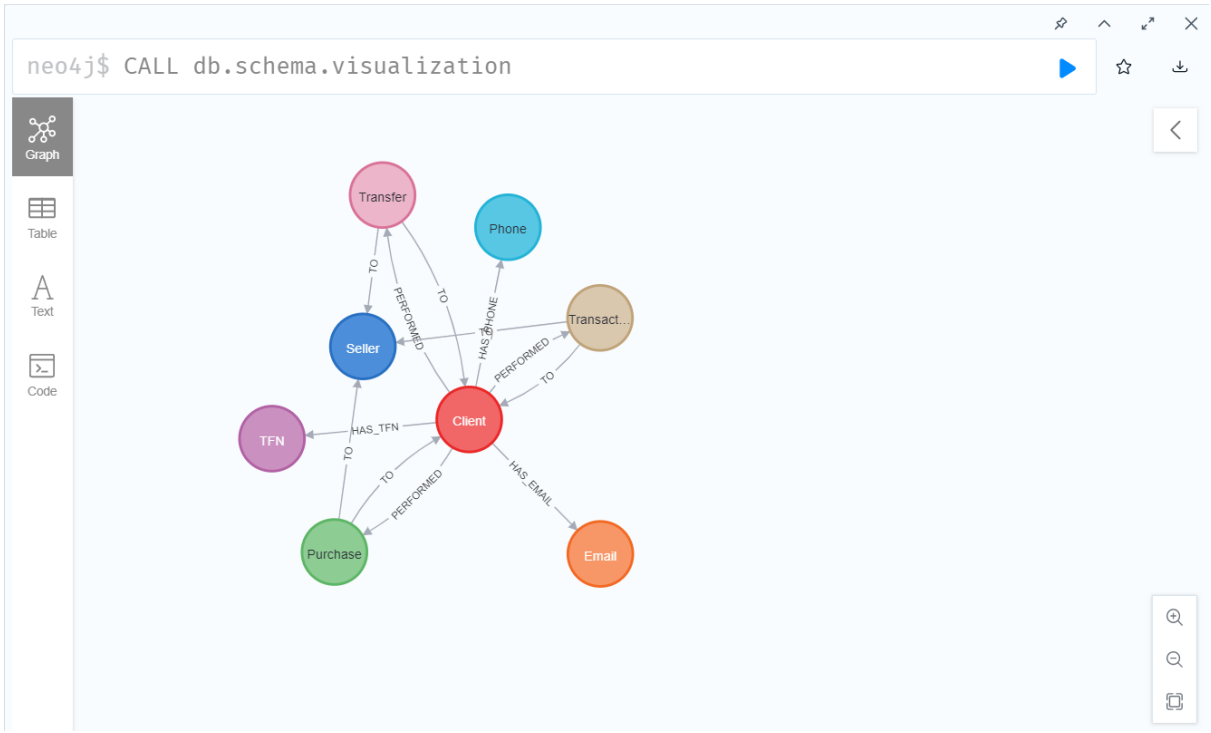
ASSIGNMENT 2 – Niyati - 47943319

Youtube video link : https://youtu.be/OejmjtQ6iQs

PART 1

```
// Ensure uniqueness of clients based on their ID
CREATE CONSTRAINT client_id_unique FOR (c:Client) REQUIRE c.id IS UNIQUE;

// Ensure uniqueness of sellers based on their ID
CREATE CONSTRAINT seller_id_unique FOR (s:Seller) REQUIRE s.id IS UNIQUE;

// Ensure uniqueness of transactions based on their ID
CREATE CONSTRAINT transaction_id_unique FOR (t:Transaction) REQUIRE t.id IS UNIQUE;

// Indexes to optimize the queries with custom names
CREATE INDEX client_name_index FOR (c:Client) ON (c.name);
CREATE INDEX seller_name_index FOR (s:Seller) ON (s.name);
CREATE INDEX transaction_time_index FOR (t:Transaction) ON (t.time);
CREATE INDEX transaction_amount_index FOR (t:Transaction) ON (t.amount);

// LOADING CSV
// Load Client data and create relationships with Phone, Email, TFN
LOAD CSV WITH HEADERS FROM 'file:///clients.csv' AS row
MERGE (c:Client {id: row.id, name: row.name})
MERGE (p:Phone {number: row.phone})
MERGE (e:Email {address: row.email})
MERGE (t:TFN {number: row.tfn})
MERGE (c)-[:HAS_PHONE]->(p)
MERGE (c)-[:HAS_EMAIL]->(e)
MERGE (c)-[:HAS_TFN]->(t);

// Load Store data and create Seller nodes
```

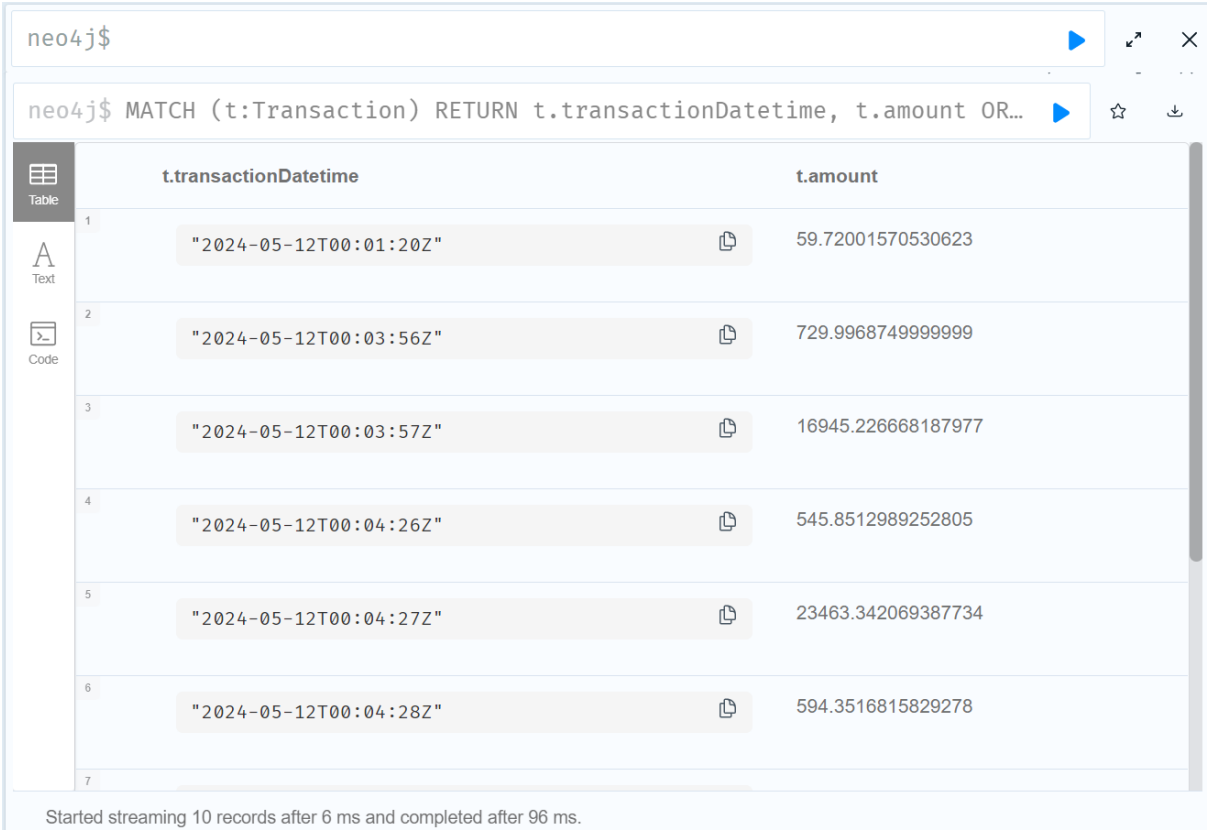
```
LOAD CSV WITH HEADERS FROM 'file:///stores.csv' AS row
MERGE (s:Seller {id: row.id, name: row.name});

LOAD CSV WITH HEADERS FROM 'file:///purchase.csv' AS row
MATCH (c:Client {id: row.idFrom}), (s:Seller {id: row.idTo})
CREATE (c)-[:PERFORMED]->(p:Purchase:Transaction {
   amount: toFloat(row.amount),
   time: datetime("2024-05-12T00:00:00").epochMillis + toInteger(row.timeOffset) * 1000
})
CREATE (p)-[:TO]->(s);
LOAD CSV WITH HEADERS FROM 'file:///xfer.csv' AS row
MATCH (c1:Client {id: row.idFrom}), (c2:Client {id: row.idTo})
CREATE (c1)-[:PERFORMED]->(t:Transfer:Transaction {
   amount: toFloat(row.amount),
   time: datetime("2024-05-12T00:00:00").epochMillis + toInteger(row.timeOffset) * 1000
})
CREATE (t)-[:TO]->(c2);

CALL db.schema.visualization
```



