

## 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 {
```

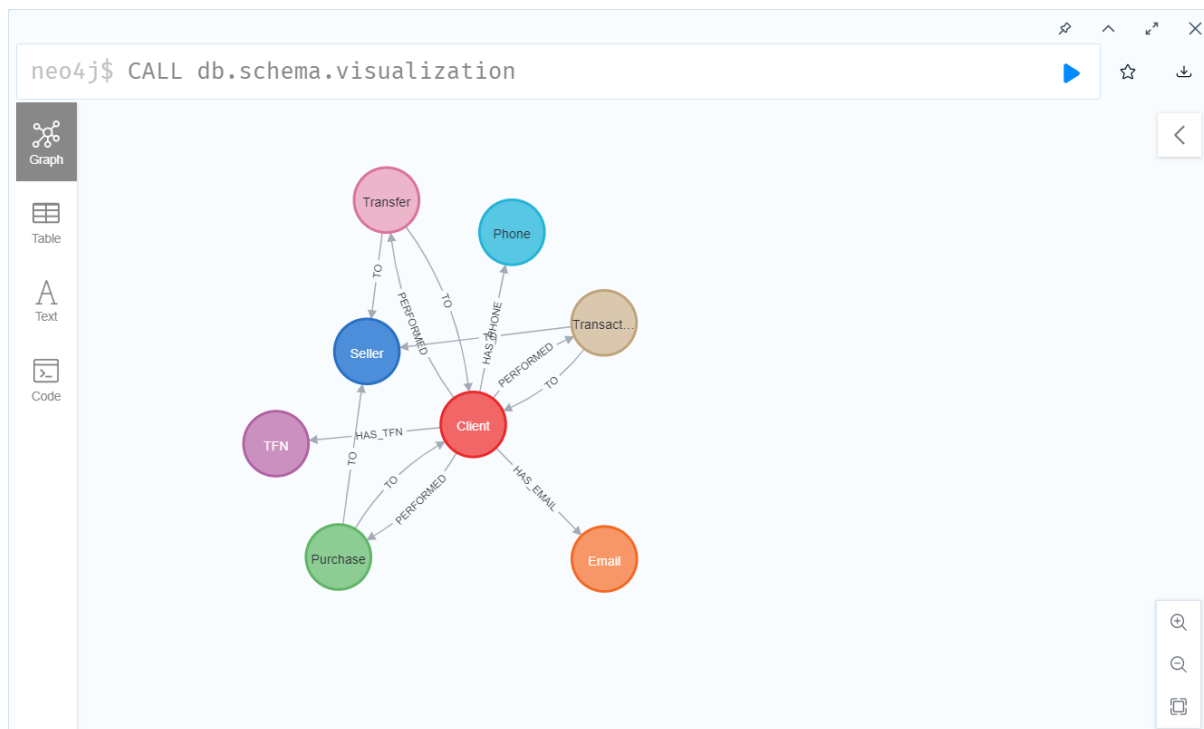
```
    amount: toFloat(row.amount),
```

```
    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;

neo4j\$

neo4j\$ MATCH (t:Transaction) RETURN t.transactionDatetime, t.amount OR...

	t.transactionDatetime	t.amount
1	"2024-05-12T00:01:20Z"	59.72001570530623
2	"2024-05-12T00:03:56Z"	729.9968749999999
3	"2024-05-12T00:03:57Z"	16945.226668187977
4	"2024-05-12T00:04:26Z"	545.8512989252805
5	"2024-05-12T00:04:27Z"	23463.342069387734
6	"2024-05-12T00:04:28Z"	594.3516815829278
7		

Started streaming 10 records after 6 ms and completed after 96 ms.

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;

```
1 MATCH (c:Client)-[:PERFORMED]->(p:Purchase)
2 WHERE p.transactionDatetime >= datetime("2024-05-12T10:00:00")
   AND p.transactionDatetime <= datetime("2024-05-12T14:00:00")
3 WITH c.name AS name, sum(p.amount) AS totalSpent
4 ORDER BY totalSpent DESC
5 LIMIT 1
6 RETURN name, totalSpent;
7
```

	name	totalSpent
1	"Logan Adams"	317962.0244923555

Started streaming 1 records after 12 ms and completed after 63 ms.

## // 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;

neo4j\$ MATCH (c:Client) OPTIONAL MATCH (c)←[:TO]-(incoming:Transaction) OPTIONAL MATCH (c)-[:PERFOR...

	name	balance	big_spend
1	"Josiah Franco"	-60540939.30839634	3596393.301257203
2	"Anthony Meadows"	-56851789.14138264	3665457.6404235195
3	"Aiden Morris"	-32295505.780635595	2679343.29623702
4	"Maya Paul"	-30947218.03559584	4804417.827526355
5	"Alexander Bowen"	-24749814.55994816	2349402.566678605

Started streaming 5 records after 44 ms and completed after 1228 ms.

