

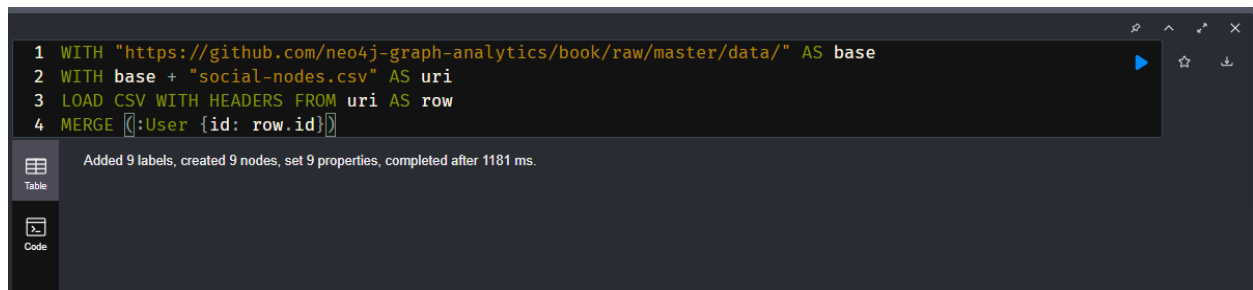
Probar códigos Neo4j



4.2.2 Importación de datos. Creación de grafos

- Importación de grafo

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:User {id: row.id})
```



```
1 WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
2 WITH base + "social-nodes.csv" AS uri
3 LOAD CSV WITH HEADERS FROM uri AS row
4 MERGE (:User {id: row.id})
```

Added 9 labels, created 9 nodes, set 9 properties, completed after 1181 ms.

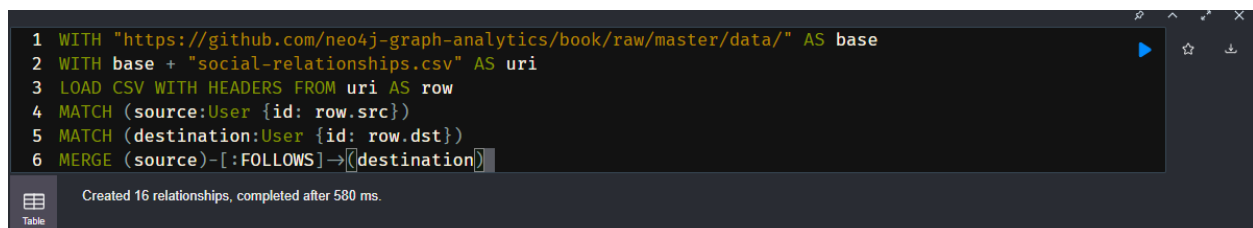
Table

Code

4.2.2 Importación de datos. Creación de grafos

- Importación de grafo

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```



```
1 WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
2 WITH base + "social-relationships.csv" AS uri
3 LOAD CSV WITH HEADERS FROM uri AS row
4 MATCH (source:User {id: row.src})
5 MATCH (destination:User {id: row.dst})
6 MERGE (source)-[:FOLLOWS]->(destination)
```

Created 16 relationships, completed after 580 ms.

Table

4.2.2 Importación de datos. Creación de grafos

- Añadir nueva conexión

```
MATCH (src {id: "David"})
MATCH (dst {id: "Mark"})
CREATE (src)-[:FOLLOWS]->(dst)
```

```
1 MATCH (src {id: "David"})
2 MATCH (dst {id: "Mark"})
3 CREATE (src)-[:FOLLOWS]->(dst)
```

Created 1 relationship, completed after 5 ms.

4.2.2 Importación de datos. Creación de grafos

- Borrar la conexión

