New Generation Data Models and DBMSs

Credit card fraud detection project

AY: 2023/2024

Prof. Marco Mesiti

Matteo Luigi Giovanni Rigat (13888A)

Introduction

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.

Detecting fraud patterns in payment card transactions is known to be a very difficult problem. With the ever-growing amount of data generated by payment card transactions, it has become impossible for a human analyst to detect fraudulent patterns in transaction datasets, often characterized by a large number of samples, many dimensions, and online updates.

For the purposes of this project, we are interested in implementing an appropriate NOSQL database, to be exploited for the identification of fraudulent patterns on large volumes of data.

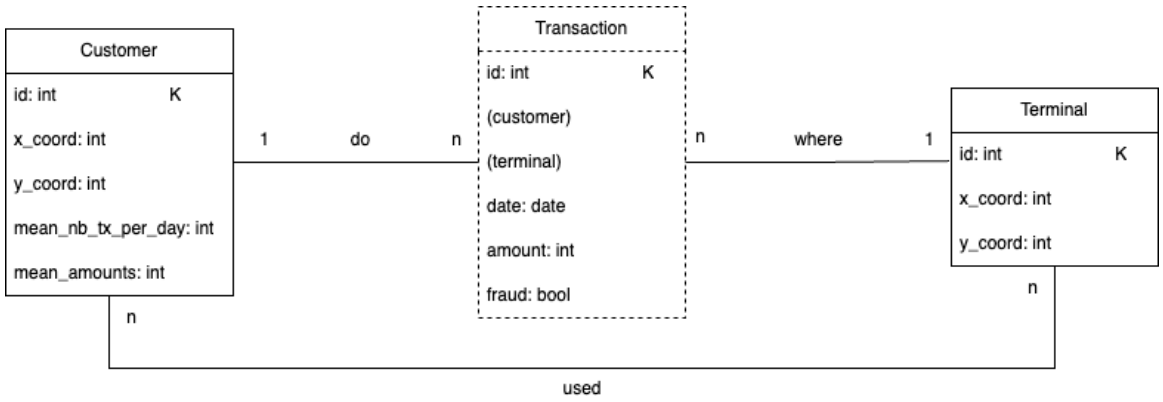
The project requires to generate three tables such that:
Customer profile. Each customer has a unique identifier and is associated with the following properties: geographical location, spending frequency and spending amounts. Moreover, the list of terminals on which the customer makes transactions is associated with his profile.
Terminal profile. Terminal properties consist of a geographical location.
Transactions. This table reports for each transaction, the customer identifier, the terminal identifier, the amount that has been paid, and the date on which the transaction occurred. Some transactions can be marked as fraudulent.

Further details on these tables and the Python scripts for the generation of the tables can be found at https://fraud-detection-handbook.github.io/fraud-detection-handbook

However, the simulator generates, for each of these tables, parameters different from those we are interested in, so we make the following assumptions:

- It generates a unique identifier also for each transaction and terminal. We'll use them since they can be useful to refer to a single transaction instead of using customer_id, terminal_id and date, and instead of geographical coordinates to identify a terminal.
- The simulator generates a list of available terminals nearby the customer. We will use this as the "list of terminals on which the customer makes transactions".
- We will also use `mean_nb_tx_per_day` as spending frequency and `mean_amount` as spending amounts for the customer profile. This last value is not a list as "spending amounts" suggests, but there isn't a list of amounts in the customer profile generator.

The resulting UML diagram is the following.