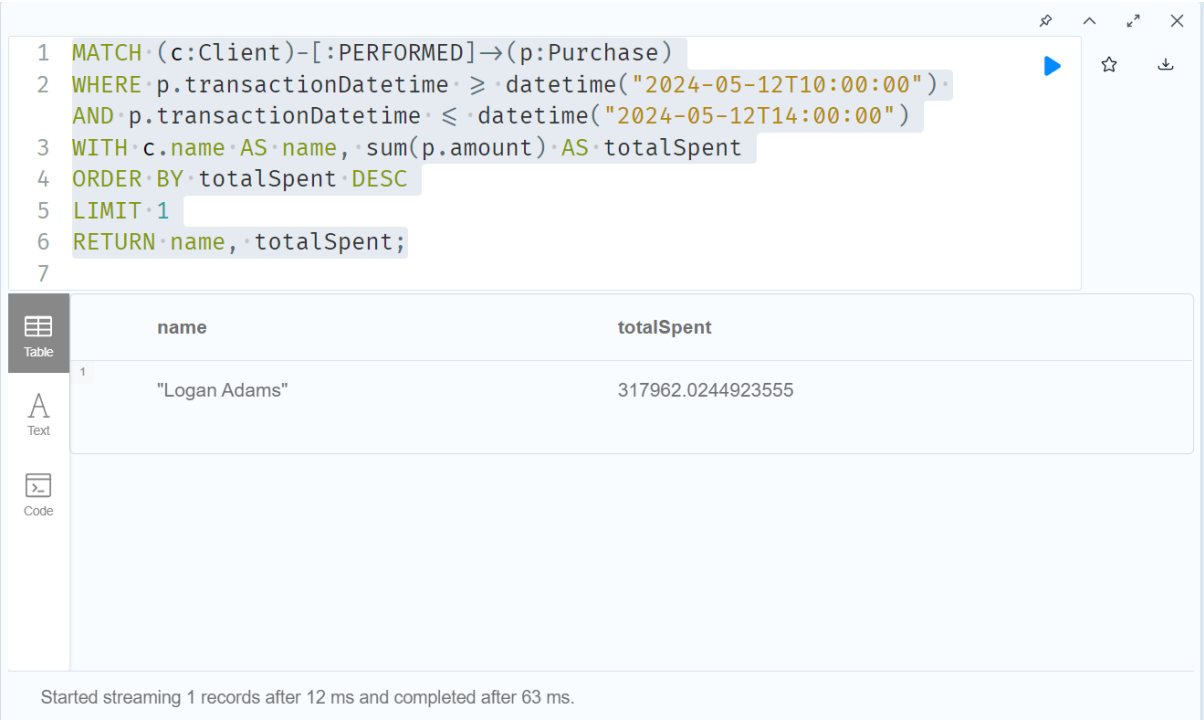
PART 2

// Problem 1

```
MATCH (t:Transaction)
SET t.transactionDatetime = datetime({epochMillis: t.time});


MATCH (t:Transaction)
RETURN t.transactionDatetime, t.amount
ORDER BY t.transactionDatetime
LIMIT 10;
```




```
MATCH (c:Client)-[:PERFORMED]->(p:Purchase)
WHERE p.transactionDatetime >= datetime("2024-05-12T10:00:00") AND p.transactionDatetime <= datetime("2024-05-12T14:00:00")
WITH c.name AS name, sum(p.amount) AS totalSpent
ORDER BY totalSpent DESC
LIMIT 1
RETURN name, totalSpent;
```



// PROBLEM 2

```
MATCH (c:Client)
OPTIONAL MATCH (c)<-[:TO]-(incoming:Transaction)
OPTIONAL MATCH (c)-[:PERFORMED]->(outgoing:Transaction)
WITH c.name AS name,
    sum(CASE WHEN incoming IS NOT NULL THEN incoming.amount ELSE 0 END) AS totalIncoming,
    sum(CASE WHEN outgoing IS NOT NULL THEN outgoing.amount ELSE 0 END) AS totalOutgoing,
    max(outgoing.amount) AS big_spend
WITH name, totalIncoming, totalOutgoing, (totalIncoming - totalOutgoing) AS balance, big_spend
WHERE balance < 0
RETURN name, balance, big_spend
ORDER BY balance ASC
LIMIT 5;
```

// Problem 3:

MATCH (s:Seller {name: 'Woods'})<-[:TO]-(p:Purchase)<-[:PERFORMED]-(c:Client)

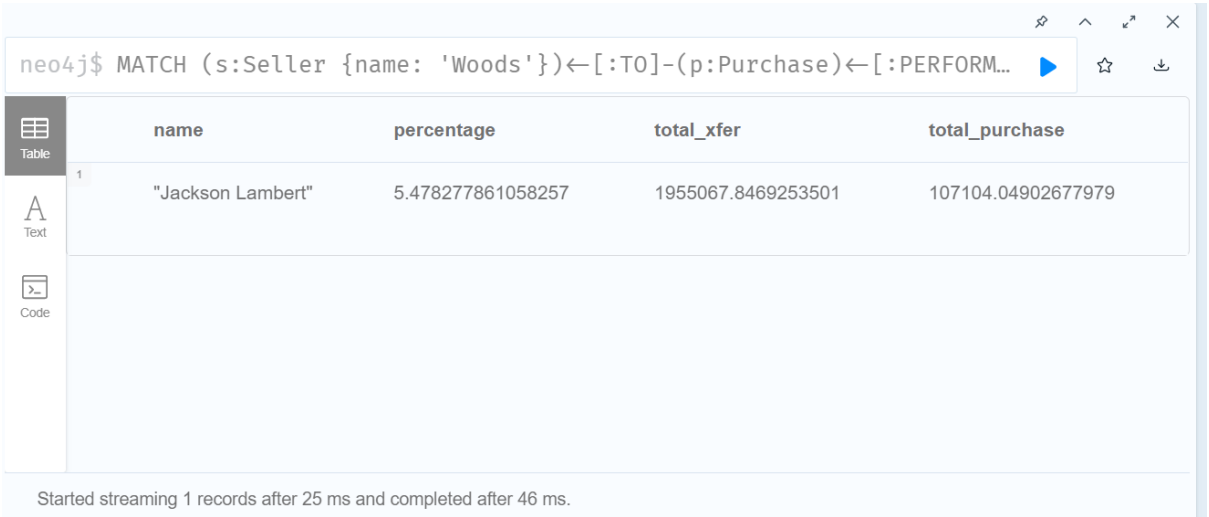
WITH s, c, sum(p.amount) AS total_purchase

