6.- Bases de datos NoSQL: Grafos

En esta práctica tomaremos contacto con la base de datos de grafos de uso más extendido, Neo4J. Se proporcionan instrucciones de instalación de la versión Community Edition en Ubuntu 20.04, veremos la sintaxis básica del lenguaje Cypher y como cargar datos en la base de datos, tanto desde archivos .csv como usando una conexión directa con una base de datos relacional.

6.1 Instalación de Neo4J Community Edition en Ubuntu 20.04

En la máquina virtual GreiBD proporcionada para esta materia, ya hay una instalación de Neo4J Community Edition, con lo que no es necesario proceder a su instalación, tal y como se describe en esta sección. Este material está aquí solo para que pueda utilizarse de querer instalar el software en otra máquina.

Requisitos de hardware y software.

- Cualquier plataforma de tipo x68 de 64 bits
- Hardware (uso personal): Mínimo core i3 (recomendado core i7). Mínimo 2GB de RAM (recomendado 16GB). Mínimo 10GB de disco.
- Software: Mínimo Ubuntu 16. OpenJDK 11 (la versión debe de ser exactamente la 11 para Neo4J 4.x, no sirve una posterior).

Instalación en sistemas operativos basados en Debian (Debian, Ubuntu, etc.). Es necesario tener instalado el OpenJDK 11. Para comprobar esta situación podemos usar los siguientes comandos. Con este primer comando vemos la lista de versiones de java instaladas.

```
sudo update-java-alternatives --list
```

En la máquina virtual GreiBD, están instaladas la versión 11 y la 13. Con el siguiente comando indicamos que la versión por defecto será la 11.

```
sudo update-java-alternatives --jre --set java-1.11.0-openjdk-amd64
```

Para comprobar que la versión por defecto es la 11 ejecutamos el siguiente comando.

```
java -version
```

Ahora ya podemos instalar Neo4J. Instalaremos la versión 4.2.1. Primero preparamos el acceso al repositorio.

```
wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo apt-key add -
echo 'deb https://debian.neo4j.com stable latest' | sudo tee -a /etc/apt/sources.list.d/neo4j.list
sudo apt-get update
```

Nos aseguramos que el repositorio universe está habilitado.

```
sudo add-apt-repository universe
```

Ahora instalamos el software.

```
sudo apt-get install neo4j=1:4.2.2
```

?

Las localizaciones de los archivos instalados pueden verse en la siguiente página web. La configuración se encuentra en el archivo `neo4j.conf` en `/etc/neo4j`.

Para la gestión del proceso de Neo4J como un servicio en Ubuntu, se usan los siguientes comandos.

Habilitar el servicio para inicio automático:	`systemctl enable neo4j`
Deshabilitar el servicio para inicio automático:	`systemctl disable neo4j`
Comprobar si el servicio está habilitado:	`systemctl is-enabled neo4j`
Iniciar, parar o reiniciar el servicio:	`systemctl {start|stop|restart} neo4j`
Comprobar el estado del servicio:	`systemctl status neo4j`
Comprobar si está activo el servicio:	`systemctl is-active neo4j`
Modificar la configuración del servicio:	`systemctl edit neo4j`

Opcionalmente, se puede instalar la librería APOC, que nos permitirá realizar entre otras cosas, exportar datos en formato csv, o la consulta de datos de una base de datos relacional. Hay dos versiones de la librería.