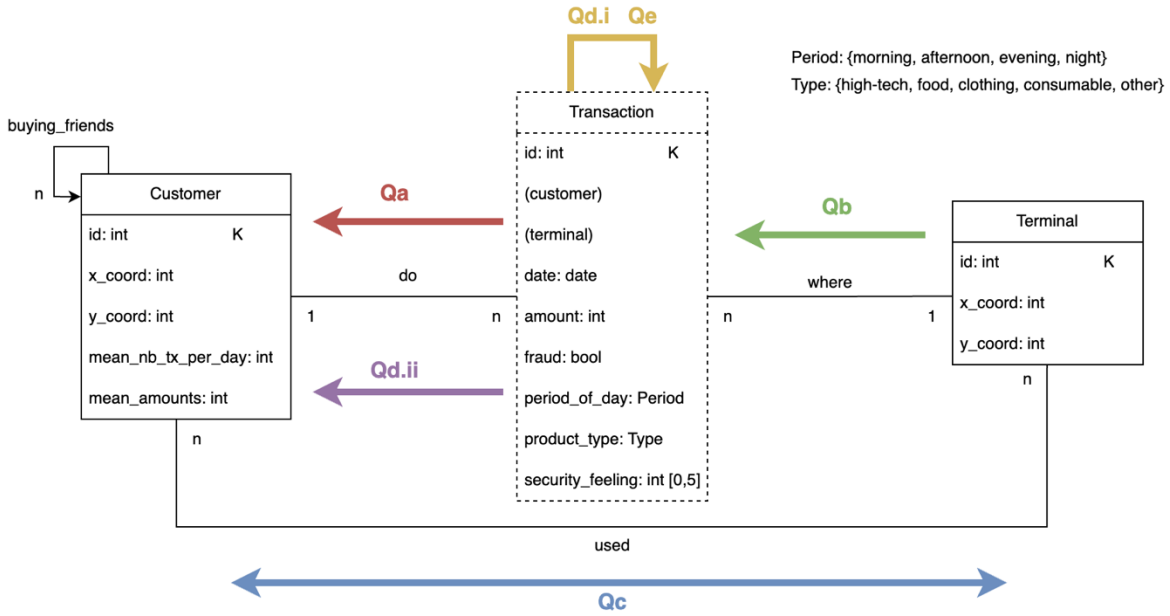
Note that transaction is a weak entity since it can't exist without a customer and a terminal. We have associated a unique id to transaction for the assumptions made previously.

The queries require also to extend the starting model so that 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 product that have been bought through the transaction {high-tech, 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 concludes the transaction.

Furthermore, customers that make more than three transactions from the same terminal expressing a similar average feeling of security should be connected as “buying_friends”.

The final UML considering also the extended tables is the following:



The model also includes the workload related to the query navigation, which we will discuss in depth later.
Now let's briefly explain the reason for the direction of the arrows:

- “a” and “d.ii” can be read as “given these types of transactions identify these customers… ”
- “b” is literally “For each terminal identify the possible fraudulent transactions”
- “c” requires navigating from customer to terminal and coming back, n times.
- “d.i” and “e” are just a search within the transaction entity

Neo4j model

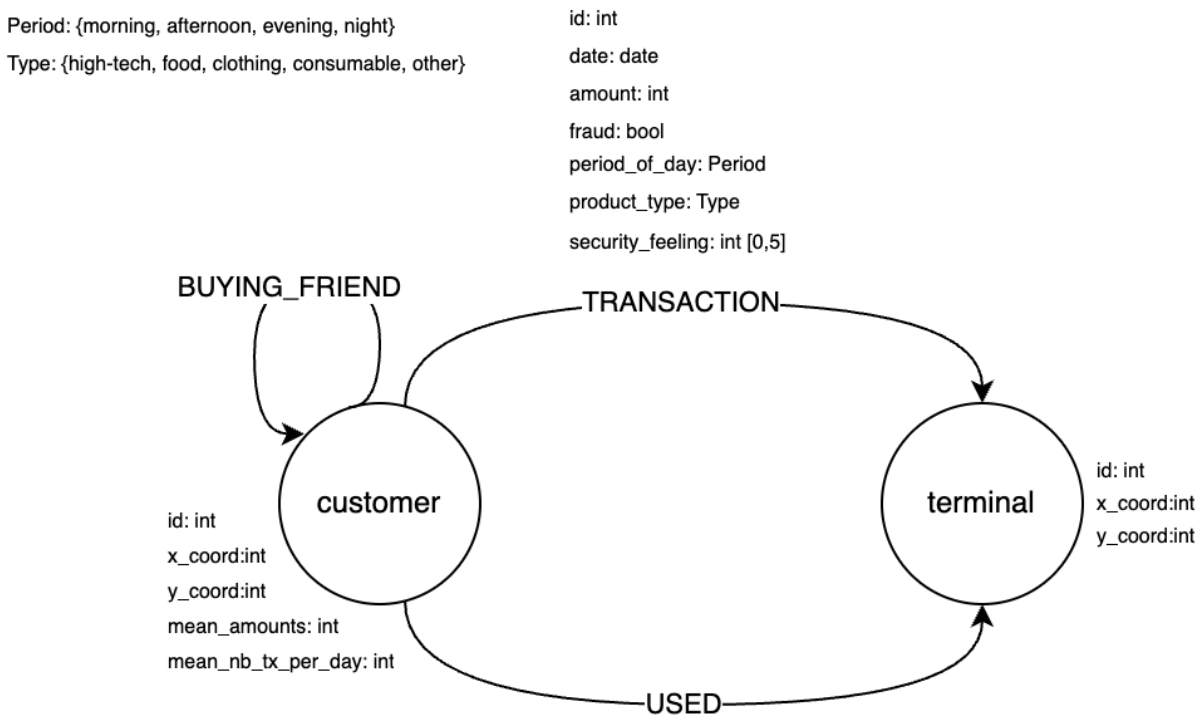
Choose Neo4j as database system is the best choice, thanks to the graph model the transaction is well suited to be a relationship between customer and terminal.
For the same reason it’s also appropriate to represent the “buying_friends” relationship between customers.

