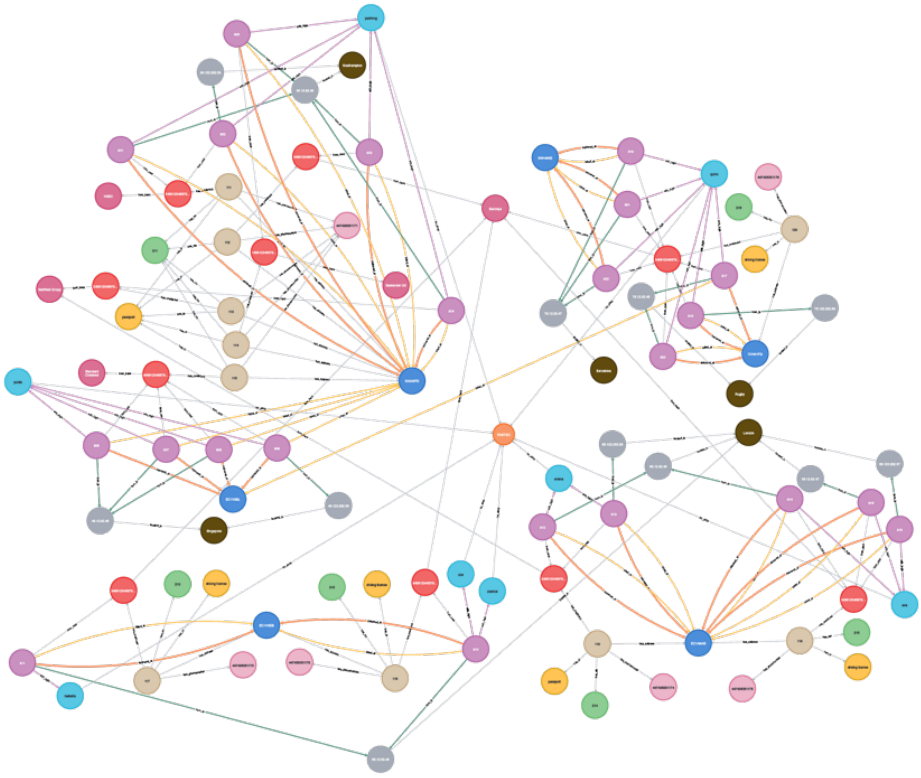
ST207 FRAUD DETECTION SOLUTION FOR E-COMMERCE WITH NEO4J



Candidate Numbers: 43737, 49139, 42583
Group 21

Table of Contents

1. Fraud Detection for E-Commerce

Our selected topic is fraud detection for e-commerce. In our scenario, a client (ecommerce company) has come to us looking to create a database which assists them in detecting fraud to their store. Lately, online businesses have become a more vulnerable target for fraudsters due to the lack of physical interaction between the legitimate cardholder and the business. The fraudsters will steal from the company in numerous ways. Below we introduce the most common ways fraudsters commit crimes against the ecommerce businesses and offer solutions to detect each one.

a. Card Testing Fraud:

In card testing fraud, a fraudster tries to determine if stolen card info can be used to make a purchase. They begin by making small purchases to test whether their transaction activity can go unnoticed in a short time frame. If their activities go unnoticed, they will proceed with the crime, completing it by making a large purchase with the stolen information.

b. Account takeover fraud:

In account take over fraud, the fraudster acquires the login credentials for a shopper's online account with the store. They complete the crime by using the saved credit card to make purchases.

Solution:

Our proposed solution for card testing and account takeover fraud is to execute a series of queries and observe patterns, as well as set a limit on the monetary size of a purchase. The queries we would carry out are the following:

- Find multiple transactions made in short amount of time
- Identify if the customers are making more than average spending.
- Identify if the customer is making transaction in unusual IP address located in different city.

c. Chargeback fraud:

Chargeback fraud is a crime that takes place between the fraudster and the victim's card issuer. A fraudster makes a claim to the credit card issuer that they did not make a certain transaction or that the transaction was invalid.

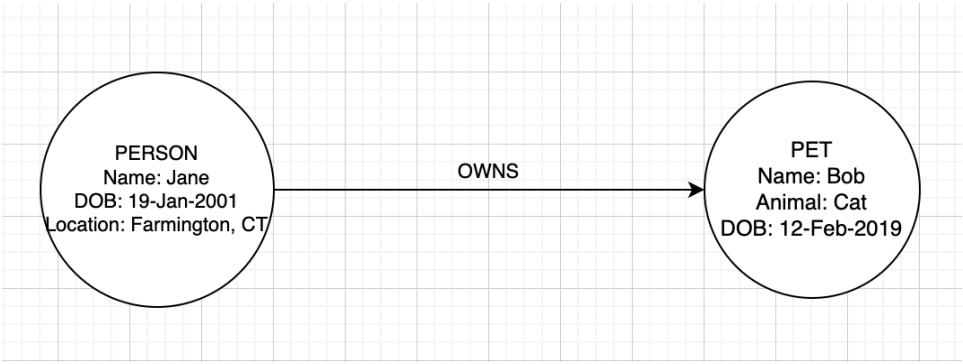
Solution:

To combat chargeback fraud, the credit card issuer can add an Address Verification Service (AVS) that allows ecommerce merchants to require that a person verify if the address provided for the bank card matches the billing address of the order.

2. Justification of Database Technology

a. Why Neo4j:

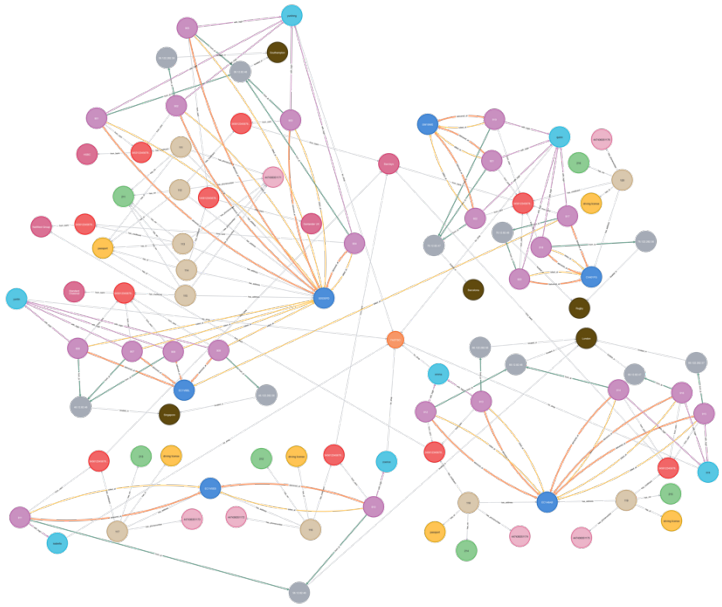
Neo4j is a graph database management system. By storing data in graph form, relationships between individuals can be saved in the database as well. This is useful in cutting down query times as well as visually representing the data. The system stores data in the following way: nodes represent individuals for whom various attributes have been stored, and edges represent relationships between individuals. For example, if we wanted to store the data that a person owns a pet in Neo4j, the nodes would be "Person" and "Pet" and the edge would be "owns". Below is an illustration:



As detailed in the description of our project, we will be relying on the relationships between data points, as well as carry out complex queries. If we were to use relational or document-based database management systems, it would be difficult to visually represent the relationships between specific instances of entities, affecting the pattern recognition needed in fraud detection. Additionally, in fraud detection, the fraudulent transactions should be caught as soon as possible. That being said, with relational or document-based databases, the data would not be represented in an efficient and useful way. Additionally, queries needed for fraud detection would be time consuming, labor intensive and highly susceptible to errors. Not only does Neo4j allow for quick querying and detection, but it also facilitates the application of essential machine learning models, such as K-Nearest Neighbors and PageRank, to predict future frauds. Although the application of machine learning models is not a focus of this project, it is a crucial idea that should be explored later.

b. Reproducing our database

To reproduce our database, first download the Neo4j server for desktop (available for MacOSX, Linux and Windows). Then, create an account with Neo4j on the Neo4j website. Finally, open the Neo4j desktop application and create a new project by clicking on the "+ New" button next to the Projects. To load our database into your newly created Neo4j project, select "+Add" and select"Local DBMS" from the dropdown menu. Then, name the database and enter password. Next click "Create" and "Open" to open with Neo4j Browser. Next, copy all the text from ST207_Database_Creation.txt and paste in Neo4j Browser. To check whether the database file was properly imported, run the code MATCH (n) RETURN n and observe the output. The output should look like the following:



Node labels

Relationship types

3. Description of Data
    a. Database Modeling – Mini World Description

Consider an ecommerce database in which merchants and banks have details on customers and transactions. The data requirements are as follows:

- Each account_holder is identified by a unique identifier, first name, last name, and birthdate. They also have a unique nin assigned to them, an address, a unique phone number, an identification, and credit cards. Identification information on the account_holder includes the type of identification document, id number, issue date and expiry date.

- Each financial_institution is identified by its name and is the sole issuer of a credit card. One financial_institution can only issue a credit card, however an account_holder may have several credit cards. Credit cards are identified using a unique identifier, expiration date, security code, balance, balance date and limit.

