

## Apunts CE\_5074 3.1

lloc: [Institut d'Ensenyaments a Distància de les Illes Balears](#)

Curs: Sistemes de Big Data

Llibre: Apunts CE\_5074 3.1

Imprès per: Carlos Sanchez Recio

Data: dimarts, 26 de novembre 2024, 09:07

# Taula de continguts

## 1. Introducció

## 2. Grafs

## 3. Àmbits d'aplicació

## 4. Grafs de propietats etiquetats

## 5. Introducció a Neo4j

### 5.1. Instal·lació

### 5.2. Sandbox

### 5.3. Creació del graf

### 5.4. Consultes bàsiques

### 5.5. Modificacions

### 5.6. Esborrats

### 5.7. Consultes avançades

### 5.8. Procediments

### 5.9. Carregar un graf des de fitxers CSV

## 6. Problemes específics de grafs amb Neo4j

### 6.1. Recorreguts del graf

### 6.2. Cerca de camins mínims

### 6.3. Estudi de centralitat

### 6.4. Detecció de comunitats

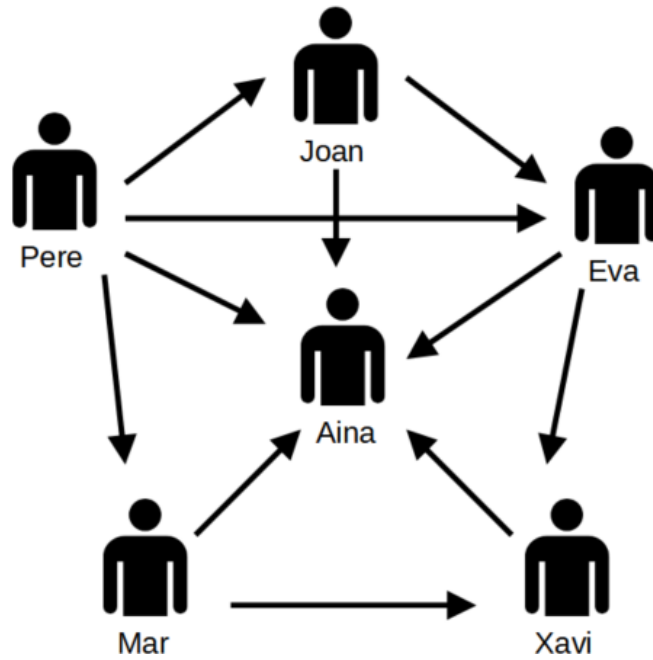
### 6.5. Predicció d'enllaços

## 1. Introducció

En el Lliurament 1 vàrem veure què són les bases de dades NoSQL i vàrem introduir els principals models en què es basen: clau-valor, basat en documents, columnar (o *wide-column*) i basat en grafs. Vàrem aprofundir en el model basat en documents i vàrem treballar amb MongoDB, el gestor NoSQL més emprat. En aquest lliurament ens centrarem en el model basat en grafs.

Aquest model utilitza l'estructura de dades graf per a representar les dades i les relacions entre elles, mitjançant nodes i arestes. És, per tant, una aproximació molt diferent a la resta de models NoSQL.

La imatge següent mostra el graf d'una petita xarxa social, on els nodes representen els usuaris i les arestes indiquen qui segueix a qui en la xarxa social.



Imatge: Exemple de graf

Amb aquesta representació podem arribar a respondre preguntes senzilles com "a qui segueix Pere?", "qui segueix a Aina?", però també altres una mica més complexes com "podem arribar des de Joan fins a Xavi?", "qui és l'usuari més influent del grup?" o "existeixen comunitats diferenciades dins del grup?".

Amazon Neptune, Apache Giraph, ArangoDB o Neo4j són alguns dels gestors basats en grafs més coneguts. En aquest lliurament ens centrarem en Neo4j, un gestor de codi obert molt potent.

En el següent vídeo pots veure un resum del que tractarem en aquest lliurament.

### SBD L3: Bases de dades basades en grafs



**Vídeo:** *Resum del Lliurament 3*

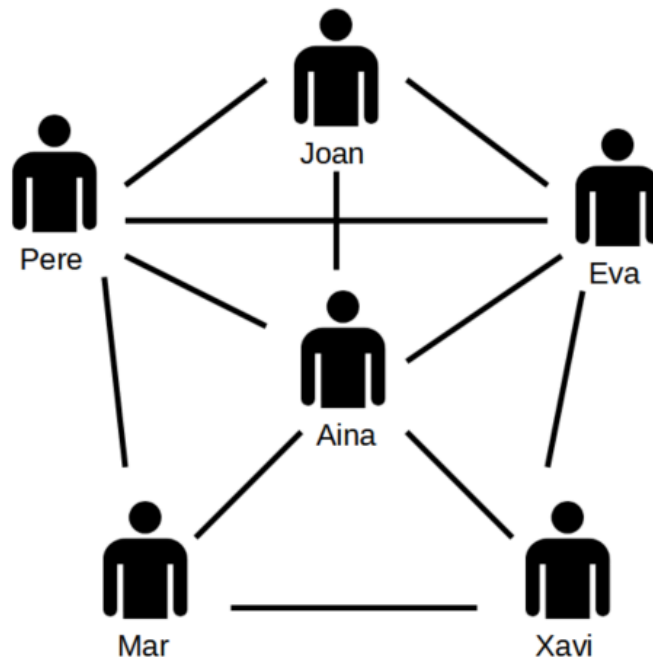
## 2. Grafs

Encara que existeixen diferents tipus de models de dades basats en grafs, cada sistema de base de dades triarà un d'ells per a representar el contingut de la base de dades. Qualsevol d'aquests models està inspirat en l'estructura matemàtica d'un **graf** per a modelitzar un conjunt de dades.

Matemàticament, un graf és un conjunt de **nodes** o **vèrtexs**, que representen entitats d'un domini, i un conjunt d'**arestes** o **enllaços**, que representen les **relacions** o interaccions que es produeixen entre els nodes.

Una característica important d'un graf és el fet que les seves arestes indiquin o no una direcció. En l'exemple que hem vist abans, la relació que tenim és "segueix a" i sí que indica una direcció. Per exemple, Pere segueix a Joan en la xarxa social, però Joan no segueix a Pere. El graf anterior és el que s'anomena un **graf dirigit**.

En canvi, altres vegades la relació no té direcció. Per exemple, el següent graf representa un grup de persones i diu qui coneix personalment a qui. Si Pere coneix a Joan, aleshores necessàriament Joan també coneix a Pere. Per això les arestes d'aquest graf no tenen direcció. Un graf com aquest s'anomena **graf no dirigit**.



Imatge: Exemple de graf no dirigit

Hi ha diverses maneres d'emmagatzemar la informació d'un graf. Per exemple, podem transformar-lo a un model relacional, on els nodes es representen mitjançant taules i les relacions entre ells mitjançant claus forànies. Aquesta és l'aproximació que segueixen alguns gestors de bases de dades relacionals que afegixen una capa per donar suport a operacions amb grafs. El problema d'aquestes solucions no natives és que quan el graf creix, són necessaris un gran nombre de *joins* i d'accessos als índexs de les taules, cosa que fa que el rendiment baixi molt. És per això que s'han dissenyat gestors que, de forma nativa, emmagatzemen l'estructura pròpia del graf, mitjançant nodes i arestes.

D'altra banda, pel que fa al processament de les dades, també podem trobar enfocaments natius i no natius. En un model amb un processament natiu, els algorismes s'apliquen directament sobre l'estructura graf i "naveguen" per les connexions entre nodes d'una manera molt eficient. En canvi, en un processament no natiu, es fan servir estructures paral·leles per implementar les operacions típiques. Per exemple, un algorisme molt conegut de la teoria de grafs per a cercar camins mínims és el de Floyd, que genera una matriu precalculada amb el cost i itinerari entre qualsevol parell de nodes del graf. Això fa que, tot i que el procés per construir la matriu és molt costós, una vegada ja la tenim calculada, recuperar el camí entre dos nodes és immediat (perquè està precalculat). El problema, de nou, és l'escalabilitat: mantenir actualitzada aquesta matriu precalculada arriba a ser extremadament costós, fins i tot inviable, quan treballem amb grafs amb una alta freqüència d'actualització i un gran volum de dades, amb milers o milions de nodes i arestes.

És per això que s'han popularitzat les **bases de dades amb emmagatzematge i processament natiu**, ja que ofereixen una solució flexible i escalable. Neo4j, amb el qual treballarem en aquest lliurament, és un d'aquests gestors de grafs natis.

### 3. Àmbits d'aplicació

Les característiques de les bases de dades orientades a grafs tenen una infinitat d'aplicacions reals en les quals el seu ús millora significativament el que ofereixen les tecnologies tradicionals. A continuació, es mostren alguns exemples d'aplicació on, actualment, aquest tipus de bases de dades és el més àmpliament utilitzat.

#### ■ *Xarxes socials*

L'anàlisi de xarxes socials és, avui dia, una de les principals fonts d'informació de qualsevol domini. L'anàlisi de xarxes socials mitjançant bases de dades orientades a grafs permet identificar relacions explícites i implícites entre usuaris i grups d'usuaris, així com identificar la forma en la qual aquests interactuen, podent inferir el comportament d'un usuari sobre la base de les seves connexions.

En una xarxa social, dos usuaris presenten una connexió explícita si estan directament connectats. Això ocorre, per exemple, entre dos usuaris de Facebook que són amics o dos companys de treball de la mateixa empresa en LinkedIn. D'altra banda, una relació implícita és aquella que es produeix entre dos usuaris a través d'un intermediari, com pot ser un altre usuari amb el qual els dos estan relacionats, un post on els dos han fet un comentari, un *like* o un article que tots dos han comprat.

#### ■ *Sistemes de recomanació*

Aquests sistemes permeten modelar en forma de graf les relacions que s'estableixen entre persones o usuaris i coses, com poden ser productes, serveis, contingut multimèdia o qualsevol altre concepte rellevant en funció del domini d'aplicació.

Les relacions s'estableixen en funció del comportament dels usuaris en comprar, consumir contingut, puntuar-ho o avaluar-ho etc.

D'aquesta manera, els sistemes de recomanació identifiquen recursos d'interès per a un usuari específic i poden predir el seu comportament en comprar un producte o contractar un servei.

#### ■ *Informació geogràfica*

Es tracta del cas d'aplicació més evident de la teoria de grafs. Les aplicacions de les bases de dades orientades a grafs en informació geogràfica van des de calcular rutes òptimes entre dos punts en qualsevol tipus de xarxa (xarxa de carreteres, de ferrocarril, aèria, logística...), fins a trobar tots els punts d'interès en una àrea concreta, trobar el centre d'una regió o obtenir la intersecció entre dues o més regions, entre moltes altres.

Així, les bases de dades orientades a grafs permeten modelar aquests casos d'aplicació com a grafs dirigits sobre els quals operar per a obtenir els resultats desitjats.

#### ■ *Altres*

Encara que en aquest apartat s'hagin analitzat les principals àrees d'aplicació, són molts els dominis d'aplicació on les bases de dades orientades a grafs s'estan convertint en una solució predominant, com ho són la **gestió de dades mestres**, **gestió de xarxes i centres de dades** o **control d'accés**, entre molts d'altres.

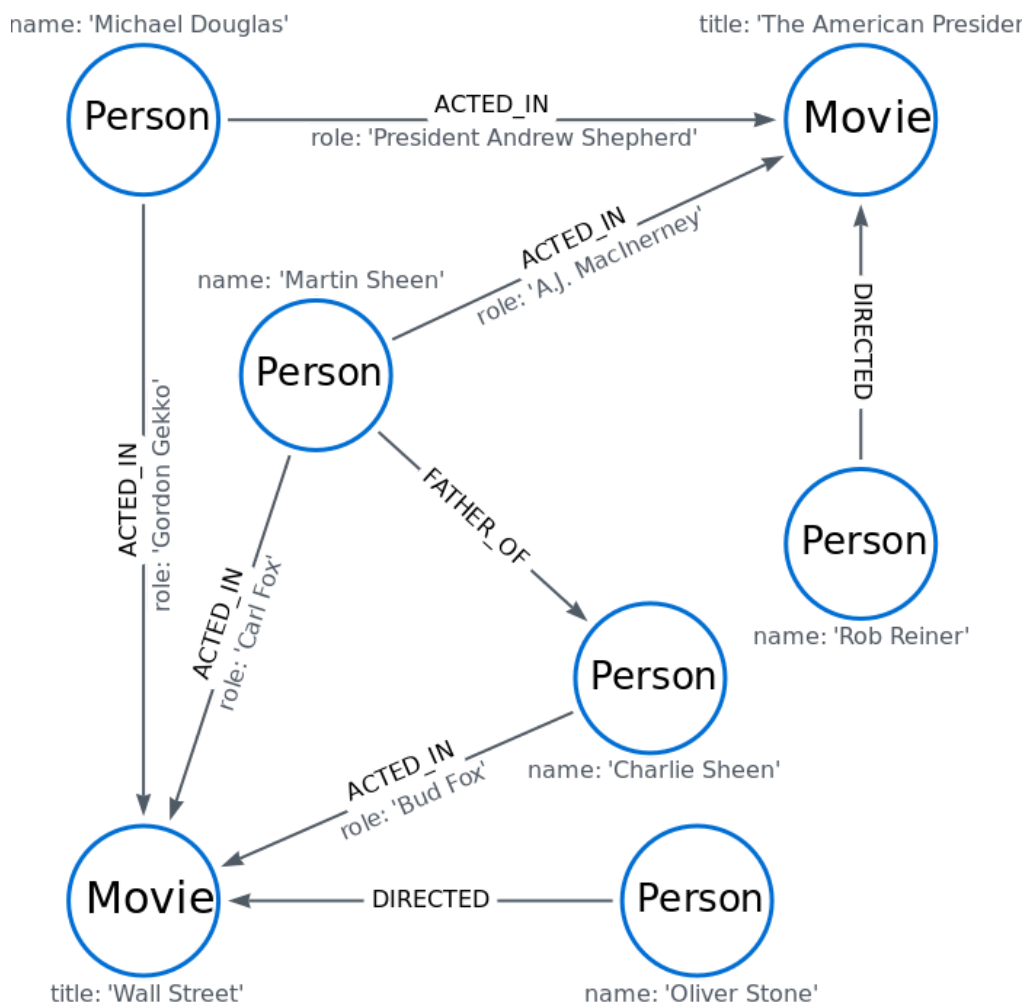
## 4. Grafs de propietats etiquetats

Un dels models de dades basat en grafs més popular és el model de **graf de propietats etiquetat** o **labeled property graph (LPG)**. Un graf representat mitjançant aquest model ha de complir les següents característiques principals:

1. Ha de contenir un conjunt de **nodes** i **arestes** (o **relacions** entre aquests nodes)
2. Els nodes contenen **propietats**, definides a través de parells *clau-valor*
3. Els nodes poden estar **etiquetats** amb una o més etiquetes
4. Les relacions entre els nodes estan **nomenades** (tenen un nom) i són **dirigides**, tenint sempre un node d'inici i un altre node de fi
5. Les relacions del graf també poden contenir propietats.

Aquest model és senzill d'entendre i permet modelar qualsevol problema i/o conjunt de dades. En concret, aquest és el model que fa servir Neo4j.

Per entendre-ho millor, vegem un exemple d'un graf de propietats etiquetat:



**Imatge:** Exemple de graf etiquetat de propietats. Font: documentació de Neo4j (<https://neo4j.com/docs>)

Podem veure que aquest graf té 7 nodes. N'hi ha de dos tipus, persona i pel·lícula. Empram una etiqueta (*Person* o *Movie*) per caracteritzar cada un dels nodes. Veim també que els nodes tenen propietats, parells clau-valor. Els nodes amb etiqueta *Movie* tenen una propietat anomenada *title*, que té els valor '*Wall Street*' (parell *title*: '*Wall Street*') i '*The American President*' (parell *title*: '*The American President*') respectivament. D'altra banda, els nodes



amb etiqueta *Person* tenen una propietat *name*, amb els valors '*Michael Douglas*', '*Martin Sheen*', '*Charlie Sheen*', '*Rob Reiner*' i '*Oliver Stone*', respectivament.

D'altra banda, veim que els nodes estan connectats per relacions o arestes, que tenen un nom que representa el tipus de relació. Per exemple, tenim una aresta amb etiqueta *FATHER\_OF*, que va del node de Martin Sheen al node de Charlie Sheen, ja que el primer és el pare del segon. També podem trobar, per exemple, una aresta que indica que Oliver Stone és el director de la pel·lícula Wall Street. I per últim, veim que hi ha diverses arestes amb nom *ACTED\_IN*, que a més afegixen una propietat *role*, per expressar quin paper representa un determinat actor en una determinada pel·lícula. Per exemple, podem veure que Martin Sheen actua en la pel·lícula Wall Street amb el paper de Carl Fox.