### // Problem 3:

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

WITH s, c, sum(p.amount) AS total\_purchase

MATCH (c)<-[: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;

neo4j\$ MATCH (s:Seller {name: 'Woods'})<-[:TO]-(p:Purchase)<-[:PERFORM...

	name	percentage	total_xfer	total_purchase
1	"Jackson Lambert"	5.478277861058257	1955067.8469253501	107104.04902677979

Started streaming 1 records after 25 ms and completed after 46 ms.

### // 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);

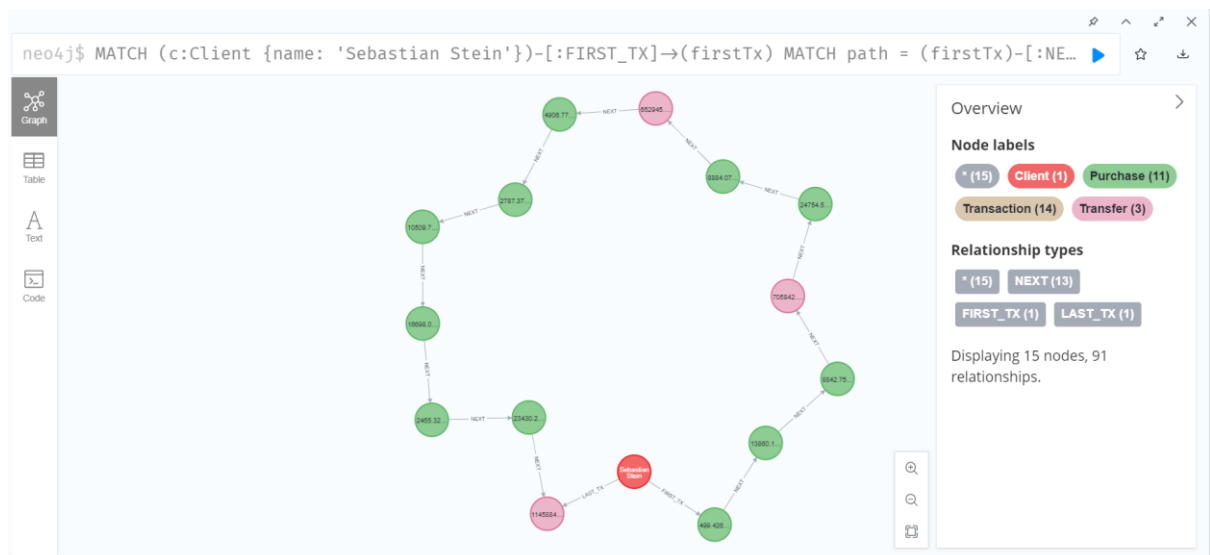
```

// 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;

```



}



);

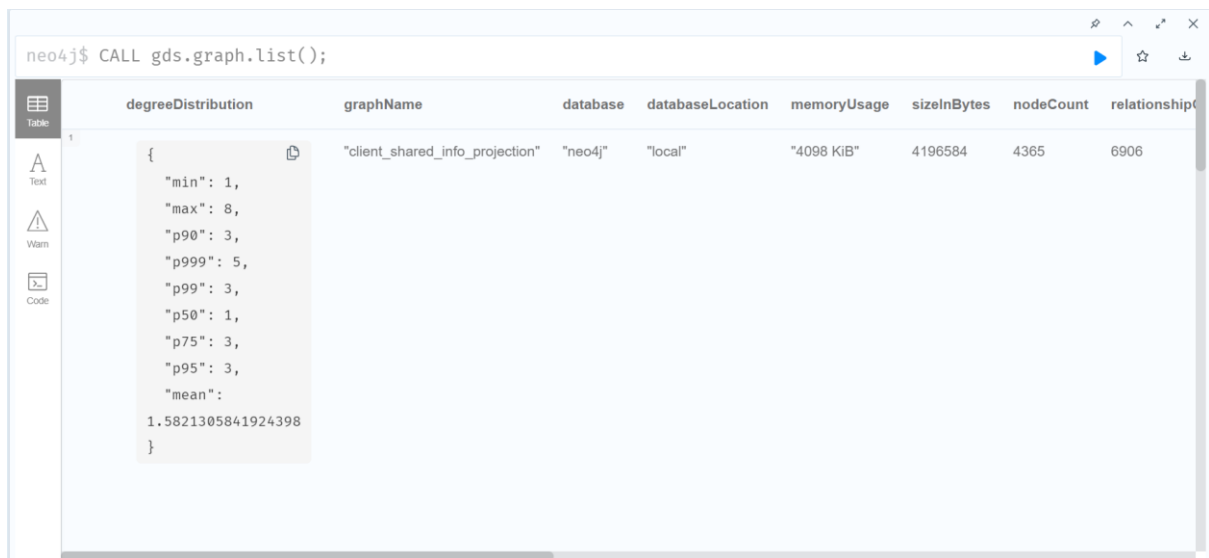


```
neo4j$ CALL gds.graph.project('client_shared_info_projection', ['Client', 'Email', 'Phone', 'TFN'], ...)
```

	nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMillis
1	<pre>{   "TFN": {     "label":     "TFN",     "properties":     {     },     "Phone": {       "label":       "Phone",       "properties":       {       }     }   } }</pre>	<pre>{   "HAS_TFN": {     "aggregation":     "DEFAULT",     "orientation":     "UNDIRECTED",     "indexInverse": false,     "properties":     {     },     "type": "HAS_TFN"   },   "HAS_EMAIL": {     "aggregation":     "DEFAULT",     "orientation":     "UNDIRECTED",     "indexInverse": false,     "properties":     {     },     "type": "HAS_EMAIL"   } }</pre>	"client_shared_info_projection"	4365	6906	438

Started streaming 1 records after 21 ms and completed after 551 ms.

CALL gds.graph.list();



```
neo4j$ CALL gds.graph.list();
```

	degreeDistribution	graphName	database	databaseLocation	memoryUsage	sizeInBytes	nodeCount	relationshipCount
1	<pre>{   "min": 1,   "max": 8,   "p90": 3,   "p99": 5,   "p999": 3,   "p50": 1,   "p75": 3,   "p95": 3,   "mean":   1.5821305841924398 }</pre>	"client_shared_info_projection"	"neo4j"	"local"	"4098 KiB"	4196584	4365	6906

// 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;

neo4j\$ CALL gds.wcc.stream('client\_shared\_info\_projection') YIELD componentId, nodeId WITH gds.util....

	componentId	clients
1	4103	["Isabella Casey", "Anthony Mitchell", "Aaliyah Bird", "Connor Christian", "Chloe Patrick", "Zoey Randolph", "Anthony Pacheco", "Sebastian Rodriguez",
2	4104	["Alexis Dale", "Hudson Howard", "Layla Valdez", "Charlotte Schroeder", "Ryder Douglas", "Camila Myers", "Anthony Rosa", "Thomas Avila", "Taylor She
3	4114	["Kayden Hoover", "Violet Richardson", "Sophia Hickman", "Emily Dunlap", "Savannah Bright", "Logan Zamora", "Violet Moran", "Noah Howell", "Grayso
4	4111	["Lucy Jenkins", "Tristan Meyer", "Samantha McClain", "Brody Dunlap", "Cameron Valenzuela", "Trinity York", "Aubree Fernandez", "Aaliyah Pugh", "Willi
5		

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;

```

1 CALL gds.wcc.stream('client_shared_info_projection')
2 YIELD componentId, nodeId
3 WITH componentId, gds.util.asNode(nodeId) AS node
4 WHERE node:Client
5 WITH componentId, collect(node.name) AS clients
6 WHERE size(clients) >= 5
7 RETURN componentId AS groupId,
8     size(clients) AS groupSize, clients AS clientNames
9 ORDER BY groupSize DESC;
10
11

```

	groupId	groupSize	clientNames
1	4103	10	["Isabella Casey", "Anthony Mitchell", "Aaliyah Bird", "Connor Christian", "Chloe Patrick", "Zoey Randolph", "Anthony Pacheco", "Sebastian Ro
2	4104	10	["Alexis Dale", "Hudson Howard", "Layla Valdez", "Charlotte Schroeder", "Ryder Douglas", "Camila Myers", "Anthony Rosa", "Thomas Avila", "T
3	4114	10	["Kayden Hoover", "Violet Richardson", "Sophia Hickman", "Emily Dunlap", "Savannah Bright", "Logan Zamora", "Violet Moran", "Noah Howell"
4			