APOC Core
Permite importar y exportar datos en formato csv

APOC Full
Permite además por ejemplo la conexión con bases de datos relacional a través de su driver JDBC. Para que funcione esto será necesario también

El jar de la versión Core ya debería de estar en la ruta `/var/lib/neo4j/labs`, y tenemos que copiarlo en `/var/lib/neo4j/plugins`. Si lo que queremos es instalar la versión Full, podemos descargarla de la siguiente página. La versión apropiada que necesitamos descargar para cada versiónd e Neo4J se describe en una tabla de la página de instalación. Por ejemplo, para la versión 4.2.2 de Neo4J, necesitamos la versión 4.2.0.1 de APOC. Una vez descargado el jar, lo colocamos en `/var/lib/neo4j/plugins` y cambiamos el propietario y el grupo del archivo, para que el usuario neo4j pueda accederlo. Necesitamos reiniciar el servicio para que funcione.

```
sudo chown neo4j:adm apoc-4.2.0.1-all.jar
```

Recordar que para que funcione la conexión JDBC con bases de datos relacionales, los drivers JDBC de las bases de datos deben de estar también en `/var/lib/neo4j/plugins`, con propietario neo4j y grupo adm.

En la máquina virtual GreiBD, ya están instalada la versión 4.2.2 de Neo4J con APOC Full y el driver JDBC de Postgesql. Con lo que para empezar a trabajar solo hará falta iniciar el servicio como se indica arriba. El usuario y contraseña establecidos son:

usuario: neo4j **password**: greibd2021
Para establecer el password inicial de Neo4J se puede usar la herramienta de administración neo4j-admin, tal y como sigue.

```
sudo neo4j-admin set-initial-password greibd2021
```

El password inicial del usuario neo4j e también neo4j. A pesar de establecer el password con `neo4j-admin`, puede ser solicitado el cambio de password de nuevo al iniciar `cypher-shell`. bnmbn

6.2 Gestión de bases de datos

Arrancamos el servicio e iniciamos la shell.

```
sudo systemctl start neo4j

cypher-shell
```

Los siguientes comandos permiten gestionar bases de datos dentro de Neo4J.

Muestra las bases de datos:	`show databases;`
Muestra la base de datos por defecto:	`show default database;`
Muestra una base de datos concreta:	`show database <nombre>;`
Inicia la base de datos:	`start database <nombre>;`
Para la base de datos:	`stop database <nombre>;`
Crea una base de datos (solo en Enterprise Edition):	`create database <nombre>;`
Elimina una base de datos (solo en Enterprise Edition):	`drop database <nombre>;`

La edición Community Edition solo admite una base de datos con datos del usuario llamada `neo4j`, y una base de datos con datos del sistema, llamada `system`.

Una base de datos parada (stop) puede iniciarse de nuevo con start. Cuando no está iniciada, no estará disponible para consulta. Una base de datos borrada (drop) se elimina del sistema permanentemente y no podrá ser recuperada. Los nombres de bases de datos se normalizan a mayúsculas. Los nombres que empiezan por guión bajo (_) y con el prefijo system se reservan para uso interno. Los parámetros de configuración, como el nombre de la base de datos por defecto o el número máximo de bases de datos, se almacenan en el archivo de configuración `neo4j.conf`.

6.3 Gestión de datos con el lenguaje Cypher

El material que usamos aquí está basado en la página del manual de introducción a Cypher. Más detalles pueden consultarse en el manual de Cypher.

Con la clausula create podemos añadir elementos al grafo usando patrones. La respuesta de cypher es el número de elementos creados. Con este comando creamos un nodo etiquetado como "Movie", con valores para las propiedades "title" y "released".

```
CREATE (:Movie { title:"The Matrix",released:1997 });
```

Podemos añadir una cláusula return para devolver los valores de las variables que se usan durante la creación. En este caso vemos el contenido de la variable p, es decir,

```
CREATE (p:Person { name:"Keanu Reeves", born:1964 })
CREATE (p:Person {name:"Halle Berry"})
RETURN p;
```

Se pueden crear varios elementos dentro de la misma cláusula CREATE y también usar varias cláusulas CREATE en la misma sentencia. En la siguiente sentencia estaríamos creando tres nodos y dos relaciones.

```
CREATE (a:Person { name:"Tom Hanks", born:1956 })
       -[r:ACTED_IN { roles: ["Forrest"]}]->
       (m:Movie { title:"Forrest Gump",released:1994 })
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[:DIRECTED]->(m)
RETURN a,d,r,m;
```