## 5. Introducció a Neo4j

Neo4j és un gestor de bases de dades orientat a grafs que proporciona un emmagatzematge i processament natiu, és a dir, que les dades s'emmagatzemen directament mitjançant estructures de tipus graf i les operacions de consulta s'implementen directament sobre aquestes estructures. Neo4j també permet treballar amb models de dades en graf de manera transaccional, raó per la qual també pot ser emprat en sistemes transaccionals. Tot això permet que Neo4j sigui una solució molt potent per a poder treballar en entorns reals, proporcionant un alt rendiment i escalabilitat.

A més, Neo4j té suport per a múltiples llenguatges, incloent Python, i ofereix tot un ventall d'eines i llibreries per als usuaris i desenvolupadors, orientades a diversos usos i aplicacions. En concret, Neo4j, permet crear models d'aprenentatge automàtic sobre grafs. Aquestes són algunes de les eines principals:

- **Graph Data Science Library:** Es tracta de la llibreria principal que conté una gran quantitat d'algorismes de cerca sobre grafs a més de models d'aprenentatge automàtic. Es tracta d'algorismes i models eficientment programats i escalables per al treball amb grafs.
- **Neo4j Bloom:** És una aplicació de visualització i exploració de grafs que permet la visualització d'aquests des de diferents perspectives i punts de vista, oferint així una eina visual molt útil per a visualitzar els resultats dels algorismes així com mostrar resultats a clients.
- **Cypher:** És el llenguatge de creació de consultes utilitzat en Neo4j. Es tracta d'un llenguatge inspirat en SQL, tot i que més senzill i optimitzat per al treball amb grafs.
- **Integradors:** Amb l'objectiu d'integrar i connectar Neo4j amb altres plataformes i llenguatges, s'han desenvolupat diferents connectors per a integrar aquesta plataforma amb tecnologies com a Apache Spark, Apache Kafka, eines de Business Intelligence, ...
- **Eines per a desenvolupadors:** Neo4j disposa de versions per a escriptori (Neo4j Desktop) i web (Neo4j Browser) així com un sandbox (Neo4j Sandbox) en el qual es proporciona un entorn controlat i integrat dels serveis Neo4j per a desenvolupadors.
- **Neo4j Aura:** Neo4j ofereix un servei de computació en el núvol per a la creació de grafs i execució d'algorismes i models sobre ells. Es tracta de Neo4j Aura, el qual es troba disponible de manera gratuïta i està integrat amb Google Cloud Platform.

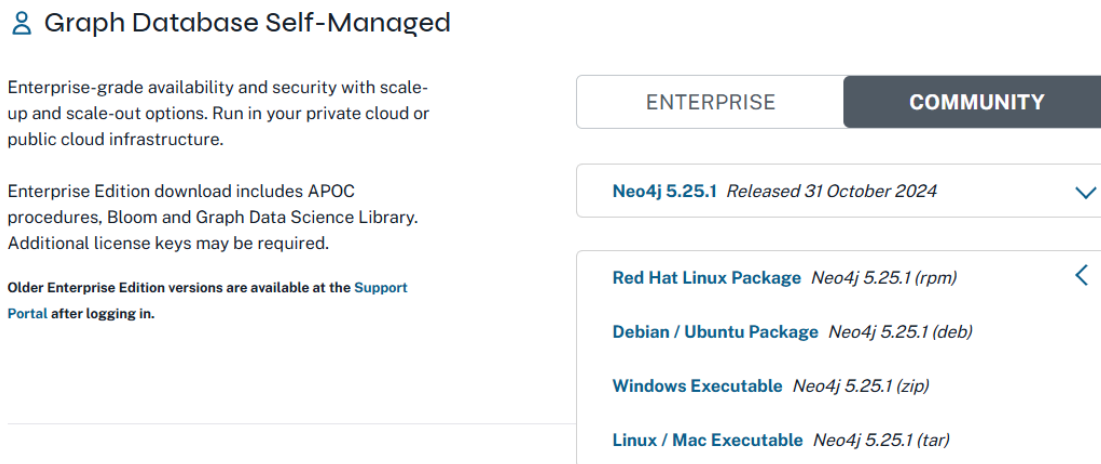
## 5.1. Instal·lació

Per instal·lar Neo4j de manera local, hem de seguir les passes següents.

Si treballem amb Linux, el més senzill és configurar els repositoris per poder fer la instal·lació amb yum o apt, seguint aquestes instruccions:

- Debian/Ubuntu (apt): <https://debian.neo4j.com/>
- RedHat (yum): <https://yum.neo4j.com/>

En canvi, si volem treballar amb Windows, en primer lloc hem de descarregar la darrera versió disponible de la versió de comunitat de Neo4j (Graph Database Self-Managed) des del Depolyment Center de Neo4j: <https://neo4j.com/deployment-center/>



**Graph Database Self-Managed**

Enterprise-grade availability and security with scale-up and scale-out options. Run in your private cloud or public cloud infrastructure.

Enterprise Edition download includes APOC procedures, Bloom and Graph Data Science Library. Additional license keys may be required.

Older Enterprise Edition versions are available at the [Support Portal](#) after logging in.

**ENTERPRISE** **COMMUNITY**

**Neo4j 5.25.1** Released 31 October 2024

**Red Hat Linux Package** Neo4j 5.25.1 (rpm)

**Debian / Ubuntu Package** Neo4j 5.25.1 (deb)

**Windows Executable** Neo4j 5.25.1 (zip)

**Linux / Mac Executable** Neo4j 5.25.1 (tar)

**Imatge:** Descàrrega de Neo4j. <https://neo4j.com/deployment-center/>

Abans de fer la instal·lació, però, necessitam tenir instal·lat Java, almenys la versió 17, ja sigui [OpenJDK](#) o [Oracle Java](#).

Una vegada tenim Java instal·lat i descarregat l'arxiu zip, l'hem de descomprimir i deixar a un directori permanent, per exemple C:\neo4j-community-5.13.0 o C:\neo4j, al qual ens referirem com <NEO4J\_HOME>.

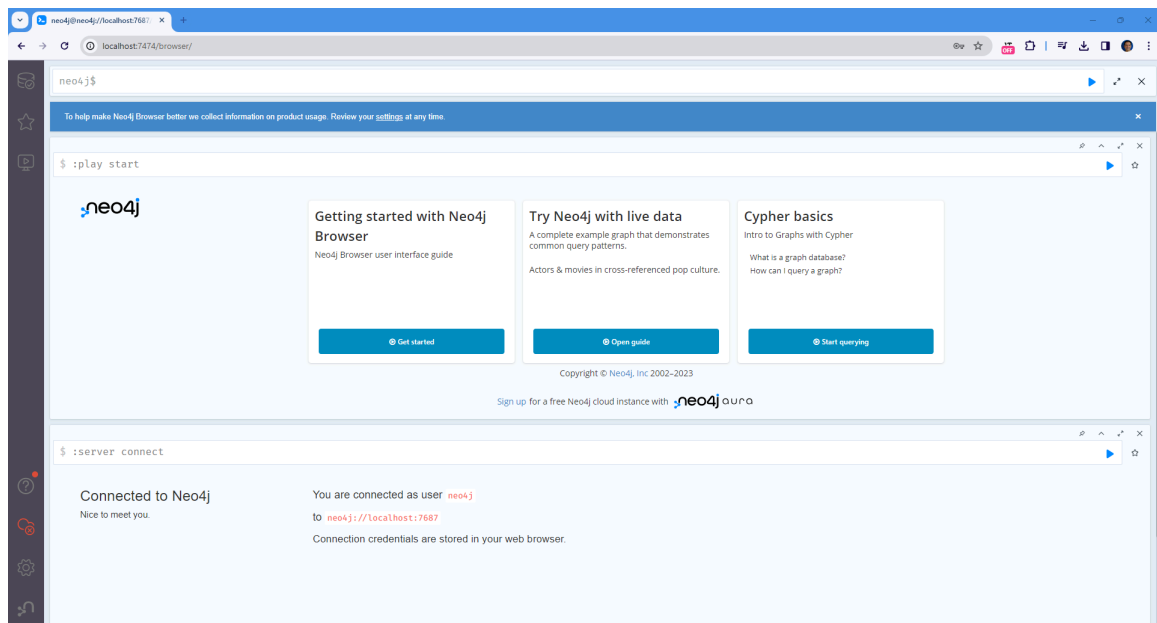
Ara podem executar Neo4j des de la consola:

```
<NEO4J_HOME>\bin\neo4j console
```

O bé instal·lar-lo com un servei:

```
<NEO4J_HOME>\bin\neo4j install-service
```

En qualsevol dels casos, tant en Linux com en Windows, podem accedir mitjançant un navegador a Neo4j Browser, la interfície web de Neo4j, des de <http://localhost:7474>, emprant l'usuari **neo4j** i contrasenya **neo4j**. La primera vegada que hi entrem, ens demanarà canviar la contrasenya.

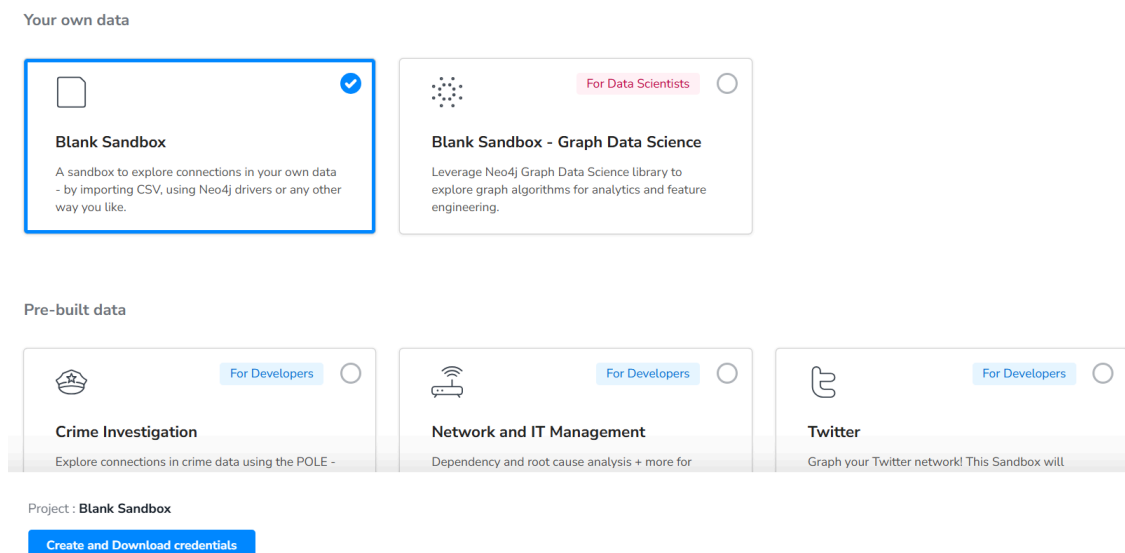


Imatge: Interfície web de Neo4j Browser

## 5.2. Sandbox

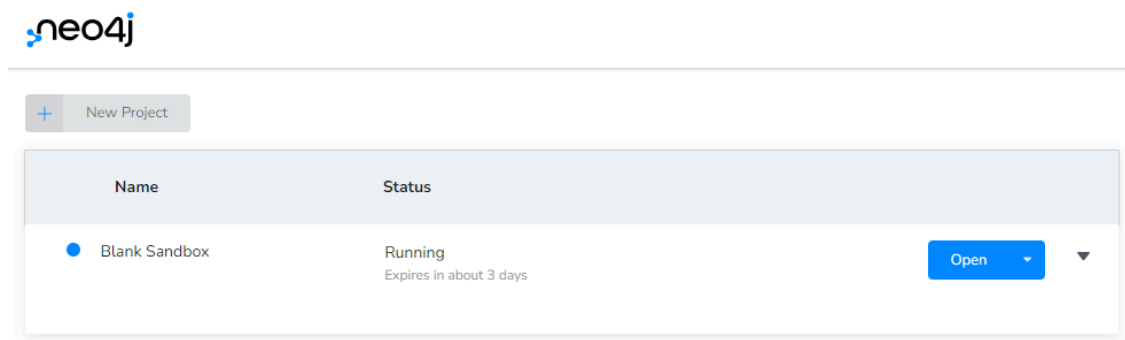
Una alternativa interessant a l'hora d'aprendre el funcionament de Neo4j és, en lloc d'instalar-ho en la nostra màquina, utilitzar el *sandbox* (un entorn controlat de proves) que ofereix Neo4j.

Per fer-ho, anam a <https://neo4j.com/sandbox/> i feim clic al botó "*Launch the Free Sandbox*". Hem de registrar-nos (podem utilitzar el nostre usuari de Google, X/Twitter o LinkedIn) i després triar un dels projectes, amb les seves dades, que s'ofereixen en el *sandbox*. De moment, per a les nostres primeres proves, anam a fer feina amb el projecte en blanc, *Blank Sandbox* (en l'apartat *Your own data*).



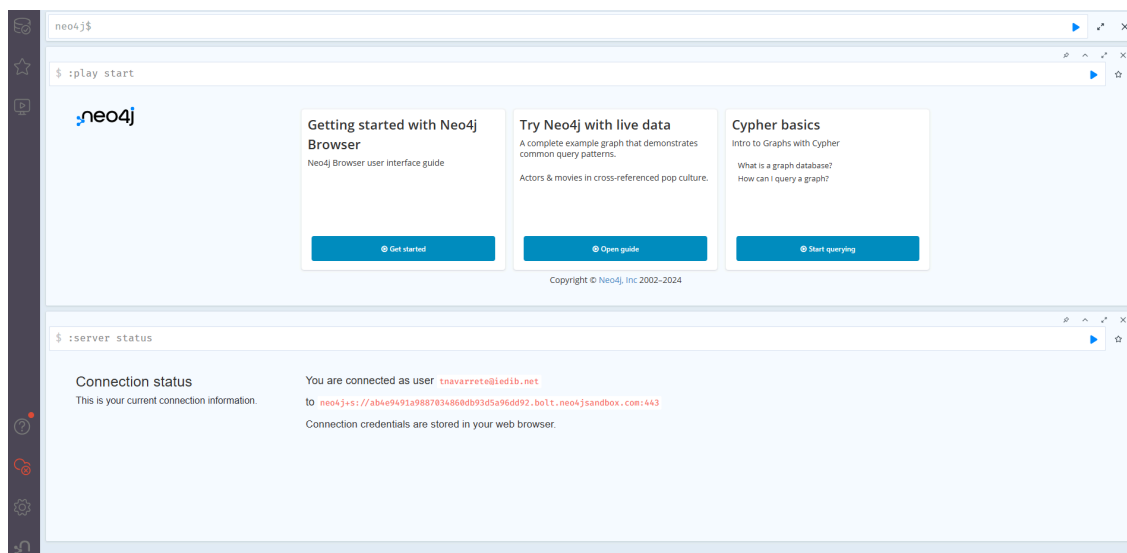
**Imatge:** Pantalla inicial del sandbox

Ens apareixerà el nostre projecte *Blank Sandbox*:



**Imatge:** Llistat de projectes en el sandbox

Si ara pitjam el botó "Open" ("*Open with Browser*") obrirà el Neo4j Browser (potser ens torna a demanar que ens autèntiquem), idèntic al que ja hem vist en l'apartat anterior.



**Imatge:** Neo4j Browser en el sandbox

Una vegada creat, el projecte estarà disponible durant **3 dies**, tot i que es pot estendre fins a **7 dies més**. A més, podem convidar a altres usuaris a col·laborar en el projecte.

## 5.3. Creació del graf

En aquest apartat veurem com es crea un graf en Neo4j emprant Cypher, el llenguatge de consultes sobre grafs de Neo4j. Desafortunadament, no hi ha cap llenguatge estàndard per a bases de dades basades en grafs, equivalent al que seria SQL en bases de dades relacional. S'ha intentat definir un GQL (Graph Query Language), però de moment encara no s'ha arribat a publicar. En aquest context, Cypher i la seva versió oberta, openCypher, s'han convertit en l'opció més àmpliament adoptada. Per exemple, hi ha diversos projectes que permeten utilitzar openCypher sobre Apache Spark.

Anam a fer feina amb el graf de pel·lícules, actors i directors que ja hem vist a l'apartat [Graf de propietats etiquetats](#).

