MongoDB for Credit Card Fraud Detection

Davide Cazzetta and Riccardo Giussani

Università degli Studi di Milano Statale, Milan MI 20133, Italy
https://www.unimi.it/it/corsi/corsi-di-laurea/informatica-magistrale

Abstract. The world of Big Data continues to grow and evolve day by day, and with it the technologies capable of handling huge amounts of data. The problem is to choose the right technology and exploit it in a way that optimises processes working with these data. The aim of this project is to study and use the potential of a NoSQL system such as MongoDB for storing, extracting and analysing large quantities of data concerning a system of credit card fraud detection.

To do this, we generated plausible data, saved it in MongoDB, distributed it in the most efficient way for us, run several queries and finally analysed the performance of the system. The results showed that the MongoDB database, if used in the right way, is able to efficiently manage large amounts of data.

Based on the results, it is clear that in the world of Big Data, if the technologies to manage it are not used and optimised properly, the performance of the system becomes unacceptable. Specifically, in MongoDB, the modelling of collections and documents proved to be fundamental.

1 Introduction

1.1 Motivation

Payment card fraud is a major challenge for business owners, payment card issuers, and transactional services companies, causing every year substantial and growing financial losses. Many Machine Learning approaches have been proposed in the literature that tries to automate the process of identifying fraudulent patterns from large volumes of data. The focus of this project aims at this last part: we exploit some algorithms that generate data about transactions to store these data into a NoSQL Database, specifically MongoDB[1]. In detail we want to see how does this system behave in the presence of large amount of data, evaluating the execution times for some operations.
The figure 1 shows the UML diagram.

[1] https://www.mongodb.com/

1.2 Conceptual Model

In this scenario there are two stand-alone entities, that are Customer and Terminal, linked by the relationship Transaction when an event of this kind occurs. The two are also related through available_terminal, relationship that subsists whether in our model the customer can reach a terminal to execute a transaction.

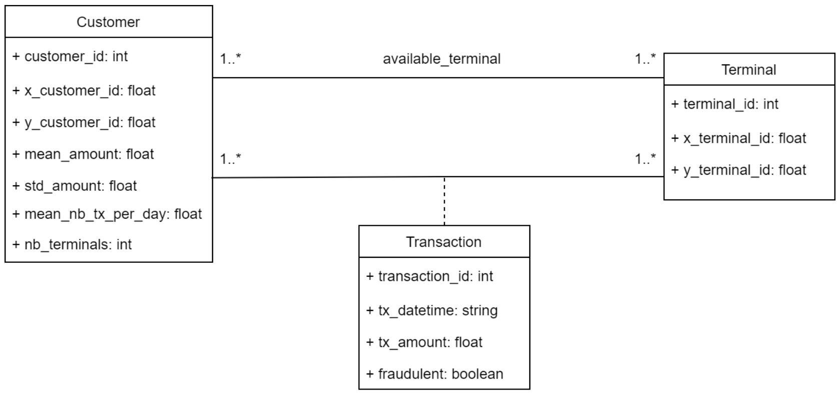

Fig. 1: UML of the generated data

In a relational database, Transaction would be the join table between Customer and Terminal.
These are constraints and suppositions that can't be represented in the UML diagram:

1. a transaction by a customer on a terminal can happen if and only if the relationship available_terminal occurs among the two available_terminal occurs among a customer and a terminal if and only if the terminal's coordinates lie within a certain range from the customer's
2. geographical coordinates are real numbers between 0 and 100

2 Modelling

Transactions are the core entity of the scenario (and consequently of the workload) and therefore all transactions are stored in a dedicated collection.
After ingesting data into the database we compute the cardinalities of the relationships (average and maximum) to grasp a general idea on how much data we have to deal with and how it scales. The figure 2 shows the results.