Muy habitualmente, necesitamos enlazar elementos que estamos creando con elementos que ya están en la base de datos, para eso necesitaremos encajar patrones existentes en la base de datos usando la cláusula MATCH. En el ejemplo siguiente, creamos una nueva película y la enlazamos con el nodo de "Tom Hanks" que ya había sido creado antes.

```
match (p:Person { name:"Keanu Reeves"})
match (m:Movie { title:"The Matrix"})
create (p)-[r:ACTED_IN { roles: ['Neo']}]->(m);

MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Dr. Henry Goose', 'Hotel Manager', 'Isaac Sachs',
                                 'Dermot Hoggins', 'Cavendish', 'Look-a-Like Actor','Zachry']}]->(m)
RETURN p,r,m;

MATCH (p:Person {name:"Halle Berry"})
MATCH (m:Movie { title:"Cloud Atlas"})
CREATE (p)-[r:ACTED_IN {roles:['Native Woman' , 'Jocasta Ayrs', 'Luisa Rey',
                               'Indian Party Guest', 'Ovid', 'Meronym']}]->(m)
```

Las cláusulas CREATE se ejecutan una vez por cada fila de valores de las variables encajadas en el patrón MATCH. Ahora podemos comprobar que "Tom Hanks" está ya asociado a dos películas.

```
match (:Person {name:"Tom Hanks"}) -[r:ACTED_IN]-> (m:Movie)
return m.title, r.roles;
```

La cláusula MERGE funciona como un MATCH y un CREATE que podría ejecutarse un número ilimitado de veces y que dejaría la base de datos en el mismo estado. La cláusula intenta hacer el MATCH, y si no lo encuentra entonces hace el CREATE. Permite especificar propiedades adicionales que se modificarán después de la creación. En este caso, si no existe una película con titulo "Cloud Atlas", se crea, y durante la creación se asigna el valor 2012 al atributo "released".

```
MERGE (m:Movie { title:"Cloud Atlas" })
ON CREATE SET m.released = 2012
RETURN m;
```

Si el patrón que colocamos en MERGE existe solo parcialmente, se creará de nuevo entero, obteniendo datos duplicados. Si necesitamos modificar propiedades directamente podemos usar SET después de MATCH. Por ejemplo, para cambiar la propiedad "released" a 2013.

```
MATCH (m:Movie { title:"Cloud Atlas" })
SET m.released = 2013
RETURN m;
```

Para borrar elementos (nodos o relaciones) usamos la cláusula DELETE después de haberlos localizado con MATCH. Si intentamos borrar un nodo que tiene relaciones, obtendremos un error. Para forzar el borrado en cascada de relaciones al borrar nodos podemos usar la cláusula DETACH DELETE. También se pueden eliminar propiedades usando REMOVE.

```
MATCH (m:Movie { title:"Cloud Atlas" })
REMOVE m.released
RETURN m;

MATCH (m:Movie { title:"Cloud Atlas" })
DELETE m;

MATCH (m:Movie { title:"Cloud Atlas" })
DETACH DELETE m;

MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m)
RETURN p,r,m;
```

La cláusula `MATCH` permite filtrar elementos, pero solo utilizando predicados de igualdad. Para establecer otro tipo de predicados se puede utilizar la cláusula `WHERE`.

```
MATCH (m:Movie)
WHERE m.title = "Forrest Gump"
RETURN m;
```

Ojo con como funciona esta clausula. En la siguiente consulta la misma persona no puede ser p1 y p2 en la misma fila del resultado.

```
MATCH (p1:Person)-[:ACTED_IN]->(m:Movie{title:"Cloud Atlas"})<-[:ACTED_IN]-(p2:Person)
RETURN p1, p2
```

También se pueden filtrar por la etiqueta en la cláusula `WHERE`.

```
MATCH (n)
WHERE n:Person OR n:Movie
return n;
```

En el siguiente ejemplo, vemos como combinar varios predicados y como usar distintos operadores, en este caso el operador `=~` para utilizar expresiones regulares (sintaxis de expresiones regulares de java), y el operador `IN` para comprobar si un elemento está contenido en una lista (o array).

