactions nodes importing phase, we used the APOC module to perform batch imports in order to avoid running out of memory.

REQUESTED_BY
This relationship identifies the request of a transaction by a customer. The nodes are linked using the `customer_id`.

```
CALL apoc.periodic.iterate(
    "MATCH (c:Customer) MATCH (tr:Transaction) WHERE c.customer_id = tr.customer_id
       RETURN c, tr",
    "MERGE (tr)-[:REQUESTED_BY]->(c)",
    {{ batchSize: 10000, batchMode: "BATCH" }}
);
```

ON
This relationship identifies the terminal on which the transaction is executed. The nodes are linked using the `terminal_id`.

```
CALL apoc.periodic.iterate(
    "MATCH (t:Terminal) MATCH (tr:Transaction) WHERE t.terminal_id = tr.terminal_id
       RETURN t, tr",
    "MERGE (tr)-[:ON]->(t)",
    {{ batchSize: 10000, batchMode: "BATCH" }}
);
```

USED
This relationship identifies the fact that a terminal has been used by a customer, based on his transactions exploiting the `REQUESTED_BY` relationship.

```
CALL apoc.periodic.iterate(
    "MATCH (tr:Transaction)-[:REQUESTED_BY]->(c:Customer) MATCH (tr)-[:ON]->(t:
       Terminal) RETURN tr, c, t",
    "MERGE (c)-[:USED]->(t)",
    {{ batchSize: 10000, batchMode: "BATCH" }}
);
```

SHARED_TERMINALS
This relationship identifies the fact that a customer has shared one or more terminals with another customer, based on his previous `USED` relationships between himself and a terminal. The relationship keeps track of the number of shared terminals with the `count` property on the relationship itself. We decided to add this relationship following the 3a request.

```
CALL apoc.periodic.iterate(
    "MATCH (c1:Customer)-[:USED]->(t:Terminal)<-[:USED]-(c2:Customer) WHERE c1 <> c2
       WITH c1, c2, COUNT(t) AS sharedTerminalCount RETURN c1, c2,
       sharedTerminalCount",
    "MERGE (c1)-[rel:SHARED_TERMINALS]-(c2) SET rel.count = sharedTerminalCount",
    {{ batchSize: 10000, batchMode: "BATCH" }}
);
```

4 Operations scripts

We decided to organize the query files based on their complexity and necessary operations. For the queries ”a”, ”b”, ”e” we used a generalized function because they don’t need parameters and can be directly run. The query ”c” requires building some clauses based on the selected number of levels for the co-customer relationship. The query ”d” has to generate some random values in order to correctly populate the db so we treated it as a completely different operation.

4.1 Simple queries

a - Shared terminals

This query exploits the SHARED_TERMINALS relationship that we build during the data import in order to count the number of shared terminals between customers.

```
MATCH (X:Customer)-[st:SHARED_TERMINALS]-(Y:Customer)
WITH X, Y, st.count AS sharedTerminals
WHERE
    sharedTerminals >= 3 AND
    abs(Y.mean_amount - X.mean_amount) <= X.mean_amount * 0.1
RETURN
    X.customer_id AS Customer_X,
    Y.customer_id AS Customer_Y,
    X.mean_amount AS SpendingAmount_X,
    Y.mean_amount AS SpendingAmount_Y;
```

b - Monthly fraudulent transactions

For this query, we start by retrieving the current month and year and the previous month and year. Then we calculate the average amount spent for the previous month and then we compare it with the current month’s amount for each transaction. If the transaction appears to be fraudulent, we set a fraudulent property as true.

```
WITH date() AS current_date
WITH current_date.year AS current_year, current_date.month AS current_month
WITH current_year, current_month, CASE
        WHEN current_month = 1 THEN 12
        ELSE current_month - 1
    END AS previous_month,
    CASE
        WHEN current_month = 1 THEN current_year - 1
        ELSE current_year
    END AS previous_year
MATCH (t_prev:Transaction)-[:ON]->(term:Terminal)
WHERE toInteger(SPLIT(t_prev.tx_datetime, "-")[1]) = previous_month
AND toInteger(SPLIT(t_prev.tx_datetime, "-")[0]) = previous_year
WITH term, AVG(t_prev.tx_amount) AS avg_prev_month_import, current_month,
    current_year
WITH term, avg_prev_month_import, avg_prev_month_import * 1.2 AS threshold,
    current_month, current_year
MATCH (t_curr:Transaction)-[:ON]->(term)
WHERE toInteger(SPLIT(t_curr.tx_datetime, "-")[1]) = current_month
AND toInteger(SPLIT(t_curr.tx_datetime, "-")[0]) = current_year
AND t_curr.tx_amount > threshold
SET t_curr.fraudulent = true
RETURN t_curr
```

