Eléonor KIOULOU
Khadija MOKHTARI
Théophile NELSON
Mohamed BOUALILI

DIA2 - A5

Dataset, mono/bipartite graph - graph data analysis

dataset:
https://www.kaggle.com/datasets/parisrohan/credit-score-classification?select=test.csv

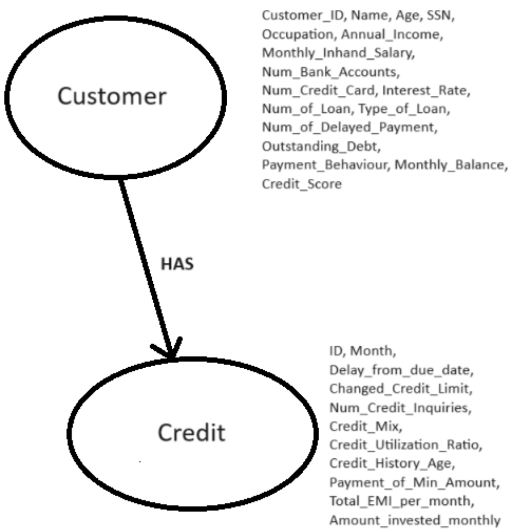
I. Description of the dataset

- **ID:** Unique identifier associated with each record / Object (`object`)
- **Customer_ID:** Unique identifier of the customer / Object (`object`)
- **Month:** Month of the record / Object (`object`)
- **Name:** Customer's name /Object (`object`)
- **Age:** Customer's age / Object (`object`)
- **SSN:** Customer's social security number / Object (`object`)
- **Occupation:** Customer's occupation / Object (`object`)
- **Annual_Income:** Customer's annual income / Object (`object`)
- **Monthly_Inhand_Salary:** Customer's monthly net salary / Float (`float`)
- **Num_Bank_Accounts:** Number of the customer's bank accounts / Integer (`int`)
- **Num_Credit_Card:** Number of the customer's credit cards / Integer (`int`)
- **Interest_Rate:** Interest rate associated with the customer / Integer (`int`)
- **Num_of_Loan:** Number of loans the customer has / Object (`object`)
- **Type_of_Loan:** Types of loans granted to the customer / Object (`object`)
- **Delay_from_due_date:** Delay from the due date / Integer (`int`)
- **Num_of_Delayed_Payment:** Number of delayed payments / Object (`object`)
- **Changed_Credit_Limit:** Modification of the credit limit / Object (`object`)
- **Num_Credit_Inquiries:** Number of credit inquiries / Float (`float`)
- **Credit_Mix:** Mix of credits / Object (`object`)
- **Outstanding_Debt:** Customer's outstanding debt / Object (`object`)
- **Credit_Utilization_Ratio:** Credit utilization ratio / Float (`float`)
- **Credit_History_Age:** Age of the credit history / Object (`object`)
- **Payment_of_Min_Amount:** Payment of the minimum amount / Object (`object`)
- **Total_EMI_per_month:** Total monthly EMI / Float (`float`)
- **Amount_invested_monthly:** Amount invested monthly / Object (`object`)
- **Payment_Behaviour:** Customer's payment behavior / Object (`object`)
- **Monthly_Balance:** Customer's monthly balance / Object (`object`)
- **Credit_Score:** Customer's credit score / Object (`object`)

This dataset provides customer information, including identifiers, financial details, and credit history for analysis.

II. Graph data models

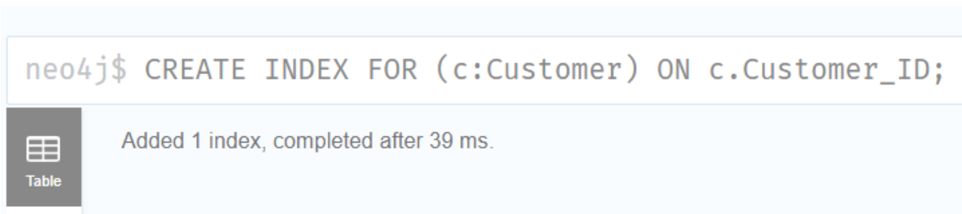
1. Bipartite Graph Model



The separation of the CSV into 'customer' and 'credit' files comes from a strategy using a bipartite graph model. This method makes it easier to show connections between customers and their credits, allowing for more accurate analyses and clearer queries within the context of a bipartite graph.

