Fraud Detection Report

New generation data models and DBMSs

Miad Alavinezhad

Prof. Marco Mesiti

November 3, 2024

1 Introduction

This is a report for creation of a NoSQL database for credit card fraud detection. The dataset for this project is generated and UML class diagram and conceptual models are made. Next the dataset is uploaded to the selected NoSQL database and operations defined by the project are executed on them. Then the performance of the query execution is measured.

2 Dataset

Three sets of datasets are produced in this project. The features of each datasets are the same but they differ in their size. First dataset is about 50 MB, second dataset is about 100 MB and the third one is 200 MB. The purpose of these datasets is to measure the performance of the queries and the database on diffrnet size of data. Each dataset contains three main classes:

1. Customer
2. Terminal
3. Transaction

Properties of each class:

1. Customer:
    - customer_id: To identify customer
    - x_customer_id, x_customer_id: Coordinate of the customer
    - std_amount: Standard deviation of the transaction amounts for the customer
    - mean_amount: Mean of the transaction amounts for the customer
    - mean_nb_tx_per_day: The average number of transactions per day for the customer
2. Terminal:

- terminal_id: To identify the terminal
- x_terminal_id, x_customer_id: Coordinate of the terminal

3. Transaction:

   - transaction_id: To identify the transaction
   - customer_id: id of the customer who made the transaction
   - terminal_id: id of the terminal where the transaction has been made
   - transaction_amount: The amount of the transaction

For each class, the corresponding records are saved into CSV files. Below the number of each class created in each dataset size is reported:

size	**customer**	**terminal**	**transaction**
50MB	3,700	5,000	960,332
100MB	6,000	5,000	2,079,080
200MB	10,000	6500	3,470,125

Table 1: Number of records for each size of dataset

3 Database

The database used in this project is **Neo4j**, a NoSQL graph database. Before uploading datasets to the database, a UML class diagram and conceptual model will be addressed.

3.1 Class Diagram and Logical Data Model

Below is the class diagram of the model in hand:

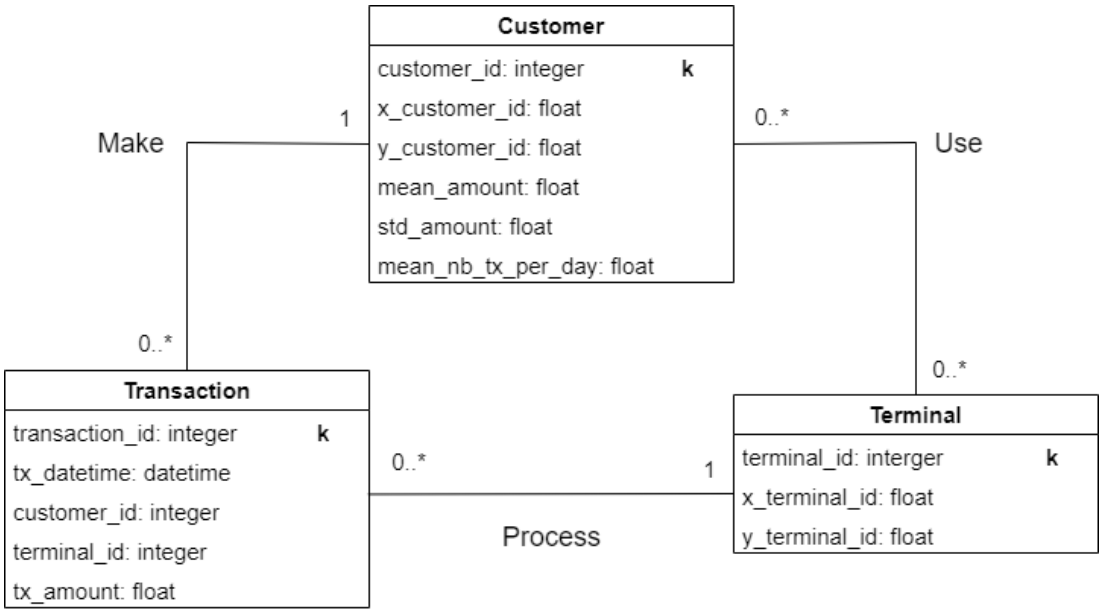

Figure 1: UML Class diagram

Each customer can make multiple transactions by using any terminal in its radius and terminals process the transactions. The assumptions of the UML class:

- The mean_amount is will be drawn from a uniform distribution (5,100) and the std_amount will be set as the mean_amount / 2.
- mean_nb_tx_per_day will be drawn from a uniform distribution (0,4).
- customer and terminal coordinates are in a 100 by 100 grid.