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "T.+" OR m.released > 2000 OR "Forrest" IN r.roles
RETURN p,r,m
```

También se pueden usar patrones en la cláusula WHERE. En este caso se buscan personas que actuaron en alguna película, pero que no dirigieron ninguna película.

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

De esta forma podemos obtener por ejemplo los nodos que no están relacionados con ningún nodo.

```
match (n)
where not (n)--()
return n;
```

Como ya hemos visto en los ejemplos anteriores, las sentencias de Cypher devuelven filas utilizando una cláusula `RETURN`. En este ejemplo, vemos como se pueden usar literales de tipos numéricos, cadenas de caracteres, arrays y maps, para construir una fila.

```
return 3 as id,
       "ventas" as departamento,
       [{nombre:"pepe", edad:23},
        {nombre:"juan", edad:45},
        {nombre:"Ernesto",idade:45}] as empleados;
```

Para acceder a las propiedades se puede usar la sintaxis `elemento.propiedad`. También podemos usar índices para acceder a elementos concretos de los arrays.

```
match(:Person)-[r:ACTED_IN]->(:Movie)
return r.roles[0];
```

Podemos utilizar también funciones definidas en el sistema (`length(array), toInteger("12"), coalesce(p.a,"n/a")`, ...). También podemos utilizar la palabra clave `DISTINCT` para eliminar filas duplicadas, igual que hacemos en SQL.

```
MATCH (n)
RETURN DISTINCT labels(n) AS Labels, labels(n)[0] as label, size(labels(n)) as size;
```

La agrupación y la agregación se gestionan de forma automática al combinar funciones de agregado con variables en la cláusula `RETURN`.

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor, director, count(*) AS collaborations;
```

También podemos ordenar las filas del resultado (`ORDER BY`) y controlar la paginación usando `SKIP` y `LIMIT`.

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a, count(*) AS appearances
ORDER BY appearances DESC LIMIT 2;

MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a, count(*) AS appearances
ORDER BY appearances DESC SKIP 2 LIMIT 2;
```

Al igual que en SQL, podemos construir arrays usando funciones de agregado.

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name) AS cast, count(*) AS actors;
```

Las palabras clave `UNION` y `WITH` nos permiten componer sentencias más largas a partir de otras más pequeñas. Por ejemplo, la siguiente consulta obtiene los nombres de las personas que actuaron o dirigieron una película, indicando si actuaron o dirigieron y el título de la película.

```
MATCH (actor:Person)-[r:ACTED_IN]->(movie:Movie)
RETURN actor.name AS name, type(r) AS type, movie.title AS title
UNION
MATCH (director:Person)-[r:DIRECTED]->(movie:Movie)
RETURN director.name AS name, type(r) AS type, movie.title AS title
```

La cláusula WITH se usa igual que la cláusula RETURN, pero solo para generar un resultado intermedio que sirva de entrada en la consulta siguiente. Por ejemplo, en la siguiente consulta nos permite filtrar en base al resultado de un agregado previamente calculado.

```
MATCH (person:Person)-[:ACTED_IN]->(m:Movie)
WITH person, count(*) AS appearances, collect(m.title) AS movies
WHERE appearances > 1
RETURN person.name, appearances, movies
```

Para terminar esta sección veremos como se pueden crear índices para acelerar las consultas y restricciones que nos permitan asegurar que los datos se ajustan a un dominio de aplicación concreto. Los índices se pueden crear en cualquier momento con una instrucción `CREATE INDEX`.

```
CREATE INDEX FOR (a:Person) ON (a.name);
```