Furthermore, if we look at the third query for example, using the cypher query language primitives for the path length, we are able to retrieve the co-customers without writing complex queries.

Obviously, customer and terminal will be the two nodes of our graph.

Therefore, the choice of Neo4j comes from the need to explore entities connected to each other and to support complex navigation operations between nodes, making the graph model the most appropriate.

The neo4j logical model is the following:



Since there are some mismatches between what the queries ask for and what the simulator provides us, we have to make some considerations.

The query “a” asks to compare spending frequency and spending amounts of the last month with the same month of the year before. We only have the overall frequency and the average amounts of a customer, so we need to calculate them from the transaction table if we want to retrieve the statistics for last year.
The mean_amounts and mean_nb_tx_per_day values are useless for the query we want to answer.

The query “c” asks to use the “USED” relationship to retrieve customers who used the same terminal, but the list the simulator provided us, is completely random generated and it doesn’t have nothing in common with the transactions that are in the transaction table and that the customer actually carried out.
So, I decided to answer the question by ignoring this list and I considered as “used terminals” only those that have a transaction relationship between the customer and the terminal.

Certainly the "USED" relationship can be resumed to apply the Materialized Paths Pattern, which we will discuss in depth later, to speed up queries of this type, where we are only interested in the single customer-terminal relationship without caring of the various transactions between them.

Uploading Queries

customer.csv

```
for customer in customers:
    records, summary, keys = driver.execute_query(
        """
        MERGE (c:Customer {customer_id: toInteger($customer.CUSTOMER_ID),
        x_customer_id: toFloat($customer.x_customer_id),
        y_customer_id: toFloat($customer.y_customer_id),
        mean_amount: toFloat($customer.mean_amount),
        std_amount: toFloat($customer.std_amount),
        mean_nb_tx_per_day: toFloat($customer.mean_nb_tx_per_day),
        available_terminals: $customer.available_terminals,
        nb_terminals: toInteger($customer.nb_terminals)})
        """,
        customer=customer, database_=database,
    )
```

Each customer is passed as a parameter, along with all its properties.
The MERGE operator is used to combine the behavior of CREATE and MATCH into a single operation.
First it checks whether the node already exists and creates it if not. If the node already exists, it retrieves the node without modifying it. This makes MERGE particularly useful for ensuring the uniqueness of the data within the graph.

terminal.csv

```
for terminal in terminals:
    records, summary, keys = driver.execute_query(
        """
        MERGE (t:Terminal {terminal_id: toInteger($terminal.TERMINAL_ID),
        x_terminal_id: toFloat($terminal.x_terminal_id),
        y_terminal_id: toFloat($terminal.y_terminal_id)})
        """,
        terminal=terminal, database_=database,
    )
```

Same thing for loading terminals.

