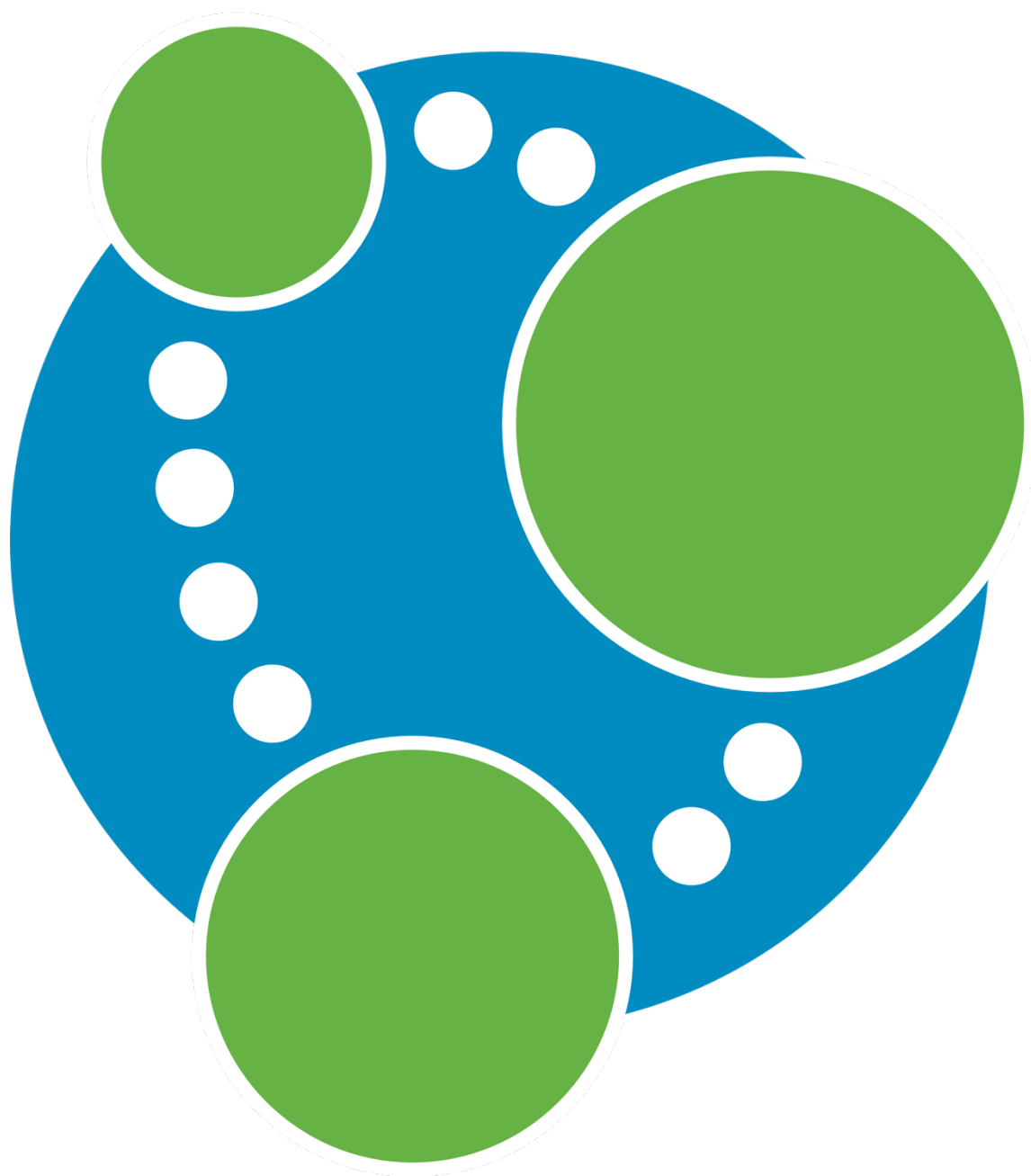


NEO4J

Juan Casado Ballesteros

Gino Cocolo Rodríguez



ÍNDICE

PROBLEMA EXISTENTE	3
SOLUCIÓN.....	4
CARACTERÍSTICAS ADICIONALES	4
DESVENTAJAS.....	5
NEO4J.....	6
MODELO DE DATOS.....	6
NODOS.....	6
ETIQUETAS	6
RELACIONES.....	7
PROPIEDADES	7
CARACTERÍSTICAS PROPIAS DE UNA BASE DE DATOS	7
LENGUAJE DE CONSULTAS.....	7
CREATE-DELETE	7
SET-REMOVE	8
MATCH.....	8
RETURN.....	8
WHERE	9
ORDER BY	9
LIMIT	9
WITH.....	9
UNION	9
FOREACH	9
REPLICACIÓN	10
DEMOSTRACIÓN	11
PRIMERA PARTE	11
SEGUNDA PARTE	14
CONCLUSIONES.....	19
BIBLIOGRAFÍA.....	20

PROBLEMA EXISTENTE

Las bases de datos relacionales almacenan información estructurada de forma rígida en forma de tablas. Para aprovechar completamente las características de estas bases de datos el diseño de tablas debe normalizarse hasta alcanzar la forma normal deseada.

La información organizada en tablas se relaciona mediante las operaciones proporcionadas por el álgebra relacional.

Esta estructura, aunque idónea para garantizar la consistencia y la no duplicidad de información no es la más adecuada para todas las necesidades.

Algunos de los problemas que plantea son los siguientes:

- Las uniones o relaciones entre tablas son operaciones del álgebra relacional que deben ejecutarse cada vez que se consulte la relación. Para evitar hacer esto en ocasiones se suele recurrir a des-normalizar las tablas lo cual rompe con los principios de diseño de las bases de datos relacionales.
- Ejecutar una unión entre tablas es equivalente a buscar valores procedentes de una tabla en otra lo cual tampoco es eficiente.
- La creación de ciertos tipos de índices necesarios para buscar conjuntos de datos por clave puede no ser óptima para buscar datos únicos. Debemos de ser conscientes en todo momento a la hora de crear nuevos índices de la forma en la que los datos serán buscados para cada clave.
- Las bases de datos relacionales se vuelven cada vez más lentas al realizar las operaciones de relación cuantos más datos hay almacenados en la base de datos.