```
MATCH (src {id: "David"})-[rel:FOLLOWS]->(dst {id: "Mark"})
DELETE rel
```

```
1 MATCH (src {id: "David"})-[rel:FOLLOWS]->(dst {id: "Mark"})
2 DELETE rel
```

Deleted 1 relationship, completed after 10 ms.

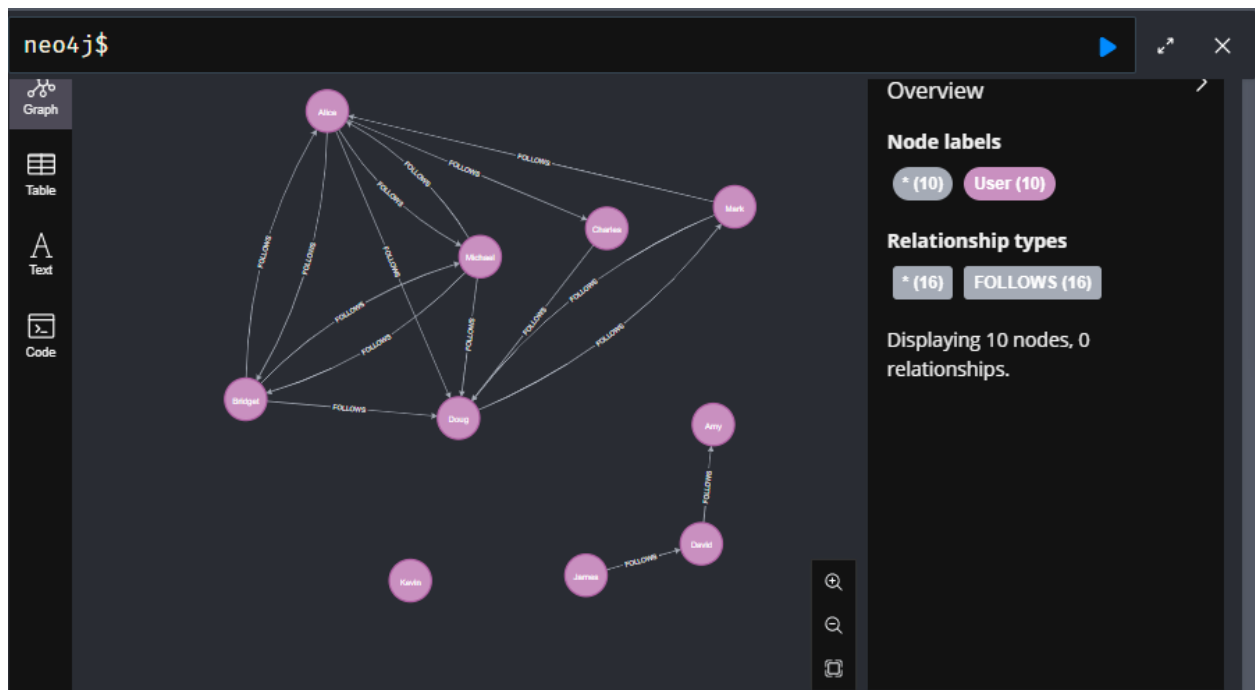
4.2.2 Importación de datos. Creación de grafos

- Añadir nuevo nodo

```
CREATE (:User {id: "Kevin"})
```

```
neo4j$ CREATE (:User {id: "Kevin"})
```

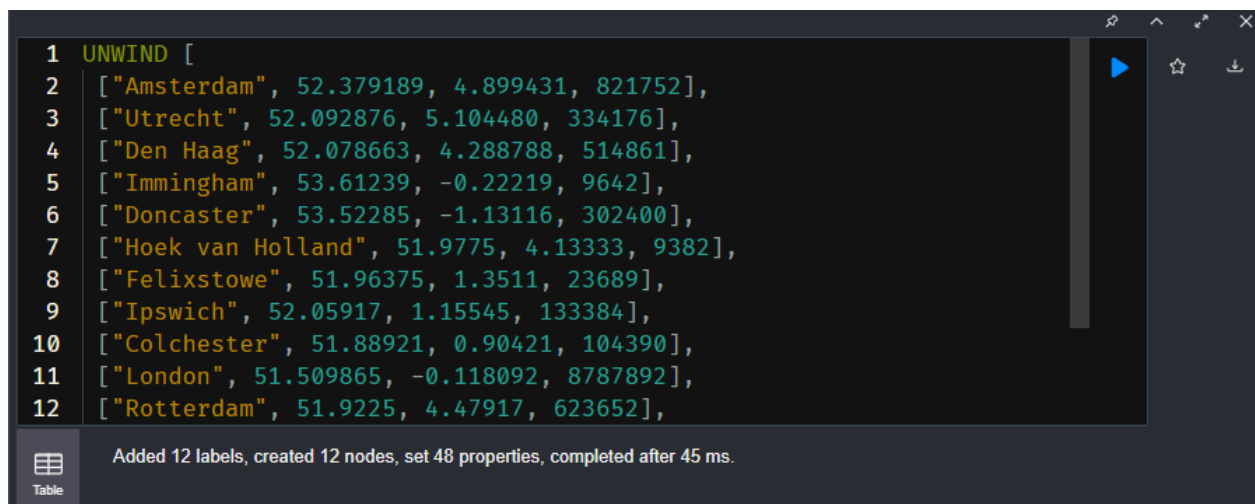
Added 1 label, created 1 node, set 1 property, completed after 3 ms.



4.2.2 Importación de datos. Creación de grafos

- Añadir lista de nodos → en lotes