transaction.csv

```
for transaction in transactions:
    transaction['TX_DATETIME'] =
         datetime.strptime(transaction['TX_DATETIME'], "%Y-%m-%d %H:%M:%S")
    records, summary, keys = driver.execute_query(
        """
        MATCH (customer:Customer
              {customer_id:toInteger($transaction.CUSTOMER_ID)})
        MATCH (terminal:Terminal
              {terminal_id: toInteger($transaction.TERMINAL_ID)})
        MERGE (customer)-[:MADE_TRANSACTION
              {transaction_id: toInteger($transaction.TRANSACTION_ID),
               tx_datetime: datetime($transaction.TX_DATETIME),
               tx_amount: toFloat($transaction.TX_AMOUNT),
               tx_time_seconds: toInteger($transaction.TX_TIME_SECONDS),
               tx_time_days: toInteger($transaction.TX_TIME_DAYS),
               tx_fraud: toInteger($transaction.TX_FRAUD)}]->(terminal)
        """,
        transaction=transaction, database_=database,
    )
```

To create the MADE_TRANSACTION relationship we first have to retrieve the customer and the terminal we want to connect.

Queries

Qa

“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.”

Since generating transactions for more than a year back becomes computationally complicated, we assume that it is meant as the average of transactions over the same month of the previous year.
Therefore, we take into consideration two different periods: the previous month and the same month of the previous year.

Query steps:
MATCH: takes all the transactions and filters the two periods in the WHERE clause
WITH: calculates the frequency and the sum of the amounts for the two periods
RETURN: compares the two periods and returns the customers

```
"""
MATCH (c:Customer)-[t:MADE_TRANSACTION]->()

WHERE (t.tx_datetime >= datetime($parameters.start_of_last_month)
   AND t.tx_datetime < datetime($parameters.start_of_current_month))

   OR (t.tx_datetime >= datetime($parameters.start_of_last_year)
   AND t.tx_datetime < datetime($parameters.end_of_last_year))

WITH c,
  COUNT(CASE
       WHEN t.tx_datetime >= datetime($parameters.start_of_last_month)
        AND t.tx_datetime < datetime($parameters.start_of_current_month)
       THEN 1 ELSE NULL END) AS lastMonthFrequency,
  SUM(CASE
       WHEN t.tx_datetime >= datetime($parameters.start_of_last_month)
        AND t.tx_datetime < datetime($parameters.start_of_current_month)
       THEN toFloat(t.tx_amount) ELSE 0.0 END) AS lastMonthAmount,

  COUNT(CASE
       WHEN t.tx_datetime >= datetime($parameters.start_of_last_year)
        AND t.tx_datetime < datetime($parameters.end_of_last_year)
       THEN 1 ELSE NULL END) AS lastYearFrequency,
  SUM(CASE
       WHEN t.tx_datetime >= datetime($parameters.start_of_last_year)
        AND t.tx_datetime < datetime($parameters.end_of_last_year)
       THEN toFloat(t.tx_amount) ELSE 0.0 END) AS lastYearAmount

RETURN c.customer_id AS customerId,
       CASE
          WHEN lastMonthFrequency < lastYearFrequency
          THEN "under the usual"
          ELSE "over the usual"
       END AS spending_frequency,
       CASE
          WHEN lastMonthAmount < lastYearAmount
          THEN "under the usual"
          ELSE "over the usual"
       END AS spending_amounts
"""
```

Note that, at the expense of readability, making a single MATCH and subsequently filtering the two periods is almost twice as fast as filtering the two periods separately.

Qb

"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."

Query steps:
MATCH: identifies every couple of transactions made in the same terminal, "fraud" is the possible fraudulent transaction.
WHERE: selects the ones that are executed within one month one from the other.
WITH: calculates the max amount for every possible fraudulent transaction
WHERE: selects only the fraudulent transaction

```
"""
MATCH ()-[t:MADE_TRANSACTION]->(tm:Terminal)<-[fraud:MADE_TRANSACTION]-()

WHERE t.tx_datetime >= fraud.tx_datetime - duration({months: 1})
  AND t.tx_datetime < fraud.tx_datetime

WITH tm, fraud, MAX(t.tx_amount) AS max_amount

WHERE fraud.tx_amount > max_amount*1.2

RETURN DISTINCT tm.terminal_id as terminalId,
       COLLECT(fraud.transaction_id) as transactionsId

ORDER BY tm.terminal_id
"""
```

Qc

"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)."

