Introducción a Neo4j

Base de datos orientada a grafos

Trabajo de Base de Datos

Autor: Juan Eugenio Cortés Díaz

Resumen

En este documento se explicará con detalles como usar neo4j, desde su instalación, recorriendo su entorno gráfico, su lenguaje particular (Cypher, cql), etc…

No sin antes explicar un poco el origen de los grafos y que son los grafos. ¿Por qué?, muy sencillo neo4j es una base de datos orientada a grafos, así que antes de nada explicaremos el origen de la teoría de grafos, en que consiste un grafo y otras datos relevantes.

También toda la información estará basada en una base de datos real creada a raíz de la european go database que alberga los jugadores de go de europa. Si necesitas información sobre el go, recomendamos buscar tutoriales por la red, hay muchos.

Para finalizar listaremos otras bases de datos orientadas a grafos y unas conclusiones finales hablando de las cosas buenas y malas de neo4j.

ÍNDICE

1. Introducción

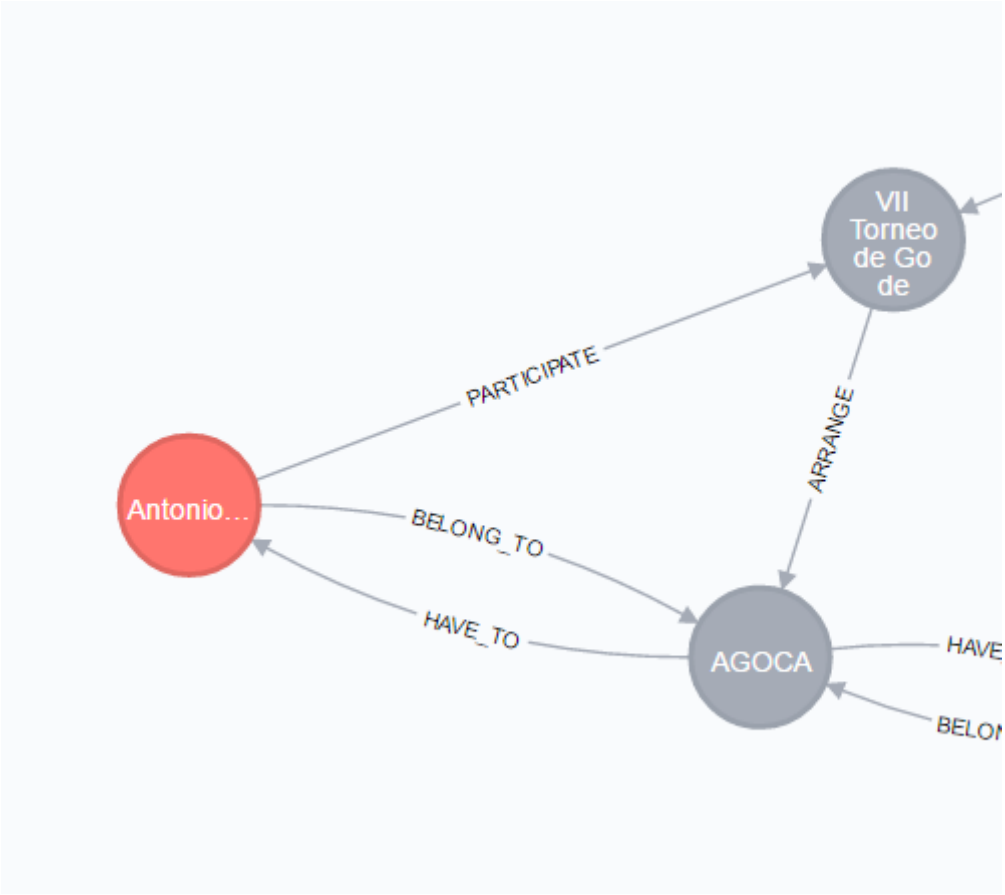
En el siglo XVIII, en la ciudad Königsberg surgió un problema que fue llamado el problema de los puentes Königsberg, la ciudad estaba divida en cuatro regiones distintas. Esto era porque en medio de la ciudad pasaba el rio Pregel.

Las cuatro regiones estaban unidas mediante siete puentes llamados Puente del herrero, Puente conector, Puente verde, Puente del mercado, Puente de madera, Puente alto y Puente de la miel. El problema consistía en encontrar un recorrido para cruzar a pie toda la ciudad, pasando sólo una vez por cada uno de los puentes, y regresando al mismo punto de inicio.

Esto lo resolvió Euler, partiendo de lo que futuramente se llamó la teoría de Grafos, considero cada puente como lo que llamamos vértice y la conexión de cada uno arista. Esto fue el comienzo de la teoría de grafos tal y como la conocemos hoy día.

A raíz de eso se empezó a hablar ya de vértices y aristas en los grafos.

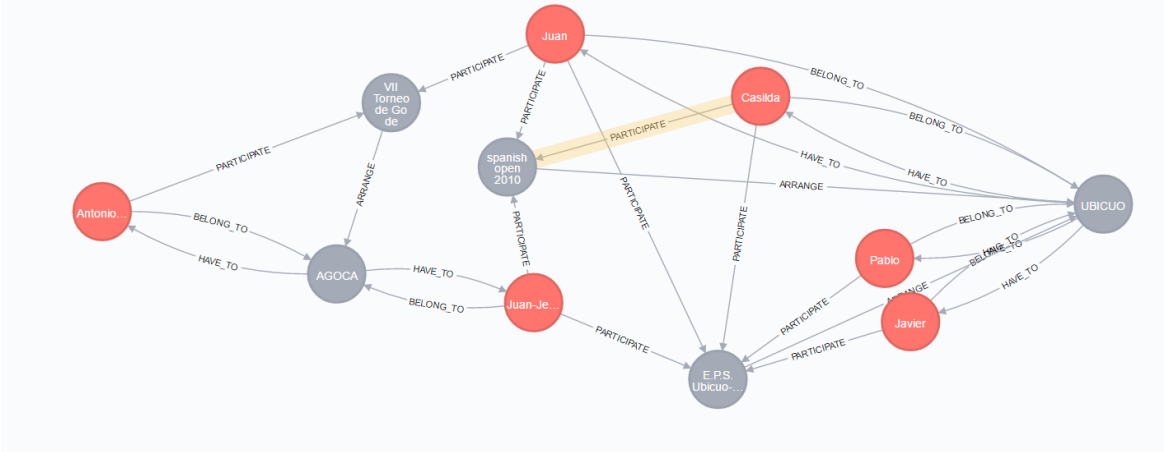
¿Pero que es un vértice?, ¿Qué es una arista?



Los vértices serían en esta imagen los círculos y las aristas serían las flechas que los conectan.

Las aristas pueden ser dirigidas o no, sino son dirigidas no tendrían flechas.

Un grafo seria todos los vértices y aristas que los conectan como por ejemplo el que se ha realizado para este trabajo:



Como se puede ver hay multitud de vértices e incluso algunos tienen ambas direcciones, en Neo4j no puedes poner bidireccionalidad al uso y haz de poner dos relaciones para conectar ambos vértices, lo que se llama aristas.

Como podéis comprobar lo que distingue a Neo4j es que es una base de datos orientada a grafos.

Esta base de datos que se usará a lo largo del documento como ejemplo está basada en la european go database.

La european go database, es una base de datos alojada en un servidor web, que informa de los jugadores que hay en activo en europa del maravilloso mundo del go, los torneos que han participado, los torneos que han ido, a que club pertenecen, etc…

Tournaments :

(0-75) of 75

Tournament Code	Class	Date	Description	Location	Club	Rank	GoR before_>_after	Placement	Rnds	Wins	Losses	Jigo
G160429A	A	2016-04-29	2nd European Go Grand Slam Tournament	DE,Berlin	RO,Bucu	5p	2677 --> 2675	6 / 12	4	2	2	0
E150726A	A	2015-07-26	59th European Go Congress - Open Tournament, 1st week	CZ,Liberec	RO,Bucu	8d	2686 --> 2677	33 / 761	5	2	2	0
T150404C	A	2015-04-04	6. China Cup Berlin	DE,Berlin	RO,Bucu	7d	2689 --> 2686	2 / 90	6	4	2	0
T150403C	A	2015-04-03	1. European Go Grand Slam Tournament	DE,Berlin	RO,Bucu	5p	2696 --> 2689	9 / 12	4	0	1	0
T140111D	B	2014-01-11	Kisei 2014 - Side tournament	ES,Madrid	RO,Bucu	7d	2707 --> 2696	11 / 125	4	2	2	0

El go es un juego de tablero estratégico para dos jugadores que se originó en la antigua China hace más de 2.500 años. Fue considerado una de las cuatro artes esenciales de la cultura china de la antigüedad. Las escrituras más antiguas que hacen referencia al go datan de las Analectas de Confucio.

Durante el juego, los jugadores se alternan turnos colocando piedras sobre las intersecciones vacías de un tablero de 19×19 intersecciones. Los principiantes usualmente juegan en tableros de menor tamaño, de 9x9 y 13x13.

El mundo del go desde la segunda guerra mundial rompió las fronteras asiáticas y llego al mundo occidental, es cierto que aún queda mucho para que en occidente llegue al nivel profesional de asia, pero cada día el número de jugadores aumenta. En europa hay miles de torneos, y hay clubs en casi cada capital de provincia.

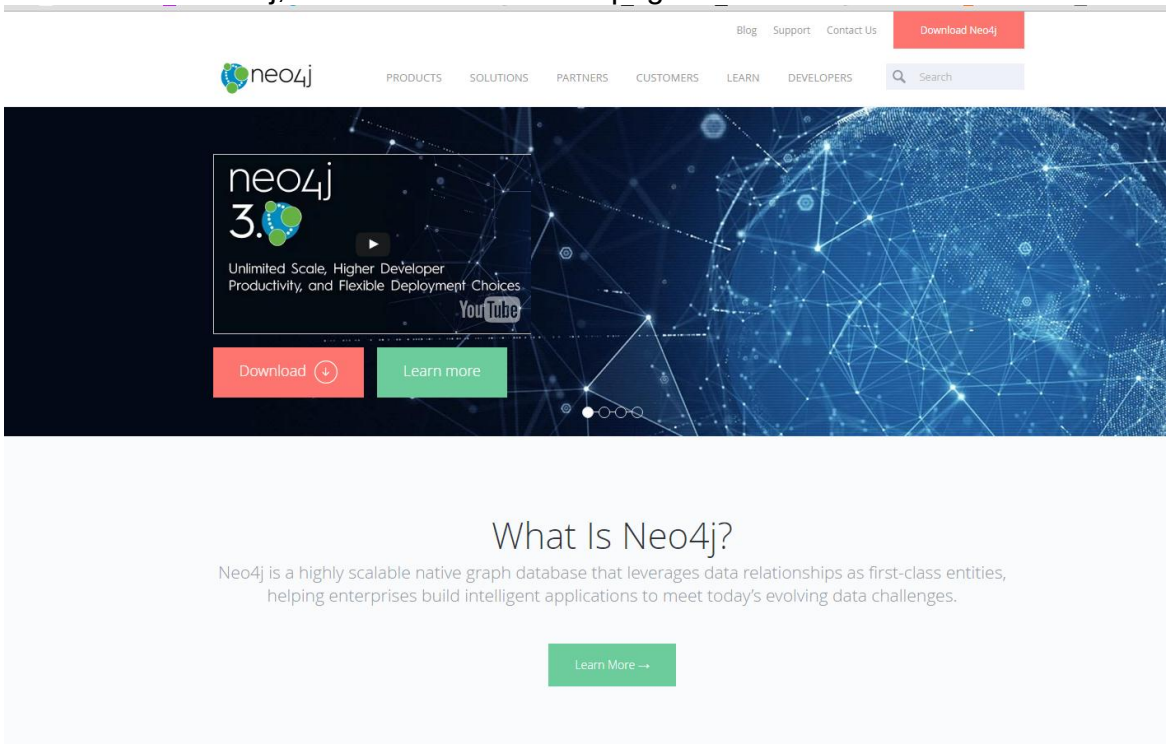
Por lo que en este documento aprenderemos a utilizar neo4j con una base de datos real de los jugadores de go europeos. Se ha cogido una muestra de jugadores de Sevilla y de Cadiz, usando 3 torneos reales, y dos asociaciones reales.