Recordem que en un graf de propietats etiquetat, un node pot tenir etiquetes (habitualment una), així com propietats. Totes dues, etiquetes i propietats, són opcionals. La sintaxi general per referenciar un node és aquesta, entre parèntesis:

```
(:etiquetes propietats)
```

Així doncs, la sentència per crear un nou node, CREATE, té aquesta forma:

```
CREATE (:etiquetes propietats)
```

Vegem com es crea un node concret, el de Martin Sheen: té l'etiqueta *Person* i una propietat *name* amb el valor "Martin Sheen":

```
CREATE (:Person {name:"Martin Sheen"})
```



És important fixar-se que:

- Els nodes sempre van entre parèntesis.
- Les propietats sempre van entre claus.
- Les arestes sempre van entre claudàtors (ho veurem més avall).
- Les etiquetes (i noms de relació) sempre van precedides per : (dos punts).
- Cypher no és *case-sensitive*, així que és indiferent escriure *CREATE*, *Create* o *create*.
- També podem utilitzar cometes simples o dobles indistintament per a les cadenes de caràcters (sempre que no hagi apòstrofs en la cadena).

Podem crear tots els nodes del nostre graf, amb diverses sentències CREATE com l'anterior, o bé tots els nodes en una única sentència, separats per comes:

```
CREATE
  (:Person {name: "Charlie Sheen"}),
  (:Person {name: "Martin Sheen"}),
  (:Person {name: "Michael Douglas"}),
  (:Person {name: "Oliver Stone"}),
  (:Person {name: "Rob Reiner"}),
  (:Movie {title: "Wall Street"}),
  (:Movie {title: "The American President"})
```

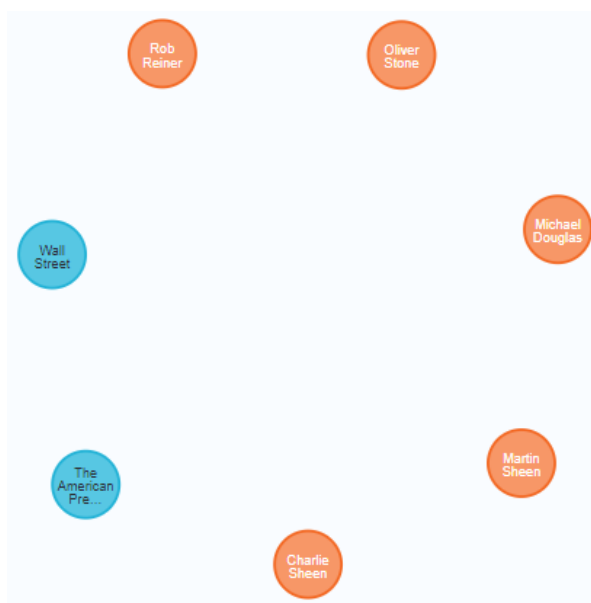
Si ara volem veure gràficament el graf, recuperam tots els nodes mitjançant la següent sentència:

```
MATCH (n) RETURN n
```

Que ens recupera tots els nodes del graf:

n	
(:Person {name: "Charlie Sheen"})	
(:Person {name: "Martin Sheen"})	
(:Person {name: "Michael Douglas"})	
(:Person {name: "Oliver Stone"})	
(:Person {name: "Rob Reiner"})	
(:Movie {title: "Wall Street"})	
(:Movie {title: "The American President"})	

I la seva representació gràfica és:



**Imatge:** Nodes del graf

La sentència MATCH selecciona els nodes que satisfan una condició. Per exemple, si volem recuperar només el node corresponent a Oliver Stone, ho faríem així:

```
MATCH (n {name:"Oliver Stone"}) RETURN n
```

També podríem haver concretat que Oliver Stone és una persona:

```
MATCH (n:Person {name:"Oliver Stone"}) RETURN n
```

Veurem més detalls sobre les consultes amb MATCH en el següent apartat.



Ara anam a crear les arestes o relacions del graf. Recordem que les arestes en un graf de propietats etiquetat tenen un nom per representar el tipus de relació. I recordem també que poden tenir, opcionalment, propietats.

Una aresta es referencia de manera general amb la següent sintaxi:

```
(node_origen)-[:NOM_ARESTA propietats]->(node_destí)
```

Per crear una aresta, per exemple, la que va de Martin Sheen a Charlie Sheen del tipus FATHER\_OF, primer hem de recuperar els nodes involucrats amb un MATCH, i assignar-los un identificador. Aquest identificador és el que emprarem en la sentència CREATE:

```
MATCH (martin:Person {name:"Martin Sheen"})
MATCH (charlie:Person {name:"Charlie Sheen"})
CREATE (martin)-[:FATHER_OF]->(charlie)
```

Això també ho podem escriure amb un únic MATCH, amb cada node separat per comes:

```
MATCH (martin:Person {name:"Martin Sheen"}), (charlie:Person {name:"Charlie Sheen"})
CREATE (martin)-[:FATHER_OF]->(charlie)
```

Com ja sabem, tenim arestes amb propietats, com per exemple la que representa que Martin Sheen fa el paper de Carl Fox a la pel·lícula Wall Street:

```
MATCH (martin:Person {name:"Martin Sheen"}), (wallstreet:Movie {title:"Wall Street"})
CREATE (martin)-[:ACTED_IN {role: "Carl Fox"}]->(wallstreet)
```

Una altra manera de crear les arestes podria haver estat fer-les totes de cop, amb una única sentència CREATE: primer hem de recuperar amb clàusules MATCH tots els nodes que tenen arestes (ja sigui com a origen o com a destí) i després els cream amb un únic CREATE, separant per comes cada aresta:

```
MATCH (charlie:Person {name: "Charlie Sheen"}),
(martin:Person {name: "Martin Sheen"}),
(michael:Person {name: "Michael Douglas"}),
(oliver:Person {name: "Oliver Stone"}),
(rob:Person {name: "Rob Reiner"}),
(wallStreet:Movie {title: "Wall Street"}),
(thePresident:Movie {title: "The American President"})
CREATE
(charlie)-[:ACTED_IN {role: "Bud Fox"}]->(wallStreet),
(martin)-[:ACTED_IN {role: "Carl Fox"}]->(wallStreet),
(michael)-[:ACTED_IN {role: "Gordon Gekko"}]->(wallStreet),
(oliver)-[:DIRECTED]->(wallStreet),
(martin)-[:ACTED_IN {role: "A.J. MacInerney"}]->(thePresident),
(michael)-[:ACTED_IN {role: "President Andrew Shepherd"}]->(thePresident),
(rob)-[:DIRECTED]->(thePresident),
(martin)-[:FATHER_OF]->(charlie)
```

Hem d'anar alerta perquè nosaltres ja havíem creat dues de les arestes i amb aquesta sentència ens les duplicaria.

També ho podríem haver fet dins de la mateixa sentència CREATE en què cream els nodes:

CREATE

```
(charlie:Person {name: "Charlie Sheen"}),
(martin:Person {name: "Martin Sheen"}),
(michael:Person {name: "Michael Douglas"}),
(oliver:Person {name: "Oliver Stone"}),
(rob:Person {name: "Rob Reiner"}),
(wallStreet:Movie {title: "Wall Street"}),
(thePresident:Movie {title: "The American President"}),
(charlie)-[:ACTED_IN {role: "Bud Fox"}]->(wallStreet),
(martin)-[:ACTED_IN {role: "Carl Fox"}]->(wallStreet),
(michael)-[:ACTED_IN {role: "Gordon Gekko"}]->(wallStreet),
(oliver)-[:DIRECTED]->(wallStreet),
(martin)-[:ACTED_IN {role: "A.J. MacInerney"}]->(thePresident),
(michael)-[:ACTED_IN {role: "President Andrew Shepherd"}]->(thePresident),
(rob)-[:DIRECTED]->(thePresident),
(martin)-[:FATHER_OF]->(charlie)
```



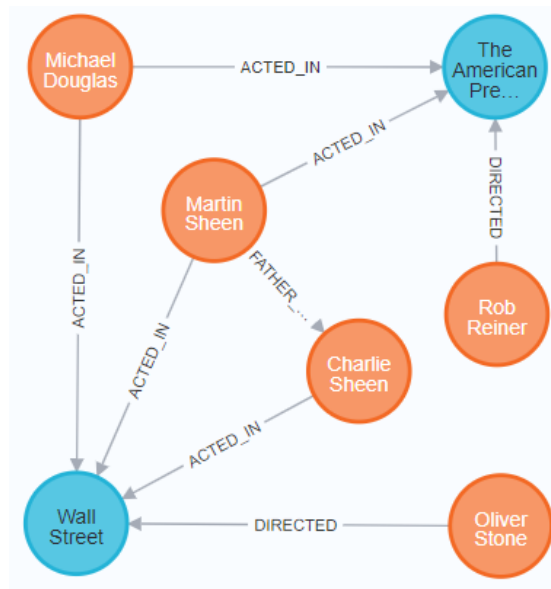
Abans d'executar aquesta sentència hem d'eliminar els nodes i arestes que ja havíem creat, per evitar tenir-los tots duplicats:

```
MATCH (n) DETACH DELETE n
```

Veurem més detalls sobre l'esborrat de nodes i arestes en un apartat posterior.

Vegem com queda el nostre graf, executant de nou la sentència:

```
MATCH (n) RETURN n
```



Imatge: Nodes i arestes del graf

## 5.4. Consultes bàsiques

En aquest apartat veurem com escriure en Cypher les consultes senzilles més habituals sobre un graf. Seguirem treballant amb el nostre graf de pel·lícules, actors i directors que hem creat en l'apartat anterior.

### ■ Recuperar tots els nodes

Ja hem vist abans que MATCH ens permet recuperar els nodes que satisfan una certa condició o patró.

Si volem recuperar tots els nodes, no hem d'especificar cap patró:

```
MATCH (n) RETURN n
```

On  $n$  és una variable, que en aquest cas contindrà nodes (veurem més endavant que també podem definir variables per a relacions). Com que no li hem aplicat cap patró, la consulta retorna tots els nodes del graf. Això és el que retorna:

n
(:Person {name: "Charlie Sheen"})
(:Person {name: "Martin Sheen"})
(:Person {name: "Michael Douglas"})
(:Person {name: "Oliver Stone"})
(:Person {name: "Rob Reiner"})
(:Movie {title: "Wall Street"})
(:Movie {title: "The American President"})

De vegades, ens pot interessar recuperar el tipus (etiqueta) d'un node etiquetes d'un node. Per fer-ho, podem emprar la funció *labels*, que ens retornarà un array ja que un node pot tenir diverses etiquetes. Vegem com retornar la primera etiqueta de cada un dels nodes (en el nostre graf, tots els nodes en tenen només una, d'etiqueta):

```
MATCH (n) RETURN labels(n)[0] AS etiqueta
```

Que retorna:

etiqueta
"Person"
"Person"
"Person"
"Person"
"Person"
"Person"
"Movie"
"Movie"

### ■ *Recuperar els nodes amb una etiqueta i recuperar valors d'una propietat*

Podem filtrar només pels nodes d'un tipus (una etiqueta). També podem fer que retorni no tot el node sinó només el valor d'una de les seves propietats. Per exemple, anam a recuperar els títols de totes les pel·lícules:

```
MATCH (m:Movie) RETURN m.title
```

Que retorna:

m.title
"Wall Street"
"The American President"

### ■ *Nodes relacionats*

El símbol -- (dos guions) es fa servir per indicar que dos nodes estan relacionats, que tenen una aresta entre ells, independentment de la seva direcció. Per exemple, si volem recuperar el títol de les pel·lícules dirigides per Rob Reiner:

```
MATCH (director {name: 'Rob Reiner'})--(movie)
RETURN movie.title
```

Que retorna:

movie.title
"The American President"

Vegem que en la clàusula MATCH hem definit les variables *director* i *movie*, que després podem emprar en la clàusula RETURN.

Podríem també afinar més la consulta especificant les etiquetes dels dos nodes:

```
MATCH (director:Person {name: 'Rob Reiner'})--(movie:Movie)
RETURN movie.title
```

### ■ *Més sobre relacions entre nodes*

Podem especificar la direcció de les arestes, emprant els símbols > o < per indicar una direcció: en lloc de -- que no suposava cap direcció tendrem --> o <-- que sí que n'indiquen. Per exemple, en la consulta anterior podem precisar que l'aresta ha d'anar de la persona a la pel·lícula:

```
MATCH (director:Person {name: 'Rob Reiner'})-->(movie:Movie)
RETURN movie.title
```

Si volem ara saber quin és el director de la pel·lícula *The American President*, necessitam especificar que només s'han de tenir en compte les arestes del tipus DIRECTED:

```
MATCH (movie:Movie {title: 'The American President'})<-[:DIRECTED]-(director:Person)
RETURN director.name
```

Que retorna:

director.name
"Rob Reiner"

En canvi, per saber-ne els actors:

```
MATCH (movie:Movie {title: 'The American President'})<-[:ACTED_IN]-(actor:Person)
RETURN actor.name
```

Que retorna:

actor.name
"Michael Douglas"
"Martin Sheen"

I si volem també saber quin paper fa cada un d'ells, hem de definir una altra variable (*rel*) per a la relació:

```
MATCH (movie:Movie {title: 'The American President'})<-[:rel:ACTED_IN]-(actor:Person)
RETURN actor.name, rel.role
```

Que retorna:

actor.name	rel.role
"Michael Douglas"	"President Andrew Shepherd"
"Martin Sheen"	"A.J. MacInerney"

Podem utilitzar diverses relacions en una única sentència. Per exemple, si volem saber qui són els directors de les pel·lícules en que Michael Douglas ha actuat:

```
MATCH (md {name: 'Michael Douglas'})-[:ACTED_IN]->(movie)<-[:DIRECTED]-(director)
RETURN movie.title, director.name
```

Que retorna:

movie.title	director.name
"Wall Street"	"Oliver Stone"
"The American President"	"Rob Reiner"

I si volem saber totes les persones, ja siguin actors o directors, que han participat en la pel·lícula *The American President*? Podem emprar l'operador | per definir una O lògica:

```
MATCH (movie:Movie {title: 'The American President'})<-[:ACTED_IN|DIRECTED]-(person)
RETURN person.name
```

Que retorna:

person.name
"Rob Reiner"
"Michael Douglas"
"Martin Sheen"

Per últim, si volem recuperar també quina funció fan (actor o director) aquestes persones, tornarem a emprar una variable per a la relació i després la funció type, que retorna el tipus (nom) de la relació:

```
MATCH (movie:Movie {title: 'The American President'})<-[:rel:ACTED_IN|DIRECTED]-(person)
RETURN person.name, type(rel)
```

Que retorna:

person.name	type(rel)
"Rob Reiner"	"DIRECTED"
"Michael Douglas"	"ACTED_IN"
"Martin Sheen"	"ACTED_IN"

## 5.5. Modificacions

La clàusula SET serveix per modificar una propietat d'un node o relació. Si la propietat no estava definida en el node o relació, es crearà.

Per exemple, en el nostre graf de pel·lícules, directors i actors, volem afegir una propietat per indicar que Oliver Stone va néixer en l'any 1946:

```
MATCH (oliver {name: 'Oliver Stone'})
SET oliver.born = 1946
```

Vegem un altre exemple on crearem una propietat *oscar* per indicar si un director o actor va guanyar un premi Oscar per la seva feina a una pel·lícula concreta. En aquest cas, la propietat no pertany al node de tipus *Person*, sino a la relació entre *Person* i *Movie*. Vegem com representam que Michael Douglas va guanyar l'Oscar per la seva actuació en *Wall Street*:

```
MATCH (michael {name: 'Michael Douglas'})-[rel]->(wallstreet {title: 'Wall Street'})
SET rel.oscar = true
```

Els dos exemples anteriors afegeixen una nova propietat, però, com ja hem dit, amb SET també podem modificar el valor d'una propietat ja existent. Per exemple, anam a canviar el nom de Rob Reiner a Robert Reiner:

```
MATCH (rob {name: 'Rob Reiner'})
SET rob.name = 'Robert Reiner'
```

Si assignam el valor *null* a una propietat, l'eliminem. Per exemple, si volem eliminar la propietat *born* que abans hem afegit a Oliver Stone:

```
MATCH (oliver {name: 'Oliver Stone'})
SET oliver.born = null
```

Podem eliminar totes les propietats (també *name*) d'un node de cop, emprant {}:

```
MATCH (oliver {name: 'Oliver Stone'})
SET oliver = {}
```



ALERTA

Després d'això, el node continua existint, amb les seves relacions, però no té cap propietat. Sí que conserva un identificador intern que permet seguir referenciant-lo.

S'ha d'anar molt alerta fent aquestes modificacions perquè ja no podem recuperar aquest node fent servir les seves propietats.