- An account_holder may have an account with the ecommerce shop, which has a username, password, email address and unique identifier. The account_holder's logins to their ecommerce account are recorded.

- Transactions made by account_holders are recorded by a unique transaction id, transaction date and time and transaction amount. Additionally, the IP address and its location are recorded with the transaction.

- The shipping and delivery address of each transaction is recorded.

- The ecommerce shop is represented by a unique shop id and the shop's name.

b. Conceptual Description of Database

i. <entity; attributes> of the conceptual model:

- <account_holder; birth_date, first_name, last_name, unique_id>
- <nin; nin>
- <phone; phone_number, provider>
- <address; add_id, flat _number, flat_name, city, postcode, country>
- <identification; issued_date,expiration_date, id_number, type>
- <financial_institution; name, bank_id>
- <credit_card; card_number, expiration_date, security_code, balance, balance_date, limit>
- <ip_address; ip, ipid>
- <transaction; transaction_id, trans_datetime, transaction_amount, >
- <city; city, city_id>
- <login; login_id, user_name, email_address, password>
- <shop; shop_id, shop_name>

ii. Relationships between entities and justifications:

- has_nin
  It is an outgoing relationship from the account_holder to nin entities. Each account could be opened with his or her unique nin. Each unique nin can open multiple accounts.

- has_phonenumber
  It is an outgoing relationship from the account_holder to phone entities. A phone number can only belong to an account holder. An account holder can have multiple phone numbers.

- has_address
  It is an outgoing relationship from the account_holder to address entities. account_holder might share the same addresss as they live together.

- has_id
  It is an outgoing relationship from the account_holder to identification entities. An person can open an account with an identification document. An identification document only belongs to a person.

- has_creditcard
  It is an outgoing relationship from the account_holder to credit_card entities. An account holder can have multiple credit cards, but each credit card belongs to only an account holder (ie. Account_holder Jane can have credit cards from Lloyds Bank, Santander, and NatWest, but each of her credit cards belongs to her only).

- from_bank
  It is an outgoing relationship from the credit_card to financial_institution entities.A credit_card must be issued by one bank only, however a financial_institution may issue several unique credit_cards.

- from_card
  It is an outgoing relationship from the transaction to credit_card entities. A credit card can have several transactions. However, a transaction cannot be paid by multiple cards.

- from_ip
  It is an outgoing relationship from the transaction to ip_address entities.It indicates where the transaction was made, which is useful in identifying anomalies in purchasing activity. A transaction can occur at an IP address.Multiple transactions can occur at only an IP address.

- located_in
  It is an outgoing relationship from the ip_address to city entities. It tells the database user where the IP address making the purchase is located. This helps in identifying purchasing patterns that seem otherwise impossible. Multiple IP addresses can be from a city.

- billed_at
  It is an outgoing relationship from the transactions to address entities. It indicates which physical location the transaction's purchase would be billed to. A transaction can only have a billing address whereas multiple transactions can occur at a billing address.

- delivered_at
  It is an outgoing relationship from the transactions to address entities. It indicates which physical location the transaction's purchase would be delivered to. A transaction can only have a billing address whereas multiple transactions can occur at a billing address.

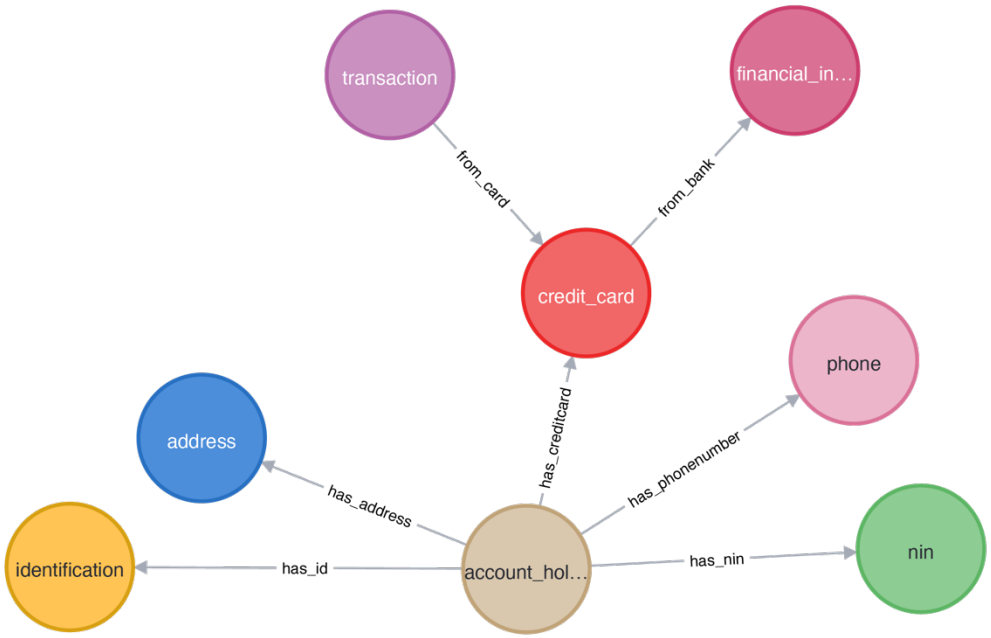
- with_login
  It is an outgoing relationship from the transactions to login entities. It indicates which customer account was used to make a specific purchase. A specific transaction could only made by an account whereas an account could make multiple transactions.

- for_shop
  It is an outgoing relationship from the login to e-commerce shop entities. It indicates which website the internet account is for. The internet account can access to multiple e-commerce shop entities. However, in this miniworld, there is only an e commerce shop entity. An e-commerce shop entity allows access to multiple internet accounts.

c. Database Creation - Ideation

During the creation of the database, data requirements of the ecommerce store were considered in 2 main parts: Subgraph A and Subgraph B. Subgraph A contains information about credit card account creation and Subgraph B contains information pertinent to online transactions from a specific e-commerce shop. Using this logic, we created the entities relating to credit cards and internet account creation. We then built on the customer-specific entities, creating transaction and internet purchasing related entities that relate to the ecommerce shop. The following diagrams demonstrate the conceptual parts of the database:

i. Subgraph A: Account holder information (Example user: Yun Lin Ng)

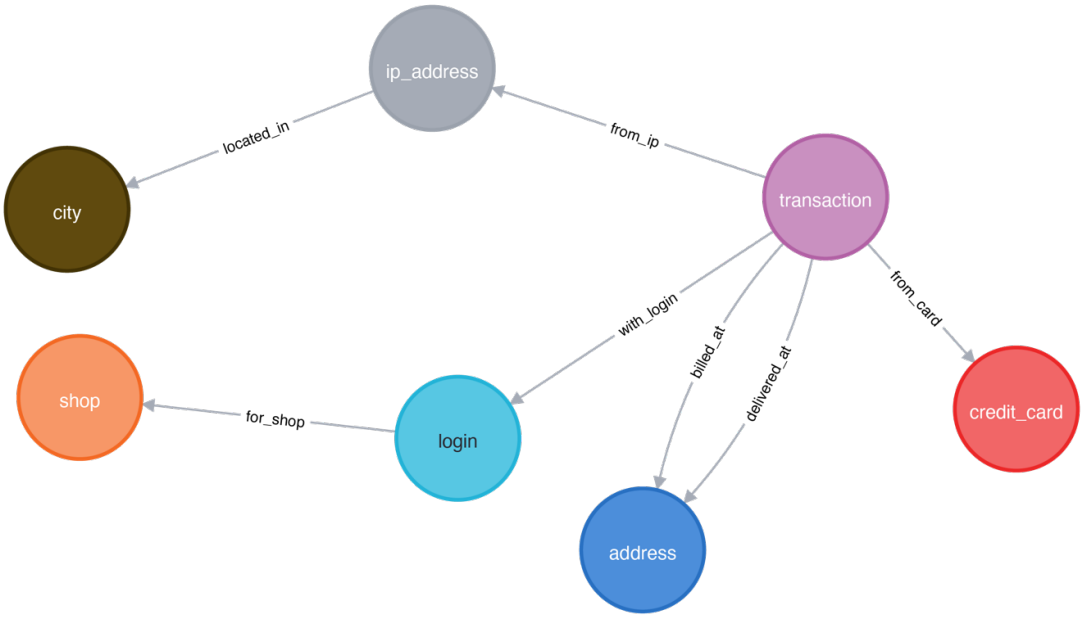


Node labels

Relationship types

- This example use case of the account holder subgraph tells database users that the person named Yun Ling Ng has National Insurance Number (NIN) of 211, identification number 11, address at flat name: Berry Court, credit card number card_number: 5453123456789101 issued by HSBC which is used to make a specific transaction, and phone number 447436351171 with Virgin Media.