Los datos de los usuarios son reales y se pueden comprobar en la página de la european go database: http://www.europeangodatabase.eu/EGD/

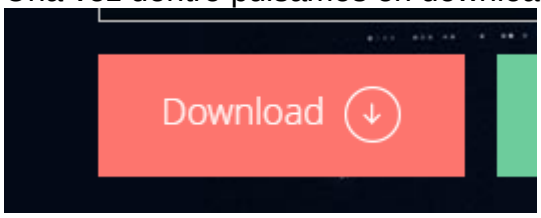
2. Tutorial

2.1. Instalación

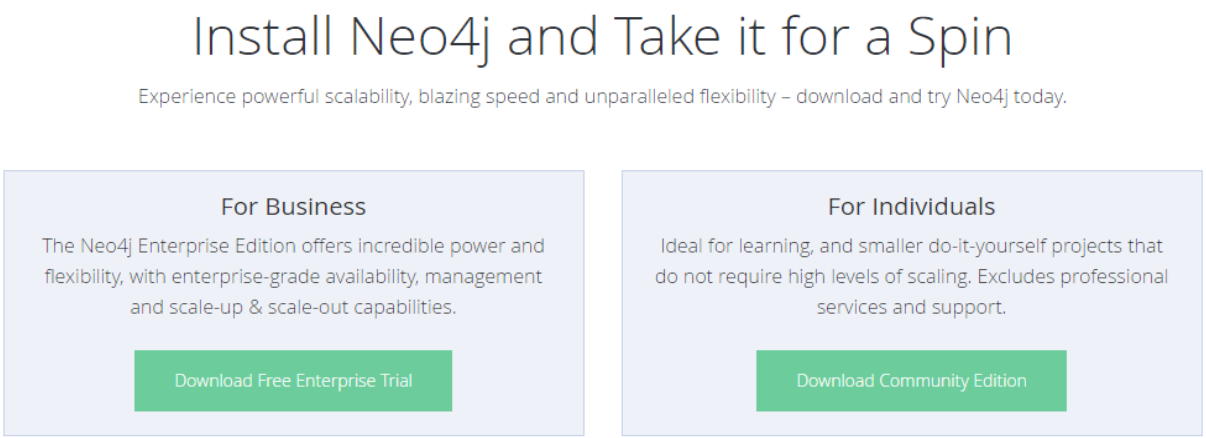
Para instalar Neo4j, nos meteremos en su página:



Una vez dentro pulsamos en download:

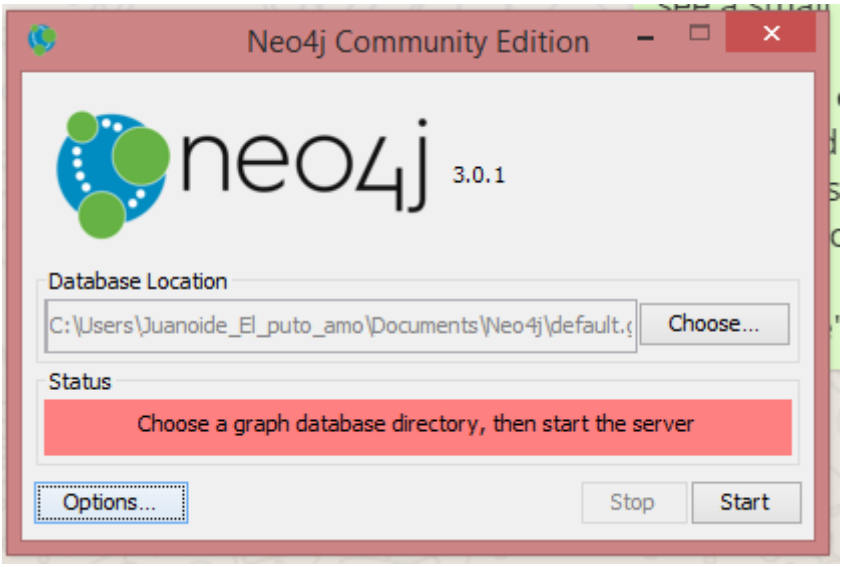


Una vez dentro de esta vista de la página web oficial de neo4j nos bajamos la versión Community Edition que es gratuita y tendrá todo lo necesario no llega a

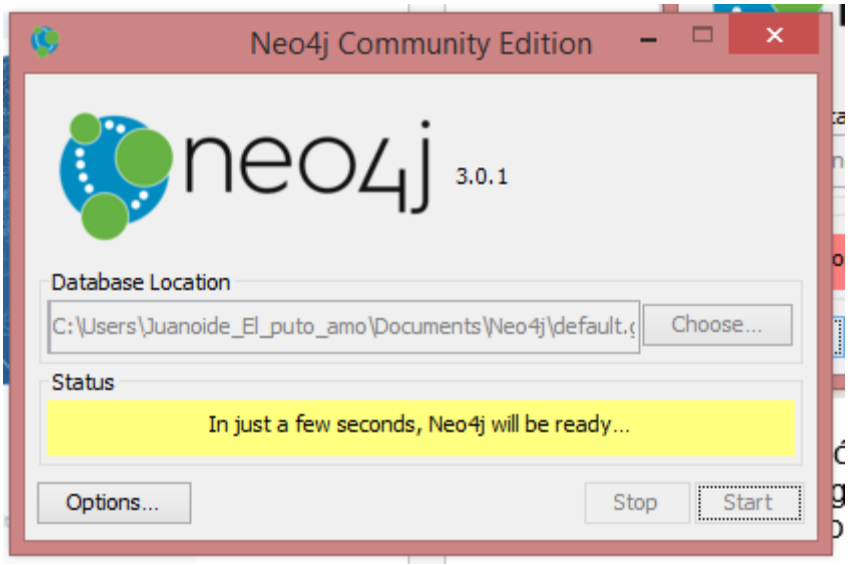


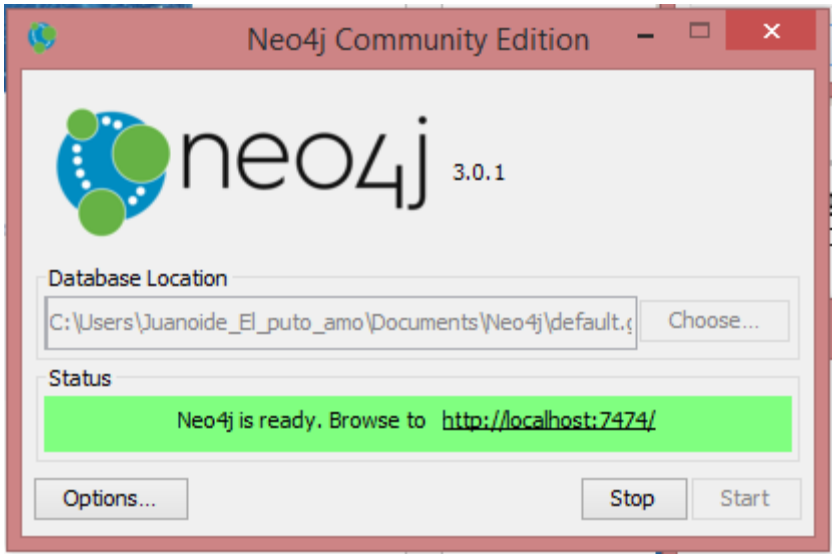
100 megas así que no tardará mucho en descargarse con una conexión media, si tu conexión es inferior a fibra óptica entonces es posible que tarde bastante.

Una vez instalada al ejecutarla nos saldrá lo siguiente:

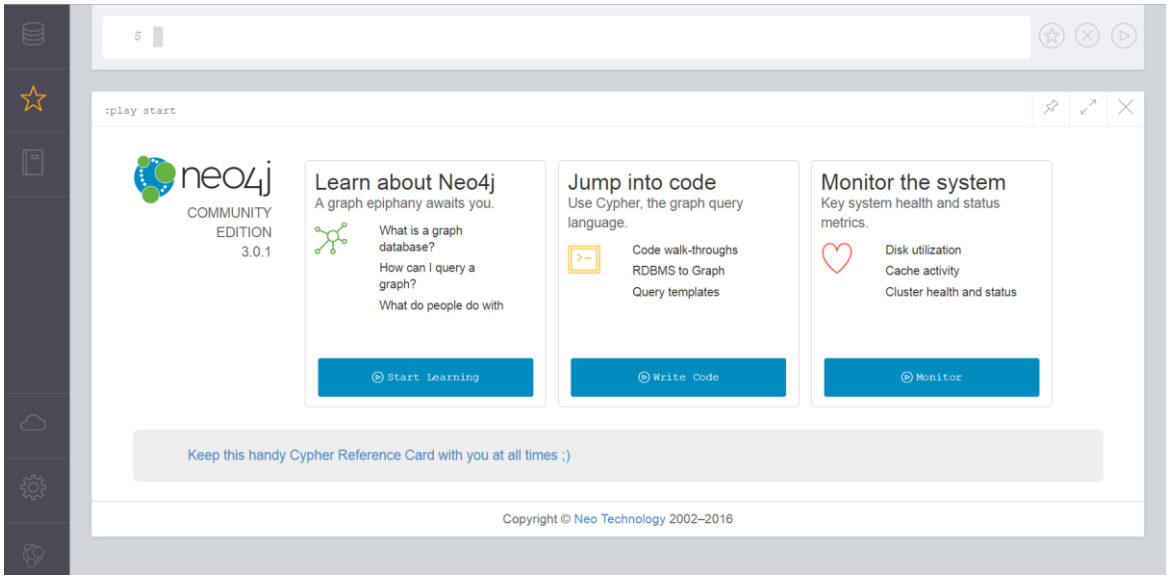


La localización de la base de datos puedes cambiarla y ponerla donde quieras. Una vez tengamos claro dónde va a estar pinchamos en start y nos cargará la base de datos tras unos segundos.





Ya está iniciado simplemente pinchando en la url que nos sale con el fondo verde.



Ahora podemos empezar a trabajar en ella desde el entorno gráfico.

2.2. Breve explicación de Cypher

Antes de explicar cómo funciona el entorno gráfico, vamos a dar una breve explicación de cypher.

Neo4j es un sistema gestor de bases de datos NoSQL que almacena toda la información en forma de grafos. Las bases de datos NoSQL se caracterizan por ofrecer alternativas a las clásicas bases de datos relacionales para problemas que se afrontan mejor con otra tecnología. Neo4j es, con una diferencia notable, el sistema gestor de bases de datos en grafos más popular, además de contar con licencia gratuita y ser de código abierto.

Para comunicar una aplicación con una base de datos Neo4j es necesario utilizar alguno de los drivers compatibles. El uso de Neo4j no limita el lenguaje para programar las aplicaciones que lo utilicen, ya que existen herramientas para conectarse a Neo4j para los lenguajes habituales: Java, .NET, JavaScript, Python, Ruby, PHP, R, Go, etc.

Neo4j cuenta con un lenguaje de consulta denominado Cypher. Con este lenguaje se pueden construir consultas de forma expresiva y eficiente para extraer información y para modificar la base de datos. Su sintaxis se basa en la búsqueda de patrones, aunque algunas de sus palabras clave se inspiran en SQL.

Un ejemplo de una relación con Cypher:

(Juan)-[variable:PARTICIPATE{Atributos}]->(VII Torneo de go de Cadiz)

Como se puede comprobar Cypher es un lenguaje bastante sencillo e intuitivo.

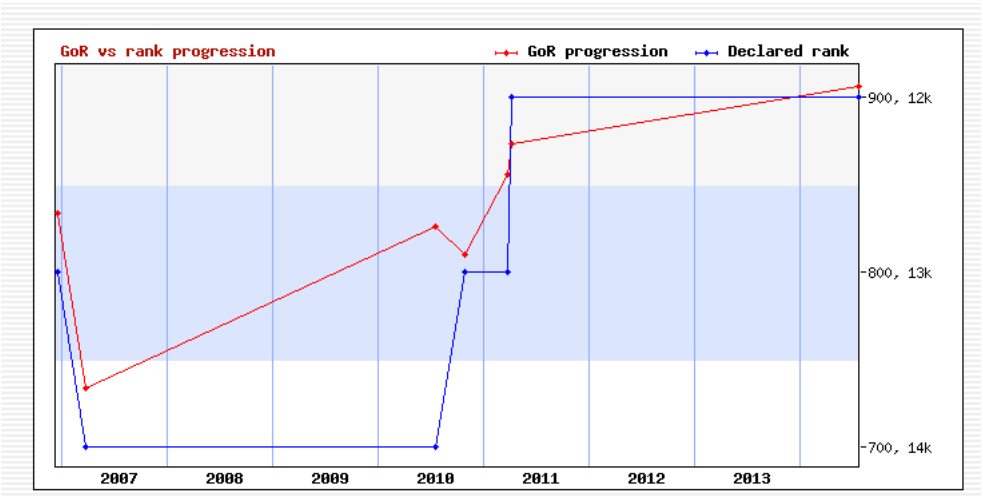
En el caso de ejemplo se ha realizado de esta manera.