Como conclusión podemos destacar que las bases de datos relacionales presentan problemas de rendimiento a la hora de realizar búsquedas que hacen uso de relaciones entre tablas.

Estos problemas no son solo propios de las bases de datos relacionales. Suceden también en otras bases de datos NoSQL como las documentales o las orientadas a clave valor. Estas bases de datos están especializadas para almacenar otros tipos de información como documentos o archivos de gran tamaño. No obstante, todas ellas son lentas a la hora de manejar relaciones de datos.

También es importante destacar que los lenguajes de consultas que estas otras bases de datos utilizan no están adecuados para manejar relaciones con una sintaxis suficientemente expresiva.

SOLUCIÓN

Las bases de datos orientadas a grafos tratan las relaciones como su principal activo. En lugar de centrarse en la forma en la que los datos son almacenados: tablas, documentos, columnas... se centran en las relaciones que existen entre ellos.

Su rapidez a la hora de manipular las relaciones proviene de la forma en la que se busca la información. En otros tipos de bases de datos de información debe de ser buscada mediante búsquedas secuenciales, binarias si la información está ordenada o por medio de índices. En el caso de las bases de datos orientadas a grafos las relaciones son directas, cada nodo contendrá punteros a las relaciones de las que participa y estas contendrán punteros a los nodos con los que el participante está relacionado de forma directa.

Esta forma de relacionar datos mediante dos direcciones es mucho más escalable que hacerlo mediante una búsqueda o que impedir crear relaciones directas por medio del propio lenguaje de consultas. Esto se traduce a que el aumento del coste de realizar las consultas será lineal con respecto a la cantidad de datos existente sin importar la cantidad de relaciones que haya que resolver.

Características adicionales

Las bases de datos orientadas a grafos suponen algo más que un cambio en la forma de almacenar la información. También proporcionan mejores técnicas de visualización de la información pensadas para visualizar grafos con los que se pueda interactuar.

No obstante, su característica más relevante es la de proporcionar nuevos lenguajes de consultas orientados a manipular relaciones. En el caso de neo4j este lenguaje es Cypher, pero otras bases de datos proporcionan lenguajes de consultas similares como es el caso de GSQL para la base de datos TigerGraph. Mediante este cambio de mentalidad que las bases de datos orientadas a grafos proporcionan y ayudados por las herramientas que ofrecen se puede de forma sencilla realizar acciones como:

- Averiguar el modelo habitual de las relaciones entre dos elementos. Lo cual nos permitiría:
 - Identificar elementos que se comportan de forma distinta para poder indagar en la razón por la que lo hacen. (Detección de fraude o usuarios maliciosos)
 - Identificar elementos similares. (permitiría por ejemplo realizar recomendaciones a un usuario en base a lo que los otros usuarios que se comportan como él estén haciendo)
- Predicción de atributos en base a los nuevos flujos de relaciones.
- Crear agrupaciones de modelos similares dentro de una misma relación.

Desventajas

Casa tipo de base de datos tiene ciertas ventajas sobre el resto sin ser ninguna la solución ideal para todos los problemas.

Se recomienda que el tamaño de los nodos sea lo menor posible no siendo aptas para almacenar archivos de gran tamaño.

La resolución de las consultas no es transaccional. Es decir, si los datos cambian mientras la consulta se realiza no sabremos cuales se nos proporcionarán, los que existían antes comenzar la consulta o los que existían después del cambio.

Cuando estas bases de datos se escalan no son capaces de proporcionar consistencia, aunque si tolerancia a particiones y alta disponibilidad. La falta de consistencia implica que puede que dos consultas consecutivas no proporcionen la misma información de forma momentánea, aunque sí vayan a hacerlo eventualmente en el futuro.

NEO4J

Neo4j es una base de datos orientada a grafos de código abierto e implementada en Java. Como lenguaje de consultas utiliza Cypher sobre un socket HTTP de modo que desde cualquier otro lenguaje que tenga acceso al stack de red se pueden realizar consultas.

Modelo de datos

Los datos almacenamos en la base de datos de organizan creando un grafo. Los elementos más relevantes de este grafo son los siguientes:

Nodos

Desde el punto de vista de la orientación a objetos los nodos serían entidades o instancias, desde el punto de vista de las bases de datos relacionales los nodos serían las filas.

Un nodo es la unidad completa mínima manejada por las bases de datos orientadas a grafos. Un nodo podrá tener un conjunto de atributos o pares clave valor de clave y valor conocidos e instanciados.

:Persona
nombre: Pepe
edad: 21

Etiquetas

Desde el punto de vista de la orientación a objetos las etiquetas serían las clases, desde el punto de vista de las bases de datos relacionales las etiquetas serían las tablas.

Las etiquetas son los distintos tipos de nodos que existirán en la base de datos. Las etiquetas pueden asignarse y retirarse de los nodos de forma dinámica a lo largo del ciclo de vida de los datos de modo que la instanciación de un nodo no nos compromete con una etiqueta concreta. Adicionalmente un nodo podrá tener asignada más de una etiqueta.

:Persona
nombre: Pepe
edad: 21

:Trabajador
trabajo: Panadero
lugar: Pontevedra

:Persona: :Trabajador
nombre: Pepe
edad: 21
trabajo: Panadero
lugar: Pontevedra

Una etiqueta puede requerir de la existencia de cero a varios pares de clave valor en el nodo, es decir, puede ocurrir que si una etiqueta no requiere de la existencia de pares de clave valor para los nodos asignados con ella esta pueda ser asignada a cualquier nodo en cualquier momento.