ii. Subgraph B: Ecommerce Data



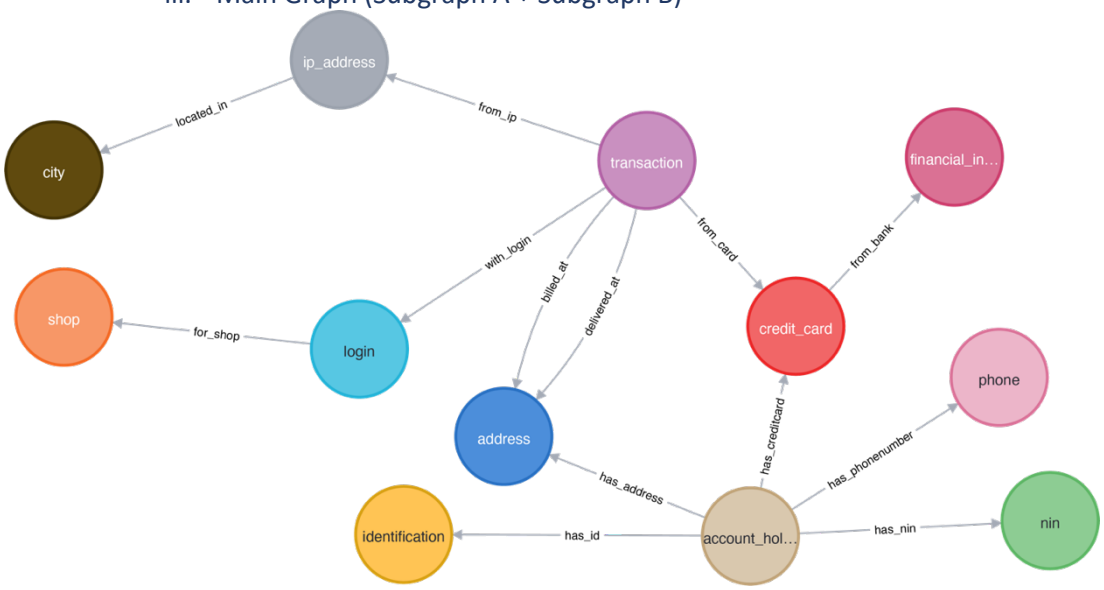
Node labels

Relationship types

- This subgraph tells database users that a specific purchase (in purple) was made with credit card number 5453123456789101, billed to and delivered to the same address, from IP address 36.12.82.46 which is located in Southampton using login account username yunlinng for the FASTSO ecommerce shop.

By typing “call db.schema.visualization” in Neo4j Browser, the following diagram as follows is generated. The schema of the main graph shows what entities exist in the database and how they are related, generally.

iii. Main Graph (Subgraph A + Subgraph B)



Node labels

Relationship types

In the final database graph, which follows the above schema, there are 97 nodes and 189 relationships.

iv. Data formulation thought process

4 cases are used to demonstrate how to build a fraud detection solution with Neo4j.

- Case 1: User Yun Lin Ng opens 5 credit card accounts, however account 115 is hijacked by a fraudster. With the fraudulent credit card, the criminal tests the card in Singapore with an IP address located in Singapore.

- Cases 2: Two people live together, meaning they share the same billing and delivery address. This also means that two accounts with the shopping website FASTSO are able to make transactions from the same IP address.

- Case 3: Two people live together, so they share the same billing and delivery address. The difference between this case and Case 2 is that this household has 3 IP addresses associated with it: the home WiFi IP address, one of the resident's work IP address and a cellular data IP address for the other resident. Transactions made from different, recognized IP addresses would not be frauds since there are 3 known and associated IP addresses with the physical address.

- Case 4: User Quinn Sanderson's account gets hacked, so a fraudster is able to make a transaction in Barcelona (identified by the IP address used to make the transaction) within minutes of Quinn making a transaction in Rugby, UK. Quinn lives alone and she has not shared her login information with anyone else (i.e. a friend or relative).

d. Database Creation: Implementation

i. Coding Technicalities

When coding the database in Neo4j, we began by creating account_owners and nin .nodes. We then create a relationship has_nin from the account_user to nin nodes. Next, we built the phone_numbers nodes and created relationships from the account_holders to the phone_numbers. A note about relationships in
To create the entities in Neo4j, use the following example:

```
CREATE (a:account_holder{birth_date:"2002-01-07", first_name:"Isabella",last_name:"Obrien",
unique_id:117})
```

The above code tells Neo4j to create a node (instance of an entity) named a of entity account_holder with the following values for the attributes of an account holder:

- *birth_date: "2002-01-07" (as String datatype)*
- *first_name: "Isabella" (as String datatype)*
- *last_name: "Obrien" (as String datatype)*
- *unique_id: 117 (as integer datatype)*

Creating relationships between entities in Neo4j uses the following codes:

```
MATCH (a:account_holder{unique_id:111}),(n:nin{nin:211})
MERGE (a)-[r:has_nin]->(n)
```

The above code tells Neo4j to:

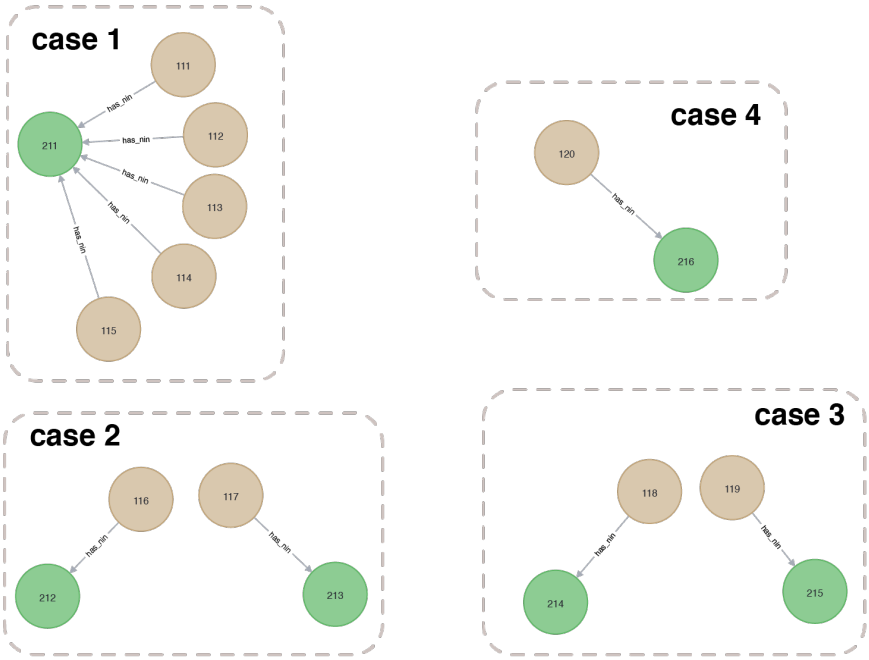
1. *Extract the account holder with unique id 111 and the national insurance number 211, then*
2. *create a directional relationship has_nin from account holder 111 and nin 211*
3. *MERGE instead of CREATE is used to prevent duplicated relationships*

This process was repeated for all entities and all relationships in our database.

ii. Building the database from scratch

When coding the database (i.e. coding it in Neo4j), we built it in the following order[1]:

1. account_holder
2. nin
    a. has_nin (relationship)

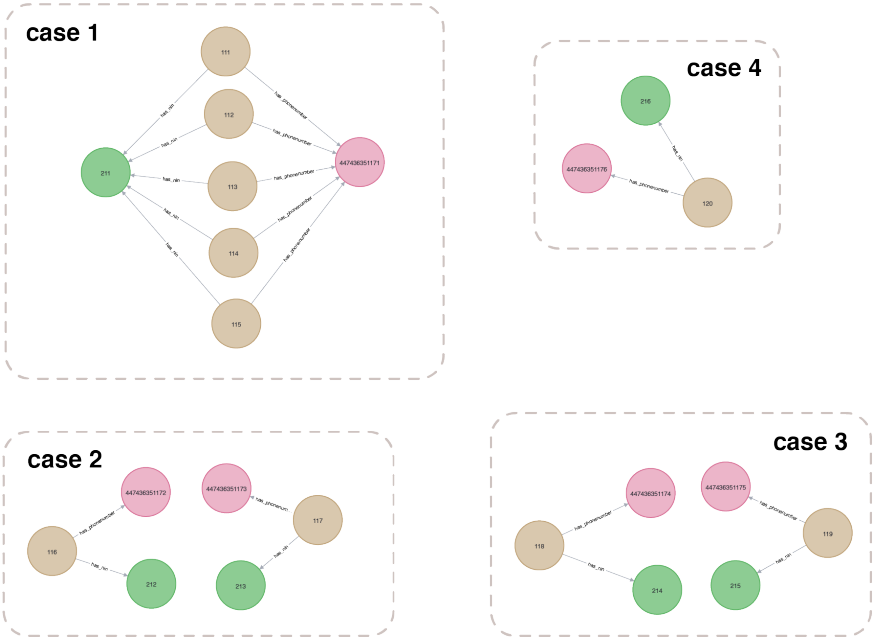


Node labels

Relationship types

[1] Each step with a relationship creation has an image of the database at that stage (i.e. step 2a has an image of the account_holder and nin entities with the relationship has_nin)