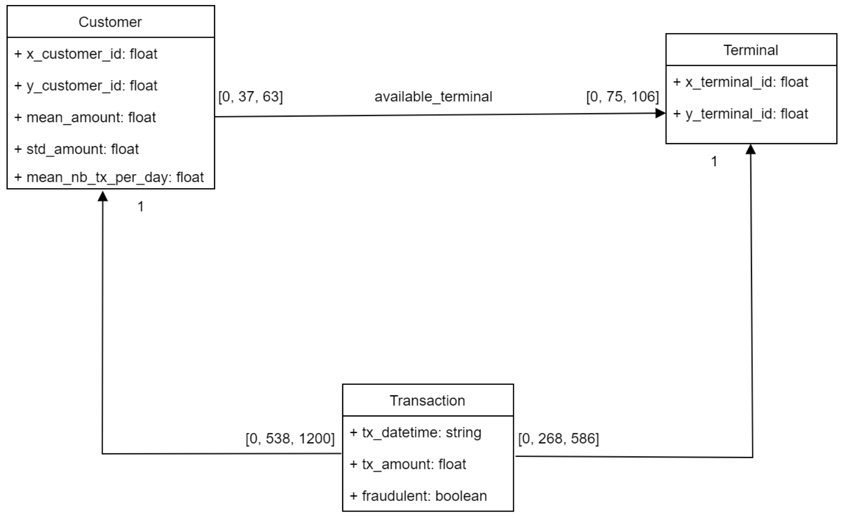

Fig. 2: Cardinality of relationships

The available_terminal relationship have manageable cardinalities on both sides. The data generating script already provides the list of terminals for each customer and therefore this navigability is maintained by reference on the Customer side.
Queries tell us how we should theoretically navigate relationships:

a) requires navigability from Customer to Transaction, to compute the sum
b) requires to navigate from Terminal to Transaction, in order to compute the average
c) requires both the navigability from Customer to Terminal and vice versa
d) only requires transactions data
e) only requires transactions data

The figure 3 represents the final logical schema with cardinalities and navigabilities.

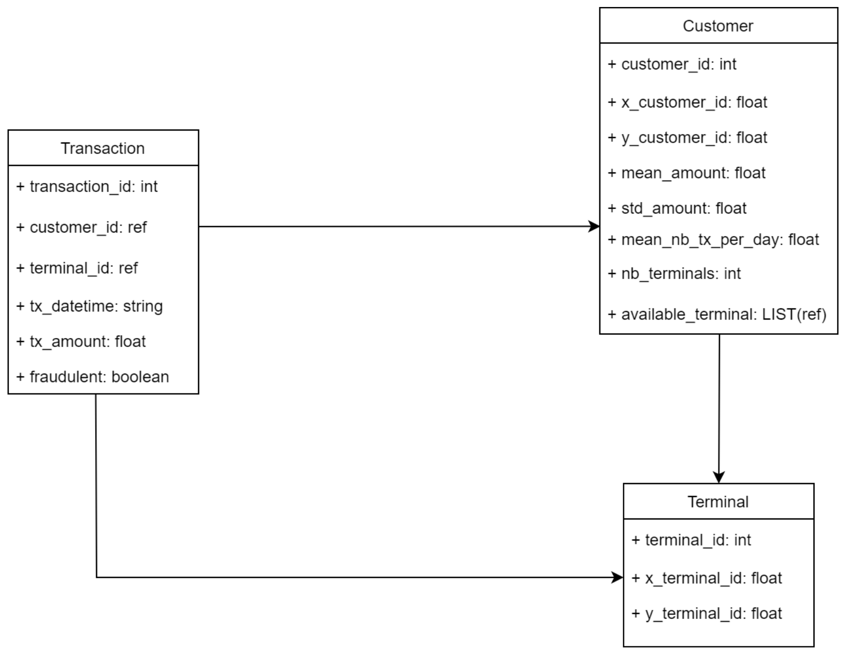

Fig. 3: Navigability in the baseline model

Both the Customer-Transaction and the Terminal-Transaction are relationships with cardinality one-to-zillion and therefore it is absolutely impractical to represent any reference on the one-side. Also, data of customers and terminals are not needed in the queries, therefore no embedding is required. For our baseline model, we keep the relationships on the many side: Transaction references to Customer and Terminal through the fields *customer_id* and *terminal_id*. The following code listings present the MongoDB collections schemes.

```
{
        "_id" : ObjectId,
        "customer_id" : Int,
        "x_customer_id" : Double,
        "y_customer_id" : Double,
        "mean_amount" : Double,
        "std_amount" : Double,
        "mean_nb_tx_per_day" : Double,
        "available_terminals" : List(refTerminalID),
        "nb_terminals" : Int
}
```

