# Índice de contenidos




***Adaptado por: Alonso Núñez, Mario***

# 1 Introducción

## 1.1 Ejemplo: búsqueda de anillos

Tenemos una tabla en una base de datos SQL que indica qué ciertas empresas venden productos a otras:

| Facturador | Facturado |
|---|---|
| Girasoles Solete | Aceites Olé |
| Aceites Olé | Maquinaria RunRun |
| Girasoles Solete | Semillas Sidi |
| Girasoles Solete | Piensos Rúmiame |
| Envasadora Paketo | Semillas Sidi |
| Piensos Rúmiame | Vaquería Mu |
| Aceites Olé | Envasadora Paketo |
| Semillas Sidi | Invernaderos Invernalia |
| Invernaderos Invernalía | Aceites Olé |
| Vaquería Mu | Girasoles Solete |

En esta tabla la columna *facturador* indica la empresa que vende o factura un producto, que *facturado* es la empresa que paga por el servicio. Esta información es fácil de representar, por ejemplo como una tabla SQL. Sin embargo, supongamos que estamos interesados en *anillos de ventas*: buscamos secuencias de empresas A, B, C...A tal que A factura a B, B factura a C, y así hasta llegar a una empresa que factura a A y cierra el anillo.

Aquí vemos un anillo de este tipo en nuestra tabla:

| Facturador | Facturado |
|---|---|
| Girasoles Solete | Aceites Olé |
| Aceites Olé | Maquinaria RunRun |
| Girasoles Solete | Semillas Sidi |
| Girasoles Solete | Piensos Rúmiame |
| Envasadora Paketo | Semillas Sidi |
| Piensos Rúmiame | Vaquería Mu |
| Aceites Olé | Envasadora Paketo |
| Semillas Sidi | Invernaderos Invernalia |
| Invernaderos Invernalía | Aceites Olé |
| Vaquería Mu | Girasoles Solete |

Tenemos que *Envasadora Paketo* vende a *Semillas Sidi*, que vende a *Invernaderos Invernalia*, que a su vez vende a *Aceites Olé*, que finalmente vende al primer elemento del anillo, *Envasadora Paketo*. Nótese que aunque hemos usado un código de colores para cada empresa aun así el anillo no resulta nada evidente en el formato de tabla. Posteriormente veremos que en una base de datos orientada a grafos este tipo de estructuras se presentan de una manera mucho más visual.

El interés de estos anillos puede ser, por ejemplo, que vendamos una aplicación de facturación y que si todas las empresas lo adquieren el intercambio de datos sea más sencillo. Si es el caso y si casi todas ya tienen nuestra aplicación, quizás merezca hacer un precio especial a las que faltan para tener el anillo completo. Otra razón para buscar anillos de este tipo puede ser lograr fidelización de los clientes; aunque alguno reciba una oferta de otra aplicación desconfiará porque sus conocidos tienen todos la misma y temerá que haya problemas de compatibilidad entre ellas.

En otras ocasiones, la detección de anillos tiene interés para detectar fraudes complejos, integrados por diversos agentes coordinados. El sector de la banca y las aseguradoras dedica mucho esfuerzo a la detección de estas estructuras organizadas, para las que resultan especialmente útiles las bases de datos orientadas a grafos.

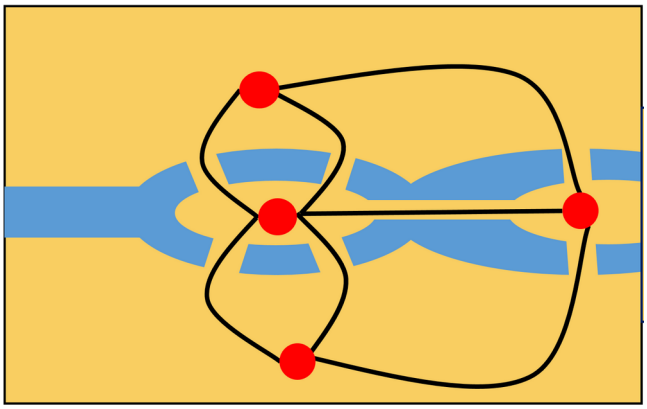
Pero, ¿no se pueden obtener estos anillos mediante sencillas consultas en SQL? La respuesta es que no, al menos de forma eficiente. Si en la tabla anterior quisiéramos buscar anillos de longitud 3, y llamamos t a la tabla, podríamos escribir:

```
SELECT t1.facturador, t1.facturado, t2.facturado, t3.facturado
FROM t as t1, t as t2, t as t3
WHERE t1.facturado=t2.facturador AND
      t2.facturado=t3.facturador AND
      t3.facturado=t1.facturador;
```

Aparte de que el código es realmente confuso, tiene un problema: solo vale para anillos de longitud 3. Si queremos buscar anillos de longitud mayor tenemos que ir generando nuevas consultas, más y más complejas. SQL tiene una solución para este tipo de situaciones, las consultas recursivas. Sin embargo estas consultas son muy ineficientes en presencia de gran cantidad de valores. Es este uno de los casos de aplicación de las bases de datos orientadas a grafos.

## 1.2.- ¿Qué es una base de datos orientada a grafo?

Las bases de datos orientadas a grafo heredan su estructura y terminología de la teoría de grafos creada por Leonard Euler alrededor de 1736 mientras intentaba resolver el problema conocido como los siete puentes de Könisberg. El problema consistía en ver si era posible dar un paseo por los siete puentes de la ciudad, pasando una única vez por cada uno de ellos. Para representar el problema, Euler sugirió una representación en la que las porciones de ciudad conectadas por los puentes eran puntos, y los puentes líneas que unían esos puntos:



En la imagen se ve en azul el río que atraviesa la ciudad y que hace dos islas centrales. La primera isla tiene 4 puentes, dos hacía cada lado, y la segunda isla dos puentes. Además, hay un séptimo puente que une ambas islas. Sobre la imagen se ve el grafo correspondiente que sugirió Euler: cada isla y cada orilla se convierte en un punto (punto rojo en la imagen), también llamado nodo o vértice en terminología de grafos, mientras que cada puente se representa por una línea llamada arista o relación.

Euler demostró que atravesar todas las aristas una sola vez (lo que hoy en día se llama un camino euleriano) solo era posible si el grafo era:

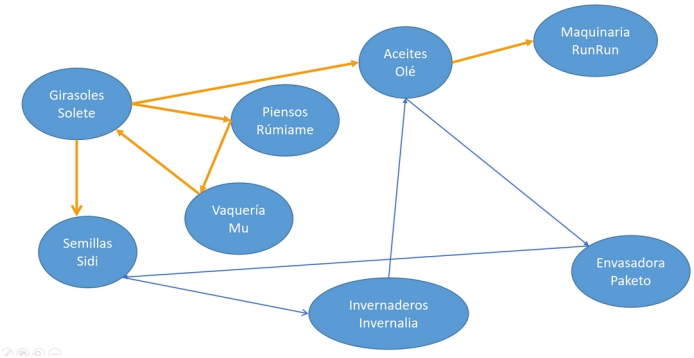
1. Conexo, lo que quiere decir que desde cualquier vértice hasta cualquier otro.
2. Con 0 o dos vértices de grado impar. El grado de un vértice es el número de aristas que inciden en él.

En el caso del grafo de los puentes de Könisberg se cumplía la primera condición pero no la segunda, porque hay 4 vértices de grado impar (todos los del grafo), por lo que el paseo buscado sin repetir puentes y cruzándolos todos una sola vez no era posible.

Pronto se vio que esa representación era útil en muchos otros casos. Hoy en día se utiliza para resolver problemas relacionados con la posición de componentes en circuitos, diseño de transportes (el mapa de metro es un buen ejemplo de grafo), análisis de redes sociales (los puntos son usuarios de la red y las línea gente que le sigue o a la que siguen), control de epidemias, etc.

Hay que notar que a menudo nos interesa que la línea tenga una dirección, por ejemplo que el usuario A siga al usuario B, no implica que el usuario B tenga que seguir al A. También sería el caso en el que queremos hacer el recorrido de los puentes en coche pero no todos los puentes son de dos direcciones. Para indica la direccionalidad se emplean aristas dirigidas que se representan como flechas. La mayor parte de las bases de datos orientadas a grafos utilizan aristas dirigidas. Esto no supone ninguna limitación, aunque sí puede ser una molestia; si queremos "simular" aristas no dirigidas, lo que debemos hacer es introducir dos aristas dirigidas cada vez que quedamos poner una no dirigida: una de "ida" y otra de "vuelta".

Por ejemplo nuestro grafo de facturación entre empresas es dirigido:




Hemos representado con líneas azules un anillo de facturación, que es en lo que estamos interesados.