If we unroll the chain of the $CC_3(u1)$ co-customers we obtain:

```
(u1)-[:MADE_TRANSACTION]->(t1)<-[:MADE_TRANSACTION]-(u1)-
[:MADE_TRANSACTION]->(t2)<-[:MADE_TRANSACTION]-(u3)
```

We can count four different "MADE_TRANSACTION" relationships. The query in these terms is only to identify this chain using the "*4" notation in the relationship declaration. If we want to find longer chains the path length is 2(k-1) with k defined as above.
Note that we can answer the query in this way only assuming that between two customer we always find a terminal. Condition verified by the domain we have.

```
"""
MATCH p=(c1:Customer {customer_id: toInteger($parameters.user_id)})-
[:MADE_TRANSACTION*4]-(c2:Customer)

WHERE c1 <> c2

RETURN DISTINCT c2.customer_id as customerId LIMIT 1000
"""
```

Qd.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 {high-tech, 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.”

Trying to execute the entire query at once will fill the entire database memory and will fail. We need to divide it into different batches.

```
"""
MATCH ()-[t:MADE_TRANSACTION]->()
RETURN t.transaction_id
"""
```

First retreive all the transactions. divide them into batches and perform them one at a time.

```
transaction_ids = [record["t.transaction_id"] for record in records]
BATCH_SIZE = 1000

for i in range(0, len(transaction_ids), BATCH_SIZE):
    batch = transaction_ids[i:i+BATCH_SIZE]
    records, summary, keys = driver.execute_query(
        """
        UNWIND $batch AS trans
        WITH collect(trans) AS transactionList
        MATCH ()-[t:MADE_TRANSACTION]->()
        WHERE t.transaction_id IN transactionList

        SET
          t.period_of_day =
          CASE
            WHEN  datetime(t.tx_datetime).hour >= 6
              AND datetime(t.tx_datetime).hour < 12   THEN "morning"
            WHEN  datetime(t.tx_datetime).hour >= 12
              AND datetime(t.tx_datetime).hour < 18   THEN "afternoon"
            WHEN  datetime(t.tx_datetime).hour >= 18
              AND datetime(t.tx_datetime).hour < 24    THEN "evening"
            ELSE "night"
          END,

          t.product_type =
          CASE
            WHEN $rand < 0.2 THEN "high-tech"
            WHEN $rand < 0.4 THEN "food"
            WHEN $rand < 0.6 THEN "clothing"
            WHEN $rand < 0.8 THEN "consumable"
            ELSE "other"
          END,

          t.security_feeling = toInteger(rand() * 5) + 1
        """,
        batch=batch, rand=random.random(), database_=database
    )
```

Probably executing the three requests separately doesn’t drain all the memory, but you have to visit all the relationships three times, and transactions are zillions.

Qd.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."

We have to divide the query into batches also in this case.

```
"""
MATCH (c1:Customer)-[t1:MADE_TRANSACTION]->(t:Terminal)<-
[t2:MADE_TRANSACTION]-(c2:Customer)

WITH c1, c2, COUNT(DISTINCT t1) as count1, COUNT(DISTINCT t2) as count2,
     AVG(t1.security_feeling) as avg1, AVG(t2.security_feeling) as avg2

WHERE c1<>c2 AND count1 > 3 AND count2 > 3 AND ABS(avg1-avg2)<1

RETURN c1.customer_id as c1, c2.customer_id as c2
"""
```

First, we retrieve all the couples of "buying_friends", the one that made more than three transactions from the same terminal and whose have similar average feelings.