Listing 1.1: Customers collection

```
{
        "_id" : ObjectId,
        "terminal_id" : Int,
        "x_terminal_id" : Double,
        "y_terminal_id" : Double
}
```

Listing 1.2: Terminals collection

```
{
        "_id" : ObjectId,
        "TRANSACTION_ID" : String,
        "TX_DATETIME" : Int,
        "customer_id" : Int,
        "terminal_id" : Int,
        "TX_AMOUNT" : Double,
        "TX_TIME_SECONDS" : Int,
        "TX_TIME_DAYS" : Int,
        "TX_FRAUD" : Int,
        "TX_FRAUD_SCENARIO" : Int
}
```

Listing 1.3: Transactions collection

3 Ingestion Layer

The code is available at the following url `https://github.com/Davydhh/MongoDB-for-Fraud-Detection.git`.

The ingestion layer is responsible for generating the datasets which will then be stored in MongoDB.

3.1 Generate datasets

The system can generate three datasets, one about customers, one related to terminals and the third regarding transactions. To generate these datasets we have exploited the scripts presented in this article: `https://fraud-detection-handbook.github.io/fraud-detection-handbook/Chapter_3_GettingStarted/SimulatedDataset.html`. In particular we have created a class *DatasetsGenerator* where the user can pass the parameters *start_date* which represents the day on which transactions begin and *nb_days*, that is days of transactions. The decision to make only these two parameters arbitrary is due to the fact that the biggest dataset is about transactions, on the other hand the customers and terminals dataset are are much smaller in comparison, as a result, the customer and terminal parameters are constant at 5000 and 10000 respectively.

We generate 5000 customers and 10000 terminals and then three datasets of transactions of increasing size, simulating iteratively 40 days, 80 days and 160 days of transactions. The first iteration starts from date 2018-04-01. Subsequently, each iteration increases the year.
The sizes of the datasets are as follows:

1. – customers: 3 MB
   – terminals: 1 MB
   – transactions (2018): 74 MB
2. – customers: 3 MB
   – terminals: 1 MB
   – transactions (2019): 147 MB
3. – customers: 3 MB
   – terminals: 1 MB
   – transactions (2020): 296 MB

3.2 Store data in MongoDB

The second part of the ingestion layer consists of storing the data in MongoDB. The datasets are generated in json format to facilitate saving in the database; the data are stored with the *updateMany*[2] method. In particular, to simulate a real scenario where the number of transactions increases over time, transaction data are saved at each iteration, whereas customer and terminal datasets are only saved at the first iteration as they are the same for all three generated datasets. In this way, transaction datasets are merged at the time of saving in the database.
The json format does not support the *DateTime* type and therefore all dates are imported as milliseconds (an *Int64* number); before running the queries we must convert the data into the *DateTime* type, available for the bson format inside MongoDB.

4 Queries

This section explains in detail how the queries were implemented to obtain the required results.
Some queries require to work on the "current month" or the "last month". We suppose to take the date of the last transaction made as the "present" date and we identify its month as the required month. This is done to ensure that the system always gives a response: in a real world scenario, we'd take the date of today.

[2] https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/

4.1 Query A

For each customer identifies the amount that he/she has spent for every day of the current month.

Since we need to have information regarding the day, month and year in which each transaction took place we need to break down the *TX_DATETIME* field so that we have this information separate. To do this we extract the day with the *$dayOfMonth*[3], the month with *$month*[4] and the year with *$year*[5]; all this operation are made inside a *$project*[6] stage.
The second step is filter data about the current month through a *$match* stage; to obtain the current month the *datetime*[7] Python library is used.
The last step consists in grouping the data by user and by day by summing the *TX_AMOUNT* field with the *$group*[8] and *$sum*[9] methods.