Relaciones

Las relaciones conectan dos etiquetas entre sí. Las relaciones dotan al grafo de una estructura. Las etiquetas relacionadas pueden ser la misma o ser distintas y como máximo pueden ser dos. Todos los nodos dotados con una etiqueta comenzarán a formar parte de todas las relaciones en las que la etiqueta intervenga.

Las relaciones son direccionales, es decir, van desde una etiqueta a otra y además pueden tener propiedades. Las propiedades de las etiquetas se asignan a parejas de nodos.



Son las parejas de clave valor que pueden encontrarse tanto en nodos como en relaciones. El valor también puede ser una lista de valores asignada a la clave.

Características propias de una base de datos

Del mismo modo que sucede en otros tipos de bases de datos en node4j podremos crear índices con la intención de aumentar el rendimiento de las consultas.

También podremos garantizar la unicidad de una propiedad de los nodos asignados a una etiqueta. En el ejemplo utilizado podría ser garantizar que dos personas no puedan tener el mismo nombre.

El problema de las restricciones de unicidad es que para poder ser implementadas debe de existir un índice sobre dicha propiedad de las etiquetas.

También podremos importar datos en formato .csv desde archivos locales o remotos como en cualquier otra base de datos.

```
LOAD CSV FROM file:///personas.csv AS personas
CREATE (:Persona {nombre: personas[0], edad: personas[1]})
```

Lenguaje de consultas

Node4j utiliza el lenguaje de consultas Cypher, el cual solo funciona en su propia base de datos.

CREATE-DELETE

Permite añadir nodos o instancias sobre una etiqueta.

```
CREATE (pepe :Persona {nombre: "Pepe", edad: 21})
CREATE (naranja :Producto {nombre: "Naranja", precio: 5.23})
```

También permite crear relaciones entre nodos.

```
CREATE (pepe)-[:COMPRAR]->(naranja)
```

Podremos borrar los nodos y las relaciones mediante DELETE

```
MATCH (()-[compras:COMPRAR]-()) DELETE compras
```

```
MATCH (persona :Persona) WHERE persona.nombre="Pepe" DELETE persona
```

SET-REMOVE

SET nos permite cambiar el valor de una propiedad por otro y REMOVE establecerá el valor de la propiedad como null.

```
MATCH (persona :Persona) WHERE persona.nombre="Pepe" SET persona.edad = 22
```

```
MATCH (persona :Persona) WHERE persona.nombre="Pepe" REMOVE n:edad
```

MATCH

Esta sentencia nos permite buscar patrones dentro del grafo de datos. Los patrones son definidos por una etiqueta ya sea de nodo o relación y por paréntesis, corchetes y flechas indicando la dirección de las relaciones.

Como podemos ver en las consultas anteriores la sentencia MATCH es muy expresiva:

- () Cualquier nodo
- [] Cualquier relación
- (:Etiqueta) Un nodo de un tipo concreto
- [:Etiqueta] Una relación de un tipo concreto
- {clave:valor, clave:valor} Restricciones sobre las propiedades de nodos o relaciones concretas.
- (nodo :Etiqueta) o [relación :Relacion] Crean variables ficticias equivalentes a cualquier nodo o relación de la etiqueta indicada que cumpla las restricciones que se le impongan.
- ()-[]-()-[]-() Tres nodos cualquiera relacionados por dos relaciones cualquiera.

Mediante construcciones similares a estas se puede crear cualquier patrón dentro de un grafo.

RETURN

La sentencia return nos permite indicar que partes del patrón creado mediante MATCH nos interesa obtener como resultado de haber ejecutado la consulta.

Se pueden obtener nodos, relaciones o subgrafos.

```
MATCH (persona :Persona)-[:COMPARA]->(:Producto) RETURN persona
```

```
MATCH (:Persona)-[compra :COMPARA]->(:Producto) RETURN compra
```

```
MATCH grafo = (:Persona)-[:COMPARA]->(:Producto) RETURN grafo
```


WHERE

Añade restricciones y filtros al patrón creado mediante MATCH. Se debe destacar que algunos de estos filtros se pueden incluir dentro del propio patrón mientras que otros no se podrían.

```
MATCH (persona :Persona) WHERE persona.edad < 30 RETURN persona
MATCH (persona :Persona) WHERE persona.nombre = "Pepe" persona
```

ORDER BY

Permite ordenar el patrón seleccionado de forma ascendente (por defecto) o descendente (DESC) según una o varias propiedades.

```
MATCH (persona :Persona)
RETURN persona.nombre
ORDER BY persona.nombre
```

LIMIT

Permite truncar la cantidad de instancias del patrón que serán devueltas por la consulta.

```
MATCH (persona :Persona) RETURN persona LIMIT 3
```

WITH

Permite aplicar filtros y operaciones matemáticas a nivel de propiedad y de grafo sobre el patrón indicado en MATCH antes de devolverlo. Los filtros más comunes son por agregación, por orden y por factor de ramificación.

```
MATCH (persona :Persona)-[:COMPRAR]->(producto :Producto)
WITH persona, producto, count(*) AS cantidad_producto WHERE cantidad_producto < 5
RETURN persona.nombre

MATCH (persona :Persona)
WITH persona ORDER BY persona.nombre DESC LIMIT 3
RETURN collect (persona.nombre)
```

UNION

Combina el resultado de dos consultas distintas. Por defecto se eliminarán los duplicados, pero podremos mantenerlo aumentando la velocidad de la consulta mediante la sentencia ALL.

```
MATCH (persona :Persona) RETURN persona.nombre AS nombre
UNION ALL
MATCH (product: Producto) RETURN product.nombre AS nombre
```

FOREACH

Permite aplicar operaciones a todos los patrones seleccionados.

```
MATCH (persona :Persona)
WITH COLLECT(persona) AS lista
FOREACH (p IN lista | SET p.edad=p.edad+1)
```

Replicación

Al igual que la mayoría de las bases de datos `node4j` proporciona herramientas de replicación.