```
UNWIND [  
  ["Amsterdam", 52.379189, 4.899431, 821752],  
  ["Utrecht", 52.092876, 5.104480, 334176],  
  ["Den Haag", 52.078663, 4.288788, 514861],  
  ["Immingham", 53.61239, -0.22219, 9642],  
  ["Doncaster", 53.52285, -1.13116, 302400],  
  ["Hoek van Holland", 51.9775, 4.13333, 9382],  
  ["Felixstowe", 51.96375, 1.3511, 23689],  
  ["Ipswich", 52.05917, 1.15545, 133384],  
  ["Colchester", 51.88921, 0.90421, 104390],  
  ["London", 51.509865, -0.118092, 8787892],  
  ["Rotterdam", 51.9225, 4.47917, 623652],  
  ["Gouda", 52.01667, 4.70833, 70939]  
] AS data  
CREATE (:City {id: data[0], latitude: data[1], longitude: data[2], population: data[3]})
```



```
1 UNWIND [  
2   ["Amsterdam", 52.379189, 4.899431, 821752],  
3   ["Utrecht", 52.092876, 5.104480, 334176],  
4   ["Den Haag", 52.078663, 4.288788, 514861],  
5   ["Immingham", 53.61239, -0.22219, 9642],  
6   ["Doncaster", 53.52285, -1.13116, 302400],  
7   ["Hoek van Holland", 51.9775, 4.13333, 9382],  
8   ["Felixstowe", 51.96375, 1.3511, 23689],  
9   ["Ipswich", 52.05917, 1.15545, 133384],  
10  ["Colchester", 51.88921, 0.90421, 104390],  
11  ["London", 51.509865, -0.118092, 8787892],  
12  ["Rotterdam", 51.9225, 4.47917, 623652],  
]
```

Added 12 labels, created 12 nodes, set 48 properties, completed after 45 ms.

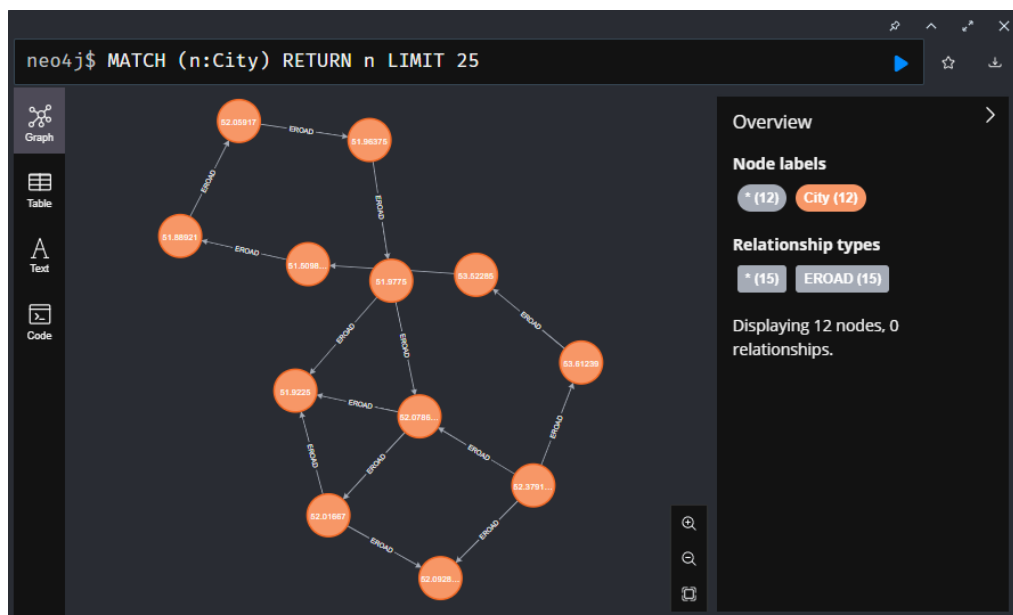
4.2.2 Importación de datos. Creación de grafos

- Añadir relaciones → en lotes