Index:

```
// Create an index on the Customer_ID property of the Customer node
CREATE INDEX FOR (c:Customer) ON (c.Customer_ID);
```

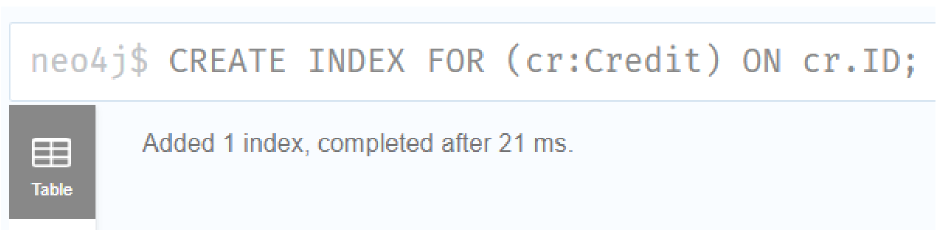


```
neo4j$ CREATE INDEX FOR (c:Customer) ON c.Customer_ID;
```

Table

Added 1 index, completed after 39 ms.

```
// Create an index on the ID property of the Credit node
CREATE INDEX FOR (cr:Credit) ON (cr.ID);
```



```
neo4j$ CREATE INDEX FOR (cr:Credit) ON cr.ID;
```

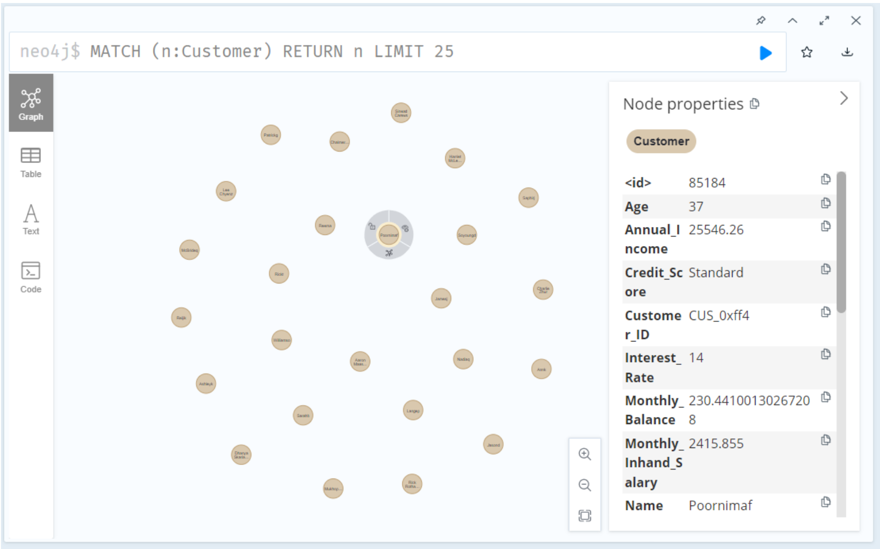
Table

Added 1 index, completed after 21 ms.

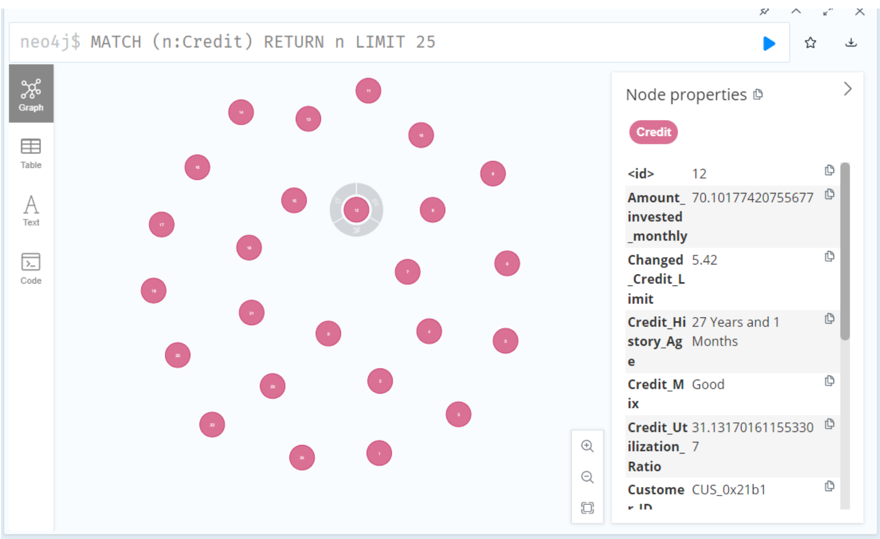
We decided to create these two indices to establish a connection between the customer and their credits.