Su modelo de replicación busca proporcionar la posibilidad de realizar particiones y alta disponibilidad de los datos. Se crearán nodos maestros y nodos réplica. Los nodos maestros serán responsables de garantizar la consistencia eventual de los datos y de realizar las escrituras de los datos. Por el contrario, los nodos réplicas son solo de lectura. Algunos de los nodos réplica tendrán la responsabilidad adicional de realizar informes y análisis sobre los datos almacenados.

Este esquema de replicación está orquestado por los llamados servidores de aplicación. Los servidores de aplicación recibirán las peticiones en el lenguaje Cypher y dependiendo de si son lecturas o escrituras las reenviarán a nodos maestros o de réplica para ser procesadas. En caso de que sean una escritura las réplicas quedarán desactualizadas hasta que eventualmente y de forma asíncrona alguno de los nodos maestros las actualice.

DEMOSTRACIÓN

Dividiremos la demostración en dos partes.

En la primera crearemos nuestra propia base de datos mediante docker y accederemos a ella por medio de la utilidad de escritorio de neo4j. Sobre esta base de datos insertaremos algunos datos similares a los que se han utilizado como ejemplo para las consultas demostrativas utilizadas en la explicación de Cypher.

En la segunda parte utilizaremos una base de datos online con datos precargados. Neo4j dispone bases de datos de este tipo sobre las que poder realizar tutoriales con el objetivo de aprender Cypher.

Primera parte

Podemos iniciar una nueva base de datos con docker mediante el comando:

```
docker run --publish=7474:7474 --publish=7687:7687 neo4j:4.0
```

En nuestro caso hemos transformado dicho comando en un servicio de docker-compose sobre el que adicionalmente montamos un contenedor con varios archivos .csv con datos de prueba con los que pode hacer consultas:

```
neo4j:
  image: neo4j:4.0
  ports:
    - 7474:7474
    - 7687:7687
  volumes:
    - ./data:/var/lib/neo4j/import
```

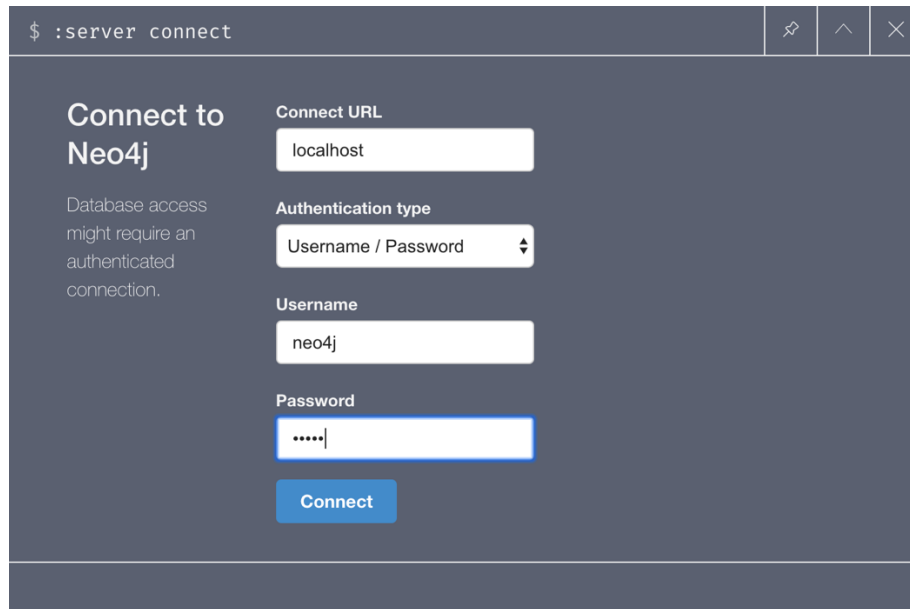
A continuación, mediante la utilidad de neo4j para realizar conexiones y consultas a sus bases de datos disponible en <https://neo4j.com/download/> nos conectaremos a la base de datos que acabos de crear.

Las siguientes capturas que se mostrarán serán realizadas sobre la interfaz gráfica que nos proporciona esta utiliad. La consulta a la que cada captura hace referencia se encuentra en la parte superior izquierda.

Para conectarlos a la base de datos por tanto la consulta ejecutada ha sido:

```
:server connect
```

El usuario y la contraseña iniciales de la base de datos son neo4j, se nos pedirá que cambiemos la contraseña por otra distinta la primera vez que nos conectemos a la base de datos.



The image shows the 'Connect to Neo4j' dialog box in a terminal window. The title bar reads ':server connect'. On the left, it says 'Connect to Neo4j' and 'Database access might require an authenticated connection.' On the right, there are input fields for 'Connect URL' (localhost), 'Authentication type' (Username / Password), 'Username' (neo4j), and 'Password' (masked with dots). A 'Connect' button is at the bottom.

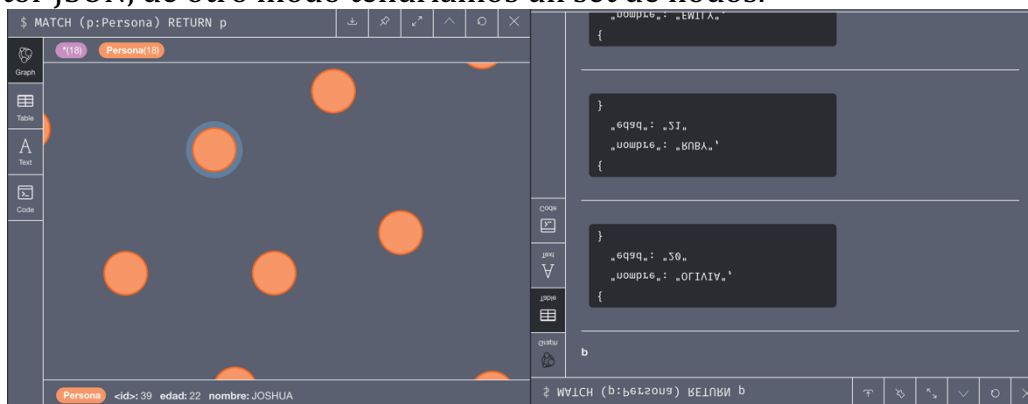
A continuación, cargaremos los datos almacenados en los archivos .csv incorporados al contenedor.

```
LOAD CSV FROM "file:///node4jNames.csv" AS personas
CREATE (:Persona{nombre: personas[0], edad: personas[1]})
```

Podremos ver los nuevos datos incorporados a la base de datos mediante:

```
MATCH (p:Persona) RETURN COLLECT(p)
```

Incluimos la sentencia COLLECT para que los datos sean transformados en un vector JSON, de otro modo tendríamos un set de nodos.