e - Period of day transactions

This query counts the transactions and checks for the fraudulent ones exploiting the added period_of_day property.

```
MATCH (t:Transaction)
WITH t.period_of_day AS period, COUNT(t) AS total_transactions,
    AVG(CASE WHEN t.fraudulent = true THEN 1 ELSE 0 END) AS
        avg_fraudulent_transactions
RETURN period, total_transactions, avg_fraudulent_transactions
ORDER BY period
```

4.2 Complex query C - Co-Customers

The code constructs a Cypher query dynamically. The process involves:

- Starting with a MATCH clause to find the initial customer node (u1).
- Iteratively adding MATCH clauses to connect customers over num_levels.
- Adding a WHERE clause to ensure that all connected customers in the chain are unique.
- Appending a RETURN statement to retrieve the ID of the customer at the final level.

```
c_query = f"MATCH (u1:Customer {{customer_id: {customer_id}}})"

for i in range(1, num_levels):
    c_query += f" MATCH (u{i})-[:SHARES_TERMINAL]-(u{i+1}:Customer)"

where_clause = f" WHERE"
for i in range(1, num_levels + 1):
    for j in range(i+1, num_levels + 1):
        where_clause += f" u{i}.customer_id <> u{j}.customer_id AND"
where_clause = where_clause[:-4] + " RETURN DISTINCT u" + str(num_levels) + ".
    customer_id AS co_customer"

c_query += where_clause
print(c_query)
```

Generated Query Example

Given the inputs:

- customer_id = 1
- num_levels = 3

The generated Cypher query would look like:

```
MATCH (u1:Customer {customer_id: 1})
MATCH (u1)-[:SHARES_TERMINAL]-(u2:Customer)
MATCH (u2)-[:SHARES_TERMINAL]-(u3:Customer)
WHERE u1.customer_id <> u2.customer_id AND
      u1.customer_id <> u3.customer_id AND
      u2.customer_id <> u3.customer_id
RETURN DISTINCT u3.customer_id AS co_customer
```

This query retrieves the IDs of all customers (u3) who are three levels away from the initial customer (u1) via shared terminals, ensuring no duplicates in the connections.

4.3 Complex query D - Extensions

The implementation consists of two core functions: `add_fields` and `create_friends`.

Extending Transactions

The `add_fields` function iteratively processes transactions and enriches them with additional attributes. It determines the period of the day based on the transaction's timestamp and assigns random values for `product_kind` and `feeling_of_security` as follows:

```
def add_fields(tx):
    query = f"""
        MATCH (t:Transaction{{transaction_id: $transaction_id}})
        WITH t, toInteger(substring(SPLIT(t.tx_datetime, " ")[1], 0, 2)) as
            intDateTime
        SET t.period_of_day = CASE
            WHEN intDateTime >= 6 AND
                intDateTime < 12 THEN "morning"
            WHEN intDateTime >= 12 AND
                intDateTime < 18 THEN "afternoon"
            WHEN intDateTime >= 18 AND
                intDateTime < 24 THEN "evening"
            ELSE "night"
        END,
        t.product_kind = $product_kind,
        t.feeling_of_security = $feeling_of_security
    """

    for record in tx.run("MATCH (t:Transaction) WHERE t.product_kind IS NULL RETURN t
        .tx_datetime AS tx_datetime, t.transaction_id AS transaction_id LIMIT 10000")
        :
        product_kind = random.choice(["high tech", "food", "clothing", "consumable",
            "other"])
        feeling_of_security = random.randint(1, 5)
        tx.run(
            query,
            product_kind=product_kind,
            feeling_of_security=feeling_of_security,
            transaction_id=record["transaction_id"]
        )
```