Primero se ha creado el vértice de jugador (player):

```
CREATE (Cortes:Player { Lastname : 'Cortes', FirstName: 'Juan',
Countrycode: 'ES', Rank: '12k', Gor: 906 })
```

Donde “Cortes” será el nombre de la variable asociada al vértice y Player el tipo de vértice, lo que hay dentro del corchete son variables, el apellido, el nombre, el

código de país, su nivel de go, en este caso 12 kyu, y su número de Gor.



Como se comprueba en el grafico el nivel de go se calcula con el número de Gor, que aumenta si se ganan partidas en torneo y disminuye si se pierden, por lo que pondremos el gor como un numero entero.

Una vez creado el vértice de jugador (player), creamos el de club:

```
CREATE (T140725A:Tournament { Class : 'C', Date: '2014-07-25', Description: 'VII Torneo de Go de Cadiz', Country: 'ES', City: 'Cadiz', Rounds: '4'})
```

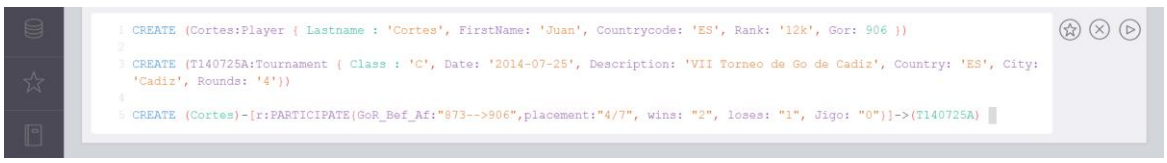
Como se puede comprobar hemos usado de nombre de la variable el código asociado al torneo que nos facilita la european go database y hemos añadido los campos que la misma almacena. La clase del torneo en este caso tipo C, la fecha, la descripción, el país, la ciudad y el número de rondas que tiene dicho torneo.

Luego simplemente para crear la relación:

```
CREATE (Cortes)-[r:PARTICIPATE{GoR_Bef_Af:"873-->906",placement:"4/7", wins: "2", loses: "1", Jigo: "0"}]->(T140725A)
```

Ponemos un nombre para la relación, en este caso “r” y podemos ponerle a la relación atributos en este caso, el número gor que tuvo al inicio y con el que acabo, el puesto que quedo en el torneo, las partidas ganadas, las perdidas, y los empates que en go se llaman “jigo” porque por sus normas es difícil que se dé y se le llama así.

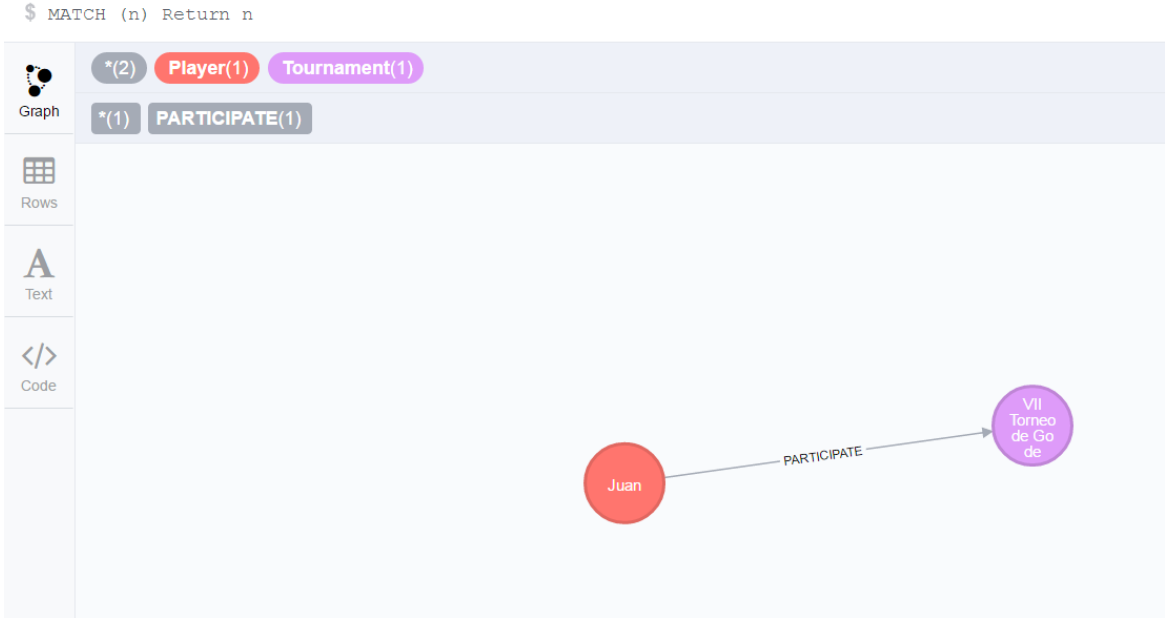
Si el script se ejecuta al mismo tiempo la relación se creará:



Para ver el grafo completo ponemos esto:

```
MATCH (n)
Return n
```

Que nos mostrará el grafo entero:



En la siguiente sección del entorno grafico nos pararemos a explicar las diferentes funcionalidades que tiene y podemos usar, además para la sección de una base de datos real veremos más código cypher.

Pero como habéis comprobado es muy fácil crear una relación, es cierto que si creásemos la relación en otro momento tendríamos que “buscar” los nodos con match tal que así:

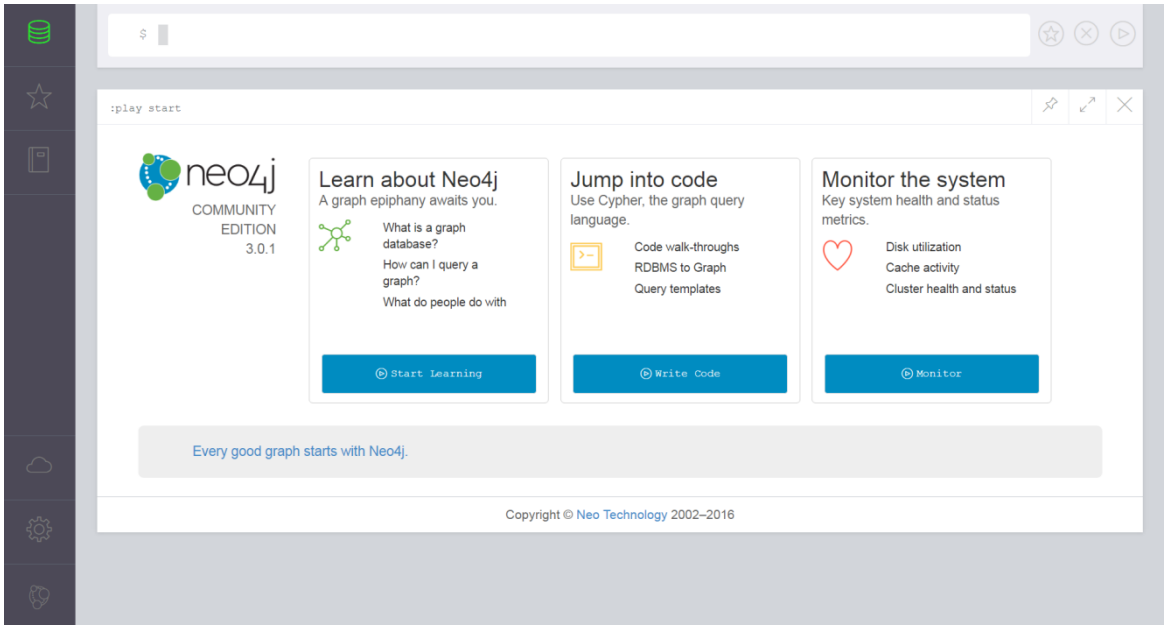
```
MATCH (Cortes:Player {Lastname:"Cortes"})

MATCH (T140725A:Tournament {City:"Cadiz"})

CREATE (Cortes)-[r:PARTICIPATE{GoR_Bef_Af:"1530-->1521",placement:"5/7", wins: "4", loses: "2", Jigo: "0"}]->(T140725A)
RETURN r
```

De todas formas en las siguientes secciones explicaremos eso más detenidamente.

Para finalizar explicar que cypher es sencillo y fácil de usar, además de muy intuitivo.

2.3. Entorno grafico

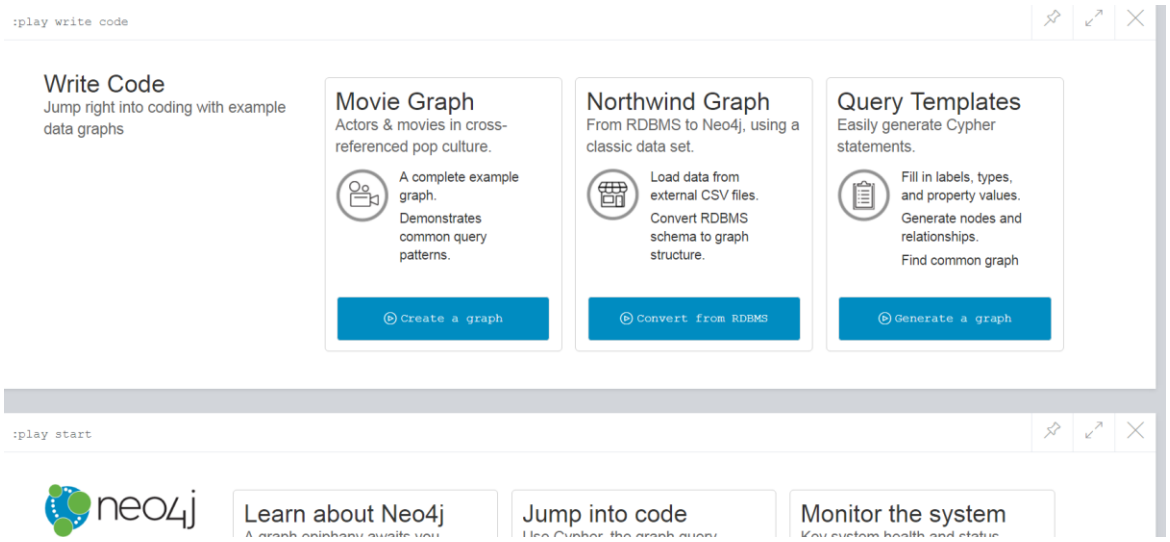
El entorno grafico que nos facilita neo4j es muy intuitivo y fácil de usar.



Cuando accedamos por primera vez nos saldrá lo que vemos en la imagen de arriba.

Si pinchamos en el primer enlace azul nos saldrá una guía básica en ingles explicando el tema de los vértices, las aristas, etc…

El segundo es bastante interesante porque nos carga esto:



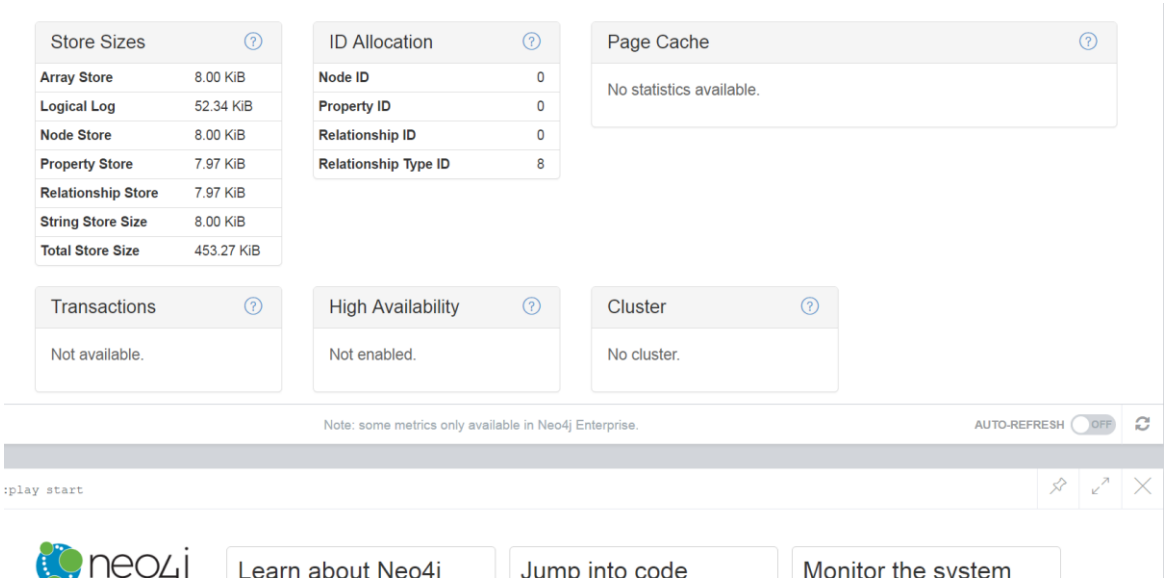
“Movie Graph” nos da el código para crear una base de datos entera de películas y actores. Es interesante para tener una de ejemplo fácil de usar.