The image shows the Neo4j Desktop interface. The top bar shows the query: 'MATCH (p:Persona) RETURN p'. The left sidebar has icons for Graph, Table, Text, and Code. The main area displays a graph with orange circular nodes. The right sidebar shows the results in JSON format, with three entries. The bottom status bar shows 'Persona <id>: 39 edad: 22 nombre: JOSHUA'.

Seguidamente incluiremos el resto de los datos:

```
LOAD CSV FROM "file:///node4jProducts.csv" AS productos
CREATE (:Producto{nombre: productos[0], precio: productos[1]})

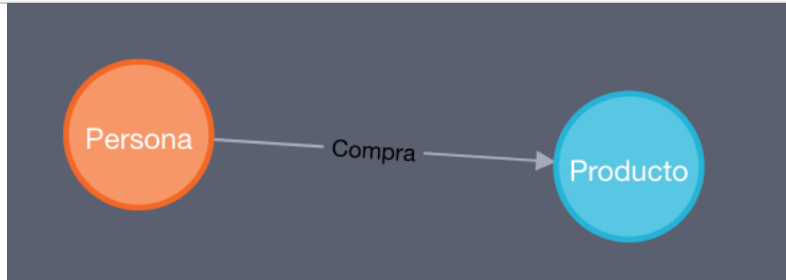
LOAD CSV FROM "file:///node4jSales.csv" AS sales
MATCH (persona :Persona), (producto :Producto)
WHERE persona.nombre=sales[0] AND producto.nombre=sales[1]
CREATE (persona)-[:Compra]->(producto)
```

Finalmente crearemos dos restricciones sobre las “claves” de los nodos lo que garantizará que sean datos únicos al mismo tiempo que nos añade índices sobre ellas.

```
CREATE CONSTRAINT ON (persona:Persona) ASSERT persona.nombre IS UNIQUE  
CREATE CONSTRAINT ON (producto:Producto) ASSERT producto.nombre IS UNIQUE
```

A continuación, podremos ver el esquema final de nuestra base de datos:

```
CALL db.schema.visualization()
```

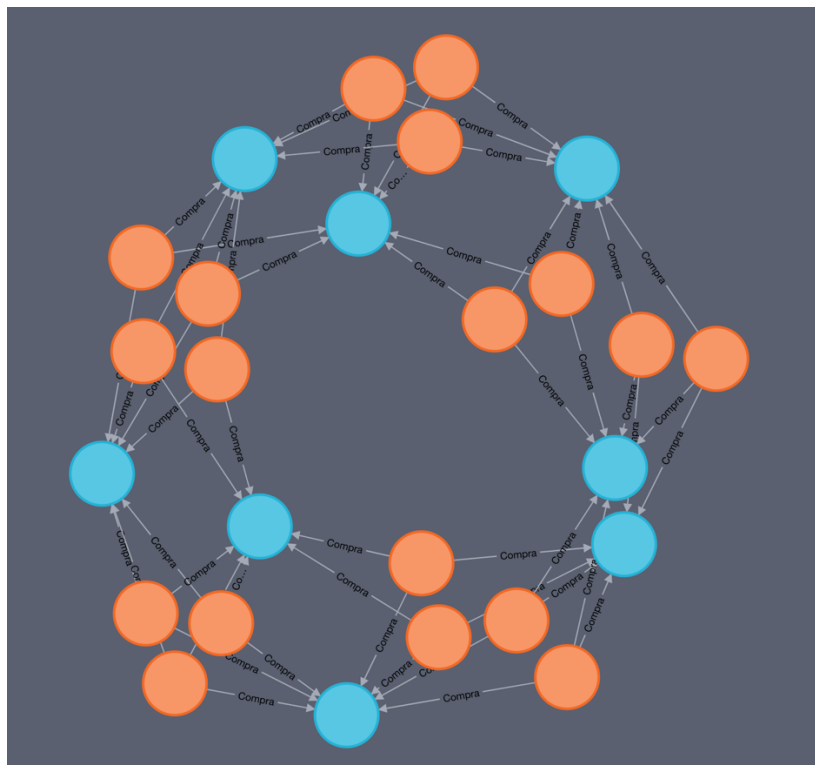


Podremos ver el listado de procedimientos disponibles para llamar con CALL mediante:

```
CALL dbms.procedures()
```

Mediante la siguiente consulta podremos ver todos los datos almacenados en la base de datos:

```
MATCH g=()-[]->() RETURN g
```