Hay que mencionar que, dentro de las bases de datos NoSQL, las bases de datos orientadas a grafo ocupan un lugar especial, ya que se utilizan para propósitos muy específicos, rara vez como bases de datos de uso general. Es habitual que ya tengamos nuestros datos almacenados en una base de datos SQL, o quizás en una NoSQL orientada a documento como MongoDB, y veamos que cierto análisis que nos interesa se podría llevar a cabo más fácilmente en una base de datos orientada a grafos. En este caso exportaríamos nuestros datos par nutrir una base de datos de este tipo y allí poder realizar los análisis. A esta coexistencia de distintas tecnologías de almacenamiento en el mismo proyecto se le llama persistencia políglota y es habitual en todo proyecto con una mínima complejidad.

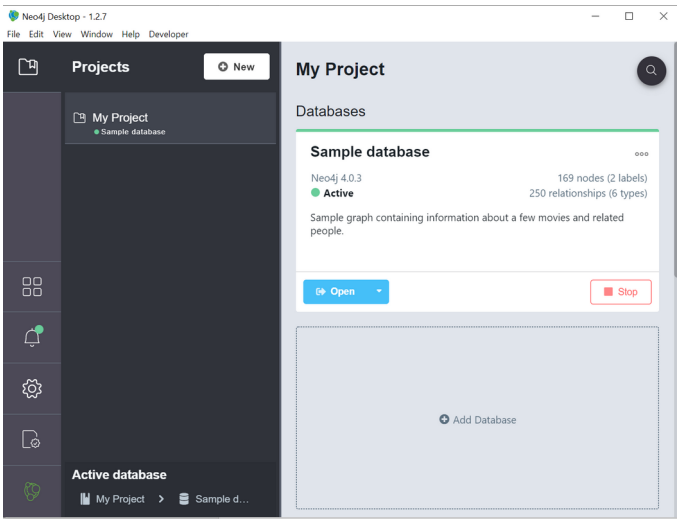
## 1.3.- Las bases de datos orientadas a grafos más comunes y sus características

Sin duda las base de datos orientada a grafos más famosa es Neo4j, y es la que vamos a ver en este capítulo. Neo4j es capaz de trabajar en entornos Big Data y es especialmente útil cuando se trata de encontrar "caminos" entre nodos concretos. Su lenguaje de consultas Cypher, que vamos a ver aquí, es muy expresivo y potente. Una alternativa con características similares pero menos extendida es JanusGraph, una iniciativa de la fundación Apache que utiliza como lenguaje de consultas el lenguaje Gremlin. Las bases de datos multiparadigma, como Microsoft Azure Cosmos DB y

OrientDB también suelen disponer de una parte orientada a grafos, al igual que hace MongoDB o Spark. Sin embargo, hay que señalar que en general estas bases de datos no tan específicas de grafos suelen ser menos eficientes.

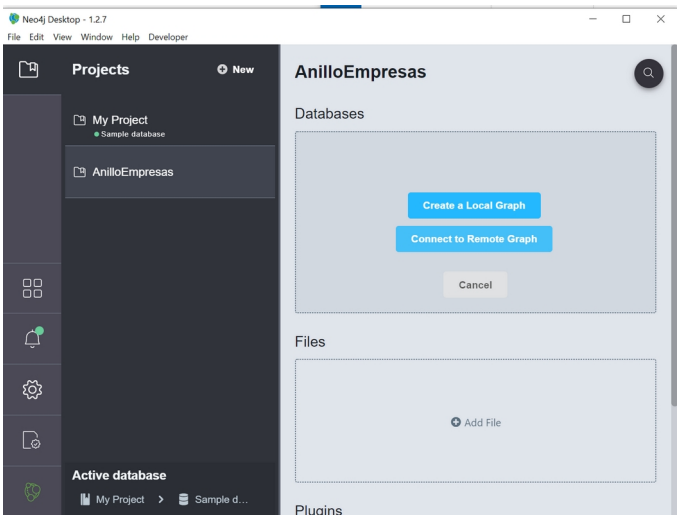
## 1.4.- Instalación arranque de Neo4j

La instalación es realmente sencilla. De hecho, si solo pretendemos hacer unas pocas pruebas, podemos emplear la Neo4j Sandbox, un acceso en la nube sencillo, que permite crear proyectos con duración limitada. Si en lugar de esto preferimos tener nuestra propia versión de Neo4j instalada en local, basta con ir a https://neo4j.com/ y buscar la sección "Downloads". Allí descargaremos el Neo4j Desktop. Tras instalarlo y abrirlo procedemos a crear un nuevo proyecto haciendo click en el botón New de la parte izquierda:




Por defecto el nombre del proyecto es Project pero si situamos el ratón sobre el nombre y pulsamos en el lápiz a su derecha podemos cambiarlo, por ejemplo a AnilloEmpresas. Cada proyecto puede tener dentro varias bases de datos. En Neo4j el término "base de datos" se denomina, simplemente grafo. De toda formas solo podemos tener un grafo activo cada vez.

Podemos crear un grafo nuevo dentro del proyecto AnilloEmpresas simplemente pulsando en la parte derecha Add a database. Allí nos preguntará si queremos crear la base de datos en local o enlazar con una base de datos externa, por ejemplo con una alojada en el servicio cloud de Neo4j, Aura:



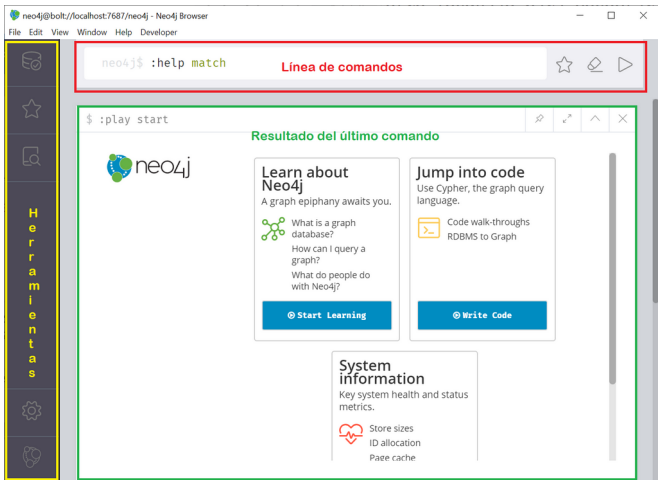
Nosotros elegimos Create a Local Graph. Nos pedirá un nombre y un password, que debemos recordar. Llamemos a esta base de datos facturación. A continuación pulsamos start para iniciar el grafo. Ya tenemos nuestro primer grafo creado, de momento con cero vértices y cero aristas. Cuando terminemos debemos recordar que habrá que dar stop para parar la base de datos con seguridad.

En Neo4j, el Desktop en local hace el papel de servidor. Como ya lo hemos iniciado, lo siguiente sería abrir un cliente desde el que podamos hacer operaciones como insertar, hacer consultas, etc. Para ello pulsaremos en Open, que abrirá el cliente en el que haremos el resto del trabajo. Y a partir de ahora, cuando queramos utilizar Neo4j el proceso será siempre el mismo:

1. Abrir el Neo4j-Dekstop.
2. Seleccionar (o crear) el proyecto que deseemos, y dentro del proyecto iniciar (o crear e iniciar) el grafo que deseemos
3. Pulsar Open para abrir el browser de columnas, o ir a Python o cualquier otro lenguaje para conectar con el servidor.

# 2.- El cliente Neo4j Browser

Tras seleccionar Project, iniciar un grafo y pulsar Open en el Desktop veremos el cliente por defecto, el Neo4j-Browser. Si lo preferimos también podemos usar como cliente cualquier navegador como Google Chrome. Para ello nos bastará con introducir en la barra de direcciones http://localhost:7474 (los puertos por defecto se pueden ver y cambiar en el Desktop pulsando en los tres puntos de arriba a la derecha del grafo que deseemos y seleccionando Manage).