Import:

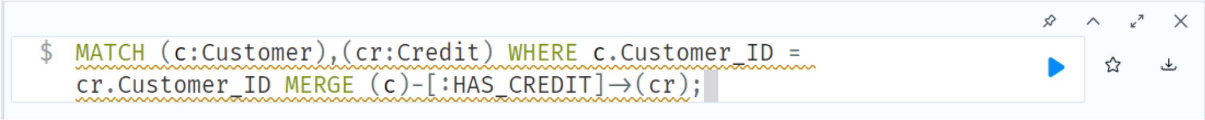
```
// Create nodes for Customers
LOAD CSV WITH HEADERS FROM 'file:/customer_data.csv' AS row
MERGE (c:Customer {Customer_ID: row.Customer_ID})
SET c = row;
```




```
// Create nodes for Credits
LOAD CSV WITH HEADERS FROM 'file:/credit_data.csv' AS row
MERGE (cr:Credit {ID: row.ID, Customer_ID: row.Customer_ID})
SET c = row;
```



```
// Create edges between Customers and Credits using ID
MATCH (c:Customer), (cr:Credit) WHERE c.Customer_ID=cr.Customer_ID
MERGE (c)-[:HAS_CREDIT]->(cr);
```



This script builds a bipartite graph, generating unique `Customer` and `Credit` nodes based on identifiers (`Customer_ID` and `ID`). It establishes clear connections between clients and loans for detailed analyses.

2. Monopartite Graph Model

Index :

```
#Index sur la propriété Customer_ID du nœud Customer
CREATE INDEX FOR (c:Customer) ON (c.Customer_ID);

#Index sur la propriété Occupation du nœud Customer
CREATE INDEX FOR (c:Customer) ON (c.Occupation);

#Index sur la propriété Age du noeud Customer
CREATE INDEX FOR (c:Customer) ON (c.Age);
```

```
neo4j$ CREATE INDEX FOR (c:Customer) ON (c.Age);
```

```
#Index sur la propriété Type_of_Loan du noeud Customer
```

```
neo4j$ CREATE INDEX FOR (c:Customer) ON (c.Type_of_Loan);
```

```
#Index sur la propriété Payment_Behaviour du noeud Customer
```

```
neo4j$ CREATE INDEX FOR (c:Customer) ON (c.Payment_Behaviour);
```

```
#Index sur la propriété Credit_Score du noeud Customer
```

```
neo4j$ CREATE INDEX FOR (c:Customer) ON (c.Credit_Score);
```

```
#Index sur la propriété Num_of_Loan du noeud Customer
```

```
neo4j$ CREATE INDEX FOR (c:Customer) ON (c.Num_of_Loan);
```

For index creation, we analyzed the distribution of our target columns to identify elements that would bring two individuals closer. We operate on the assumption that “Two customers know each other if they share the same occupation, age, annual salary range, and have taken the same type of credit”.

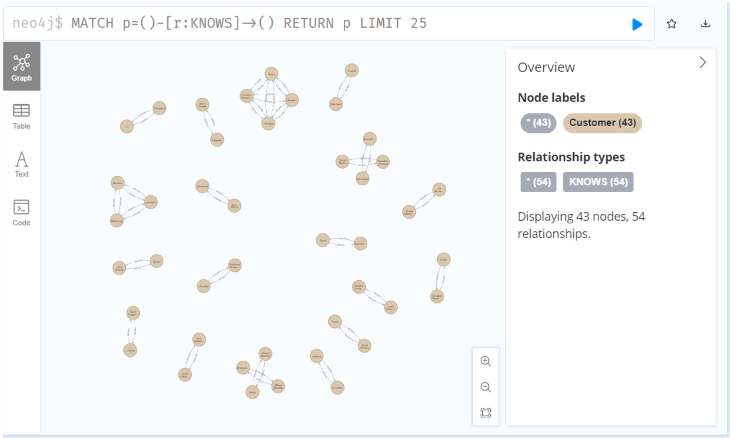
Import :

```
// Create nodes for Customers
LOAD CSV WITH HEADERS FROM 'file:/customer_data.csv' AS row
MERGE (c:Customer {Customer_ID: row.Customer_ID})
SET c = row;
```

In contrast, within the monopartite graph model, all nodes belong to the same set, specifically clients (`Customer`). Edges connect clients who share a common characteristic, like the same occupation. This approach offers a different perspective, exploring relationships among clients themselves based on specific characteristics, revealing groups or trends within this set.