MATCH (c)<-[r:TO]-(t:Transfer)<-[:PERFORMED]-(d:Client)

WITH c.name AS name, sum(t.amount) AS total_xfer, total_purchase, (total_purchase / sum(t.amount)) * 100 AS percentage

WHERE percentage >= 5

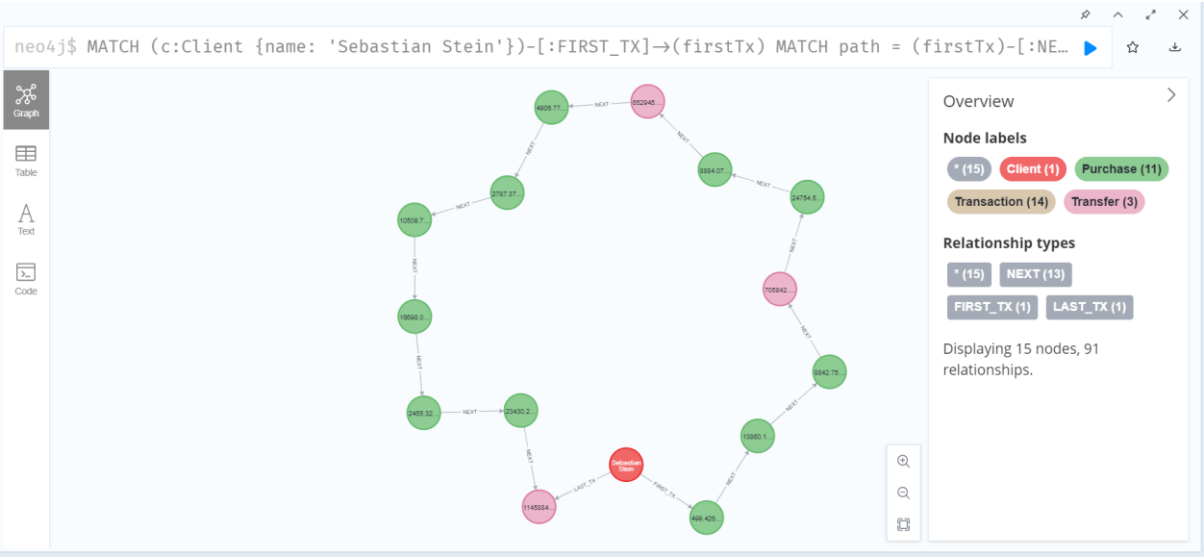
RETURN name, percentage, total_xfer, total_purchase;



// Problem 4

// Step1 : Creating NEXT Relationships Between Transactions

```
 MATCH (c:Client)-[:PERFORMED]->(t:Transaction)
WITH c, t
ORDER BY c.id, t.transactionDatetime
WITH c, collect(t) AS transactions
UNWIND range(0, size(transactions)-2) AS idx
WITH transactions[idx] AS t1, transactions[idx+1] AS t2
CREATE (t1)-[:NEXT]->(t2);

// Step2: Creating FIRST_TX and LAST_TX Relationships
MATCH (c:Client)-[:PERFORMED]->(t:Transaction)
WITH c, t
ORDER BY c.id, t.transactionDatetime
WITH c, collect(t) AS transactions
WITH c, transactions[0] AS firstTx, transactions[-1] AS lastTx
CREATE (c)-[:FIRST_TX]->(firstTx)
CREATE (c)-[:LAST_TX]->(lastTx);

// Step3: Ordering relationship
MATCH (c:Client)-[:LAST_TX]->(lastTx)-[:NEXT]->(firstTx)
WHERE (c)-[:FIRST_TX]->(firstTx)
CREATE (lastTx)-[:NEXT]->(firstTx);

// Step 4: Take any example of client name
MATCH (c:Client {name: 'Sebastian Stein'})-[:FIRST_TX]->(firstTx)
MATCH path = (firstTx)-[:NEXT*]->(lastTx)
WITH c, path
OPTIONAL MATCH (c)-[r:PERFORMED]->()
DELETE r
RETURN c,
    nodes(path) AS transactions,
    [rel IN relationships(path) WHERE type(rel) = 'NEXT'] AS nextRelationships;
```



PART 3

PART A

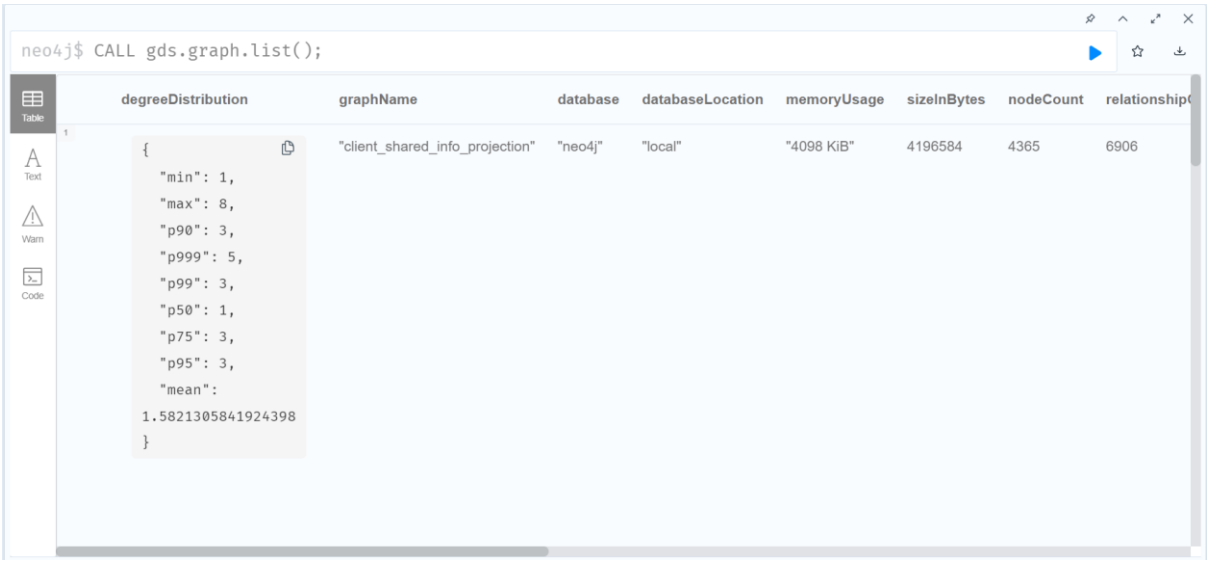
i)