Podem emprar els operadors i funcions que necessitem en les sentències SET. Per exemple, anam a passar tots els noms dels actors i directors a majúscules, fent servir la funció *toUpper()*.

```
MATCH (p:Person)
SET p.name = toUpper(p.name)
```

Per últim, també és possible afegir una etiqueta a un node. Per exemple, volem afegir l'etiqueta Actor al node de Michael Douglas (que ara el tenim en majúscules):

```
MATCH (michael {name: 'MICHAEL DOUGLAS'})  
SET michael:Actor
```

El resultat és que aquest node tindrà dues etiquetes, Person i Actor. Podríem fer això per a tots els nodes que són origen d'una relació ACTED\_IN:

```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)  
SET p:Actor
```

I el mateix per als directors:

```
MATCH (p:Person)-[:DIRECTED]-(m:Movie)  
SET p:Director
```



## 5.6. Esborrats

Tenim dos tipus de sentències diferents per als esborrats:

- REMOVE esborra una propietat o una etiqueta
- DELETE esborra un node o una aresta (relació)

### ■ REMOVE

Ja hem vist abans que amb SET assignant el valor *null* podem eliminar una propietat. REMOVE és equivalent. Anam a eliminar la propietat *name* del node de Martin Sheen (que ara està en majúscules):

```
MATCH (m {name: 'MARTIN SHEEN'})
REMOVE m.name
```

El node segueix existint, però ja no té cap propietat (només el seu identificador intern). L'esborrat d'una propietat d'una relació es fa de la mateixa manera. Anam a eliminar la propietat *role* de Michael Douglas en la pel·lícula The American President:

```
MATCH (n {name: 'MICHAEL DOUGLAS'})-[r]-(m {title: 'The American President'})
REMOVE r.role
```

Amb REMOVE també podem eliminar una etiqueta, per exemple, Actor de Charlie Sheen:

```
MATCH (c {name: 'CHARLIE SHEEN'})
REMOVE c:Actor
```

També podem eliminar més d'una etiqueta a la vegada. Per exemple, eliminarem les dues etiquetes que té Rob Reiner (a qui li havíem canviat el nom per Robert Reiner i l'hem posat en majúscules en l'apartat anterior)

```
MATCH (r {name: 'ROBERT REINER'})
REMOVE r:Director:Person
```

### ■ DELETE

Començarem eliminant la relació DIRECTED de Rob Reiner:

```
MATCH (p:Person {name: 'ROBERT REINER'})-[rel:DIRECTED]->(m:Movie)
DELETE rel
```

I ara eliminarem el node:

```
MATCH (p:Person {name: 'ROBERT REINER'})
DELETE p
```

Hem pogut eliminar el node perquè ja no participava a cap relació. Intentem eliminar ara el node de Michael Douglas:

```
MATCH (p:Person {name: 'MICHAEL DOUGLAS'})
DELETE p
```

Ens dona el següent error:

```
ERROR Neo.ClientError.Schema.ConstraintValidationFailed
Cannot delete node<6>, because it still has relationships. To delete this node, you must first
delete its relationships.
```

Així que abans d'eliminar el node de Michael Douglas, hauríem d'eliminar les dues relacions que té. També, però, podem eliminar de forma directa un node i totes les relacions en què participa, fent servir la paraula DETACH:

```
MATCH (p:Person {name: 'MICHAEL DOUGLAS'})
DETACH DELETE p
```

Així, tal com vàrem anunciar en l'apartat de [Creació del graf](#), si volem eliminar el graf complet, eliminaríem amb DETACH DELETE tots els seus nodes:

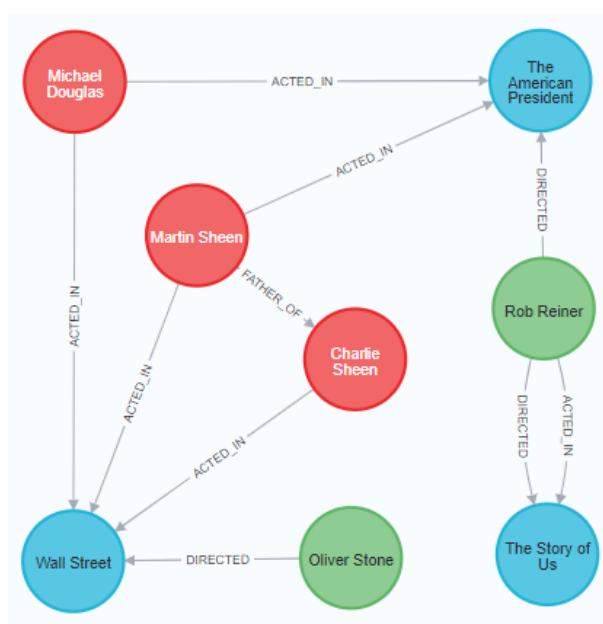
```
MATCH (n) DETACH DELETE n
```

## 5.7. Consultes avançades

Una vegada que hem esborrat el graf en l'apartat anterior, anam a crear de nou el nostre graf, però introduïrem alguns petits canvis. Anam a distingir amb etiquetes diferents els actors i directors. A més, afegirem els anys de naixement dels actors i directors. Per últim, anam a afegir la pel·lícula *The Story of Us*, dirigida per Rob Reiner, qui també hi actua amb el paper de Stan.

### CREATE

```
(charlie:Actor {name: "Charlie Sheen", born: 1965}),
(martin:Actor {name: "Martin Sheen", born: 1940}),
(michael:Actor {name: "Michael Douglas", born: 1944}),
(oliver:Director {name: "Oliver Stone", born: 1946}),
(rob:Director:Actor {name: "Rob Reiner", born: 1947}),
(wallStreet:Movie {title: "Wall Street"}),
(thePresident:Movie {title: "The American President"}),
(storyOfUs:Movie {title: "The Story of Us"}),
(charlie)-[:ACTED_IN {role: "Bud Fox"}]->(wallStreet),
(martin)-[:ACTED_IN {role: "Carl Fox"}]->(wallStreet),
(michael)-[:ACTED_IN {role: "Gordon Gekko"}]->(wallStreet),
(oliver)-[:DIRECTED]->(wallStreet),
(martin)-[:ACTED_IN {role: "A.J. MacInerney"}]->(thePresident),
(michael)-[:ACTED_IN {role: "President Andrew Shepherd"}]->(thePresident),
(rob)-[:DIRECTED]->(thePresident),
(martin)-[:FATHER_OF]->(charlie),
(rob)-[:DIRECTED]->(storyOfUs),
(rob)-[:ACTED_IN {role: "Stan"}]->(storyOfUs)
```



**Imatge:** Nou graf de pel·lícules, actors i directors

Tot i que Rob Reiner es mostra en verd (directors), té les dues etiquetes, Director i Actor.

### ■ La clàusula WHERE

La clàusula WHERE es pot utilitzar amb un MATCH per afegir restriccions a la cerca. Podem filtrar segons el valor d'una propietat d'un node o una relació. Per exemple, volem recuperar les persones que han nascut després de 1945 (inclòs):

```
MATCH (n)
WHERE n.born >= 1945
RETURN n
```

Veim que ens retorna els nodes d'Oliver Stone, Rob Reiner i Charlie Sheen.

n
(:Director {born: 1946,name: "Oliver Stone"})
(:Actor:Director {born: 1947,name: "Rob Reiner"})
(:Actor {born: 1965,name: "Charlie Sheen"})

També podem filtrar per etiquetes d'un node. Per exemple, volem recuperar només els actors:

```
MATCH (n)
WHERE n:Actor
RETURN n
```

Podem observar que entre els resultats es troba Rob Reiner, perquè també és un actor:

n
(:Actor {born: 1944,name: "Michael Douglas"})
(:Actor:Director {born: 1947,name: "Rob Reiner"})
(:Actor {born: 1965,name: "Charlie Sheen"})
(:Actor {born: 1940,name: "Martin Sheen"})

Fixau-vos que aquesta consulta també l'hauríem pogut escriure especificant l'etiqueta en la clàusula MATCH:

```
MATCH (n:Actor)
RETURN n
```

En la clàusula WHERE del primer exemple, hem emprat l'operador de comparació >=. Però Cypher permet emprar molts més operadors. A continuació es mostren els més habituals:

Tipus	Operadors
Comparació	=, <>, <, >, <=, >=, IS NULL i IS NOT NULL
Aritmètics	+, -, *, /, % i ^ (potència)
De cadenes de caràcters	STARTS WITH, ENDS WITH, CONTAINS i + (concatenació)
Lògics	AND, OR, XOR, NOT
De llista	IN, + (concatenació) i [ ] (per definir una llista de valors)
D'agregació	DISTINCT

**Taula:** Principals operadors de Cypher

Vegem alguns d'exemples.

Comencem recuperant els actors nascuts en la dècada dels 40 (entre 1940 i 1949):

```
MATCH (n)
WHERE n:Actor AND n.born >= 1940 AND n.born <=1949
RETURN n
```

Que retorna:

n
(:Actor {born: 1944,name: "Michael Douglas"})
(:Actor:Director {born: 1947,name: "Rob Reiner"})
(:Actor {born: 1940,name: "Martin Sheen"})

Ara recuperarem els actors que han interpretat un personatge de la família Fox (el seu paper acaba amb la cadena Fox):

```
MATCH (a:Actor)-[r]->(m:Movie)
WHERE r.role ENDS WITH 'Fox'
RETURN a
```

Que retorna:

a
(:Actor {born: 1965,name: "Charlie Sheen"})
(:Actor {born: 1940,name: "Martin Sheen"})

No ho havíem vist fins ara, però en una clàusula RETURN podem emprar el símbol \* perquè es retornin totes les variables que s'han emprat en la consulta:

```
MATCH (a:Actor)-[r]->(m:Movie)
WHERE r.role ENDS WITH 'Fox'
RETURN *
```

Que retorna:

a	m	r
(:Actor {born: 1965,name: "Charlie Sheen"})	(:Movie {title: "Wall Street"})	[[:ACTED_IN {role: "Bud Fox"}]]
(:Actor {born: 1940,name: "Martin Sheen"})	(:Movie {title: "Wall Street"})	[[:ACTED_IN {role: "Carl Fox"}]]

Podem emprar el resultat d'una clàusula MATCH com a entrada d'una altra clàusula MATCH. Vegem-ho amb el següent exemple, on volem recuperar els directors que han treballat amb Michael Douglas.

```
MATCH (a {name: 'Michael Douglas'})-[:ACTED_IN]->(m:Movie)
MATCH (m)<-[:DIRECTED]-(d)
RETURN d.name
```

En la primera clàusula MATCH obtenim la variable *m* amb els nodes amb etiqueta pel·lícula relacionats amb Michael Douglas. En la segona clàusula, empram aquesta variable *m* per obtenir els directors relacionats amb aquesta pel·lícula. El resultat és:

d.name
"Rob Reiner"
"Oliver Stone"

Fixau-vos que aquesta consulta també l'haguéssim pogut escriure amb una única clàusula MATCH:

```
MATCH (a {name: 'Michael Douglas'})-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d)
RETURN d.name
```

### ■ La clàusula WITH

La clàusula WITH permet encadenar diverses parts d'una consulta, prenent els resultats d'una part i utilitzar-los en una altra. En aquesta situació, WITH ens permet manipular la sortida d'una part i definir noves variables, que seran emprades en la següent part.

Vegem-ho amb un exemple. Volem cercar els actors de la pel·lícula *Wall Street* tals que el seu nom comenci per 'CHARL', però de manera que no sigui sensible a les majúscules (suposem que no sabem si el nom dels actors està escrit en majúscules o minúscules).

```
MATCH (m {title: 'Wall Street'})<--(a:Actor)
WITH a, toUpper(a.name) AS nomMajus
WHERE nomMajus STARTS WITH 'CHARL'
RETURN a.name
```

Podem veure que primer hem recuperat la variable *a* per als actors de Wall Street i que a partir d'ella, amb la clàusula WITH obtenim la variable *nomMajus*, que empram per a executar el WHERE. El resultat és Charlie Sheen:

a.name
"Charlie Sheen"

Vegem un altre exemple d'ús habitual, on definim una variable *year* que després empram en el MATCH, en aquest cas per recuperar els actors nascuts després de 1950:

```
WITH 1950 AS year
MATCH (a:Actor)
WHERE a.born > year
RETURN a
```

El resultat és:

a
(:Actor {born: 1965,name: "Charlie Sheen"})

Un altre ús molt freqüent té a veure amb les agregacions. Suposem que volem saber el nombre d'actors que participen en la pel·lícula Wall Street:

```
MATCH (m:Movie)-[:ACTED_IN]-(a:Actor)
WITH m, count(*) AS num
RETURN m, num
```

El resultat és:

m	num
(:Movie {title: "The American President"})	2
(:Movie {title: "Wall Street"})	3
(:Movie {title: "The Story of Us"})	1

Afinant una mica més, ara volem recuperar el títol de les pel·lícules on tenim més d'un actor:

```
MATCH (m:Movie)-[:ACTED_IN]-(a:Actor)
WITH m, count(*) AS num
WHERE num > 1
RETURN m.title
```

El resultat és:

m.title
"The American President"
"Wall Street"

### ■ Les clàusules **ORDER BY** i **LIMIT**

La clàusula ORDER BY permet ordenar els resultats, tant d'una clàusula RETURN com d'una WITH. Per defecte, l'ordenació és ascendent (ASC o ASCENDING), i si volem canviar-la, hem d'afegir la paraula DESC (o DESCENDING). Vegem un exemple típic, on volem recuperar els actors del nostre graf, ordenats per any de naixement, de més jove a més vell:

```
MATCH (n:Actor)
RETURN n.name, n.born
ORDER BY n.born DESC
```

El resultat és:

n.name	n.born
"Charlie Sheen"	1965
"Rob Reiner"	1947
"Michael Douglas"	1944
"Martin Sheen"	1940

D'altra banda, la clàusula LIMIT permet limitar el número de respostes. Per exemple, continuant amb la consulta anterior, si només volem saber quin és l'actor més jove, podem limitar a 1 els resultats:

```
MATCH (n:Actor)
RETURN n.name, n.born
ORDER BY n.born DESC
LIMIT 1
```

El resultat és:

n.name	n.born
"Charlie Sheen"	1965

Tal com hem dit anteriorment, ORDER BY també es pot emprar amb la clàusula WITH. En aquest cas, l'ordenació es fa abans d'aplicar la clàusula WHERE, RETURN o LIMIT que puguin seguir a WITH. Per exemple, volem recuperar una llista (emprant la funció *collect*) amb el nom dels dos actors més joves:

```
MATCH (n:Actor)
WITH n
ORDER BY n.born DESC
LIMIT 2
RETURN collect(n.name)
```

Notau que és important que l'ordenació es faci abans que el LIMIT i que el *collect* del RETURN. El resultat és:

collect(n.name)
["Charlie Sheen", "Rob Reiner"]



## 5.8. Procediments

Neo4j incorpora tot un conjunt de procediments o subrutines. Podeu veure el llistat complet, amb la seva definició, en la [documentació de Neo4j](#).



AMPLIACIÓ

L'usuari també pot declarar els seus propis procediments, fent servir el llenguatge Java.

Aquest no és un procés senzill i queda fora de l'abast d'aquest curs.

Si voleu aprofundir en aquest tema, podeu consultar la [documentació de Neo4j](#).

Cipher permet l'execució d'aquests procediments mitjançant la clàusula CALL. Vegem un exemple que crida al procediment `db.labels()`, el qual recupera totes les etiquetes afegides als nodes d'un graf (d'una base de dades).

```
CALL db.labels()
```

El resultat és:

label
"Movie"
"Actor"
"Director"

No és obligatori posar els parèntesis si no hi ha arguments (podríem haver escrit `CALL db.labels`), però sí recomanable. Altres procediments sí que tenen arguments. Per exemple, `checkConfigValue` és un procediment que proporciona comentaris sobre la validesa d'un valor de la configuració, tot i que no modifica la configuració (no és important entendre ara el funcionament, només veure com es fa la crida i què retorna):

```
CALL dbms.checkConfigValue('server.bolt.enabled', 'true')
```

Que retorna:

valid	message
true	"requires restart"

A més de consultant la [documentació](#), també podem saber els detalls sobre els procediments que incorpora Neo4j mitjançant la sentència SHOW PROCEDURES:

```
SHOW PROCEDURES
```

Per defecte, SHOW PROCEDURES retorna el nom, descripció, mode i un flag `worksOnSystem`. Aquestes són les primeres files retornades:

name	description
mode worksOnSystem	
"apoc.algo.aStar"	"Runs the A* search algorithm to find the optimal path between two `NODE` values, using the given `RELATIONSHIP` property name for the cost function."
"DEFAULT" false	
"apoc.algo.aStarConfig"	"Runs the A* search algorithm to find the optimal path between two `NODE` values, using the given `RELATIONSHIP` property name for the cost function.\nThis procedure looks for weight, latitude and longitude properties in the config."
"DEFAULT" false	
"apoc.algo.aStarWithPoint"	"apoc.algo.aStarWithPoint(startNode, endNode, 'relTypesAndDirs', 'distance','pointProp') - equivalent to apoc.algo.aStar but accept a Point type as a pointProperty instead of Number types as latitude and longitude properties"
"DEFAULT" false	
"apoc.algo.allSimplePaths"	"Runs a search algorithm to find all of the simple paths between the given `RELATIONSHIP` values, up to a max depth described by `maxNodes`. \n\nThe returned paths will not contain loops."
"DEFAULT" false	
"apoc.algo.cover"	"Returns all `RELATIONSHIP` values connecting the given set of `NODE` values."
"DEFAULT" false	

Amb YIELD podem obtenir-ne també la signatura, per saber quins arguments i valors retornats té cada procediment. Per exemple, si volem saber la signatura del primer procediment del llistat anterior (*apoc.algo.aStar*):

```
SHOW PROCEDURES YIELD name, signature
WHERE name = 'apoc.algo.aStar'
RETURN signature
```

Que retorna:

signature
"apoc.algo.aStar(startNode :: NODE, endNode :: NODE, relTypesAndDirections :: STRING, weightPropertyName :: STRING, latPropertyName :: STRING, lonPropertyName :: STRING) :: (path :: PATH, weight :: FLOAT)"

Acabam aquest apartat fixant-nos en aquesta subclàusula YIELD que acabam d'emprar en la darrera sentència.