Create relation between customers:

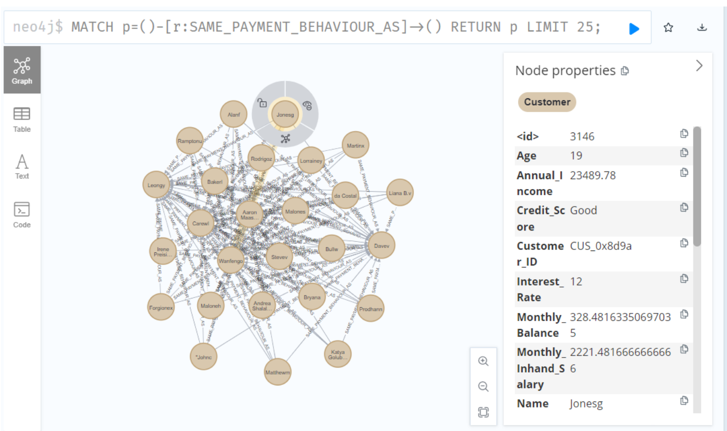
```
MATCH (c1:Customer), (c2:Customer)
WHERE c1.Occupation = c2.Occupation
AND c1.Age = c2.Age
AND c1.Type_of_Loan = c2.Type_of_Loan
AND c1.Customer_ID <> c2.Customer_ID
WITH c1, c2 LIMIT 50000
MERGE (c1)-[:KNOWS {Occupation: c1.Occupation, Age: c1.Age,
Type_of_Loan: c1.Type_of_Loan}]→(c2);
```



They have the same occupation, the same age and the same type of loan credited. We see that they do know each other.

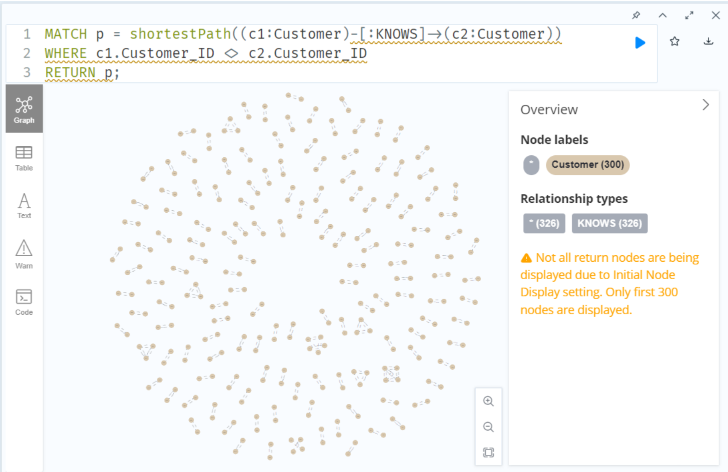
Another type of relation : same "*Payment_Behaviour*" and "*Credit_Score*":

```
MATCH (c1:Customer), (c2:Customer)
WHERE c1.Payment_Behaviour = c2.Payment_Behaviour
AND c1.Credit_Score = c2.Credit_Score
AND c1.Num_of_Loan = c2.Num_of_Loan
AND c1.Customer_ID <> c2.Customer_ID
WITH c1, c2 LIMIT 50000
MERGE (c1)-[:SAME_PAYMENT_BEHAVIOUR_AS {Payment_Behaviour:
c1.Payment_Behaviour, Credit_Score: c1.Credit_Score, Num_of_Loan:
c1.Num_of_Loan}]→(c2);
```