// creating projection to analysis client who share info

```
CALL gds.graph.project(
  'client_shared_info_projection',
  ['Client', 'Email', 'Phone', 'TFN'], // Node labels
  {
    HAS_EMAIL: {type: 'HAS_EMAIL', orientation: 'UNDIRECTED'},
    HAS_PHONE: {type: 'HAS_PHONE', orientation: 'UNDIRECTED'},
    HAS_TFN: {type: 'HAS_TFN', orientation: 'UNDIRECTED'}
  }
```

);

CALL gds.graph.list();



// WCC finds clusters of nodes in a graph where there is a path between any two nodes

CALL gds.wcc.stream('client_shared_info_projection')

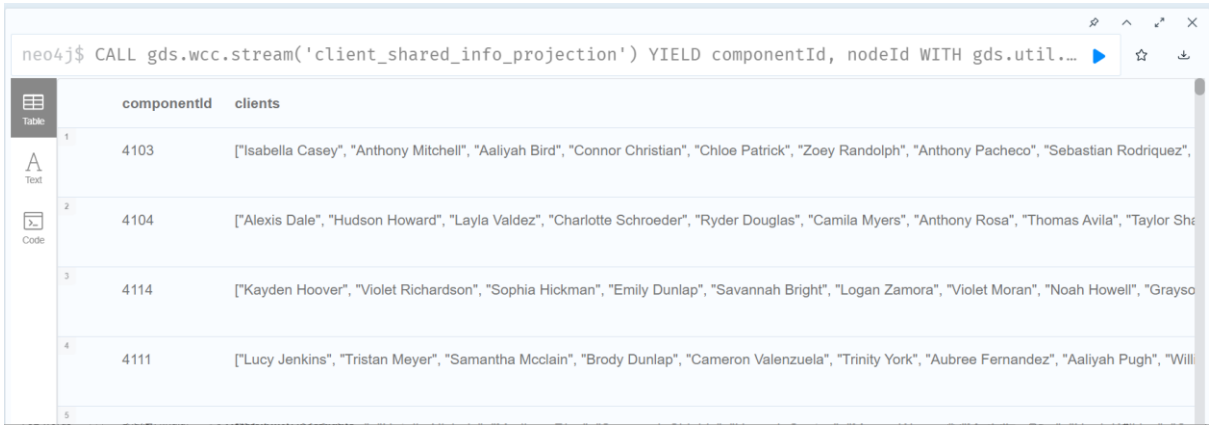
YIELD componentId, nodeId

WITH gds.util.asNode(nodeId) AS node, componentId

WHERE node:Client

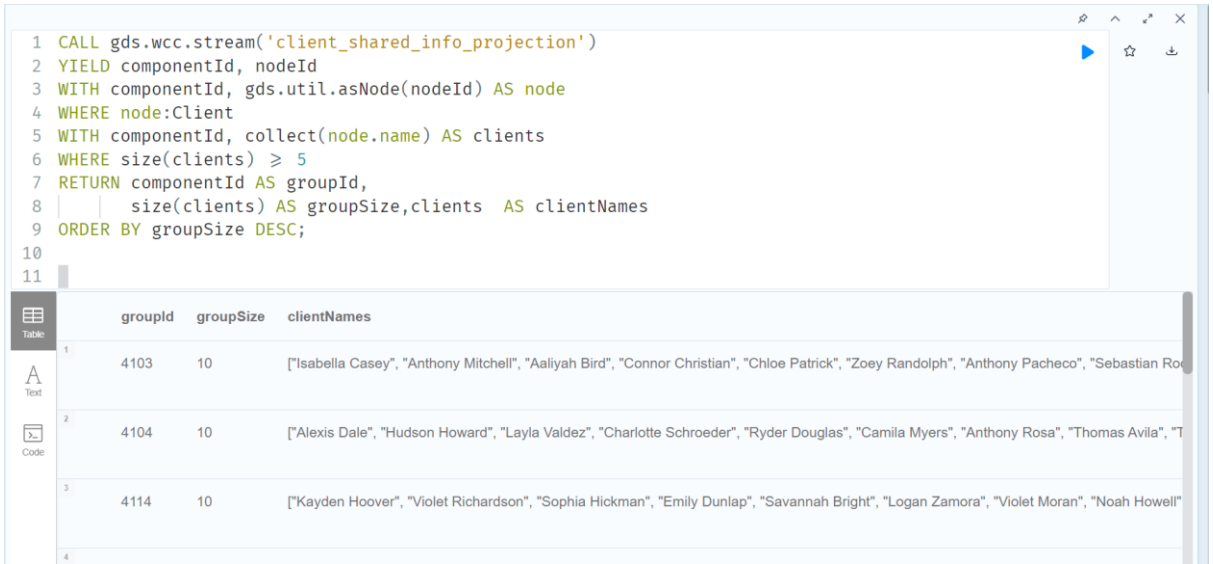
RETURN componentId, collect(node.name) AS clients

ORDER BY size(clients) DESC;



ii)

```
// identifying larger groups
CALL gds.wcc.stream('client_shared_info_projection')
YIELD componentId, nodeId
WITH componentId, gds.util.asNode(nodeId) AS node
WHERE node:Client
WITH componentId, collect(node.name) AS clients
WHERE size(clients) >= 5
RETURN componentId AS groupId,
    size(clients) AS groupSize,clients  AS clientNames
ORDER BY groupSize DESC;
```



iii)

// Assigning a groupId to each Client node based on the component they belong to.

CALL gds.wcc.stream('client_shared_info_projection')

YIELD componentId, nodeId

WITH componentId, gds.util.asNode(nodeId) AS node

WHERE node:Client

SET node.groupId = componentId;

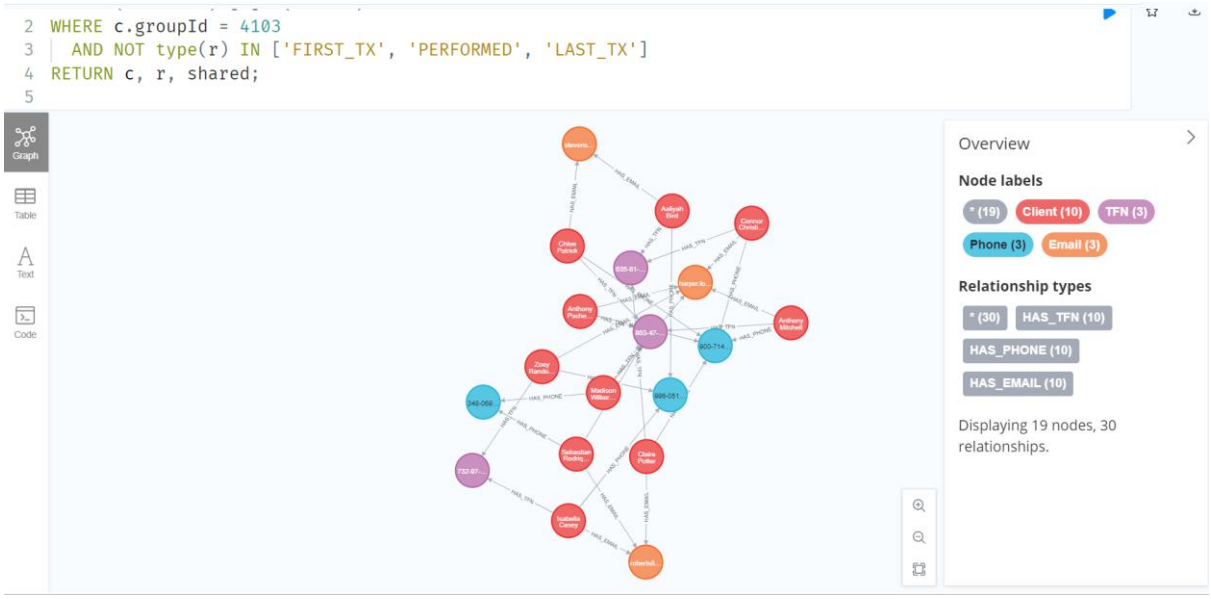
// visualisation for each largest group size