```
UNWIND [  
  ["Amsterdam","Utrecht","EROAD",46],  
  ["Amsterdam","Den Haag","EROAD",59],  
  ["Den Haag","Rotterdam","EROAD",26],  
  ["Amsterdam","Immingham","EROAD",369],  
  ["Immingham","Doncaster","EROAD",74],  
  ["Doncaster","London","EROAD",277],  
  ["Hoek van Holland","Den Haag","EROAD",27],  
  ["Felixstowe","Hoek van Holland","EROAD",207],  
  ["Ipswich","Felixstowe","EROAD",22],  
  ["Colchester","Ipswich","EROAD",32],  
  ["London","Colchester","EROAD",106],  
  ["Gouda","Rotterdam","EROAD",25],  
  ["Gouda","Utrecht","EROAD",35],  
  ["Den Haag","Gouda","EROAD",32],  
  ["Hoek van Holland","Rotterdam","EROAD",33]  
] AS data  
MATCH (src:City {id: data[0]}), (dst:City {id: data[1]})  
CREATE (src)-[:EROAD {cost: (data[3])}]->(dst)
```

```
1 UNWIND [  
2  ["Amsterdam","Utrecht","EROAD",46],  
3  ["Amsterdam","Den Haag","EROAD",59],  
4  ["Den Haag","Rotterdam","EROAD",26],  
5  ["Amsterdam","Immingham","EROAD",369],  
6  ["Immingham","Doncaster","EROAD",74],  
7  ["Doncaster","London","EROAD",277],  
8  ["Hoek van Holland","Den Haag","EROAD",27],  
9  ["Felixstowe","Hoek van Holland","EROAD",207],  
10 ["Ipswich","Felixstowe","EROAD",22],  
11 ["Colchester","Ipswich","EROAD",32],  
12 ["London","Colchester","EROAD",106],  
13 ] AS data  
14 MATCH (src:City {id: data[0]}), (dst:City {id: data[1]})  
15 CREATE (src)-[:EROAD {cost: (data[3])}]->(dst)
```

Set 15 properties, created 15 relationships, completed after 49 ms.



4.2.3 Recorridos sobre grafos. BFS recorrido en anchura



```
CALL gds.graph.create('myGraph', 'City', 'EROAD', {relationshipProperties: 'cost'}) //creación del grafo llamado "myGraph"
YIELD graphName, nodeCount, relationshipCount, createMillis; // asignamos los valores a las variables
```

```
MATCH (a:City {id: 'Doncaster'}) WITH id(a) AS startNode
CALL gds.alpha.bfs.stream('myGraph', { startNode: startNode }) // realizamos el recorrido BFS -> nº de caminos
YIELD path UNWIND [n in nodes(path) | n.id] AS tags // asignamos resultados
RETURN tags; // mostramos datos
```

El resultado del recorrido es: Doncaster, London, Colchester, Ipswich, Felixstowe, Hoek van Holland, Den Haag, Rotterdam, Gouda, Utrecht.