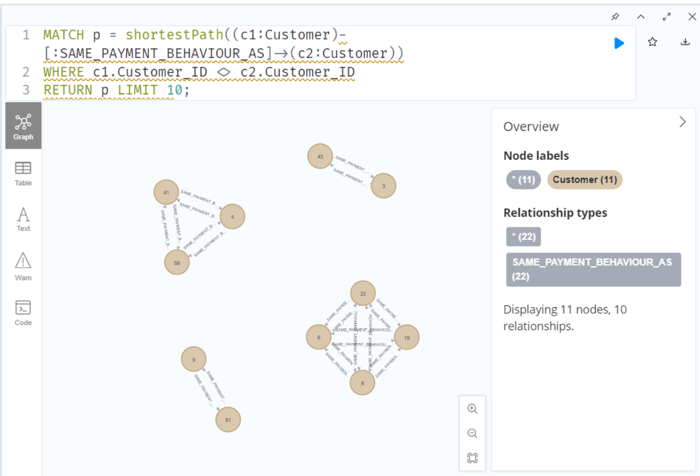


III. GDS queries on graphs Monopartite

1. Pathfinding

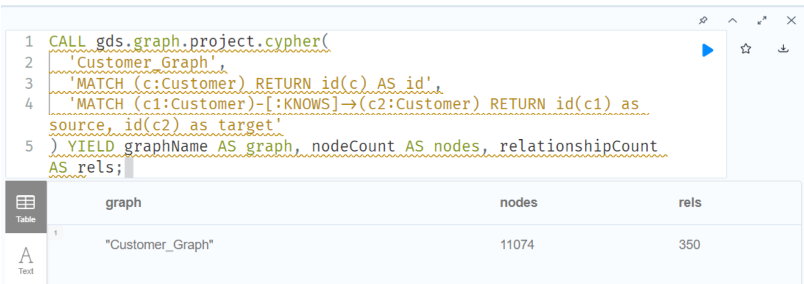


`Shortest_path` on all `Customer_ID`: This query aims to find the shortest path between two clients (`Customer`) using the `KNOWS` relationship as the link between clients. The path length is unspecified, allowing the query to return the shortest path without length restrictions.



This query looks for the shortest path between `c1` and `c2` without specifying the path length, but it will only display up to 10 nodes in the result. Adjust the `LIMIT` value to 10.

Création du graph "`Customer_Graph`"



This query establishes a graph named `'Customer_Graph'` by defining nodes as Customers and relationships as `'KNOWS'` connections between Customers.

2. Communities

```
CALL gds.labelPropagation.write('Customer_Graph', {
  writeProperty: 'communityLabelProp'
});
```

	writeMillis	nodePropertiesWritten	ranIterations	didConverge	communityCount	communityDistributi
1	561	11074	2	true	10909	{ "min": 1, "p5": 1, "max": 4, "p999": 2,

The query assigns labels to nodes in the `'Customer_Graph'` based on how they are connected, helping to find groups of nodes with similar relationships in the graph.

```
neo4j$ CALL gds.louvain.write('Customer_Graph', { writeProperty: 'comm…
```

	writeMillis	nodePropertiesWritten	modularity	modularities	ranLevels	community
1	268	11074	0.9919346938775511	[0.9919346938775511]	1	10909

The Louvain community detection algorithm is applied to the `'Customer_Graph`,' identifying communities within the customer network and assigning a `'communityLouvain'` property to each node.

3. Centralities

```
CALL gds.pageRank.stream('Customer_Graph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).Customer_ID AS customerID, score
ORDER BY score DESC
LIMIT 10;
```

	customerID	score
1	"CUS_0xfcc"	0.9612404689154856
2	"CUS_0xd05"	0.9612404689154856

This query employs the `gds.pageRank.stream` procedure on the '`Customer_Graph`' graph, providing the top 10 nodes along with their PageRank scores.

```
CALL gds.degree.stream('Customer_Graph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).Customer_ID AS customerID, score
ORDER BY score DESC
LIMIT 10;
```

customerID	score
"CUS_0x11b1"	3.0
"CUS_0x56d1"	3.0

This query uses the `gds.degree.stream` procedure on the graph named '`Customer_Graph`' and returns the top 10 nodes with their Degree Centrality scores.