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;
```

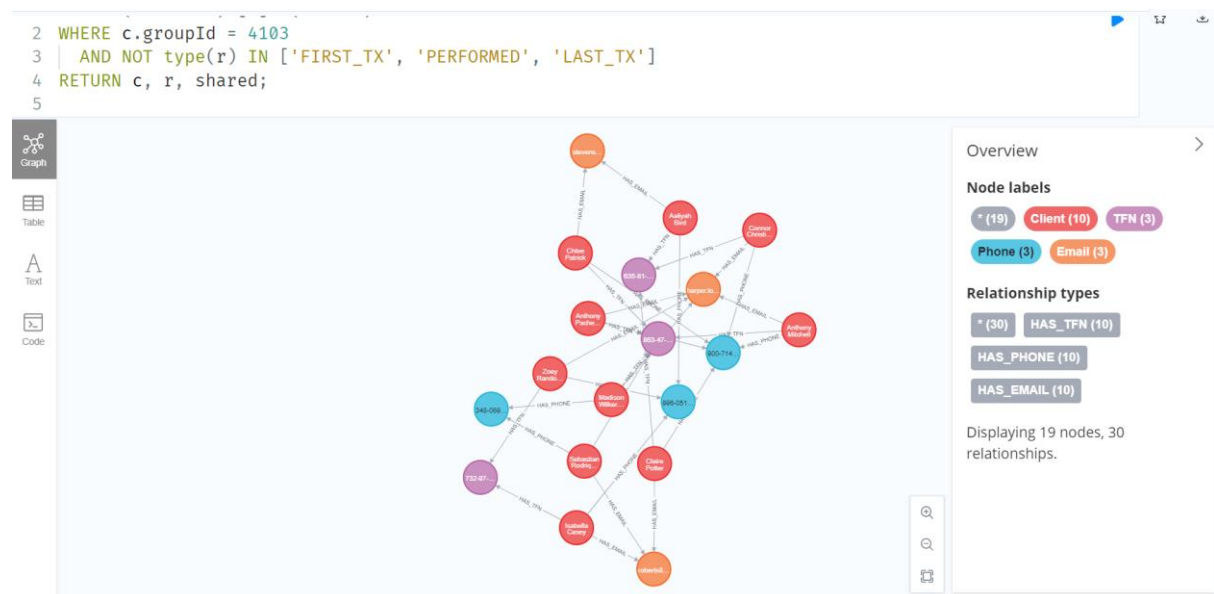
// visualisation for each largest group size

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

```
WHERE c.groupId = 4103
```

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

```
RETURN c, r, shared;
```

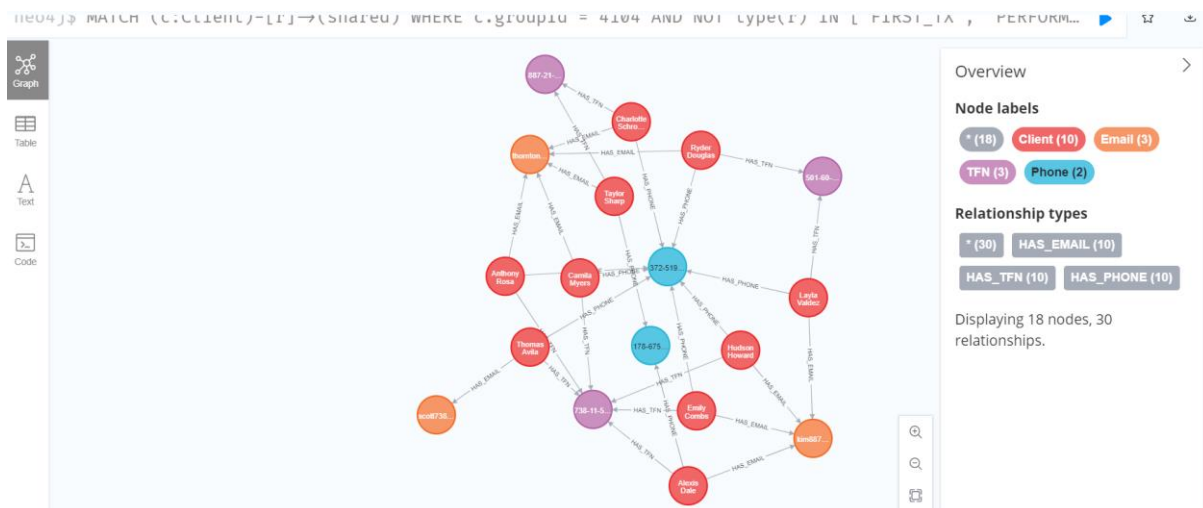


```
MATCH (c:Client)-[r]->(shared)
```

```
WHERE c.groupId = 4104
```

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