The constraints of the UML class:

- Each customer uses the terminal which is in the defined radius.
- Each transaction can be done on 1 terminal.
- Each transaction is done by 1 customer.
- Customers can make transactions one at the time.

The workloads for this project is as follows:

- **a.** 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.

- **b.** 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.

- **c.** 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).

- **d.** Extend the logical model that you have stored in the NOSQL database by introducing the following information (pay attention that this operation should be done once the NoSQL database has been already loaded with the data extracted from the datasets):

  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.

- **e.** For each period of the day identifies the number of transactions that occurred in that period, and the average number of fraudulent transactions

Now in order to better understand how the queries should navigate through the dataset, the diagram below can be used:

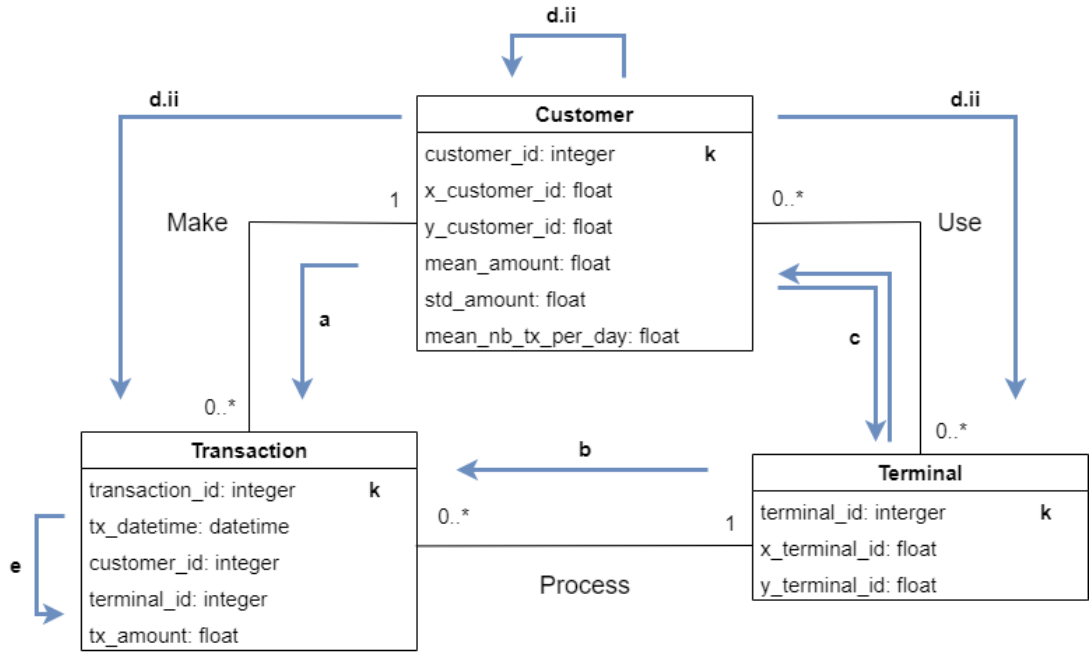

Figure 2: Logical Model of Dataset

In operation **a.** the query should first use the customers id and from there access and check their amount and number of spending in the transaction record.
In operation **b.** query use terminal id to access the transactions done using that specific terminal.
In operation **c.** starting from each customer, the query accesses the terminal used and checks other customers that used that terminal. This process repeats for 3 time until the chain with the

length 3 is complete. The query should check customers and terminals in order.
In Operation **d.ii** from each customer the used terminals are checked and then the number of transactions that the customer has made, after that customers having the said condition in the workload are connected with a new relation.
In operation **e.** the time of each transaction is check and count and the number of fraudulent transactions are averaged. So the query just stays in the transaction nodes.