Dentro del browser tenemos distintas zonas:

- La línea de comandos permite introducir consultas Cypher y comandos. Los comandos empiezan siempre precedidos por dos puntos, mientra que las consultas Cypher no. Podemos movernos con flecha arriba abajo para seleccionar las consultas anteriores.
- Las cajas de resultados nos mostrarán la salida de la consulta/comando. Se van añadiendo hacia abajo, así podremos ver varios resultados anteriores.
- A la derecha temos la barra de herramientas que nos permitirán hacer varios ajustes

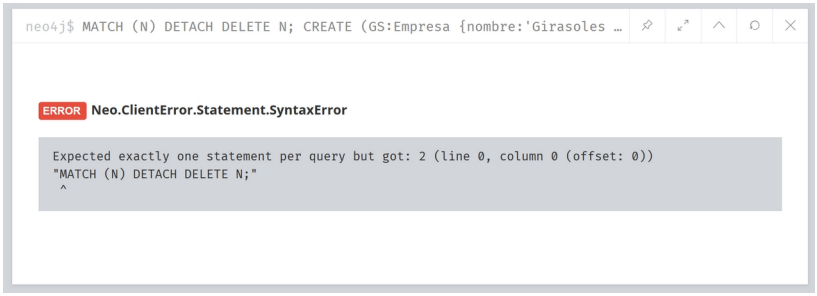
Algunos comandos que nos pueden ser de utilidad:

- :clearBorra la lista de resultados
- :help. Muy útil para acceder al manual. Por ejemplo, probad :help MATCH
- :history

Vamos a hacer un par de ajustes de configuración útiles para el resto de este capítulo. Para ellos hacemos click en la rueda dentada de la derecha, en la barra de herramientas. Allí marcaremos las opciones Enable multi statement query editor y desmarcaremos Connect result nodes.

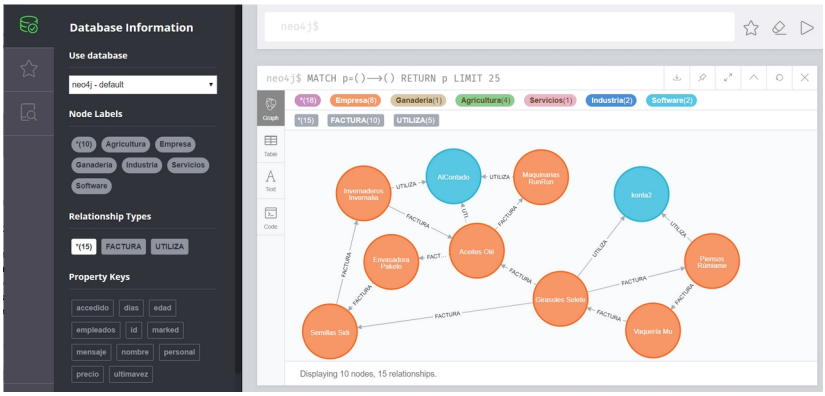
La primera opción, marcar Enable multi statement query editor, nos permitirá copiar y pegar scripts conteniendo más de una instrucción, separadas por punto y coma, simultáneamente en la línea de comandos. La segunda, desmarcar Connect result nodes hará que al mostrar la salida de una consulta solo se muestren las aristas que corresponden a la consulta y no todas las del grafo que haya entre los nodos seleccionados.

Para ver el aspecto de un grafo dentro del browser vamos a copiar, aunque de momento no las entendamos, las instrucciones del fichero factura.txt dentro del browser. Son dos instrucciones una primera para borrar el grafo antiguo y o/tra para crear el nuevo. Simplemente las copiamos y pegamos en la línea de comandos, y el triángulo de la derecha (play). Si obtuviéramos un error de este estilo:



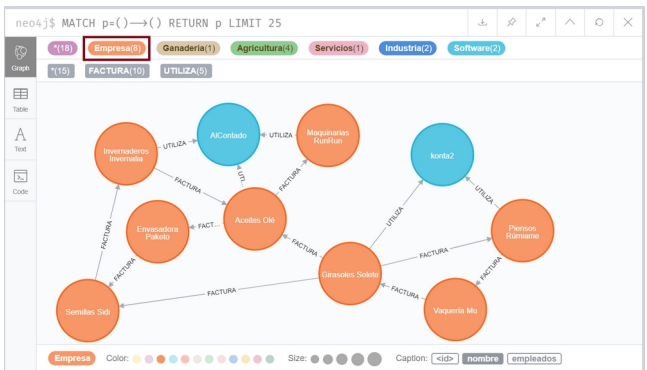

significaría que nuestra configuración en Neo4j no soporta más de una instrucción, lo que no debe pasar si hemos activado Enable multi statement query editor.

Una vez procesadas estas instrucciones podemos pulsar Database en la barra de herramientas (la primera opción, con el dibujo de un disco). Allí nos mostrará los tipos de nodos y aristas de nuestro grafo. Nos dice que hace 10 nodos y 15 relaciones o aristas. Los nodos son de dos tipos: Empresa que representa a las empresas que o bien facturan o bien son facturadas, y Software que va a representar diferentes software de contabilidad o facturación. Igualmente, tenemos dos tipos de relaciones: FACTURA que corresponde a la facturación de una empresa a otra tal y como mostrábamos en la tabla del principio del capítulo, y UTILIZA que indica que una empresa utiliza un software determinado. Si ahora pulsamos sobre la casilla con leyenda *(15) debajo de Relationship keys, obtenemos una representación del grafo completo:




Los colores indican los dos tipos de nodos. Y cada relación incluye el nombre del tipo de relación. Como veremos, tanto nodos como relaciones tienen propiedades, datos que contienen. En el caso de los nodos de tipo Empresa se está mostrando el nombre de la empresa, pero por desgracia no se ve completo. Para arreglar esto hacemos click sobre el nombre Empresa en la parte superior del propio grafo. En la linea de abajo nos aparecerán las características que definen la apariencia del nodo. Allí podemos hacer click sobre el círculo de mayor tamaño en Size para aumentar el radio del nodo y que se vea el nombre completo:

De esta forma podemos cambiar el color y elegir qué propiedad concreta se muestra (característica Caption). En este caso el nodo de tipo Empresa tiene dos: el nombre de la empresa, que es el seleccionado, y el número de empleados. Como ejercicio podemos probar ahora a hacer algo similar con el tipo de nodo Software. En la imagen se muestra la propiedad precio, pero podemos querer seleccionar el id, que tiene el nombre del software y por tanto parece mejor visualmente.



# 3.- Creación de nodos y relaciones en Cypher

En el apartado anterior copiamos de forma "ciega" instrucciones del lenguaje de consultas Cypher para crear el grafo; ahora vamos a empezar a entenderlas mejor.

## 3.1.- Representación de nodos y relaciones

La estructura de un nodo en Cypher es:


La única parte obligatoria son los paréntesis, así () representa un nodo cualquiera (nótese que recuerda a la forma de un círculo). La variable es un identificador que usaremos en una expresión Cypher para referirnos a ese nodo en concreto (esto se entenderá mejor al presentar la creación de nodos y relaciones). La etiquetas indican el tipo del nodo. Por ejemplo (:Empresa) representa cualquier nodo de tipo Empresa. Un nodo puede tener varios tipos simultáneamente, separados por :. De esta forma si en un catálogo de unos grandes almacenes donde los objetos se representan por nodos con tipo su sección queremos encontrar productos que están en la sección cocina y también en la de electrodomésticos, podríamos escribir (:Cocina:Electrodomesticos). Opcionalmente un nodo puede incluir entre llaves propiedades como en el siguiente ejemplo:

```
(EP:Empresa {nombre:'Envasadora Paketo', empleados:20})
```

Una relación se define mediante dos nodos una flecha que los une. La relación más genérica sería ()-->(). La forma general es:


La sintaxis de los nodos inicial y final es la ya descrita para nodos. Además, la fecha que los une puede llevar en la mitad, entre corchetes, un nombre de variable, etiquetas de tipos y propiedades. Un ejemplo, representando una relación entre un nodo de tipo Empresa y otro de tipo Software:

```
(:Empresa)-[:UTILIZA]->(:Software)
```

Atención a las mayúsculas: Neo4j distingue mayúscula y minúsculas. Para evitar errores difíciles de detectar es mejor seguir algún convenio, como escribir las propiedades siempre en minúsculas, las etiquetas (tipos) de nodos con la primera en mayúsculas, y las variables y tipos de relaciones totalmente en mayúsculas. Solo da igual al escribirlas palabras reservadas.