```
RETURN c, r, shared;
```

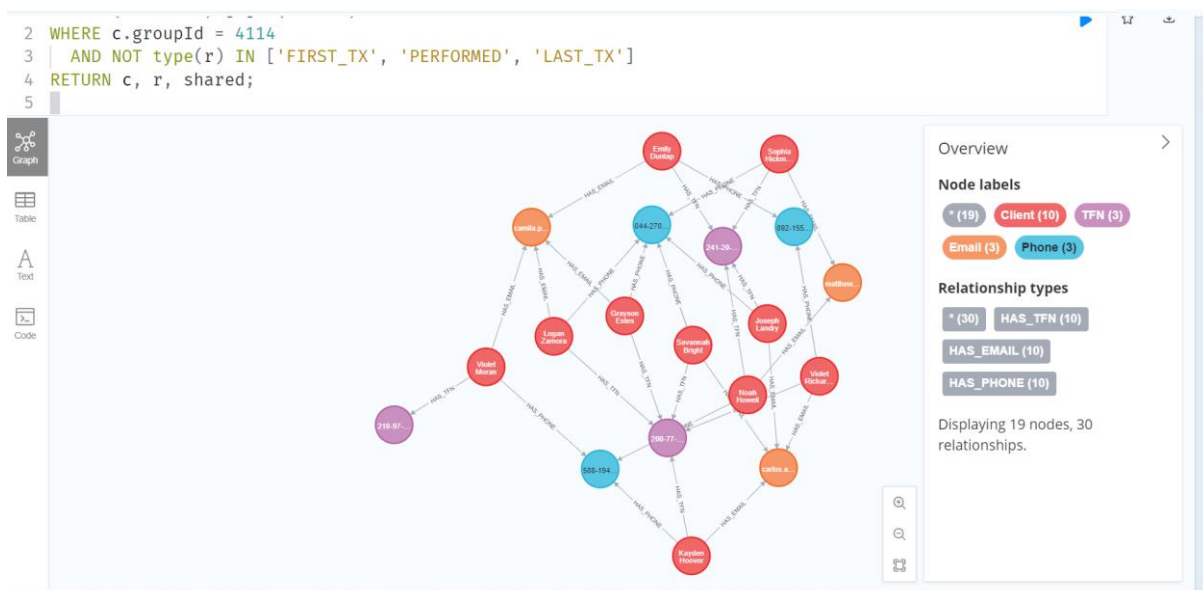


MATCH (c:Client)-[r]->(shared)

WHERE c.groupId = 4114

AND NOT type(r) IN ['FIRST\_TX', 'PERFORMED', 'LAST\_TX']

RETURN c, r, shared;



## Part B

1)

// 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

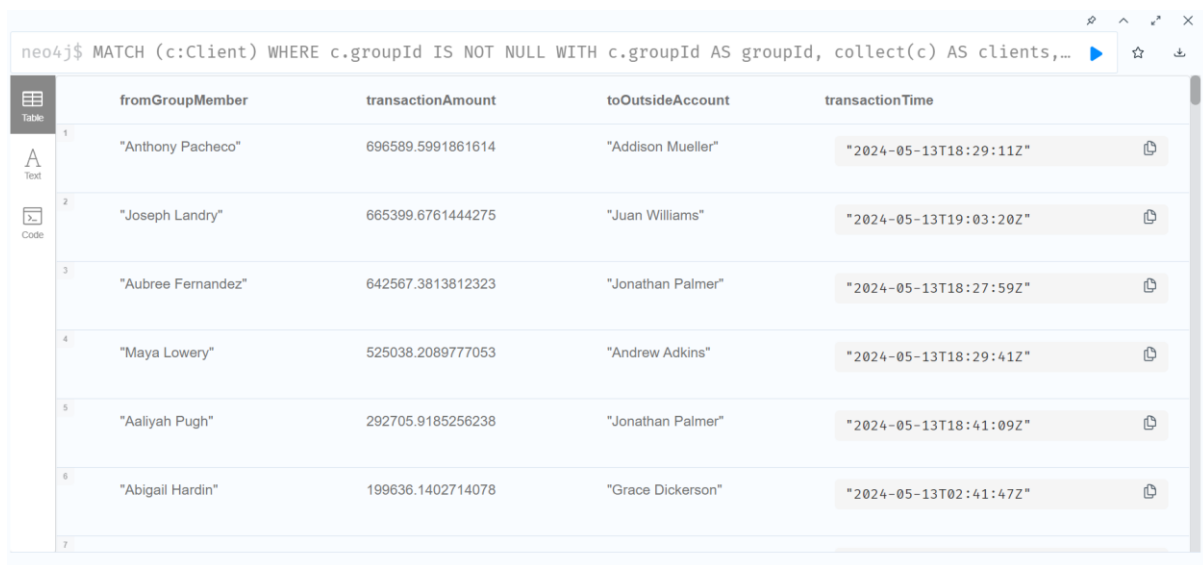
RETURN groupMember.name AS fromGroupMember,

t.amount AS transactionAmount,

outside.name AS toOutsideAccount,

t.transactionDatetime AS transactionTime

ORDER BY transactionAmount DESC;



The image shows a screenshot of the Neo4j Cypher query interface. At the top, a Cypher query is entered: `neo4j$ MATCH (c:Client) WHERE c.groupId IS NOT NULL WITH c.groupId AS groupId, collect(c) AS clients,...