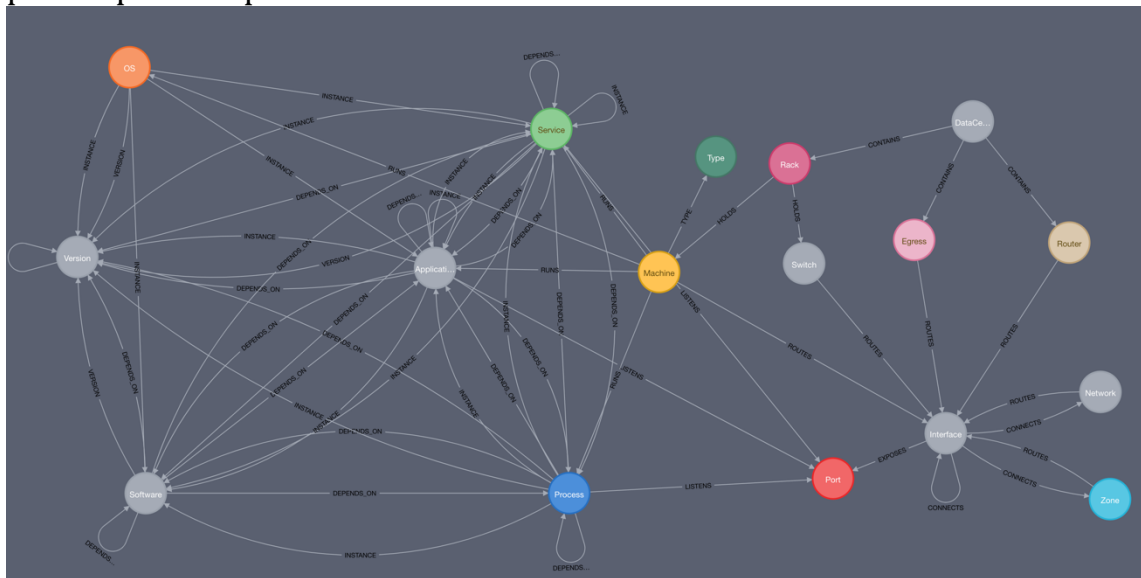
En este punto nuestra base de datos estará lista y podremos hacer consultas como ver los productos más comprados:

```
MATCH ()-[:Compra]->(producto :Producto)
WITH producto.nombre AS nombre, count(producto) AS cantidad
RETURN nombre, cantidad
ORDER BY cantidad DESC
```

Segunda parte

Crearemos a continuación una base de datos en el sandbox de neo4j. Se nos ofrecen gran cantidad de data sets disponibles para insertar dentro de nuestra base de datos.

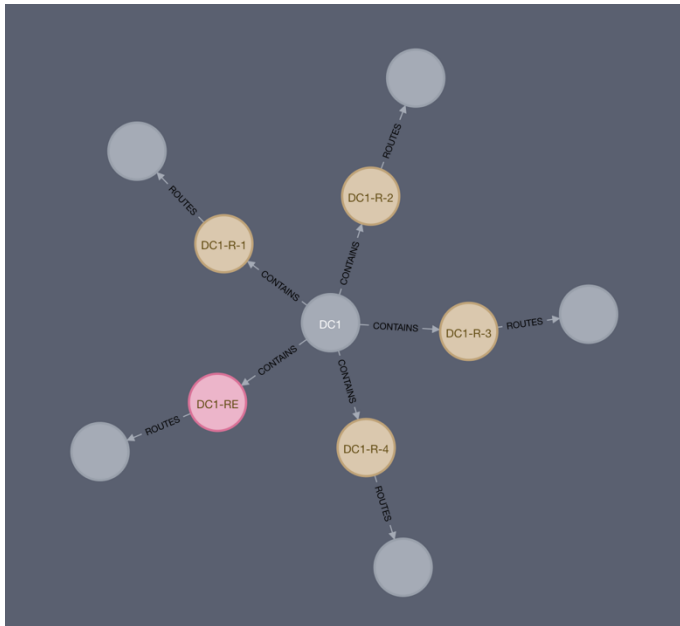
Elegiremos uno que contiene datos de la arquitectura de un data center. Rápidamente podremos ver que la complejidad del esquema y la cantidad de datos insertados es mucho mayor en esta base de datos que en la que creamos para la primera parte.



Obtener un grafo en el que se muestran los routers y las interfaces que estos tienen de un data center concreto.

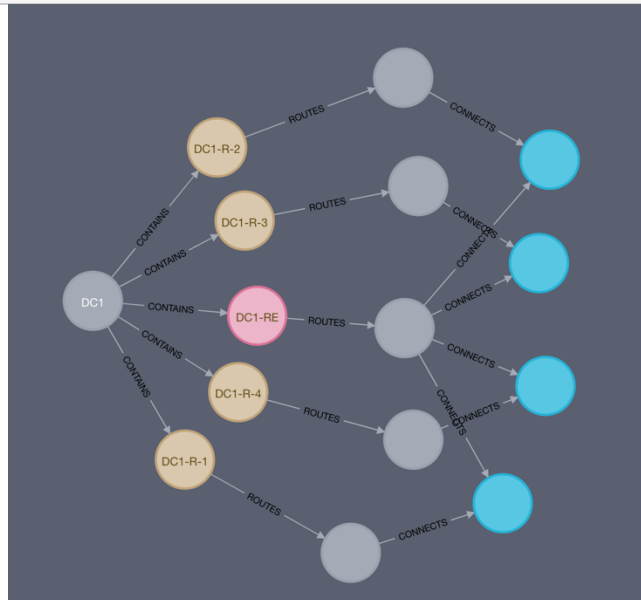
```
MATCH network = (dc:DataCenter {name:"DC1",location:"Iceland, Reykjavik"})
-[:CONTAINS]->(:Router)
-[:ROUTES]->(:Interface)
RETURN network;
```

El data center tiene 5 routers uno de los cuales es a la vez un punto de ingreso. Cada router lo conecta con una sola interfaz.



Mostramos ahora todas las zonas desde las que se puede llegar al data center por alguna de sus interfaces.

```
MATCH (nr:Network:Zone)<-[:CONNECTS]-(re)
MATCH (dc)-[:CONTAINS]->(r:Router)-[:ROUTES]->(ri:Interface)-[:CONNECTS]->(nr)
RETURN *;
```



Podemos ver que tan solo son cuatro zonas pudiendo a cada una llegarse desde dos redes distintas y por tanto también desde dos routers distintos.