```
1 CALL gds.graph.create('myGraph', 'City', 'EROAD', {relationshipProperties: 'cost'}) //creación del grafo llamado "myGraph"
2 YIELD graphName, nodeCount, relationshipCount, createMillis;
```

	graphName	nodeCount	relationshipCount	createMillis
1	"myGraph"	12	30	53

```
1 MATCH (a:City {id: 'Doncaster'}) WITH id(a) AS startNode
2 CALL gds.alpha.bfs.stream('myGraph', { startNode: startNode }) // realizamos el recorrido BFS -> nº de caminos
3 YIELD path UNWIND [n in nodes(path) | n.id] AS tags // asignamos resultados
4 RETURN tags;
```

	tags
1	"Doncaster"
2	"London"
3	"Colchester"
4	"Ipswich"
5	"Felixstowe"
6	"Hoek van Holland"
7	

4.2.4 Caminos mínimos

En caso de querer obtener el camino mínimo entre un par de nodos concretos, es posible utilizar el código mostrado a continuación:

```
// Define la ciudad de origen y destino
```

```
WITH "Amsterdam" AS startCity, "London" AS endCity
```

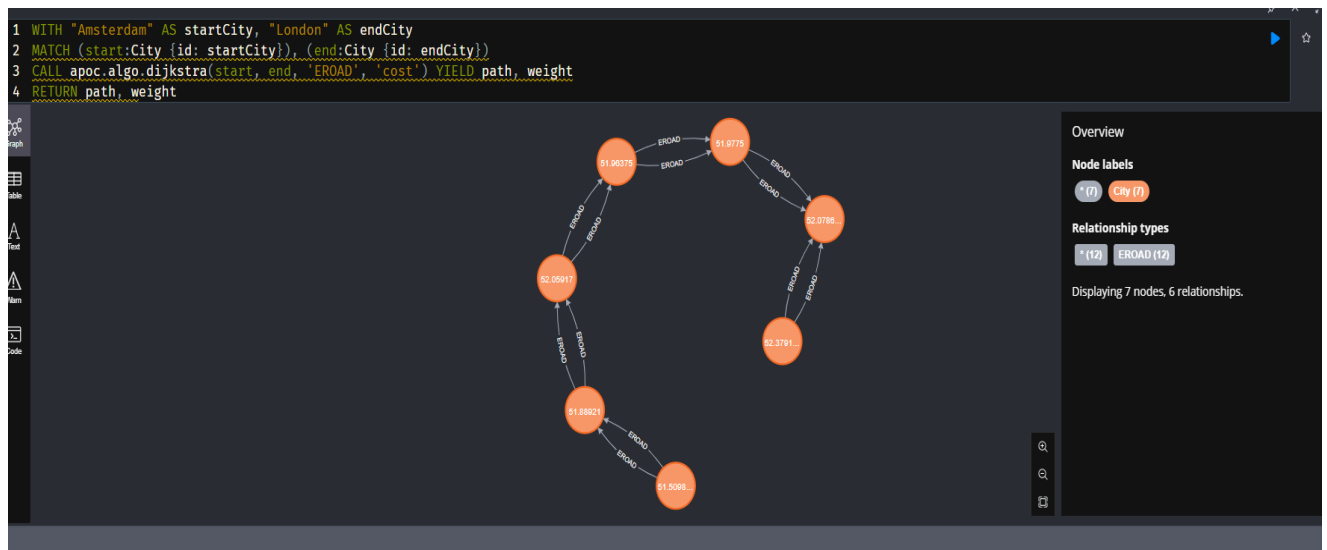
```
// Encuentra los nodos correspondientes a las ciudades de origen y destino
```

```
MATCH (start:City {id: startCity}), (end:City {id: endCity})
```

```
// Ejecuta el algoritmo de Dijkstra para encontrar el camino mínimo
```

```
CALL apoc.algo.dijkstra(start, end, 'EROAD', 'cost') YIELD path, weight
```

```
RETURN path, weight
```



4.2.5 Medidas de centralidad

Respecto a las medidas de centralidad, **no existe una única medida sino que, en función de los datos y del propósito que se pretende alcanzar, se utilizan unas métricas u otras**. A continuación, se van a definir tres medidas de centralidad con las que se realizarán ejemplos en Neo4j: centralidad de grado, cercanía e intermediación.

Para trabajar con medidas de centralidad, se va a importar un grafo social que servirá de ejemplo para el cálculo de estas métricas. Este es el código necesario para su creación.