La majoria dels procediments retornen un seguit de registres amb un conjunt fix de camps de resultats, similar al que retorna una consulta Cypher. La subclàusula YIELD s'utilitza per seleccionar explícitament quins dels camps de resultat disponibles es retornen com a noves variables, que podran ser emprades per l'usuari en la resta de la query.

En la sentència anterior, quan hem escrit *SHOW PROCEDURES YIELD name, signature* estam especificant que, com a resultat de SHOW PROCEDURES, tendrem dues variables *name* i *signature*, que podrem emprar després, en el nostre cas en el WHERE i el RETURN.

Per exemple, tornant al procediment *db.labels()* que hem vist al principi d'aquest apartat, si volem saber quantes etiquetes tenim en el nostre graf, ho faríem així:

```
CALL db.labels() YIELD label  
RETURN count(label) AS numLabels
```

Que retorna 3 (tenim tres etiquetes diferents en el nostre graf, Movie, Actor i Director):

numLabels
3

## 5.9. Carregar un graf des de fitxers CSV

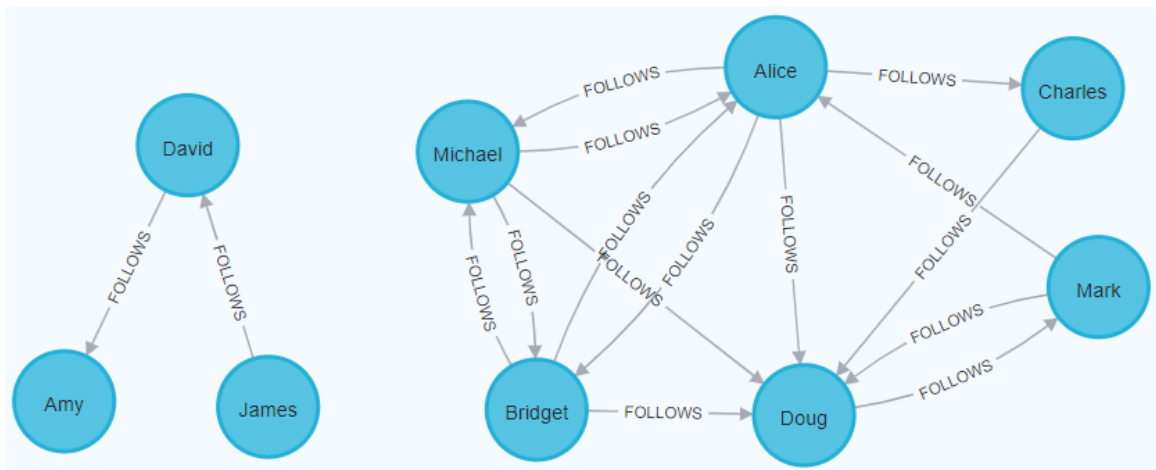
En aquest apartat veurem com carregar un graf a partir de fitxers CSV, on tindrem guardats els nodes i arestes.

Ho farem amb un graf que representa com se segueixen un conjunt d'usuaris d'una xarxa social, publicat a <https://github.com/neo4j-graph-analytics/book/tree/master/data>. Al fitxer *social-nodes.csv* tenim els nodes del graf, mentre que a *social-relationships.csv* en tenim les arestes. Vegem com crear el nostre graf a partir d'aquests fitxers:

```
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})

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)
```

Aquesta és la representació gràfica del graf:



**Imatge:** Graf d'una xarxa social

## 6. Problemes específics de grafs amb Neo4j

En el capítol anterior hem vist com definir un graf de propietats etiquetat en Neo4j i com fer operacions de consulta i modificacions mitjançant Cypher sobre els seus nodes i arestes.

Però l'ús de grafs normalment involucra alguns problemes més complexos que recuperar els nodes tals que les seves propietats satisfan una determinada condició o que estan relacionats de certa manera amb altres nodes.

Alguns dels problemes més típics que apareixen quan treballem amb grafs són:

- Com podem recuperar tots els nodes d'un graf, seguint les seves arestes? És el que anomenem **recorreguts sobre grafs** i veurem que tenim dues maneres diferents, prioritzant l'amplitud o la profunditat.
- Com podem trobar la manera més curta d'anar des d'un node fins a un altre? És el que anomenem **cerca de camins mínims**.
- Com determinem quins són els nodes més rellevants en un graf? És el que anomenem **estudi de centralitat**.
- Podem saber si es formen agrupacions de nodes dins el graf? És el que anomenem **detecció de comunitats**.
- En un graf dinàmic, que evoluciona en el temps, podem arribar a predir relacions entre els nodes? És el que anomenem **predicció d'enllaços**.

En Neo4j aquests problemes estan resolts mitjançant la llibreria **Graph Data Science (GDS)**. GDS ofereix versions paral·leles i molt eficients dels principals algorismes sobre grafs, exposats com a procediments de Cypher. Per treballar amb aquests algorismes, és necessari convertir (**projectar**) el nostre graf en un graf GDS, una estructura de dades en memòria, comprimida i optimitzada per a aquests tipus de problemes. GDS manté un catàleg dels grafs GDS que s'han registrat, on a cada graf se li assigna un nom que pot ser referenciat des dels algorismes. El catàleg existeix mentre la instància de Neo4j està executant-se i quan Neo4j s'atura, els grafs emmagatzemats en el catàleg es perden i s'han de tornar a registrar.

A continuació veurem amb més detall aquests cinc problemes i com podem treballar amb els algorismes que ens proporciona GDS.

## 6.1. Recorreguts del graf

Els recorreguts sobre grafs són una utilitat imprescindible en el treball amb aquesta estructura de dades i, en conseqüència, amb bases de dades orientades a grafs.

Un recorregut sobre grafs és un **procés que permet explorar un graf examinant tots els seus nodes, començant des d'un node inicial**. A l'hora de recórrer un graf, existeixen dues aproximacions, dos tipus de recorreguts:

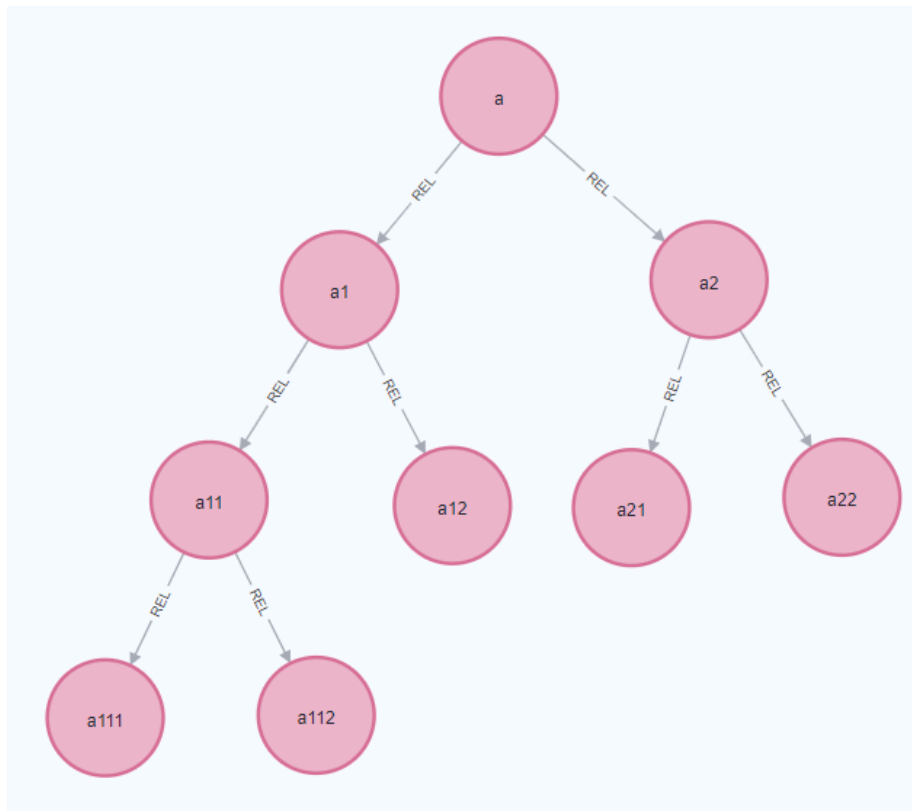
- **Recorregut en amplitud** (o en amplària o amplada), **Breadth First Search (BFS)** en anglès.
- **Recorregut en profunditat**, **Depth First Search (DFS)** en anglès.

Els algorismes de recorregut són molt utilitzats a l'hora de fer consultes i recuperar informació en un graf i són la base per a algorismes més avançats com els de cerca de camins mínims o estudis de centralitat.

### ■ Recorregut en amplitud (BFS)

El recorregut en amplitud parteix d'un node inicial. A continuació, s'exploren tots els seus nodes fills, és a dir, aquells nodes als quals podem arribar a través d'una única aresta des del node inicial. Podríem dir que obtenim la primera generació dels descendents del node inicial. I després repetim, generació a generació aquest procés, explorant tots els nodes fills dels nodes de la generació anterior.

Vegem-ho amb un exemple d'un graf senzill:



Imatge: Graf (arbre) senzill

Aquest graf és un **arbre**. Un arbre és un tipus de graf en el qual qualsevol parell de nodes està connectat per un únic camí. Per tant, no hi ha cicles (camins tancats) en un arbre. Els arbres són una estructura de dades molt utilitzada en programació. I existeixen molts tipus diferents d'arbres.

La sentència per crear aquest graf és aquesta:

```
CREATE
(a:Node {name: "a"}),
(a1:Node {name: "a1"}),
(a2:Node {name: "a2"}),
(a11:Node {name: "a11"}),
(a12:Node {name: "a12"}),
(a111:Node {name: "a111"}),
(a112:Node {name: "a112"}),
(a21:Node {name: "a21"}),
(a22:Node {name: "a22"}),
(a)-[:REL]->(a1),
(a)-[:REL]->(a2),
(a1)-[:REL]->(a11),
(a1)-[:REL]->(a12),
(a11)-[:REL]->(a111),
(a11)-[:REL]->(a112),
(a2)-[:REL]->(a21),
(a2)-[:REL]->(a22)
```

En un recorregut en amplitud partint del node "a", començant explorant tots els seus fills: "a1" i "a2". A continuació passam a la següent generació, és a dir, els fills de "a1" i "a2": "a11", "a12", "a21" i "a22". Per últim, passam a la següent i darrera generació, en aquest cas els fills de "a12": "a111" i "a112". Així doncs, l'ordre en què s'exploren els nodes és:

```
a-a1-a2-a11-a12-a21-a22-a111-a112
```

Els recorreguts en amplitud són molt utilitzats quan se cerca un camí amb el nombre mínim d'arestes entre dos nodes donats.

### ■ Recorregut en profunditat (DFS)

Aquesta és una altra manera per recórrer un graf. En aquest cas, començant pel node inicial, obtenim el seu primer fill i, de manera recursiva, recorrem en profunditat el graf a partir d'aquest node fill. Una vegada que hem explorat tots els descendents del primer fill del node inicial, n'obtenim el seu segon fill i, de nou, explorem recursivament tots els seus descendents. I així successivament per a tots els seus fills.

Seguint amb el graf anterior, en un recorregut en profunditat partint del node "a", primer explorem el primer dels seus fills, "a1". Abans de passar al segon fill, "a2", haurem de fer tot el recorregut en profunditat partint del node "a1". Així doncs, obtenim el seu primer fill, "a11" i farem el recorregut en profunditat partint de "a11". Obtenim el seu primer fill "a111" i com que aquest no té fills, tornam al nivell superior i passam al segon fill de "a11", "a112". Com que aquest és el darrer fill de "a11" (ja hem acabat el seu recorregut en profunditat), tornam al nivell superior i seguim amb el segon fill de "a1", "a12". Com que és el darrer fill de "a1" amb això acabam el seu recorregut en profunditat i tornam al nivell superior i passam al segon fill de "a", que és "a2". Explorarem primer "a21" i després "a22", amb la qual cosa acabam el recorregut en profunditat de "a2" i tornam al nivell superior. Com que "a" ja no té més fills, el recorregut ha finalitzat.

Així doncs, l'ordre en què s'exploren els nodes és:

```
a-a1-a11-a111-a112-a12-a2-a21-a22
```

## ■ Recorreguts en Neo4j

Recordem que GDS és la llibreria de Neo4j que dona suport als principals algorismes sobre grafs. Abans de poder cridar als procediments que implementen aquests algorismes, hem de projectar el nostre graf a un graf GDS, indicant-li el nom que li donarem al nostre graf GDS (*arbre1*), així com els nodes (etiquetes *Node*) i arestes (amb nom *REL*) que ens interessen:

```
CALL gds.graph.project('arbre1', 'Node', 'REL')
```

Que ens retorna un resum amb les característiques del graf GDS:

nodeProjection			relationshipProjection	
graphName	nodeCount	relationshipCount	projectMillis	
{Node: {label: "Node", properties: {}}}	{REL: {aggregation: "DEFAULT", orientation: "NATURAL", indexInverse: f}}	"arbre1"	9	8
			23	
			alse, properties: {}, type: "REL"}}	

El recorregut en amplitud està implementat en GDS mitjançant l'algorisme **gds.bfs**. En GDS els algorismes tenen diversos modes d'execució, que són diferents procediments que poden ser cridats mitjançant sentències Cypher. En el nostre cas, el que ens interessa és el mode **Stream**, que retorna els resultats com un conjunt de files de Cypher, similar a l'execució de qualsevol consulta Cypher. Així doncs, el procediment que hem d'utilitzar per a calcular un recorregut en amplitud és **gds.bfs.Stream**. Vegem com fer el recorregut en amplitud sobre el nostre graf GDS *arbre1*, partint del node "a":

```
MATCH (origen:Node{name:'a'})
CALL gds.bfs.stream('arbre1', {
  sourceNode: origen
})
YIELD path
RETURN path
```

Això ens retorna el camí (*path*) que segueix el recorregut:

path
(:Node {name: "a"})-[:NEXT]->(:Node {name: "a1"})-[:NEXT]->(:Node {name: "a2"})-[:NEXT]->(:Node {name: "a12"})-[:NEXT]->(:Node {name: "a11"})-[:NEXT]->(:Node {name: "a21"})-[:NEXT]->(:Node {name: "a22"})-[:NEXT]->(:Node {name: "a111"})-[:NEXT]->(:Node {name: "a112"})

Podem comprovar que el recorregut és correcte, amb una petita diferència respecte al que havíem vist abans: passa per "a12" abans que per "a11". La raó és que, en realitat, no hi ha cap ordre establert entre els fills d'un node, pot començar per qualsevol d'ells. L'important és veure que fa el recorregut per generacions, primer recorre els fills de "a", després els néts i després els besnéts.