Since Neo4j database is being used, another diagram forms as the data is uploaded to the database. Nodes and the relations between them are set as the diagram shown below:

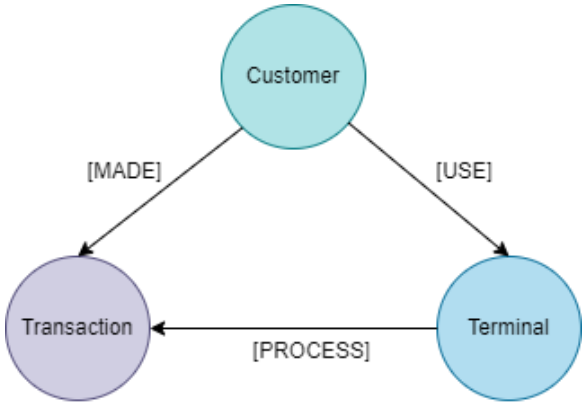

Figure 3: Neo4j Graph Model

4 Loading Dataset in to Database

Datasets are uploaded to the database using Python driver. Queries for creating nodes and relationships between them is written. Then the driver is connect to the database and runs the queries. First the queries to create the nodes and the relations:

```
load_customer = """
LOAD CSV WITH HEADERS
    FROM 'https://media.githubusercontent.com/media/
    miadalavinezhad/Fraud_detection_neo4j/main/data_sets_50
    /customer.csv' AS row

CALL {
WITH row
CREATE (c:Customer {customer_id: row.CUSTOMER_ID,
                    x_customer_id: row.x_customer_id,
                    y_customer_id:  row.y_customer_id,
                    mean_amount: row.mean_amount,
                    std_amount: row.std_amount,
                    mean_nb_tx_per_day:
                     ↪  row.mean_nb_tx_per_day,
                    available_terminals:
                     ↪  row.available_terminals,
                    nb_terminals: row.nb_terminals})
} IN TRANSACTIONS OF 200 ROWS

"""
```

Listing 1: Loading and creating customer nodes

The generated CSV files are uploaded to a GitHub repository and they can be accessed from there. Customer CSV is loaded from the repository with "LOAD CSV". Then the nodes are created using "CREATE" and its properties are set.

```
load_terminal = """
    LOAD CSV WITH HEADERS
        FROM 'https://media.githubusercontent.com/media
        /miadalavinezhad/Fraud_detection_neo4j/main
        /data_sets_50/terminal.csv' AS row

    CALL {
    WITH row
    CREATE (t:Terminal {terminal_id: row.TERMINAL_ID,
                        x_terminal_id:
                         ↪  row.x_terminal_id,
                        y_terminal_id:
                         ↪  row.y_terminal_id})
    } IN TRANSACTIONS OF 200 ROWS
"""
```

Listing 2: Loading and creating customer nodes

Same as customer, terminal data is loaded and its nodes is created.

```
load_transaction = """
    LOAD CSV WITH HEADERS
    FROM'https://media.githubusercontent.com/media
    /miadalavinezhad/Fraud_detection_neo4j/main
    /data_sets_50/transaction.csv' AS row

    CALL {
    WITH row
    MATCH (c:Customer {customer_id: row.CUSTOMER_ID})
    MATCH (t:Terminal {terminal_id: row.TERMINAL_ID})
    CREATE (c)-[:MADE]->(tx:Transaction
        {transaction_id:row.TRANSACTION_ID,
        transaction_datetime:row.TX_DATETIME,
        transaction_amount:row.TX_AMOUNT})
    CREATE (t)-[:PROCESS]->(tx)
    CREATE (c)-[:USE]->(t)
    } IN TRANSACTIONS OF 500 ROWS
"""
```

Listing 3: Creating relations and transaction nodes

For transactions, creating its the nodes and relationships is done simultaneously. Transaction nodes and its properties are created while creating the three relations for this model.