```
LOAD CSV WITH HEADERS FROM
'https://raw.githubusercontent.com/neo4j-graph-analytics/book/master/data/social-nodes.csv' AS row
MERGE (:User {id: row.id})

LOAD CSV WITH HEADERS FROM
'https://raw.githubusercontent.com/neo4j-graph-analytics/book/master/data/social-relationships.csv'
AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```

```
1 LOAD CSV WITH HEADERS FROM
2 'https://raw.githubusercontent.com/neo4j-graph-analytics/book/master/data/social-nodes.csv' AS row
3 MERGE (:User {id: row.id})
```

Added 9 labels, created 9 nodes, set 9 properties, completed after 571 ms.

Code

```
1 LOAD CSV WITH HEADERS FROM
2 'https://raw.githubusercontent.com/neo4j-graph-analytics/book/master/data/social-relationships.csv'
3 AS row
4 MATCH (source:User {id: row.src})
5 MATCH (destination:User {id: row.dst})
6 MERGE (source)-[:FOLLOWS]->(destination)
7
```

Created 16 relationships, completed after 147 ms.

Table

Code

4.2.5 Medidas de centralidad

Centralidad de un grado

Este código cuenta el número de relaciones `FOLLOWS` que tiene cada nodo `User` y lo utiliza para calcular su centralidad de grado. Luego, devuelve los resultados ordenados de mayor a menor centralidad.

```
MATCH (u:User)-[:FOLLOWS]->(other:User)
WITH u, count(*) AS degree
RETURN u.id AS UserId, degree AS DegreeCentrality
ORDER BY DegreeCentrality DESC
```

`CALL apoc.node.degree(['User'], ['FOLLOWS'])`: Utiliza la función `apoc.node.degree` para calcular la centralidad de grado de los nodos → error en el uso de la función

.....

`ORDER BY DegreeCentrality DESC`: Ordena los resultados en orden descendente según la centralidad de grado, de modo que los nodos más centrales aparezcan primero en la tabla de resultados.

*** error en el uso de la función***

```
CALL apoc.node.degree(['User'], ['FOLLOWS'])
YIELD node, degree
RETURN node.id AS UserId, degree AS
DegreeCentrality
ORDER BY DegreeCentrality DESC
```

```
1 MATCH (u:User)-[:FOLLOWS]->(other:User)
2 WITH u, count(*) AS degree
3 RETURN u.id AS UserId, degree AS DegreeCentrality
4 ORDER BY DegreeCentrality DESC
5
```

	UserId	DegreeCentrality
1	"Alice"	4
2	"Bridget"	3
3	"Michael"	3
4	"Mark"	2
5	"Charles"	1
6	"Doug"	1

```
1 CALL apoc.node.degree(['User'], ['FOLLOWS'])
2 YIELD node, degree
3 RETURN node.id AS UserId, degree AS
4 DegreeCentrality
5 ORDER BY DegreeCentrality DESC
6
7
```

ERROR Neo.ClientError.Procedure.ProcedureNotFound
There is no procedure with the name 'apoc.node.degree' registered for this database instance. Please ensure you've spelled the procedure name correctly and that the procedure is properly deployed.
List available procedures

4.2.5 Medidas de centralidad

Centralidad de un grado

Cercanía

La cercanía es una medida de centralidad que determina qué nodos del grafo expanden rápida y eficientemente la información a través del grafo. Para calcular esta medida, se obtiene la suma del inverso de las distancias de un nodo al resto.

La cercanía es una métrica muy utilizada para estimar tiempos de llegada en redes logísticas, para descubrir actores en posiciones privilegiadas en redes sociales o para estudiar la prominencia de palabras en un documento en el campo de la minería de textos.

```
CALL gds.alpha.closeness.stream({
  nodeProjection: 'User',
  relationshipProjection: {
    FOLLOWS: {
      type: 'FOLLOWS',
      orientation: 'UNDIRECTED' // Opcional, ajusta según tu
                                // modelo de datos
    }
  }
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).id AS UserId, centrality
ORDER BY centrality DESC
```

En el grafo de la imagen, Alice, Doug y David tienen una cercanía de 1.

```
1 CALL gds.alpha.closeness.stream({
2   nodeProjection: 'User',
3   relationshipProjection: {
4     FOLLOWS: {
5       type: 'FOLLOWS',
6       orientation: 'UNDIRECTED'
7     }
8   }
9 })
10 YIELD nodeId, centrality
11 RETURN gds.util.asNode(nodeId).id AS UserId, centrality
12 ORDER BY centrality DESC
```