`. Below the query, a table of results is displayed. The table has four columns: `fromGroupMember`, `transactionAmount`, `toOutsideAccount`, and `transactionTime`. The results are ordered by `transactionAmount` in descending order. The table contains 6 rows of data, each with a copy icon to the right. At the bottom of the interface, a status bar indicates: `Started streaming 442 records after 32 ms and completed after 40 ms`.

	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"

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

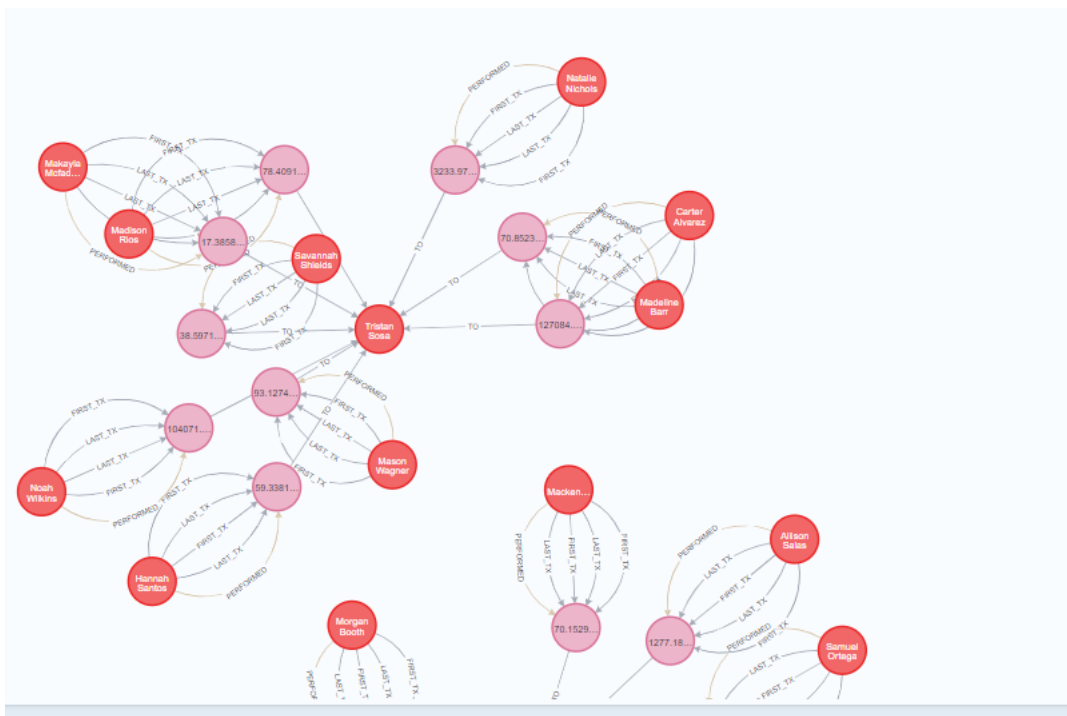
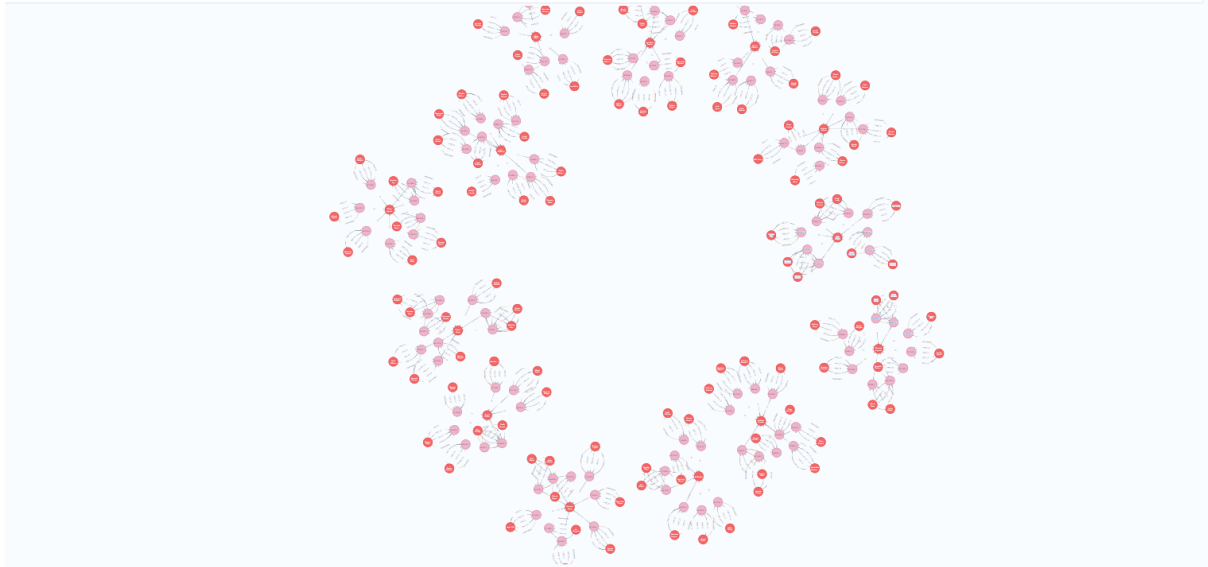
MATCH (groupMember)-[:PERFORMED]->(t:Transaction)-[:TO]->(outside:Client)

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 1: Finding clients in large fraud groups (more than 5 members) MATCH (c:Client) WHERE...



// 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;
```