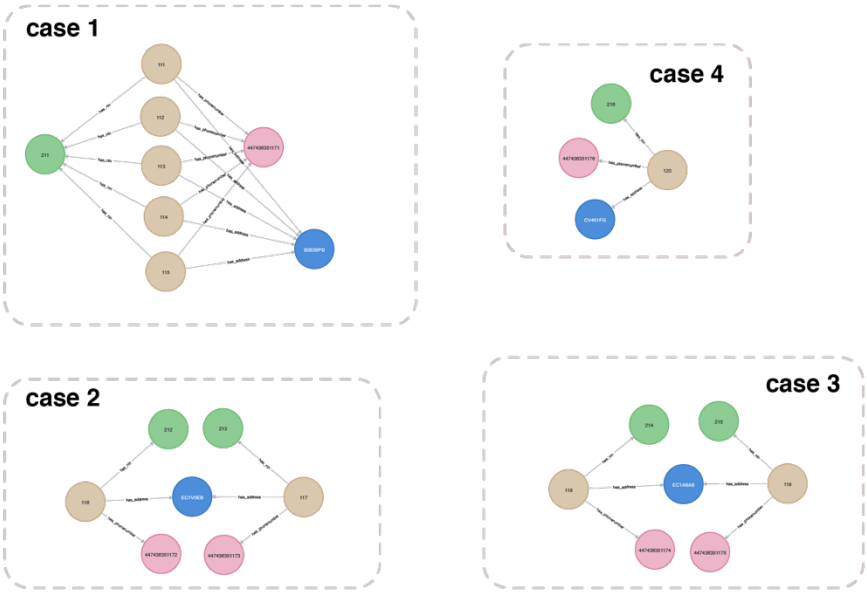
3. phone
    a. has_phonenumber (relationship)



Node labels

Relationship types

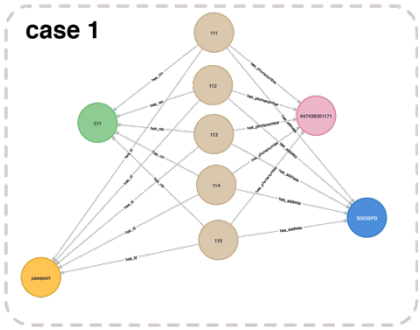
4. address
    a. has_address (relationship)

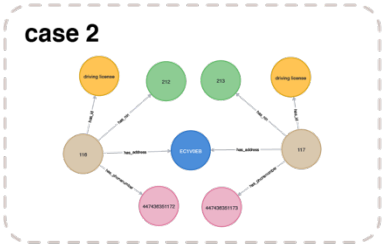


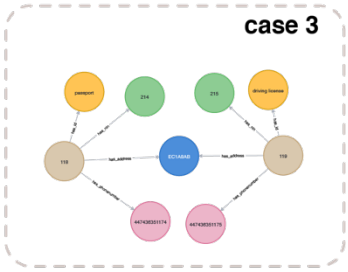
Node labels

Relationship types

5. identification
    a. has_id (relationship)



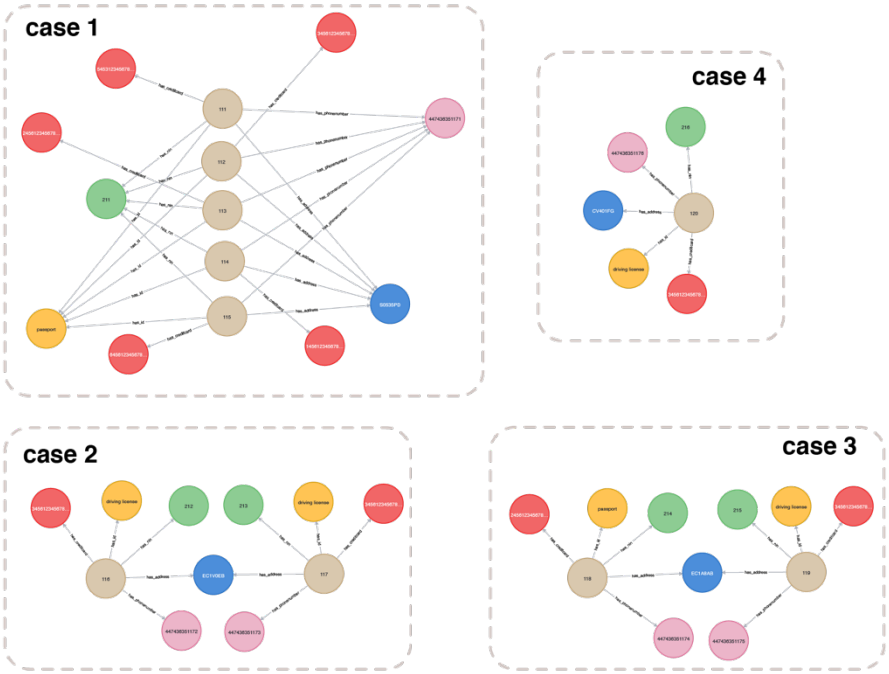




Node labels

Relationship types

6. credit_card
    a. has_creditcard (relationship)

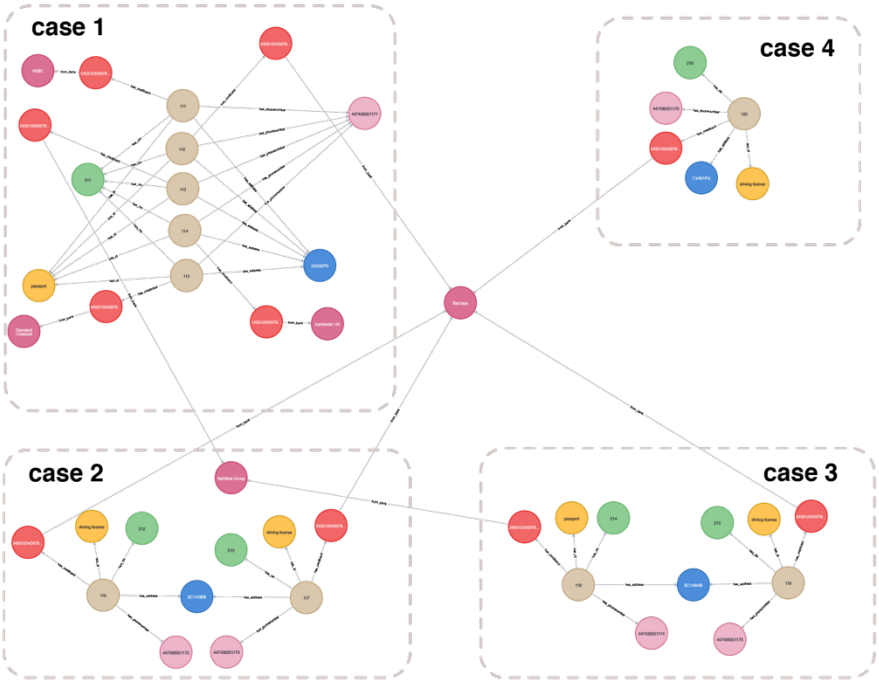


Node labels

Relationship types

7. financial_institution

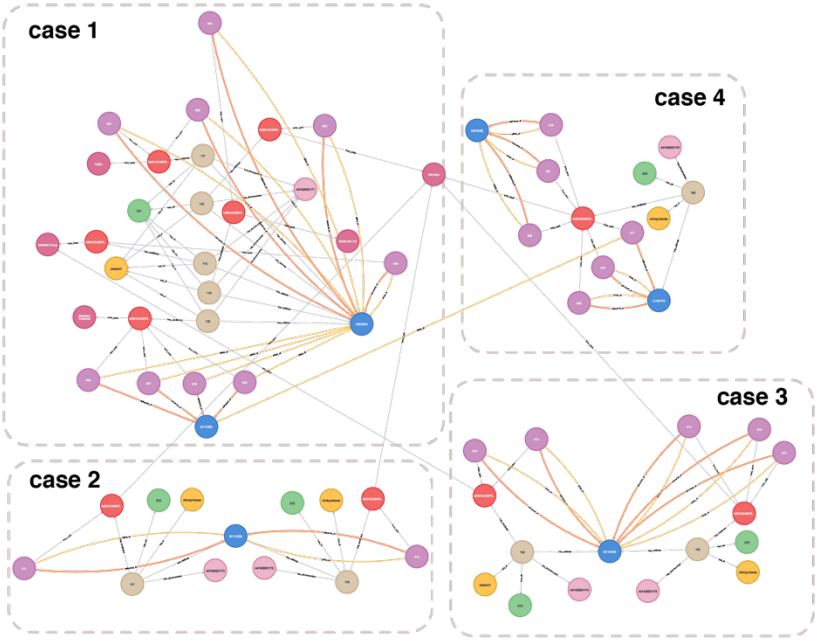
   a. from_bank (relationship)



Node labels

Relationship types

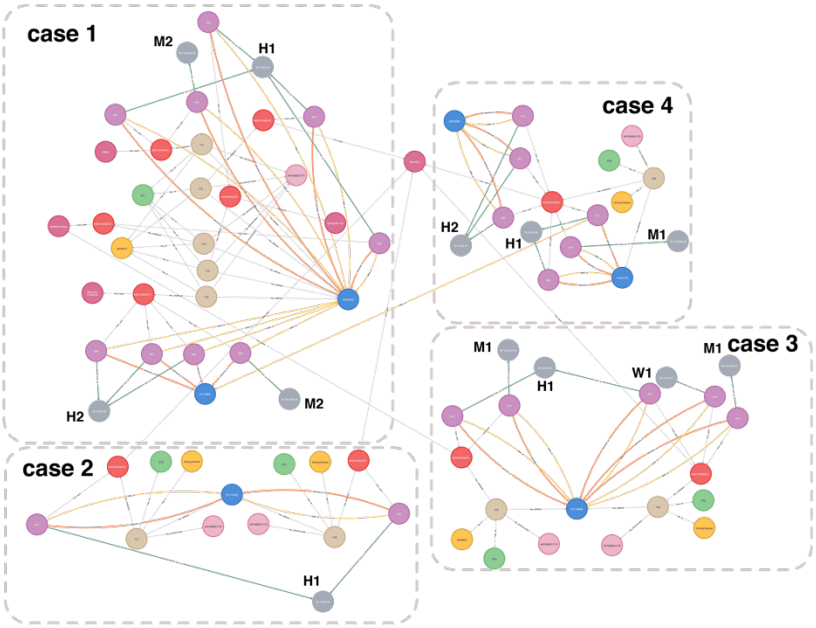
8. transaction
    a. from_card (relationship)
    b. delivered_at (relationship)
    c. billed_at (relationship)