For uploading the data, the "CALL IN TRANSACTIONS OF n ROWS" is used. By using it, multiple rows are processed within a transaction and can save a lot of time by not communicating with the database after creating each node or relationship. In this case for customer and terminal, each 200 rows are put into one transaction and for transaction and relationships, each 500 rows are put together.

```
    customer_constraint = """
        CREATE CONSTRAINT customer IF NOT EXISTS
        FOR (c:Customer) REQUIRE c.customer_id IS UNIQUE
    """
    terminal_constraint = """
        CREATE CONSTRAINT terminal IF NOT EXISTS
        FOR (t:Terminal) REQUIRE t.terminal_id IS UNIQUE
    """
    transaction_constraint = """
        CREATE CONSTRAINT transaction IF NOT EXISTS
        FOR (tx:Transaction) REQUIRE tx.transaction_id IS
        ↪  UNIQUE
    """
```

Listing 4: Setting the constrains

These constrains are made, so if a node is already exist, database won't be allowed to create the nodes again.

```
URI = "bolt://localhost:7687/"
AUTH = ("neo4j", "12345678")

# Session
with GraphDatabase.driver(URI, auth=AUTH) as driver:
    driver.verify_connectivity()

    with driver.session() as session:
        session.run(load_customer).data()
        session.run(customer_constraint)

        session.run(load_terminal).data()
        session.run(terminal_constraint)

        session.run(load_transaction).data()
        session.run(transaction_constraint)
```

Listing 5: Set the driver, connect to database and execute queries

By making a session and authorizing the database, the driver is set and can run the queries for creating and setting the appropriate constraints over them.

5 Implementing the Operations

For better performance on queries and searches in implementing the operations, indexes are created on the node in the database.

```
CREATE INDEX index_customer IF NOT EXISTS
FOR (c:Customer) ON (c.customer_id)
CREATE INDEX index_terminal IF NOT EXISTS
FOR (t:Terminal) ON (t.terminal_id)
CREATE INDEX index_transaction IF NOT EXISTS
FOR (tr:Transaction) ON (tr.transaction_id)
```

Listing 6: Create index on property of node

5.1 Operation a

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.

```
MATCH (c:Customer)-[:MADE]->(tr:Transaction)
WITH c, datetime(tr.transaction_datetime) AS
↪  transactionDate,
tr.transaction_amount AS amount
ORDER BY transactionDate DESC
WITH c, COLLECT(transactionDate)[0] AS lastTransactionDate
WITH c, lastTransactionDate.month AS lastMonth

MATCH (c)-[:MADE]->(tr:Transaction)
WHERE datetime(tr.transaction_datetime).month = lastMonth
WITH c, lastMonth, SUM(toInteger(tr.transaction_amount)) AS
lastMonthSpending, COUNT(tr) as lastMonthFrequency

MATCH (c)-[:MADE]->(tr:Transaction)
WHERE datetime(tr.transaction_datetime).month < lastMonth
WITH c, lastMonthSpending, lastMonthFrequency,
(SUM(toInteger(tr.transaction_amount)) / 5) AS
↪  usualSpending,
(COUNT(tr) / 5) AS usualFrequency

WHERE lastMonthSpending < usualSpending AND
lastMonthFrequency < usualFrequency
RETURN c.customer_id AS customer_id, lastMonthSpending,
↪  usualSpending, lastMonthFrequency, usualFrequency
```

Listing 7: Operation a query

First last month of the transactions is found. Then the transaction amount and frequency of the last month is calculated. Next the average(usual) spending and frequency of previous months are calculated and at last customers which their last month spending and frequency is less than usual are returned.