	graphName	nodeCount	relationshipCount
1	"fraud_movement_projection"	46523	109996

### //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

```
neo4j$ CALL gds.louvain.stream('fraud_movement_projection') YIELD nodeId, communityId WITH gds.util.a...
```

	communityId	groupSize	clientNames
1	375	27	["Faith Wolf", "Morgan Owen", "Jackson Gay", "Melanie Charles", "Evelyn Hines", "Caroline Frederick", "Connor Silva", "Juan Sullivan", "Ca
2	721	26	["Juan Williams", "Christian Dawson", "Olivia Castro", "Anna Fuentes", "Morgan Clark", "Brayden Levine", "William Berry", "Samantha Hamil
3	620	26	["Hannah Byers", "Oliver Sellers", "Katherine Higgins", "Mia Best", "Chase Short", "Bentley Benson", "Riley Wilder", "Evelyn Weeks", "Naom
4	811	23	["Madison Peck", "Alexa Lynch", "Arianna Mercer", "Riley William", "Benjamin Blankenship", "Sarah York", "Joseph York", "Stella Mclaughlin
5	896	23	["Lillian Elliott", "Jose Wright", "Katherine Morales", "Carson Mcneil", "Piper Baird", "Logan Adams", "Evan Becker", "Josiah Key", "Carter Ell
6	155	21	["Alexander Fleming", "Grace Dickerson", "Julia Lawson", "Lucy Blanchard", "Zachary Avery", "Carlos Kemp", "Layla Love", "Aubrey Cooper

Started streaming 165 records after 20 ms and completed after 2187 ms.

```
neo4j$ CALL gds.louvain.stream('fraud_movement_projection') YIELD nodeId, communityId WITH gds.util...
```

	communityId	groupSize	clientNames
	913	5	["Anna Gardner", "Jackson Howard", "Bentley Robles", "Jacob Hartman", "Xavier Savage"]
84	1024	5	["Maya Paul", "Angel Garcia", "Anna Noble", "Nolan Spencer", "Genesis Jensen"]
85	561	4	["Liam Rivas", "Daniel Frank", "Gavin Little", "Damian Craft"]
86	344	4	["Samantha Woodward", "Taylor Lambert", "Victoria Pearson", "Anthony Sellers"]
87	497	4	["Jason Sandoval", "Mackenzie Jackson", "Logan Santos", "Alexa Barnett"]
88	529	3	["John Kirby", "Autumn Clemons", "William Berger"]
89			

iii)

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

CALL gds.louvain.stream('fraud\_movement\_projection')