MATCH (c:Client)-[r]->(shared)

WHERE c.groupId = 4103

AND NOT type(r) IN ['FIRST_TX', 'PERFORMED', 'LAST_TX']

RETURN c, r, shared;

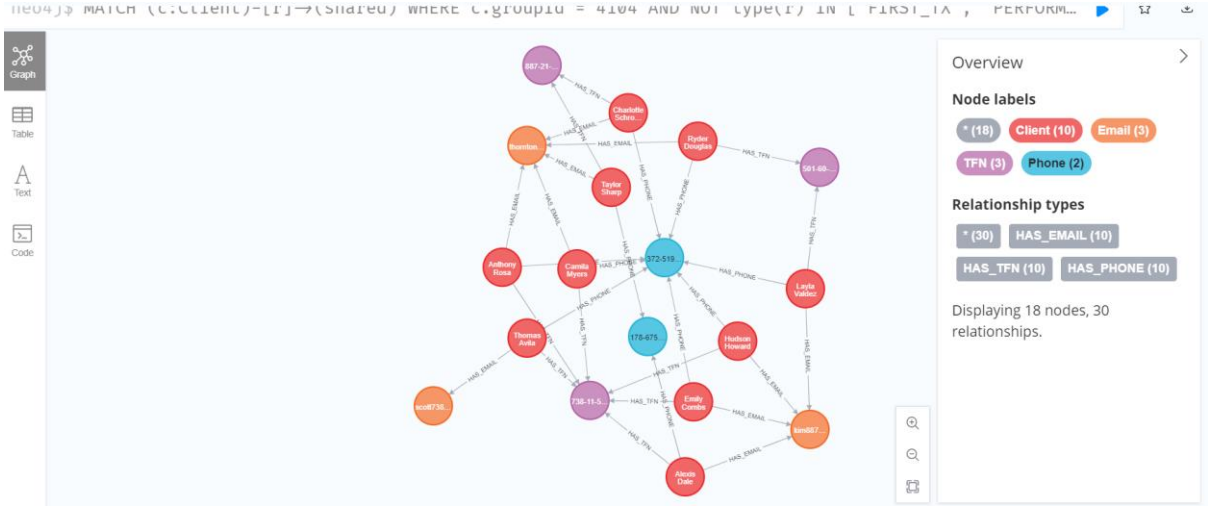


MATCH (c:Client)-[r]->(shared)

WHERE c.groupId = 4104

AND NOT type(r) IN ['FIRST_TX', 'PERFORMED', 'LAST_TX']

RETURN c, r, shared;

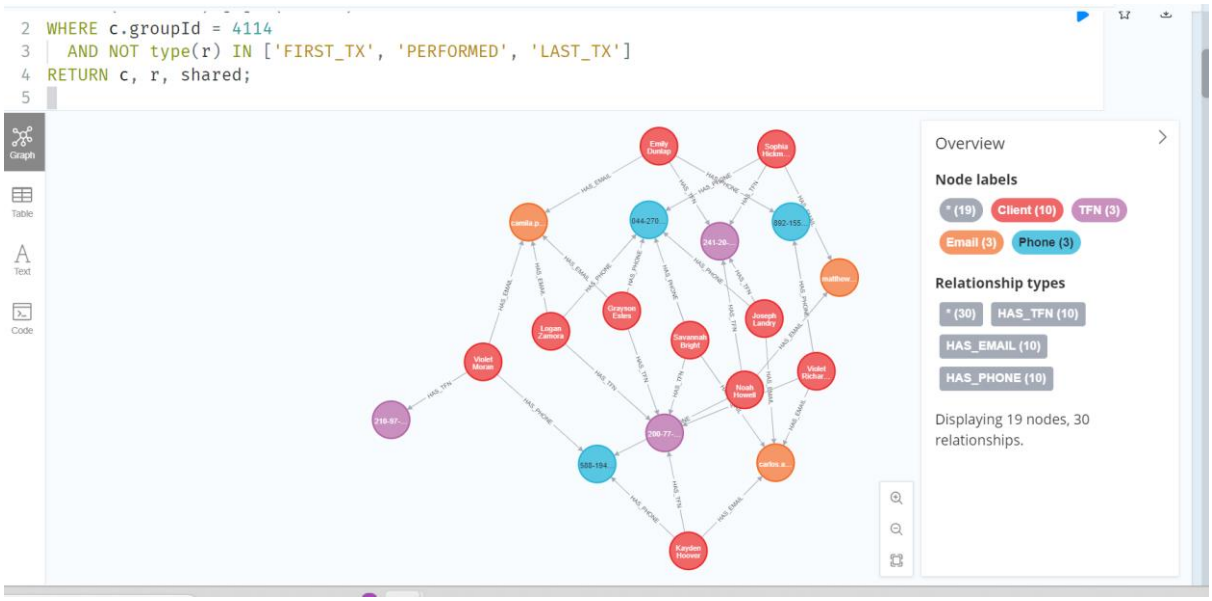


MATCH (c:Client)-[r]->(shared)

WHERE c.groupId = 4114

AND NOT type(r) IN ['FIRST_TX', 'PERFORMED', 'LAST_TX']

RETURN c, r, shared;



Part B

I)

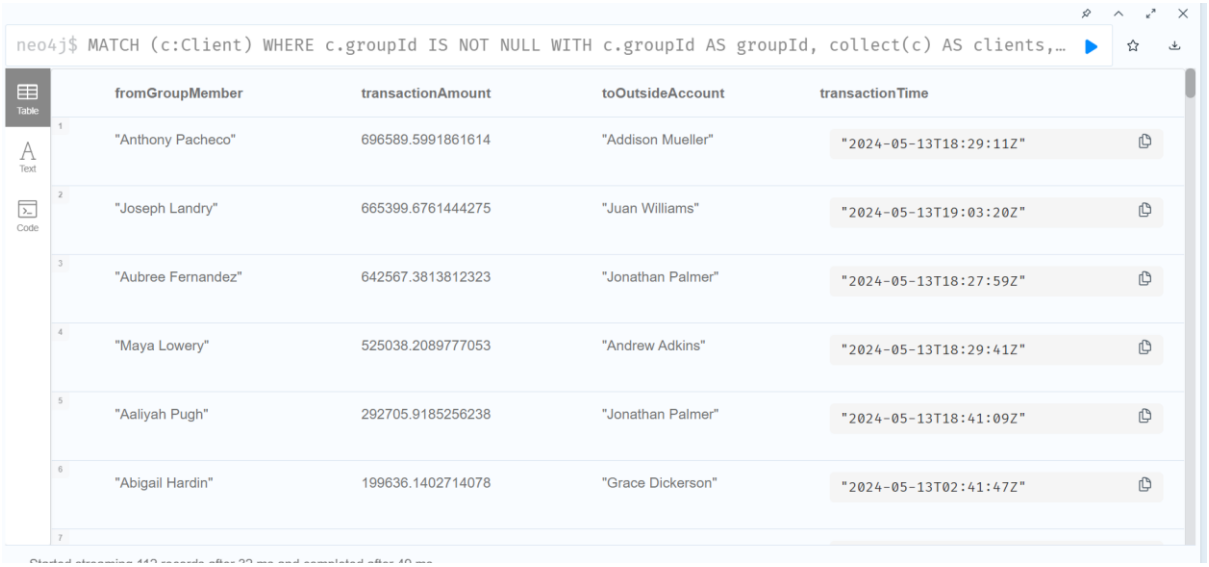
// identifies transactional relationships that members of larger fraud groups (more than 5 members) have with accounts outside of their immediate group