Para ver los índices creados podemos usar el siguiente comando.

```
show indexes;
```

Para borrar un índice usamos el comando DROP INDEX. El nombre del índice que necesitamos para borrarlo lo obtenemos de la información del comando show indexes. También se pude borrar el índice por la etiqueta y propiedad usadas para crearlo.

```
drop index index_5c0607ad;

drop index on :Person(name);
```

Podemos crear también restricciones de tipo UNIQUE. Cuando se crea una restricción de estas, se crea también un índice del mismo tipo.

```
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE

show constraints;

DROP CONSTRAINT constraint_e26b1a8b;
```

6.4 Importación de pequeñas cantidades de datos con el comando LOAD

En esta sección vamos a ver algunos ejemplos de uso de la cláusula `LOAD` para importar datos obtenidos en archivos CSV. Con esta cláusula tenemos control sobre como importamos los datos, y como los integramos con el grafo existente, sin embargo no está diseñada para importar grandes cantidades de datos. Para eso usaremos el comando `neo4j-admin import`. Sin embargo, `neo4j-admin import` no puede usarse sobre una base de datos que ya haya sido utilizada ya, y dada la limitación que tienes la edición Community para usar varias bases de datos, no es recomendable su uso con esta edición.

Antes de empezar usaremos SQL para generar los archivos necesarios a partir de la base de datos de películas que tenemos en PostgreSQL. Exportaremos a CSV con un formato concreto los datos de las 20 películas de mayor presupuesto.

```
copy
(select id, titulo, presupuesto, fecha_emision, ingresos, duracion
from peliculas
order by presupuesto desc, id
limit 20)
to '/home/alumnogreibd/public/peliculas.csv'
with csv header

copy
(select pel.id as id_pelicula, per.id as id_persona, per.nombre as nombre, pp.trabajo as trabajo
from (select id
      from peliculas p
      order by presupuesto desc, id
      limit 20) as pel,
     pelicula_personal pp, personas per
where pel.id = pp.pelicula and pp.persona=per.id)
to '/home/alumnogreibd/public/personal.csv'
with csv header

copy
(select pel.id as id_pelicula, per.id as id_persona, per.nombre as nombre, pr.personaje as personaje
from (select id
      from peliculas p
      order by presupuesto desc, id
      limit 20) as pel,
     pelicula_reparto as pr, personas as per
where pel.id = pr.pelicula and pr.persona = per.id)
to '/home/alumnogreibd/public/reparto.csv'
with csv header
```

Vaciamos ahora la base de dato Neo4J, para tener solo el grafo que vamos a importar. Limitamos el borrado a 10000 nodos para no tener problemas en la ejecución del comando. Si después de borrar 10000 nodos todavía quedan datos, repetimos el comando las veces necesarias.

```
match(n) with n limit 10000 detach delete n;
```

Para asegurarnos de que los datos se importan correctamente, y que después van a poder accederse de forma ágil, podemos definir algunas restricciones e índices.

```
CREATE CONSTRAINT idPelicula ON (p:Pelicula) ASSERT p.id IS UNIQUE;
CREATE CONSTRAINT idPersona ON (p:Persona) ASSERT p.id IS UNIQUE;
CREATE INDEX FOR (p:Pelicula) ON (p.titulo);
CREATE INDEX FOR (p:Persona) ON (p.nombre);
```

Para poder importar los datos, debemos moverlos al directorio de importación `/var/lib/neo4j/import`. Generamos ahora un nodo para cada película que hemos exportado en el archivo "peliculas.csv".

```
LOAD CSV WITH HEADERS FROM "file:///peliculas.csv" AS csvPelicula
CREATE (p:Pelicula {id: toInteger(csvPelicula.id),
                    titulo: csvPelicula.titulo,
                   presupuesto: csveplicula.presupuesto,
                   fechaEmision: date(csvPelicula.fecha_emision),
                   ingresos: toInteger(csvPelicula.ingresos),
                   duracion: toInteger(csvPelicula.duracion)});
```