The resulting query consists of a pipeline aggregation on the transactions collection.

4.2 Query B

For each terminal identify the possible fraudulent transactions. The fraudulent transactions are those whose import is higher than 50% of the average import of the transactions executed on the same terminal in the last month.

Since for each terminal we need to have all transactions, we use the *$lookup*[10] method, starting from the terminals collection to the transactions collection. After that we create another field (in a *$project* stage) named *transactions_avg* in which we compute the average import of the transactions. Before doing all the calculations, it is important to note that we have interpreted "last month" as the current month of the current year, so, similar to query A, it is necessary to extract information about the month and year in which each transaction took place. Since it is necessary to process a field belonging to objects within an array, we used the *$map*[11] method inside a *$project* stage, which allowed us to process each individual element of the *transaction_avg* array by adding the *month* and *year* fields. In the next step we eliminate the terminals that had no transactions in the last month through a *$match* method over the *transactions_avg.month* and *transactions_avg.year* fields. Than, the method

[3] https://docs.mongodb.com/manual/reference/operator/aggregation/dayOfMonth/
[4] https://docs.mongodb.com/manual/reference/operator/aggregation/month/
[5] https://docs.mongodb.com/manual/reference/operator/aggregation/year/
[6] https://docs.mongodb.com/manual/reference/operator/aggregation/project/
[7] https://docs.python.org/3/library/datetime.html
[8] https://docs.mongodb.com/manual/reference/operator/aggregation/group/
[9] https://docs.mongodb.com/manual/reference/operator/aggregation/sum/
[10] https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/
[11] https://docs.mongodb.com/manual/reference/operator/aggregation/map/

$filter*[12] inside a *$project stage is used to get only the transactions executed in the last month. At this point it is possible to obtain the value relative to 50% of the average import of the transactions with the *$avg* and *$divide*[13] methods. Finally, the last operations to be carried out serve to obtain the fraudulent transactions because for each terminal we now have the reference to the threshold value; a *$match* and *$project* stages are exploited to get only transactions whose import is higher than the *transactions_avg* field.

The resulting query consists of a pipeline aggregation on the terminals collection.

4.3 Query 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 "$u_1 - t_1 - u_2 - t_2 - \ldots t_k - 1 - u_k$" such that $u_1 = u$, $u_k = u'$, and for each $1 \leq I, j \leq k$, $u_i <> u_j$, and $t_1, \ldots t_k - 1$ are the terminals on which a transaction has been executed. Therefore, $CC_k(u) = u'$— a chain exists between u and u' of degree k. Please, note that depending on the adopted model, the computation of $CC_k(u)$ could be quite complicated. Consider therefore at least the computation of $C_C3(u)$ (i.e. the co-costumer relationships of degree 3).

The determination of such a relationship would require, at an high-level perspective, the traversal of a graph where each customer is linked to each terminal that they had used at least once and vice versa. The only information explicitly reported in the database linking customers and terminals is the available_terminal list inside each document of the Customer collection. This is however not sufficient, as we can't say whether the customer has used a terminal at least once and we'd also need to link each terminal to customers that have used it at least once.
Getting for each terminal the customers that have used it requires a simple grouping over the Transaction collection by *terminal_id* that accumulates all the *customer_id* in the set called *cust_used_once* through the command *$addToSet*[14].
As we are allowed to restrain ourselves to compute only the relationship of degree 3, we opt for a set manipulation approach.
Given two terminals, each with its list of customers *cust_used_once*$_1$ and *cust_used_once*$_2$ (from now on simply t_1 and t_2), we identify three sets:

1. $t_1 \cap t_2$: customers that have used both terminals
2. $t_1 \setminus t_2$ customers that have used terminal 1 but not terminal 2
3. $t_2 \setminus t_1$ customers that have used terminal 2 but not terminal 1

[12] https://docs.mongodb.com/manual/reference/operator/aggregation/filter/
[13] https://docs.mongodb.com/manual/reference/operator/aggregation/divide/
[14] https://docs.mongodb.com/manual/reference/operator/aggregation/addToSet/