Node labels

Relationship types

9. ip_address
    a. from_ip (relationship)

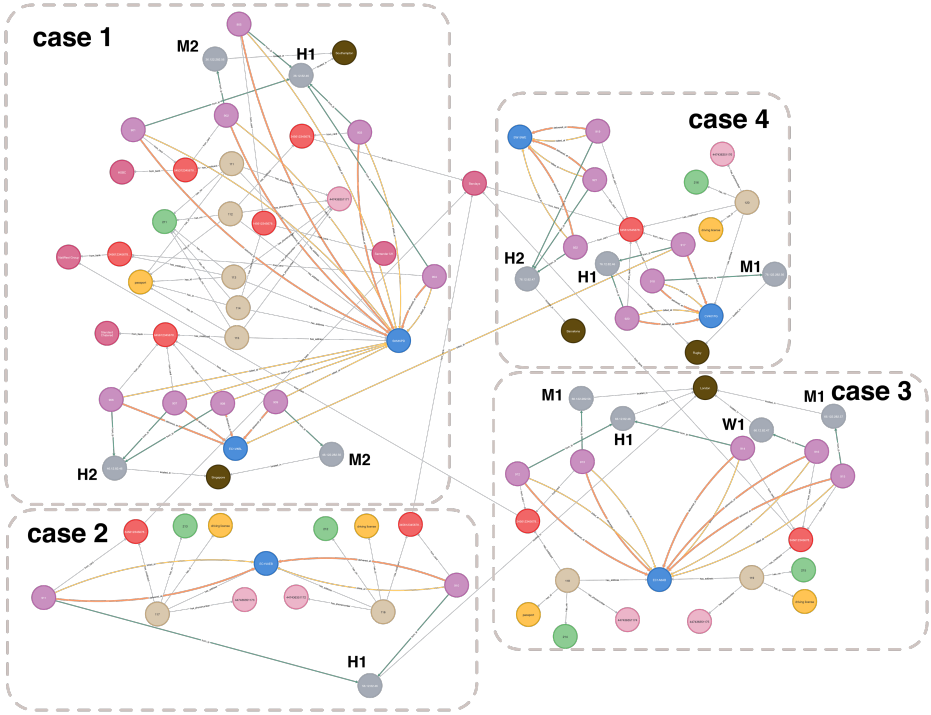


Node labels

Relationship types

H1: Home Wi-Fi 1
H2: Home Wi-Fi 1
M1: Mobile data 1
M2: Mobile data 2
W1: Work Wi-Fi 1

10. city
    a. located_in



Node labels

Relationship types

11. shop

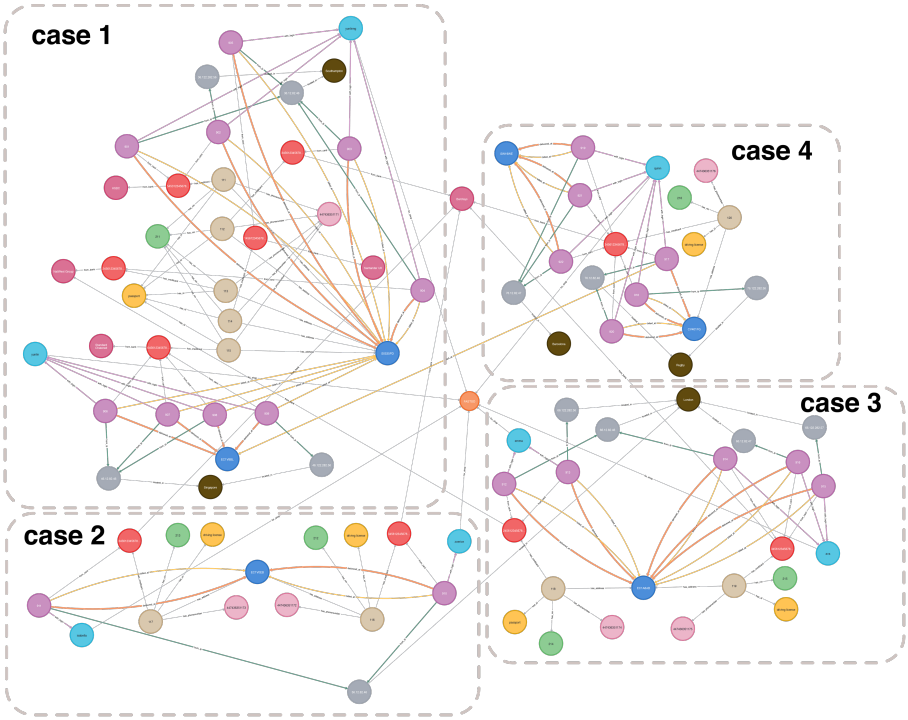
Node labels

account_holder (10) nin (6) phone (6) address (6) identification (6) financial_institution (5)
credit_card (10) ip_address (12) transaction (22) city (5) login (8) shop (1)

Relationship types

has_creditcard (10) has_id (10) has_address (10) has_phonenumber (10) has_nin (10)
from_bank (10) located_in (12) with_login (23) delivered_at (22) billed_at (22) from_ip (22)
from_card (22) for_shop (8)

12. login
    a. with_login (relationship from transaction to login)
    b. for_shop (relationship from login to shop)



Node labels

Relationship types

This diagram is equivalent to the result by typing *MATCH (n) RETURN n*

iii. Adding constraints before creating nodes and relationships

1. constraint for account_holder node

- unique_id is unique

```
CREATE CONSTRAINT accountholder_unique IF NOT EXISTS
FOR (a:account_holder) REQUIRE a.unique_id IS UNIQUE;
```

2. constraint for national insurance number node

- nin is unique

```
CREATE CONSTRAINT nin_unique IF NOT EXISTS
FOR (n:nin) REQUIRE n.nin IS UNIQUE;
```

3. constraint for phone node

- phone_number is unique

```
CREATE CONSTRAINT phone_unique IF NOT EXISTS
FOR (p:phone) REQUIRE p.phone_number IS UNIQUE;
```

4. constraint 1 for address

- add_id is unique

```
CREATE CONSTRAINT addid_unique IF NOT EXISTS
FOR (ad:address)
REQUIRE (ad.add_id) IS UNIQUE;
```

5. constraint 2 for address

- combination of ad.flat_number, ad.flat_name, ad.city, ad.postcode, ad.country is unique

```
CREATE CONSTRAINT address_unique IF NOT EXISTS
FOR (ad:address)
REQUIRE (ad.flat_number, ad.flat_name, ad.city, ad.postcode, ad.country) IS UNIQUE;
```

6. constraint for identification

- id_numberis unique

```
CREATE CONSTRAINT id_unique IF NOT EXISTS
FOR (id:identification) REQUIRE id.id_number IS UNIQUE;
```

7. constraint 1 for financial institution

- bank_id is unique

```
CREATE CONSTRAINT institutionid_unique IF NOT EXISTS
FOR (fi:financial_institution) REQUIRE fi.bank_id IS UNIQUE;
```

8. **constraint 2 for financial institution**

- name is unique

```
CREATE CONSTRAINT institutionname_unique IF NOT EXISTS
FOR (fi:financial_institution) REQUIRE fi.name IS UNIQUE;
```

9. **constraint for credit card**

- card_number is unique

```
CREATE CONSTRAINT creditcard_unique IF NOT EXISTS
FOR (cc:credit_card) REQUIRE cc.card_number IS UNIQUE;
```

10. **constraint 1 for IP address**

- ipid is unique

```
CREATE CONSTRAINT ipid_unique IF NOT EXISTS
FOR (ip:ip_address) REQUIRE ip.ipid IS UNIQUE;
```

11. **constraint 2 for IP address**

- ip is unique

```
CREATE CONSTRAINT ip_unique IF NOT EXISTS
FOR (ip:ip_address) REQUIRE ip.ip IS UNIQUE;
```

12. **constraint for transaction**

- transaction_id is unique

```
CREATE CONSTRAINT transaction_unique IF NOT EXISTS
FOR (t:transaction) REQUIRE t.transaction_id IS UNIQUE;
```

13. **constraint 1 for city**

- city_id is unique

```
CREATE CONSTRAINT city_unique IF NOT EXISTS
FOR (c:city) REQUIRE c.city_id IS UNIQUE;
```

14. **constraint 2 for city**

- city is unique

```
CREATE CONSTRAINT cityname_unique IF NOT EXISTS
FOR (c:city) REQUIRE c.city IS UNIQUE;
```