Comprobamos que los datos han sido importados con una consulta simple.

```
MATCH (p) return (p);
```

Ahora vamos a importar los datos de los créditos de cada película, empezando por el personal que ha trabajado en la película. Antes de proceder a importar los datos, comprobamos que las condiciones que vamos a colocar funciona con una consulta.

```
LOAD CSV WITH HEADERS FROM "file:///personal.csv" AS csvPersonal
MATCH (pe:Pelicula {id:toInteger(csvPersonal.id_pelicula)})
return pe.titulo, csvPersonal.nombre, csvPersonal.trabajo;

USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///personal.csv" AS csvPersonal
MATCH (pe:Pelicula {id:toInteger(csvPersonal.id_pelicula)})
MERGE (p:Persona {id:toInteger(csvPersonal.id_persona)})
ON CREATE SET p.nombre = csvPersonal.nombre
CREATE (p)-[:TRABAJO_EN {trabajo:csvPersonal.trabajo}]->(pe);
```

Comprobamos ahora con una consulta que los datos están importados. Recupera para cada película cuyo título empiece por "Pirat", su id, su título, el número de personas que han trabajado en la película, y un mapeao con la lista de nombres y trabajos de las personas involucradas.

```
match (persona)-[t:TRABAJO_EN]->(pelicula)
WHERE pelicula.titulo =~ "Pirat.+"
return pelicula.id, pelicula.titulo,
       count(*) as numPersonas,
       collect({nombre:persona.nombre,trabajo:t.trabajo}) as personal;
```

Para terminar, importamos los datos del reparto de la película, es decir, de las personas que actuaron en la película y de los personajes que interpretaron.

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///reparto.csv" AS csvReparto
MATCH (pe:Pelicula {id:toInteger(csvReparto.id_pelicula)})
MERGE (p:Persona {id:toInteger(csvReparto.id_persona)})
ON CREATE SET p.nombre = csvReparto.nombre
CREATE (p)-[:ACTUO_EN {personaje:csvReparto.personaje}]->(pe);
```

Comprobamos que los datos están cargados con una consulta simple.

```
match (p:Persona)-[r:ACTUO_EN]->(pe)
WHERE r.personaje =~ ".*Jack.*"
return p.nombre, r.personaje, pe.titulo, pe.fechaEmision;
```

6.5 Consulta de datos relacionales con JDBC

Una alternativa a la importación de datos CSV, cuando los datos ya están almacenados dentro de un SGBDs relacional, es el uso de JDBC para acceder directamente gestor y recuperar los datos. Para pode hacer esto, debemos tener instalada la versión FULL de APOC y tener el driver JDBC del gestor en la carpeta de plugins. Para ilustrar esto, vamos a realizar las consultas necesarias para importar datos de películas directamente de nuestra bases de datos PostgreSQL.

Empezamos generando los datos de las películas.

```
with "jdbc:postgresql:xine?user=alumnogreibd&password=greibd2021" as url
CALL apoc.load.jdbc(url,
"select id, titulo, presupuesto, fecha_emision, ingresos, duracion
from peliculas order by presupuesto desc, id limit 1000") yield row
CREATE (p:Pelicula {id:row.id,
                    titulo:row.titulo,
                   presupuesto: row.presupuesto,
                   fechaEmision:row.fecha_emision,
                   ingresos:row.ingresos,
                   duracion:row.duracion});
```

Ahora generamos los nodos de las personas.