4. Link prediction

```
MATCH (c1:Customer {Customer_ID: 'CUS_0xd40'}), (c2:Customer {Customer_ID: 'CUS_0x21b1'})
RETURN gds.alpha.linkprediction.commonNeighbors(c1, c2, {relationshipQuery: 'KNOWS'}) AS score;
```

score
0.0

The query applies the link prediction algorithm to compute the number of common neighbors between two customers (identified as '`CUS_0xd40`' and '`CUS_0x21b1`') in the graph. It assigns a score based on their shared connections via the '`KNOWS`' relationship. In this instance, there are no common neighbors between these two customers.

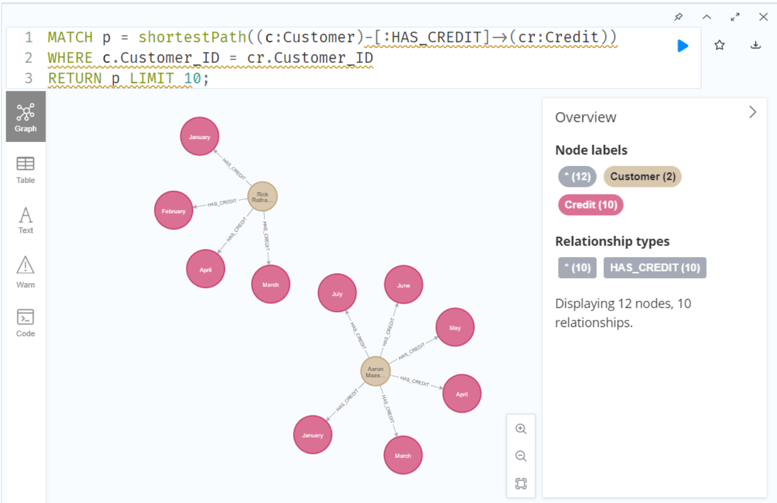
```
MATCH (c1:Customer {Customer_ID: 'CUS_0xd40'}), (c2:Customer {Customer_ID: 'CUS_0x21b1'})
RETURN
  gds.alpha.linkprediction.totalNeighbors(c1, c2, {relationshipQuery: 'KNOWS'}) AS tn,
  gds.alpha.linkprediction.preferentialAttachment(c1, c2, {relationshipQuery: 'KNOWS'}) AS pa,
  gds.alpha.linkprediction.resourceAllocation(c1, c2, {relationshipQuery: 'KNOWS'}) AS ra,
  gds.alpha.linkprediction.adamicAdar(c1, c2, {relationshipQuery: 'KNOWS'}) AS aa;
```

tn	pa	ra	aa
0.0	0.0	0.0	0.0

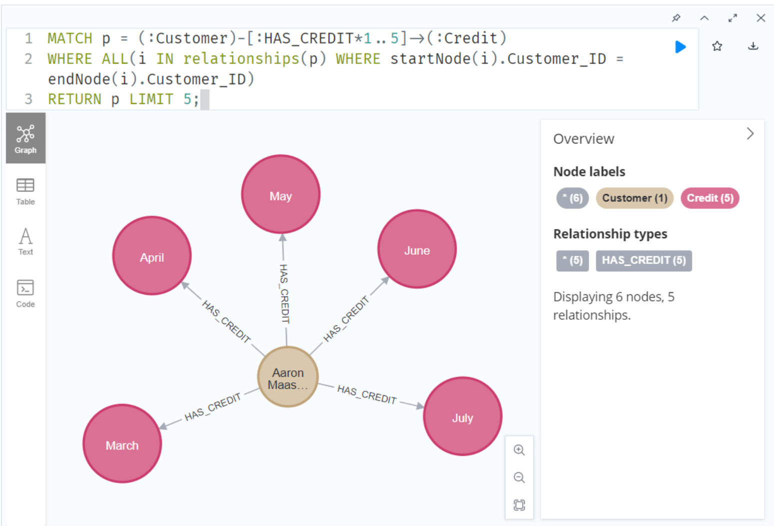
This query evaluates link prediction metrics (total neighbors, preferential attachment, resource allocation, Adamic-Adar) for these two customers, resulting in a score of 0.0. This indicates no connection strength between them as we saw before.

IV. GDS queries on graphs Bipartite

1. pathfinding



This query identifies the shortest paths between Customer and Credit nodes using the `'HAS_CREDIT'` relationship, with the condition that the `Customer_ID` must match that of the Credit node. The results are limited to 10 paths.