En el segundo te explican como usar archivos externos de CSV y cargarlos para crear una base de datos de grafos o convertir un esquema RDBMS en una estructura de grafos.

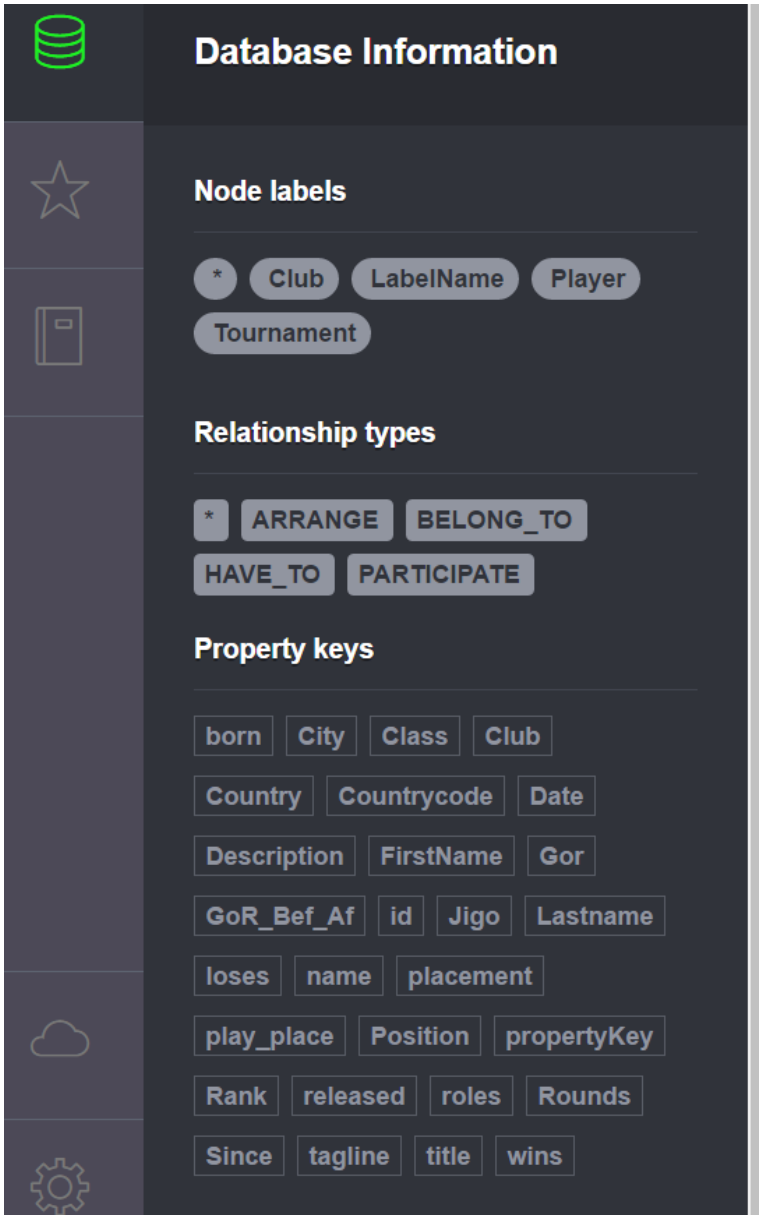
Y el más interesante el tercero que al pulsar te carga este formulario:

Donde se ve el código para crear los nodos que quisiéramos con su conexión es muy básico pero nos ayuda no solo a entender cómo funciona neo4j sino a crear ejemplos de manera muy sencilla.

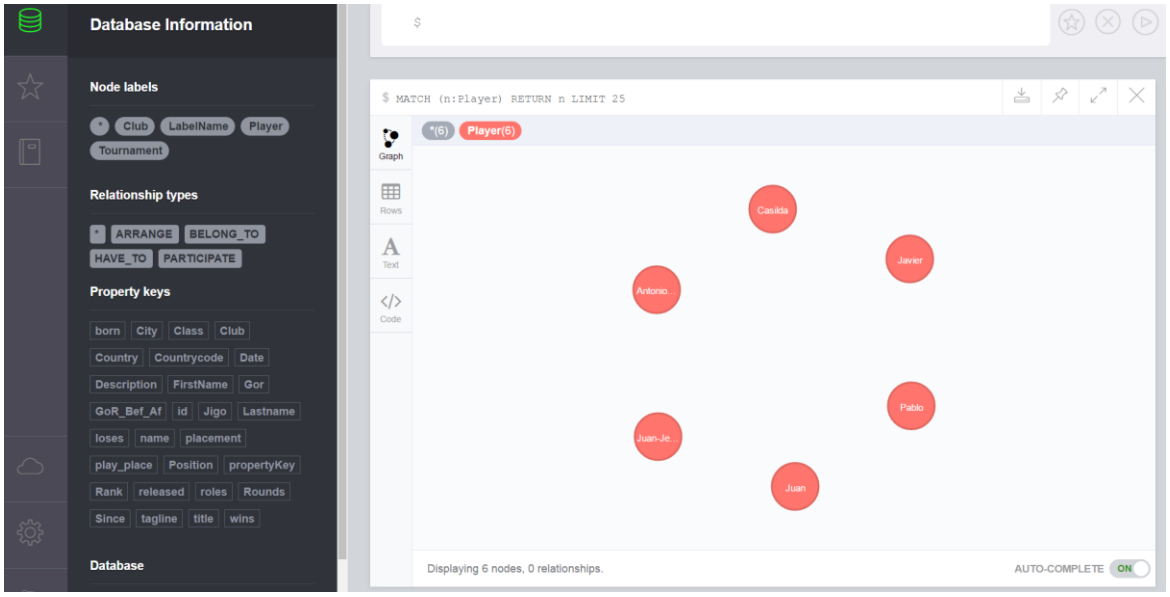
Volviendo al inicio Monitor the System simplemente nos da una serie de información que puede sernos útil:



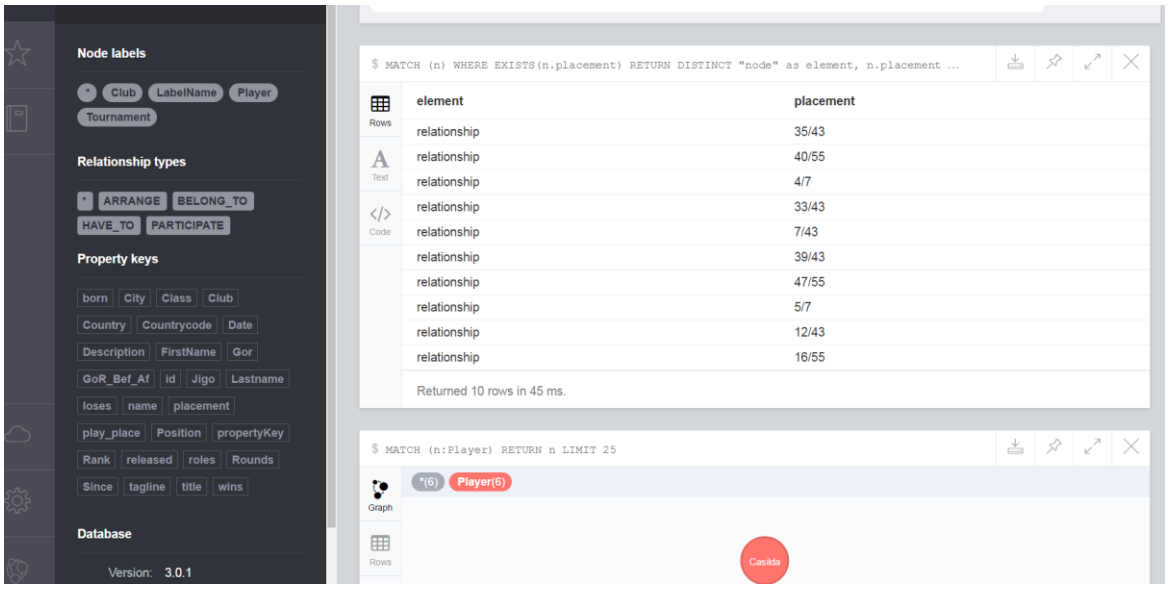
Ahora vamos a irnos a los iconos de la izquierda, el primero se llama “Database Information”



Que nos muestra los nodos, las relaciones y propiedades que hemos creado. Cuando pinchamos en uno de ellos por ejemplo en “players” nos cargará los vértices con ese nombre:

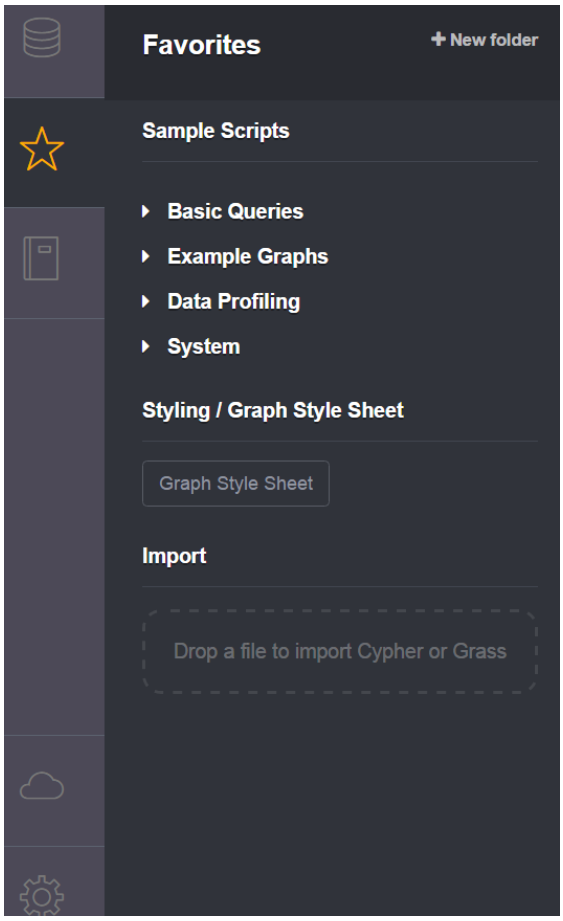


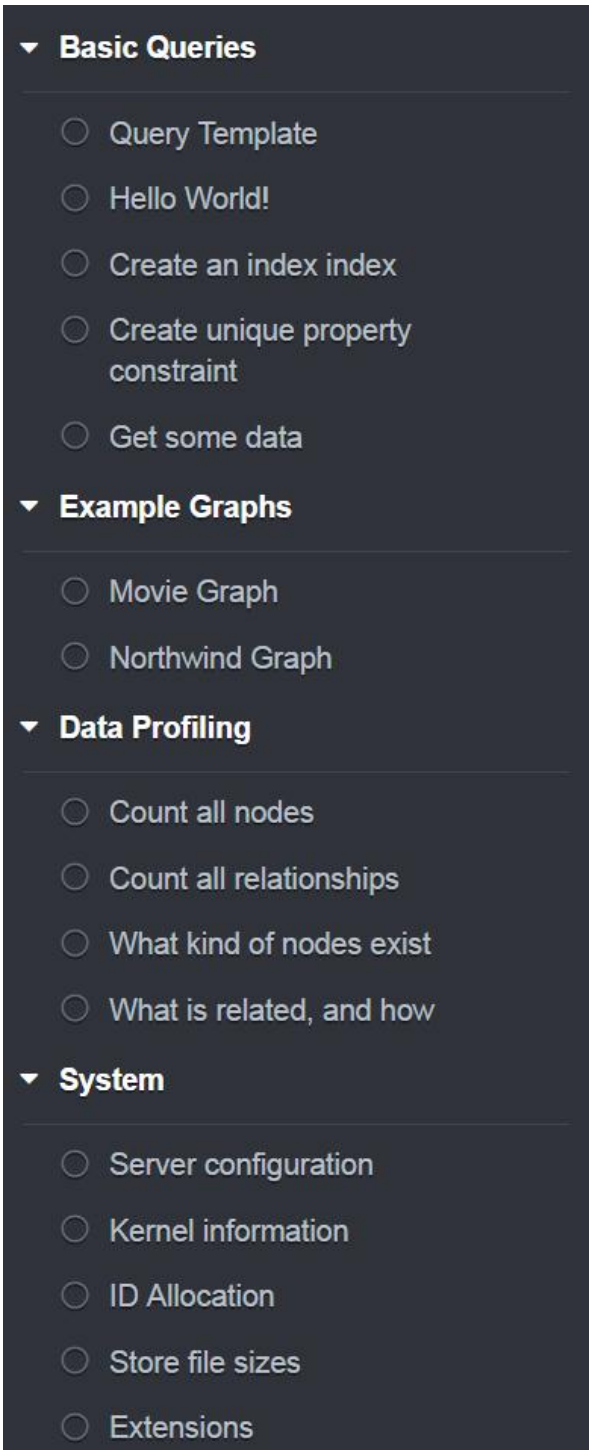
Sin embargo si pulsaramos en “placement” que es un atributo de la relación de player y tournament, que significa la posición en la que quedo en dicho torneo, nos muestra en una tabla todos los valores que hay con eso:



Esta interfaz grafica como podemos comprobar nos da muchas facilidades ahorrándonos tener que hacer un comando de código para algunas cosas.