Helper Function: `get_period_of_day`

This function determines the period of the day based on the hour extracted from the transaction's timestamp:

```
def get_period_of_day(tx_datetime):
    hour = int(tx_datetime.split(" ")[1].split(":")[0])
    if 6 <= hour < 12:
        return "morning"
    elif 12 <= hour < 18:
        return "afternoon"
    elif 18 <= hour < 24:
        return "evening"
    else:
        return "night"
```

Creating Buying Friends Relationships

The `create_friends` function identifies customers who have executed more than three transactions at the same terminal and calculates their average `feeling_of_security`. If the average feelings of two customers differ by less than 1, a `FRIENDS_WITH` relationship is established between them:

```python
def create_friends(tx):
    query_avg_feeling = """
        MATCH (c:Customer)-[:REQUESTED_BY]-(t:Transaction)-[:ON]->(terminal:Terminal)
        WITH c, terminal, AVG(t.security_feeling) AS avg_feeling, COUNT(t) AS count
        WHERE count > 3
        RETURN c.customer_id AS customer_id, terminal.terminal_id AS terminal_id,
            avg_feeling
    """
    results = tx.run(query_avg_feeling)
    customers = list(results)

    query_create_friends = """
        MATCH (c1:Customer), (c2:Customer)
        WHERE c1.customer_id = $customer1_id AND c2.customer_id = $customer2_id
        MERGE (c1)-[:FRIENDS_WITH {
            terminal_id: $terminal_id,
            avg_feeling_diff: $avg_feeling_diff
        }]->(c2)
    """

    for c1 in customers:
        for c2 in customers:
            if c1["customer_id"] != c2["customer_id"] and c1["terminal_id"] == c2["
                terminal_id"]:
                diff = abs(c1["avg_feeling"] - c2["avg_feeling"])
                if diff < 1:
                    tx.run(query_create_friends,
                           customer1_id=c1["customer_id"],
                           customer2_id=c2["customer_id"],
                           terminal_id=c1["terminal_id"],
                           avg_feeling_diff=diff)
```

5 Performances

5.1 Data import performances

Import type	50mb	100mb	200mb
Time to generate customer profiles table	0.057s	0.067s	0.052s
Time to generate terminal profiles table	0.0059s	0.0059s	0.0059s
Time to associate terminals to customers	1.8s	1.8s	1.8s
Time to generate transactions	58s	100s	190s

We measured these times while importing the different datasets in a Neo4J local instance. Since the number of customers and terminals doesn't vary between the various dataset, their import times are similar. On the other hand, transactions massively increase and their import times are different.

5.2 Queries performances

Query	50mb	100mb	200mb
a	2139 ms	2211 ms	2067 ms
b	2453 ms	2361 ms	2090 ms
c	2125 ms	2156 ms	2104 ms
d	69.42 h	352.8 h	1192.6 h
e	2100 ms	2096 ms	2059 ms

Observations:

- For the query `c` we used a co-customer degree of 3 with `customer_id` set to 0.
- For the query `d` we estimated the total time based on a subset of data (1000 for the 50mb dataset, 100 for the 100mb and 200mb datasets) because of its execution length.

5.3 Possible improvements

We observed we could add some indexes on `customer_id`, `terminal_id` and `transaction_id` regarding the execution of queries `c` and `d`:

- About the query `c`, the caller provides an ID for the customer for which the co-customer relationship is retrieved. If we create an index on the ID, the number of customers scanned in order to find that specific ID is lower than without an index.
- About the query `d`, we have several parameters used in more steps:
  - When adding the `period_of_day`, `product_kind` and `feeling_of_security` properties, we pass a `transaction_id` in order to set the new properties for the right transaction node.
  - When creating friendships, we have a similar situation in which we need to find the right customers ("friends") after verifying that they have a similar feeling of security. Using an index we can avoid scanning more customers than needed.

We also observed we could use an anchor node when creating friends exploiting `customer_id`, as we did in the step about adding the `period_of_day`, `product_kind` and `feeling_of_security` properties for the transactions. This would result in having a better starting point for the query execution, possibly reducing the times.