## 3.2.- CREATE

Ahora la sintaxis de la instrucción create debe ser sencilla de entender:


es decir, se trata de la palabra reservada CREATE seguida de una secuencia de nodos a crear y de relaciones que implican a estos nodos o a otros ya existentes en el grafo. Como ejemplo veamos un fragmento del código que hemos copiado y pegado en la sección anterior:

```
CREATE (GS:Empresa {nombre:'Girasoles Solete', empleados:15}),
       (AO:Empresa {nombre:'Aceites Olé', empleados:120}),
       (GS)-[:FACTURA {dias:90}]→(AO);
```

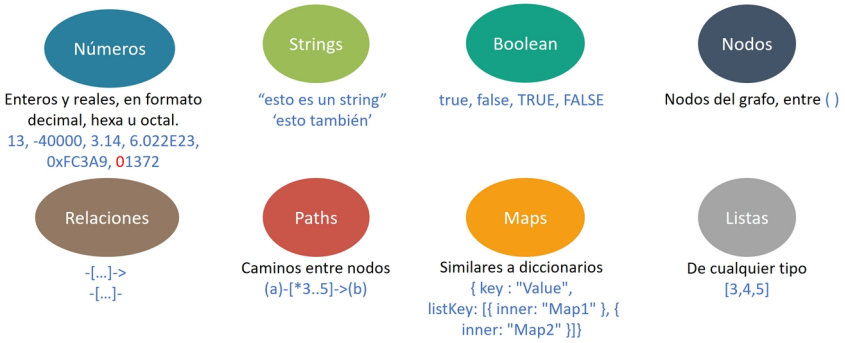
Primero se crean dos nodos, correspondientes a dos empresas, y luego la relación de tipo FACTURA que expresa que la primera empresa factura a la segunda en 90 días. Ahora se debe entender mejor el papel de las variables; al crear la relación se utilizan los nombres de las variables en representanción de cada nodo. Otra forma de hacerlo hubiera sido:

```
CREATE (:Empresa {nombre:'Girasoles Solete', empleados:15})
           -[:FACTURA {dias:90}]->
       (AO:Empresa {nombre:'Aceites Olé', empleados:120});
```

donde en lugar de los nombres de las variables se escriben directamente los nodos y la relación.

## 3.3.- Tipos en Cypher

Ya hemos visto la mayor parte de los tipos de valores en Cypher: nodos, relaciones, strings, maps para las propiedades... la siguiente figura muestra todos los tipos posibles:




Los tipos básicos son números, en distintas base, cadenas de caracteres o strings y valores lógicos. Además, tenemos los ya mencionados nodos y relaciones. Finalmente tenemos los caminos, muy útiles en las consultas y que veremos en detalle posteriormente, y los dos únicos tipos compuestos: los diccionarios o maps y las listas de valores. En el resto del capítulo iremos viendo más en profundidad el papel que juegan los caminos y el diseño de consultas complejas. De momento, baste decir que los valores de las propiedades de nodos y relaciones pueden emplear lists y maps anidados para representar información estructurada.

## 3.4.- MERGE

CREATE añade todos los elementos especificados al grafo. Aunque en general esto será lo deseado, en ocasiones puede generar duplicidades. Por ejemplo, podemos querer añadir una empresa, con nombre 'Invernaderos Invernalia', pero si hacemos

```
CREATE (:Empresa {nombre:'Invernaderos Invernalia'});
```

y la empresa ya existe, habremos producido una duplicidad.

Esto se podría arreglar añadiendo una restricción de unicidad sobre la propiedad nombre, pero entonces el create completo fallaría, y en casos de creaciones de muchos nodos simultáneos tendríamos que tampoco se han añadido los que sí eran nuevos. También después veremos cómo con una sentencia MATCH podemos verificar previamente si los nodos ya existen, pero lugar de hacerlo tan complicado, podemos confiar en MERGE que hace lo mismo que CREATE pero evitando crear elementos que ya existan.

```
MERGE (:Empresa {nombre:'Invernaderos Invernalia'});
```

La operación MERGE recuerda a los upsert de otras bases de datos como MongoDB. Como sucede con las operaciones upsert podemos especificar qué sucede tanto si el valor es nuevo como si ya existe:

```
MERGE (II:Empresa {nombre:'Invernaderos Invernalia'})
ON CREATE SET II.creado = timestamp()
ON MATCH SET II.accedido = date();
```

si la empresa ya existía se modifica su propiedad accedido (o se crea si no existe la propiedd), y si es nuevo se añade la propiedad creado

## 3.5.- Importación masiva de datos

Por supuesto si queremos importar o exportar grandes cantidades de datos, utilizar CREATE no parece la mejor opción. Aunque queda fuera del ámbito de este curso entrar en detalle, debemos mencionar la existencia de dos formas de importar ficheros CSV:

1. Desde Cypher podemos utilizar una instrucción LOAD CSV para importar ficheros CSV sencillos
2. Para importaciones más complejas que incluya ficheros con nodos, otros con relaciones, etc podemos utilizar la herramienta externa neo4j-admin import.

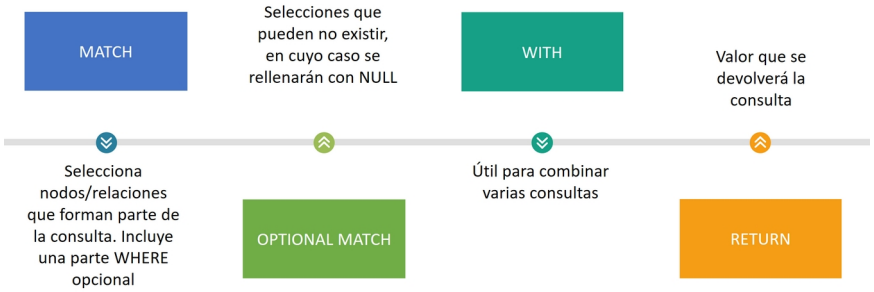
# 4.- Consultas básicas con Match

## 4.1 MATCH: forma general

Las instrucciones MATCH son la forma fundamental de realizar consultas en Cypher. Serían el equivalente al SELECT de SQL. En nuestro ejemplo lo queremos para buscar anillos de empresas, y esa va a ser una de las especialidades de MATCH: encontrar caminos en un grafo. La sintaxis general de esta instrucción, con las partes opcionales entre corchetes es:

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Un vistazo rápido a cada una de las secciones:




## 4.2 RETURN

Es la única sección obligatoria de MATCH e indica el valor final que devolverá la consulta y que nos mostrará el navegador. Sería el equivalente a la proyección en MongoDB. Puede incluir partes opcionales SKIP n y LIMIT n para saltar los primeros resultados o mostrar solo los primeros n resultados, respectivamente. Además admite ORDER BY para ordenar información, de forma ascendente (sufijo ASC, por defecto) o descendente (DESC). Algunos ejemplos

```
return 1 // muestra 1
return "hola" // muestra “hola”
return 1 as uno // renombra la columna
return 1, 2, 3, ”pepe” // secuencia (4 columnas)
return {nombre:’Bertoldo’, edad:31} // se pueden mostrar maps
return [1,2,3,’uff’] // y listas
return (:Ciudad) // error: no se pueden “inventar” nodos ni relaciones
```

## 4.3 MATCH

La parte MATCH indica qué nodos/relaciones del grafo debe seleccionar la consulta, sería la parte where de SQL o el primer argumento del find en MongoDB. Consta de dos partes:

1. En la primera parte se indica la forma de los elementos que deben estar.
2. La segunda se filtran por características de esos elementos, normalmente por el valor de sus propiedades.

### *4.3.1 Patrones de nodos y relaciones*

Para poder extraer toda la potencia de MATCH debemos conocer la sintaxis de patrones para nodos y relaciones. Para nodos la sintaxis es ya conocida:

| Patrón Cypher | Encaja con… |
|---|---|
| ( ) | Cualquier nodo |
| ( N) | Cualquier nodo, la variable N toma el valor de ese nodo |
| (:Empresa) | Nodos que tengan (al menos) la etiqueta Empresa |
| (N:Empresa {nombre: "IBM"}) | Nodos con la etiqueta Empresa y la propiedad nombre con valor "IBM" |
| (N:Empresa:Producto) | Nodos con (al menos) las etiquetas Empresa, Producto |

En cambio, para relaciones tenemos que considerar nuevos patrones que nos serán muy útiles:

| Patrón Cypher | Encaja con… |
|---|---|
| ( a)--(b) | a y b están unidos, no importa la dirección |
| (a)-[r]->(b) | r es la relación que une a y b |
| (a)-[r:REL_TYPE]->(b) | REL_TYPE es el tipo de relación que une a y b |
| (a)-[r:TYPE1\|TYPE2]->(b) | a y b están unidos por una relación que es o bien de tipo TYPE1 o tipo TYPE2 |
| (a)-[*2]->(b) | Camino de longitud 2 entre a y b |
| (a)-[*3..5]->(b) | Camino de longitud 3,4,5 entre a y b |
| (a)-[*..5]->(b) | Camino de longitud menor que 5 entre a y b |
| (a)-[*]->(b) | Camino de cualquier longitud entre a y b |

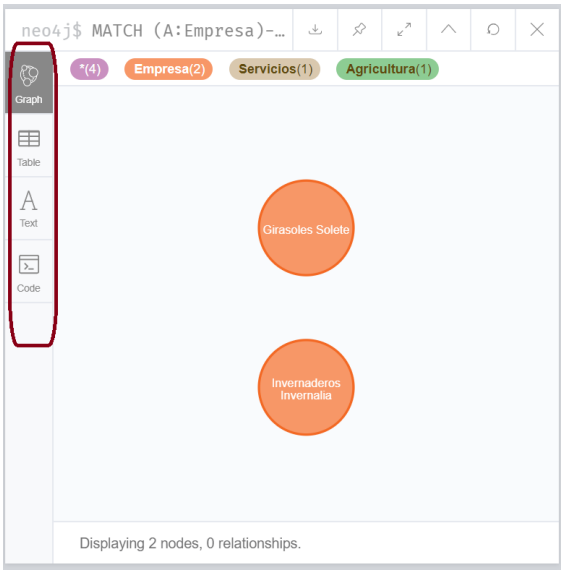
Los primeros cuatro patrones corresponden a caminos de longitud uno, y los siguientes a caminos de longitud mayor. Hay que ser cuidadoso al utilizar patrones de tipo * (caminos de cualquier longitud) en grafos grandes, o al menos añadir la opción LIMIT en el resultado.