El recorregut en profunditat està implementat mitjançant l'algorisme **gds.dfs** i el cridarem mitjançant el procediment **gds.dfs.stream**:



```

MATCH (origen:Node{name:'a'})
CALL gds dfs.stream('arbre1', {
  sourceNode: origen
})
YIELD path
RETURN path

```

Això ens retorna aquest camí (*path*):

```

path

(:Node {name: "a"})-[:NEXT]->(:Node {name: "a2"})-[:NEXT]->(:Node {name: "a22"})-[:NEXT]->(:Node {name: "a21"})-[:NEXT]->(:Node {name: "a1"})-[:NEXT]->(:Node {name: "a11"})-[:NEXT]->(:Node {name: "a112"})-[:NEXT]->(:Node {name: "a111"})-[:NEXT]->(:Node {name: "a12"})

```

Podem veure també que, com ja s'ha explicat abans, hi ha alguns canvis en l'ordre respecte del que havíem vist. Però l'important és que quan explora "a1", passa immediatament a explorar els seus fills "a11" i "a12". En aquest cas comença per "a11" i per tant, abans d'explorar "a12", explorarà primer els fills de "a11", que són "a112" i "a111".

Quan feim un recorregut, ja sigui en amplitud o profunditat, podem especificar un (o més) node destí, mitjançant el paràmetre *targetNodes*. El recorregut s'aturarà quan s'hi arribi. Per exemple:

```

MATCH (origen:Node{name:'a'}), (desti:Node{name: 'a21'})
CALL gds bfs.stream('arbre1', {
  sourceNode: origen,
  targetNodes: desti
})
YIELD path
RETURN path

```

Que retorna:

```

path

(:Node {name: "a"})-[:NEXT]->(:Node {name: "a1"})-[:NEXT]->(:Node {name: "a2"})-[:NEXT]->(:Node {name: "a11"})-[:NEXT]->(:Node {name: "a12"})-[:NEXT]->(:Node {name: "a21"})

```

I també podem definir una profunditat màxima a la qual volem arribar. Per exemple:

```

MATCH (origen:Node{name:'a'})
CALL gds dfs.stream('arbre1', {
  sourceNode: origen,
  maxDepth: 2
})
YIELD path
RETURN path

```

Que retorna:

path
(:Node {name: "a"})-[:NEXT]->(:Node {name: "a2"})-[:NEXT]->(:Node {name: "a22"})-[:NEXT]->(:Node {name: "a21"})-[:NEXT]->(:Node {name: "a1"})-[:NEXT]->(:Node {name: "a12"})-[:NEXT]->(:Node {name: "a11"})



AMPLIACIÓ

Podeu trobar més detalls sobre aquests dos algorismes a la documentació de Neo4j:

Recorregut en amplitud (BFS): <https://neo4j.com/docs/graph-data-science/current/algorithms/bfs/>

Recorregut en profunditat (DFS): <https://neo4j.com/docs/graph-data-science/current/algorithms/dfs/>

## 6.2. Cerca de camins mínims

El càlcul i l'obtenció de camins mínims dins d'un graf és un dels problemes clàssics en teoria de grafs, amb multitud d'aplicacions en el món real. Un camí dins d'un graf és un **conjunt de nodes i arestes que connecten un node origen i un node destí**. En el cas dels **grafs no pesats**, és a dir, aquells les arestes dels quals no tenen un cost associat, el càlcul del camí mínim s'obté minimitzant el nombre de salts entre el node origen i el node destí, és a dir, el nombre d'arestes en el camí. En el cas dels **grafs pesats**, aquells les arestes dels quals sí tenen un cost associat, el càlcul del camí mínim s'obté a través del camí en el qual la suma dels costos de les seves arestes és mínima.

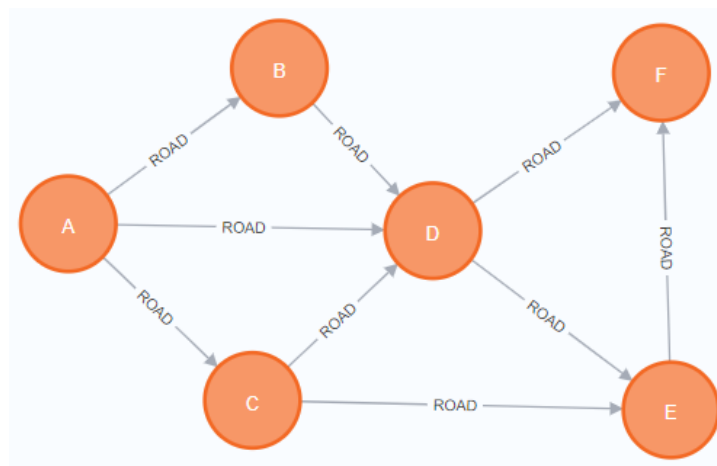
Un exemple típic de graf no pesat és el d'una xarxa de relacions entre persones. Els nodes són persones i les arestes podrien ser les relacions d'amistat. No hi ha cap pes associat a aquesta amistat, o són amics o no ho són. Per saber a quina distància es troba una persona d'una altra, hem de comptar per quantes relacions d'amistat hi podem arribar d'una a l'altra. En canvi, un exemple típic de graf pesat és el d'una xarxa de carreteres o ferrocarrils. Aquí els nodes serien ciutats de la xarxa i les arestes, les carreteres o línies de ferrocarril que les uneixen. Cada aresta té un pes associat (o diversos). Per exemple, en aquest graf podríem emprar dos pesos, la distància en quilòmetres o bé el temps de viatge. Per trobar el camí mínim no ens interessa quin és el que ha de passar per menys arestes, sinó el que té un pes total (la suma dels pesos de totes les seves arestes) menor. És a dir, el que té menys quilòmetres o menys temps de viatge, segons el cost que hàgim triat.

Juntament amb els recorreguts de grafs, els mètodes d'obtenció de camins mínims són molt útils quan es treballa en entorns dinàmics, on apareixen i desapareixen contínuament arestes del graf o els seus costos canvien dinàmicament. El càlcul de camins mínims també és molt útil en aplicacions que han de donar respostes en temps real.

### ■ Algorisme de Dijkstra

L'**algorisme de Dijkstra** (Edsger Dijkstra és un científic de la computació neerlandès, considerat un dels pares de la informàtica) és un dels algorismes més utilitzats en la teoria de grafs per a l'obtenció de camins mínims.

Vegem un exemple amb el següent graf que representa una xarxa de carreteres.



Imatge: Graf de carreteres

Cada aresta té una propietat *cost* que representa el cost per anar del node origen al node destí, podrien ser quilòmetres o minuts. Vegem la definició del graf en Cypher:

```
CREATE (a:Location {name: 'A'}),
      (b:Location {name: 'B'}),
      (c:Location {name: 'C'}),
      (d:Location {name: 'D'}),
      (e:Location {name: 'E'}),
      (f:Location {name: 'F'}),
      (a)-[:ROAD {cost: 50}]->(b),
      (a)-[:ROAD {cost: 50}]->(c),
      (a)-[:ROAD {cost: 100}]->(d),
      (b)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 80}]->(e),
      (d)-[:ROAD {cost: 30}]->(e),
      (d)-[:ROAD {cost: 80}]->(f),
      (e)-[:ROAD {cost: 40}]->(f);
```

Abans de poder aplicar qualsevol algorisme, ja hem explicat abans que hem de projectar el nostre graf a un graf GDS, al qual li direm *carreteres*. Aquí també hem d'especificar, a més dels nodes i arestes, quina és la propietat que emprarem per definir el pes de les arestes (*relationshipProperties*):

```
CALL gds.graph.project(
  'carreteres',
  'Location',
  'ROAD',
  {
    relationshipProperties: 'cost'
  }
)
```

L'algorisme de Dijkstra està implementat en GDS mitjançant **`gds.allShortestPaths.dijkstra`**. En concret ens interessa el mode Stream, per tant, haurem de cridar el procediment **`gds.allShortestPaths.dijkstra.stream`**. Podeu trobar els detalls a la [documentació de Neo4j](#).

Vegem com utilitzam l'algorisme de Dijkstra amb Cypher per trobar el camí mínim des de A fins a F:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.stream('carreteres', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```

El resultat és:

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs
	path				
0	"A"	"F"	160.0	["A", "B", "D", "E", "F"]	[0.0, 50.0, 90.0, 120.0, 160.0]
	[(:Location {name: "A"}), (:Location {name: "B"}), (:Location {name: "D"}), (:Location {name: "E"}), (:Location {name: "F"})]				

Podem veure que el camí més curt és A-B-D-E-F i té un cost total de 160. També podem veure en les columnes *nodeNames* i *costs* el cost dels camins mínims de cada fragment del camí: per anar des de A fins a A (cost 0), fins a B (50), fins a D (90), fins a E (120) i fins a F (160).

### ■ Algorisme A\*

L'**algorisme A\*** és un algorisme de cerca heurístic que permet, igual que l'algorisme de Dijkstra, calcular el camí mínim entre dos nodes donats qualssevol. Al contrari que l'algorisme de Dijkstra, quan es va a seleccionar el següent node del camí, A\* utilitza una funció heurística que fa una estimació del cost per arribar d'aquest node al següent i pren el que té una estimació menor. Aquesta tècnica, que es coneix també com de cerca informada, continua el recorregut del graf en cada iteració des del node amb el menor cost combinat entre la distància ja calculada i la funció heurística. Un exemple d'heurística seria utilitzar la distància en línia recta entre dos nodes i, per tant, prendre el que estigui més a prop com a principal candidat a ser el node següent del camí.

En GDS, l'algorisme A\* està implementat mitjançant l'algorisme **gds.shortestPath.astar** i en concret, ens interessa el mode Stream, així que ens centrarem en el procediment **gds.shortestPath.astar.stream**. Podeu trobar els detalls a la [documentació de Neo4j](#).

Per aplicar les heurístiques de l'algorisme A\*, hem de disposar de la longitud i latitud (o coordenades X i Y de cada node). Vegem un altre exemple de graf, en aquest cas d'estacions de metro de Londres:

```
CREATE (a:Station {name: 'Kings Cross', latitude: 51.5308, longitude: -0.1238}),
       (b:Station {name: 'Euston', latitude: 51.5282, longitude: -0.1337}),
       (c:Station {name: 'Camden Town', latitude: 51.5392, longitude: -0.1426}),
       (d:Station {name: 'Mornington Crescent', latitude: 51.5342, longitude: -0.1387}),
       (e:Station {name: 'Kentish Town', latitude: 51.5507, longitude: -0.1402}),
       (a)-[:CONNECTION {distance: 0.7}]->(b),
       (b)-[:CONNECTION {distance: 1.3}]->(c),
       (b)-[:CONNECTION {distance: 0.7}]->(d),
       (d)-[:CONNECTION {distance: 0.6}]->(c),
       (c)-[:CONNECTION {distance: 1.3}]->(e)
```

Projectam el nostre graf a un graf GDS, especificant les propietats dels nodes a tenir en compte per les heurístiques (*latitude* i *longitude*) i el pes de les arestes (*distance*):

```
CALL gds.graph.project(
  'metroLondres',
  'Station',
  'CONNECTION',
  {
    nodeProperties: ['latitude', 'longitude'],
    relationshipProperties: 'distance'
  }
)
```

Vegem com obtenim el camí mínim entre les estacions de Kings Cross i Kentish Town, fent servir l'algorisme A\*:

```

MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.aster.stream('metroLondres', {
  sourceNode: source,
  targetNode: target,
  latitudeProperty: 'latitude',
  longitudeProperty: 'longitude',
  relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index

```

El resultat és:

index	sourceNodeName	targetNodeName	totalCost	nodeNames
	costs	path		
0	"Kings Cross"	"Kentish Town"	3.3	["Kings Cross", "Euston", "Camden Town", "Kentish Town"]
	[0.0, 0.7, 2.0, 3.3]	[(:Station {name: "Kings Cross",latitude: 51.5308,longitude: -0.1238})		
		, (:Station {name: "Euston",latitude: 51.5282,longitude: -0.1337}),		
		Station {name: "Camden Town",latitude: 51.5392,longitude: -0.1426}),		
		:Station {name: "Kentish Town",latitude: 51.5507,longitude: -0.1402})]]		

Podem comprovar que el camí mínim és Kings Cross-Euston-Camden Town-Kentish Town i té un cost total de 3.3.

Aquest és el mateix resultat que ens dona Dijkstra:

```

MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.dijkstra.stream('metroLondres', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index

```

Que dona com a resultat:

index	sourceNodeName	targetNodeName	totalCost	nodeNames
	costs	path		
0	"Kings Cross"	"Kentish Town"	3.3	["Kings Cross", "Euston", "Camden Town", "Kentish Town"]
	[0.0, 0.7, 2.0, 3.3]	[(:Station {name: "Kings Cross", latitude: 51.5308, longitude: -0.1238})		
		, (:Station {name: "Euston", latitude: 51.5282, longitude: -0.1337}),		
		(:Station {name: "Camden Town", latitude: 51.5392, longitude: -0.1426}),		
		(:Station {name: "Kentish Town", latitude: 51.5507, longitude: -0.1402})		

La diferència entre Dijkstra i A\* és que en grafs molt grans, l'heurística de l'algorisme A\* fa que la seva execució sigui molt més ràpida. En canvi, el camí que retorna A\* no sempre serà el millor (el de Dijkstra sí que ho és).



Si tenim un graf molt gran i necessitem trobar el camí mínim molt ràpidament, en mil·lisegons, hem d'emprar l'algorisme **A\***. Però hem de ser conscients que potser hi ha altres camins millors.



En canvi, si necessitem ser precisos i estar segurs que el camí serà el mínim, hem d'emprar l'algorisme de **Dijkstra**.



AMPLIACIÓ

Podeu trobar més detalls sobre aquests algorismes de cerques de camins, així com d'altres suportats (incloent els de recorregut en amplitud i profunditat) per Neo4j a la documentació:

<https://neo4j.com/docs/graph-data-science/current/algorithms/pathfinding/>

## 6.3. Estudi de centralitat

La **centralitat** es defineix com la **rellevància d'un node dins d'un graf**. Per tant, l'estudi de mesures de centralitat permet identificar nodes rellevants dins d'un graf. Aquesta identificació permetrà entendre el comportament de la xarxa, quins nodes permeten viralitzar amb major rapidesa el contingut de la xarxa, des de quins nodes la informació està més accessible, etc. L'estudi de mesures de centralitat té multitud d'aplicacions, encara que són especialment notables les relacionades amb els àmbits de la publicitat i el màrqueting, on l'estudi d'aquestes mètriques permet identificar actors rellevants dins d'una xarxa als quals es pot contactar per a anunciar un producte o identificar sobre quins actors enviar informació per a aconseguir a més usuaris.

Respecte a les mesures de centralitat, no existeix una única mesura sinó que, en funció de les dades i del propòsit que es pretén aconseguir, s'utilitzen unes mètriques o altres. A continuació, es definiran tres mesures de centralitat amb les quals realitzarem exemples en Neo4j: **centralitat de grau**, **proximitat** i **intermediació**.

### ■ Centralitat de grau

El **grau d'un node** en un graf es defineix com el **nombre d'arestes o connexions que entren o surten del node**. Així doncs, en un node d'un graf es distingeixen dos graus: el **grau d'entrada**, corresponent al conjunt de connexions que entren a un node, i el **grau de sortida**, que es correspon amb el conjunt d'arestes que surten d'un node. El primer indica la prominència d'un node dins de la xarxa, mentre que el segon la influència del node dins del graf.

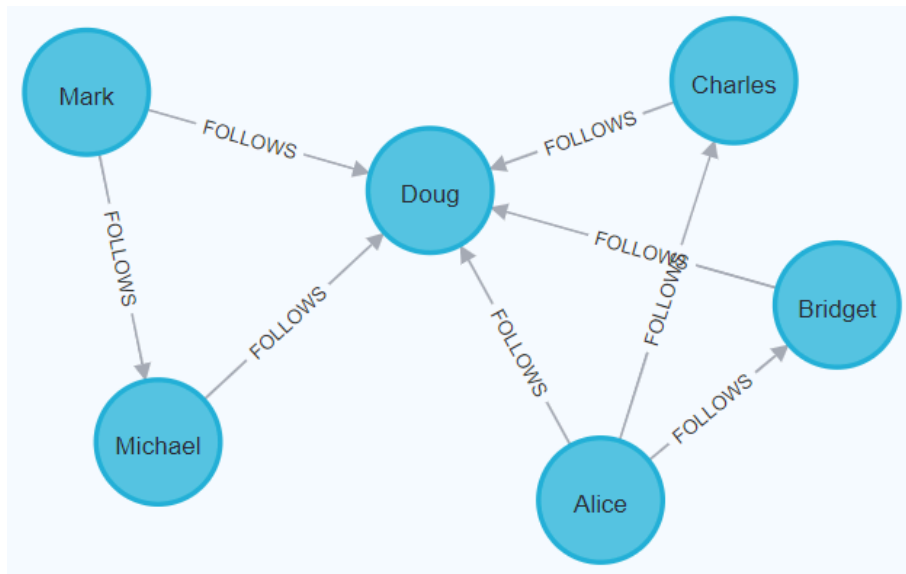
Aquesta mesura de centralitat és molt utilitzada per a identificar actors prominents o influents o per a identificar conductes fraudulentes, caracteritzades a vegades per una activitat anormal.