Then, we divide them into batches and we have only to create the relationship between the two customers.

```
customers_ids = list((record["c1"], record["c2"]) for record in records)
BATCH_SIZE = 1000

for i in range(0, len(records), BATCH_SIZE):
    batch = customers_ids[i:i+BATCH_SIZE]
    records, summary, keys = driver.execute_query(
        """
        UNWIND $batch AS customers
        MATCH (c1:Customer {customer_id: toInteger(customers[0])})
        MATCH (c2:Customer {customer_id: toInteger(customers[1])})
        MERGE (c1)-[:BUYING_FRIENDS]-(c2)
        """,
        batch=batch, database_=database
    )
```

Qe

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

Here we only have to group by period_of_the_day and counting.

```
"""
MATCH ()-[t:MADE_TRANSACTION]->()

WITH DISTINCT t.period_of_day as period, count(t) as num, COUNT(CASE WHEN
t.tx_fraud > 0 THEN 1 END) as numFraud

RETURN period, num, round(toFloat(numFraud)/toFloat(num)*100, 2) as
fraudPERCENT
"""
```

Times and performance improvements with patterns

To generate the three datasets, I used the following parameters:

parameters	50MB (64,5 MB real)	100MB (109,4 MB)	200MB (222,8 MB)
n_customer	1500	2500	5000
n_terminals	1500	2500	5000
nb_days	400	400	400
start_date	2023-02-01	2023-02-01	2023-02-01

generated transactions	1.147.068	1.912.484	3.835.997

All the following times take into account the database after the updates introduced by the query "d".
Upload queries were calculated only once due to their high values and may not be accurate.

query	50MB	100MB	200MB
upload	3h 9m 42s	6h 41m 52s	12h 1m 15s
a	876ms	1s 648ms	6s 814ms
b	7m 16s 114ms	11m 25s 978ms	21m 39s 852ms
c	4m 16s 633ms	7m 5s 647ms	10m 19s 922ms
d.i	7m 56s 443ms	21m 38s 527ms	1h 32m 19s
d.ii	4m 33s 338ms	7m 34s 629ms	33m 36s 116ms
e	703ms	1s 115ms	2s 579ms

Looking at the times above it seems the performances are very low, but we don't have to forget that we continuously iterate on millions and millions of results.
Are there faster methods? is driver.execute_query( ) the right way to execute queries? Let's compare it with session.run.

Timing comparison for 50MB dataset

query	driver.execute_query	session.run	Neo4j Desktop
a	876ms	996ms	943ms
b	6m 19s 653ms	6m 40s 472ms	6m 49s 114ms
c	4m 16s 633ms	4m 16s 857ms	4m 16s 986ms
e	703ms	730ms	687ms

There is no difference between the two methods, and also compared to the Neo4j desktop terminal.

So, the only way is to simplify the queries or to adopt patterns to make the database more responsive.
Let's discuss some approaches below.

Certainly, one of the patterns that we can apply to significantly increase the performance, and which we mentioned before, is the Materialized Paths Pattern which consists in storing pre-computed paths between nodes to improve the performance of graph traversal queries.
An example has already been applied to the query "d.ii" where we have saved the "buying_friends" relationships and it will no longer be necessary to calculate the query again.
We also mentioned before that it is possible, for example to answer query "c", to memorize the "USED" relationship instead of continuously navigating through multiple transactions between a customer and a terminal.

The Property Graph Indexes Pattern would also be useful: it uses indexes on node properties to speed up search queries, for example the query “c” required us to retrieve a customer starting from his id, to improve the performance we could create indexes on the entity id.
In this way:

```
"""
CREATE INDEX customer_index IF NOT EXISTS
FOR (c:Customer) ON (c.customer_id)
"""
```

Similar to the previous one there is the Index Node Pattern. This pattern involves the creation of index nodes that act as pointers to specific nodes in the graph. It could be useful for queries in which we were forced to split them into smaller batches, indeed reducing the complexity of the nodes we might have been able to execute them in one shot, only at the end we would have recovered the original nodes.
This pattern is particularly useful if we are only interested in one property of a node and we want to exclude all the others, if they are many.

Finally there is also Computed Pattern that calculate relationships or node attributes in advance to reduce query times. as in the case of queries “a”, “b” and “e” where statistics were calculated.

All the patterns discussed above are not standard practices in neo4j, at least the names aren't, but are taken from other nosql system patterns and from graph theory with which Neo4j obviously has a lot in common.