### *4.3.2 Where*

Veamos un ejemplo utilizando la cláusula WHERE, empresas que facturan a otras con pago a 30 días:

```
MATCH (A:Empresa)-[R:FACTURA]-> (B:Empresa)
WHERE R.dias=30
RETURN A
```

La primera parte busca caminos (A)-[R]->(B) con A y B nodos de tipo Empresa y una relación de tipo FACTURA. Además, la parte WHERE se encarga de comprobar que el número de días para la facturación sea 30. Finalmente, la parte RETURN devuelve a la empresa que factura. El resultado:




En este caso hemos empleado el operador de igualdad, pero está bien sabe que tenemos un repertorio más amplio:

1. Aritméticos: +, -, *, / , %, ^
2. Relacionales: =, <>, <, >, ⇐, >=, IS NULL, IS NOT NULL
3. Cadenas: STARTS WITH, ENDS WITH, CONTAINS, + (concatenación) y =~ para expresiones regulares
4. Booleanos: AND, OR, XOR, NOT
5. Listas: IN (pertenecia), + (concatenación).

En particular el uso del operador NOT es muy útil para asegurar que una relación no está presente en el grafo. Por ejemplo si queremos un listado de empresas tal a las que "Semillas Sidi" no factura directa ni indirectamente, podemos escribir

```
MATCH (A:Empresa {nombre:"Semillas Sidi"}), (B:Empresa)
WHERE NOT (A)-[:FACTURA*]->(B)
RETURN B
```

que nos devolverá 3 empresas.

### *4.3.3 Formas de mostrar la salida*

Parece que solo dos empresas facturan a otras a 30 días. Puede que en este caso la información en forma de grafo no sea la más útil, sino que prefiramos ver los datos de estas empresas en formato texto. Para ello tenemos las opciones de la parte izquierda (en un rectángulo rojo en la imagen) que nos permiten seleccionar cómo se muestra la información de salida.

La opción Table nos permite ver los resultados en forma de documentos JSON, la opción Text muestra algo similar a las salidas textuales en shells de SQL. Finalmente, Code nos muestra una estructura JSON compleja que incluye tanto la consulta como las respuestas en forma de lista.

### *4.3.4 Mostrando el camino*

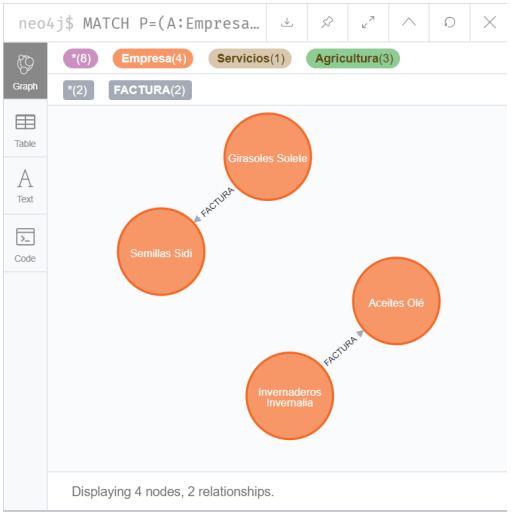
Supongamos ahora que no solo queremos mostrar las empresas que facturan a otra 30 días sino también las empresas facturadas. Una idea puede ser añadir estos nodos en el return:

```
MATCH (A:Empresa)-[R:FACTURA]-> (B:Empresa)
WHERE R.dias=30
RETURN A,B
```

Sin embargo, esto tiene el problema de que mostrará 4 nodos aislados, no sabremos cuál hace el papel de facturador ni el de facturado. Tampoco sabremos quién factura a quién. Para representar con más precisión esta consulta podemos devolver caminos en lugar de nodos:

```
MATCH P=(A:Empresa)-[R:FACTURA]-> (B:Empresa)
WHERE R.dias=30
RETURN P
```

Ahora, nombramos el camino que cumple la condición con la variable P, y devolvemos precisamente ese camino. El resultado, ahora sí, es el esperado:

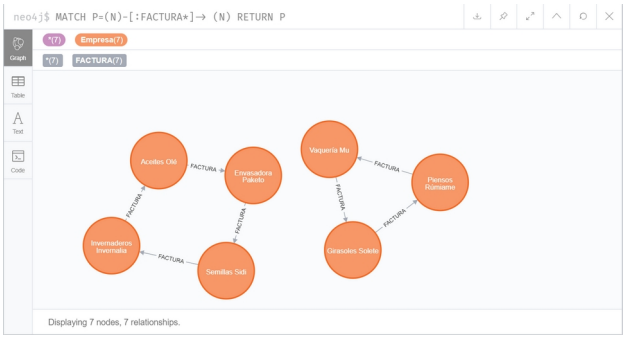



## 4.4 Anillos de empresas ¡por fin!

Ahora ya podemos resolver el problema de localizar anillos de empresas que se facturan unas a otras:

```
MATCH P=(N)-[:FACTURA*]-> (N) RETURN P;
```

La consulta es muy sencilla y revela toda la pontencia de Neo4j: Buscamos caminos de relaciones FACTURA desde un node N hasta si mismo. De esta forma tan sencilla obtenemos el resultado deseado, y además de una forma muy visual:

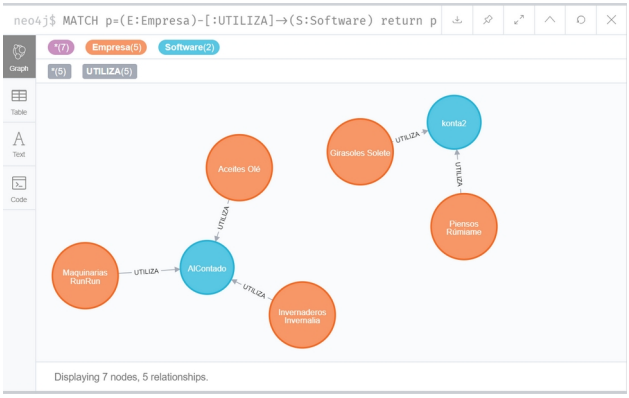



## 4.5 Optional Match

Esta opción permite añadir información a la salida de la consulta si existe, pero no elimina datos. Recuerda a los outer join de SQL. Supongamos que queremos saber qué software utiliza cada empresa. Una primera opción con lo que sabemos hasta ahora:

```
MATCH p=(E:Empresa)-[:UTILIZA]->(S:Software) return p
```

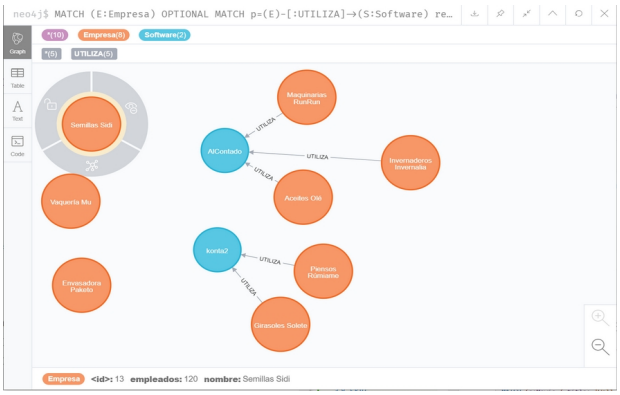
El resultado es:



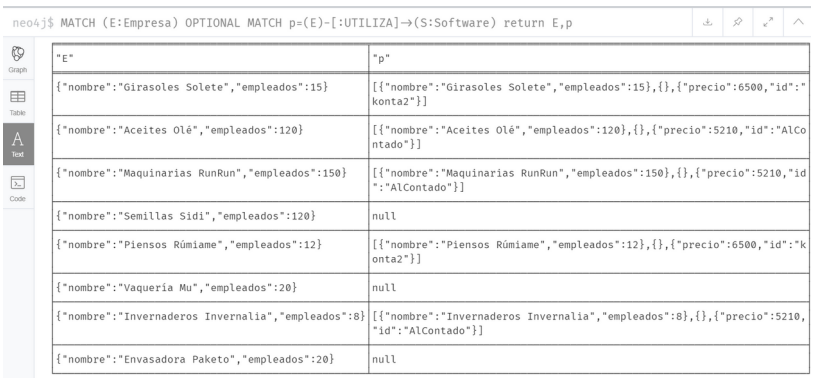

Sin embargo, supongamos que la respuesta incluya todas las empresas, también las que todavía no disponen de ningún software. En ese caso podemos utilizar OPTIONAL MATCH, seleccionando primero todas las empresas y luego añadiendo el software si existe:

```
MATCH (E:Empresa)
OPTIONAL MATCH p=(E)-[:UTILIZA]->(S:Software)
return E,p
```

El resultado ahora:




Si mostramos el resultado en forma de tabla vemos que, en efecto, esta consulta es análoga a un outer join de SQL en el sentido de que rellena con null la información que falta:


neo4j$ MATCH (E:Empresa) OPTIONAL MATCH p=(E)-[:UTILIZA]→(S:Software) return E,p

| "E" | "p" |
|---|---|
| {"nombre":"Girasoles Solete","empleados":15} | [{"nombre":"Girasoles Solete","empleados":15},{},{"precio":6500,"id":"konta2"}] |
| {"nombre":"Aceites Olé","empleados":120} | [{"nombre":"Aceites Olé","empleados":120},{},{"precio":5210,"id":"AlContado"}] |
| {"nombre":"Maquinarias RunRun","empleados":150} | [{"nombre":"Maquinarias RunRun","empleados":150},{},{"precio":5210,"id":"AlContado"}] |
| {"nombre":"Semillas Sidi","empleados":120} | null |
| {"nombre":"Piensos Rúmiame","empleados":12} | [{"nombre":"Piensos Rúmiame","empleados":12},{},{"precio":6500,"id":"konta2"}] |
| {"nombre":"Vaquería Mu","empleados":20} | null |
| {"nombre":"Invernaderos Invernalia","empleados":8} | [{"nombre":"Invernaderos Invernalia","empleados":8},{},{"precio":5210,"id":"AlContado"}] |
| {"nombre":"Envasadora Paketo","empleados":20} | null |

# 5.- Listas

Las listas son una estructura básica en Neo4j por las siguientes razones:

1. Son una estructura útil para representar arrays en las propiedades, al estilo de JSON.
2. Permiten "recolectar" resultados diversos de una consulta (collect).
3. Permiten generar resultados diferentes (unwind).
4. Los caminos pueden analizarse porque se convierten automáticamente en listas con nodos y relaciones alternos; esto permite añadir requerimientos sobre los caminos en las propias consultas.

## 5.1.- Operaciones

Comencemos viendo algunas operaciones sobre listas:

```
return size([1,2,3,4,5]) //5

return [1,2,3,4]+[5,6] //[1,2,3,4,5,6] (concatenación)

return [1,2,3][1] // 2 (el primer elemento es el 0)

return range(1,3)[4] //  null

return range(1,3)[-1] // 3 (primero por el final)

return range(0,20)[1..3] // 1,2 (sin incluir el último)
```

Como vemos los accesos son muy similares a los de los lenguajes como Python, permitiendo tanto accesos con índices desde el principio como desde el final, e incluso slices para devolver fragmentos de listas, con la característica particular de que los accesos fuera de rango dan null, mientras que los fragmentos fuera de rango se truncan. También se permite el uso de listas intensionales con un aspecto muy similar al de los lenguajes funcionales como Haskell o Erlang:

```
return [x in range(0,10) where x % 2 = 0] // [0,2,4,6,8,10]

return [x in range(0,10) where x % 2 = 0 | x+1 ] // [1, 3, 5, 7,
9, 11]
```

## 5.2.- Predicados y funciones para listas

Además, una serie de funciones hacen que las listas se puedan usar en las condiciones where de las instrucciones match:

1. ALL, ANY, NONE, SINGLE se cumplen si todos/alguno/ninguno/uno de los elementos cumplen una subcondición con listas y where. La sintaxis es pred(variable IN list WHERE predicate), con pred cualquiera de los predicados anteriores.
2. EXISTS(patron o propiedad) devuelve true si el patrón o propiedad existe en el grafo
3. Las relaciones/nodos de un camino c se obtienen con relationships(c)/nodes(c), respectivamente.
4. El tipo de una relación r se obtiene con type(r) . Las etiquetas de un nodo n con labels(n)

## 5.3.- Ejemplos

Veamos un ejemplo. Supongamos que estamos interesados en conocer las etiquetas de la empresa con nombre "Semillas Sidi

```
MATCH (E:Empresa {nombre:"Semillas Sidi"})

RETURN labels(E)
```

en este caso nos muestra la salida directamente en formato texto, ya que el return son etiquetas. La salida será ["Empresa", "Agricultura"].

Veamos una consulta más compleja: queremos conocer el nombre y número de empleados de todas las empresas que participen en caminos de facturación de cualquier longitud que partan de la empresa "Aceites Olé" y terminan en una empresa de tipo Industria

```
MATCH p=(A:Empresa {nombre:"Semillas Sidi"}) -[:FACTURA *]->(B:Empresa:Industria)

RETURN [x in nodes(p) where 'Empresa' in labels(x) | [x.nombre,x.empleados] ]
```

El resultado, de nuevo de tipo textual, son dos listas porque hay dos caminos de este tipo:

```
[["Semillas Sidi", 120], ["Invernaderos Invernalia", 8], ["Aceites Olé", 120], ["Envasadora Paketo", 20]]

[["Semillas Sidi", 120], ["Invernaderos Invernalia", 8], ["Aceites Olé", 120], ["Maquinarias RunRun", 150]]
```

Cada lista contiene a su vez listas, una por cada empresa que forma parte del camino, con dos elementos, el nombre de la empresa y el número de empleados.

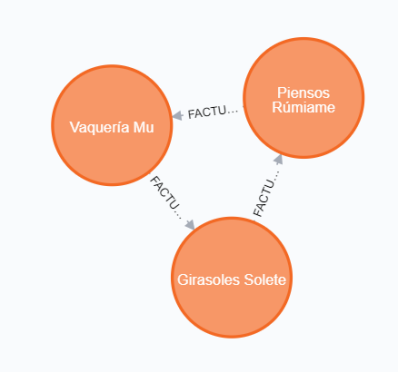
Finalmente, un ejemplo implicando alguno de los predicados vistos anteriormente. Supongamos ahora que nuestra empresa vende software a pequeñas empresas. Buscamos anillos de facturación en los que todas las empresas participantes tengan menos de 30 empleados. Esto lo podemos lograr con el predicado all:

```
MATCH p=(A) -[:FACTURA *]->(A)

WHERE all(x in nodes(p) where 'Empresa' in labels(x) and x.empleados<30)

RETURN p
```

El resultado esta vez si es un grafo, el único anillo de facturación con estas características



# 6.- Agregaciones

Las agregaciones son similares a otros lenguajes de consultas como SQL. En principio, la agregación considera como único grupo el grafo completo tendremos que utilizar consultas anidadas.

## 6.1- Agregaciones globales

Las operaciones de agregación son las usuales:

- COUNT: Número de elementos devuelto por match Ejemplo: match (n:Empresa) return count(n) .
- MAX: máximo de una propiedad entre el conjunto de nodos devueltos Ejemplo: match (n:Empresa) return max(n.empleados)
- MIN: mínimo de una propiedad entre el conjunto de nodos devueltos
- SUM: suma de una propiedad entre el conjunto de nodos devueltos
- AVG: media aritmética de una propiedad entre el conjunto de nodos devueltos
- COLLECT: recopila salidas en una lista
- Otras

## 6.2- Anidando consultas con WITH

Supongamos que queremos determinar el nombre de la empresa con etiqueta Industria con más empleados a la que factura, directa o indirectamente, 'Semillas Sidi'

```
MATCH (A:Empresa {nombre:"Semillas Sidi"}) -[:FACTURA *]->(B:Empresa:Industria)

RETURN max (B.empleados),B.nombre
```

Sin embargo esto no funciona como esperamos porque hace un grupo por cada valor de B. La solución es hacerlo en dos etapas:

1. Buscar el mayor número de empleados, pero en lugar de devolverlo con RETURN usaremos WITH para ponerle nombre que se usará en el segundo paso.
2. Buscar el nombre de la empresa con ese número de empleados