Vegem-ho amb aquest graf que representa una petita xarxa social:

```
CREATE
  (alice:User {name: 'Alice'}),
  (bridget:User {name: 'Bridget'}),
  (charles:User {name: 'Charles'}),
  (doug:User {name: 'Doug'}),
  (mark:User {name: 'Mark'}),
  (michael:User {name: 'Michael'}),
  (alice)-[:FOLLOWS {score: 1}]->(doug),
  (alice)-[:FOLLOWS {score: -2}]->(bridget),
  (alice)-[:FOLLOWS {score: 5}]->(charles),
  (mark)-[:FOLLOWS {score: 1.5}]->(doug),
  (mark)-[:FOLLOWS {score: 4.5}]->(michael),
  (bridget)-[:FOLLOWS {score: 1.5}]->(doug),
  (charles)-[:FOLLOWS {score: 2}]->(doug),
  (michael)-[:FOLLOWS {score: 1.5}]->(doug)
```

I aquesta és la representació gràfica del graf:





**Imatge:** Graf d'una petita xarxa social

Igual que amb els algorismes anteriors, en primer lloc hem de projectar-lo a un graf GDS (li donarem el nom *xarxasocial1*):

```
CALL gds.graph.project(
  'xarxasocial1',
  'User',
  {FOLLOWS: {properties: 'score'}}
)
```

L'algorisme que calcula la centralitat de grau en GDS és **gds.degree** i, com sempre, ens interessa el mode Stream, és a dir, el procediment **gds.degree.stream**. Per defecte, calcula el grau de sortida (compta el número d'arestes que surten de cada node). Però en aquest cas, per saber qui és l'usuari amb més influència, és a dir, amb més seguidors, hem de calcular el grau d'entrada (comptar el número d'arestes que entren en cada node). És per això que especifiquem la propietat *orientation* com a *'REVERSE'*. El valor per defecte d'orientation és *'NATURAL'* i també podem especificar el valor *'UNDIRECTED'*, que calcularà la suma de les arestes que entren i que surten (suma del grau d'entrada més el grau de sortida). Vegem com cridem a **gds.degree.stream** i ordenem els resultats, de més influència (major grau) a menys:

```
CALL gds.degree.stream('xarxasocial1', { orientation: 'REVERSE' })
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS followers
ORDER BY followers DESC, user DESC
```

El resultat és que l'usuari amb més influència (més seguidors) és Doug, amb un grau de 5:

user	followers
"Doug"	5.0
"Michael"	1.0
"Charles"	1.0
"Bridget"	1.0
"Mark"	0.0
"Alice"	0.0

És freqüent tenir en compte els pesos de les arestes en aquest càlcul. Xerram, en aquests casos, d'una centralitat de grau ponderada. Seguint amb el nostre exemple, seria així:

```
CALL gds.degree.stream(
  'xarxasocial1',
  { orientation: 'REVERSE',
    relationshipWeightProperty: 'score' }
)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS followers
ORDER BY followers DESC, user DESC
```

I el resultat és:

user	followers
"Doug"	7.5
"Charles"	5.0
"Michael"	4.5
"Mark"	0.0
"Bridget"	0.0
"Alice"	0.0

## ■ Proximitat

La **proximitat** (*closeness*) és una altra mesura de centralitat que determina **quins nodes del graf expandeixen ràpida i eficientment la informació** a través del graf. Per a calcular aquesta mesura, s'obté la **suma de l'invers de les distàncies d'un node a la resta**. D'aquesta manera, donat un node  $u$ , la proximitat del mateix es calcula mitjançant la següent fórmula:

$$C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$$

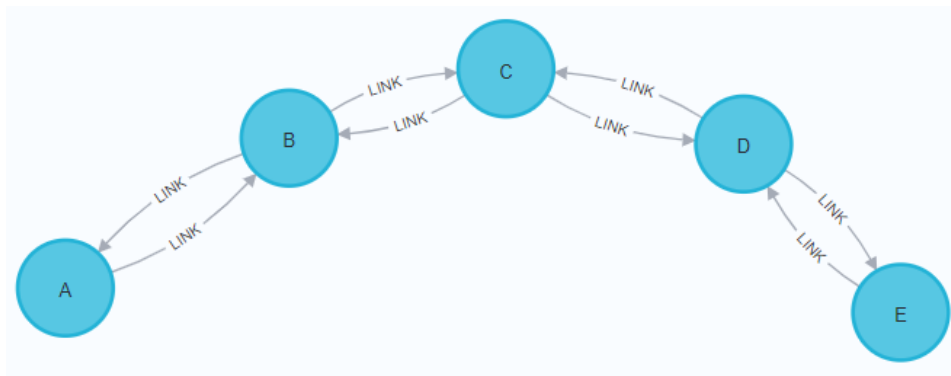
On  $n$  és el nombre de nodes del graf i  $d(u,v)$  és la distància del camí mínim entre  $u$  i  $v$ .

La proximitat és una mètrica molt utilitzada per a estimar el temps d'arribada en xarxes logístiques, per a descobrir actors en posicions privilegiades en xarxes socials o per a estudiar la prominència de paraules en un document en el camp de la mineria de textos.

Ho veurem més clarament amb aquest petit graf on els nodes formen una mena de cadena:

```
CREATE (a:Node {id:"A"}),
      (b:Node {id:"B"}),
      (c:Node {id:"C"}),
      (d:Node {id:"D"}),
      (e:Node {id:"E"}),
      (a)-[:LINK]->(b),
      (b)-[:LINK]->(a),
      (b)-[:LINK]->(c),
      (c)-[:LINK]->(b),
      (c)-[:LINK]->(d),
      (d)-[:LINK]->(c),
      (d)-[:LINK]->(e),
      (e)-[:LINK]->(d);
```

La representació gràfica del graf és:



**Imatge:** Graf en forma de cadena

Com sempre, primer hem de projectar el graf a un graf GDS (li direm *cadena*):

```
CALL gds.graph.project(
  'cadena',
  'Node',
  'LINK'
)
```

GDS proporciona l'algorisme **gds.closeness**, de manera que haurem de cridar el procediment **gds.closeness.stream**. Veurem com calcular la proximitat de tots els nodes del graf:

```
CALL gds.closeness.stream('cadena')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS user, score
ORDER BY score DESC
```

El resultat és:

user	score
"C"	0.6666666666666666
"D"	0.5714285714285714
"B"	0.5714285714285714
"E"	0.4
"A"	0.4

Podem veure que, com era d'esperar, C és el node millor connectat del graf, seguit per D i B. Els dos extrems, A i E, són els que tenen un pitjor valor de proximitat.

### ■ Intermediació

La **intermediació** (*betweenness*) és una altra mesura de centralitat que permet detectar la **influència que té un node o actor del graf en el flux d'informació o de recursos de la xarxa**. El càlcul de la intermediació permet identificar a nodes que fan de **colls de botella** en el graf, és a dir, **ponts entre diferents porcions del graf**. Aquesta mesura de centralitat és molt utilitzada per a la identificació d'*influencers* i l'estudi de la *virilització* de missatges en xarxes socials.

Intuïtivament, la intermediació d'un node serà major mentre aquest node aparegui en els camins mínims de qualsevol altre parell de nodes. Més formalment, la intermediació pot calcular-se segons aquesta fórmula:

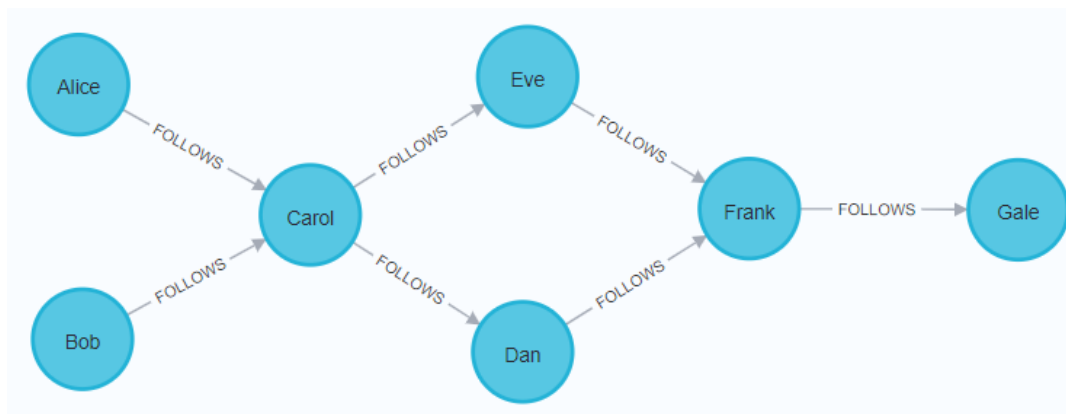
$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

On  $u$  és el node del qual es calcula la intermediació ( $B$ ),  $s$  i  $t$  són altres nodes del graf,  $p(u)$  és el número de camins mínims entre  $s$  i  $t$  que passen per  $u$  i  $p$  és el número total de camins mínims entre  $s$  i  $t$ .

Ho veurem més clar amb aquest graf que també representa una petita xarxa social:

```
CREATE
  (alice:User {name: 'Alice'}),
  (bob:User {name: 'Bob'}),
  (carol:User {name: 'Carol'}),
  (dan:User {name: 'Dan'}),
  (eve:User {name: 'Eve'}),
  (frank:User {name: 'Frank'}),
  (gale:User {name: 'Gale'}),
  (alice)-[:FOLLOWS {weight: 1.0}]->(carol),
  (bob)-[:FOLLOWS {weight: 1.0}]->(carol),
  (carol)-[:FOLLOWS {weight: 1.0}]->(dan),
  (carol)-[:FOLLOWS {weight: 1.3}]->(eve),
  (dan)-[:FOLLOWS {weight: 1.0}]->(frank),
  (eve)-[:FOLLOWS {weight: 0.5}]->(frank),
  (frank)-[:FOLLOWS {weight: 1.0}]->(gale);
```

Que té aquesta representació gràfica:



**Imatge:** Graf d'una altra petita xarxa social

Una vegada més, primer hem de projectar el graf a un graf GDS (li direm *xarxasocial2*):

```
CALL gds.graph.project(
  'xarxasocial2',
  'User',
  {FOLLOWS: {properties: 'weight'}}
)
```

GDS proporciona l'algorisme **gds.betweenness**, de manera que haurem de cridar el procediment **gds.betweenness.stream**. Veurem com calcular la intermediació de tots els nodes del graf:

```
CALL gds.betweenness.stream('xarxasocial2')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
```

El resultat és que Carol és l'usuària amb una intermediació més alta, seguida de Frank:

name	score
"Carol"	8.0
"Frank"	5.0
"Eve"	3.0
"Dan"	3.0
"Alice"	0.0
"Gale"	0.0
"Bob"	0.0

Si ens interessàs, també podríem obtenir una intermediació ponderada, tenint en compte els pesos de les arestes, igual que vàrem veure amb la centralitat de grau:

```
CALL gds.betweenness.stream('xarxasocial2', {relationshipWeightProperty: 'weight'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
```

El resultat és:

name	score
"Carol"	8.0
"Eve"	6.0
"Frank"	5.0
"Alice"	0.0
"Bob"	0.0
"Dan"	0.0
"Gale"	0.0



Podeu trobar més detalls sobre aquestes mesures de centralitat, així com d'altres suportades per Neo4j a la documentació:

<https://neo4j.com/docs/graph-data-science/current/algorithms/centrality/>

## 6.4. Detecció de comunitats

A l'hora de treballar amb grafs reals, que presenten una gran quantitat de nodes i enllaços, moltes vegades es pretén **identificar comunitats** dins del graf per a aplicar algorismes sobre elles. Una comunitat és, per tant, un **conjunt de nodes que presenten més relacions entre sí que amb la resta de nodes fora de la comunitat**.

Els algorismes de detecció de comunitats són emprats per evaluar com grups de nodes poden ser agrupats o particionats, així com la tendència d'aquestes agrupacions a enfortir-se o rompre's. Aquí veurem tres aproximacions diferents: **recompte de triangles**, **components fortament connexes** i **clustering (agrupació) amb k-means**.

La detecció i identificació de comunitats permet identificar comportaments emergents i de ramat dins d'una xarxa. D'aquesta manera, és possible detectar i predir hàbits dels usuaris.

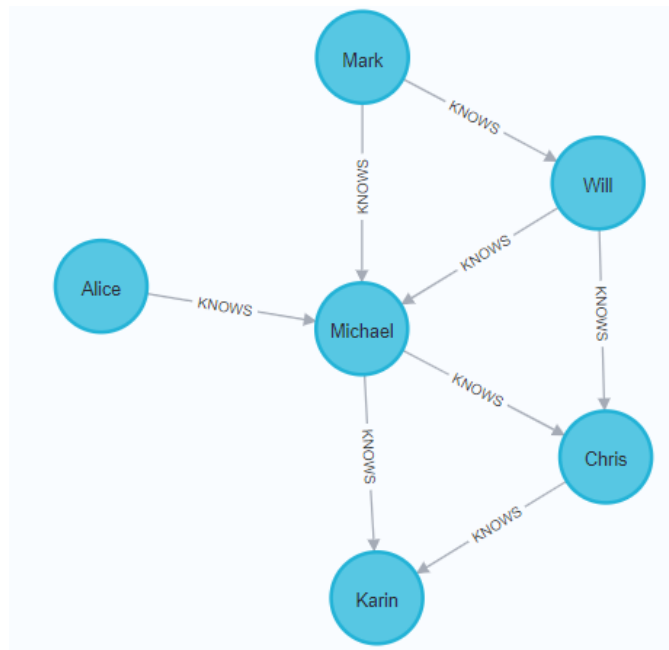
### ■ *Recompte de triangles (triangle count)*

Un triangle és un conjunt de tres nodes que tenen relacions entre sí. El recompte de triangles dona una mesura de com de cohesionada està una comunitat: com més triangles, més cohesionada. També aporta una idea de la seva fortalesa: si hi ha molts de triangles, encara que caiguin alguns enllaços, la comunitat pot seguir existint.

Farem feina amb el següent graf:

```
CREATE
(alice:Person {name: 'Alice'}),
(michael:Person {name: 'Michael'}),
(karin:Person {name: 'Karin'}),
(chris:Person {name: 'Chris'}),
(will:Person {name: 'Will'}),
(mark:Person {name: 'Mark'}),
(michael)-[:KNOWS]->(karin),
(michael)-[:KNOWS]->(chris),
(will)-[:KNOWS]->(michael),
(mark)-[:KNOWS]->(michael),
(mark)-[:KNOWS]->(will),
(alice)-[:KNOWS]->(michael),
(will)-[:KNOWS]->(chris),
(chris)-[:KNOWS]->(karin)
```

Que té aquesta representació gràfica:



**Imatge:** Graf per a la detecció de comunitats

Per aplicar el mètode del recompte de triangles (i altres similars com el dels coeficients locals d'agrupament) és necessari fer feina amb un graf GDS **no dirigit**. Per això, quan projectem el nostre graf, hem d'especificar que sigui amb *orientation UNDIRECTED*:

```
CALL gds.graph.project(
  'grafcomunitats',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

GDS proporciona l'algorisme **gds.triangleCount**, amb el procediment **gds.triangleCount.stream**. Vegem com calcular els triangles de cada node del graf:

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

El resultat és que Michael és qui pertany a més triangles (3), mentre que Alice no en pertany a cap. Això ens indica que si volguéssim agafar una persona d'aquesta comunitat que ens presentàs els seus amics, el més indicat seria Michael.



name	triangleCount
"Michael"	3
"Chris"	2
"Will"	2
"Karin"	1
"Mark"	1
"Alice"	0

També podem obtenir quins són els triangles del graf:

```
CALL gds.triangles('grafcomunitats')
YIELD nodeA, nodeB, nodeC
RETURN
  gds.util.asNode(nodeA).name AS nodeA,
  gds.util.asNode(nodeB).name AS nodeB,
  gds.util.asNode(nodeC).name AS nodeC
```

El resultat és:

nodeA	nodeB	nodeC
"Karin"	"Chris"	"Michael"
"Chris"	"Will"	"Michael"
"Will"	"Mark"	"Michael"

Un node pot pertànyer a un alt número de triangles, però també tenir molts altres enllaços que no formen triangles. Un altre mètode que té aquest fet en compte és l'anomenat del **coeficient local d'agrupament** (*local clustering coefficient*). Es basa també en comptar els triangles als quals pertany un node, però també està influït pel grau del node. No entrarem en més detalls, només direm que GDS ho implementa mitjançant l'algorisme **gds.localClusteringCoefficient** i el procediment **gds.localClusteringCoefficient.stream**.

### ■ Components fortament connexos

Mentre que el mètode del recompte de triangles ens dona una idea de quin és l'usuari amb qui hem de contactar per poder arribar millor a una comunitat, el mètode dels components fortament connexos ens permet trobar diferents comunitats dins d'un graf.

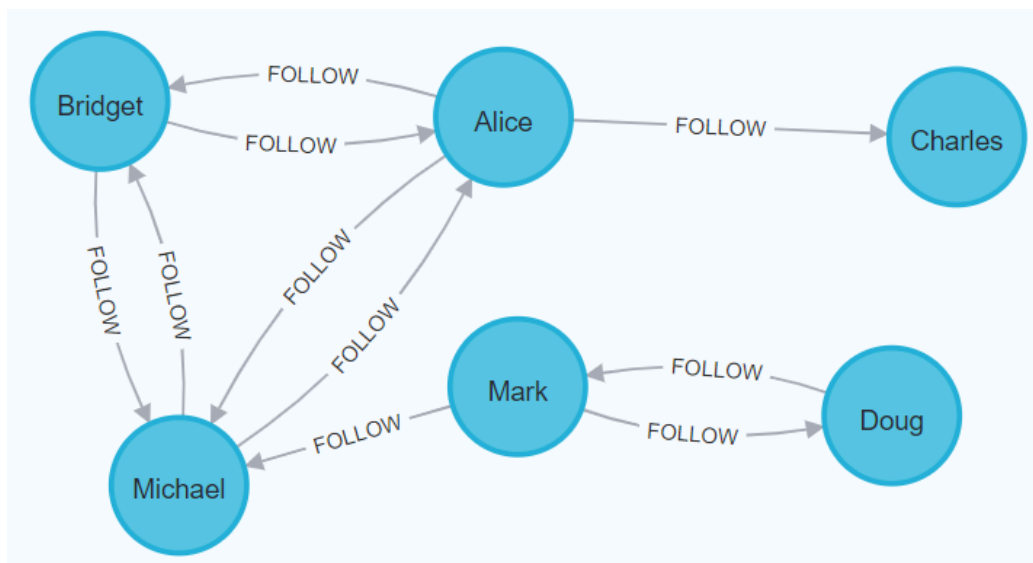
En un graf dirigit, un **component fortament connex** és aquell **grup de nodes en el qual, per a cada parell de vèrtexs  $u$  i  $v$ , existeix un camí per anar de  $u$  a  $v$  i un altre de  $v$  a  $u$** . Cada un d'aquests components fortament connexos conformen una comunitat dins el graf. Els diferents components fortament connexos formen el que s'anomena una partició del graf: tendrem una divisió en comunitats (subgrafs), on tots els nodes del graf pertanyen a una única comunitat.

L'estudi de les components fortament connexes en un graf permet estudiar la connectivitat de la xarxa.

Vegem-ho amb aquest graf dirigit:

```
CREATE (alice:User {name:'Alice'}),
(bridget:User {name:'Bridget'}),
(charles:User {name:'Charles'}),
(doug:User {name:'Doug'}),
(mark:User {name:'Mark'}),
(michael:User {name:'Michael'}),
(alice)-[:FOLLOW]->(bridget),
(alice)-[:FOLLOW]->(charles),
(mark)-[:FOLLOW]->(doug),
(mark)-[:FOLLOW]->(michael),
(bridget)-[:FOLLOW]->(michael),
(doug)-[:FOLLOW]->(mark),
(michael)-[:FOLLOW]->(alice),
(alice)-[:FOLLOW]->(michael),
(bridget)-[:FOLLOW]->(alice),
(michael)-[:FOLLOW]->(bridget);
```

La seva representació gràfica és:



**Imatge:** Exemple de graf dirigit

El projectam a un graf GDS:

```
CALL gds.graph.project('cfc', 'User', 'FOLLOW')
```

L'algorisme que implementa l'obtenció dels components fortament conexas és **gds.scc** i el procediment que haurem de cridar és **gds.scc.stream**. Vegem com els obtenim:

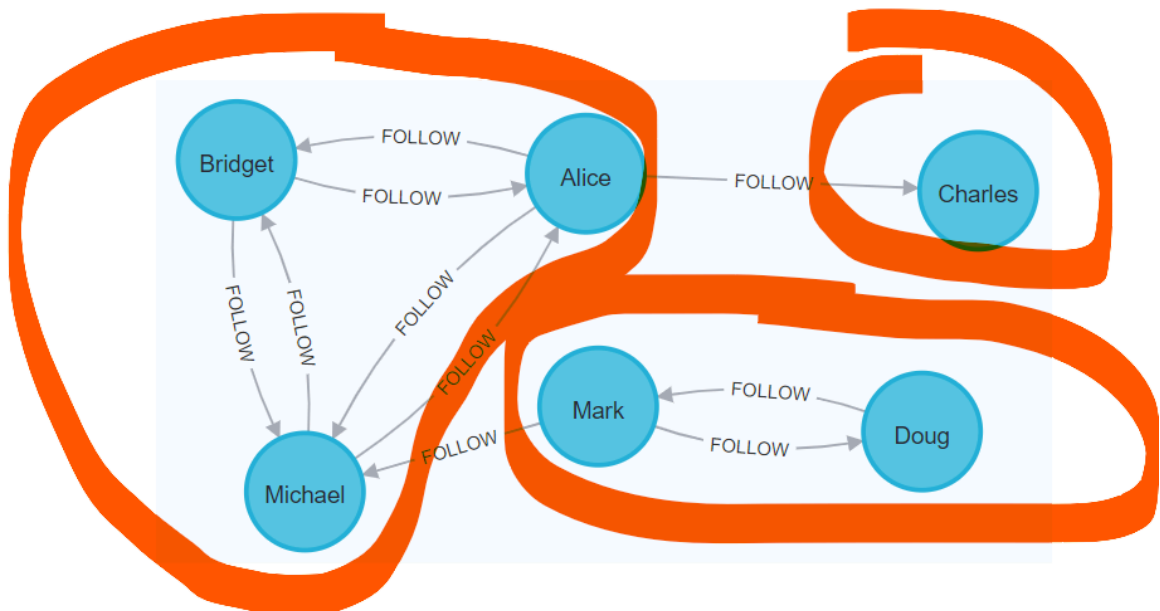
```
CALL gds.scc.stream('cfc', {})
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Community
ORDER BY Community DESC
```

El resultat és:

Name	Comunity
"Michael"	3
"Alice"	3
"Bridget"	3
"Doug"	1
"Mark"	1
"Charles"	0

Això ens indica que podem particionar el nostre graf en tres comunitats:

- Michael, Alice i Bridget
- Doug i Mark
- Charles



**Imatge:** Comunitats del graf

### ■ Clustering k-means

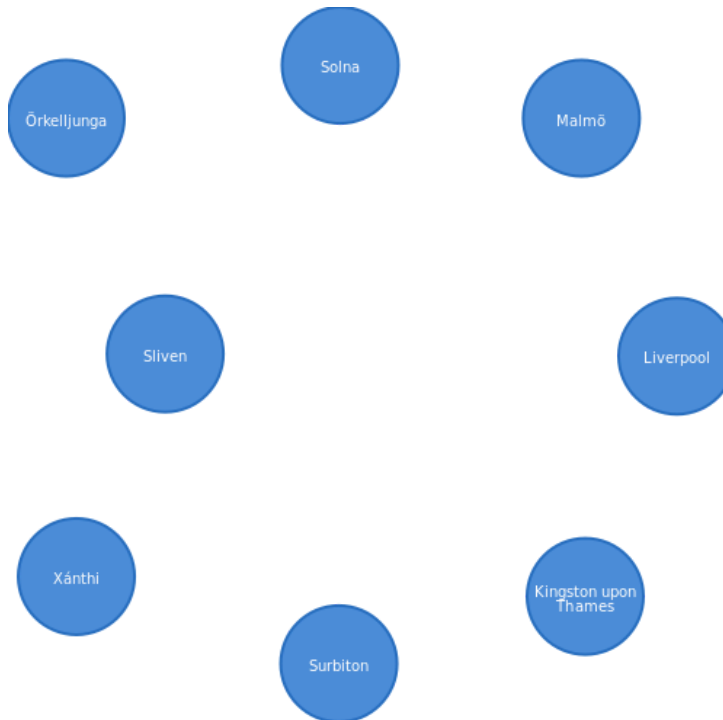
L'objectiu de l'algorisme k-means també és trobar comunitats o agrupacions (clústers) dins d'un graf, però amb una aproximació completament diferent. Aquí no es tenen en compte les relacions entre els nodes, sinó la seva disposició espacial. És per això que, per fer feina amb aquest algorisme necessitem tenir la longitud i latitud (coordenades X i Y) dels nodes.

k-means és un algorisme general per a clustering, que es veu en detall en el mòdul de Sistemes d'aprenentatge automàtic. Aquí veurem només un exemple del seu funcionament. Partirem d'aquest graf de ciutats, amb les seves coordenades de longitud i latitud:

CREATE

```
(:City {name: 'Surbiton', coordinates: [51.39148, -0.29825]}),
(:City {name: 'Liverpool', coordinates: [53.41058, -2.97794]}),
(:City {name: 'Kingston upon Thames', coordinates: [51.41259, -0.2974]}),
(:City {name: 'Sliven', coordinates: [42.68583, 26.32917]}),
(:City {name: 'Solna', coordinates: [59.36004, 18.00086]}),
(:City {name: 'Örkelljunga', coordinates: [56.28338, 13.27773]}),
(:City {name: 'Malmö', coordinates: [55.60587, 13.00073]}),
(:City {name: 'Xánthi', coordinates: [41.13488, 24.888]});
```

Com es pot comprovar, en aquest graf ni tan sols hem definit cap aresta, per al clustering amb k-means només consideram la localització dels nodes.



**Imatge:** Graf de ciutats

Com sempre, primer hem de projectar el graf a un graf GDS:

```
CALL gds.graph.project(
  'cities',
  {
    City: {
      properties: 'coordinates'
    }
  },
  '*',
  '*'
)
```

GDS incorpora l'algorisme **gds.kmeans**, amb el procediment **gds.kmeans.stream**.

Per aplicar l'algorisme de k-means hem de decidir en quants grups (clústers) volem dividir els nodes. Vegem-ho, per exemple, per a k=3 clústers:

```
CALL gds.kmeans.stream('cities', {
  nodeProperty: 'coordinates',
  k: 3
})
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
ORDER BY communityId, name ASC
```

El resultat és:

name	communityId
"Sliven"	0
"Xánthi"	0
"Malmö"	1
"Solna"	1
"Örkelljunga"	1
"Kingston upon Thames"	2
"Liverpool"	2
"Surbiton"	2

Podem observar que ens ha agrupat els nodes en 3 clústers o comunitats:

- 0: Sliven i Xánthi
- 1: Malmö, Solna i Örkelljunga
- 2: Kingston upon Thames, Liverpool i Surbiton



AMPLIACIÓ

Podeu trobar més detalls sobre aquests algorismes de detecció de comunitats, així com d'altres suportats per Neo4j a la documentació:

<https://neo4j.com/docs/graph-data-science/current/algorithms/community/>

## 6.5. Predicció d'enllaços

Els grafs són estructures de dades que representen sistemes dinàmics, que evolucionen al llarg del temps. Per aquest motiu, és molt comú que en un graf apareguin i desapareguin nous nodes i connexions entre aquests nodes. Els mètodes de predicció d'enllaços permeten predir quins enllaços es formaran pròximament entre els nodes del graf, permetent avançar-se als esdeveniments i prevenir eventualitats. Com a norma general, els mètodes de predicció d'enllaços es basen en mesures de proximitat i de centralitat, assumint que els nous enllaços es produiran, majoritàriament, al voltant dels nodes més rellevants.

El mètode més senzill de predicció d'enllaços és el dels **veïnats comuns** (*common neighbours*). Es basa en la idea genèrica que dos nodes de la xarxa que tenen una relació amb un node comú tindran més possibilitat de connectar-se entre sí. Formalment, donats dos nodes  $u$  i  $v$ , la possibilitat que es produeixi un enllaç entre ells ve donada per la fórmula següent:

$$VC(u,v) = |N(u) \cap N(v)|$$

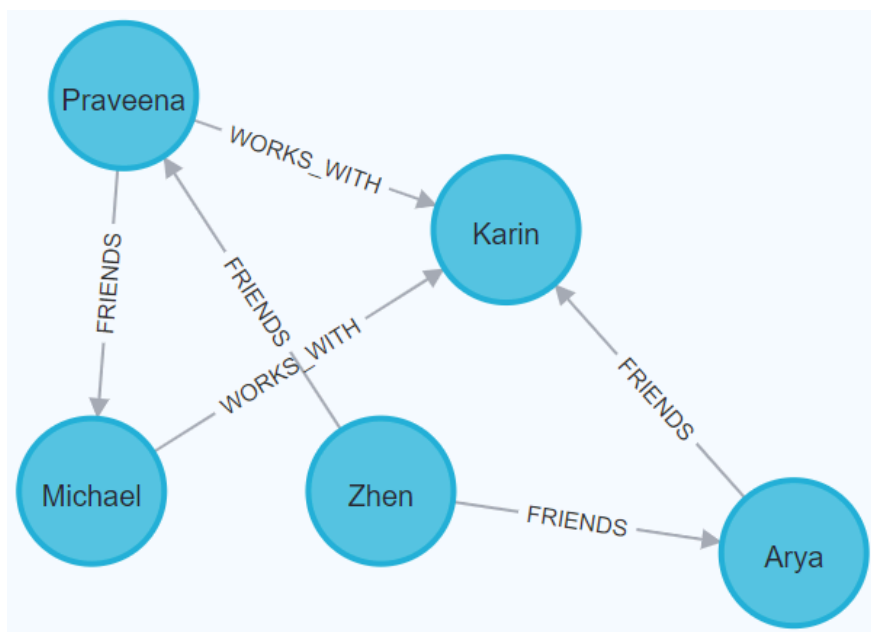
On  $N(u)$  i  $N(v)$  són els conjunts de nodes adjacents a  $u$  i a  $v$ , respectivament. Això vol dir que el número de veïnats comuns entre els nodes  $u$  i  $v$ , és el número de nodes que pertanyen a la intersecció dels conjunts de nodes adjacents a  $u$  i del de nodes adjacents a  $v$ .

Com més gran sigui el valor VC de dos nodes, més probable és que es produeixi un nou enllaç entre ells.

Vegem-ho amb aquest graf d'exemple:

```
CREATE (zhen:Person {name: 'Zhen'}),
      (praveena:Person {name: 'Praveena'}),
      (michael:Person {name: 'Michael'}),
      (arya:Person {name: 'Arya'}),
      (karin:Person {name: 'Karin'}),
      (zhen)-[:FRIENDS]->(arya),
      (zhen)-[:FRIENDS]->(praveena),
      (praveena)-[:WORKS_WITH]->(karin),
      (praveena)-[:FRIENDS]->(michael),
      (michael)-[:WORKS_WITH]->(karin),
      (arya)-[:FRIENDS]->(karin)
```

La seva representació gràfica és:



**Imatge:** Exemple de graf per a predir enllaços

Neo4j té el procediment **gds.alpha.linkprediction.commonNeighbors** per a calcular aquest valor VC. En aquest cas no és necessari treballar sobre un graf GDS, ho podem fer directament sobre un graf normal. Vegem com calculam el nombre de veïnats comuns entre Zhen i Karin:

```
MATCH (p1:Person {name: 'Zhen'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2) AS score
```

El resultat és 2, un valor alt tenint en compte que el graf només té 5 nodes. Això ens indica que és probable que acabi havent un enllaç entre ells.

## ■ Altres mètodes

El càlcul dels veïnats comuns és un mètode per a predir possibles enllaços. Però n'hi ha d'altres, més sofisticats i precisos.

Entre els més emprats, en podem destacar el d'**adhesió preferencial** (*preferential attachment*), implementat mitjançant el procediment `gds.alpha.linkprediction.preferentialAttachment`, i el d'**assignació de recursos** (*resource allocation*), implementat mitjançant el procediment `gds.alpha.linkprediction.resourceAllocation`. Tots dos, que també es poden executar sobre un graf normal (no GDS), retornen un valor que ens dona una indicació de la possibilitat que s'afegeixi un enllaç entre dos nodes: a valors més alts, major possibilitat.



AMPLIACIÓ

Podeu trobar més detalls sobre aquests mètodes de predicció d'enllaços, així com d'altres suportats per Neo4j a la documentació:

<https://neo4j.com/docs/graph-data-science/current/algorithms/linkprediction/>