MATCH (c:Client)

WHERE c.groupId IS NOT NULL

WITH c.groupId AS groupId, collect(c) AS clients, count(c) AS groupSize

```
WHERE groupSize > 5
UNWIND clients AS groupMember
MATCH (groupMember)-[:PERFORMED]->(t:Transaction)-[:TO]->(outside:Client)
WHERE outside.groupId IS NULL OR outside.groupId <> groupMember.groupId
RETURN groupMember.name AS fromGroupMember,
    t.amount AS transactionAmount,
    outside.name AS toOutsideAccount,
    t.transactionDatetime AS transactionTime
ORDER BY transactionAmount DESC;
```



	fromGroupMember	transactionAmount	toOutsideAccount	transactionTime
1	"Anthony Pacheco"	696589.5991861614	"Addison Mueller"	"2024-05-13T18:29:11Z"
2	"Joseph Landry"	665399.6761444275	"Juan Williams"	"2024-05-13T19:03:20Z"
3	"Aubree Fernandez"	642567.3813812323	"Jonathan Palmer"	"2024-05-13T18:27:59Z"
4	"Maya Lowery"	525038.2089777053	"Andrew Adkins"	"2024-05-13T18:29:41Z"
5	"Aaliyah Pugh"	292705.9185256238	"Jonathan Palmer"	"2024-05-13T18:41:09Z"
6	"Abigail Hardin"	199636.1402714078	"Grace Dickerson"	"2024-05-13T02:41:47Z"
7				

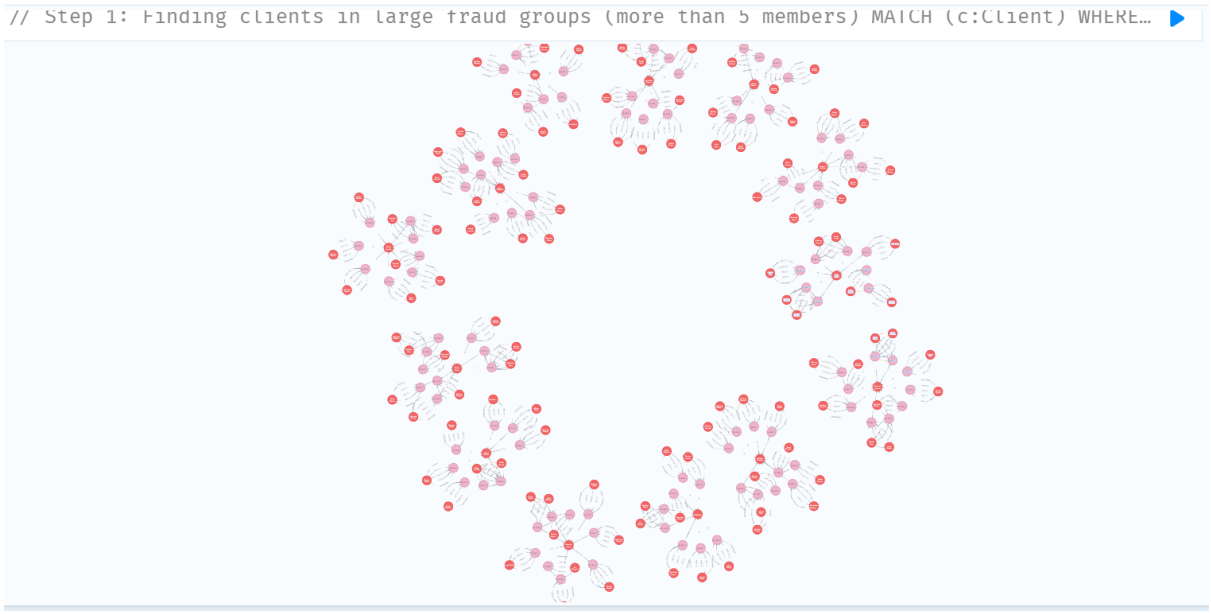
ii)

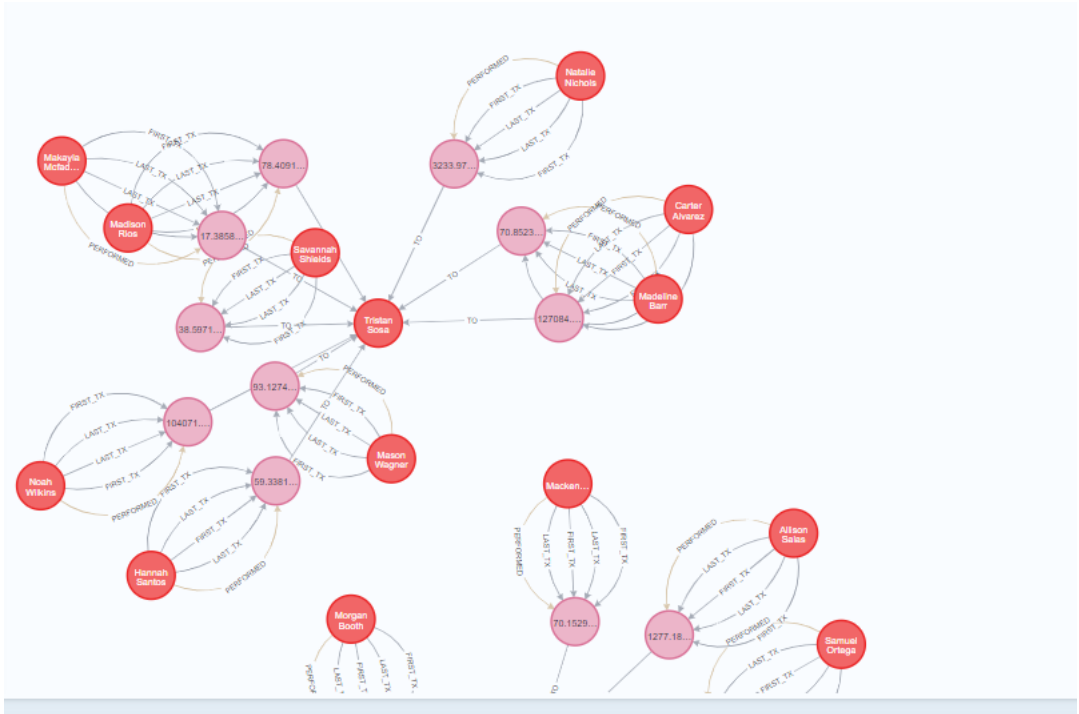
```
// Step 1: Find clients in large fraud groups (more than 5 members)
MATCH (c:Client)
WHERE c.groupId IS NOT NULL
WITH c.groupId AS groupId, collect(c) AS clients, count(c) AS groupSize
WHERE groupSize > 5
UNWIND clients AS groupMember

// Step 2: Find transactions from group members to clients outside their group
MATCH (groupMember)-[:PERFORMED]->(t:Transaction)-[:TO]->(outside:Client)
```

WHERE outside.groupId IS NULL OR outside.groupId <> groupMember.groupId

// Step 3: Return the relevant nodes and relationships for GDS projection

RETURN groupMember, outside, t;





// Step 4: Project the Subgraph into GDS

Now, we'll project the subgraph of the relevant clients and their transactional relationships into GDS.