La consulta:

```
MATCH (A:Empresa {nombre:"Semillas Sidi"}) -[:FACTURA *]->(B:Empresa:Industria)

WITH  max (B.empleados) as m

MATCH (A:Empresa {nombre:"Semillas Sidi"}) -[:FACTURA *]->(B:Empresa:Industria {empleados:m})

RETURN B.nombre,m
```

devuelve, ahora sí "Maquinarias RunRun" 150

Sin embargo, si observamos en la consulta anterior se busca dos veces la empresa 'Semillas Sidi'. Para evitar esta redundancia innecesaria, podemos valernos de nuevo de WITH para "almacenar" el valor y lograr una consulta más eficiente:

```
MATCH (A:Empresa {nombre:"Semillas Sidi"}) -[:FACTURA *]->(B:Empresa:Industria)

WITH max (B.empleados) as m, A as Sidi

MATCH (Sidi) -[:FACTURA *]->(B:Empresa:Industria {empleados:m})

RETURN B.nombre,m
```

mejorando así la eficiencia

# 7.- Modificación y Eliminación

## 7.1 Modificación con SET

SET permite:

- Añadir una propiedad a un nodo. Ejemplo:
  - match (n:Empresa {nombre:"Semillas Sidi"}) set n.personal = ["Bertoldo", "Herminia"]
    - return n
- Borrar una propiedad Ejemplo:
  - match (n:Empresa {nombre:"Semillas Sidi"}) set n.personal = NULL
    - return n
- Copiar todas las propiedades de un nodo a otro Ejemplo:
  - match (a:Empresa {nombre:"Semillas Sidi"}) , (b:Empresa {nombre:”A4”}) set b=a
    - return a,b (no hará si alguna de las empresas no existe)
- Añadir varias propiedades Ejemplo:
  - match (n:Empresa {nombre:"Semillas Sidi"}) set n += {país:"España", continente:"Europa"}
    - return n
- Añadir una etiqueta Ejemplo:
  - match (a:Empresa {nombre:"Semillas Sidi"}) set n:Cooperativa
    - return n
- Borrar una etiqueta Ejemplo:
  - match (a:Empresa {nombre:"Semillas Sidi"}) remove n:Cooperativa
    - return n

## 7.2 Cambiando datos con foreach

foreach permite recorrer una lista aplicando una acción a cada uno de sus nodos. Por ejemplo, supongamos que queremos añadir una propiedad marked con valor TRUE a todos los nodos que facturan directa o indirectamente (uno o más pasos) a "Semillas Sidi". Podemos escribir:

```
MATCH p=(E:Empresa) -[:FACTURA *]-> (S:Empresa {nombre:"Vaquería Mu"})

FOREACH (n in nodes(p) | set n.marked=TRUE)
```

Si ahora queremos buscar esos nodos podemos utilizar la función exists como condición:

```
MATCH (E:Empresa)

WHERE exists(E.marked)

RETURN E
```

## 7.3 Borrado de nodos

Para borrar simplemente seleccionamos con MATCH y borramos con DELETE. Por ejemplo, si queremos borrar la facturación de 'Piensos Rúmiame' a 'Vaquería Mu' podemos escribir:

```
MATCH ({nombre:'Piensos Rúmiame'})-[R:FACTURA]->({nombre:'Vaquería Mu'})

DELETE R
```

Ojo al borrar un nodo que tiene relaciones; no nos lo permitirá porque tendríamos una arista sin nodo al final o al comienzo. Si queremos que al borrar el nodo se borren también todas las relaciones que empiezan o acaban en él debemos añadir la palabra reservada DETACH. Ahora entenderemos que la forma de borrar un grafo completo sea:

```
MATCH (N)

DETACH DELETE N
```

# 8.- Otras instrucciones

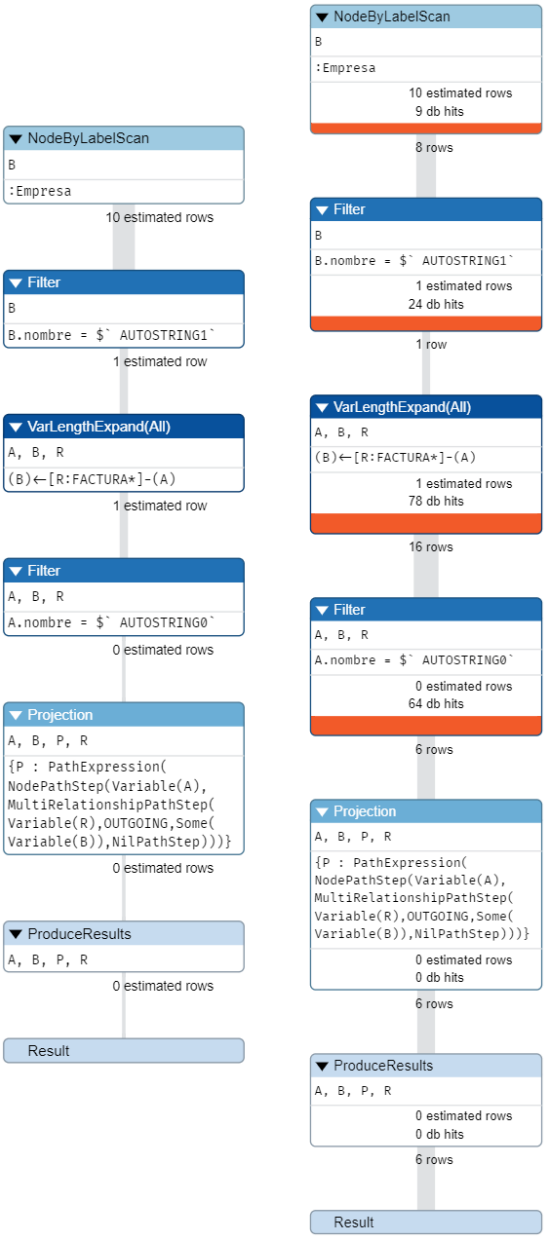
## 8.1 Índices y planes de ejecución

Si queremos mejorar la eficiencia de una consulta podemos precederla por la palabra EXPLAIN que mostrará gráficamente los pasos seguidos para resolver la consulta. También podemos usar PROFILE que muestra más detalles, como el número de accesos. Por ejemplo consideremos la consulta

```
EXPLAIN MATCH P=(A:Empresa {nombre:"Girasoles Solete"})-
[R:FACTURA*]-> (B:Empresa {nombre:"Semillas Sidi"})

RETURN P
```

La siguiente gráfica muestra el resultado de EXPLAIN (izquierda) y PROFILE (derecha):



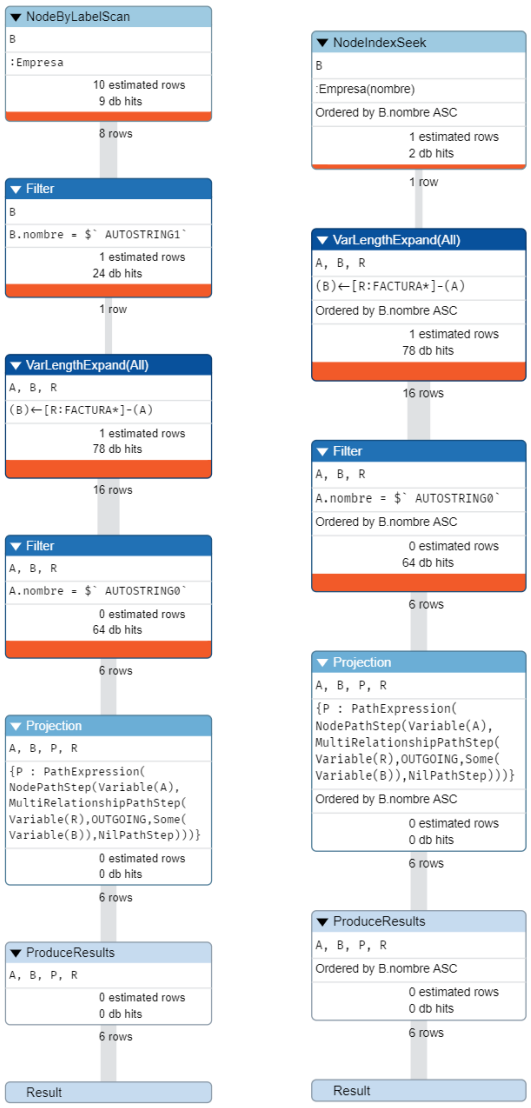

Los pasos son los mismos en ambos casos:

1. Primero se seleccionan los nodos de tipo Empresa que se almacenan en una variable B, la variable final.
2. A continuación se aplica el filtro para que B.nombre tome el valor adecuado
3. Luego se buscan caminos entre nodos cualesquiera A y el nodo destino ya fijado, B
4. Se obliga a que A.nombre tome el nombre adecuado
5. Se proyectan los resultados
6. Se muestran

El número de filas estimadas es una estimación estadística a partir de consultas similares y Neo4j lo emplea para seleccionar el plan de ejecución. Si queremos mejorar el número de accesos reales (el número de "hits" o accesos a la base de datos), podemos ayudarnos por un índice. Por ejemplo, en nuestra consulta podríamos crear un índice sobre el atributo nombre para los nodos de tipo Empresa:

```
CREATE INDEX ON :Empresa(nombre)
```

los índices se pueden crear sobre varias propiedades y también se pueden crear sobre relaciones. Si repetimos el PROFILE veremos que el plan de ejecución ha cambiado. La siguiente imagen muestra el paln anterior (izquierda( y el nuevo (derecha)



La diferencia principal es que los dos primeros pasos del antiguo plan de tipo NodeByLabelScan y Filter se han convertido en un solo paso NodeIndexSeek, que corresponde a una búsqueda utilizando índices. El número de hits de los dos pasos originales era 9+24=33 y en el nuevo quedan reducidos a solo 2. En una base de datos realmente grande la diferencia puede ser muy notable en términos de tiempo. Cabe preguntarse por qué el indice no se utiliza para buscar el segundo nombre, el de A. La razón es que los índices solo se pueden aplicar cuando se trata de buscar en el grafo completo´, pero no cuando se busca en el resultado de un paso anterior, que es lo que sucede con A.