El siguiente icono sería “favorites:



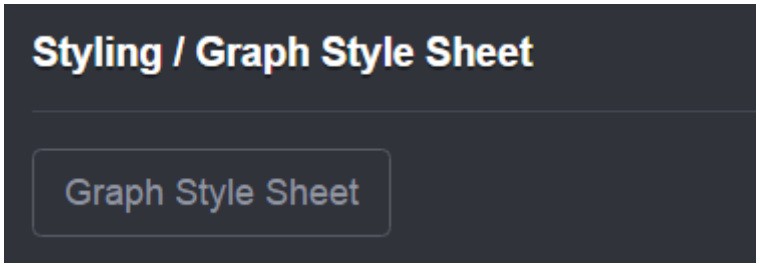


Como podemos comprobar hay cuatro puntos clave aquí, el primero nos crea cosas básicas, el segundo grafos generados, uno el de películas y otro desde un archivo externo.

El tercero nos da el número de nodos, el número de las relaciones que hay etc…

Y por ultimo el de System nos da información del sistema.

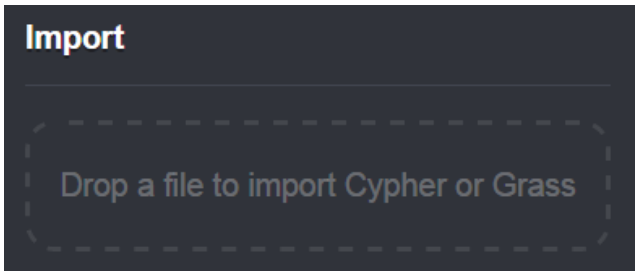
De aquí lo más interesante seria esto:



Que al pinchar nos permite cambiar los colores de los nodos, su tamaño, etc…, dando una facilidad de edición increíble a nuestra base de datos:

```
node {
  diameter: 50px;
  color: #A5ABB6;
  border-color: #9AA1AC;
  border-width: 2px;
  text-color-internal: #FFFFFF;
  font-size: 10px;
}

relationship {
  color: #A5ABB6;
  shaft-width: 1px;
  font-size: 8px;
  padding: 3px;
  text-color-external: #000000;
  text-color-internal: #FFFFFF;
  caption: '<type>';
}

node.Movie {
  color: #68BDF6;
  border-color: #5CA8DB;
  text-color-internal: #FFFFFF;
  caption: '{title}';
}

node.Person {
  color: #6DCE9E;
  border-color: #60B58B;
  text-color-internal: #FFFFFF;
```

Para ello debemos bajárnoslo editarlo y soltarlo en import:

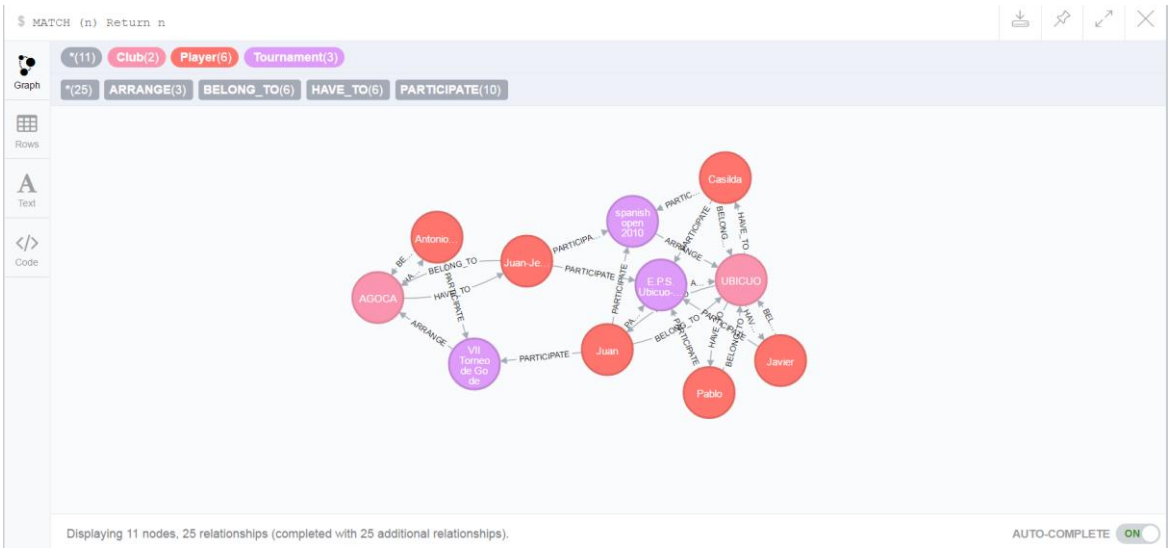


Los cuatro últimos iconos son menos interesantes, el de forma de libro es documentación en ingles bastante extensa. El de forma de nube para sincronizar con neo4j registrándonos, etc… el penúltimo es para editar cosas como el puerto donde carga la base de datos y por último un icono que nos da información de la versión de neo4j usada, de la empresa que lo creó etc…

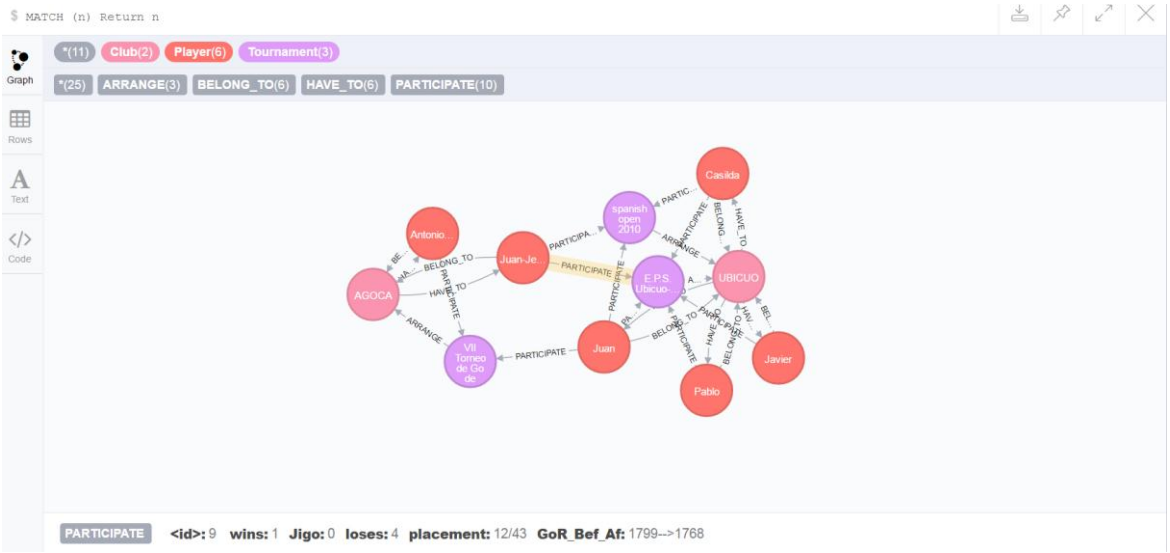
Ahora vamos a pararnos a explicar la parte donde se escribe el código si ponemos el siguiente código:

```
MATCH (n)
Return n
```

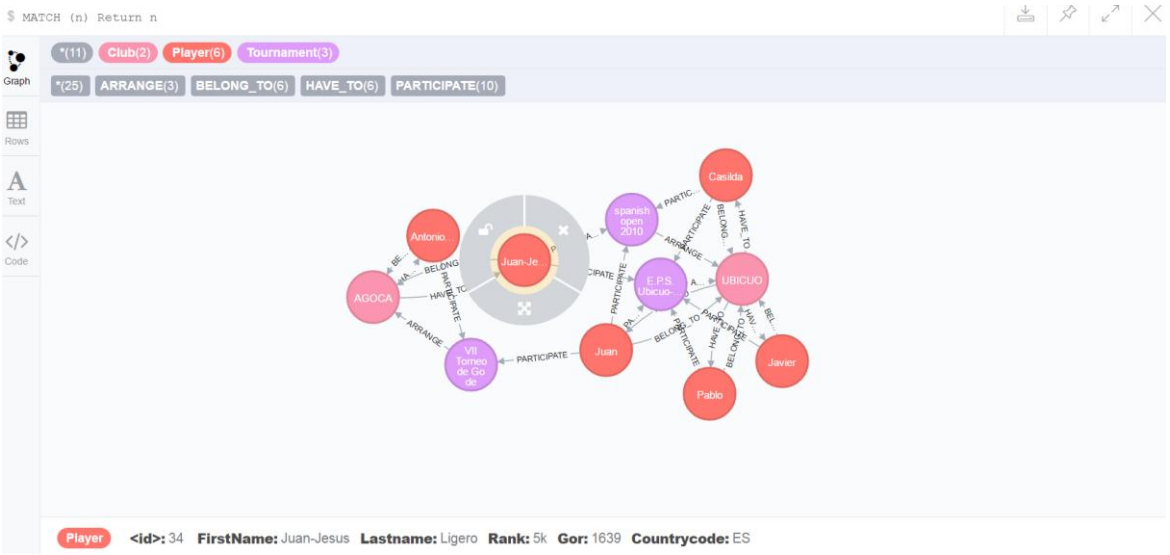
Y pinchamos en la estrella se nos guardará en favorites, si pinchamos en play (el icono de la derecha) nos cargara el script, en este caso nos cargará el grafo entero.



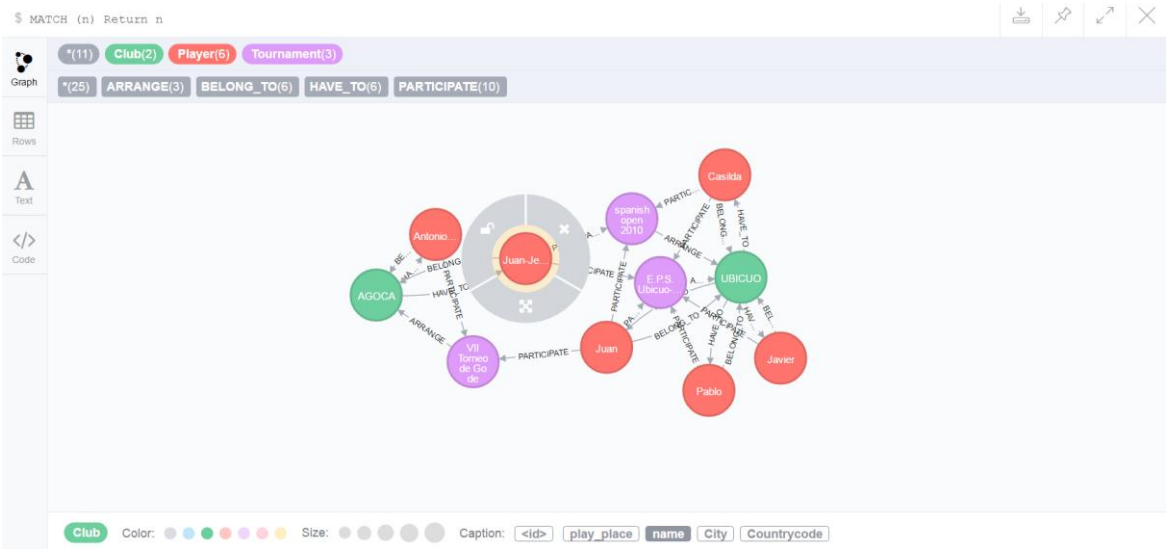
Si pinchamos en cualquier relación en este caso la del player “Juan Jesus” con el torneo de Ubicuo, podemos ver las partidas que gano, las que empato, las que perdió, etc…