This query employs a variable-length path pattern (`[:HAS_CREDIT*1..5]`) and ensures consistency in `Customer_ID` along the entire path. The `LIMIT 5` clause restricts the output to the first 5 paths discovered.

Creation of the graph '`Credit_and_Customer_Graph`':

```
MATCH (c:Customer), (cr:Credit)
WHERE c.Customer_ID = cr.Customer_ID
WITH gds.graph.project( 'Credit_and_Customer_Graph',
  c, cr,
  {
    sourceNodeLabels: labels(c),
    targetNodeLabels: labels(cr),
    relationshipType: 'HAS_CREDIT'
  }
) AS g
RETURN g.graphName AS graph, g.nodeCount AS nodes,
g.relationshipCount AS rels;
```

graph	nodes	rels
"Credit_and_Customer_Graph"	86493	75419

This query will generate a GDS graph named `'Credit_and_Customer_Graph'` with the nodes we labeled as `'Customer'` and `'Credit`,' and relationships of type `'HAS_CREDIT'` we created before for connecting them based on the `Customer_ID`.

2. Communities

```
CALL gds.labelPropagation.write('Credit_and_Customer_Graph', {
  relationshipTypes: ['HAS_CREDIT'],
  writeProperty: 'communityLabel'
});
```

writeMillis	nodePropertiesWritten	ranIterations	didConverge	communityCount	communityDistributi
57	86493	2	true	75419	{ "min": 1, "p5": 1, "max": 2, "p999": 2,

This query uses the `gds.labelPropagation.write` procedure to apply the Label Propagation algorithm to the graph. The results are stored in a property named 'communityLabel' on nodes. The subsequent query retrieves and displays the community labels assigned to each node.

```
CALL gds.louvain.write('Credit_and_Customer_Graph', {
  relationshipTypes: ['HAS_CREDIT'],
  writeProperty: 'communityLabel'
});
```

writeMillis	nodePropertiesWritten	modularity	modularities
544	86493	0.99990771331908	[0.1467407394795989, 0.29357376564011545, 0.4403

This query employs the `gds.labelPropagation.write` procedure to apply the Label Propagation algorithm to the graph. The outcomes are saved in a property named '`communityLabel`' on nodes. The subsequent query retrieves and displays the community labels assigned to each node.

3. Centralities

```
CALL gds.pageRank.write('Credit_and_Customer_Graph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  writeProperty: 'pagerank'
});
```

writeMillis	nodePropertiesWritten	ranIterations	didConverge	centralityDistribution	postP
924	86493	2	true	{ "min": 0.14999961853027344, "max":	130

This query first computes the PageRank centrality for nodes in the '`Credit_and_Customer_Graph`' using the `gds.pageRank.write` procedure. The result is stored in a property called '`pagerank`'.

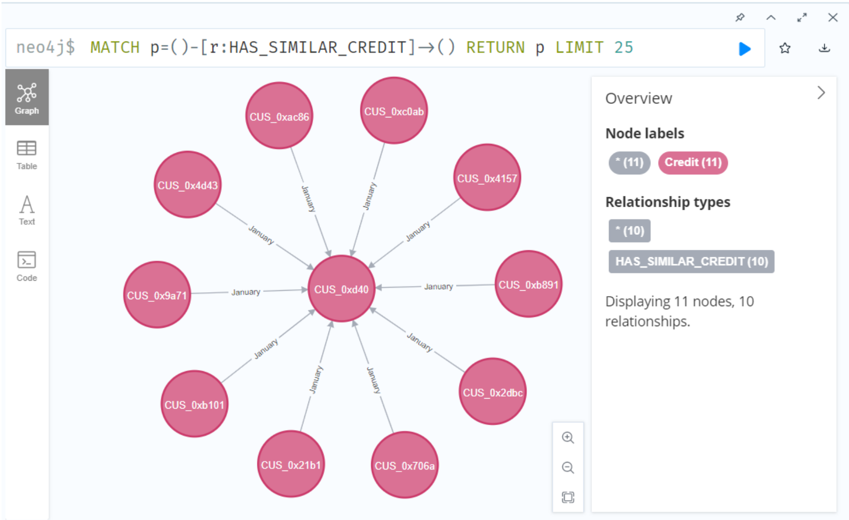
```
CALL gds.betweenness.write('Credit_and_Customer_Graph', {
  writeProperty: 'betweenness'
});
```

nodePropertiesWritten	writeMillis	centralityDistribution	postProcessingMillis	preProcessingMillis	c
86493	638	{ "min": 0.0, "max":	21	0	6

This query calculates the Betweenness Centrality for nodes in the '`Credit_and_Customer_Graph`' using the `gds.betweenness.write procedure`. The result is stored in a property named '`betweenness`'.