Debemos recordar que los índices aceleran las consultas a costa de retrasar la inserción y modificación de datos, que exigen modificar también todos los índices. Por ello, antes de una ingesta masiva de datos es habitual eliminar todos los índices. Para borrar un índice podemos utilizar DROP INDEX ON:

```
DROP INDEX ON :Empresa(nombre)
```

Si en cualquier momento queremos ver la lista de indices debemos teclear:

```
CALL db.indexes
```

## 8.2 Restricciones

Un aspecto relacionado con la creación de índices es la imposición de restricciones. Neo4j incluye actualmente 4 tipos de restricciones:

1. Restricciones de unicidad, que evitan que un valor se pueda repetir (análoga a las claves primarias en bases de datos relacionales. En nuestro caso tendría sentido pedir que no se repita el nombre de la empresa:

   ```
   CREATE CONSTRAINT NombreUnico
   ON (n:Empresa)
   ASSERT n.nombre IS UNIQUE
   ```

   Para asegurar unicidad, Neo4j crea un índice sobre la propiedad, es decir la propiedad siempre debe existir y no repetirse. Tiene el efecto de que además ese índice se utilizará en las consultas. Es decir, si borramos el índice creado en la sección anterior, añadimos esta restricción de unicidad y probamos:

   ```
   PROFILE MATCH P=(A:Empresa {nombre:"Girasoles Solete"})-
   [R:FACTURA*]-> (B:Empresa {nombre:"Semillas Sidi"})
   RETURN P
   ```

2. Tendremos un plan de ejecución idéntico al que vimos con el índice, salvo que el primer paso se titulará NodeUniqueIndexSeek en lugar de NodeIndexSeek. Restricciones de clave de nodo. Extienden el tipo anterior a varias propiedades.

   ```
   CREATE CONSTRAINT NoRepiteNombreCompleto
   ```

```
ON (n:Persona)
ASSERT (n.Nombre, n.Apellido1, n.Apellido2)
IS NODE KEY
```

3. Existencia de propiedades. Aseguran que todo nodo de un tipo contendrá, con seguridad, una propiedad determinada. Un ejemplo:

```
CREATE CONSTRAINT ExistenciaEmpleados
ON (n:Empresa)
ASSERT EXISTS (n.empleados)
```

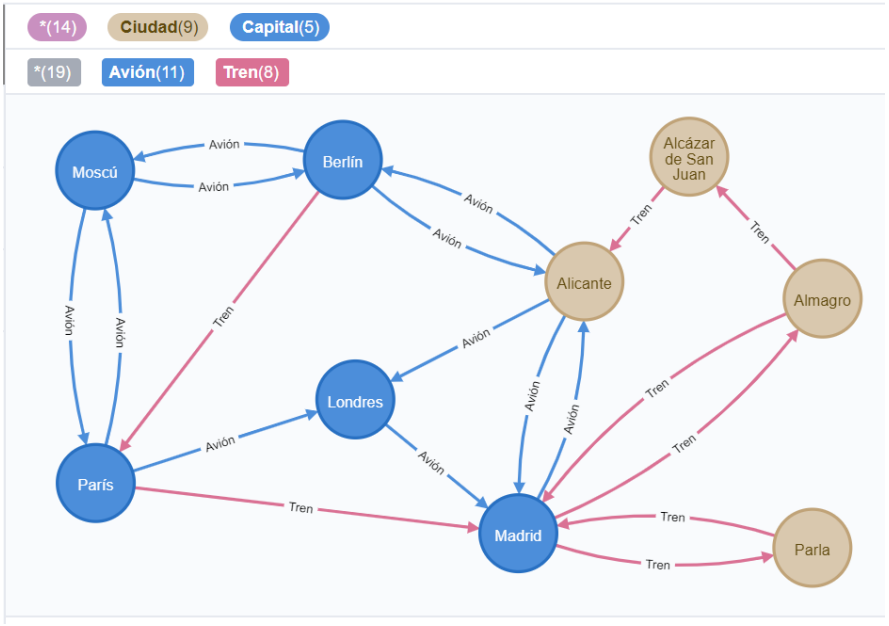
4. Existencia de propiedad en una relación. Análogo al anterior, con la misma sintaxis, pero para relaciones.

Solo la primera propiedad está incluida en la versión gratuita de Neo4J, las 3 restantes solo están disponibles si se adquiere la versión Enterprise. Si en cualquier momento queremos ver la lista de restricciones, debemos teclear:

```
CALL db.constraints
```

# Caso práctico

Consideramos un imaginario grafo de transportes que podemos crear con estas instrucciones Neo4j:




Como se ve se incluyen nodos de tipo Ciudad y de tipo Capital (las capitales también están etiquetadas como ciudades). Ambos tipos de localidades tienen propiedades habitantes y el nombre del lugar. Si además es solo Ciudad también tiene una propiedad provincia. Además, entre dos localidades puede haber transporte por avión y/o por tren. En el caso del avión se conoce la Aerolínea y la Hora del vuelo (solo incluimos un máximo de un vuelo entre cada par de lugares). Queremos escribir consulltas para encontrar:

1. Ciudades con conexión directa a Madrid por cualquier medio de transporte (mostrar el camino, es decir el punto de partida, la relación y la llegada).
2. Ciudades desde las que es posible llegar a Alicante en 2 o más pasos.
3. Queremos conocer todos los caminos de longitud menor o igual a 5 tales que el penúltimo lugar visitado sea “París”.
4. Camino más corto para llegar desde París hasta Alicante (idea: investigar el predicado shortestpath.
5. Longitud del camino más corto desde París hasta Alicante.
6. Camino más corto de Berlín a París que no incluya ningún trayecto en tren.
7. Parejas B, A/i>de ciudades en las que hay un vuelo directo de A a B, ordenadas alfabéticamente por B.
8. Por avería, el trayecto entre tren entre Almagro y Alcázar de San Juan ha quedado momentáneamente suspendido. Queremos añadir un flag estado:"Suspendido" para indicar esta contingencia. ¿Cómo se haría?
9. Finalmente la avería al a que hacía referencia la pregunta anterior va para largo, por lo que se decide suprimir este viaje en tren del grafo. Escribir la instrucción para hacerlo.
10. Número de conexiones de París con otras ciudades (en terminología de grafos: grado de salida del nodo París).

11. Número de conexiones que parten de París, pero esta vez considerando solo las de tipo avión.

12. Número de habitantes de la Ciudad/Capital con más habitantes de todas las que tienen acceso viajando en avión desde “París” (conexión con cualquier número de estaciones intermedias)

13. ¿Qué instrucción debemos escribir para comprobar el plan de ejecución de la instrucción anterior incluyendo el número de consultas a la base de datos?

14. ¿Qué indices podemos crear para asegurar que la instrucción anterior realiza menos accesos? Crearlos y comprobar que es así en efecto

15. Crear uná restricción para asegurar que no podemos introducir por descuído dos capitales con el mismo nombre.

# Enlaces de interés

- Neo4j Cypher Refcard 4.0. Muy útil como referencia rápida
- Graph Databases. Ian Robinson, Jim Webber and Emil Eifren. Editorial O'Reilly.. Un libro básico para aprender Neo4j, se puede descargar gratis aquí.
- The Practitioner’s Guide to Graph Data. Denise Gosnell and Matthias Broecheler. O'ReillyNo utiliza Neo4j sino el otro estándar para bases de datos, Gremlin. Enfatiza el paso desde bases de datos relacionales a bases de dtos orientadas a grafos comparando implementaciones en SQL y en grafos con Gremlin.