```
with "jdbc:postgresql:xine?user=alumnogreibd&password=greibd2021" as url
CALL apoc.load.jdbc(url,
"select distinct id, first_value(nombre) over (partition by id order by nombre) as nombre
from (select per.id as id, per.nombre as nombre
          from pelicula_personal as pp, personas as per
          where pp.persona = per.id and pp.pelicula in (select id from peliculas order by presupuesto desc, id limit 1000)
          union all
          select per.id as id, per.nombre as nombre
          from pelicula_reparto as pr, personas as per
          where pr.persona = per.id and pr.pelicula in (select id from peliculas order by presupuesto desc, id limit 1000)) as t") yield
row
CREATE (p:Persona {id:row.id,
                   nombre:row.nombre});
```

Para acelerar la inserción de las relaciones, añadimos primero índices sobre las Peliculas y Personas, para que su búsqueda en cada inserción se realice más rápido. Vamos a añadir también índices sobre el nombre de la persona y sobre el Título de la película para acelerar consultas sobre estos campos.

```
CREATE INDEX FOR (a:Persona) ON (a.id);
CREATE INDEX FOR (a:Pelicula) ON (a.id);
CREATE INDEX FOR (a:Persona) ON (a.nombre);
CREATE INDEX FOR (a:Pelicula) ON (a.titulo);
```

Ahora ya podemos proceder a insertar las relaciones que definen que persona ha actuado en que película.

```
with "jdbc:postgresql:xine?user=alumnogreibd&password=greibd2021" as url
CALL apoc.load.jdbc(url,
"select per.id as persona, pr.personaje as personaje, pr.pelicula
from pelicula_reparto as pr, personas as per
where pr.persona = per.id and pr.pelicula in (select id from peliculas order by presupuesto desc, id limit 1000)") yield row
MATCH (pelicula:Pelicula {id:row.pelicula})
MATCH (persona:Persona {id:row.persona})
CREATE (persona)-[r:ACTUO_EN {personaje:row.personaje}]->(pelicula);
```

Finalmente procedemos a insertar las relaciones que definen que persona ha trabajado en que película.

```
with "jdbc:postgresql:xine?user=alumnogreibd&password=greibd2021" as url
CALL apoc.load.jdbc(url,
"select per.id as persona, pp.trabajo as trabajo, pp.pelicula
from pelicula_personal as pp, personas as per
where pp.persona = per.id and pp.pelicula in (select id from peliculas order by presupuesto desc, id limit 1000)") yield row
MATCH (pelicula:Pelicula {id:row.pelicula})
MATCH (persona:Persona {id:row.persona})
CREATE (persona)-[r:TRABAJO_EN {trabajo:row.trabajo}]->(pelicula);
```

Ahora ya podemos realizar consultas que exploten las relaciones entre personas y películas. Vamos a ver en primer lugar como usamos patrones de tamaño variable.

Obtener el número de personas que han colaborado de forma directa con "Penélope Cruz", es decir, que han actuado o trabajado en la misma película.

```
match (penelope:Persona{nombre: "Penélope Cruz"})-->()<--(p)
return count(distinct p.id);
```

Obtener el número de personas que han actuado con alguna persona con la que haya actuado "Penélope Cruz" (relaciones de segundo nivel de tipo ACTUO_EN)

```
match (penelope:Persona{nombre: "Penélope Cruz"})-[:ACTUO_EN*..4]-(p:Persona)
return count(distinct p.id);
```

Obtener el número de personas que ha colaborado con alguna persona con la que haya colaborado "Penélope Cruz" (relaciones de segundo nivel de cualquier tipo).

```
match (penelope:Persona{nombre: "Penélope Cruz"})-[*..4]-(p:Persona)
return count(distinct p.id);
```

Para la siguiente consulta necesitamos calcular el camino más corto en un par de nodos.

Obtener el camino más corto entre "Kevin Bacon" y "Penélope Cruz" a través de relaciones tipo ACTUO_EN. Obtén el camino más corto usando cualquier tipo de relación también.