We'll project both the Client nodes and the Transaction relationships into GDS, focusing on the transactional relationships between the fraud group members and the clients outside their group.

// GDS Projection Query:

```
CALL gds.graph.project(
 'fraud_movement_projection',
 ['Client', 'Transaction'],  // Include both Client and Transaction nodes
 {
  PERFORMED: {
   type: 'PERFORMED',
   orientation: 'UNDIRECTED'  // Use UNDIRECTED for connected groups
  },
  TO: {
   type: 'TO',  // Include TO relationships between Transactions and Clients
   orientation: 'UNDIRECTED'
  }
 }
)
YIELD graphName, nodeCount, relationshipCount;
```

//Step 5 :Running a GDS Algorithm to Find Tightly Connected Groups

Now that we have projected the subgraph into GDS, we can use a **community detection algorithm** to find tightly connected groups within this subset of clients. One appropriate algorithm for this is the **Louvain** algorithm, which detects communities in large graphs based on modularity. This means evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network.

```
CALL gds.louvain.stream('fraud_movement_projection')
```

YIELD nodeId, communityId

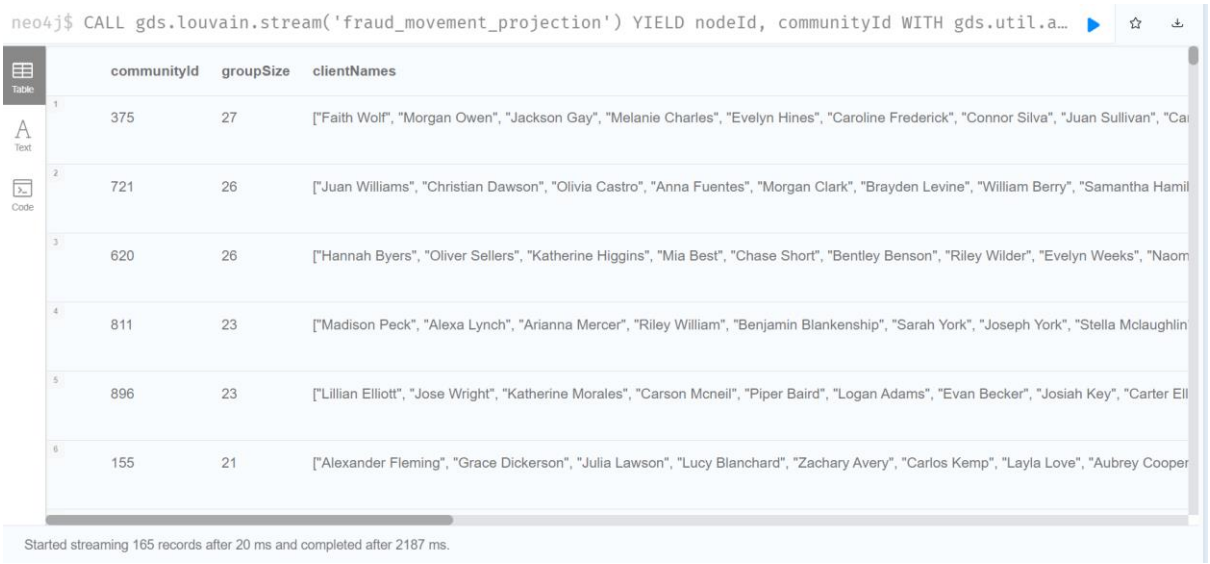
WITH gds.util.asNode(nodeId) AS client, communityId