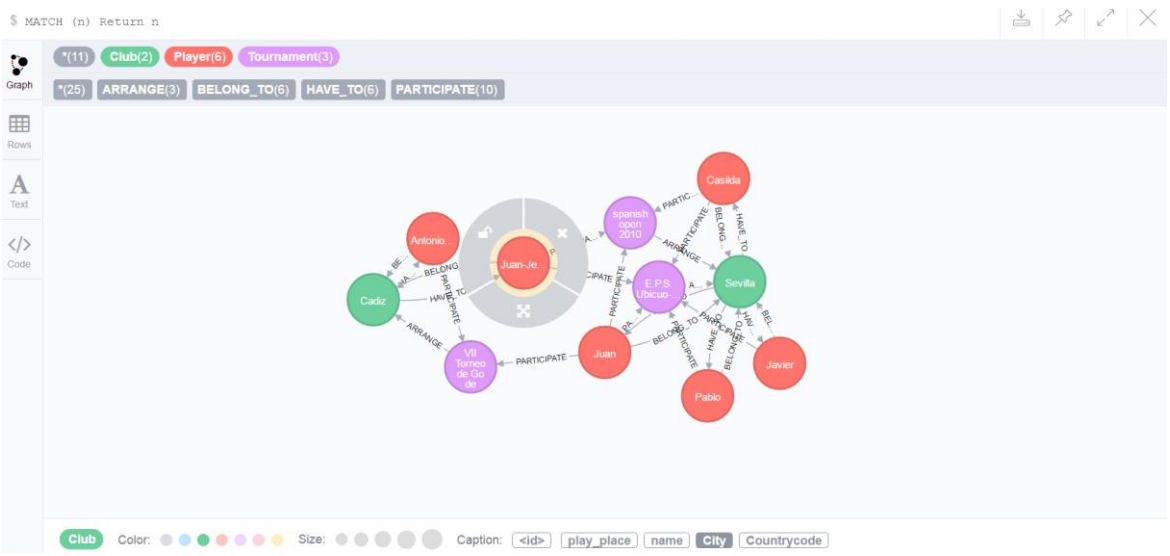
Además si pinchamos en un vértice podremos ver sus atributos igual que con las relaciones (aristas) y podremos bloquearlo para que no se mueva (en el entorno grafico), con el icono del candado, borrar de la imagen sus conexiones para una visión mejor con el icono de la derecha en forma de cruz y luego recuperarlas con el botón de abajo del nodo.



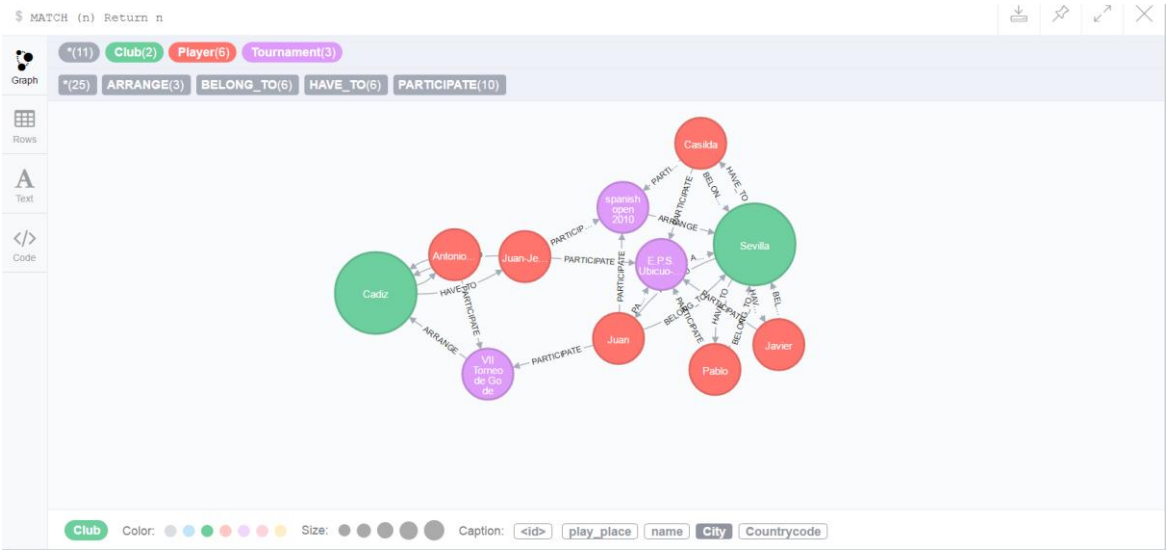
Además podemos cambiar el color de cualquier tipo, en este caso hemos seleccionado club arriba y seleccionado otro color ya que era muy similar al de los players.



Además el entorno grafico nos permite cambiar la representación del nodo en este caso vamos a cambiar el nombre de los clubs por su ciudad.

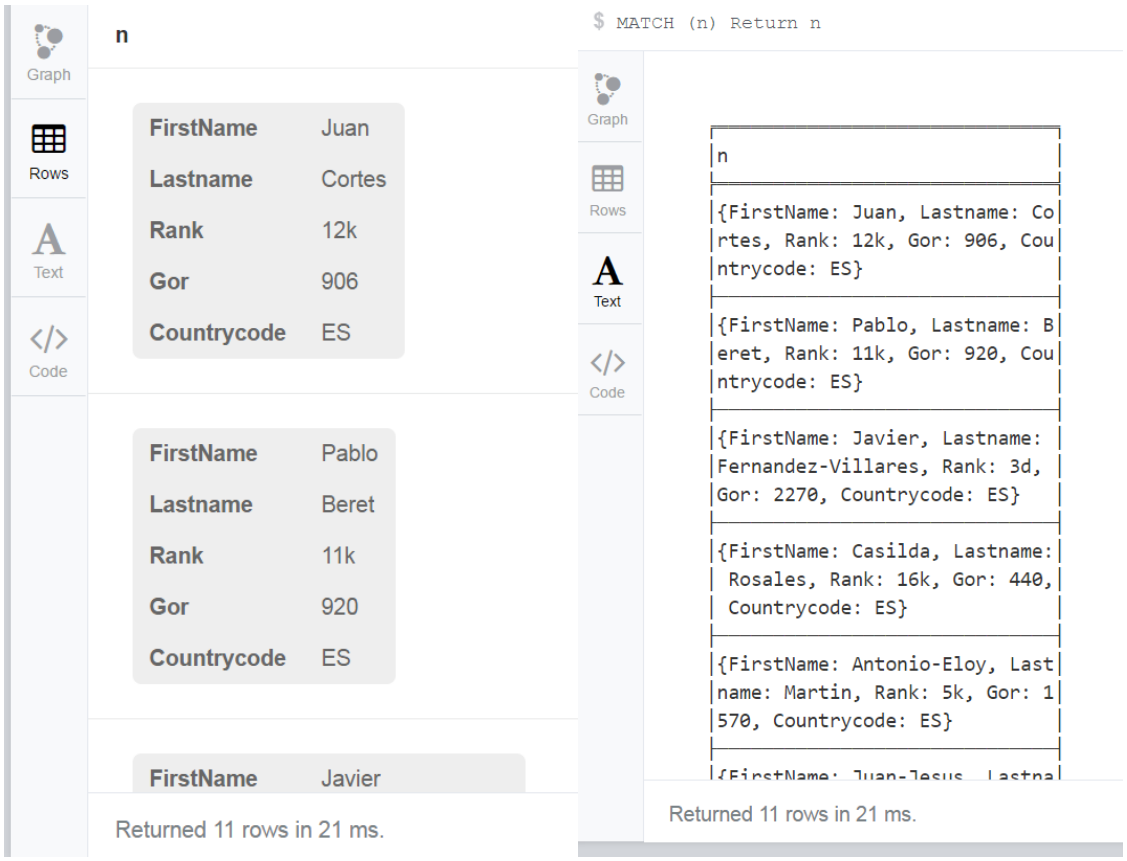


Y además podemos aumentar el tamaño del que queramos, por ejemplo el de club para que destaque del resto:



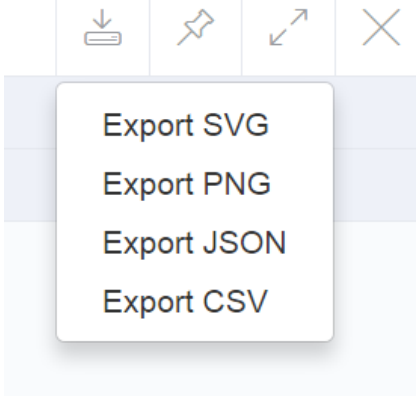
Una vez hecho esto, vamos a ver las diferentes opciones que tenemos a la izquierda, en este caso el “rows” y el “text” que nos muestran el primero el grafo en forma de filas y cuadrados y el text que de manera similar pero más “formato código”.

Para acabar con el “code” que nos lo pone directamente en código.



Para acabar con esta sección mostrar los siguientes botones que uno exporta en diferentes formatos el grafo, el siguiente pone una chincheta a esta ventana para que se quede arriba, las dos flechitas maximizan la venta y por último la X la cierra.

Podemos tener las ventanas que queramos no hay límites.



2.4. Trabajando con una base de datos real

En este trabajo se ha utilizado una base de datos real.

Hemos creado tres tipos “players”, “clubs” y “tournaments”, que equiparan a jugadores, los clubs a los que pertenecen y los torneos que jugaron.

Para darle cierta lógica las relaciones que se han creado entre ellos son la de un jugador pertenece a un club, un club tiene un jugador, un jugador participa en un torneo y por ultimo un club organiza un torneo.

Todo se ha puesto en inglés, basándonos en los datos de la european go database, usando datos reales se ha cogido una muestra significativa, lo suficientemente grande como para comprobar el funcionamiento de Neo4j y las diferentes posibilidades y funcionalidades que permite.

Aquí pondremos el script usado:

```
//--Jugadores--

//Ubicuos

CREATE (Cortes:Player { Lastname : 'Cortes', FirstName: 'Juan', Countrycode: 'ES', Rank: '12k', Gor: 906 })

CREATE (Beret:Player { Lastname : 'Beret', FirstName: 'Pablo', Countrycode: 'ES', Rank: '11k', Gor: 920 })

CREATE (Fernandez:Player { Lastname : 'Fernandez-Villares', FirstName: 'Javier', Countrycode: 'ES', Rank: '3d', Gor: 2270 })

CREATE (Rosales:Player { Lastname : 'Rosales', FirstName: 'Casilda', Countrycode: 'ES', Rank: '16k', Gor: 440 })


//Agocas:


CREATE (Martin:Player { Lastname : 'Martin', FirstName: 'Antonio-Eloy', Countrycode: 'ES', Rank: '5k', Gor: 1570})

CREATE (Ligero:Player { Lastname : 'Ligero', FirstName: 'Juan-Jesus', Countrycode: 'ES', Rank: '5k', Gor: 1639})


//--Clubs--

CREATE (GoCadiz:Club { name : 'AGOCA', Countrycode: 'ES', City: 'Cadiz', play_place: 'Casa de la juventud de Cadiz'})

CREATE (Ubicuo:Club { name : 'UBICUO', Countrycode: 'ES', City: 'Sevilla', play_place: 'Cafe de la prensa'})


//--Torneos--
```

CREATE (T140725A:Tournament { Class : 'C', Date: '2014-07-25', Description: 'VII Torneo de Go de Cadiz', Country: 'ES', City: 'Cadiz', Rounds: '4'})

CREATE (T101030K:Tournament { Class : 'A', Date: '2010-10-30', Description: 'spanish open 2010', Country: 'ES', City: 'Sevilla', Rounds: '6'})

CREATE (T110326F:Tournament { Class : 'A', Date: '2011-03-26', Description: 'E.P.S. Ubicuo-kiin Club Tournament', Country: 'ES', City: 'Sevilla', Rounds: '5'})

//--Relationship--

CREATE (Cortes)-[r:PARTICIPATE{GoR_Bef_Af:"873-->906",placement:"4/7", wins: "2", loses: "1", Jigo: "0"}]->(T140725A)

CREATE (Martin)-[r2:PARTICIPATE{GoR_Bef_Af:"1530-->1521",placement:"5/7", wins: "4", loses: "2", Jigo: "0"}]->(T140725A)

CREATE (Cortes)-[r3:PARTICIPATE{GoR_Bef_Af:"826-->810",placement:"40/55", wins: "3", loses: "3", Jigo: "0"}]->(T101030K)

CREATE (Ligero)-[r4:PARTICIPATE{GoR_Bef_Af:"1763-->1799",placement:"16/55", wins: "3", loses: "3", Jigo: "0"}]->(T101030K)

CREATE (Rosales)-[r5:PARTICIPATE{GoR_Bef_Af:"600-->500",placement:"47/55", wins: "2", loses: "4", Jigo: "0"}]->(T101030K)

CREATE (Beret)-[r6:PARTICIPATE{GoR_Bef_Af:"895-->979",placement:"33/43", wins: "3", loses: "2", Jigo: "0"}]->(T110326F)

CREATE (Cortes)-[r7:PARTICIPATE{GoR_Bef_Af:"810-->856",placement:"35/43", wins: "3", loses: "2", Jigo: "0"}]->(T110326F)