	UserId	centrality
1	"Alice"	1.0
2	"Doug"	1.0
3	"David"	1.0
4	"Bridget"	0.7142857142857143
5	"Michael"	0.7142857142857143
6	"Amy"	0.6666666666666666
7		

4.2.5 Medidas de centralidad

Centralidad de un grado

Intermediación

La intermediación es una medida de centralidad que permite detectar la influencia que tiene un nodo o actor del grafo en el flujo de información o de recursos de la red. El cálculo de la intermediación permite identificar a nodos que hacen de puentes entre distintas porciones del grafo. Esta medida de centralidad es muy utilizada para la identificación de influencers y el estudio de la viralización de mensajes en redes sociales. Intuitivamente, la intermediación de un nodo será mayor en tanto en cuanto dicho nodo aparezca en los caminos mínimos de cualquier otro par de nodos.

```
// Crear el grafo 'myGraph'  
CALL gds.graph.create(  
  'myGraph',  
  'User',  
  {  
    FOLLOWS: {  
      orientation: 'REVERSE'  
    }  
  }  
)
```

```
CALL gds.betweenness.stream('myGraph')  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).id AS name,  
score  
ORDER BY score DESC
```

```
1  
2 CALL gds.graph.create(  
3   'myGraph',  
4   'User',  
5   {  
6     FOLLOWS: {  
7       orientation: 'REVERSE'  
8     }  
9   }  
10  )
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>{ "User": { "properties": { }, "label": "User" } }</pre>	<pre>{ "FOLLOWS": { "orientation": "REVERSE", "aggregation": "DEFAULT", "type": "FOLLOWS", "properties": { } } }</pre>	myGraph	9	16	4

```
1  
2 CALL gds.betweenness.stream('myGraph')  
3 YIELD nodeId, score  
4 RETURN gds.util.asNode(nodeId).id AS name,  
5 score  
6 ORDER BY score DESC
```

name	score
"Alice"	10.0
"Doug"	7.0
"Mark"	7.0
"David"	1.0
"Bridget"	0.0
"Charles"	0.0

Started streaming 9 records after 7 ms and completed after 17 ms.

4.2.6 Detección de comunidades

Conteo de triángulos

Un triángulo es un conjunto de tres nodos que tienen relaciones entre sí. El conteo de triángulos para un nodo dado, permite estudiar o inspeccionar de forma global un grafo y, aplicado sobre componentes conexas, permite inspeccionar regiones de un grafo.

Para aplicar este método, es necesario almacenar el grafo teniendo en cuenta que las aristas deben almacenarse como UNDIRECTED.

La ejecución de este método da como resultado que Alice y Doug son aquellos que pertenecen a más triángulos, un total de 5.

```
CALL gds.graph.create(
  'myGraph2',
  'User',
  {
    FOLLOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

```
CALL gds.triangleCount.stream('myGraph2')
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).id AS name,
triangleCount
ORDER BY triangleCount DESC
```

```
1
2 CALL gds.graph.create(
3   'myGraph2',
4   'User',
5   {
6     FOLLOWS: {
7       orientation: 'UNDIRECTED'
8     }
9   }
10 )
```

	nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
1	{ "User": { "properties": { }, "label": "User" } }	{ "FOLLOWS": { "orientation": "UNDIRECTED", "aggregation": "DEFAULT", "type": "FOLLOWS", "properties": { } } }	"myGraph2"	9	32	3

Started streaming 1 records after 6 ms and completed after 12 ms.

```
1
2 CALL gds.triangleCount.stream('myGraph2')
3 YIELD nodeId, triangleCount
4 RETURN gds.util.asNode(nodeId).id AS name,
5 triangleCount
6 ORDER BY triangleCount DESC
```

	name	triangleCount
1	"Alice"	6
2	"Doug"	5
3	"Bridget"	4
4	"Michael"	4
5	"Charles"	1
6	"Mark"	1
7		

4.2.6 Detección de comunidades

```
CALL gds.localClusteringCoefficient.stream('myGraph2')
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).id AS name, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Por último, el coeficiente global de clustering se calcula como la suma normalizada de los coeficientes de clustering locales. Así, estos coeficientes nos permiten encontrar medidas cuantitativas para detectar comunidades, pudiendo especificar incluso un umbral para establecer la comunidad (por ejemplo, especificar que los nodos han de estar conectados en un 40 %).