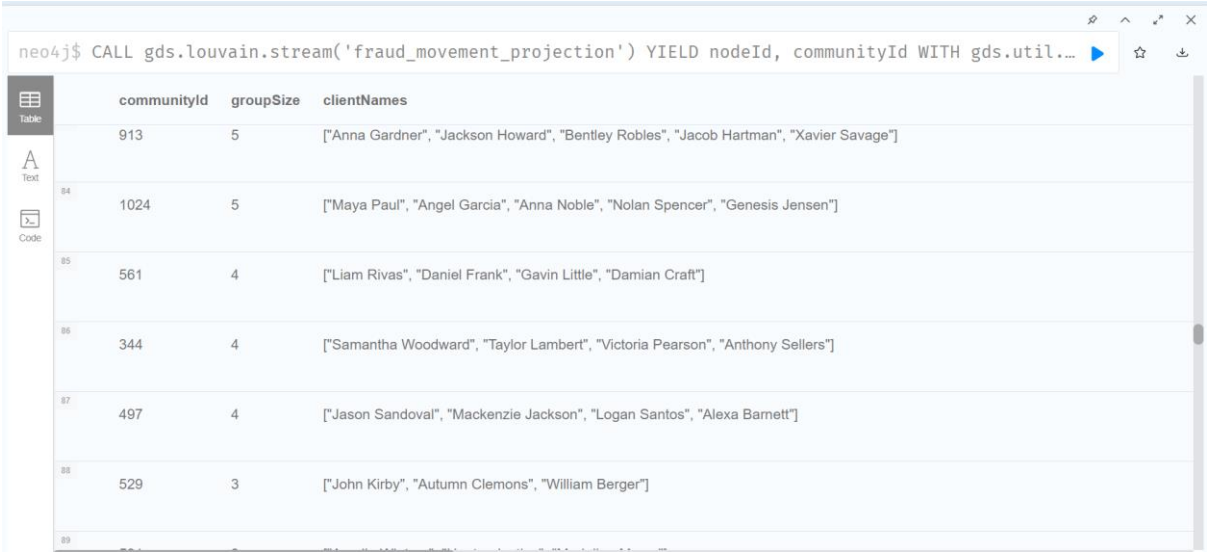
RETURN communityId,

size(collect(DISTINCT client.name)) AS groupSize,

collect(DISTINCT client.name) AS clientNames

ORDER BY groupSize DESC



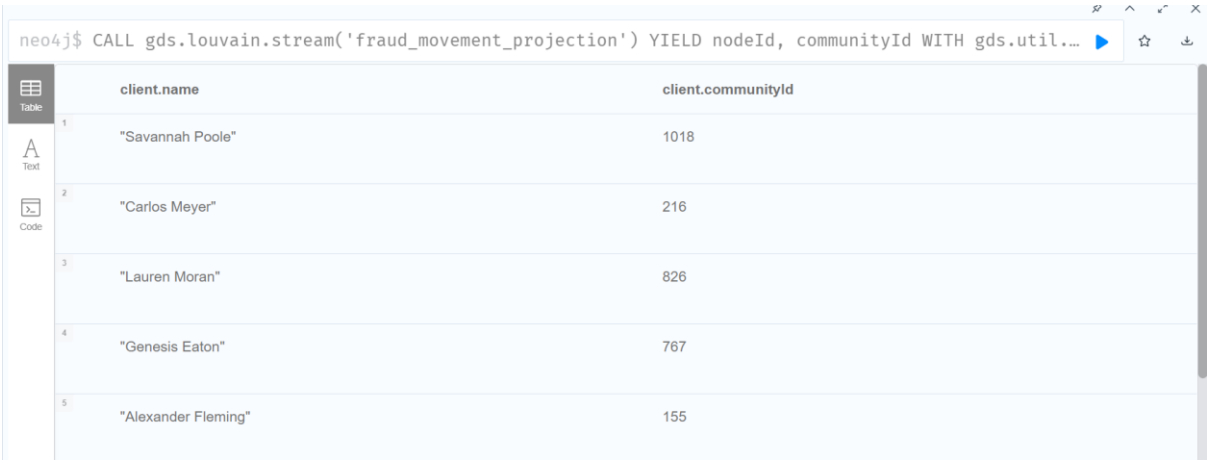


iii)

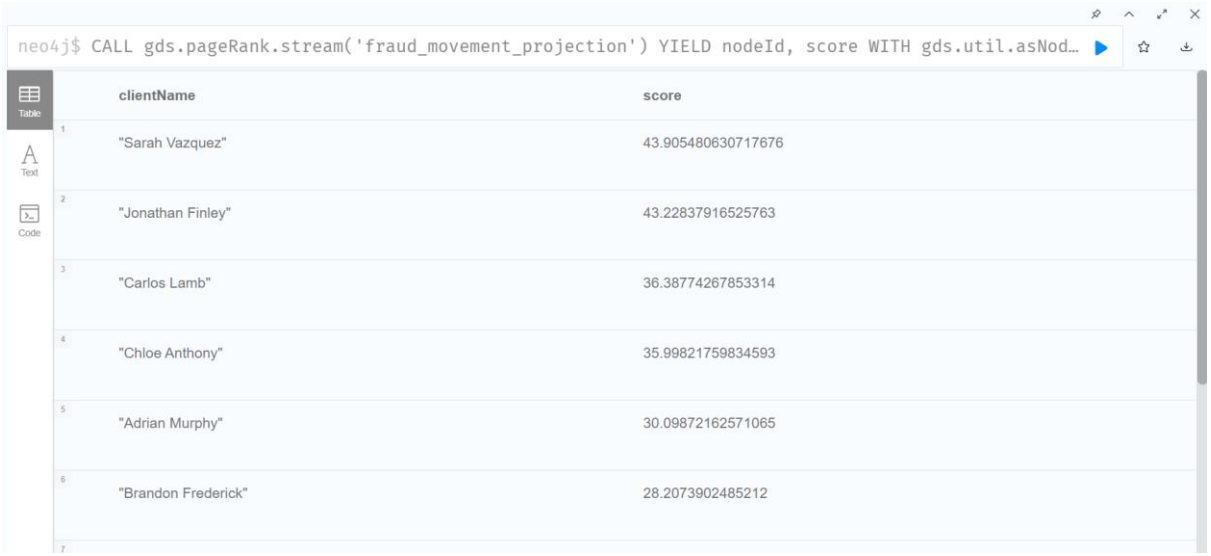
// Ensuring that the communityId is written back to the actual Client nodes in the database.

CALL gds.louvain.stream('fraud_movement_projection')

```
YIELD nodeId, communityId

WITH gds.util.asNode(nodeId) AS client, communityId

SET client.communityId = communityId

RETURN client.name, client.communityId

LIMIT 10;
```




```
// using algorithms like pageRank to identify key players in the network.

CALL gds.pageRank.stream('fraud_movement_projection')

YIELD nodeId, score

WITH gds.util.asNode(nodeId) AS client, score

WHERE client.communityId = 375

SET client.pageRank = score

RETURN client.name AS clientName, client.pageRank AS score

ORDER BY score DESC

LIMIT 10
```

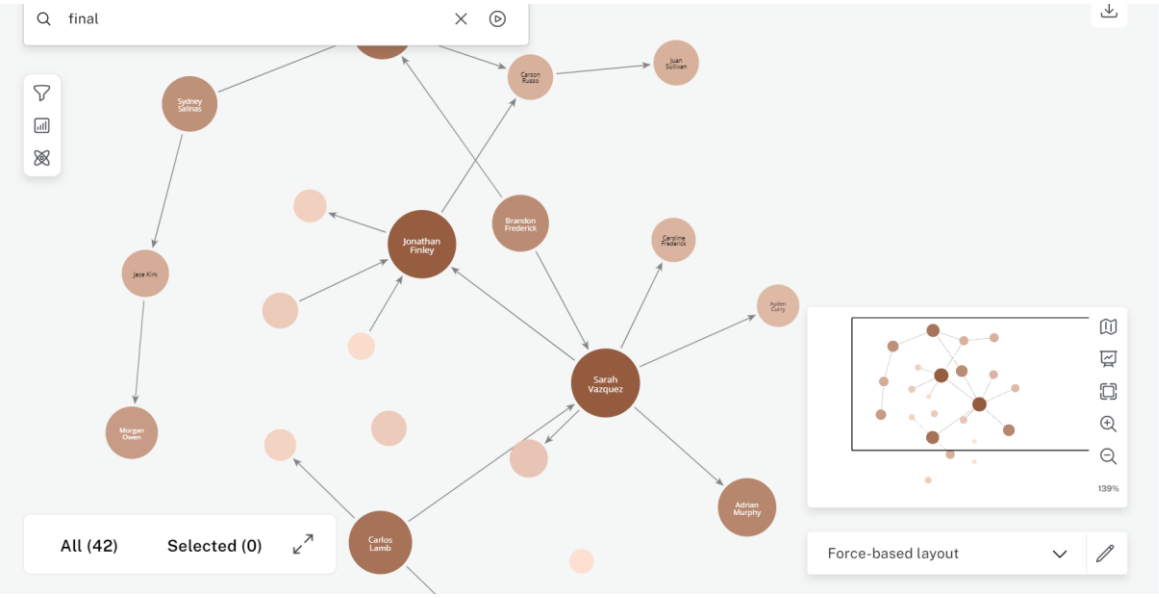


These accounts (e.g., Sarah Vazquez, Jonathan Finley, etc.) are likely key suspects in funneling funds within the detected fraudulent communities.

iv)

//visualisation of the community having largest group size and then doing value based styling in neo 4j bloom

```
MATCH (c:Client)-[p:PERFORMED]->(t:Transaction)-[r:TO]->(outside:Client)

WHERE c.communityId = 375

WITH c, outside, COUNT(t) AS transactionCount

CALL apoc.create.vRelationship(c, 'PERFORMED_WITH', {count: transactionCount}, outside) YIELD rel

RETURN c, rel;
```



1. **apoc.create.vRelationship**: This part creates a **virtual relationship** between the two **Client** nodes (c and outside).
2. **'PERFORMED_WITH'**: The type of the virtual relationship is 'PERFORMED_WITH'.
3. **{count: transactionCount}**: The relationship has a property called **count**, which stores the number of transactions that have occurred between the two clients (i.e., transactionCount).