El mismo resultado podría haberse obtenido de forma abreviada mediante esta otra consulta en las que los nodos y las rutas son implícitos ya que se conocen las etiquetas de origen y destino.

```
MATCH path = (dc:DataCenter)-[*3]-(:Network)
RETURN path;
```

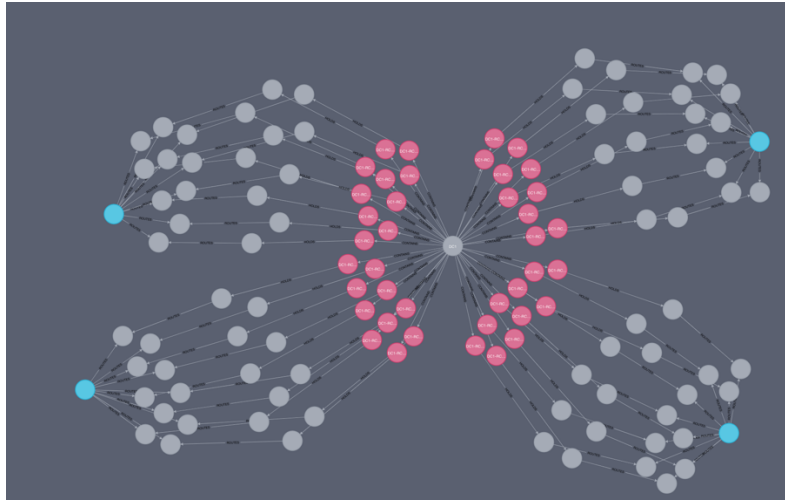
A continuación, veremos los racks pertenecientes a la data center y las zonas a las que cada equipo tiene acceso.

```
MATCH (dc:DataCenter {name:"DC1"})-[:CONTAINS]->(rack:Rack)-[:HOLDS]->(s:Switch)-[:ROUTES]->(si:Interface)-[:ROUTES]->(nr:Network:Zone)
RETURN *;
```

Lo cual también puede escribirse de forma resumida como:

```
MATCH network = (dc:DataCenter)-[*6]-(:Rack)
RETURN network;
```

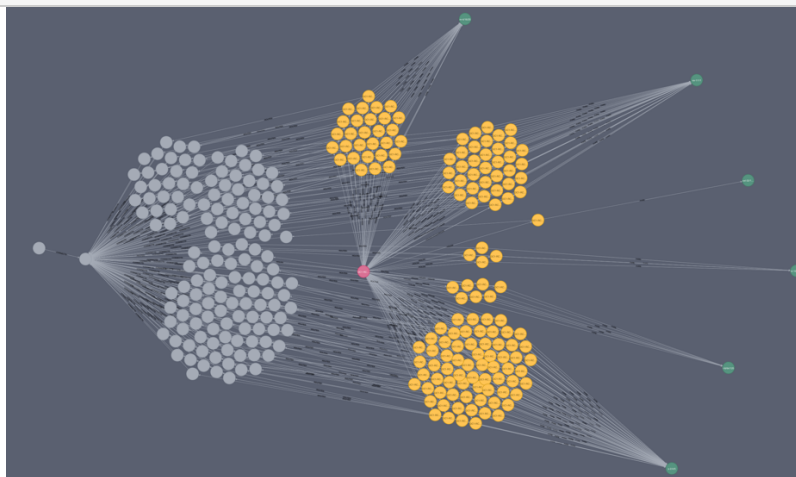
Debido a la cantidad de datos que la consulta debe de procesar hemos notado que mientras que la consulta explícita se ejecuta de forma casi instantánea la consulta resumida tarda considerablemente más.



Podemos ver que el centro de datos tiene cuatro grandes grupos de racks conectados cada uno a una sola de las zonas.

En cada rack habrá una gran cantidad de equipos, podremos visualizar estos mediante la siguiente consulta.

```
MATCH (r:Rack {name:"DC1-RCK-2-1"})-[:HOLDS]->(m:Machine),
      (m)-[:ROUTES]->(i:Interface)-[:CONNECTS]->(si)-[:ROUTES]->(s:Switch),
      (m)-[:TYPE]->(type:Type)
RETURN *
```



Podemos ver que en el rack hay muchos equipos distintos, cada uno de una categoría concreta, en total seis categorías. Exploraremos a continuación la distribución de una las categorías a lo largo de todos los racks.

La siguiente consulta proporciona un vector con el tipo de equipo y su cantidad en el data center.

```
MATCH (r:Rack)-[:HOLDS]->(m:Machine)-[:TYPE]->(type:Type)
RETURN type.id, count(*) as c
ORDER BY c DESC;
```

type.id	c
1	3760
0	2080
2	1360
3	520
4	200
5	80

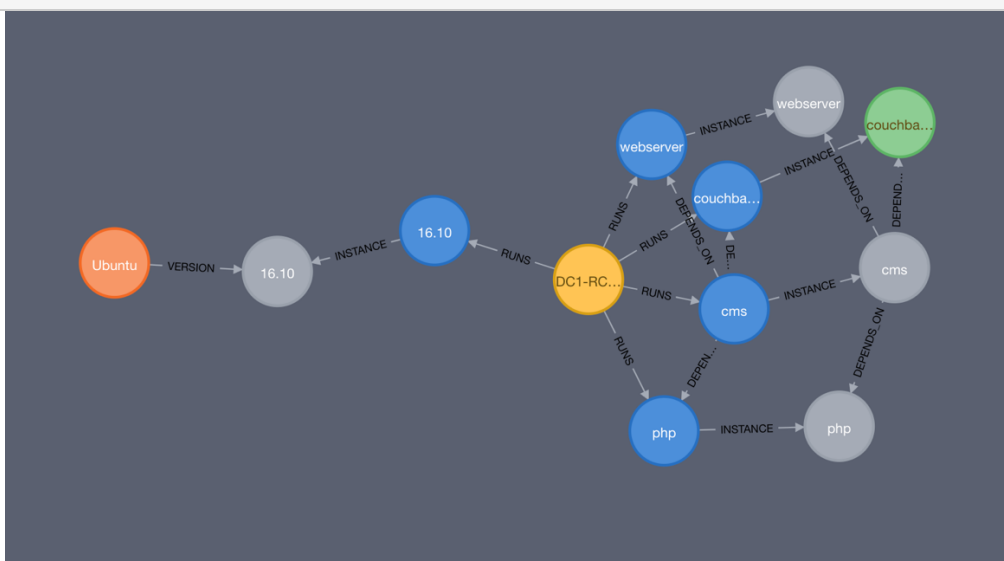
También podemos calcular las características combinadas de todos los equipos en el data center.

```
MATCH (m:Machine)-[:TYPE]->(type:Type)
RETURN count(*) as count, sum(type.cpu) as cpus, sum(type.ram) as ram,
sum(type.disk) as disk;
```