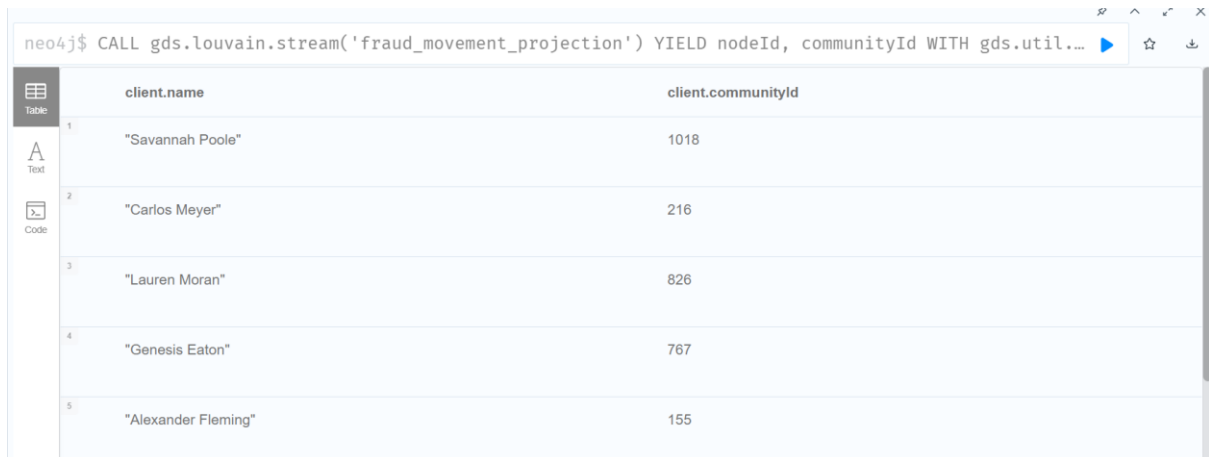
```
YIELD nodeId, communityId

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

SET client.communityId = communityId

RETURN client.name, client.communityId

LIMIT 10;
```



The screenshot shows a Neo4j query result window. The query is: `neo4j$ CALL gds.louvain.stream('fraud_movement_projection') YIELD nodeId, communityId WITH gds.util...`. The result is displayed as a table with two columns: `client.name` and `client.communityId`. The table contains five rows of data, numbered 1 to 5 in the first column.

	client.name	client.communityId
1	"Savannah Poole"	1018
2	"Carlos Meyer"	216
3	"Lauren Moran"	826
4	"Genesis Eaton"	767
5	"Alexander Fleming"	155

// 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
```

neo4j\$ CALL gds.pageRank.stream('fraud\_movement\_projection') YIELD nodeId, score WITH gds.util.asNod...

	clientName	score
1	"Sarah Vazquez"	43.905480630717676
2	"Jonathan Finley"	43.22837916525763
3	"Carlos Lamb"	36.38774267853314
4	"Chloe Anthony"	35.99821759834593
5	"Adrian Murphy"	30.09872162571065
6	"Brandon Frederick"	28.2073902485212
7		

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 neo4j bloom

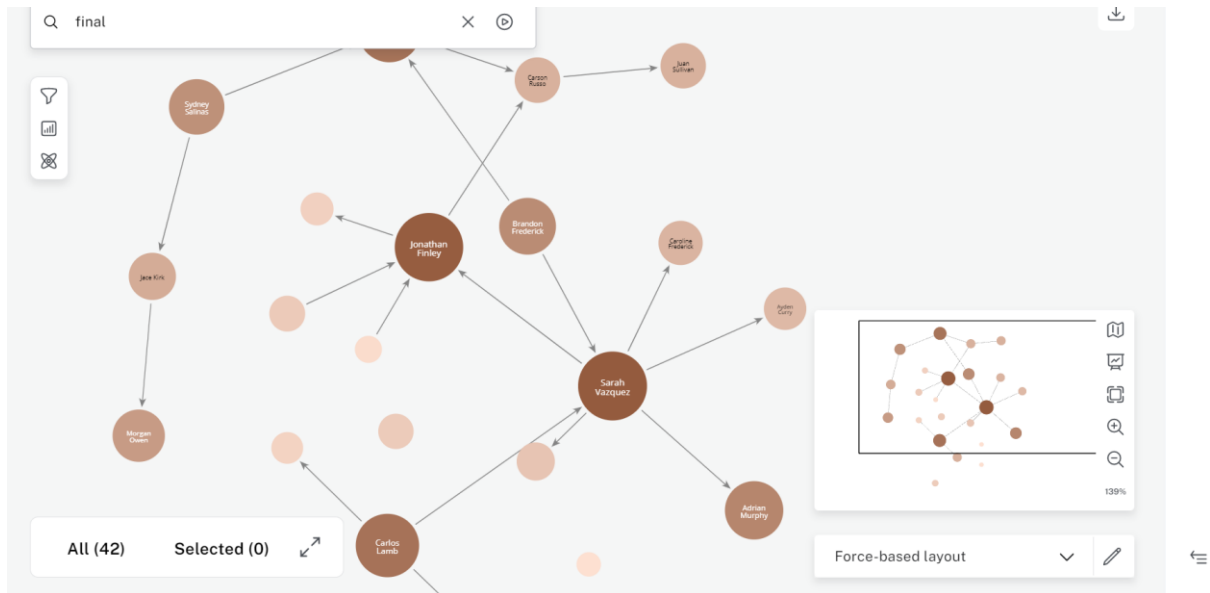
```
MATCH (c:Client)-[p:PERFORMED]->(t:Transaction)-[r:TO]->(outside:Client)
```

```
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).