CREATE (Rosales)-[r8:PARTICIPATE{GoR_Bef_Af:"500-->400",placement:"39/43", wins: "1", loses: "4", Jigo: "0"}]->(T110326F)

CREATE (Fernandez)-[r9:PARTICIPATE{GoR_Bef_Af:"2291-->2270",placement:"7/43", wins: "2", loses: "3", Jigo: "0"}]->(T110326F)

CREATE (Ligero)-[r10:PARTICIPATE{GoR_Bef_Af:"1799-->1768",placement:"12/43", wins: "1", loses: "4", Jigo: "0"}]->(T110326F)

CREATE (Martin)-[r112:BELONG_TO{Position:"President"}]->(GoCadiz)

CREATE (Ligero)-[r122:BELONG_TO{Position:"Member"}]->(GoCadiz)

CREATE (Beret)-[r132:BELONG_TO{Position:"President"}]->(Ubicuo)

CREATE (Fernandez)-[r142:BELONG_TO{Position:"Member"}]->(Ubicuo)

CREATE (Rosales)-[r15:BELONG_TO{Position:"Member"}]->(Ubicuo)

CREATE (Cortes)-[r162:BELONG_TO{Position:"Member"}]->(Ubicuo)

CREATE (GoCadiz)-[r173:HAVE_TO{Since:"2008"}]->(Martin)

```
CREATE (GoCadiz)-[r183:HAVE_TO{Since:"2008"}]->(Ligero)

CREATE (Ubicuo)-[r193:HAVE_TO{Since:"2005"}]->(Beret)

CREATE (Ubicuo)-[r23:HAVE_TO{Since:"2009"}]->(Cortes)

CREATE (Ubicuo)-[r223:HAVE_TO{Since:"2006"}]->(Fernandez)

CREATE (Ubicuo)-[r213:HAVE_TO{Since:"2007"}]->(Rosales)


CREATE (Ubicuo)<-[r244:ARRANGE{}]-(T110326F)

CREATE (Ubicuo)<-[r254:ARRANGE{}]-(T101030K)

CREATE (GoCadiz)<-[r264:ARRANGE{}]-(T140725A)
```

Ahora lo explicaremos con detalle.

La primera parte:

```
//--Jugadores--

//Ubicuos

CREATE (Cortes:Player { Lastname : 'Cortes', FirstName: 'Juan', Countrycode: 'ES', Rank: '12k', Gor: 906 })

//Agocas:


CREATE (Martin:Player { Lastname : 'Martin', FirstName: 'Antonio-Eloy', Countrycode: 'ES', Rank: '5k', Gor: 1570})
```

Tiene unos comentarios (que es el texto que va detrás de "//") que dejan claro que son Jugadores de dos clubs distintos, que se puso así para que al crearlo me facilitará las cosas.

Como podemos comprobar, como variable se ha usado el apellido y los atributos son los mencionados en la sección 2.2, donde se habló brevemente de cypher.

La segunda parte:

```
//--Clubs--

CREATE (GoCadiz:Club { name : 'AGOCA', Countrycode: 'ES', City: 'Cadiz', play_place: 'Casa de la juventud de Cadiz'})


//--Torneos--

CREATE (T140725A:Tournament { Class : 'C', Date: '2014-07-25', Description: 'VII Torneo de Go de Cadiz', Country: 'ES', City: 'Cadiz', Rounds: '4'})
```

Se pone los clubs y torneos que hay, los atributos están basados en la european go

database. Como podemos comprobar, la variable que se ha usado en los torneos es el código que se le asigna en la european go database y en clubs se ha puesto un seudónimo del club. Los atributos están dentro de los corchetes.

Y por último la tercera:

```
//--Relationship--

CREATE (Cortes)-[r:PARTICIPATE{GoR_Bef_Af:"873-->906",placement:"4/7", wins: "2", loses: "1", Jigo: "0"}]->(T140725A)


CREATE (Martin)-[r2:PARTICIPATE{GoR_Bef_Af:"1530-->1521",placement:"5/7", wins: "4", loses: "2", Jigo: "0"}]->(T140725A)
```

Que trata el tema de las relaciones, como podemos comprobar se llama a los nodos por su variable y se les conecta mediante la relación y los atributos de la misma, a la relación se la da variables “rx”, siendo x un número para agilizar su creación.

En el caso hipotético de que quisiéramos crear la relación una vez creados los nodos, es decir en otro script habría que buscar los nodos de esta forma:

```
MATCH (Martin:Player {Lastname:"Martin"})

MATCH (GoCadiz:Club {name : "AGOCA"})

CREATE (GoCadiz)-[r:HAVE_TO{Since:"2008"}]->(Martin)
RETURN r
```

El return r, nos devolvería la relación creada, ya que las variables que ponemos en el script para llamar a las relaciones o a los nodos una vez ejecutada no se guardan en ningún sitio.

Por lo que si crearamos las relaciones una a una en script distintos, podríamos usar siempre la misma variable, pero en el caso de que crearamos un script único entero, habría que hacer como viene arriba, variables diferentes.

Comentar que si queremos borrar todo lo creado debemos usar el delete con detach, que borra todo lo relacionado con todo:

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

Por ejemplo si quisiéramos borrar un nodo concreto y todas sus relaciones sería así:

```
MATCH (n)
WHERE id(n)= 9
DETACH DELETE n
```

2.5. Syntax

Ahora pondremos la sintasis de Cypher adquirida en la página oficial de neo4j, para la comodidad del lector hispanohablante de este documento se ha traducido al castellano las explicaciones de cada una.

Operadores	
Matematicos	`+`, `-`, `*`, `/`, `%`, `^`
Comparación	`=`, `<>`, `<`, `>`, `<=`, `>=`
Boolean	`AND`, `OR`, `XOR`, `NOT`
String	`+`
Collection	`+`, `IN`, `[x]`, `[x .. y]`
Expresiones regulares	`=~`

Patterns
`(n)-->(m)` Una relacion de n a m.
`(n:Person)` Encuentra el nodo con el campo Person.
`(n:Person:Swedish)` Encuentra los nodos que tengan ambos campos Person y Swedish.
`(n:Person {name: {value}})` Encuentra los nodos con las propiedad declarada
`(n:Person)-->(m)` Nodo n con el campo Persona tiene una relación con m
`(n)--(m)` Relación de cualquier dirección entre n y m.
`(m)<-[:KNOWS]-(n)` Una relación de n a m del tipo `KNOWS`.
`(n)-[:KNOWS|LOVES]->(m)` Una relación de n a m del tipo `KNOWS` o `LOVES`.
`(n)-[r]->(m)` Enlaza e identifica a una relación.
`(n)-[*1..5]->(m)` Longitud del camino entre n y m.
`(n)-[*]->(m)` Cualquier camino. Ver los diferentes caminos.
`(n)-[:KNOWS]->(m {property: {value}})`

Patterns

Encuentra o modifica las propiedades
con `MATCH`, `CREATE`, `CREATE UNIQUE` o `MERGE` clases.

```
shortestPath((n1:Person)-[*..6]-(n2:Person))
```
Encuentra el camino más corto entre dos vértices.

```
allShortestPaths((n1:Person)-->(n2:Person))
```
Encuentra todos los caminos más cortos entre dos vértices.

Functions

```
coalesce(n.property, {defaultValue})
```
La primera expresión no nula-

```
timestamp()
```
Milisegundos desde la media noche, Enero 1, 1970 UTC.

```
id(node_or_relationship)
```
El id interno de la relación de un nodo.

```
toInt({expr})
```
Convertir la expression metida en un integer si es possible, en otro caso devolver `NULL`.

```
toFloat({expr})
```
Convertir la expression metida en un float si es possible, en otro caso devolver `NULL`.

String Functions

```
toString({expression})
```
Representación en cadena de la expresión.

```
replace({original}, {search}, {replacement})
```
Cambia lo encontrado por `search` con `replacement.` Todos los argumentos deben ser expresiones.

```
substring({original}, {begin}, {sub_length})
```
Me da una parte de una cadena. `Sub_length` es un argumento opcional.

```
left({original}, {sub_length}),
  right({original}, {sub_length})
```
La primera parte del string. La última parte del string.

```
trim({original}), ltrim({original}),
  rtrim({original})
```

Recorta todos los espacioes en blanco o solo los de la izquierda o solo los de la derecha.

```
upper({original}), lower({original})
```
Mayusculas y minusculas.

```
split({original}, {delimiter})
```
Dividir una cadena en una “collection” de cadena

Read Query Structure

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

MATCH

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name="Alice"
```

Para buscar puedes usar propiedades o campos.

```
MATCH (n)-->(m)
```

Vale cualquier cosa para buscar con `MATCH`.

```
MATCH (n {name:'Alice'})-->(m)
```

Buscar con propiedad del nudo

```
MATCH p = (n)-->(m)
```

Asignar a p

```
OPTIONAL MATCH (n)-[r]->(m)
```

Buscar cuando hay cosas desaparecidas para igualarlas a null.

WHERE

```
WHERE n.property <> {value}
```

Usar un predicate para filtrar. Tener en cuenta que `WHERE` es siempre una parte de las clausulas `MATCH`, `WITH, START O OPTIONAL MATCH`. Poner después una clausula diferente en la query siempre alterara lo que hace.

UNION

```
MATCH (a)-[:KNOWS]->(b)
RETURN b.name
UNION
MATCH (a)-[:LOVES]->(b)
RETURN b.name
```

Devuelve una union distinta de todas las querys resultante. La columna resultante tendrá el tipo y nombres que tenga la búsqueda.

```
MATCH (a)-[:KNOWS]->(b)
RETURN b.name
UNION ALL
MATCH (a)-[:LOVES]->(b)
RETURN b.name
```

Devuelve una union de todas las querys resultantes, incluyendo las duplicadas.

Labels

```
CREATE (n:Person {name:{value}})
```

Crea un nodo de tipo `Person` con la propiedad `name`.

UNION

```
MERGE (n:Person {name:{value}})
```
Encuentra o crea un nodo único de tipo `Person` con la propiedad `name`.

```
SET n:Spouse:Parent:Employee
```
Añade un tipo al nodo.

```
MATCH (n:Person)
```
Encuentra nodos de tipo `Person` .

```
MATCH (n:Person)
WHERE n.name = {value}
```
Encuentra nodos tipo `Person` con la propiedad `name` igual al value

```
WHERE (n:Person)
```
Comprueba si existe algun nodo de tipo `Person`

```
labels(n)
```
Tipo del nodo.

```
REMOVE n:Person
```
Borra los nodos tipo `Person`.

Relationship Functions

```
type(a_relationship)
```
Una cadena representación de los tipos de relaciones.

```
startNode(a_relationship)
```
Nodo inicial de una relación.

```
endNode(a_relationship)
```
Nodo final de una relación

```
id(a_relationship)
```
Id interno de una relación.

Predicates

```
n.property <> {value}
```
Usar comparadores de valores.

```
has(n.property)
```
Usa funciones

```
n.number >= 1 AND n.number <= 10
```
Usa operadores boolean para combinar predicados.

```
n:Person
```
Comprueba el tipo del nodo.

```
identifier IS NULL
```
Comprueba si algo es `NULL`.

Relationship Functions

```
NOT has(n.property) OR n.property = {value}
```
Si alguna de las dos propiedades hace que no se cumpla el predicado es `TRUE`.

```
n.property = {value}
```
No devuelve `NULL` cuando no hay nada que de cierta la igualdad.

```
n.property =~ "Tob.*"
```
Expression regular.

```
(n)-[:KNOWS]->(m)
```
Asegúrese que las variables n y m han sido buscadas antes con match.

```
NOT (n)-[:KNOWS]->(m)
```
Excluye la búsqueda `(n)-[:KNOWS]->(m)` del resultado.

```
n.property IN [{value1}, {value2}]
```
Comprueba si el elemento existe en la collection.

Collection Predicates

```
all(x IN coll WHERE has(x.property))
```
Devuelve `TRUE` si el predicado es `TRUE` para todos los elementos de la collection.

```
any(x IN coll WHERE has(x.property))
```
Devuelve `TRUE` si el predicado es `TRUE` para al menos un elemento de la collection.

```
none(x IN coll WHERE has(x.property))
```
Devuelve `TRUE` si el predicado es `FALSE` para todos los elementos de la collection.

```
single(x IN coll WHERE has(x.property))
```
Devuelve `TRUE` si el predicado es `TRUE` para exactamente un elemento de la collection.

Aggregation

```
count(*)
```
El número de elementos de una busqueda.

```
count(identifier)
```
El número de nombres no nulos que da ese valor.

```
count(DISTINCT identifier)
```
Todas las funciones agregadas toman el valor `DISTINCT` que modifica los valores duplicados.