Given the relationship definition of degree 3:
$u_1 - t_1 - u_2 - t_2 - u_3$
Intuitively we see that customers in $t_1 \setminus t_2$ can only cover the role of u_1 and customers in $t_2 \setminus t_1$ the role of u_3. Those in $t_1 \cap t_2$ are the only ones that can be in the place of u_2. Note that the three sets are disjointed.
We notice that it's sufficient that $t_1 \cap t_2$ is nonempty to determine that each customer in $t_1 \setminus t_2$ is a co-customer of degree 3 to each customer in $t_2 \setminus t_1$, as a "bridge" can be created. Also, if $|t_1 \cap t_2| > 1$ each customer in $t_1 \setminus t_2$ is a co-customer of degree 3 to each customer in $t_1 \cap t_2$.
We are only interested in co-customers for a user-given customer, therefore among all possible terminals we select as t_1 only those that have the specified customer inside. Then, each t_1 is matched with any terminal's t_2 and we compute the sets $t_1 \cap t_2 \setminus \{target\_customer\}$ and $t_2 \setminus t_1$. The following logic is applied:

```
if $|t_1 \cap t_2| \geq 0$ then
    $co\_customers \leftarrow t_1 \cap t_2$
    if $|t_1 \cap t_2| \geq 1$ then
        $co\_customers \leftarrow t_2 \setminus t_1$
```

The arrows are to be interpreted as "insert all elements of the specified set in the target set".

4.4 Query D

Point i

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

The label corresponding to the period of the day strictly depends on the value of the field *TX_DATETIME*, precisely from the time of day. For this reason we have extract the hour with the *$hour* method creating a new field using the *$addToField*[15] method. After that we generate a new field named *period* which will contain the string representing the period of the day. To determine the value we use the Mongo command *$switch*[16]. The logic is:

```
if $6 \leq$ hour $\leq 12$ then
    $period \leftarrow$ morning
else
    if $12 <$ hour $\leq 18$ then
        $period \leftarrow$ afternoon
    else
        if $18 <$ hour $\leq 22$ then
```

[15] https://docs.mongodb.com/manual/reference/operator/aggregation/addFields/
[16] https://docs.mongodb.com/manual/reference/operator/aggregation/switch/

```
        period ← evening
    else
        period ← night
```

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

Since we need to add a field whose content is to be generated automatically from a series of words, we apply a Javascript function to each row using the Mongo command *forEach*[17]. This method works but the library *PyMongo* has no API to call this Mongo function, so we code this in Python.
We retrieve all transactions where the field *product_kind* doesn't exist and then for each of them we apply an *updateOne*[18] where the value of the field *product_kind* is chosen randomly from a list of possible choices.

Point ii *Customers that make more than three transactions related to the same types of products from the same terminal should be connected as "buying_friends". Therefore also this kind of relationship should be explicitly stored in the NOSQL database and can be queried.*

We take into account the Transaction collection and specify an aggregation pipeline. As first step we *$group* by *terminal_id*, *customer_id* and *product_kind* and count the number of occurrences of each combination of values; then we *$match* only those triplets that have a count of occurrences strictly greater than 3. The last step requires a *$group* using as id *terminal_id* and *product_kind*. The accumulator function is an *$addToSet* that puts in a list (without duplication) all the *customer_id*.
This pipeline returns, for each combination of *terminal_id* and *product_kind*, the customers that have made more than 3 transactions with those values. Notice that the buying_friend relationship, in general, is not transitive and the cardinality can be too big to represent by reference all the ids of friends inside the customer document. See figure 4.
When considering the same relationship constrained on terminal and product kind, we see that the cardinality is much smaller and it is transitive, which has the effect that a customer is buying friend to all customers that have bought the same product kind on the same terminal more than three times. See figure 5.
We specify a new entity, Buying_Group, toward which the Customer is linked to by a relationship part_of. When a customer is a part of a group, they are friends with all other customers that are part of the same group.
We keep the array of references to customer inside the buying_group document. See figure 6.
This representation is good when we want all the buying_friends given *termi-*