```
MATCH (k:Persona {nombre: 'Kevin Bacon'} ),
      (a:Persona {nombre: 'Penélope Cruz'}),
      p = shortestPath((k)-[:ACTUO_EN*]-(a))
RETURN length(p), p;
```

```
MATCH (k:Persona {nombre: 'Kevin Bacon'} ),
      (a:Persona {nombre: 'Penélope Cruz'}),
      p = shortestPath((k)-[*]-(a))
RETURN length(p), p;
```

Ejercicios

En este ejercicio vamos a importar datos de la base de datos de películas y realizar algunas consultas sobre las mismas y sus créditos (personal y reparto). Antes de importar los nuevos datos, debemos asegurarnos de que la base de datos esté vacía. Para eso, borramos todos los nodos y relaciones.

```
match(n) with n limit 10000 detach delete n;
```

La consulta anterior elimina 10000 nodos, con sus relaciones respectivas con otros nodos. Repetimos esta sentencia hasta que no existan nodos en la base de datos. Borramos los nodos de 10000 en 10000 para evitar problemas de memoria de la máquina virtual de Java durante la operación.

Importación de los datos de a través de JDBC

1. Importa los datos de las 2000 películas que más ingresos han generado, de cada película almacena su identificador, título, presupuesto, fecha de emisión, ingresos y duración. Crea índices por id y título.
2. Importa los datos de las personas involucradas en las 2000 películas ya importadas. De cada persona importa su identificador y su nombre. Crea índices por id y nombre.
3. Importa ahora las relaciones que tienen que ver con el reparto de cada una de las películas antes importadas. Para cada relación, además de registrar la persona y película involucrada, almacena también el personaje interpretado y el orden de aparición en los créditos.
4. Importa las relaciones que tienen que ver con el resto de personal involucrado en la película. Para cada relación, además de registrar la persona y película involucrada, almacena también el trabajo ('job') y el departamento ('department').

Consultas

1. Crea una relación DIRIGE entre cada director y las películas que ha dirigido.
2. Obtén un listado del reparto de la película "Star Wars", ordenado por el atributo orden de la relación ACTUO_EN. Para cada miembro del reparto, muestra el nombre del personaje interpretado y el nombre del actor/actriz.
3. Obtén la lista de las 10 películas que mayor beneficio hayan generado (ingresos - presupuesto).
4. Mostrar las películas en las que participó "Quentin Tarantino", como actor o como director. Para cada película, muestra su título, fecha de emisión, presupuesto e ingresos. Ordena el resultado por fecha de emisión. Muestra una propiedad "participacion" que indique si dirigió o si actuó.
5. Para la pelicula "The Godfather", obtén para cada departamento, el número de personas involucradas y la lista de nombres y trabajos de cada persona.
6. Para la película con más ingresos de entre las que trabajó "Steven Spielberg", obtener el título, el trabajo realizado por el, el presupuesto, los ingresos y el número de personas que actuaron y el número de personas que trabajaron (sin contar los miembros del reparto).
7. Obtén los nombres de actores y actrices dirigidos por algún director que haya dirigido a "Marlon Brando". Ordena el resultado por el nombre.
8. Obtén una lista de películas que tengan más de un director. Para cada película obtén también una lista de los nombres de sus directores. Ordena el resultado por el número de directores.
9. Obtén las 10 personas que más roles han desempeñado en una película, incluyendo participaciones en reparto y personal. Para cada persona, mostrar su id, nombre, el título de la película, el número de roles que desempeño en la película y los nombres de los roles (usa el trabajo para la relación TRABAJO_EN y el texto 'reparto' para la relación ACTUO_EN). Si te sirve de ayuda, puedes usar la cláusula CALL para realizar una subconsulta.
10. Obtén la lista de actores o actrices que, o han trabajado con "Quentin Tarantino", o han podido oír de primera mano como es actuar bajo su mando (han actuado en alguna película en la que haya actuado también alguien que ha sido dirigido por el).

Última modificación: luns, 13 de setembro de 2021, 15:54