5.2 Operation b

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.

```
MATCH (t:Terminal)-[:PROCESS]->(tr:Transaction)
WITH t, datetime(tr.transaction_datetime) AS
 ↪  transactionDate,
tr.transaction_amount AS amount
ORDER BY transactionDate DESC
WITH t, COLLECT(transactionDate)[0] AS lastTransactionDate
WITH t, lastTransactionDate.month AS lastMonth

MATCH (t)-[:PROCESS]->(tr:Transaction)
WHERE datetime(tr.transaction_datetime).month = lastMonth
WITH t, MAX(tr.transaction_amount) AS maxImport, lastMonth

MATCH (t)-[:PROCESS]->(tr:Transaction)
WHERE datetime(tr.transaction_datetime).month = lastMonth
AND toInteger(tr.transaction_amount) > toInteger(maxImport)
 ↪  * 1.2
RETURN t.terminal_id AS terminalId,
       tr.transaction_id AS transactionId,
       tr.transaction_amount AS transactionAmount ,
       maxImport, 1.2 * toInteger(maxImport) AS threshold;
```

Listing 8: Operation b query

Query first find last month and the maximum amount imported(amount) for each terminal. Then it compares the transactions of each terminal to its last month maximal and if it was %20 higher, it is fraudulent and returns it.

5.3 Operation c

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 trans-

action 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).

```
MATCH (u1:Customer
↪  {customer_id:"0"})-[:USE*4]-(u2:Customer)
WHERE u1 <> u2
RETURN distinct u1.customer_id, u2.customer_id
```

Listing 9: Operation c query

Co-customer relationship with degree $k = 3$ means 2 customers must be connected to each other with 2 terminals($k-1$) in between. The path will have 4 [:USE] relation in between. Since the query wants co-customer for a given user, user with customer_id: "0" is set for $u1$ as example.

5.4 Operation d

5.4.1 i

Each transaction should be extended with:
1. The period of the day morning, afternoon, evening, night in which the transaction has been executed.

```
def add_time_period(session):
    query = """
    MATCH (tr:Transaction)
    CALL {
    WITH tr
    WITH
    CASE
        WHEN datetime(tr.transaction_datetime).hour > 5 AND
        datetime(tr.transaction_datetime).hour <= 12 THEN
        ↪  'Morning'
        WHEN datetime(tr.transaction_datetime).hour > 12 AND
        datetime(tr.transaction_datetime).hour <= 17 THEN
        ↪  'Afternoon'
        WHEN datetime(tr.transaction_datetime).hour > 17 AND
        datetime(tr.transaction_datetime).hour <= 21 THEN
        ↪  'Evening'
        ELSE 'Night'
    END AS day_period, tr

    SET tr.transaction_period = day_period
    } IN TRANSACTIONS OF 500 ROWS
    """
    session.run(query)
```

Listing 10: Operation d.i.1 query

Categorizing time of transactions using "CASE" and "CALL" for better performance.

2. The kind of products that have been bought through the transaction high-tech, food, clothing, consumable, other

```
def add_product_category(session):
    query = """
    MATCH (tr:Transaction)
    CALL {
    WITH tr
    WITH
    CASE
        WHEN tr.transaction_amount >= 0 AND
        tr.transaction_amount < 25 THEN 'Food'
        WHEN tr.transaction_amount >= 25 AND
        tr.transaction_amount < 50 THEN 'Clothing'
        WHEN tr.transaction_amount >= 50 AND
        tr.transaction_amount < 55 THEN 'Consumable'
        ELSE 'High-Tech'
    END AS category, tr

    SET tr.transaction_category = category
    } IN TRANSACTIONS OF 500 ROWS
    """

    session.run(query)
```

Listing 11: Operation d.i.2 query

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.

```
def add_security(session):
    query = """
        MATCH (tr:Transaction)
        CALL {
        WITH tr
        SET tr.transaction_security = FLOOR(rand() * 5) + 1
        } IN TRANSACTIONS OF 500 ROWS
    """

    session.run(query)
```

Listing 12: Operation d.i.3 query

Generating a random number using random() and put it in the range of 1 to 5.