[17] https://docs.mongodb.com/manual/reference/method/cursor.forEach/
[18] https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/

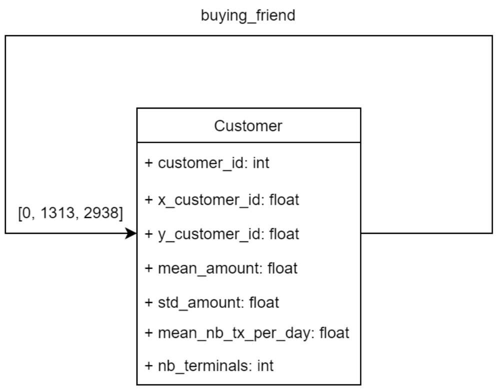

Fig. 4: buying_friend Relationship, not constrained by terminal and kind

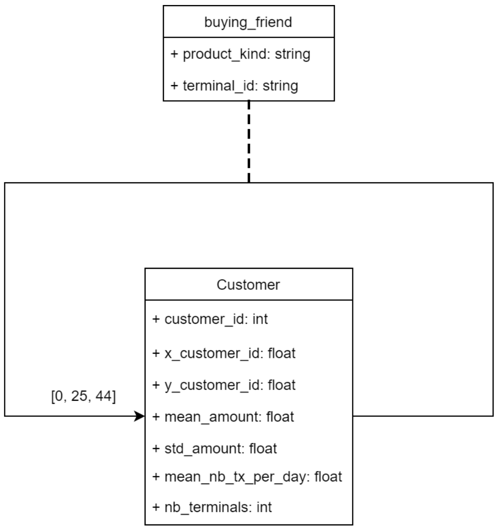

Fig. 5: buying friend Relationship

nal_id and *product_kind*, but is lacking if we require all the buying friends for a given customer.

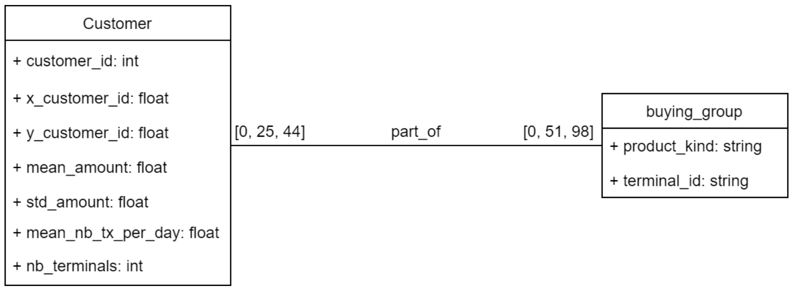

Fig. 6: part_of buying_group relationship

4.5 Query E

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

This query involves a simple *$group* stage by *$period* in aggregation where we use two accumulators: a *$sum*, to count the number of transactions, and *$avg*[19], to compute the average number of fraudulent transactions.

5 Performance

To evaluate the performance we calculate the execution times for all the queries described before. To do this, in the case of an aggregation, the database command *explain*[20] is used. In all other cases, e.g. for update operations, the time is obtained through the *time*[21] Python library, calculating the difference between the time the method finished and the time it started.

To simulate a real scenario, the best thing to do would be to perform all operations in a database residing on the cloud on a different machine from the one where the queries are executed. Unfortunately, having subscribed a free plan on the Atlas website[22], at a certain point the script generates the following exception: *pymongo.errors.OperationFailure: Executing this update would put you over your space quota, full error: 'ok': 0, 'errmsg': 'Executing this update would put you over your space quota', 'code': 8000, 'codeName': 'AtlasError'*. For this reason we decided to perform all the operations in a local database, so the execution time is reduced compared to a real scenario.