4. Link prediction

Creation of a new relationship

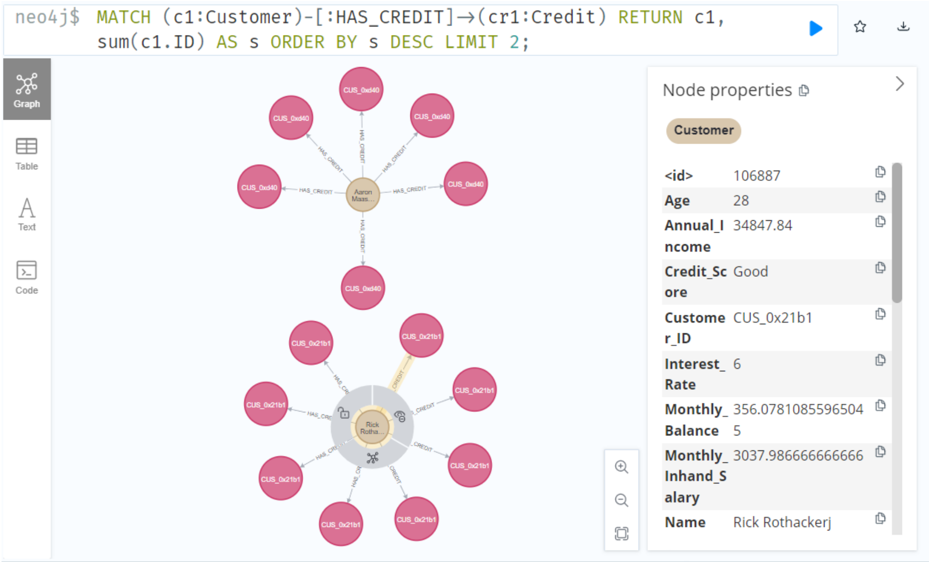


We established this relationship to perform the `'linkprediction'` function on it. The relationship links clients who took out a loan in the same `'month'` and share the same '`payment behavior`'. They are connected by the '`HAS_SIMILAR_CREDIT`' relationship.

```
MATCH (c1:Customer {Customer_ID: 'CUS_0×21b1'})-[:HAS_CREDIT]→(cr1:Credit)
MATCH (c2:Customer {Customer_ID: 'CUS_0×d40'})-[:HAS_CREDIT]→(cr2:Credit)
WITH c1, c2, gds.alpha.linkprediction.commonNeighbors(c1, c2, {
  relationshipQuery: 'HAS_SIMILAR_CREDIT'
}) AS commonNeighbors
RETURN c1.Customer_ID, c2.Customer_ID, commonNeighbors LIMIT 1;
```

	c1.Customer_ID	c2.Customer_ID	commonNeighbors
1	"CUS_0x21b1"	"CUS_0xd40"	0.0

There are no common neighbors between these two customers. Indeed, we have done clusters on the Month of creation of the credit which leads to having no relations between customers because the only relation that we will have is between customers who have done a credit during the same month.



We focused on the top 2 customers who have the highest number of credits, so we repeated the same query, correcting the relationship query to '`has_credit`'.

```
MATCH (c1:Customer {Customer_ID: 'CUS_0×21b1'})-[:HAS_CREDIT]→(cr1:Credit)
MATCH (c2:Customer {Customer_ID: 'CUS_0×d40'})-[:HAS_CREDIT]→(cr2:Credit)
WITH c1, c2, gds.alpha.linkprediction.totalNeighbors(c1, c2, {
  relationshipQuery: 'HAS_CREDIT'
}) AS totalNeighbors
RETURN c1.Customer_ID, c2.Customer_ID, totalNeighbors LIMIT 1;
```

	c1.Customer_ID	c2.Customer_ID	totalNeighbors
1	"CUS_0x21b1"	"CUS_0xd40"	14.0

The total number of neighbors is equal to 14. Our query works but as our previous relation was not relevant we would have a score of 0.