15. **constraint 1 for login**

- login_id is unique

```
CREATE CONSTRAINT loginid_unique IF NOT EXISTS
FOR (l:login)
```

```
REQUIRE l.login_id IS UNIQUE;
```

16. constraint 2 for login

- user_name is unique

```
CREATE CONSTRAINT username_unique IF NOT EXISTS
FOR (l:login)
REQUIRE l.user_name IS UNIQUE;
```

17. constraint 3 for login

- email_address is unique

```
CREATE CONSTRAINT email_unique IF NOT EXISTS
FOR (l:login)
REQUIRE l.email_address IS UNIQUE;
```

18. constraint for shop

- shop_id is unique

```
CREATE CONSTRAINT shopid_unique IF NOT EXISTS
FOR (s:shop) REQUIRE s.shop_id IS UNIQUE;
```

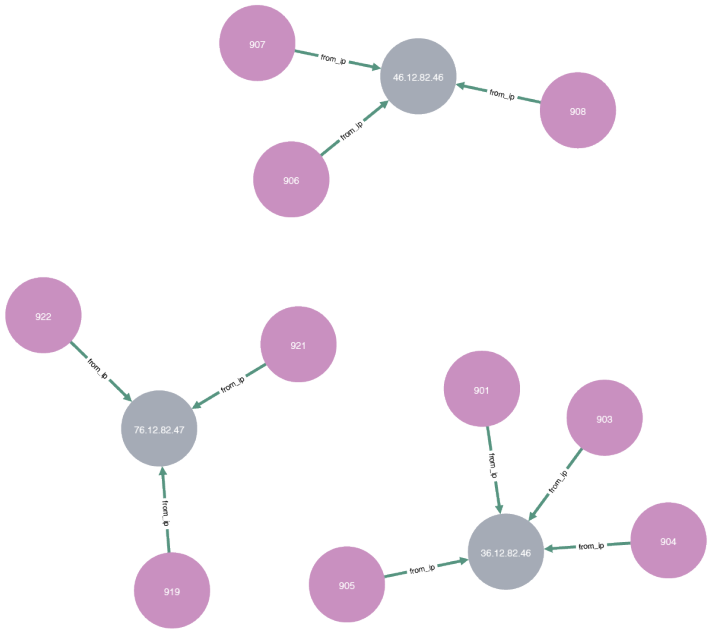
4. Queries and Update Operations

The following queries and update operations are executed to demonstrate fraud detection solution with Neo4j.

a. Find top 3 Internet Protocol (IP) addresses by number of transactions

If an IP address is making a lot of transactions, it could be an indicator of fraud. Fraud rings could hide behind an IP address which went through Network Address Transation (NAT) to commit fraud. However, it is not a clear and definite indicator of fraud as it might be a household where there are many people using the home Wi-Fi to make transactions. In this case, IP address 46.12.82.46 and 76.12.82.47 are the IP addresses of Case 1 and Case 4 respectively, which demonstrates e-commerce fraud but IP address ip: 36.12.82.46 is from Case 2 which has no fraud.

```
MATCH (t:transaction)-[:from_ip]->(ip:ip_address)
WITH ip, COUNT(t) as no_of_transaction
ORDER BY no_of_transaction DESC
LIMIT 3
MATCH path = (t:transaction)-[:from_ip]->(ip:ip_address)
RETURN path;
```



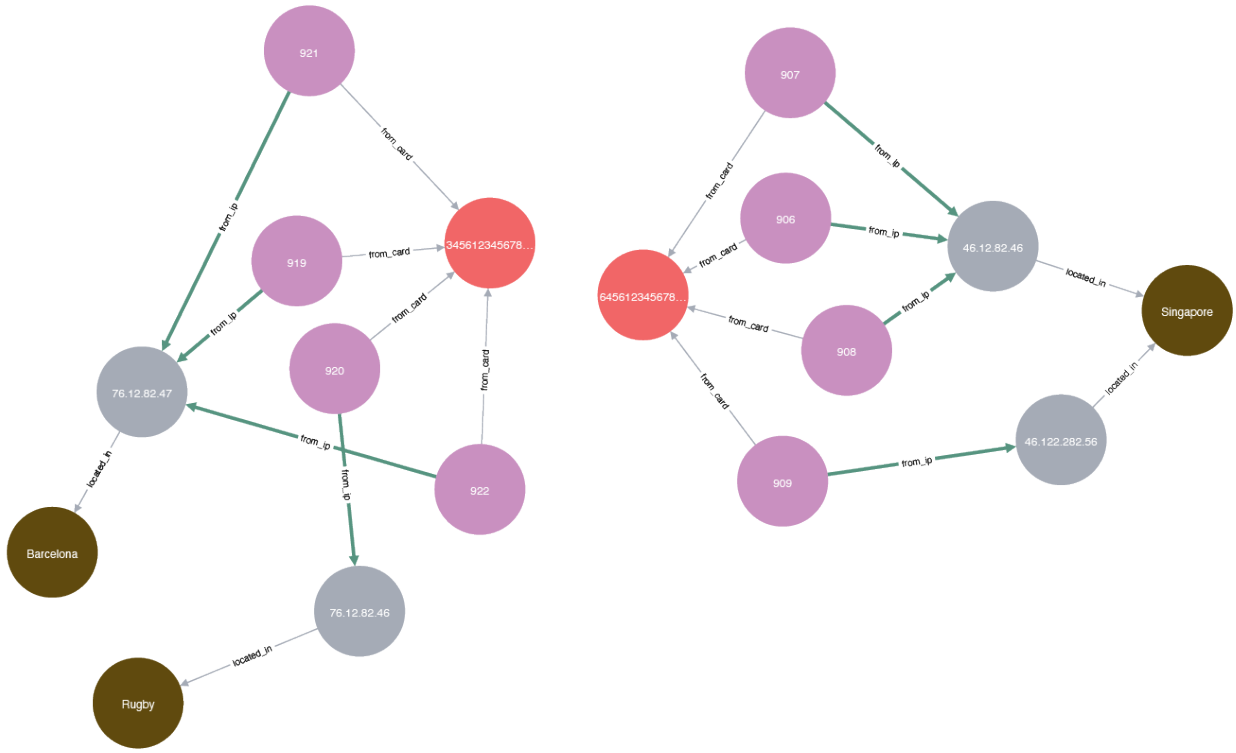
[Diagram D1: top 3 Internet Protocol (IP) addresses by number of transactions]

b. Identify multiple transactions from the same card within 5 minutes (300 seconds) and show their respective IP address and located cities.

Index is used to compute the difference between the previous transaction time and the current transaction time. The result of the query which filters the transactions within a transaction time is stored in *transactionRange*, which acts like a view. *transactionRange* is then used in *MATCH* clause for the *credit_card, ip_address* and city nodes as well as *from_card, from_ip* and *located_in* relationships.

```
MATCH (t:transaction)-[:from_card]->(cc:credit_card)
WITH t, cc
ORDER BY t.transaction_datetime
WITH cc, collect(t) as transaction, collect(t.transaction_datetime) as date
UNWIND range(0, size(transaction)-2) as index
WITH cc, transaction, date, index
WHERE duration.inSeconds(date[index], date[index+1]).seconds <= 300
UNWIND [transaction[index], transaction[index + 1]] as transactionRange
MATCH path = (cc) <- [:from_card]-(transactionRange)-[:from_ip]-> (ip:ip_address)-
[li:located_in]->(c:city)
```

RETURN path



[Diagram D2: Multiple transactions from the same card within 5 minutes and their respective IP address and cities]

[Diagram D2] The node properties of transaction for the cluster on the left are as follows:

<id>: 79
transaction_amount: 5
transaction_datetime: "2022-11-21T07:01:32.142000000Z"
transaction_id: 919

<id>: 80
transaction_amount: 30
transaction_datetime: "2022-11-21T07:03:32.142000000Z"
transaction_id: 920

<id>: 81
transaction_amount: 10
transaction_datetime: "2022-11-21T07:03:32.142000000Z"
transaction_id: 921

<id>: 82
transaction_amount: 2000
transaction_datetime: "2022-11-21T07:08:32.142000000Z"
transaction_id: 922

[Diagram D2] The node properties of credit card for the cluster on the left are as follows:

<id>: 48
balance: 20000
balance_date: "2022-12-01T23:08:32.142000000Z"
card_number: 3456123456789104
expiration_date: "2025-04-01"
limit: 2000
security_code: 804

It seems like it is a card testing fraud. Fraudster might be testing the card with transaction amount of £5, £30, £10 and lastly £2000, which the limit of the transaction amount of the card in a short amount of time. To further illustrate the occurrece of card testing fraud, the transaction with transaction_id 919 occurred on 2022-11-21 at 07:01:32am in Barcelona, transaction with transaction_id 920 on 2022-11-21 07:03:32am in Rugby and transaction with transaction_id 921 on 2022-11-21 07:03:32 am in Barcelona.