```
collect(n.property)
```
Collection de los valores, ignorando los `NULL`.

Collection Predicates

```
sum(n.property)
```
Suma de los valores numéricos. Otras funciones similares son: `avg`, `min`, `max`.

```
percentileDisc(n.property, {percentile})
```
Discrete percentile. Continuous percentile is`percentileCont`. The `percentile` argument is from `0.0` to `1.0`.
Percentile discreta. Continuamente el percentile es `percentileCont.` El `percentile` tiene un valor de `0.0` a `1.0`.

```
stdev(n.property)
```
Desviación estándar para una parte de la población. Para toda la población usamos `stdevp`.

CREATE UNIQUE

```
CREATE UNIQUE
    (n)-[:KNOWS]->(m {property: {value}})
```
Busca el camino y lo crea sino existe. El camino puede no incluir algunas partes opciones.

RETURN

```
RETURN *
```
Devuelve el valor de todos los identificadores.

```
RETURN n AS columnName
```
Usa un alias para el resultado del nombre de la columna.

```
RETURN DISTINCT n
```
Devuelve un dato único.

```
ORDER BY n.property
```
Ordena el resultado.

```
ORDER BY n.property DESC
```
Ordena el resultado en orden descendente.

```
SKIP {skip_number}
```
Se salta algunos resultados.

```
LIMIT {limit_number}
```
Limita el número de resultados.

```
SKIP {skip_number} LIMIT {limit_number}
```
Se salta algunos resultados del límite de arriba del número de resultados.

```
RETURN count(*)
```
El numero de búsquedas encontradas. Se pueden agregar mas.

WITH

```
MATCH (user)-[:FRIEND]-(friend)
WHERE user.name = {name}
WITH user, count(friend) AS friends
WHERE friends > 10
RETURN user
```

`WITH` es similar a `RETURN`. Esta separado como una query aparte explícitamente, porque tú puedes declarar algunos identificadores para llevar en la próxima vez.

```
MATCH (user)-[:FRIEND]-(friend)
WITH user, count(friend) AS friends
ORDER BY friends DESC
SKIP 1 LIMIT 3
RETURN user
```

Tu puedes usar `ORDER BY`, `SKIP`, `LIMIT` con `WITH`.

Collections

```
['a','b','c'] AS coll
```

Las collections de literales se declaran entre corchetes

```
length({coll}) AS len, {coll}[0] AS value
```

Las collections pueden ser pasadas como parámetros.

```
range({first_num},{last_num},{step}) AS coll
```

Range crea una collection de números (`step` es opcional), otras funciones que devuelven collections son: `labels`, `nodes`, `relationships`, `rels`, `filter` y `extract`.

```
MATCH (a)-[r:KNOWS*]->()
RETURN r AS rels
```

Los identificadores de relaciones son una variable del tamaño que contiene las collection de relaciones.

```
RETURN matchedNode.coll[0] AS value,
       length(matchedNode.coll) AS len
```

Properties can be arrays/collections of strings, numbers or booleans.

Las propiedades pueden ser arrays/collections de strings, numeros o booleans.

```
coll[{idx}] AS value,
coll[{start_idx}..{end_idx}] AS slice
```

A los elementos de la collection se puede acceder con subíndices `idx` en corchetes. Los índices no validos devuelven `NULL`. Se pueden tomar también porciones de la collection tomando un intervalo desde `start_idx` a `end_idx` donde cada uno puede omitir o negar. Fuera del rango los elementos serán ignorados.

WITH

```
UNWIND {names} AS name
MATCH (n {name:name})
RETURN avg(n.age)
```

Con `UNWIND`, puedes transformer cualquier collection en datos individuales. El ejemplo busca todos los nombres de una lista de nombres.

Maps

```
{name:'Alice', age:38,
 address:{city:'London', residential:true}}
```

Maps de literales son declarados en llaves igual que las propiedades de los mapas. Soporta mapas anidados y collections.

```
MERGE (p:Person {name: {map}.name})
ON CREATE SET p={map}
```

Los maps pueden ser pasados con parámetros y usar los maps por su key.

```
MATCH (matchedNode:Person)
RETURN matchedNode
```

Los nodos y sus relaciones devuelven un maps de su data.

```
map.name, map.age, map.children[0]
```

Las entradas de map pueden ser accedidas por su key. Si su key es invalida devolverá un error.

Path Functions

```
length(path)
```

El tamaño de un path.

```
nodes(path)
```

Los nodos de un path en una collection.

```
relationships(path)
```

La relaciones de un path en una collection.

```
MATCH path=(n)-->(m)
RETURN extract(x IN nodes(path) | x.prop)
```

Asignar un path y un proceso en el nodo.

```
MATCH path = (begin) -[*]-> (end)
FOREACH
  (n IN rels(path) | SET n.marked = TRUE)
```

Ejecuta una operacion de modificación por cada relación del path.

Collection Functions

```
length({coll})
```

Tamaño de una collection.

Collection Functions

```
head({coll}), last({coll}), tail({coll})
```
`head` devuelve el primero, `last` devuelve el ultimo de la collection. `tail` devuelve el residuo de la collection. Si tuviera que devolver null, devuelve una collection vacía.

```
[x IN coll WHERE x.prop <> {value} | x.prop]
```
Combinación de filtro y el extracto en una notación concisa.

```
extract(x IN coll | x.prop)
```
Una collection dell valor de la expresión para cada elemento de la collection original.

```
filter(x IN coll WHERE x.prop <> {value})
```
Una colección filtrada de los elementos en los que el predicado está `TRUE`.

```
reduce(s = "", x IN coll | s + x.prop)
```
Evaluar la expresión para cada elemento de la colección, se acumulan los resultados.

```
FOREACH (value IN coll |
 CREATE (:Person {name:value}))
```
Ejecutar una operación de mutación para cada elemento de una collection.

CASE

```
CASE n.eyes
 WHEN 'blue' THEN 1
 WHEN 'brown' THEN 2
 ELSE 3
END
```
Devuelve el valor del `THEN` cuando la busqueda es cierta para el valor de `WHEN`. El valor del `ELSE` es opcional y puede ser sustituido por `NULL` si no esta.

```
CASE
 WHEN n.eyes = 'blue' THEN 1
 WHEN n.age < 40 THEN 2
 ELSE 3
END
```
Devuelve el valor de `THEN` cuando el primer valor del predicado de `WHEN` es `TRUE`. Los predicates se evalúan en orden.

START

```
START n=node(*)
```
Empezar desde todos los nodos.

```
START n=node({ids})
```
Empezar desde un nodo o más nodos especificando su id.

```
START n=node({id1}), m=node({id2})
```
Empezar desde varios nodos.

START

```
START n=node:nodeIndexName(key={value})
```

Query del índice con una consulta exacta. Utilice `node_auto_index` para el índice automático.

Write-Only Query Structure

```
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Read-Write Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

CREATE

```
CREATE (n {name: {value}})
```

Crea un nodo con la propiedad dada.

```
CREATE (n {map})
```

Crea un nodo con las propiedades dadas por el map.

```
CREATE (n {collectionOfMaps})
```

Crea un nodo con las propiedades dadas por la collectionOfMaps

```
CREATE (n)-[r:KNOWS]->(m)
```

Crea una relación con el tipo y la dirección dada; enlaza un identificador a la misma.

```
CREATE (n)-[:LOVES {since: {value}}]->(m)
```

Crea una relación con el tipo, la dirección, y las propiedades dadas.

MERGE

```
MERGE (n:Person {name: {value}})
ON CREATE SET n.created=timestamp()
ON MATCH SET
    n.counter= coalesce(n.counter, 0) + 1,
    n.accessTime = timestamp()
```

Match pattern or create it if it does not exist. Use `ON CREATE` and `ON MATCH` for conditional updates.

Busca el patrón o lo crea sino existe. Usa `ON CREATE` y `ON MATCH` para actualizaciones condicionadas.

```
MATCH (a:Person {name: {value1}}),
      (b:Person {name: {value2}})
MERGE (a)-[r:LOVES]->(b)
```

`MERGE` encuentra o crea una relación entre nodos.

MERGE

```
MATCH (a:Person {name: {value1}})
MERGE
  (a)-[r:KNOWS]->(b:Person {name: {value3}})
```

`MERGE` encuentra o crea subgrafos conectados al nodo.

SET

```
SET n.property = {value}, n.property2 = {value2}
```

Actualiza o crea una propiedad.

```
SET n = {map}
```

Modifica toda las propiedades. Esto eliminará cualquier propiedad existente.

```
SET n += {map}
```

Añade o actualiza propiedades, mientras mantiene las existentes.

```
SET n:Person
```

Añade un tipo `Person` al nodo.

DELETE

```
DELETE n, r
```

Borra un nodo y su relación.

REMOVE

```
REMOVE n:Person
```

Elimina un tipo `Person` de `n`.

```
REMOVE n.property
```

Elimina una propiedad.

INDEX

```
CREATE INDEX ON :Person(name)
```

Crea un indice del tipo `Person` y una propiedad `name`.

```
MATCH (n:Person) WHERE n.name = {value}
```

Un índice puede ser utilizado de forma automática para la comparación de igualdad. Tenga en cuenta que, por ejemplo, inferior (n.name) = {valor} no va a usar un índice.

```
MATCH (n:Person) WHERE n.name IN [{value}]
```

Un índice puede ser utilizado de forma automática por IN de las comprobaciones de collection

```
MATCH (n:Person)
USING INDEX n:Person(name)
WHERE n.name = {value}
```

El uso de índices se puede hacer cumplir, cuando se utiliza un índice de Cypher subóptima o más de un índice debe ser utilizado.

```
DROP INDEX ON :Person(name)
```

Quitar el índice en la etiqueta `Person` y de la propiedad `name`.

CONSTRAINT

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

Crear una restricción única en el tipo `Person` y en la propiedad `name`. Si cualquier otro nodo con esa etiqueta se actualiza o se crea con un nombre que ya existe, la operación de escritura fallará. Para esta limitación se creará un índice de acompañamiento.

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

Quita la restricción única y el índice sobre el tipo Person y la propiedad name

Mathematical Functions

```
abs({expr})
```

El valor absoluto

```
rand()
```

Un valor aleatorio. Devuelve un nuevo valor para cada llamada. También son útiles para la selección de subconjunto o ordenación aleatoria.

```
round({expr})
```

Redondea al número entero más cercano, teniendo en cuenta el `ceil` y el `floor` es decir no más alto que `ceil` y no más pequeño que `floor`

```
sqrt({expr})
```

La raiz cuadrada.

```
sign({expr})
```

`0` si es cero, `-1` si es negativo, `1` si es positivo.

```
sin({expr})
```

Funcion trigonométrica como: `cos`, `tan`, `cot`, `asin`, `acos`, `atan`, `atan2`, `haversin`.

```
degrees({expr}), radians({expr}), pi()
```

Convierte radianes a grados, usa `radians` cuando quieras hacerlo al revez. Pi es π.

```
log10({expr}), log({expr}), exp({expr}), e()
```

Logaritmo en base 10, logaritmo normal, exponencial, y el valor de `e`.

3. Otras base de datos orientadas a grafos

En esta sección expondremos una lista de otras bases de datos orientadas a grafos:

Listado de bases de datos orientadas a grafo

- AllegroGraph: escalable y de alto rendimiento.
- Bigdata: RDF/base de datos orientada a grafo.
- CloudGraph:. NET usa tanto los grafos como clave/valor para almacenar los datos.
- Cytoscape: bioinformática.
- DEX (DEX/Sparksee): de alto rendimiento, permite escalar billones de objetos. Comercializada por Sparsity Technologies.
- Filament.
- GraphBase.
- Graphd, backend de Freebase.
- Horton.
- HyperGraphDB: base de datos opensource basada en la idea de hipergrafo.
- InfiniteGraph.
- InfoGrid9 open source.
- OrientDB: base de datos orientada a grafos y documental.
- OQGRAPH.
- Sones GraphDB
- VertexDB.
- Virtuoso Universal Server.
- R2DF.
- ArangoDB: Base de datos orientada a grafos, llave-valor y documental.

4. Conclusiones

Neo4j como hemos podido observar, tiene un entorno gráfico, fácil y sencillo, que facilita mucho a nivel de usuario. Tiene una sintaxis fácil de entender, ya que su lenguaje Cypher es muy parecido a sql y eso ayuda, es como un sql para grafos. La navegación de los datos por el sistema de grafos, lo hace fácil e intuitivo. El único problema que sabemos es que no es compatible con Sharding. ¿Qué es sharding?, un sistema para escalar los datos. Eso complica algo las cosas, pero por lo demás neo4j es una excelente herramienta. Muy recomendable.