5.4.2 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.

```
def buying_friends(session):
    query = """
CALL apoc.periodic.iterate(
"MATCH
 ↪  (c1:Customer)-[:MADE]-(tr1:Transaction)-[:PROCESS]-(t:Terminal)
MATCH (c2:Customer)-[:MADE]-(tr2:Transaction)-[:PROCESS]-(t)
WHERE c1 <> c2
WITH c1, c2, t,
    COLLECT(tr1.transaction_security) AS
    c1_security_ratings,
    COLLECT(tr2.transaction_security) AS
    c2_security_ratings
WHERE SIZE(c1_security_ratings) > 3 AND
SIZE(c2_security_ratings) > 3
RETURN c1, c2, t, c1_security_ratings, c2_security_ratings",

"WITH c1, c2, c1_security_ratings, c2_security_ratings
WITH c1, c2,
    REDUCE(s = 0.0, x IN c1_security_ratings | s + x) /
    SIZE(c1_security_ratings) AS c1_avg_security,
    REDUCE(s = 0.0, x IN c2_security_ratings | s + x) /
    SIZE(c2_security_ratings) AS c2_avg_security
WHERE ABS(c1_avg_security - c2_avg_security) < 1
MERGE (c1)-[:BUYING_FRIENDS]-(c2)",

{batchSize: 1000, parallel: false}
);
    """

    session.run(query)
```

Listing 13: Operation d.ii query

The query first gets the customers using the same terminal and check to see that they are distinct. Then the transaction security for each customer is gathered in a collection(array) and filtered based on the size of the collection. If the size is less than 3, it

means the number of transactions for a customer is also less than 3. Next the average security rating is calculated using reduce in which it sums all the security ratings and divides them to their size. At the end the absolute difference of security rates are calculated and filtered and [:BUYING_FRIENDS] relation ship is created between eligible customers. "apoc.periodic.iterate" is used here to execute large and complex query in manageable batches. Here the size of each batch is 1000. It can reduce memory usage since the amount of records needed for this query can get very high.

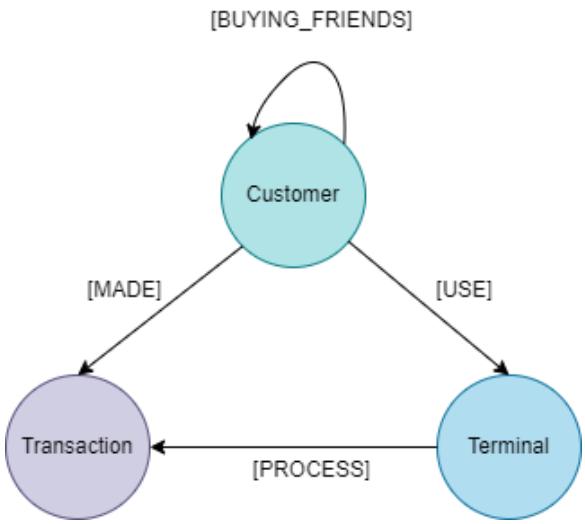

Figure 4: Updated Neo4j Graph Model

5.5 Operation e

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

```
MATCH (tr:Transaction)
WITH tr.transaction_period AS period,
     COUNT(tr) AS totalTransactions,
AVG(CASE WHEN tr.transaction_security < 3 THEN 1.0 ELSE 0.0
 ↪  END)
AS avgFraudulent
RETURN period, totalTransactions, avgFraudulent
```

Listing 14: Operation e query

Assuming that transaction with security number less than 3 is fraudulent, the query shows the percentage of fraudulent transactions in each period of day and total number of transactions.

5.6 Performance

The performance of the queries on the database is show in the table below with the time of uploading datasets to database. The numbers are in **seconds**:

Size	a	b	c	d.i.1	d.i.2	d.i.3	d.ii	e
50 MB	11.84	13.27	3.65	9.52	75	68	635	0.8
100 MB	72	78	52.74	206	183	153	2396	1.25
200 MB	73.22	86.14	152.69	332.61	276.6	251.35	3299	2.41

Table 2: Performance of queries on different sizes of datasets

Executing operations d.ii was quite challenging specifically for 100MB and 200MB since there were lots of customers and terminals and checking each of them without batching the operations and early filtering caused the system to be out of memory. This can be seen in the chart above that after taking care of the memory issues, it still took a considerable amount of time to be executed.