Performances results (expressed in seconds) are as follows (note that every transactions dataset is added with the previous one, so at the first iteration we

[19] https://docs.mongodb.com/manual/reference/operator/aggregation/avg/
[20] https://docs.mongodb.com/manual/reference/command/explain/#mongodb-dbcommand-dbcmd.explain
[21] https://docs.python.org/3/library/time.html
[22] https://www.mongodb.com/atlas/database

have transactions about 2018, at the second transactions about 2018 and 2019 and at the last iteration transactions about 2018, 2019 and 2020):

	Query A	Query B	Query C	Query D			Query E	Ingestion
				Point i		Point ii		
				1	2			
Dataset 1	1,5	0,1	1,5	10	7,7	27	0,6	18,3
Dataset 2	4,2	0,2	3,6	19,5	14,9	71	3,4	34,2
Dataset 3	9,4	0,3	8	38,9	30	84,2	4	72

Table 1: Performances

6 Design Patterns

The queries investigated rely on aggregations. Here we propose possibilities to speed-up the queries by making some computation during ingestion to avoid aggregating after.

6.1 Computed pattern

This pattern is very useful for responding to the requests of query B as it is a complex computation and also generally returns the same (or almost the same) results if requested frequently and within a short interval of time. In particular the query requests for each terminal the average import of the transactions executed in the last month. Since we interpret "last month" with the current month we can precompute the total import of the transactions executed in the current month and the count of them. To do this we insert new fields in terminals document:

- *last_month*: in which we save the month in which the last transaction took place
- *total_amount_last_month*: represents the total amount for the current month
- *n_transactions_last_month*: is the number of transactions executed in the current month

The logic is as follows:

```
for all $t \in \mathcal{T}ransactions$ do                    ▷ $t$ is the transaction to insert
    if $t.timestamp.month = terminal.last\_month$ then
        db.terminals.update_one({"terminal_id":t.terminal_id},{$inc:
{"total_amount_last_month":t.tx_amount,"n_transactions_last_month": 1}})
    else
```

```
        **if** $t.timestamp.month > terminal.last\_month$ **then**
            *db.terminals.update_one({"terminal_id":t.terminal_id},{$set: {"last_month": t.timestamp.month}, {"n_transactions_last_month": 1, "total_amount_last_month": t.tx_amount}})*
```

This way we avoid calculating the average transaction amount of the current month for each terminal each time, which is computationally a costly operation.

6.2 Subset pattern

We stated that the relationship Customer-Transaction is of the order one to zillion and so unsuitable for embedding or reference in the many side. To obtain the result required by query A we need for each customer the transactions executed in the current month, so it may be useful to embed only these transactions in each *Customer* document. This will certainly lead to some duplication, but we consider it acceptable as we assume that in one month a customer performs a manageable number of transactions. In particular when a transaction takes place, we also duplicate the date and the amount in an array in customer document. When a month passes (and we see that when a new transaction is done in the next month) the data in the array is erased.

This pattern is particularly useful as the number of transactions continues to grow over time, so periodically matching month and year for each transaction becomes a costly operation when there are many transactions. In this way we only need to perform calculations on a small proportion of transactions.
The logic is as follows:

```
$current \leftarrow (0, 0)$
for all $t \in \mathcal{T}ransactions$ do                    ▷ t is the transaction to insert
    $t\_time \leftarrow (t.tx\_datetime.year, t.tx\_datetime.month)$
    db.transactions.insert_one(t)
    if $t\_time = current$ then
        db.customers.update_one({"customer_id": t.customer_id}, {$addToSet: {"month_transactions": t}})
    else
        if $t\_time > current$ then                           ▷ a month has passed
            $current \leftarrow t\_time$
            db.customers.update_all({}, {$set: {"month_transactions": [t]}})
```

7 Conclusion

This project aims to develop a NoSQL database to handle and process a large amount of data about a credit card fraud detection system. Our approach was to analyse the entities and the workload in order to obtain an optimised model for the required operations. Based on the results, it is possible to conclude that

MongoDB is a suitable database for managing Big Data, but only if the modelling is done from the workload, trying to optimise the structure on the basis of the operations to be performed. Surely if there had been no limitation on the Atlas cloud plan, we would have obtained more representative and real results, so a possible improvement could be to carry out operations and analyses on a cloud database through a plan without limitations.