```
1 CALL gds.localClusteringCoefficient.stream('myGraph2')
2 YIELD nodeId, localClusteringCoefficient
3 RETURN gds.util.asNode(nodeId).id AS name, localClusteringCoefficient
4 ORDER BY localClusteringCoefficient DESC
```

	name	localClusteringCoefficient
1	"Charles"	1.0
2	"Bridget"	0.4
3	"Michael"	0.4
4	"Doug"	0.3333333333333333
5	"Mark"	0.3333333333333333
6	"Alice"	0.2857142857142857
7		

Started streaming 9 records after 6 ms and completed after 19 ms.

4.2.6 Detección de comunidades

Componentes fuertemente conexos

En un grafo dirigido, una componente fuertemente conexa es aquel grupo de nodos en el que cualquier nodo puede ser alcanzado por cualquier otro en ambas direcciones. El estudio de las componentes fuertemente conexas en un grafo permite estudiar la conectividad de la red.

nodes: Cantidad total de nodos en el grafo.
communityCount: Cantidad de componentes fuertemente conexas encontradas en el grafo.
setCount: Cantidad total de conjuntos dentro de las componentes conexas.
minSetSize: Tamaño mínimo de un conjunto dentro de las componentes conexas.
maxSetSize: Tamaño máximo de un conjunto dentro de las componentes conexas.

```
CALL gds.alpha.scc.write({
  nodeProjection: 'User',
  relationshipProjection: {
    FOLLOWS: {
      type: '*',
      orientation: 'UNDIRECTED'
    }
  },
  writeProperty: 'componentId'
})
```



	createMillis	computeMillis	writeMillis	postProcessingMillis	nodes	communityCount	setCount	minSetSize	maxSetSize	p1	p5	p10	p25	p50	p75	p90	p95	p99	p100	writeProperty
	3	0	2	8	9	2	2	3	6	3	3	3	3	3	6	6	6	6	6	"componentId"

4.2.7 Predicción de enlaces

Vecinos comunes: El resultado de este cálculo es 2.

```
MATCH (x:User {id: 'Charles'})
MATCH (y:User {id: 'Bridget'})
RETURN gds.alpha.linkprediction.commonNeighbors(x, y) AS score
```



	score
1	2.0

4.2.7 Predicción de enlaces

Adhesión preferencial

Este método se basa en la idea general de que cuanto más conectado está un nodo, es más probable que reciba nuevos enlaces.

En el grafo social de ejemplo, el cálculo de adhesión preferencial para Charles y Bridget se puede realizar a través del siguiente código, el resultado de este cálculo es 10.

```
MATCH (x:User {id: 'Charles'})
MATCH (y:User {id: 'Bridget'})
RETURN gds.alpha.linkprediction.preferentialAttachment(x, y) AS score
```

```
1 MATCH (x:User {id: 'Charles'})
2 MATCH (y:User {id: 'Bridget'})
3 RETURN gds.alpha.linkprediction.preferentialAttachment(x, y) AS score
4
5
```

	score
1	10.0

4.2.7 Predicción de enlaces

Asignación de recursos

Se trata de una métrica compleja que evalúa la cercanía de un par de nodos para determinar la posibilidad de que, entre ellos, se produzca un nuevo enlace. En el grafo social de ejemplo, el cálculo de la asignación de recursos para Charles y Bridget se puede realizar a través del siguiente código, el resultado de este cálculo es 0.309.

```
MATCH (x:User {id: 'Charles'})
MATCH (y:User {id: 'Bridget'})
RETURN gds.alpha.linkprediction.resourceAllocation(x, y) AS score
```

```
1 MATCH (x:User {id: 'Charles'})
2 MATCH (y:User {id: 'Bridget'})
3 RETURN gds.alpha.linkprediction.resourceAllocation(x, y) AS score
4
```

	score
1	0.30952380952380953