count	cpus	ram	disk
8000	24960	205280	494880

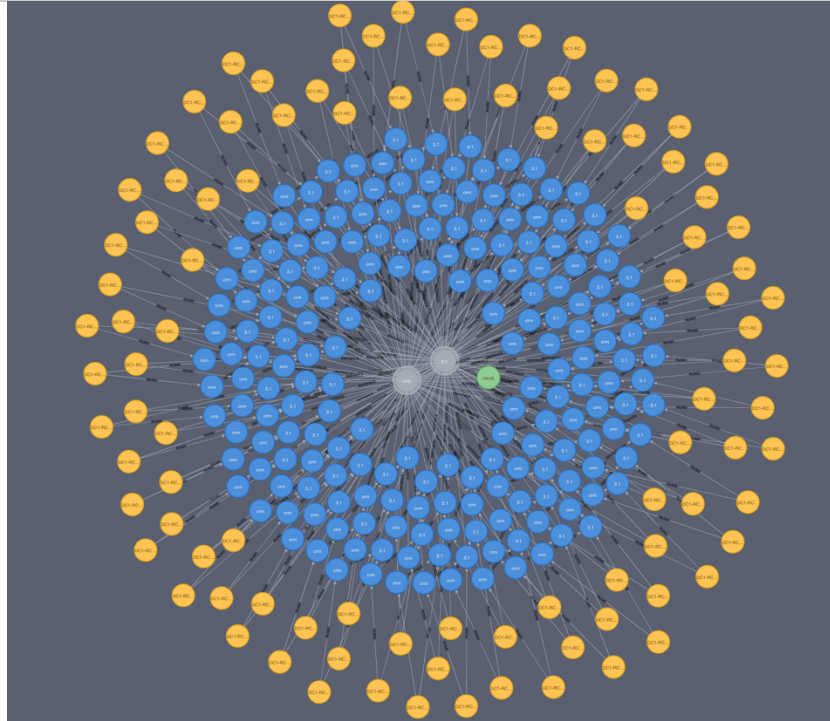
En cada equipo hay un sistema operativo instalado sobre el que están corriendo múltiples procesos, ambos con versiones concretas. Los procesos dependen los unos de los otros para poder funcionar. Podemos ver un ejemplo de esto para un equipo elegido de forma aleatoria.

```
MATCH (m:Machine) WHERE (m)-[:RUNS]->() AND rand() < 0.05 WITH m LIMIT 1
MATCH (m)-[r:RUNS]->(p:Process)-[i:INSTANCE]->(sv)
OPTIONAL MATCH (sv)-[v:VERSION]->(sw)
RETURN *
```



Veremos ahora todos los procesos que dependen de cualquiera de las versiones en cualquiera de las máquinas de la aplicación neo4j.

```
MATCH (s)-[:DEPENDS_ON]->(nv:Version)<-[:VERSION]-(n:Software:Service
{name:"neo4j"})
MATCH (s)<-[:INSTANCE]-(sp)<-[:RUNS]-(sm:Machine)
MATCH (sp)-[:DEPENDS_ON]->(np)-[:INSTANCE]->(nv)
MATCH (np)<-[:RUNS]-(nm:Machine)
RETURN sm as software_machine, sp as software_process, s as software, nv as
neo_version,np as neo4j_process, np as neo_machine n as sw
```



CONCLUSIONES

A lo largo de la demostración hemos podido ver las capacidades de visualización y análisis que neo4j nos proporciona. En caso de que sean las relaciones el activo principal a manejar dentro de la base de datos sin duda una base de datos orientada a grafos parece ser el mejor candidato. Reducir el tiempo en realizar consultas con múltiples JOIN suele ser una tarea común cuando se utilizan bases de datos relacionales de modo que eliminarlos directamente por una solución mejor parece adecuado.

A lo largo de los ejemplos hemos podido ver que la sintaxis de Cypher es sencilla de leer y de escribir permitiendo gran flexibilidad. En combinación con las herramientas de visualización se crea un entorno bastante potente para manipular y explorar grafos.

Debido a la posibilidad de expresar los resultados de las consultas en formato JSON estos quedan listos para ser exportados a otras aplicaciones de forma directa lo cual suele ser habitual en las bases de datos NoSQL trabajen directamente con datos en este formato o no.

Finalmente, la posibilidad de convertir el resultado de las consultas en un vector sobre el que aplicar una función a todos sus elementos nos ha parecido otra de las características destacables de esta base de datos.

BIBLIOGRAFÍA

Información general sobre neo4j, instalación, configuración y uso:

<https://neo4j.com/docs/operations-manual/current/>

Información sobre Cypher, el lenguaje de consultas para neo4j:

<https://neo4j.com/docs/cypher-manual/current/>

Presentaciones con introducción y ejemplos a neo4j:

<https://es.slideshare.net/maxdemarzi/>

Ejemplos de uso de neo4j con distintos lenguajes y distintos fines:

<https://github.com/maxdemarzi>

Documentación de otra base de datos orientada a grafos:

<https://docs.tigergraph.com>

Buenas prácticas para utilizar bases de datos orientadas a grafos:

<https://neo4j.com/blog/data-modeling-basics/>

Bases de datos neo4j con tutoriales para aprender Cypher:

<https://neo4j.com/sandbox/>