[Diagram D2] The node properties of transaction for the cluster on the right are as follows:

<id>: 66
transaction_amount: 5
transaction_datetime: "2022-08-12T10:11:32.142000000Z"
transaction_id: 906

<id>: 67
transaction_amount: 5
transaction_datetime: "2022-08-12T10:13:32.142000000Z"
transaction_id: 907

<id>: 68
transaction_amount: 5
transaction_datetime: "2022-08-12T10:15:32.142000000Z"
transaction_id: 908

<id>: 69
transaction_amount: 5000
transaction_datetime: "2022-08-12T10:17:32.142000000Z"
transaction_id: 909

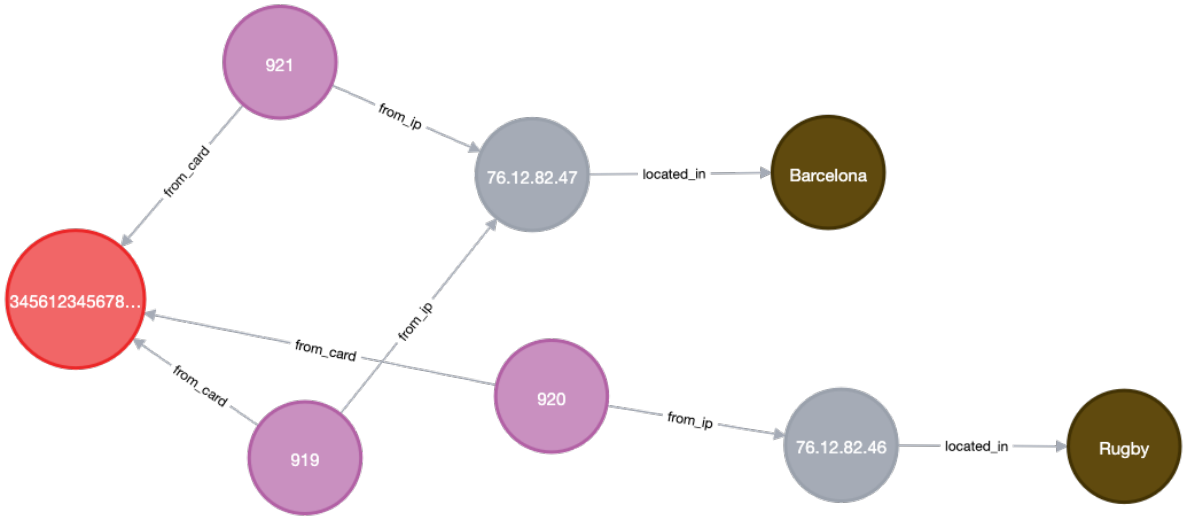
It seems like it is a card testing fraud. Fraudster might be testing the card with transaction amount of £5, £5, £5 and lastly £5000, which the limit of the transaction amount of the card in a short time.

c. Identify multiple transactions from the same card within 5 minutes (300 seconds) from different IP addresses located in different cities.

Multiple transactions within a short time frame of fraud. Multiple transactions within a short time frame with different IP addresses located in different cities would be a stronger indicator of fraud.

This query is similar to query in D2, with addition of IP address and city of the current transaction not equal to the previous transaction. The output is the cluster on the left in Diagram D2 without transaction with transaction_id 922.

```
MATCH (c:city)<-[:located_in]-(ip:ip_address)<-[:from_ip]-(t:transaction)-[:from_card]->(cc:credit_card)
WITH t, cc, ip, c
ORDER BY t.transaction_datetime
WITH cc, collect(t) as transaction, collect(t.transaction_datetime) as date,
collect(ip) as ipp, collect(c) as cityy
UNWIND range(0, size(transaction)-2) as index
WITH cc, transaction, date, ipp, cityy, index
WHERE duration.inSeconds(date[index], date[index+1]).seconds <= 300
AND ipp[index] <> ipp[index+1]
AND cityy[index] <> cityy[index+1]
WITH cc,[[transaction[index], transaction[index + 1]], [ipp[index], ipp[index + 1]],[cityy[index],
cityy[index + 1]]] AS nested
UNWIND nested as x
UNWIND x AS y
RETURN cc, y
```



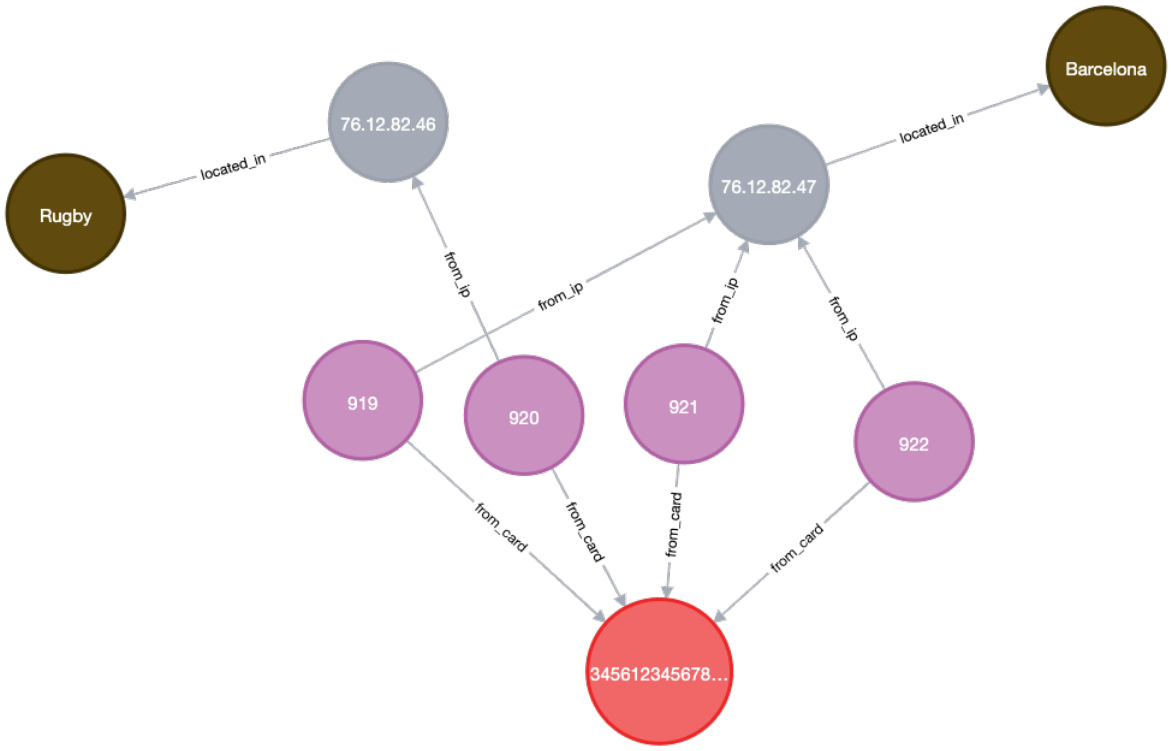
[Diagram D3: Multiple transactions from the same card within 5 minutes from different IP addresses located in different cities]

d. Recognize subsequent transaction that took place after the last transaction in D4 and cluster them with the previous transactions.

Since the result in D3 is a strong indicator of fraud, it is important to recognize the subsequent transactions as it is of high risk that the subsequent purchases are fraudulent.

The query here is similar to query in D3, with addition of OPTIONAL MATCH clause. OPTIONAL MATCH clause is used as there might not be subsequent transactions after transactions identified in D3. OPTIONAL MATCH could be considered Cypher equivalent of the outer join in SQL based on the documentation on Neo4j Website.

```
MATCH (c:city)<-[:located_in]-(ip:ip_address)<-[:from_ip]-(t:transaction)-[:from_card]->(cc:credit_card)
WITH t, cc, ip, c
ORDER BY t.transaction_datetime
WITH cc, collect(t) as transaction, collect(t.transaction_datetime) as date,
collect(ip) as ipp, collect(c) as cityy
UNWIND range(0, size(transaction)-2) as index
WITH cc, transaction, date, ipp, cityy, index
WHERE duration.inSeconds(date[index], date[index+1]).seconds <= 1800
AND ipp[index] <> ipp[index+1]
AND cityy[index] <> cityy[index+1]
WITH cc,[[transaction[index], transaction[index + 1]], [ipp[index], ipp[index + 1]],[cityy[index],
cityy[index + 1]]] AS nested
UNWIND nested as x
UNWIND x AS y
OPTIONAL MATCH (cc)<- [:from_card]-(t)
WHERE t.transaction_datetime > y.transaction_datetime
RETURN cc,t,y
```

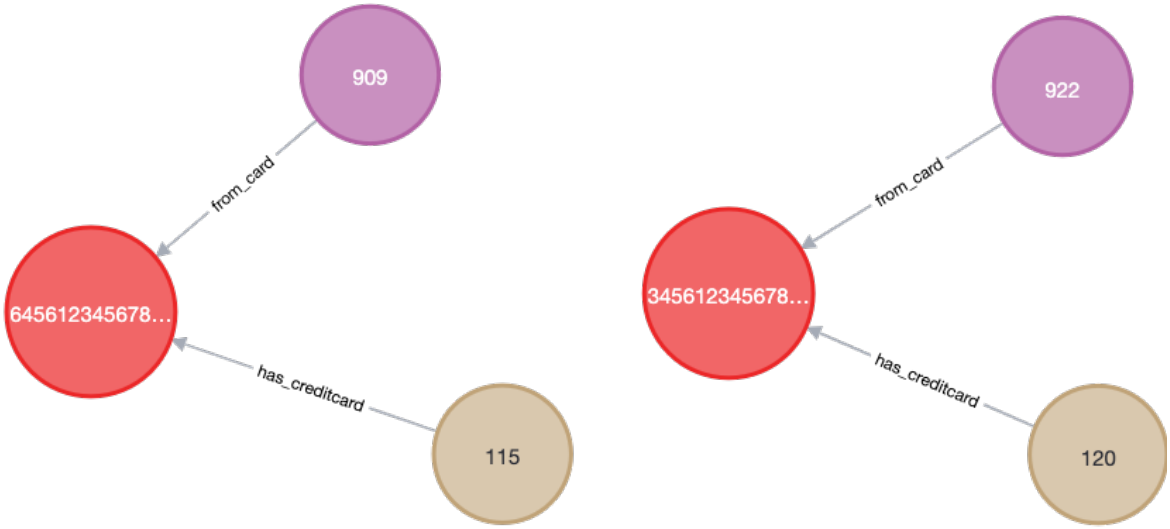


[Diagram D4: Identify multiple transactions from the same card within 5 minutes from different IP addresses located in different cities and the subsequent transactions]

e. Identify transactions, account holder and card used for the transaction when transaction amount is beyond 2 times of average transaction amount from each card

From D3, the query result demonstrated that transaction with transaction_id 909 and 922 has transacted the maximum amount of the card limit, which might be unusual.
The following query identifies transactions with this peculiar behavior.

```
MATCH (a:account_holder)-[:has_creditcard]->(cc:credit_card)<-[:from_card]-(t:transaction)
WITH cc, AVG(t.transaction_amount) as avg_amount, a
OPTIONAL MATCH path =(a:account_holder)-[:has_creditcard]->(cc:credit_card)<-[:from_card]-
(t2:transaction)
WHERE t2.transaction_amount >= 2*avg_amount
RETURN path
```

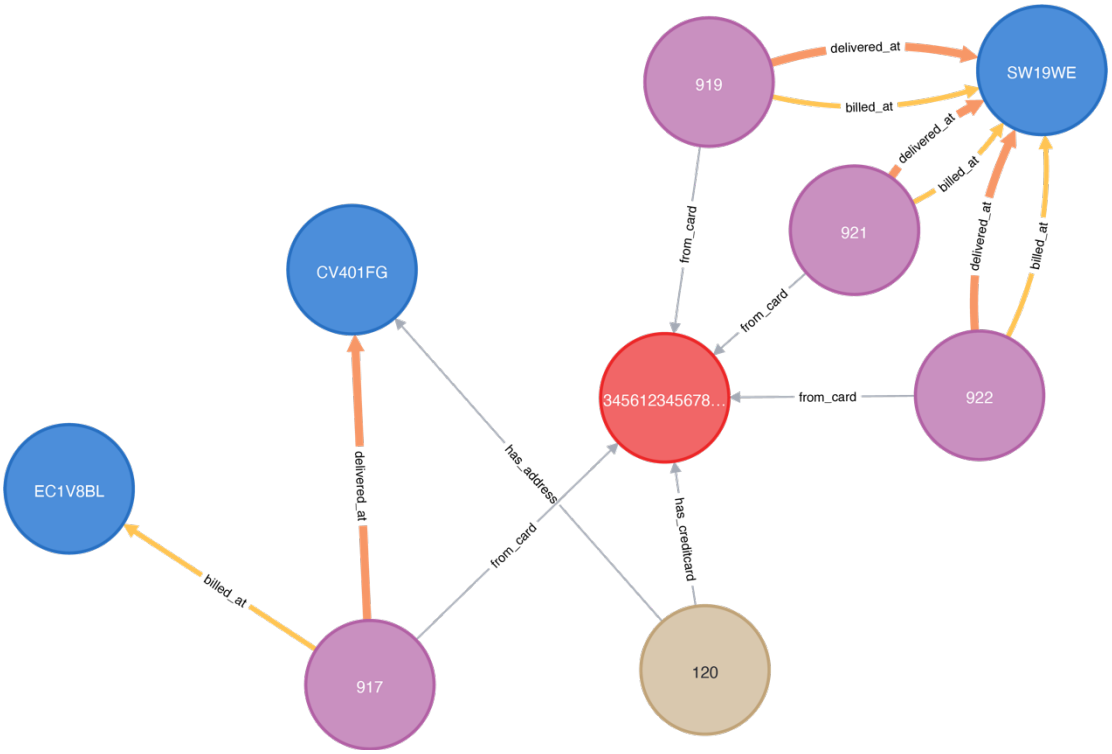


[Diagram D5: Identify transactions, account holder and card used for the transaction when transaction amount is beyond 2 times of average transaction amount from each card]

f. Perform an Address Verification Service (AVS) to find mismatch of customers' addresses and billing address of the transaction

If the billing address of the transaction does not match the address of the customer provided to the bank, it might not be a legitimate transaction.

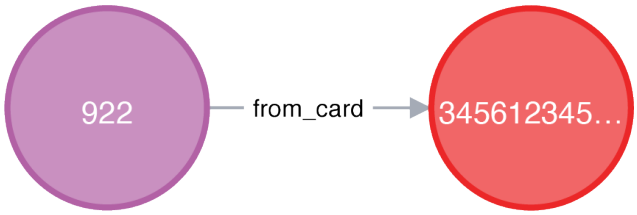
MATCH (original_add:address) <- [has:has_address] -(a:account_holder) - [c:has_creditcard] ->
(cc:credit_card) <- [fc:from_card] - (t:transaction) - [b:billed_at] - (billed_address:address)
WHERE original_add <> billed_address
RETURN a,c,cc,fc,t,b,original_add,has,billed_address;



g. Update balance of credit card after deduction of transaction amount from latest transaction and update date to the latest transaction date

Find the last transaction ID and card used for the transaction

MATCH (t0:transaction)
WITH t0
ORDER BY t0.transaction_id DESC
LIMIT 1
WITH t0
OPTIONAL MATCH (t0:transaction)-[:from_card]->(cc:credit_card)
RETURN t0,cc;



[Diagram D6a: Latest transaction ID and card used for the transaction]

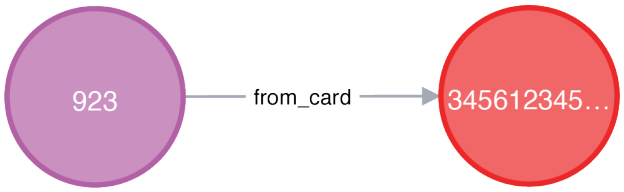
Node properties of transaction
<id>: 82
: success
transaction_amount: 2000
transaction_datetime: "2022-11-21T07:08:32.142000000Z"
transaction_id: 922

Node properties of credit_card
<id>: 48
balance: 20000
balance_date: "2022-12-01T23:08:32.142000000Z"
card_number: 3456123456789104
expiration_date: "2025-04-01"
limit: 2000
security_code: 804

Find new transaction ID by adding 1 to the last transaction ID (which is 922)

```
MATCH (t0:transaction)
WITH t0
ORDER BY t0.transaction_id DESC
LIMIT 1
WITH t0.transaction_id +1 as new_transaction_id,t0

// Then, create a new transaction node with the new transaction ID and set the datetime and
transaction amount

CREATE (t:transaction{transaction_id:new_transaction_id, transaction_datetime: datetime("2022-11-
21T07:12:32.142Z"),transaction_amount: 2000})

// Then, create the billed_at, delivered_at, from_ip, with_login relationship for the new transaction
node

WITH new_transaction_id,t,t0
MATCH(t:transaction{transaction_id:new_transaction_id}),(ad:address{add_id:1005})
MERGE(t)-[:billed_at]->(ad)
WITH new_transaction_id,t,t0
```

```
MATCH(t:transaction{transaction_id:new_transaction_id}),(ad:address{add_id:1005})
MERGE(t)-[:delivered_at]->(ad)
WITH new_transaction_id,t,t0
MATCH(t:transaction{transaction_id:new_transaction_id}),(ip:ip_address{ipid:3012})
MERGE(t)-[:from_ip]->(ip)
WITH new_transaction_id,t,t0
MATCH(t:transaction{transaction_id:new_transaction_id}),(l:login{login_id: 508})
MERGE(t)-[:with_login]->(l)

// Update balance of credit card after deduction of transaction amount from latest transaction and
update date to the latest transaction date

WITH new_transaction_id,t,t0
MATCH(t:transaction{transaction_id:new_transaction_id}),(cc:credit_card{card_number:345612345678
9104})
MERGE(t)-[:from_card]->(cc)
SET cc.balance_date = datetime() , cc.balance = cc.balance - t0.transaction_amount
RETURN cc,t
```



[Diagram D6b: New transaction ID and card used for the transaction]

Node properties of transaction
<id>: 99
transaction_amount: 2000
transaction_datetime: "2022-11-21T07:12:32.142000000Z"
transaction_id: 923

Node properties of credit_card

<id>: 48
balance: 18000
balance_date: "2023-01-09T01:50:58.433000000Z"
card_number: 3456123456789104
expiration_date: "2025-04-01"
limit: 2000
security_code: 804

The balance of credit card is deducted 2000 from 20000 and it is now 18000. The balance_date is updated to current date and time.